Computational Statistics Summary

Maximilian Pfundstein

2019-03-12

Contents

1 Handling Computational Errors **2**

2 Difference Quotient **2**

3 Variance Estimators **2**

4 Optimization **3**
4.1 `optimize()` . . . 3
4.2 `optim()` . . . 3

5 Sampling Based on Size **4**

6 Inverse CDF Method **5**

7 Acceptance / Rejection Method **7**

8 Metropolis-Hastings Algorithm **9**

9 Gelman-Rubin Factor **11**

10 Monte Carlo Integration **13**

11 Gibbs Sampling **14**

12 Hypothesis Testing **15**
12.1 Defining a Test-Statistics . . . 15
12.2 Non-Parametric Bootstrap . . . 15
12.3 Density of non-parametric T-Values . . . 16
12.4 Defining the Hypothesis Test (Permutation Test) . . . 16
12.5 Crude Estimate of the Power . . . 17

13 Boostrap with Bias-Correction **18**

14 Variance Estmation using Jackknife **19**

15 Confidence Intervals with Respect to Length and Location **19**

16 Genetic Algorithms **20**

17 Expectation-Maximization Algorithm (EM) **23**
17.1 Expectation-Step (E-Step) . . . 23
17.2 Maximazation-Step (M-Step) . . . 24
17.3 EM-Implementation . . . 24

18 Miscellaneous Plots **25**
18.1 Histrogram . . . 25
18.2 Histrogram with Mean . . . 26

18.3 Simple X/Y Plot 26
18.4 Variance Plot 26
18.5 Scatterplot With Geom Smoother 26

19 Useful Code Snippets **27**
19.1 RMarkdown Setup 27
19.2 Knitr options 27
19.3 Including Source Code 27

1 Handling Computational Errors

```
x1 = 1/3
x2 = 1/4

if (all.equal(x1-x2, 1/12)) {
  print("Substraction is correct.")
} else {
  print("Substraction is wrong.")
}
```

2 Difference Quotient

$$f'(x) = \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

```
f_prime = function(x, epsilon = 10^(-5))  {
  return( (f(x + epsilon) - f(x)) / epsilon)
}
```

3 Variance Estimators

Krzysztof:

$$\text{Var}(\vec{x}) = \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2\right)$$

```
myvar = function(x) return(1/(length(x)-1) * (sum(x^2) - (sum(x)^2)/length(x)))
```

My:

$$s = \frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}$$

```
custom_variance = function(x) {
  diff_mean = x - mean(x)
  return(sum(diff_mean^2 / (length(x) - 1)))
}
```

4 Optimization

4.1 optimize()

```
myMSE = function(lambda, pars) {
  model = loess(Y ~ X, data=pars, enp.target = lambda)
  prediction = predict(model, newdata = pars$Xtest)
  mse = sum((prediction - pars$Ytest)^2)/length(pars$Ytest)
  return(mse)
}

# parameters for the myMSE function ----------------------
pars = list(X = train$Day, Y = train$LMR, Xtest = test$Day, Ytest = test$LMR)
lambdas = seq(from = 0.1, to = 40, by = 0.1)
# applying the myMSE function to all lambdas ------------
mses = sapply(X = lambdas, FUN = myMSE, pars = pars)

o = optimize(myMSE, tol = 0.01, interval = c(0.1, 40), pars = pars)
o$minimum
o$objective
```

Plotting a function with a minimum:

```
lambdas[which.min(mses)]
length(lambdas)

df = data.frame(lambdas, mses)

ggplot(df) +
  geom_line(aes(x = lambdas, y = mses), color = "#C70039") +
  geom_point(aes(x = seq(0.1, 40, by = 0.1)[which.min(mses)],
  y = mses[which.min(mses)], color = "min MSE"),
  colour = "blue") +
  labs(title = "Lambdas VS MSEs", y = "MSE", x = "Lambda") +
  theme_minimal()
```

4.2 optim()

General usage:

```
optim(35, myMSE, method = "BFGS", pars = pars, control = list(fnscale = 1))
```

For optimizing likelihood:

```
# c(mu, sigma)
neg_llik_norm = function(par) {
  n = nrow(as.matrix(data))
  p1 = (n/2)*log(2*pi)
  p2 = (n/2)*log(par[2]^2)
  sum = sum((data - par[1])^2)
  p3 = 1/(2*par[2]^2) * sum
  return(p1+p2+p3)
}
```

```r
# c(mu, sigma)
neg_llik_norm_prime = function(par) {
  n = nrow(as.matrix(data))
  mu_prime = -1/(n*par[2]^2) * sum(data-par[1])
  sigma_prime = 1/(2*par[2]^2) * (n - (1/(par[2]^2)) * sum((data-par[1])^2))

  return(c(mu_prime, sigma_prime))
}

optim(c(0, 1), neg_llik_norm, method = "CG")
optim(c(0, 1), neg_llik_norm, method = "CG", gr = neg_llik_norm_prime)
optim(c(0, 1), neg_llik_norm, method = "BFGS")
optim(c(0, 1), neg_llik_norm, method = "BFGS", gr = neg_llik_norm_prime)
```

Answer: The negative log-likelihood function for the normal distribution is defined by:

$$\mathcal{L}(\mu, \sigma^2, x_1, ..., x_{100}) = \frac{n}{2} ln(2\pi) + \frac{n}{2} ln(\sigma^2) + \frac{1}{2\sigma^2} \sum_{j=1}^{n} (x_j - \mu)^2$$

The estimators are:

$$\hat{\mu}_n = \frac{1}{n} \sum_{j=1}^{n} x_j$$

and

$$\hat{\sigma}_n^2 = \frac{1}{n} \sum_{j=1}^{n} (x_j - \hat{\mu})^2$$

Answer: The partial derivates for the negative log-likelihood are given by:

$$\frac{\partial \mathcal{L}(\mu, \sigma^2, x_1, ..., x_{100})}{\partial \mu} = -\frac{1}{n\sigma^2} \sum_{j=1}^{n} (x_j - \mu)$$

$$\frac{\partial \mathcal{L}(\mu, \sigma^2, x_1, ..., x_{100})}{\partial \sigma^2} = \frac{1}{2\sigma^2} \left(n - \frac{1}{\sigma^2} \sum_{j=1}^{n} (x_j - \mu)^2 \right)$$

5 Sampling Based on Size

Task: Use a uniform random number generator to create a function that selects 1 city from the whole list by the probability scheme offered above (do not use standard sampling functions present in R).

```r
get_city_by_urn_wo = function(city_pool) {

  # We take the cumulative sum and then runif from 1 to max(cumulative sum).
  # This way we respect the proportions. As we need every intermediate result,
  # we use a loop
  cumulative_pop_sum = 0
```

```
  for (i in 1:nrow(city_pool)) {
    cumulative_pop_sum = cumulative_pop_sum + city_pool$Population[i]
    city_pool$CumSum[i] = cumulative_pop_sum
  }

  # Now we get a random value between 1 to max(cumulative sum). As larger muni-
  # cipalities have larger ranges, this works as expected
  selection =
    floor(runif(n = 1, min = 1, max = city_pool$CumSum[nrow(city_pool)]))

  # Return the first city which has a greater CumSum than the selection
  return(city_pool[city_pool$CumSum > selection,][1, c(1, 2)])
}
```

Task: Use the function you have created in step 2 as follows:

a. Apply it to the list of all cities and select one city
b. Remove this city from the list
c. Apply this function again to the updated list of the cities
d. Remove this city from the list
e. ... and so on until you get exactly 20 cities.

Answer: We will combine all of these steps in one function. We're lazy.

```
get_n_cities = function(data, n) {

  # Create a copy to not touch the original data.
  city_pool = data
  selected_cities = data.frame()

  # As long as we don't have n samples, get one and remove it from the pool,
  # as we sample without replacement
  while(nrow(selected_cities) < n) {
    selected_city = get_city_by_urn_wo(city_pool)
    selected_cities = rbind(selected_cities, selected_city)
    city_pool = city_pool[!rownames(city_pool) %in% rownames(selected_cities),]
  }

  return(selected_cities)
}

sample = get_n_cities(data, 20)
```

6 Inverse CDF Method

The double exponential (Laplace) distribution is given by formula

$$DE(\mu, \alpha) = \frac{\alpha}{2} e^{-\alpha|x-\mu|}$$

Task: Write a code generating double exponential distribution $DE(0, 1)$ from $Unif(0, 1)$ by using the inverse CDF method. Explain how you obtained that code step by step. Generate `10000` random numbers from this distribution, plot the histogram and comment whether the result looks reasonable.

1. Derive the CDF from the PDF (`dfunc()`)by taking the integral $\int_{-\infty}^{x} \text{CDF} dx$. This function is the `pfunc()` (CDF!, cumulative).
2. Swap `x` and `y` to receive the quantile function `qfunc()`.
3. Combine both functions to create the `rfunc()`.

This can look like this:

```
# double exponential (Laplace) distribution

# PDF
ddel = function(x = 1, mu = 0, b = 1) {
return(1/(2*b) * exp(-abs(x-mu)/(b)))
}

# CDF
pdel = function(x = 1, mu = 0, b = 1) {
  return(1/2 + 1/2 * sgn(x-mu) * (1 - exp(-abs(x-mu)/b)))
}

# Quantile
qdel = function(p, mu = 0, b = 1) {
  if (p < 0 | p > 1) stop("p must be in range (0, 1)")
  if (p <= 0.5) return(mu + b * log(2 * p))
  return (mu - b * log(2 - 2 * p))
}

# Random
rdel = function(n = 1, mu = 0, b = 1) {
  quantiles = runif(n = n, min = 0, max = 1)
  rdels = sapply(X = quantiles, FUN = qdel, mu = mu, b = b)
  return(rdels)
}
```

To look how the distribution looks like with different parameters, use the following code and plots:

```
sample_rdel_0_1 = rdel(10000, mu = 0, b = 1)
sample_rdel_0_2 = rdel(10000, mu = 0, b = 2)
sample_rdel_0_4 = rdel(10000, mu = 0, b = 4)
sample_rdel_m5_4 = rdel(10000, mu = -5, b = 4)

df = data.frame(sample_rdel_0_1, sample_rdel_0_2,
sample_rdel_0_4, sample_rdel_m5_4)

p1 = ggplot(df) +
  geom_histogram(aes(x = sample_rdel_0_1),
  color = "#FFC300", fill = "#FFC300", binwidth = 0.01) +
  xlim(-10, 10) +
  ylim(0, 60) +
  ggtitle("RD(0, 1)") +
  theme_minimal()

# p2, p3, p4

grid.arrange(p1, p2, p3, p4, nrow = 2)
```

7 Acceptance / Rejection Method

For plotting the PDF and CDF to compare for instance two of them:

```
sequence = seq(from = -10, to = 10, by = 0.01)

dnorm_samples = sapply(X = sequence, FUN = dnorm)
ddel_samples = sapply(X = sequence, FUN = ddel)
pnorm_samples = sapply(X = sequence, FUN = pnorm)
pdel_samples = sapply(X = sequence, FUN = pdel)

df = data.frame(dnorm_samples, ddel_samples, pnorm_samples, pdel_samples)

ggplot(df) +
  geom_line(aes(x = sequence, y = dnorm_samples,
                colour = "Normal Distribution (PDF)")) +
  geom_line(aes(x = sequence, y = ddel_samples,
                colour = "Double Exponential Distribution (PDF)")) +
  labs(title = "dnorm() and ddel()", y = "Density",
  x = "X", color = "Legend") +
  scale_color_manual(values = c("#17202A", "#C70039")) +
  theme_minimal()

ggplot(df) +
  geom_line(aes(x = sequence, y = pnorm_samples,
                colour = "Normal Distribution (CDF)")) +
  geom_line(aes(x = sequence, y = pdel_samples,
                colour = "Double Exponential Distribution (CDF)")) +
  labs(title = "pnorm() and pdel()", y = "Cumulative Density",
  x = "X", color = "Legend") +
  scale_color_manual(values = c("#17202A", "#C70039")) +
  theme_minimal()
```

For calculating the `C` value, use this code:

```
c = max(dnorm_samples / ddel_samples)
```

Sometimes we cannot do this this easily or have to derive the optimal C by hand. This can be done as follows:

$$C \geq \frac{\text{target}}{\text{proposal}} = \frac{f(x)}{g(x)}$$

We then calculate:

$$\max_x \frac{f(x)}{g(x)}$$

Which is taking the derivative and setting it to 0. Then we solve for x and plug in x into

$$\frac{f(x)}{g(x)} = C$$

Note: Sometimes $C = M$.

For creating nice plots, use this:

```r
df$scaled_envelop = c * df$ddel_samples

ggplot(df) +
  geom_line(aes(x = sequence, y = dnorm_samples,
                colour = "Normal Distribution (PDF)")) +
  geom_line(aes(x = sequence, y = ddel_samples,
                colour = "Double Exponential Distribution (PDF)")) +
  geom_line(aes(x = sequence, y = scaled_envelop,
                colour = "Scaled Double Exponential Distribution (PDF)")) +
  labs(title = "Envelope", y = "Density",
  x = "X", color = "Legend") +
  scale_color_manual(values = c("#17202A", "#C70039", "#581845")) +
  theme_minimal()

ggplot(df) +
  geom_ribbon(aes(x = sequence, ymin = df$dnorm_samples, ymax = df$scaled_envelop),
              alpha = 0.8, fill = "#C70039", color = "#C70039") +
  geom_ribbon(aes(x = sequence, ymin = 0, ymax = df$dnorm_samples),
              alpha = 0.8, fill = "#DAF7A6", color = "#DAF7A6") +
  labs(title = "Acceptance and Rejection Regions", y = "Density",
  x = "X", color = "Legend") +
  scale_color_manual(values = c("#17202A", "#C70039", "#581845")) +
  theme_minimal()
```

And for actually using it, use:

```r
rs = c()
rs_rejected = c()

while (length(rs) < 2000) {
  # Take a random sample from our proposal (x-axis)
  z = rdel(n = 1, mu = 0, b = 1)

  # Take a uniform, thus a random y value
  u = runif(n = 1, min = 0, max = c * ddel(z))

  # Check in which region this on lies
  if (u <= dnorm(z)) {
    rs = c(rs, z)
  }
  else {
    rs_rejected = c(rs_rejected, z)
  }
}
```

Plot the drawn samples:

```r
df2 = as.data.frame(rs)

ggplot(df2) +
  geom_histogram(aes(x = rs),
                 color = "#C70039", fill = "#C70039", binwidth = 0.01) +
  xlim(-5, 5) +
  ylim(0, 30) +
  ggtitle("N(0, 1) sampled from DE(0, 1)") +
```

```
  theme_minimal()
```

Expected rejection rate:

```
1 - 1/c
```

Observed rejection rate:

```
length(rs_rejected) / (length(rs)+length(rs_rejected))
```

8 Metropolis-Hastings Algorithm

We have given a target function (probably only a proportional one).

```
# Target function with original scaling
f = function(x) {
  return(120 * x^5 * exp(-x))
}

# Target function
df = function(x) {
  x = ifelse(x <= 0, 0.000001, x)
  return(x^5 * exp(-x))
}

sequence = seq(from = 0.01, to = 20, by = 0.01)

real_f = f(sequence)
plotdf = data.frame(sequence, real_f)

ggplot(plotdf) +
  geom_line(aes(x = sequence, y = real_f), color = "#6091ec") +
  labs(title = "Target Density Function", y = "Density",
  x = "X", color = "Legend") +
  theme_minimal()
```

The Metropolis-Hastings algorthm is implemented by:

```
#' Metropolis Hasting Algorithm
#'
#' @param n Number of samples from the target distribution.
#' @param x_0 Initial state from Pi.
#' @param b Burn-in steps to remove from samples.
#' @param proposal Selects the proposal function.
#' @param keep_burnin Decides if to keep the samples during the burn-in period. #'
#' @return Returns a list containing samples.
#' @export
#'
#' @examples
metropolis_hastings = function(n = 1, x_0 = 1, b = 50, proposal = "lnorm",
                               keep_burnin = FALSE) {

  # Vectors to store the samples in
  samples = c()
```

```r
# Samples from proposel from the random walk (MC)
xt = x_0
xt_1 = x_0

while (length(samples) < n)  {

  # Generate proposal state
  if (proposal == "lnorm") {
    x_star = rlnorm(n = 1, meanlog = log(xt), sdlog = 1)

    # Calculate correction factor C
    c = dlnorm(xt_1, meanlog = log(xt), sdlog = 1) /
    dlnorm(x_star, meanlog = log(xt), sdlog = 1)
  }
  else if (proposal == "chisquared") {
    x_star = rchisq(n = 1, df = floor(xt + 1))

    # Calculate correction factor C
    c = dchisq(x = xt_1, df = floor(xt + 1)) /
    dchisq(x_star, df = floor(xt + 1)) }
  else {
    stop("Invalid proposal.")
  }

  # Calculate acceptance probability alpha
  if (df(xt_1) <= 0) {
    # We need this to avoid troublesome areas where the density is so low that
    # it's 0 from a computiationally point of view.
    alpha = 0
  }
  else {
    alpha = min(1, df(x_star)/df(xt_1) * c)
  }

  # Generate u from uniform
  u = runif(n = 1, min = 0, max = 1)

  # Decide if to accept or reject the proposal
  if (u <= alpha) { # Accept
    xt_1 = xt
    xt = x_star
    samples = c(samples, x_star)
  }
  else {
    # Reject
    xt = xt_1
    }
}

# Return samples
if (keep_burnin) return(samples) {
  return(samples[b+1:length(samples)])
}
```

```
}
```

Create a fancy plot of the burn-in period using:

```
burnin = 10

results = metropolis_hastings(100, x_0 = 40, b = burnin, proposal = "lnorm",
                              keep_burnin = TRUE)

plotdf = data.frame(index = 1:length(results), values = results)

ggplot(plotdf) +
  geom_line(aes(x = index, y = values), color = "#6091ec") +
  labs(title = "Traceplot For The Burn-In Period (burnin = 10)", y = "X*",
  x = "Iteration", color = "Legend") +
  geom_vline(xintercept = burnin, color = "#C70039") +
  theme_minimal()
```

And to plot the distribution of the samples, use:

```
results_lnorm = metropolis_hastings(10000, x_0 = 5, b = 100, proposal = "lnorm")

plotdf = data.frame(results_lnorm)

ggplot(plotdf) +
  geom_histogram(aes(x = results_lnorm),
                 color = "#000000", fill = "#C70039",
                 bins = length(results_lnorm)/100) +
  labs(title = "Samples From Target Function Using Normal Proposal",
       y = "Density",
  x = "X", color = "Legend") +
  theme_minimal()
```

9 Gelman-Rubin Factor

The custom implementation looks like this:

```
gelman_rubin_factor = function(sequence_matrix) {

  k = nrow(sequence_matrix)
  n = ncol(sequence_matrix)

  B = n/(k-1) * sum((rowMeans(sequence_matrix) - mean(sequence_matrix))^2)
  S_squared = rowSums((sequence_matrix - rowMeans(sequence_matrix))^2 / (n - 1))
  W = sum(S_squared / k)
  V = ((n - 1) / n * W) + (1/n * B)
  R = sqrt(V / W)
  return(R)
}
```

And calling it with the previously defined Metropolis-Hastings algorithm, use:

```
k = 10 # row
n = 1000 # col
```

```
sequence_matrix = matrix(NaN, nrow = k, ncol = n)

for (i in 1:k) {
  sequence_matrix[i,] = metropolis_hastings(n, x_0 = i, b = 0,
                                    proposal = "chisquared", keep_burnin = TRUE)
}

print(gelman_rubin_factor(sequence_matrix))

k = 10 # row
n = 5 # col

sequence_matrix = matrix(NaN, nrow = k, ncol = n)

for (i in 1:k) {
  sequence_matrix[i,] = metropolis_hastings(n, x_0 = i, b = 0,
                                    proposal = "chisquared", keep_burnin = TRUE)
}

print(gelman_rubin_factor(sequence_matrix))
```

There is also a built-in implementation which can be invoked with the following code:

```
library(coda)

results5 = list()
results10 = list()
results50 = list()
k = 10

for (i in 1:k) {
  results5[[i]] = mcmc(metropolis_hastings(5, x_0 = i, b = 0,
                                    proposal = "chisquared", keep_burnin = TRUE))
}

for (i in 1:k) {
  results10[[i]] = mcmc(metropolis_hastings(10, x_0 = i, b = 0,
                                    proposal = "chisquared", keep_burnin = TRUE))
}

for (i in 1:k) {
  results50[[i]] = mcmc(metropolis_hastings(50, x_0 = i, b = 0,
                                    proposal = "chisquared", keep_burnin = TRUE))
}

mcmc_list5 = mcmc.list(results5)
gelman.diag(mcmc_list5)

mcmc_list10 = mcmc.list(results10)
gelman.diag(mcmc_list10)

mcmc_list50 = mcmc.list(results50)
gelman.diag(mcmc_list50)
```

10 Monte Carlo Integration

Let's say we want to integrate the following integral:

$$\int_0^\infty x^4 e^{-2x} dx$$

so

$$f(x) = x^4 e^{-2x}$$

For estimating this integral with MCMC we need to split $f(x)$ into $d(x)g(x)$ which can be chosen in any way, as long as the following conditions for $g(x) $are met:

- $g(x) \geq 0$ for the drawn interval.
- The integral inside the interval is finite, so $\int_a^b g(x) = C < \infty$

$g(x)$ must drawn from the whole integral range, so from a to b, and exactly the range. If one of the limits its ∞, $g(x)$ must have the same limit (like a gamma distribution for instance for 0 to ∞).

Then draw samples from $g(x)$, x_i.

Then calculate:

$$\frac{1}{n} \sum_{i=1}^{n} d(x_i)$$

Which is the mean of the $d(x_i)$ values.

In this example this looks like this:

The sampling function is given by:

$$d(x, \alpha = 2, \beta = 2) = 4xe^{-2x}$$

Note that Gamma$(\alpha) = (\alpha - 1)!$

We then identify:

$$g(x) = \frac{1}{4}x^3$$

So we simply put our drawn samples into $g(x)$ and calculate the mean.

```
g = function(x) return(1/4 * x^3)

mean(g(samples))
```

Note: When the limits do not include infinity, we have to scale the results. So we take the result times C where $C = b - a$. This might be wrong, but that's what I got from different examples.

11 Gibbs Sampling

For Gibbs Sampling it is mandatory to find the Marginals. For doing that, do the following:

- First write down the prior and the likelihood. The is no general way for this, it depends on the problem.
- Then write down the posterior which is the product of both.
- For finding the marginals, drop every factor that is not dependend on the marginal variable.

If we have to combine two normal distributions, it can be done using the following rules:

The product of two normal distributions is given by (see https://www.johndcook.com/blog/2012/10/29/product-of-normal-pdfs/ as a recap if not present):

$$\sigma^2_{new} = \frac{\sigma_1^2 \sigma_2^2}{\sigma_1^2 + \sigma_2^2}$$

$$\mu_{new} = \frac{\sigma_1^{-2}\mu_1 + \sigma_2^{-2}\mu_2}{\sigma_1^{-2} + \sigma_2^{-2}}$$

For implementing the Gibbs Sampler we write one function for getting the marginals and one main function for doing the actual sampling.

```
get_mu = function(i, sigma = 0.2, mu, Y) {

  if (i == 1) {
    # First Marginal
    return(rnorm(n = 1, mean = (mu[2] + Y[1])/2, sd = sqrt(1/2 * sigma)))
  }
  else if (i == 50) {
    # Last Marginal
    return(rnorm(n = 1, mean = (2 * mu[i-1] - Y[i-1] + 2 * Y[i])/3,
                 sd = sqrt(2/3 * sigma)))
  }

  # General Marginals
    return(rnorm(n = 1, mean = (2 * mu[i-1] + 2 * mu[i+1] + 2 * Y[i] - Y[i-1])/5,
                 sd = sqrt(2/5 * sigma)))
}

gibbs_sample = function(n, Y, include_matrix = FALSE) {

  # Vectors to store the samples in
  mu_matrix = matrix(0, ncol = length(Y), nrow = n)
  mu = rep(0, length(Y))

  for(j in 1:n) {
    for (i in 1:length(Y)) {
      mu[i] = get_mu(i, sigma = 0.2, mu = mu, Y = Y)
    }
    mu_matrix[j,] = mu
  }

  if (include_matrix) {
    return(list(means = colMeans(mu_matrix), matrix = mu_matrix))
  }
```

```
  return(colMeans(mu_matrix))
}
```

Then this function can be used like this. This also includes a plot of the random walk.

```
gibbs_sample_result = gibbs_sample(1000, Y, TRUE)

data$gibbs_samples = gibbs_sample_result$means

ggplot(data) +
  geom_line(aes(x = X, y = Y), color = "#C70039") +
  geom_line(aes(x = X, y = gibbs_samples), color = "#581845") +
  labs(title = "Dependence from Concentration on Day of Measurement", y =
         "Concentration",  x = "Day of Measurement", color = "Legend") +
  theme_minimal()
```

If a trace plot is needed, use this:

```
df = data.frame(
  iteration = seq(from = 1, to = length(gibbs_sample_result$matrix[,50]), by = 1),
  mu_n = gibbs_sample_result$matrix[,50])

ggplot(df) +
  geom_line(aes(x = iteration, y = mu_n), color = "#C70039") +
  labs(title = "Trace of mu_n", y = "Value of mu_n",  x = "Iteration",
       color = "Legend") +
  theme_minimal()
```

12 Hypothesis Testing

12.1 Defining a Test-Statistics

Pay attention how **exactly** this on is defined, as here for instance Y and $\hat{Y}$ mixed.

$$T = \frac{\hat{Y}(X_b) - \hat{Y}(X_a)}{X_b - X_a}, \text{ where } X_b = \text{argmax}_X Y(X), X_a = \text{argmin}_X Y(X)$$

```
data = data.frame(X = lottery$Day_of_year, Y = lottery$Draft_No)

test_statistics = function(X, Y, Y_hat) {

  b_index = which.max(Y)
  a_index = which.min(Y)

  return((Y_hat[b_index] - Y_hat[a_index]) / (X[b_index] - X[a_index]))
}
```

12.2 Non-Parametric Bootstrap

```
f = function(data, ind) {
  data1 = data[ind,]
```

```
  model = loess(Draft_No ~ Day_of_year, data1)

  T_value =
    test_statistics(data1$Day_of_year, data1$Draft_No, Y_hat = model$fitted)

  return(T_value)
}

# T(D) for the original data
data$Y_hat = loess(Draft_No ~ Day_of_year, lottery)$fitted
T_value_original = test_statistics(data$X, data$Y, data$Y_hat)

# T for the bootstrapped samples
nonparam_bootstrap =
  boot(lottery, statistic = f, R = 2000, parallel = "multicore")
p_value_original = mean(nonparam_bootstrap$t > 0)
```

12.3 Density of non-parametric T-Values

```
df = data.frame(nonparam_bootstrap$t)

ggplot(df) +
  geom_density(aes(x = nonparam_bootstrap.t), color = "black",
               fill = "#dedede", alpha = 0.25) +
  geom_vline(aes(xintercept = T_value_original), color = "orange") +
  labs(title = "Density of non-parametric T-Values",
       y = "Density",  x = "T-Value", color = "Legend") +
  theme_minimal()
```

12.4 Defining the Hypothesis Test (Permutation Test)

```
test_hypothesis = function (data_input, statistics, B = 2000, T_org) {

  t_values = rep(NA, B)

  for (i in 1:B) {

    ind = sample(1:nrow(data_input))
    data_input$X = data_input$X[ind]

    model = loess(Y ~ X, data_input)

    t_values[i] = statistics(data_input$X,
                             data_input$Y, Y_hat = model$fitted)
  }

  return(sum(abs(t_values) >= abs(T_org))/B)
}

p_value_permutated = test_hypothesis(data, test_statistics, 2000, T_value_original)
```

12.5 Crude Estimate of the Power

Question: Make a crude estimate of the power of the test constructed in Step 4:

a) Generate (an obviously non-random) dataset with $n = 366$ observations by using same X as in the original data set and $Y(x) = max(0, min(\alpha x + \beta, 366))$, where $\alpha = 0.1$ and $\beta \sim N(183, sd = 10)$.
b) Plug these data into the permutation test with $B = 200$ and note whether it was rejected.
c) Repeat Steps 5a-5b for $\alpha = 0.2, 0.3, ..., 10$.

What can you say about the quality of your test statistics considering the value of the power?

```
simulate_data = function(X, Y_hat, hypothesis = test_hypothesis,
                         statistics = test_statistics, alpha = 0.1,
                         beta_mean = 183, beta_sd = 10, b = 200, limit = 366) {

  artificial = function(X, alpha) {
    beta = rnorm(n = nrow(lottery), mean = beta_mean, sd = beta_sd)
    return(max(0, min(alpha * X + beta, limit)))
  }

  X_dataframe = X
  Y_dateframe = sapply(X, artificial, alpha)

  Y_hat = loess(Y_dateframe ~ X_dataframe)$fitted

  Y_hat_dataframe = Y_hat

  data_artificial = data.frame(X_dataframe, Y_dateframe, Y_hat_dataframe)
  colnames(data_artificial) = c("X", "Y", "Y_hat")

  T_val = test_statistics(data_artificial$X, data_artificial$Y, data_artificial$Y_hat)

  return(test_hypothesis(data_artificial, statistics, b, T_val))
}

alphas = seq(from = 0.1, to = 10.0, by = 0.1)

no_cores = detectCores()
cl = makeCluster(no_cores)

clusterExport(cl, list("simulate_data", "lottery", "test_hypothesis",
                       "test_statistics", "data"))

simulated_p_values =
  parSapply(cl, alphas, FUN = function(alpha) {
    simulate_data(alpha = alpha, X = lottery$Day_of_year, Y_hat = data$Y_hat)
  })

stopCluster(cl)
```

Plot for p-values with respect to α and p-value line:

```
df = data.frame(alphas, simulated_p_values)

ggplot(df)+
  geom_point(aes(x = alphas, y = simulated_p_values), color = "black",
```

```
            fill = "#dedede", shape = 21) +
  geom_hline(aes(yintercept = 0.05), color = "orange") +
  labs(title = "p-values depending on alpha",
       y = "p-value",  x = "Alpha", color = "Legend") +
  theme_minimal()
```

Calculating the actual power is done by:

```
1 - mean(simulated_p_values > 0.05)
```

13 Boostrap with Bias-Correction

Doing a normal bootstrap.

```
f_prices = function(data, ind) {
  data1 = data[ind,]

  return(mean(data1$Price))
}

house_bootstrap = boot(prices, statistic = f_prices, R = 2000,
                       parallel = "multicore")

plot(house_bootstrap)
```

The estimate of the mean price is:

```
bootstrap_mean_price = mean(house_bootstrap$t)
print(bootstrap_mean_price)
```

The bootstrap-bias-correction is given by the following values. The first one is the bias-correction and the second one is the bias corrected mean.

```
prices_mean = mean(prices$Price)
print(prices_mean - bootstrap_mean_price)
print(2 * prices_mean - bootstrap_mean_price)
```

The variance of the mean is given by:

```
bootstrap_variance_price = as.numeric(var(house_bootstrap$t))
# bootstrap_mean_price = 1 / (B-1) * sum((house_bootstrap$t -
#                                         mean(house_bootstrap$t))^2)

print(bootstrap_variance_price)
```

Now we will create the 95% confidence intervals.

```
confidence_interval = boot.ci(house_bootstrap)
print(confidence_interval)
```

As this output does not include the BCa, we will print the intervals manually. Note that the last to values in each row represent the confidence interval.

```
confidence_interval$percent
confidence_interval$bca
confidence_interval$normal
```

14 Variance Estmation using Jackknife

```
f_prices_jackknife = function(ind, data) {
  data1 = data[-ind,]
  return(mean(data1$Price))
}

# First create the statistics using jackknife
n = length(prices$Price)

indices = seq(from = 1, to = n, by = 1)
jackknife_statistics = sapply(indices, f_prices_jackknife, prices)

# For the variance estimate we will first calculate Ti_star
Ti_star = sapply(jackknife_statistics, FUN = function(tdi, n, prices_mean) {
  return(n * prices_mean - ((n - 1) * tdi))
}, n = n, prices_mean = prices_mean)

# Now we calculate J_T
J_T = mean(Ti_star)

# And now we can calculate the Variance
variance_jackknife = 1 / (n * (n - 1)) * sum((Ti_star - J_T)^2)

print(variance_jackknife)
```

15 Confidence Intervals with Respect to Length and Location

```
boostrap_estimated_mean_corrected = (2 * prices_mean - bootstrap_mean_price)
```

The length of the CI is:

```
print(confidence_interval$percent[5] - confidence_interval$percent[4])
```

The mean is located the following "percent" of the CI range.

```
(boostrap_estimated_mean_corrected - confidence_interval$percent[4]) /
  (confidence_interval$percent[5] - confidence_interval$percent[4]) * 100
```

```
ggplot(prices) +
  geom_histogram(aes(x = prices$Price),
                 color = "#000000", fill = "#dedede") +
  annotate("rect", xmin=confidence_interval$percent[4],
           xmax=confidence_interval$percent[5], ymin=0, ymax=Inf,
           alpha=0.5, fill="#86b4ff") +
  geom_vline(aes(xintercept = boostrap_estimated_mean_corrected),
             color = "orange") +
  labs(title = "Histrogram of Prices",
       y = "Frequency",
  x = "Prices", color = "Legend") +
  theme_minimal()
```

16 Genetic Algorithms

First define the function to minimize or maximize.

```
f = function(x) {
  left = x^2/exp(x)
  exponent = (-9*sin(x))/(x^2+x+1)
  right = 2*exp(exponent)
  return(left-right)
}
```

Plotting this function:

```
sequence = seq(from = 0, to = 30, by = 0.1)
f.sequence = f(sequence)
df = data.frame(sequence, f.sequence)

ggplot(df) +
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  labs(title = "f(x)", y = "f(x)", x = "x") +
  theme_minimal()
```

Max value at `x`.

```
print(sequence[which.max(f.sequence)])
```

And `y` or respect `f(x)` for that:

```
print(max(f.sequence))
```

Then the crossover function, might be different though.

```
crossover = function(x, y) return((x+y)/2)
```

Then the mutation.

```
mutate = function(x) return((x^2)%%30)
```

Initial Population.

```
X = seq(from = 0, to = 30, by = 5)
```

Corresponding values.

```
Values = f(X)
```

And then the algorithm itself. Keep in mind that this one **does not save the overall best individual**.

```
genetic = function(X, Values, maxiter = 100, mutprob = 0.05) {

  best_individual = NaN

  for (i in 1:maxiter) {

    # 1)
    parents = sample(1:length(X), size = 2)

    # 2) I don't know why we should order here!?
    victim = which.min(Values)
```

```
    # 3)
    child = crossover(X[parents][1], X[parents][2])
    if (mutprob > runif(n = 1, min = 0, max = 1)) {
      child= (mutate(child))
    }

    # 4)
    X[victim] = child
    Values[victim] = f(child)

    # 5)
    best_index = which.max(Values)
    best_individual = list(cx = X[best_index], cy = Values[best_index])
  }
  return(list(best = best_individual, population = X))
}

best_individual = genetic(X, Values, 100, 0.05)$best
```

Plot the best individual.

```
df = data.frame(sequence, f.sequence)
best_individual = as.data.frame(best_individual)

ggplot(df) +
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  geom_point(aes(x = cx, y = cy ), data = best_individual, color = "black",
             fill = "#7BA9FF", shape = 21, size = 3, stroke = 2) +
  labs(title = "f(x)", y = "f(x)", x = "x") +
  theme_minimal()
```

Plot the initial distribution.

```
df = data.frame(sequence, f.sequence)
initial.population = data.frame(X, Values)

ggplot(df) +
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  geom_point(aes(x = X, y = Values ), data = initial.population, color = "black",
             fill = "#DAF7A6", shape = 21, size = 3, stroke = 2) +
  labs(title = "f(x)", y = "f(x)", x = "x") +
  theme_minimal()
```

Six grid plot with different settings.

```
set.seed(12345)

genetic1 = genetic(X, Values, 10, 0.1)
genetic2 = genetic(X, Values, 10, 0.5)
genetic3 = genetic(X, Values, 10, 0.9)
genetic4 = genetic(X, Values, 100, 0.1)
genetic5 = genetic(X, Values, 100, 0.5)
genetic6 = genetic(X, Values, 100, 0.9)

population1 = data.frame(genetic1$population, f(genetic1$population))
best1 = as.data.frame(genetic1$best)
```

```
population2 = data.frame(genetic2$population, f(genetic2$population))
best2 = as.data.frame(genetic2$best)

population3 = data.frame(genetic3$population, f(genetic3$population))
best3 = as.data.frame(genetic3$best)

population4 = data.frame(genetic4$population, f(genetic4$population))
best4 = as.data.frame(genetic4$best)

population5 = data.frame(genetic5$population, f(genetic5$population))
best5 = as.data.frame(genetic5$best)

population6 = data.frame(genetic6$population, f(genetic6$population))
best6 = as.data.frame(genetic6$best)

p1 = ggplot(df) +
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  geom_point(aes(x = genetic1.population, y = f.genetic1.population.),
             data = population1, color = "black",
             fill = "#FFC300", shape = 21, size = 2, stroke = 1) +
  geom_point(aes(x = cx, y = cy), data = best1, color = "black",
             fill = "#7BA9FF", shape = 21, size = 2, stroke = 1) +
  labs(title = "maxiter = 10, mutprob = 0.1", y = "f(x)", x = "x") +
  theme_minimal()

p2 = ggplot(df) +
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  geom_point(aes(x = genetic2.population, y = f.genetic2.population.),
             data = population2, color = "black",
             fill = "#FFC300", shape = 21, size = 2, stroke = 1) +
  geom_point(aes(x = cx, y = cy), data = best2, color = "black",
             fill = "#7BA9FF", shape = 21, size = 2, stroke = 1) +
  labs(title = "maxiter = 10, mutprob = 0.5", y = "f(x)", x = "x") +
  theme_minimal()

p3 = ggplot(df) +
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  geom_point(aes(x = genetic3.population, y = f.genetic3.population.),
             data = population3, color = "black",
             fill = "#FFC300", shape = 21, size = 2, stroke = 1) +
  geom_point(aes(x = cx, y = cy), data = best3, color = "black",
             fill = "#7BA9FF", shape = 21, size = 2, stroke = 1) +
  labs(title = "maxiter = 10, mutprob = 0.9", y = "f(x)", x = "x") +
  theme_minimal()

p4 = ggplot(df) +
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  geom_point(aes(x = genetic4.population, y = f.genetic4.population.),
             data = population4, color = "black",
             fill = "#FFC300", shape = 21, size = 2, stroke = 1) +
  geom_point(aes(x = cx, y = cy), data = best4, color = "black",
             fill = "#7BA9FF", shape = 21, size = 2, stroke = 1) +
  labs(title = "maxiter = 100, mutprob = 0.1", y = "f(x)", x = "x") +
```

```
  theme_minimal()

p5 = ggplot(df) +
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  geom_point(aes(x = genetic5.population, y = f.genetic5.population.),
             data = population5, color = "black",
             fill = "#FFC300", shape = 21, size = 2, stroke = 1) +
  geom_point(aes(x = cx, y = cy), data = best5, color = "black",
             fill = "#7BA9FF", shape = 21, size = 2, stroke = 1) +
  labs(title = "maxiter = 100, mutprob = 0.5", y = "f(x)", x = "x") +
  theme_minimal()

p6 = ggplot(df) +
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  geom_point(aes(x = genetic6.population, y = f.genetic6.population.),
             data = population6, color = "black",
             fill = "#FFC300", shape = 21, size = 2, stroke = 1) +
  geom_point(aes(x = cx, y = cy), data = best6, color = "black",
             fill = "#7BA9FF", shape = 21, size = 2, stroke = 1) +
  labs(title = "maxiter = 100, mutprob = 0.9", y = "f(x)", x = "x") +
  theme_minimal()

grid.arrange(p1, p2, p3, p4, p5, p6, nrow = 3)
```

17 Expectation-Maximization Algorithm (EM)

Plot for two lines in one graph.

```
ggplot(data) +
  geom_line(aes(x = X, y = Z, color = "Z")) +
  geom_line(aes(x = X, y = Y, color = "Y")) +
  labs(title = "Dependence of Z and Y versus X", y = "Z/Y", x = "X") +
  scale_color_manual(values = c("#C70039", "#581845")) +
  theme_minimal()
```

In general, perform:

- E-Step (Expectation)
- M-Step (Maximazation)

17.1 Expectation-Step (E-Step)

For the E-Step, we write down:

$$Q(\theta, \theta^k) = E[\mathcal{L}(\theta|Y, Z)|\theta^k, Y]$$

where θ is the unknown parameter (for instance λ). Then we take the log likelihood and estimate the latent parameters, for instance by using the expected value. This depends on the distribution. The log likelihood is usually the log of the product of the given distributions.

It could look like this:

$$\mathcal{L}(\lambda|Y,Z) \sim ln\left(\prod_{i=1}^{n}\frac{X_i}{\lambda}e^{-X_i/\lambda} * \frac{X_i}{2\lambda}e^{-X_i/2\lambda}\right)$$

And then taking the expected value for the missing variables.

$$E[\mathcal{L}(\lambda|Y,Z)] = -n * ln(2\lambda^2) + 2\sum_{i=1}^{n} ln(X_i) - \sum_{i=1}^{n}\frac{X_iY_i}{\lambda} - E\left[\sum_{i=1}^{n}\frac{X_iZ_i}{2\lambda}\right]$$

So the E-Step concludes with:

$$Q(\lambda,\lambda^k) = E[\mathcal{L}(\lambda|Y,Z)] = -n * ln(2\lambda^2) + 2\sum_{i=1}^{n} ln(X_i) - \sum_{i=1}^{n}\frac{X_iY_i}{\lambda} - \sum_{i=1}^{\beta}\frac{X_iZ_i}{2\lambda} - (n-\beta)\frac{\lambda_k}{\lambda}$$

17.2 Maximazation-Step (M-Step)

Usually this can be done by taking the derivative with respect to the parameters to estimate. There might be different ways to do the maximization though.

$$\frac{\partial Q(\lambda,\lambda^k)}{\partial\lambda} = -\frac{2n}{\lambda} + \sum_{i=1}^{n}\frac{X_iY_i}{\lambda^2} + \frac{1}{2}\sum_{i=1}^{\beta}\frac{X_iZ_i}{\lambda^2} + (n-\beta)\frac{\lambda_k}{\lambda^2}$$

And then simply solve for the parameter that maximizes.

$$\lambda = \frac{\sum_{i=1}^{n} X_iY_i + \frac{1}{2}\sum_{i=1}^{\beta} X_iZ_i + (n-\beta)\lambda_k}{2n}$$

17.3 EM-Implementation

The implementatio can look like this:

```
# To keep it simple we will not pass the data to the function.
lambda_estimate_em = function (input_iterations = 100, input_threshold = 0.0001) {

  iterations_max = input_iterations
  iterations = 0
  threshold = input_threshold

  n = nrow(data)
  lambdas = c()
  lambda = NaN
  lambda_k = 100

  Z = data$Z[!is.na(data$Z)]
  A = data$Z[is.na(data$Z)]

  Z_index =  which(!is.na(data$Z))
  A_index =  which(is.na(data$Z))

  X = data$X[Z_index]
```

```
  X_A = data$X[A_index]

  beta = length(Z)

  for (i in 1:iterations_max) {
    iterations = iterations + 1

    # E/M-Step
    lambda = lambda_k
    lambda_k = (sum(data$X * data$Y) + 0.5 * sum(X * Z) +
                  ((n - beta) * lambda_k)) / (2 * n)
    lambdas = c(lambdas, lambda_k)

    if (abs(lambda_k - lambda) < threshold ) break
  }

  return(list(lambdas, iterations))
}

em_result = lambda_estimate_em()
lambda_result = em_result[[1]][em_result[[2]]]

print(em_result)
```

If we have to plot the expected values, take the formula from that from the distribution, for instance for the exponential one it would look like this.

$$E[Y] = \frac{\lambda}{X_i}, \quad E[Y] = \frac{2\lambda}{X_i}$$

```
data$Y_E = (lambda_result)/data$X
data$Z_E = (2*lambda_result)/data$X

ggplot(data) +
  geom_line(aes(x = X, y = Z, color = "Z")) +
  geom_line(aes(x = X, y = Y, color = "Y")) +
    geom_line(aes(x = X, y = Z_E, color = "E[Z]")) +
  geom_line(aes(x = X, y = Y_E, color = "E[Y]")) +
  labs(title = "Dependence of Z, Y, E[Z] and E[Y] versus X", y = "Z/Y", x = "X") +
  scale_color_manual(values = c("#581845", "#FFC300", "#C70039", "#6091ec")) +
  theme_minimal()
```

18 Miscellaneous Plots

18.1 Histrogram

General rool for n datapoints: $\sqrt{n}$ bars.

```
ggplot(sample)+
  geom_histogram(aes(x = Population), bins = nrow(sample), color = "black",
                 fill = "#C70039") +
  ggtitle("Histogram of selected cities")
```

18.2 Histrogram with Mean

```
prices_mean = mean(prices$Price)
print(prices_mean)

ggplot(prices) +
  geom_histogram(aes(x = prices$Price, y=..density..), color = "black", fill = "#dedede") +
  geom_vline(aes(xintercept = prices_mean), color = "#FFC300") +
  labs(title = "Histrogram of Prices",
       y = "Density",
  x = "Prices", color = "Legend") +
  theme_minimal()
```

18.3 Simple X/Y Plot

```
ggplot(data) +
  geom_line(aes(x = X, y = Y), color = "#c70039") +
  labs(title = "Dependence from Concentration on Day of Measurement",
       y = "Concentration",  x = "Day of Measurement", color = "Legend") +
  theme_minimal()
```

18.4 Variance Plot

```
for (i in 1:length(v)) {
  Xi = v[1:i]
  X$vec_customvar[[i]] = custom_variance(as.vector(Xi))
}

ggplot(X[2:nrow(X),]) +
  geom_point(aes(x = index, y = value, colour = "Difference")) +
  geom_point(aes(x = index, y = vec_myvar,  colour = "my_var()")) +
  geom_point(aes(x = index, y = vec_var, colour = "var()")) +
  geom_point(aes(x = index, y = vec_customvar, colour = "custom_variance()")) +
  labs(title = "Difference in Variance", y = "Variance",
  x = "Sequence", color = "Legend") +
  scale_color_manual(values = c("#17202A", "#C70039", "#407AFF", "#FFC300")) +
  scale_x_log10() +
  theme_minimal()
```

18.5 Scatterplot With Geom Smoother

```
ggplot(lottery) +
  geom_point(aes(x = Day_of_year, y = Draft_No), color = "black",
             fill = "#dedede", shape = 21) +
  geom_smooth(mapping = aes(x = Day_of_year, y = Draft_No),
              method = "loess", size = 1.5, color = "#000000") +
  labs(title = "Day of Year VS Draft Number",
       y = "Draft Number",  x = "Day of Year", color = "Legend") +
  theme_minimal()
```

19 Useful Code Snippets

19.1 RMarkdown Setup

```
# ---
#title: "Computational Statistics Summary"
#author: "Maximilian Pfundstein"
#date: "`r Sys.Date()`"
#output:
#  html_document:
#    df_print: paged
#    toc: true
#    toc_float: true
#    number_sections: true
#  pdf_document:
#    toc: true
#    toc_depth: 3
#    number_sections: true
# ---
```

19.2 Knitr options

```
#{r setup, include=FALSE}
#knitr::opts_chunk$set(echo = TRUE, cache = TRUE, include = TRUE, eval = FALSE,
#                      fig.pos = 'H')
```

19.3 Including Source Code

```
#{r, ref.label=knitr::all_labels(), echo = TRUE, eval = FALSE, results = 'show'}
```

The Clever Machine

Topics in Computational Neuroscience & Machine Learning

MCMC: The Gibbs Sampler

NOV 5
Posted by **dustinstansbury**

In the previous post, (https://theclevermachine.wordpress.com/2012/11/04/mcmc-multivariate-distributions-block-wise-component-wise-updates/) we compared using block-wise and component-wise implementations of the Metropolis-Hastings algorithm for sampling from a multivariate probability distribution$p(\mathbf{x})$. Component-wise updates for MCMC algorithms are generally more efficient for multivariate problems than blockwise updates in that we are more likely to accept a proposed sample by drawing each component/dimension independently of the others. However, samples may still be rejected, leading to excess computation that is never used. The Gibbs sampler, another popular MCMC sampling technique, provides a means of avoiding such wasted computation. Like the component-wise implementation of the Metropolis-Hastings algorithm, the Gibbs sampler also uses component-wise updates. However, unlike in the Metropolis-Hastings algorithm, all proposed samples are accepted, so there is no wasted computation.

The Gibbs sampler is applicable for certain classes of problems, based on two main criterion. Given a target distribution $p(\mathbf{x})$, where $\mathbf{x} = (x_1, x_2, \ldots, x_D)$, The first criterion is 1) that it is necessary that we have an analytic (mathematical) expression for the conditional distribution of each variable in the joint distribution given all other variables in the joint. Formally, if the target distribution $p(\mathbf{x})$ is D -dimensional, we must have D individual expressions for

$$p(x_i|x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_D)$$

$$= p(x_i|x_j), j \neq i.$$

Each of these expressions defines the probability of the i-th dimension given that we have values for all other ($j \neq i$) dimensions. Having the conditional distribution for each variable means that we don't need a proposal distribution or an accept/reject criterion, like in the Metropolis-Hastings algorithm. Therefore, we can simply sample from each conditional while keeping all other variables held fixed. This leads to the second criterion 2) that we must be able to sample from each conditional distribution. This caveat is obvious if we want an implementable algorithm.

The Gibbs sampler works in much the same way as the component-wise Metropolis-Hastings algorithms except that instead drawing from a proposal distribution for each dimension, then accepting or rejecting the proposed sample, we simply draw a value for that dimension according to the variable's corresponding conditional distribution. We also accept all values that are drawn.

Similar to the component-wise Metropolis-Hastings algorithm, we step through each variable sequentially, sampling it while keeping all other variables fixed. The Gibbs sampling procedure is outlined below

1. set $t = 0$
2. generate an initial state $\mathbf{x}^{(0)} \sim \pi^{(0)}$
3. repeat until $t = M$

set $t = t + 1$

for each dimension $i = 1..D$

draw x_i from $p(x_i|x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_D)$

To get a better understanding of the Gibbs sampler at work, let's implement the Gibbs sampler to solve the same multivariate sampling problem addressed in the previous post.

Example: Sampling from a bivariate a Normal distribution

This example parallels the examples in the previous post where we sampled from a 2-D Normal distribution using block-wise and component-wise Metropolis-Hastings algorithms. Here, we show how to implement a Gibbs sampler to draw samples from the same target distribution. As a reminder, the target distribution $p(\mathbf{x})$ is a Normal form with following parameterization:

$$p(\mathbf{x}) = \mathcal{N}(\mu, \Sigma)$$

with mean

$$\mu = [\mu_1, \mu_2] = [0, 0]$$

and covariance

$$\Sigma = \begin{bmatrix} 1 & \rho_{12} \\ \rho_{21} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0.8 \\ 0.8 & 1 \end{bmatrix}$$

In order to sample from this distribution using a Gibbs sampler, we need to have in hand the conditional distributions for variables/dimensions x_1 and x_2:

$p(x_1|x_2^{(t-1)})$ (i.e. the conditional for the first dimension, x_1)

and

$p(x_2|x_1^{(t)})$ (the conditional for the second dimension, x_2)

Where $x_2^{(t-1)}$ is the previous state of the second dimension, and $x_1^{(t)}$ is the state of the first dimension after drawing from $p(x_1|x_2^{(t-1)})$. The reason for the discrepancy between updating x_1 and x_2 using states $(t-1)$ and (t), can be is seen in step 3 of the algorithm outlined in the previous section. At iteration t we first sample a new state for variable x_1 conditioned on the most recent state of variable x_2, which is from iteration $(t-1)$. We then sample a new state for the variable x_2 conditioned on the most recent state of variable x_1, which is now from the current iteration, t.

After some math (which which I will skip for some brevity, but see the following

(http://fourier.eng.hmc.edu/e161/lectures/gaussianprocess/node7.html) for some details), we find that the two conditional distributions for the target Normal distribution are:

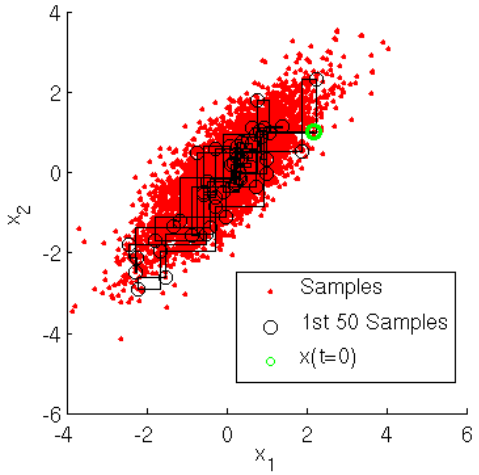
$$p(x_1|x_2^{(t-1)}) = \mathcal{N}(\mu_1 + \rho_{21}(x_2^{(t-1)} - \mu_2), \sqrt{1-\rho_{21}^2})$$

and

$$p(x_2|x_1^{(t)}) = \mathcal{N}(\mu_2 + \rho_{12}(x_1^{(t)} - \mu_1), \sqrt{1-\rho_{12}^2}),$$

which are both univariate Normal distributions, each with a mean that is dependent on the value of the most recent state of the conditioning variable, and a variance that is dependent on the target covariances between the two variables.

Using the above expressions for the conditional probabilities of variables x_1 and x_2, we implement the Gibbs sampler using MATLAB below. The output of the sampler is shown here:



(https://theclevermachine.files.wordpress.com/2012/11/gibbssampler-2dnormal1.png)
Gibbs sampler Markov chain and samples for bivariate Normal target distribution

Inspecting the figure above, note how at each iteration the Markov chain for the Gibbs sampler first takes a step only along the x_1 direction, then only along the x_2 direction. This shows how the Gibbs sampler sequentially samples the value of each variable separately, in a component-wise fashion.


```
% EXAMPLE: GIBBS SAMPLER FOR BIVARIATE NORMAL
rand('seed' ,12345);
nSamples = 5000;

mu = [0 0]; % TARGET MEAN
rho(1) = 0.8; % rho_21
rho(2) = 0.8; % rho_12

% INITIALIZE THE GIBBS SAMPLER
propSigma = 1; % PROPOSAL VARIANCE
minn = [-3 -3];
maxx = [3 3];
```

Wrapping Up

The Gibbs sampler is a popular MCMC method for sampling from complex, multivariate probability distributions. However, the Gibbs sampler cannot be used for general sampling problems. For many target distributions, it may difficult or impossible to obtain a closed-form expression for all the needed conditional distributions. In other scenarios, analytic expressions may exist for all conditionals but it may be difficult to sample from any or all of the conditional distributions (in these scenarios it is common to use univariate sampling methods such as rejection sampling and (surprise!) Metropolis-type MCMC techniques to approximate samples from each conditional). Gibbs samplers are very popular for Bayesian methods where models are often devised in such a way that conditional expressions for all model variables are easily obtained and take well-known forms that can be sampled from efficiently.

Gibbs sampling, like many MCMC techniques suffer from what is often called "slow mixing." Slow mixing occurs when the underlying Markov chain takes a long time to sufficiently explore the values of $\mathbf{x}$ in order to give a good characterization of $p(\mathbf{x})$. Slow mixing is due to a number of factors including the "random walk" nature of the Markov chain, as well as the tendency of the Markov chain to get "stuck," only sampling a single region of $\mathbf{x}$ having high-probability under $p(\mathbf{x})$. Such behaviors are bad for sampling distributions with multiple modes or heavy tails. More advanced techniques, such as Hybrid Monte Carlo have been developed to incorporate additional dynamics that increase the efficiency of the Markov chain path. We will discuss Hybrid Monte Carlo in a future post.

About dustinstansbury

I recently received my PhD from UC Berkeley where I studied computational neuroscience and machine learning.

View all posts by dustinstansbury »

Posted on November 5, 2012, in Algorithms, Density Estimation, Sampling, Sampling Methods, Statistics and tagged Conditional Distribution, Gibbs Sampler, MCMC, Multivariate Normal. Bookmark the permalink. 4 Comments.

Leave a comment

Trackbacks 2

Comments 2

markovmagic | August 28, 2013 at 6:21 pm
Reblogged this on Machine Learning Magic and commented:
This man is a genius.

chjko0206 | February 9, 2015 at 12:11 am
Reblogged this on michaelhjc.

1. Pingback: **A Gentle Introduction to Markov Chain Monte Carlo (MCMC) « The Clever Machine**
2. Pingback: **MCMC: The Gibbs Sampler, simple example w/ Matlab code | Victor Fang's Computing Space**

The Clever Machine

Topics in Computational Neuroscience & Machine Learning

Blog Archives

A Gentle Introduction to Markov Chain Monte Carlo (MCMC)

NOV 19
Posted by **dustinstansbury**

Applying probabilistic models to data usually involves integrating a complex, multi-dimensional probability distribution. For example, calculating the expectation/mean of a model distribution involves such an integration. Many (most) times, these integrals are not calculable due to the high dimensionality of the distribution or because there is no closed-form expression for the integral available using calculus. Markov Chain Monte Carlo (MCMC) is a method that allows one to approximate complex integrals using stochastic sampling routines. As MCMC's name indicates, the method is composed of two components, the ***Markov chain*** and ***Monte Carlo integration***.

Monte Carlo integration is a powerful technique that exploits stochastic sampling of the distribution in question in order to approximate the difficult integration. However, in order to use Monte Carlo integration it is necessary to be able to sample from the probability distribution in question, which may be difficult or impossible to do directly. This is where the second component of MCMC, the Markov chain, comes in. A Markov chain is a sequential model that transitions from one state to another in a probabilistic fashion, where the next state that the chain takes is conditioned on the previous state. Markov chains are useful in that if they are constructed properly, and allowed to run for a long time, the states that a chain will take also sample from a target probability distribution. Therefore we can construct Markov chains to sample from the distribution whose integral we would like to approximate, then use Monte Carlo integration to perform the approximation.

Here I introduce a series of posts where I describe the basic concepts underlying MCMC, starting off by describing Monte Carlo Integration (https://theclevermachine.wordpress.com/2012/09/22/monte-carlo-approximations/), then giving a brief introduction of Markov chains (https://theclevermachine.wordpress.com/2012/09/24/a-brief-introduction-to-markov-chains/) and how they can be constructed to sample from a target probability distribution. Given these

foundation principles, we can then discuss MCMC techniques such as the Metropolis (https://theclevermachine.wordpress.com/2012/10/05/mcmc-the-metropolis-sampler/) and Metropolis-Hastings (https://theclevermachine.wordpress.com/2012/10/20/mcmc-the-metropolis-hastings-sampler/) algorithms, the Gibbs sampler (https://theclevermachine.wordpress.com/2012/11/05/mcmc-the-gibbs-sampler/), and the Hybrid Monte Carlo (https://theclevermachine.wordpress.com/2012/11/18/mcmc-hamiltonian-monte-carlo-a-k-a-hybrid-monte-carlo/) algorithm.

As always, each post has a somewhat formal/mathematical introduction, along with an example and simple Matlab implementations of the associated algorithms.

Posted in Algorithms, MCMC, Sampling Methods, Statistics

Tags: Gibbs Sampler, Hamiltonian Monte Carlo, Hybrid Monte Carlo, integral approximation, integration, Markov Chain, Markov Chain Monte Carlo, MCMC, Metropolis sampler, Metropolis-Hastings Sampler, Monte Carlo Integration

9 Comments

Monte Carlo Approximations

SEP 22
Posted by **dustinstansbury**

Monte Carlo Approximation for Integration

Using statistical methods we often run into integrals that take the form:

$I = \int_a^b h(x)g(x)dx$

For instance, the expected value of a some function $f(x)$ of a random variable X

$\mathbb{E}[x] = \int_{p(x)} p(x)f(x)dx$

and many quantities essential for Bayesian methods such as the marginal likelihood a.k.a "model evidence"

$p(x) = \int_\theta p(x|\theta)p(\theta)dx$

involve integrals of this form. Sometimes (not often) such an integral can be evaluated analytically. When a closed form solution does not exist, numeric integration methods (http://en.wikipedia.org/wiki/Numerical_integration) can be applied. However numerical methods quickly become intractable for any practical application that requires more than a small number of dimensions. This is where Monte Carlo approximation comes in. Monte Carlo approximation allows us to calculate an estimate for the value of I by transforming the integration problem into a procedure of sampling values from a tractable probability distribution and calculating the average of those samples. Here's what I mean:

If the function $g(x)$ fullfills two simple criteria, namely that the function is always positive on the interval (a, b)

$g(x) \geq 0, x \in (a, b)$

and that the integral of the function is finite

$\int_a^b g(x) = C < \infty$

then we can define a corresponding probability distribution on the interval (a, b):

$p(x) = \frac{g(x)}{C}$

Another way to think of it is that $g(x)$ is a probability distribution scaled by a constant C.

Using this link between probability distributions $p(x)$ and $g(x)$, we can restate the original integration as

$I = C \int_a^b h(x)p(x)dx = C\mathbb{E}_{p(x)}[h(x)]$

This statement says that if we can sample values of x using $p(x)$, then the value of the original integral I is simply a scaled version of the expected value of the integrand function $h(x)$ calculated using those samples. Turns out that the expected value $\mathbb{E}_{p(x)}[h(x)]$ can be easily approximated by the sample mean:

$\mathbb{E}_{p(x)}[h(x)] \approx \frac{1}{N} \sum_i^N h(x_i)$

where samples $x_i, i = 1..N$ are drawn independently from $p(x)$. This leads to a simple 4-Step Procedure for performing Monte Carlo approximation to the integral I:

1. Identify $h(x)$
2. Identify $g(x)$ and from it determine $p(x)$ and C
3. Draw N independent samples from $p(x)$
4. Evaluate $I = C\mathbb{E}[h(x)] \approx \frac{C}{N} \sum_i^N h(x_i)$

The larger the number of samples N we draw, the better our approximation to the actual value of I. This 4-step procedure is demonstrated in a some toy examples below:

Example 1: Approximating the integral xe^x

Say we want to calculate the integral:

$I = \int_0^1 xe^x dx$

We can calculate the closed form solution of this integral using integration by parts:

$u = x, dv = e^x$

$du = dx, v = e^x$

and

$I = uv - \int vdu$

$= xe^x - \int e^x dx$

$= xe^x - e^x|_0^1$

$= e^x(x-1)|_0^1$

$= 0 - (-1) = 1$

Orr…we could calculate the Monte Carlo approximation of this integral.

Step 1 we identify

$h(x) = xe^x$

Step 2 we identify

$g(x) = 1$

and from this can also determine the probability distribution function $p(x) \in (a, b) = (0, 1)$. According to the definition expression for $p(x)$ given above we detemine $p(x)$ to be:

$p(x) = \frac{g(x)}{\int_a^b g(x)dx} = \frac{1}{b-a}$.

Step 3: The expression on the right is the definition for the uniform distribution $Unif(0, 1)$, which is easy to sample from using the MATLAB $\mathrm{rand}()$ (Notice too that the constant $C = b - a = 1$).

Step 4: we calculate the Monte Carlo approximation as

$I = C\mathbb{E}_{p(x)}h(x)$

$\approx \frac{1}{N} \sum_{i=1}^N x_i e^{x_i}$,

where each x_i is sampled from the standard uniform distribution. Below is some MATLAB code running the Monte Carlo Approximation for two different values of N

```
% MONTE CARLO APPROXIMATION OF INT(xexp(x))dx
% FOR TWO DIFFERENT SAMPLE SIZES
rand('seed',12345);

% THE FIRST APPROXIMATION USING N1 = 100 SAMPLES
N1 = 100;
x = rand(N1,1);
Ihat1 = sum(x.*exp(x))/N1

% A SECOND APPROXIMATION USING N2 = 5000 SAMPLES
N2 = 5000;
x = rand(N2,1);
Ihat2 = sum(x.*exp(x))/N2
```

Comparing the values of the variables $\mathrm{Ihat1}$ and $\mathrm{Ihat2}$ we see that the Monte Carlo approximation is better for a larger number of samples.

Example 2: Approximating the expected value of the Beta distribution

Lets look at how the 4-step Monte Carlo approximation procedure can be used to calculate expectations. In this example we will calculate

$\mathbb{E}[x] = \int_{p(x)} p(x)xdx,$

where $x \sim p(x) = Beta(\alpha_1, \alpha_2)$.

Step 1: we identify $h(x) = x$.

Step 2: the function $g(x)$ is simply the probability density function $p(x)$ due the expression for $p(x)$ above:

$p(x)^\star = \frac{p(x)}{\int p(x)dx} = p(x).$

Step 3 we can use MATLAB to easily draw N independent samples $p(x)$ using the function $\text{betarnd}()$. And finally,

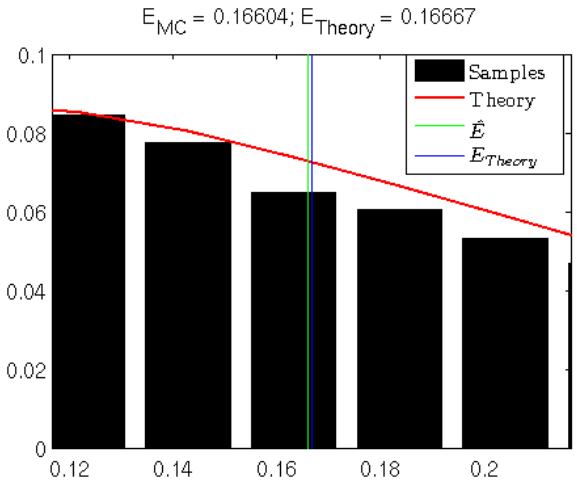
Step 4 we approximate the expectation with the expression

$\mathbb{E}[x]_{Beta(\alpha_1,\alpha_2)} \approx \frac{1}{N}\sum_i x_i$

Below is some MATLAB code that performs this approximation of the expected value.

```
rand('seed',12345);
alpha1 = 2; alpha2 = 10;
N = 10000;
x = betarnd(alpha1,alpha2,1,N);

% MONTE CARLO EXPECTATION
expectMC = mean(x);

% ANALYTIC EXPRESSION FOR BETA MEAN
expectAnalytic = alpha1/(alpha1 + alpha2);

% DISPLAY
figure;
bins = linspace(0,1,50);
counts = histc(x,bins);
probSampled = counts/sum(counts);
probTheory = betapdf(bins,alpha1,alpha2);
b = bar(bins,probSampled); colormap hot; hold on;
t = plot(bins,probTheory/sum(probTheory),'r','Linewidth',2)
m = plot([expectMC,expectMC],[0 .1],'g')
e = plot([expectAnalytic,expectAnalytic],[0 .1],'b')
xlim([expectAnalytic - .05,expectAnalytic + 0.05])
legend([b,t,m,e],{'Samples','Theory','$\hat{E}$','$E_{Theory}$'},'interpret
title(['E_{MC} = ',num2str(expectMC),'; E_{Theory} = ',num2str(expectAnalyt
hold off
```

And the output of the code:



(https://theclevermachine.files.wordpress.com/2012/09/montecarloexpectation1.png)
Monte Carlo Approximation of the the Expected value of Beta(2,10)

The analytical solution (http://en.wikipedia.org/wiki/Beta_distribution) for the expected value of this Beta distribution:

$\mathbb{E}_{Beta(2,10)}[x] = \frac{\alpha_1}{\alpha_1+\alpha_2} = \frac{2}{12} = 0.167$

is quite close to our approximation (also indicated by the small distance between the blue and green lines on the plot above).

Monte Carlo Approximation for Optimization

Monte Carlo Approximation can also be used to solve optimization problems of the form:

$\hat{x} = \underset{x\in(a,b)}{\text{argmax}}\ g(x)$

If $g(x)$ fulfills the same criteria described above (namely that it is a scaled version of a probability distribution), then (as above) we can define the probability function

$p(x) = \frac{g(x)}{C}$

This allows us to instead solve the problem

$\hat{x} = \underset{x}{\text{argmax}}\ Cp(x)$

If we can sample from $p(x)$, the solution $\hat{x}$ is easily found by drawing samples from $p(x)$ and determining the location of the samples that has the highest density (Note that the solution is not dependent of the value of C). The following example demonstrates Monte Carlo optimization:

Example: Monte Carlo Optimization of $g(x) = e^{-\frac{(x-4)^2}{2}}$

Say we would like to find the value of x_{opt} which optimizes the function $g(x) = e^{-((x-4)^2)/2}$. In other words we want to solve the problem

$$x_{opt} = \underset{x}{\operatorname{argmax}}\ e^{-\frac{(x-4)^2}{2}}$$

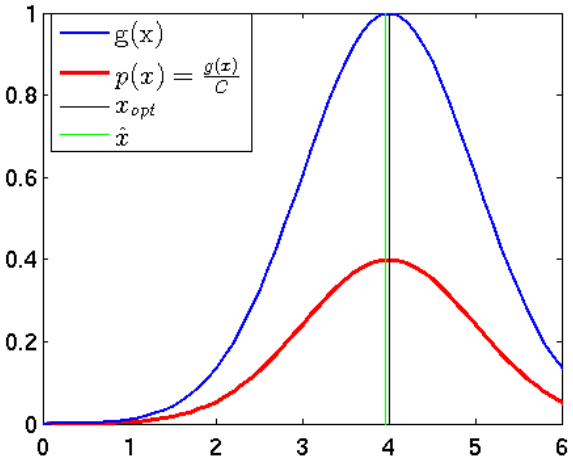
We could solve for x_{opt} using standard calculus methods, but a clever trick is to use Monte Carlo approximation to solve the problem. First, notice that $g(x)$ is a scaled version of a Normal distribution with mean equal to 4 and unit variance:

$$g(x) = C \times \frac{1}{\sqrt{2\pi}} e^{-\frac{(x-4)^2}{2}}$$

$$= C \times \mathcal{N}(4,1)$$

where $C = \sqrt{2\pi}$. This means we can solve for x_{opt} by drawing samples from the normal distribution and determining where those samples have the highest density. The following chunk of matlab code solves the optimization problem in this way.

```
% MONTE CARLO OPTIMIZATION OF exp(x-4)^2
randn('seed',12345)

% INITIALZIE
N = 100000;
x = 0:.1:6;
C = sqrt(2*pi);
g = inline('exp(-.5*(x-4).^2)','x');
ph = plot(x,g(x)/C,'r','Linewidth',3); hold on
gh = plot(x,g(x),'b','Linewidth',2); hold on;
y = normpdf(x,4,1);

% CALCULATE MONTE CARLO APPROXIMATION
x = normrnd(4,1,1,N);
bins = linspace(min(x),max(x),100);
counts = histc(x,bins);
[~,optIdx] = max(counts);
xHat = bins(optIdx);

% OPTIMA AND ESTIMATED OPTIMA
oh = plot([4 4],[0,1],'k');
hh = plot([xHat,xHat],[0,1],'g');

set(gca,'fontsize',16)
legend([gh,ph,oh,hh],{'g(x)','$p(x)=\frac{g(x)}{C}$','$x_{opt}$','$\hat{x}$
```

(https://theclevermachine.files.wordpress.com/2012/09/montecarlooptimization.png)
Monte Carlo Optimization

In the code output above we see the function $g(x)$ we want to optimize in blue and the Normal distribution $p(x)$ from which we draw samples in red. The Monte Carlo method provides a good approximation (green) to the real solution (black).

Wrapping Up

In the toy examples above it was easy to sample from $p(x)$. However, for practical problems the distributions we want to sample from are often complex and operate in many dimensions. For these problems more clever sampling methods have to be used. Such sampling methods include Inverse Transform Sampling, Rejection Sampling, Importance Sampling, and Markov Chain Monte Carlo methods such as the Metropolis Hasting algorithm and the Gibbs sampler, each of which I plan to cover in separate posts.

Posted in Sampling Methods, Uncategorized

Tags: Monte Carlo Approximation, Monte Carlo Integration, Monte Carlo Optimization 4 Comments

The Clever Machine

Topics in Computational Neuroscience & Machine Learning

MCMC: The Metropolis-Hastings Sampler

OCT 20
Posted by **dustinstansbury**

In an earlier post (https://theclevermachine.wordpress.com/2012/10/05/mcmc-the-metropolis-sampler/) we discussed how the Metropolis sampling algorithm can draw samples from a complex and/or unnormalized target probability distributions using a Markov chain. The Metropolis algorithm first proposes a possible new state x^* in the Markov chain, based on a previous state $x^{(t-1)}$, according to the proposal distribution $q(x^*|x^{(t-1)})$. The algorithm accepts or rejects the proposed state based on the density of the the target distribution $p(x)$ evaluated at x^*. (If any of this Markov-speak is gibberish to the reader, please refer to the previous posts on Markov Chains (https://theclevermachine.wordpress.com/2012/09/24/a-brief-introduction-to-markov-chains/), MCMC, and the Metropolis Algorithm (https://theclevermachine.wordpress.com/2012/10/05/mcmc-the-metropolis-sampler/) for some clarification).

One constraint of the Metropolis sampler is that the proposal distribution $q(x^*|x^{(t-1)})$ must be symmetric. The constraint originates from using a Markov Chain to draw samples: a necessary condition for drawing from a Markov chain's stationary distribution is that at any given point in time t, the probability of moving from $x^{(t-1)} \rightarrow x^{(t)}$ must be equal to the probability of moving from $x^{(t-1)} \rightarrow x^{(t)}$, a condition known as ***reversibility*** or ***detailed balance***. However, a symmetric proposal distribution may be ill-fit for many problems, like when we want to sample from distributions that are bounded on semi infinite intervals (e.g. $[0, \infty)$).

In order to be able to use an asymmetric proposal distributions, the Metropolis-Hastings algorithm implements an additional correction factor c, defined from the proposal distribution as

$$c = \frac{q(x^{(t-1)}|x^*)}{q(x^*|x^{(t-1)})}$$

The correction factor adjusts the transition operator to ensure that the probability of moving from $x^{(t-1)} \rightarrow x^{(t)}$ is equal to the probability of moving from $x^{(t-1)} \rightarrow x^{(t)}$, no matter the proposal distribution.

The Metropolis-Hastings algorithm is implemented with essentially the same procedure as the Metropolis sampler, except that the correction factor is used in the evaluation of acceptance probability α. Specifically, to draw M samples using the Metropolis-Hastings sampler:

1. set t = 0

2. generate an initial state $x^{(0)} \sim \pi^{(0)}$
3. repeat until $t = M$

set $t = t + 1$

generate a proposal state x^* from $q(x|x^{(t-1)})$

calculate the proposal correction factor $c = \frac{q(x^{(t-1)}|x^*)}{q(x^*|x^{(t-1)})}$

calculate the acceptance probability $\alpha = \min\left(1, \frac{p(x^*)}{p(x^{(t-1)})} \times c\right)$

draw a random number u from $\text{Unif}(0, 1)$

if $u \leq \alpha$ accept the proposal state x^* and set $x^{(t)} = x^*$

else set $x^{(t)} = x^{(t-1)}$

Many consider the Metropolis-Hastings algorithm to be a generalization of the Metropolis algorithm. This is because when the proposal distribution is symmetric, the correction factor is equal to one, giving the transition operator for the Metropolis sampler.

Example: Sampling from a Bayesian posterior with improper prior

For a number of applications, including regression and density estimation, it is usually necessary to determine a set of parameters θ to an assumed model $p(y|\theta)$ such that the model can best account for some observed data y. The model function $p(y|\theta)$ is often referred to as the likelihood function. In Bayesian methods there is often an explicit prior distribution $p(\theta)$ that is placed on the model parameters and controls the values that the parameters can take.

The parameters are determined based on the posterior distribution $p(\theta|y)$, which is a probability distribution over the possible parameters based on the observed data. The posterior can be determined using Bayes' theorem:

$$p(\theta|y) = \frac{p(y|\theta)p(\theta)}{p(y)}$$

where, $p(y)$ is a normalization constant that is often quite difficult to determine explicitly, as it involves computing sums over every possible value that the parameters and y can take.

Let's say that we assume the following model (likelihood function):

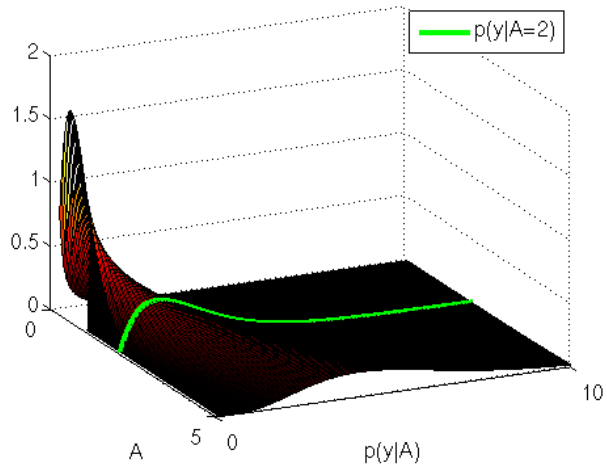
$p(y|\theta) = \text{Gamma}(y; A, B)$, where

$\text{Gamma}(y; A, B) = \frac{B^A}{\Gamma(A)} y^{A-1} e^{-By}$, where

$\Gamma()$ is the gamma function (http://en.wikipedia.org/wiki/Gamma_function). Thus, the model parameters are

$$\theta = [A, B]$$

The parameter A controls the shape of the distribution, and B controls the scale. The likelihood surface for $B = 1$, and a number of values of A ranging from zero to five are shown below.



(https://theclevermachine.files.wordpress.com/2012/10/metropolishastings-gamma-nonstandard-prior-likelihood5.png)
Likelihood surface and conditional probability p(y | A=2,B=1) in green

The conditional distribution $p(y|A = 2, B = 1)$ is plotted in green along the likelihood surface. You can verify this is a valid conditional in MATLAB with the following command:

```
1 | plot(0:.1:10,gampdf(0:.1:10,4,1)); % GAMMA(4,1)
```

Now, let's assume the following priors on the model parameters:

$$p(B = 1) = 1$$

and

$$p(A) = \sin(\pi A)^2$$

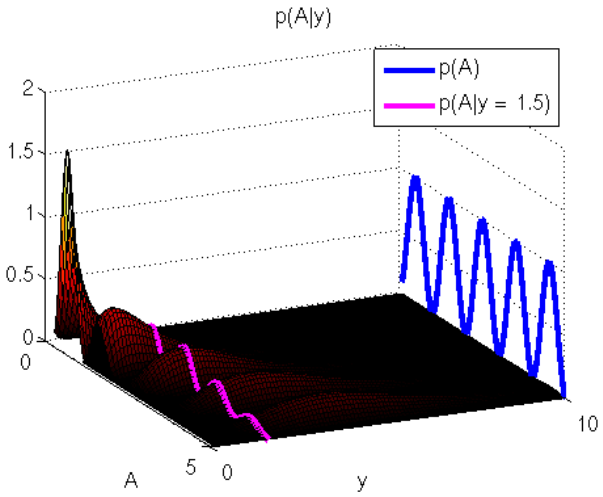
The first prior states that B only takes a single value (i.e. 1), therefore we can treat it as a constant. The second (rather non-conventional) prior states that the probability of A varies as a sinusoidal function. (Note that both of these prior distributions are called ***improper priors*** because they do not integrate to one). Because B is constant, we only need to estimate the value of A.

It turns out that even though the normalization constant $p(y)$ may be difficult to compute, we can sample from $p(A|y)$ without knowing $p(x)$ using the Metropolis-Hastings algorithm. In particular, we can ignore the normalization constant $p(x)$ and sample from the unnormalized posterior:

$$p(A|y) \propto p(y|A)p(A)$$

The surface of the (unnormalized) posterior for y ranging from zero to ten are shown below. The prior $p(A)$ is displayed in blue on the right of the plot. Let's say that we have a datapoint $y = 1.5$ and

would like to estimate the posterior distribution $p(A|y = 1.5)$ using the Metropolis-Hastings algorithm. This particular target distribution is plotted in magenta in the plot below.

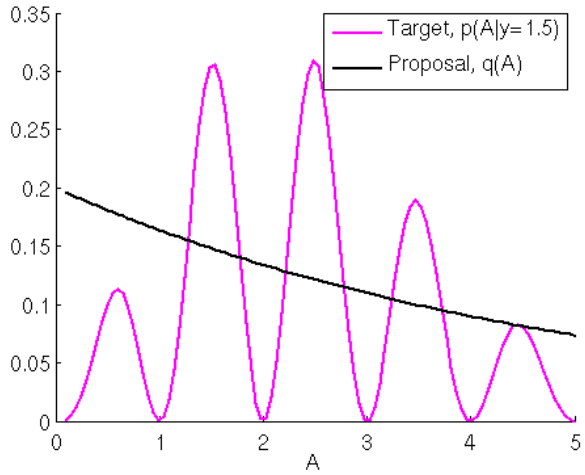


(https://theclevermachine.files.wordpress.com/2012/10/metropolishastings-gamma-nonstandard-prior-posterior4.png)
Posterior surface, prior distribution (blue), and target distribution (pink)

Using a symmetric proposal distribution like the Normal distribution is not efficient for sampling from $p(A|y = 1.5)$ due to the fact that the posterior only has support on the real positive numbers $A \in [0, \infty)$. An asymmetric proposal distribution with the same support, would provide a better coverage of the posterior. One distribution that operates on the positive real numbers is the exponential distribution.

$$q(A) = \text{Exp}(\mu) = \mu e^{-\mu A},$$

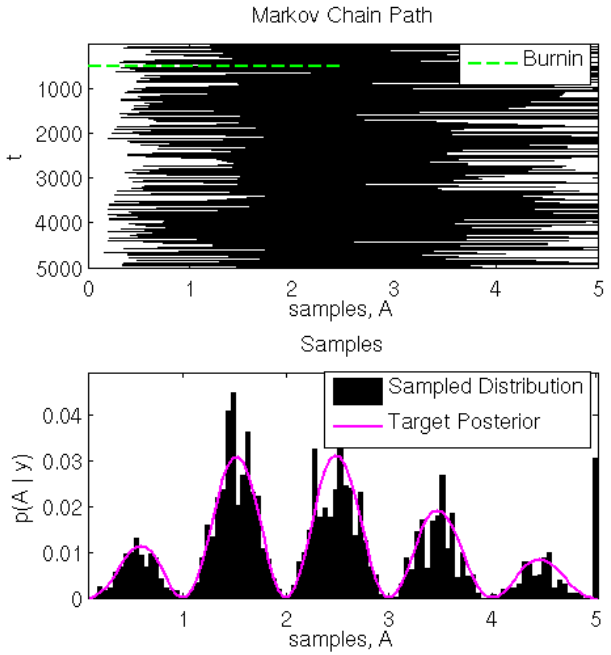
This distribution is parameterized by a single variable μ that controls the scale and location of the distribution probability mass. The target posterior and a proposal distribution (for $\mu = 5$) are shown in the plot below.



(https://theclevermachine.files.wordpress.com/2012/10/metropolishastings-gamma-nonstandard-prior-target-proposal2.png)

Target posterior p(A | y) and proposal distribution q(A)

We see that the proposal has a fairly good coverage of the posterior distribution. We run the Metropolis-Hastings sampler in the block of MATLAB code at the bottom. The Markov chain path and the resulting samples are shown in plot below.



(https://theclevermachine.files.wordpress.com/2012/10/metropolishastings-gamma-nonstandard-prior-samples2.png)

Metropolis-Hastings Markov chain and samples

As an aside, note that the proposal distribution for this sampler does not depend on past samples, but only on the parameter μ (see line 88 in the MATLAB code below). Each proposal states x^* is drawn independently of the previous state. Therefore this is an example of an ***independence sampler***, a specific type of Metropolis-Hastings sampling algorithm. Independence samplers are notorious for being either very good or very poor sampling routines. The quality of the routine depends on the choice of the proposal distribution, and its coverage of the target distribution. Identifying such a proposal distribution is often difficult in practice.

The MATLAB code for running the Metropolis-Hastings sampler is below. Use the copy icon in the upper right of the code block to copy it to your clipboard. Paste in a MATLAB terminal to output the figures above.

```
% METROPOLIS-HASTINGS BAYESIAN POSTERIOR
rand('seed',12345)

% PRIOR OVER SCALE PARAMETERS
B = 1;

% DEFINE LIKELIHOOD
likelihood = inline('(B.^A/gamma(A)).*y.^(A-1).*exp(-(B.*y))','y','A','B')

% CALCULATE AND VISUALIZE THE LIKELIHOOD SURFACE
yy = linspace(0,10,100);
AA = linspace(0.1,5,100);
likeSurf = zeros(numel(yy),numel(AA));
for iA = 1:numel(AA); likeSurf(:,iA)=likelihood(yy(:),AA(iA),B); end;

figure;
surf(likeSurf); ylabel('p(y|A)'); xlabel('A'); colormap hot

% DISPLAY CONDITIONAL AT A = 2
hold on; ly = plot3(ones(1,numel(AA))*40,1:100,likeSurf(:,40),'g','linewid
xlim([0 100]); ylim([0 100]);  axis normal
set(gca,'XTick',[0,100]); set(gca,'XTickLabel',[0 5]);
set(gca,'YTick',[0,100]); set(gca,'YTickLabel',[0 10]);
view(65,25)
legend(ly,'p(y|A=2)','Location','Northeast');
hold off;
title('p(y|A)');

% DEFINE PRIOR OVER SHAPE PARAMETERS
prior = inline('sin(pi*A).^2','A');

% DEFINE THE POSTERIOR
p = inline('(B.^A/gamma(A)).*y.^(A-1).*exp(-(B.*y)).*sin(pi*A).^2','y','A'

% CALCULATE AND DISPLAY THE POSTERIOR SURFACE
postSurf = zeros(size(likeSurf));
for iA = 1:numel(AA); postSurf(:,iA)=p(yy(:),AA(iA),B); end;

figure
surf(postSurf); ylabel('y'); xlabel('A'); colormap hot

% DISPLAY THE PRIOR
hold on; pA = plot3(1:100,ones(1,numel(AA))*100,prior(AA),'b','linewidth',

% SAMPLE FROM p(A | y = 1.5)
y = 1.5;
target = postSurf(16,:);

% DISPLAY POSTERIOR
psA = plot3(1:100, ones(1,numel(AA))*16,postSurf(16,:),'m','linewidth',3)
xlim([0 100]); ylim([0 100]);  axis normal
set(gca,'XTick',[0,100]); set(gca,'XTickLabel',[0 5]);
set(gca,'YTick',[0,100]); set(gca,'YTickLabel',[0 10]);
view(65,25)
legend([pA,psA],{'p(A)','p(A|y = 1.5)'},'Location','Northeast');
hold off
title('p(A|y)');

% INITIALIZE THE METROPOLIS-HASTINGS SAMPLER
```



```
% DEFINE PROPOSAL DENSITY
q = inline('exppdf(x,mu)','x','mu');

% MEAN FOR PROPOSAL DENSITY
mu = 5;

% DISPLAY TARGET AND PROPOSAL
figure; hold on;
th = plot(AA,target,'m','Linewidth',2);
qh = plot(AA,q(AA,mu),'k','Linewidth',2)
legend([th,qh],{'Target, p(A)','Proposal, q(A)'});
xlabel('A');

% SOME CONSTANTS
nSamples = 5000;
burnIn = 500;
minn = 0.1; maxx = 5;

% INTIIALZE SAMPLER
x = zeros(1 ,nSamples);
x(1) = mu;
t = 1;

% RUN METROPOLIS-HASTINGS SAMPLER
while t < nSamples
    t = t+1;

    % SAMPLE FROM PROPOSAL
    xStar = exprnd(mu);

    % CORRECTION FACTOR
    c = q(x(t-1),mu)/q(xStar,mu);

    % CALCULATE THE (CORRECTED) ACCEPTANCE RATIO
    alpha = min([1, p(y,xStar,B)/p(y,x(t-1),B)*c]);

    % ACCEPT OR REJECT?
    u = rand;
    if u < alpha
        x(t) = xStar;
    else
        x(t) = x(t-1);
    end
end

% DISPLAY MARKOV CHAIN
figure;
subplot(211);
stairs(x(1:t),1:t, 'k');
hold on;
hb = plot([0 maxx/2],[burnIn burnIn],'g--','Linewidth',2)
ylabel('t'); xlabel('samples, A');
set(gca , 'YDir', 'reverse');
ylim([0 t])
axis tight;
xlim([0 maxx]);
title('Markov Chain Path');
legend(hb,'Burnin');
```

```
% DISPLAY SAMPLES
subplot(212);
nBins = 100;
sampleBins = linspace(minn,maxx,nBins);
counts = hist(x(burnIn:end), sampleBins);
bar(sampleBins, counts/sum(counts), 'k');
xlabel('samples, A' ); ylabel( 'p(A | y)' );
title('Samples');
xlim([0 10])

% OVERLAY TARGET DISTRIBUTION
hold on;
plot(AA, target/sum(target) , 'm-', 'LineWidth', 2);
legend('Sampled Distribution',sprintf('Target Posterior'))
axis tight
```

Wrapping Up

Here we explored how the Metorpolis-Hastings sampling algorithm can be used to generalize the Metropolis algorithm in order to sample from complex (an unnormalized) probability distributions using asymmetric proposal distributions. One shortcoming of the Metropolis-Hastings algorithm is that not all of the proposed samples are accepted, wasting valuable computational resources. This becomes even more of an issue for sampling distributions in higher dimensions. This is where Gibbs sampling comes in. We'll see in a later post that Gibbs sampling can be used to keep all proposal states in the Markov chain by taking advantage of conditional probabilities.

About dustinstansbury

I recently received my PhD from UC Berkeley where I studied computational neuroscience and machine learning.

View all posts by dustinstansbury »

Posted on October 20, 2012, in Algorithms, Sampling Methods, Simulations, Statistics and tagged Bayesian likelihood, Bayesian methods, Bayesian posterior, improper prior, Independence sampler, MCMC, Metropolis sampling, Metropolis-Hastings Sampling. Bookmark the permalink. 5 Comments.

Leave a comment

Trackbacks 2

Comments 3

markovmagic | August 28, 2013 at 6:15 pm
Reblogged this on Machine Learning Magic.

Michael Cheng | September 12, 2017 at 12:00 am
I appreciate your series. The best tutorial I've ever seen!
I'm not sure, around line 14-15, it seems that the second $x^{(t-1)} \to x^{(t)}$ should be $x^{(t)} \to x^{(t-1)}$.

John | May 27, 2018 at 4:00 pm
One of the two states expressions in the sentence here under should be flipped, small typo 🙂
Thanks a lot for those good explanations!

«The correction factor adjusts the transition operator to ensure that the probability of moving from x^{(t-1)} \rightarrow x^{(t)} is equal to the probability of moving from x^{(t-1)} \rightarrow x^{(t)}, no matter the proposal distribution.»

1. Pingback: **MCMC: Multivariate Distributions, Block-wise, & Component-wise Updates « The Clever Machine**
2. Pingback: **A Gentle Introduction to Markov Chain Monte Carlo (MCMC) « The Clever Machine**

The Clever Machine

Topics in Computational Neuroscience & Machine Learning

MCMC: The Metropolis Sampler

OCT 5
Posted by **dustinstansbury**

As discussed in an earlier post (https://theclevermachine.wordpress.com/2012/09/24/a-brief-introduction-to-markov-chains/), we can use a Markov chain to sample from some ***target probability distribution*** $p(x)$ from which drawing samples directly is difficult. To do so, it is necessary to design a transition operator for the Markov chain which makes the chain's stationary distribution match the target distribution. The Metropolis sampling algorithm (and the more general Metropolis-Hastings sampling algorithm) uses simple heuristics to implement such a transition operator.

Metropolis Sampling

Starting from some random initial state $x^{(0)} \sim \pi^{(0)}$, the algorithm first draws a possible sample x^* from a ***proposal distribution*** $q(x|x^{(t-1)})$. Much like a conventional transition operator for a Markov chain, the proposal distribution depends only on the previous state in the chain. However, the transition operator for the Metropolis algorithm has an additional step that assesses whether or not the target distribution has a sufficiently large density near the proposed state to warrant accepting the proposed state as a sample and setting it to the next state in the chain. If the density of $p(x)$ is low near the proposed state, then it is likely (but not guaranteed) that it will be rejected. The criterion for accepting or rejecting a proposed state are defined by the following heuristics:

1. If $p(x^*) \geq p(x^{(t-1)})$, the proposed state is kept x^* as a sample and is set as the next state in the chain (i.e. move the chain's state to a location where $p(x)$ has equal or greater density).
2. If $p(x^*) < p(x^{(t-1)})$–indicating that $p(x)$ has low density near x^*–then the proposed state may still be accepted, but only randomly, and with a probability $\frac{p(x^*)}{p(x^{(t-1)})}$

These heuristics can be instantiated by calculating the ***acceptance probability*** for the proposed state.

$$\alpha = \min\left(1, \frac{p(x^*)}{p(x^{(t-1)})}\right)$$

Having the acceptance probability in hand, the transition operator for the metropolis algorithm

works like this: if a random uniform number u is less than or equal to α, then the state x^* is accepted (as in (1) above), if not, it is rejected and another state is proposed (as in (2) above). In order to collect M samples using Metropolis sampling we run the following algorithm:

1. set t = 0
2. generate an initial state $x^{(0)}$ from a prior distribution $\pi^{(0)}$ over initial states
3. repeat until $t = M$

set $t = t + 1$

generate a proposal state x^* from $q(x|x^{(t-1)})$

calculate the acceptance probability $\alpha = \min\left(1, \frac{p(x^*)}{p(x^{(t-1)})}\right)$

draw a random number u from $\text{Unif}(0, 1)$

if $u \leq \alpha$, accept the proposal and set $x^{(t)} = x^*$

else set $x^{(t)} = x^{(t-1)}$

Example: Using the Metropolis algorithm to sample from an unknown distribution

Say that we have some mysterious function

$$p(x) = (1 + x^2)^{-1}$$

from which we would like to draw samples. To do so using Metropolis sampling we need to define two things: (1) the prior distribution $\pi^{(0)}$ over the initial state of the Markov chain, and (2) the proposal distribution $q(x|x^{(t-1)})$. For this example we define:

$$\pi^{(0)} \sim \mathcal{N}(0, 1)$$

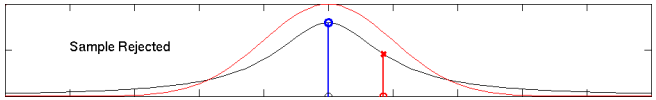
$$q(x|x^{(t-1)}) \sim \mathcal{N}(x^{(t-1)}, 1),$$

both of which are simply a Normal distribution, one centered at zero, the other centered at previous state of the chain. The following chunk of MATLAB code runs the Metropolis sampler with this proposal distribution and prior.

```
% METROPOLIS SAMPLING EXAMPLE
randn('seed',12345);

% DEFINE THE TARGET DISTRIBUTION
p = inline('(1 + x.^2).^-1','x')

% SOME CONSTANTS
nSamples = 5000;
burnIn = 500;
nDisplay = 30;
sigma = 1;
minn = -20; maxx = 20;
xx = 3*minn:.1:3*maxx;

target = p(xx);
pauseDur = .8;

% INITIALZE SAMPLER
x = zeros(1 ,nSamples);
x(1) = randn;
t = 1;

% RUN SAMPLER
while t < nSamples
    t = t+1;

    % SAMPLE FROM PROPOSAL
    xStar = normrnd(x(t-1) ,sigma);
    proposal = normpdf(xx,x(t-1),sigma);

    % CALCULATE THE ACCEPTANCE PROBABILITY
    alpha = min([1, p(xStar)/p(x(t-1))]);

    % ACCEPT OR REJECT?
    u = rand;
    if u < alpha
        x(t) = xStar;
        str = 'Accepted';
    else
        x(t) = x(t-1);
        str = 'Rejected';
    end

    % DISPLAY SAMPLING DYNAMICS
    if t < nDisplay + 1
        figure(1);
        subplot(211);
        cla
        plot(xx,target,'k');
        hold on;
        plot(xx,proposal,'r');
        line([x(t-1),x(t-1)],[0 p(x(t-1))],'color','b','linewidth',2)
        scatter(xStar,0,'ro','Linewidth',2)
        line([xStar,xStar],[0 p(xStar)],'color','r','Linewidth',2)
        plot(x(1:t),zeros(1,t),'ko')
        legend({'Target','Proposal','p(x^{(t-1)})','x^*','p(x^*)','Kept Sa
        switch str
            case 'Rejected'
```



```
                scatter(xStar,p(xStar),'rx','Linewidth',3)
            case 'Accepted'
                scatter(xStar,p(xStar),'rs','Linewidth',3)
        end
            scatter(x(t-1),p(x(t-1)),'bo','Linewidth',3)
        title(sprintf('Sample % d %s',t,str))
        xlim([minn,maxx])
        subplot(212);
        hist(x(1:t),50); colormap hot;
        xlim([minn,maxx])
        title(['Sample ',str]);
        drawnow
        pause(pauseDur);
    end
end

% DISPLAY MARKOV CHAIN
figure(1); clf
subplot(211);
stairs(x(1:t),1:t, 'k');
hold on;
hb = plot([-10 10],[burnIn burnIn],'b--')
ylabel('t'); xlabel('samples, x');
set(gca , 'YDir', 'reverse');
ylim([0 t])
axis tight;
xlim([-10 10]);
title('Markov Chain Path');
legend(hb,'Burnin');

% DISPLAY SAMPLES
subplot(212);
nBins = 200;
sampleBins = linspace(minn,maxx,nBins);
counts = hist(x(burnIn:end), sampleBins);
bar(sampleBins, counts/sum(counts), 'k');
xlabel('samples, x' ); ylabel( 'p(x)' );
title('Samples');

% OVERLAY ANALYTIC DENSITY OF STUDENT T
nu = 1;
y = tpdf(sampleBins,nu)
hold on;
plot(sampleBins, y/sum(y) , 'r-', 'LineWidth', 2);
legend('Samples',sprintf('Theoretic\nStudent''s t'))
axis tight
xlim([-10 10]);
```



(https://theclevermachine.files.wordpress.com/2012/10/metropolis2.gif)
Using the Metropolis algorithm to sample from a continuous distribution (black)

In the figure above, we visualize the first 50 iterations of the Metropolis sampler.The black curve represents the target distribution $p(x)$. The red curve that is bouncing about the x-axis is the proposal distribution $q(x|x^{(t-1)})$ (if the figure is not animated, just click on it). The vertical blue line (about

which the bouncing proposal distribution is centered) represents the quantity $p(x^{(t-1)})$, and the vertical red line represents the quantity $p(x^*)$, for a proposal state x^* sampled according to the red curve. At every iteration, if the vertical red line is longer than the blue line, then the sample x^* is accepted, and the proposal distribution becomes centered about the newly accepted sample. If the blue line is longer, the sample is randomly rejected or accepted.

But why randomly keep "bad" proposal samples? It turns out that doing this allows the Markov chain to every-so-often visit states of low probability under the target distribution. This is a desirable property if we want the chain to adequately sample the entire target distribution, including any tails.

An attractive property of the Metropolis algorithm is that the target distribution $p(x)$ does not have to be a properly normalized probability distribution. This is due to the fact that the acceptance probability is based on the ratio of two values of the target distribution. I'll show you what I mean. If $p(x)$ is an unnormalized distribution and

$$p^*(x) = \frac{p(x)}{Z}$$

is a properly normalized probability distribution with normalizing constant Z, then

$$p(x) = Zp^*(x)$$

and a ratio like that used in calculating the acceptance probability α is

$$\frac{p(a)}{p(b)} = \frac{Zp^*(a)}{Zp^*(b)} = \frac{p^*(a)}{p^*(b)}$$

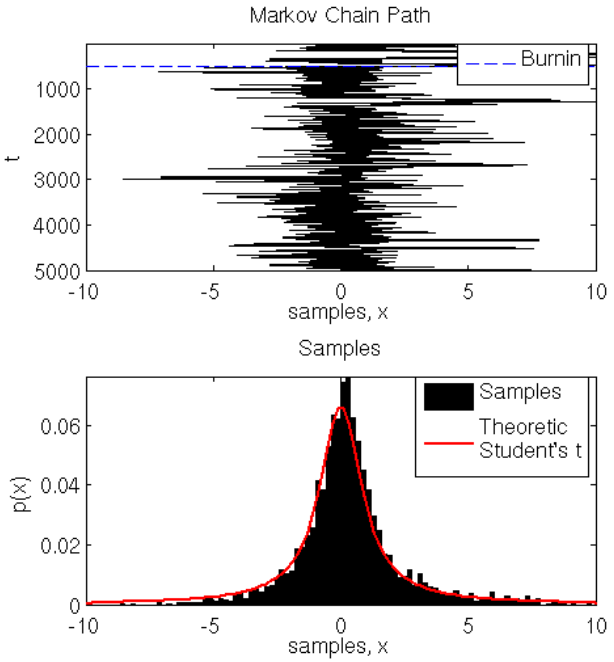
The normalizing constants Z cancel! This attractive property is quite useful in the context of Bayesian methods, where determining the normalizing constant for a distribution may be impractical to calculate directly. This property is demonstrated in current example. It turns out that the "mystery" distribution that we sampled from using the Metropolis algorithm is an unnormalized form of the Student's-t distribution with one degree of freedom. Comparing $p(x)$ to the definition of the definition Student's-t

$$Student(x,\nu) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)}\left(1+\frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}} = \frac{(1+x^2)^{-1}}{Z} = \frac{p(x)}{Z}$$

we see that $p(x)$ is a Student's-t distribution with degrees of freedom $\nu = 1$, but missing the normalizing constant

$$Z = \left(\frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)}\right)^{-1}$$

Below is additional output from the code above showing that the samples from Metropolis sampler draws samples that follow a *normalized* Student's-t distribution, even though $p(x)$ is not normalized.



(https://theclevermachine.files.wordpress.com/2012/10/metropolisstudentst2.png)

Metropolis samples from an unnormalized t-distribution follow the normalized distribution

The upper plot shows the progression of the Markov chain's progression from state $x^{(0)}$ (top) to state $x^{(5000)}$ (bottom). The burn in period for this chain was chosen to be 500 transitions, and is indicated by the dashed blue line (for more on burnin see this previous post (https://theclevermachine.wordpress.com/2012/09/24/a-brief-introduction-to-markov-chains/)).

The bottom plot shows samples from the Markov chain in black (with burn in samples removed). The theoretical curve for the Student's-t with one degree of freedom is overlayed in red. We see that the states kept by the Metropolis sampler transition operator sample from values that follow the Student's-t, even though the function $p(x)$ used in the transition operator was not a properly normalized probability distribution.

Reversibility of the transition operator

It turns out that there is a theoretical constraint on the Markov chain the transition operator in order for it settle into a stationary distribution (i.e. a target distribution we care about). The constraint states that the probability of the transition $x^{(t)} \rightarrow x^{(t+1)}$ must be equal to the probability of the reverse transition $x^{(t+1)} \rightarrow x^{(t)}$. This reversibility property is often referred to as ***detailed balance***. Using the Metropolis algorithm transition operator, reversibility is assured if the proposal distribution

$q(x|x^{(t-1)})$ is symmetric. Such symmetric proposal distributions are the Normal, Cauchy, Student's-t, and Uniform distributions.

However, using a symmetric proposal distribution may not be reasonable to adequately or efficiently sample all possible target distributions. For instance if a target distribution is bounded on the positive numbers $0 < x \leq \infty$, we would like to use a proposal distribution that has the same support, and will thus be assymetric. This is where the ***Metropolis-Hastings*** sampling algorithm comes in. We will discuss in a later post how the Metropolis-Hastings sampler uses a simple change to the calculation of the acceptance probability which allows us to use non-symmetric proposal distributions.

About dustinstansbury

I recently received my PhD from UC Berkeley where I studied computational neuroscience and machine learning.

View all posts by dustinstansbury »

Posted on October 5, 2012, in Algorithms, Sampling Methods, Statistics and tagged Acceptance Probability, Detailed Balance, Markov Chain Monte Carlo, MCMC, Metropolis sampling, Metropolis-Hastings Sampling, Proposal Distribution, Reversibility, Target Distribution. Bookmark the permalink. 3 Comments.

- ## Leave a comment
- ## Trackbacks 2
- ## Comments 1

Aparna Sen | July 21, 2018 at 10:01 pm
Great post. Unpretentious and easy to follow.

1. Pingback: **MCMC: The Metropolis-Hastings Sampler « The Clever Machine**
2. Pingback: **A Gentle Introduction to Markov Chain Monte Carlo (MCMC) « The Clever Machine**

The Clever Machine

Topics in Computational Neuroscience & Machine Learning

MCMC: Multivariate Distributions, Block-wise, & Component-wise Updates

NOV 4
Posted by **dustinstansbury**

In the previous posts on MCMC methods, we focused on how to sample from univariate target distributions. This was done mainly to give the reader some intuition about MCMC implementations with fairly tangible examples that can be visualized. However, MCMC can easily be extended to sample multivariate distributions.

In this post we will discuss two flavors of MCMC update procedure for sampling distributions in multiple dimensions: block-wise, and component-wise update procedures. We will show how these two different procedures can give rise to different implementations of the Metropolis-Hastings sampler to solve the same problem.

Block-wise Sampling

The first approach for performing multidimensional sampling is to use ***block-wise updates***. In this approach the proposal distribution $q(\mathbf{x})$ has the same dimensionality as the target distribution $p(\mathbf{x})$. Specifically, if $p(\mathbf{x})$ is a distribution over D variables, ie. $\mathbf{x} = (x_1, x_2, \ldots, x_D)$, then we must design a proposal distribution that is also a distribution involving D variables. We then accept or reject a proposed state $\mathbf{x}^*$ sampled from the proposal distribution $q(\mathbf{x})$ in exactly the same way as for the univariate Metropolis-Hastings algorithm (https://theclevermachine.wordpress.com/2012/10/20/mcmc-the-metropolis-hastings-sampler/). To generate M multivariate samples we perform the following block-wise sampling procedure:

1. set $t = 0$
2. generate an initial state $\mathbf{x}^{(0)} \sim \pi^{(0)}$
3. repeat until $t = M$

set $t = t + 1$

generate a proposal state $\mathbf{x}^*$ from $q(\mathbf{x}|\mathbf{x}^{(t-1)})$

calculate the proposal correction factor $c = \frac{q(\mathbf{x}^{(t-1)}|\mathbf{x}^*)}{q(\mathbf{x}^*|\mathbf{x}^{(t-1)})}$

calculate the acceptance probability $\alpha = \min\left(1, \frac{p(\mathbf{x}^*)}{p(\mathbf{x}^{(t-1)})} \times c\right)$

draw a random number u from $\mathrm{Unif}(0, 1)$

if $u \leq \alpha$ accept the proposal state $\mathbf{x}^*$ and set $\mathbf{x}^{(t)} = \mathbf{x}^*$

else set $\mathbf{x}^{(t)} = \mathbf{x}^{(t-1)}$

Let's take a look at the block-wise sampling routine in action.

Example 1: Block-wise Metropolis-Hastings for sampling of bivariate Normal distribution

In this example we use block-wise Metropolis-Hastings algorithm to sample from a bivariate (i.e. $D = 2$) Normal distribution:

$$p(\mathbf{x}) = \mathcal{N}(\mu, \Sigma)$$

with mean

$$\mu = [0, 0]$$

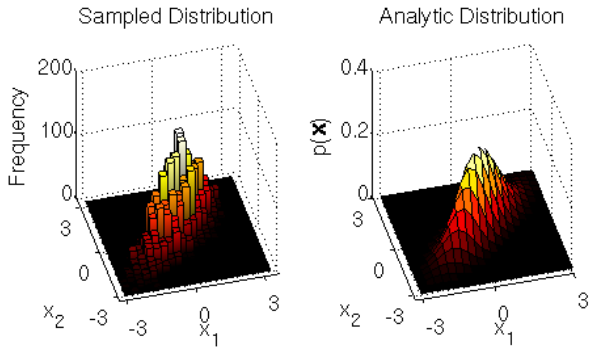
and covariance

$$\Sigma = \begin{bmatrix} 1 & 0.8 \\ 0.8 & 1 \end{bmatrix}$$

Usually the target distribution $p(\mathbf{x})$ will have a complex mathematical form (http://en.wikipedia.org/wiki/Multivariate_normal_distribution), but for this example we'll circumvent that by using MATLAB's built-in function mvnpdf to evaluate $p(\mathbf{x})$. For our proposal distribution, $q(\mathbf{x})$, let's use a circular Normal centered at the the previous state/sample of the Markov chain/sampler, i.e:

$$q(\mathbf{x}|\mathbf{x}^{(t-1)}) \sim \mathcal{N}(\mathbf{x}^{(t-1)}, \mathbf{I}),$$

where $\mathbf{I}$ is a 2-D identity matrix, giving the proposal distribution unit variance along both dimensions x_1 and x_2, and zero covariance. You can find an MATLAB implementation of the block-wise sampler at the end of the section. The display of the samples and the target distribution output by the sampler implementation are shown below:



(https://theclevermachine.files.wordpress.com/2012/11/metropolishastings-blockwise-samples1.png)

Samples drawn from block-wise Metropolis-Hastings sampler

We can see from the output that the block-wise sampler does a good job of drawing samples from the target distribution.

Note that our proposal distribution in this example is symmetric, therefore it was not necessary to calculate the correction factor c per se. This means that this Metropolis-Hastings implementation is identical to the simpler Metropolis sampler.

```
+ expand source
```

Component-wise Sampling

A problem with block-wise updates, particularly when the number of dimensions D becomes large, is that finding a suitable proposal distribution is difficult. This leads to a large proportion of the samples being rejected. One way to remedy this is to simply loop over the the D dimensions of $\mathbf{x}$ in sequence, sampling each dimension independently from the others. This is what is known as using ***component-wise updates***. Note that now the proposal distribution $q(x)$ is univariate, working only in one dimension, namely the current dimension that we are trying to sample. The component-wise Metropolis-Hastings algorithm is outlined below.

1. set $t = 0$
2. generate an initial state $\mathbf{x}^{(0)} \sim \pi^{(0)}$
3. repeat until $t = M$

set $t = t + 1$

for each dimension $i = 1..D$

generate a proposal state x_i^* from $q(x_i|x_i^{(t-1)})$

calculate the proposal correction factor $c = \frac{q(x_i^{(t-1)})|x_i^*)}{q(x_i^*|x_i^{(t-1)})}$

calculate the acceptance probability $\alpha = \min\left(1, \frac{p(x_i^*,\mathbf{x}_j^{(t-1)})}{p(x_i^{(t-1)},\mathbf{x}_j^{(t-1)})} \times c\right)$

draw a random number u from $\text{Unif}(0, 1)$

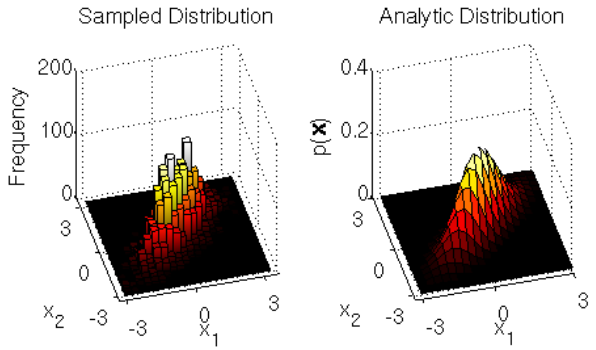
if $u \leq \alpha$ accept the proposal state x_i^* and set $x_i^{(t)} = x_i^*$

else set $x_i^{(t)} = x_i^{(t-1)}$

Note that in the component-wise implementation a sample for the i-th dimension is proposed, then accepted or rejected while all other dimensions ($j \neq i$) are held fixed. We then move on to the next ($(i+1)$-th) dimension and repeat the process while holding all other variables ($j \neq (i+1)$) fixed. In each successive step we are using updated values for the dimensions that have occurred since increasing $(t-1) \rightarrow t$.

Example 2: Component-wise Metropolis-Hastings for sampling of bivariate Normal distribution

In this example we draw samples from the same bivariate Normal target distribution described in Example 1, but using component-wise updates. Therefore $p(x)$ is the same, however, the proposal distribution $q(x)$ is now a univariate Normal distribution with unit unit variance in the direction of the i-th dimension to be sampled. The MATLAB implementation of the component-wise sampler is at the end of the section. The samples and comparison to the analytic target distribution are shown below.



(https://theclevermachine.files.wordpress.com/2012/11/metropolishastings-blockwise-cw.png)
Samples drawn from component-wise Metropolis-Hastings algorithm compared to target distribution

Again, we see that we get a good characterization of the bivariate target distribution.

```
+ expand source
```

Wrapping Up

Here we saw how we can use block- and component-wise updates to derive two different implementations of the Metropolis-Hastings algorithm. In the next post we will use component-wise updates introduced above to motivate the Gibbs sampler, which is often used to increase the efficiency of sampling well-defined probability multivariate distributions.

About dustinstansbury

I recently received my PhD from UC Berkeley where I studied computational neuroscience and machine learning.

View all posts by dustinstansbury »

Posted on November 4, 2012, in Algorithms, Sampling, Sampling Methods, Simulations, Statistics and tagged Bivariate Gaussian, Block-wise Updates, Component-wise Updates, Gibbs Sampler,

MCMC, Metropolis-Hastings Sampling, Multivariate Sampling Methods. Bookmark the permalink. 8 Comments.

- **Leave a comment**
- **Trackbacks 2**
- **Comments 6**

A | April 14, 2014 at 7:49 am
I guess the location of the xStart in the p([xStar xCurrent(dims~=iD)]) array, should vary with the dimension that you are updating.

> ***dustinstansbury*** | September 19, 2014 at 3:44 pm
> Thanks for catching the typo A. You're correct the proposal should vary with each dimension. The code has been updated.

Anon | May 20, 2014 at 3:00 pm
pratio = p([xStar xCurrent(dims~=iD)])/ …

In your component-wise MH implementation, it seems that xStar is always in the first position even if you are dealing with the 2nd dimension. xStar should probably shift when other dimensions are considered. Please clarify. Very nice tutorial by the way…

> ***dustinstansbury*** | September 19, 2014 at 3:45 pm
> You're correct Anon, the proposal should consider each dimenion. I've fixed the typo/bug.

rblilja | December 27, 2016 at 2:09 am
Thank you for a fantastic site. Incredible work. May I ask how the acceptance probability shall be calculated if I have more than two dimensions? For an instance, if I have four states, is the following reasoning correct for e.g. i = 1, p(xi,xj) = p(x1,x0)*p(x1,x2)*p(x1,x3)? And by taking the log-probability I can sum instead of multiplying? Thank you!

Hugo S. | January 18, 2018 at 6:22 am
Thank you very much for such nice and well explained tutorial.

1. Pingback: **MCMC: The Gibbs Sampler « The Clever Machine**
2. Pingback: **MCMC: The Gibbs Sampler, simple example w/ Matlab code | Victor Fang's Computing Space**

The Clever Machine

Topics in Computational Neuroscience & Machine Learning

Rejection Sampling

SEP 10
Posted by **dustinstansbury**

Suppose that we want to sample from a distribution $f(x)$ that is difficult or impossible to sample from directly, but instead have a simpler distribution $q(x)$ from which sampling is easy. The idea behind Rejection sampling (aka Acceptance-rejection sampling) is to sample from $q(x)$ and apply some rejection/acceptance criterion such that the samples that are accepted are distributed according to $f(x)$.

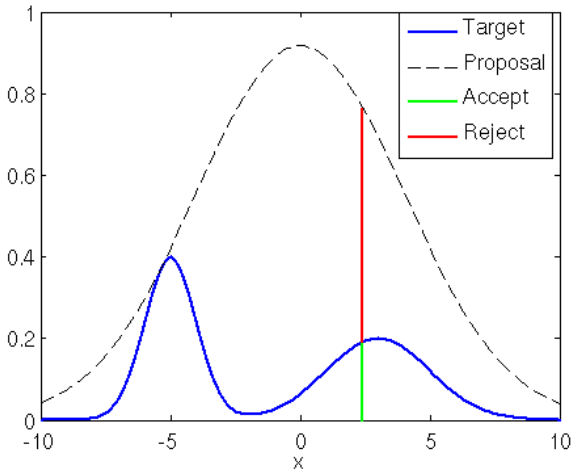
Envelope distribution and rejection criterion

In order to be able to reject samples from $q(x)$ such that they are sampled from $f(x)$, $q(x)$ must "cover" or envelop the distribution $f(x)$. This is generally done by choosing a constant $c > 1$ such that $cq(x) > f(x)$ for all x. For this reason $cq(x)$ is often called the *envelope distribution*. A common criterion for accepting samples from $x \sim q(x)$ is based on the ratio of the target distribution to that of the envelope distribution. The samples are accepted if

$$\frac{f(x)}{cq(x)} > u$$

where $u \sim Unif(0,1)$, and rejected otherwise. If the ratio is close to one, then $f(x)$ must have a large amount of probability mass around x and that sample should be more likely accepted. If the ratio is small, then it means that $f(x)$ has low probability mass around x and we should be less likely to accept the sample. This criterion is demonstrated in the chunk of MATLAB code and the resulting figure below:


```
rand('seed',12345);
x = -10:.1:10;
% CREATE A "COMPLEX DISTRIBUTION" f(x) AS A MIXTURE OF TWO NORMAL
% DISTRIBUTIONS
f = inline('normpdf(x,3,2) + normpdf(x,-5,1)','x');
t = plot(x,f(x),'b','linewidth',2); hold on;

```



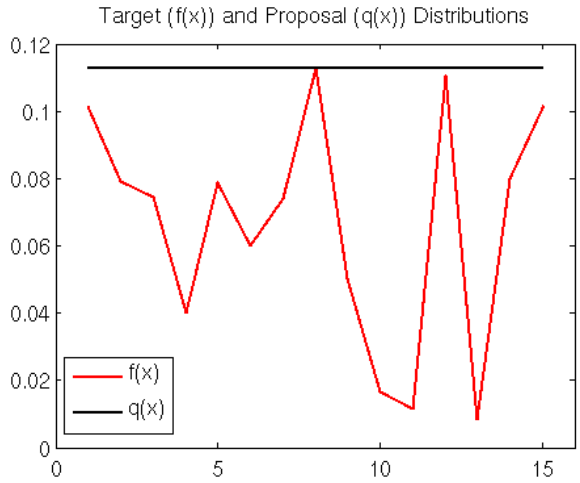
(https://theclevermachine.files.wordpress.com/2012/09/rejectionsamplingcriterion.png)
Rejection Sampling with a Normal proposal distribution

Here a zero-mean Normal distribution is used as the proposal distribution. This distribution is scaled by a factor $c = 9.2$, determined from $f(x)$ and $q(x)$ to ensure that the proposal distribution covers $f(x)$. We then sample from $q(x)$, and compare the proportion of $cq(x)$ occupied by $f(x)$. If we compare this proportion to a random number sampled from $Unif(0,1)$ (i.e. the criterion outlined above), then we would accept this sample with probability proportional to the length of the green line segment and reject the sample with probability proportional to the length of the red line

segment.

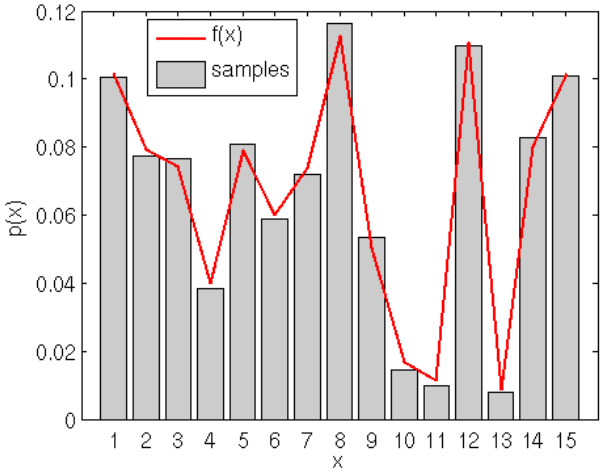
Rejection sampling of a random discrete distribution

This next example shows how rejection sampling can be used to sample from any arbitrary distribution, continuous or not, and with or without an analytic probability density function.



(https://theclevermachine.files.wordpress.com/2012/09/rejectionsamplingtargetproposal.png)
Random Discrete Target Distribution and Proposal that Bounds It.

The figure above shows a random ***discrete*** probability density function $f(x)$ generated on the interval (0,15). We will use rejection sampling as described above to sample from $f(x)$. Our proposal/envelope distribution is the uniform discrete distribution on the same interval (i.e. any of the integers from 1-15 are equally probable) multiplied by a constant c that is determined such that the maximum value of $f(x)$ lies under (or equal to) $cq(x)$.



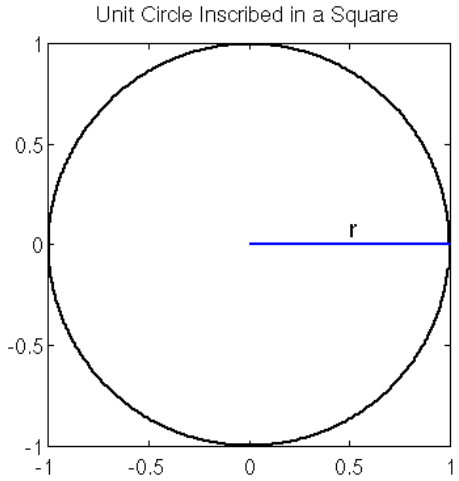
(https://theclevermachine.files.wordpress.com/2012/09/rejectionsamplingdiscrete.png)
Rejection Samples For Discrete Distribution on interval [1 15]

Plotted above is the target distribution (in red) along with the discrete samples obtained using the rejection sampling. The MATLAB code used to sample from the target distribution and display the plot above is here:

```
rand('seed',12345)
randn('seed',12345)

fLength = 15;
% CREATE A RANDOM DISTRIBUTION ON THE INTERVAL [1 fLength]
f = rand(1,fLength); f = f/sum(f);

figure; h = plot(f,'r','Linewidth',2);
hold on;
l = plot([1 fLength],[max(f) max(f)],'k','Linewidth',2);

legend([h,l],{'f(x)','q(x)'},'Location','Southwest');
xlim([0 fLength + 1])
xlabel('x');
ylabel('p(x)');
title('Target (f(x)) and Proposal (q(x)) Distributions');

% OUR PROPOSAL IS THE DISCRETE UNIFORM ON THE INTERVAL [1 fLength]
% SO OUR CONSTANT IS
c = max(f/(1/fLength));

nSamples = 10000;
i = 1;
while i < nSamples
   proposal = unidrnd(fLength);
   q = c*1/fLength; % ENVELOPE DISTRIBUTION
   if rand < f(proposal)/q
      samps(i) = proposal;
      i = i + 1;
```

Rejection sampling from the unit circle to estimate π

Though the ratio-based acceptance-rejection criterion introduced above is a common choice for drawing samples from complex distributions, it is not the only criterion we could use. For instance we could use a different set of criteria to generate some geometrically-bounded distribution. If we wanted to generate points uniformly within the unit circle (i.e. a circle centered at $(y, x) = 0$ and with radius $r = 1$), we could do so by sampling Cartesian spatial coordinates x and y uniformly from the interval (-1,1)–which samples form a square centered at (0,0)–and reject those points that lie outside of the radius $r = \sqrt{x^2 + y^2} = 1$



(https://theclevermachine.files.wordpress.com/2012/09/unitcircleinsquare2.png)
Unit Circle Inscribed in Square

Something clever that we can do with such a set of samples is to approximate the value π: Because a square that inscribes the unit circle has area:
$A_{square} = (2r)^2 = 4r^2$

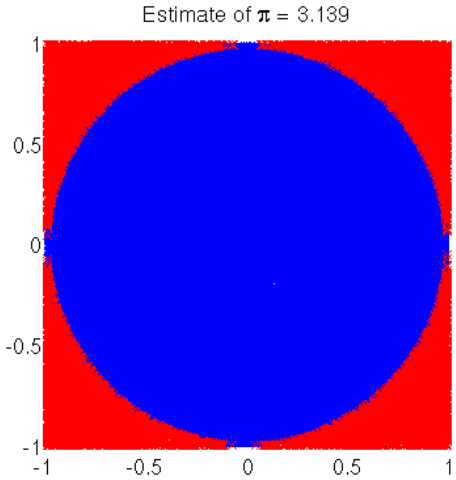
and the unit circle has the area:

$A_{circle} = \pi r^2$

We can use the ratio of their areas to approximate π:

$\pi = 4\frac{A_{circle}}{A_{square}}$

The figure below shows the rejection sampling process and the resulting estimate of π from the samples. One-hundred thousand 2D points are sampled uniformly from the interval (-1,1). Those points that lie within the unit circle are plotted as blue dots. Those points that lie outside of the unit circle are plotted as red x's. If we take four times the ratio of the area in blue to the entire area, we get a very close approximation to 3.14 for π.



(https://theclevermachine.files.wordpress.com/2012/09/rejectionsamplingpi.png)
Rejection Criterion

The MATLAB code used to generate the example figures is below:


```
% DISPLAY A CIRCLE INSCRIBED IN A SQUARE

figure;
a = 0:.01:2*pi;
x = cos(a); y = sin(a);
hold on
plot(x,y,'k','Linewidth',2)

t = text(0.5, 0.05,'r');
l = line([0 1],[0 0],'Linewidth',2);
axis equal
box on
xlim([-1 1])
ylim([-1 1])
title('Unit Circle Inscribed in a Square')

pause;
rand('seed',12345)
randn('seed',12345)
delete(l); delete(t);

% DRAW SAMPLES FROM PROPOSAL DISTRIBUTION
samples = 2*rand(2,100000) - 1;

% REJECTION
reject = sum(samples.^2) > 1;

% DISPLAY REJECTION CRITERION
scatter(samples(1,~reject),samples(2,~reject),'b.')
scatter(samples(1,reject),samples(2,reject),'rx')
hold off
```

Wrapping Up

Rejection sampling is a simple way to generate samples from complex distributions. However, Rejection sampling also has a number of weaknesses:

- Finding a proposal distribution that can cover the support of the target distribution is a non-trivial task.
- Additionally, as the dimensionality of the target distribution increases, the proportion of points that are rejected also increases. This curse of dimensionality makes rejection sampling an inefficient technique for sampling multi-dimensional distributions, as the majority of the points proposed are not accepted as valid samples.
- Some of these problems are solved by changing the form of the proposal distribution to "hug" the target distribution as we gain knowledge of the target from observing accepted samples. Such a process is called *Adaptive Rejection Sampling*, which will be covered in another post.

About dustinstansbury

I recently received my PhD from UC Berkeley where I studied computational neuroscience and machine learning.

View all posts by dustinstansbury »

Posted on September 10, 2012, in Density Estimation, Sampling Methods, Statistics and tagged Curse of Dimensionality, Envelope Distribution, Proposal Distribution, Rejection Sampling, Sampling Methods, Target Distribution. Bookmark the permalink. 4 Comments.

Leave a comment

Trackbacks 1

Comments 3

leo | June 30, 2014 at 7:19 pm
example 2, line 26

q = c*/fLenth; % ENVELOPE DISTRIBUTION

should be

q = c*1/fLength; % ENVELOPE DISTRIBUTION

> ***dustinstansbury*** | September 19, 2014 at 3:33 pm
> Thanks leo! The code has been corrected.

Albert Yao | October 2, 2014 at 5:06 pm
it's so clear explained. Thanks for ur sharing !

1. Pingback: **Bayes' Theorem Primer – Connections**

Bivariate Distributions

Independent: $f(x) \cdot f(y) = f(x,y)$

$E[Y_1] = \int_0^1 y_1 \cdot$ (Marginal)

Example: $f(x,y) = k(x+2y)$ where $0 < 2y < x < 2$

$k \cdot \int_0^2 \int_0^x x+2y \, dy\,dx$

$k \cdot \int_0^1 \int_{2y}^2 x+2y \, dx\,dy$

Area under: Inner $\int_f^1$ } Look where the „other" function
Area over: Inner $\int_0^f$ } enters and leaves the area and which value it is.

CDF

$F(y) = \int_0^y \ldots dy$

Is CDF, when
① $F(-\infty) = \lim_{y \to -\infty} = 0$
② $F(\infty) = \lim_{y \to \infty} = 1$
③ $F(y)$ is monotonically increasing

Density Function

$P(a \le y \le b) = \int_a^b f(y)\,dy$

$= F(b) - F(a)$

Properties of Variance

$V[c] = 0$

$V[Y+c] = V[c]$

$V[c \cdot g] = c^2 \cdot V[g]$

Partial Integration

$\int u' \cdot v = u \cdot v - \int u \cdot v'$

Select v that v' gets easier.

Mean and Median

Median: $\int_{-\infty}^{y} f(y)\,dy = 0.5$

Mean: $E[Y] = \int_{-\infty}^{\infty} y \cdot f(y)\,dy$

$E[g(y)] = \int_{-\infty}^{\infty} y(y) \cdot f(y)\,dy$

Unbiased

The mean of the estimate is equal to the parameter

Consistent

The variance of the estimator goes to 0 as $n \to \infty$.

V and E

Method of Moments

$M'k = E[Y^k]$ Population Moments
$m'k = 1/n \cdot \sum_{i=1}^{n} y_i^k$ Sample Moments
k = # parameters to estimate
Solve $M'k = m'k$

Maximum Likelyhood

$L(\lambda) = \prod_{i=1}^{n} f(y)$
Find maximum

Rules for E and V

E: ① constants out
② $E(\Sigma g) = \Sigma E(g)$
V: ① constants out + square
② $V(\Sigma g) = \Sigma V(g)$

$N(10,4)$ → mean to 0, deviation to 1 (squared)

$Z = \frac{x-\mu}{\sigma} \sim N(0,1)$

$V[Y] = E[Y^2] - E^2[Y]$

Hypothesis Testing

$P(z_{1-\alpha/2} \le \frac{\lambda - \mu}{\sigma} \le z_{1-\alpha/2})$ $\mu = 10$, $\sigma = 2$ | $\cdot\sigma$ $+10$

Linear Regression

$E[Y] = \beta_0 + \beta_1 X$

$V[Y] = \sigma^2$

$\beta_1 = \frac{S_{xy}}{S_{xx}}$

$\beta_0 = \bar{y} - \beta_1 \cdot \bar{x}$

For SSE: calculate difference between function values and given ones; square, add, profit.

Conditional Probability

$P(x|y) = \frac{P(x,y)}{P(y)}$

Marginals

$f(y_1) = \int f(y_1, y_2)\,dy_2$ } limit around these, so x in limits
$f(y_2) = \int f(y_1, y_2)\,dy_1$

$E[X|Y=0.5] = \int P(x|y=0.5) \cdot x \, dx$

MSE

$MSE = S^2 = \frac{SSE}{n-2}$ unbiased estimator of σ^2

Table

Y	X	$Y-\bar{Y}$	$X-\bar{X}$	$(X-\bar{X})^2$	$(X-\bar{X})(Y-\bar{Y})$
				↓	↓
				S_{XX}	S_{XY}

$SSE = \sum_{i=1}^{n} (y_i - \bar{y})^2 = e_i^2$

β_0: $T = \frac{\hat{\beta}_0 - 0}{S \cdot \sqrt{\frac{\Sigma x_i^2}{n \cdot S_{XX}}}}$ Test value; Original x values, square, sum, Profit

„check if T is inside region"

β_1: $T = \frac{\hat{\beta}_1 - 0}{\sqrt{\frac{\sigma^2}{S_{XX}}}} = \sigma^2 = \frac{SSE}{n-2}$ n = # parameters

$\sigma = S$ T = Acceptance Reg

$\sum_{i=1}^{n} (x_i - \bar{x})^2$ $\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})$

Sampling Distribution

$\bar{Y} = \frac{1}{n} \cdot \sum_{i=1}^{n} Y_i$

$S^2 = \frac{1}{n-1} \cdot \sum_{i=1}^{n} (Y_i - \bar{Y})^2$

$\chi^2(n) \sim \frac{(n-1) \cdot S^2}{\sigma^2}$ df

$t(v) \sim T = \frac{Z = N(0,1)}{\sqrt{\frac{W}{v}}}$ $Z \perp W$

Suggest estimator

$\bar{Y} = \frac{1}{n} \cdot \sum_{i=1}^{n} Y_i$ check $E(y) = \lambda$

Likelyhood

$Z = \frac{\hat{\Theta} - \Theta}{\sigma_{\hat{\Theta}}}$ } pivotal quantities

$\hat{\Theta} \pm z_{\alpha/2} \sigma_{\hat{\Theta}}$

Sum + Product

$\sum_{i=1}^{n} c = n \cdot c$ $\prod_{i=1}^{n} c = c^n$

Covariance

$cov(y_1, y_2) = E[y_1 y_2] - \mu_1 \mu_2$

$cov(y_1, y_2) = 0$ if Y_1, Y_2 are independent

if $Y_1 \perp Y_2$ then $cov(Y_1, Y_2) = E[Y_1] \cdot E[Y_2] = \mu_1 \cdot \mu_2$

$U = \sum_{i=1}^{n} a_i Y_i$ and $V = \sum_{j=1}^{m} b_i X_i$, then

a) $E[U] = \sum a_i \mu_i$ b) $V[U] = \sum a_i^2 \cdot Var(Y_i) +$

$2 \cdot \sum_{1 \le i < j \le n} \sum a_i a_j \, cov(y_i, y_j)$

c) $cov(U,V) = \sum_i \sum_j a_i b_j \, cov(Y_i, X_j)$

Bayesian Method for $\hat{\theta}$ estimate

① Prior $P(\Theta)$. Replace λ with Θ
② Likelyhood $P(y|\Theta)$ is $L(\lambda)$ → from LME
③ Write $P(\Theta|y) \propto P(\Theta) \cdot P(y|\Theta)$ Posterior; ignore constants
④ Form to known distribution. Consugate says, which family
⑤ Get λ^x (and β^x), spot difference
⑥ Probably E of distribution with new λx and b^x
⑦ For λ substitute $\lambda + \beta x$ in the dropped out constant with $\lambda^x + \beta^x$. Other values should be given.

Confidence Intervals

$100(1-\alpha)\%$
CI for $E[y] = \beta_0 + \beta_1 x^x$

$\hat{\beta}_1 \pm t_{\alpha/2} \cdot S \cdot \sqrt{c_{11}}$

$c_{00} = \frac{\Sigma x_i^2}{n \cdot S_{XX}}$ $c_{11} = \frac{1}{S_{XX}}$

$\hat{\beta}_0 + \hat{\beta}_1 x^x \pm t_{\alpha/2} \cdot t(n-2)$

$\cdot \frac{1}{n} \cdot \sqrt{\frac{(x^x - \bar{x})^2}{S_{XX}}} \cdot S$

$\chi^2(n) = \frac{(n-1) S^2}{\sigma^2}$ Population variance; Sample variance; n: df

Point Estimators

target parameter Θ	Sample size	point estimator $\hat{\Theta}$	$E(\hat{\Theta})$	Std. error $\sigma_{\hat{\Theta}}$
μ	n	$\bar{Y}$	μ	$\sigma/\sqrt{n}$
p	n	$\hat{p} = Y/n$	p	$\sqrt{p \cdot q / n}$
$\mu_1 - \mu_2$	n_1, n_2	$\bar{Y}_1 - \bar{Y}_2$	$\mu_1 - \mu_2$	$\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}$
$p_1 - p_2$	n_1, n_2	$\hat{p}_1 - \hat{p}_2$	$p_1 - p_2$	$\sqrt{\frac{p_1 q_1}{n_1} + \frac{p_2 q_2}{n_2}}$

Prediction Intervals

$100(1-\alpha)\%$ PI for y when $x = x^x$

$\hat{\beta}_0 + \hat{\beta}_1 x^x \pm z_{\alpha/2} \cdot S \cdot \sqrt{1 + \frac{1}{n} + \frac{(x^x - \bar{x})^2}{S_{XX}}}$

$\sigma^2 < \ldots$ S^2

abb.com

Properties of Least-Squares Estimators Simple Linear Regression

① $\hat{\beta}_0$ and $\hat{\beta}_1$ are unbiased, $E[\hat{\beta}_i] = \beta_i$

② $V(\hat{\beta}_0) = \left(\sum x_i^2 / (n \cdot S_{xx})\right) \cdot \sigma^2$

③ $V(\hat{\beta}_1) = \sigma^2 / S_{xx}$

④ $\text{cov}(\hat{\beta}_0, \hat{\beta}_1) = \frac{-\bar{x}}{S_{xx}} \cdot \sigma^2$

⑤ Unbiased estimator of σ^2 is $S^2 = SSE/(n-2)$
where $SSE = S_{YY} - \hat{\beta}_1 S_{xy}$ and
$S_{YY} = \sum (y_i - \bar{y})^2$

If ε_i are normally distributed

⑥ $\hat{\beta}_0, \hat{\beta}_1$ are normally distributed

⑦ $\frac{(n-2)S^2}{\sigma^2}$ has χ^2 with $n-2$ df

⑧ Statistics S is independant of both $\hat{\beta}_0$ and $\hat{\beta}_1$

$\perp$ = independent

$E[\bar{Y}] = E[\frac{1}{n} \cdot \sum Y] = \frac{1}{n} \sum E[Y] = \frac{1}{n} \cdot n \cdot \mu = \mu$

$V[\bar{Y}] = V[\frac{1}{n} \sum Y] = \frac{1}{n^2} \cdot \sum V[Y] = \frac{1}{n^2} \cdot n \cdot \sigma^2 = \frac{\sigma^2}{n}$ → 0 as $n \to \infty$
↳ consistent

Poisson
$e^{-\theta(n+1/\beta)} \Rightarrow \beta^* = \frac{1}{n + 1/\beta}$

T-Testing

- Calculate T
- Look up $t_{\alpha/2}$
- $|T| > t$ reject
- $|T| < t$ don't reject

$$T = \frac{a^T\beta - a^T\beta_0}{S \cdot \sqrt{a^T (X^TX)^{-1} a}}$$

General Linear Models

OLS solution:
$\hat{\beta} = (X^TX)^{-1} X^TY$
$SSE = Y^TY - \hat{\beta} X^TY$
$Y^TY = \sum y_i^2$
$S^2 = \frac{SSE}{n-2}$

t.ex. Test H_0: $a^T\beta = (a^T\beta)_0$

$$T = \frac{a^T\beta - a^T\beta_0}{S \cdot \sqrt{a^T (X^TX)^{-1} a}}$$

A $100(1-\alpha)\%$ CI for $a^T\beta$ is given by

$$a^T\beta \pm t_{\alpha/2} S \sqrt{a^T (X^TX)^{-1} a}$$

↳ $t(n-K-1)$ df

K: order of the moment, #parameters −1
K: highest $\beta_K \to x = k$

t.ex inference for β_1? $a = (0,1,0,\dots)$

$a^T\beta = \beta_1$

Bayes Estimator

$\hat{\theta} = E[p(\theta|y)]$
↳ „Posterior Mean"
The posterior mean minimizes the MSE = $E[(\hat{\theta} - \theta)^2]$
MAE is without square

Gamma Function

$\Gamma(n) = (n-1)!$
$0! = 1! = 1$

x^x with $E[Y] =$

$$\beta_0 + \beta_1 \cdot x^* \pm t_{\alpha/2} \cdot S \cdot \sqrt{\frac{1}{n} + \frac{(x^* - \bar{x})^2}{S_{xx}}}$$

CI

$\frac{1}{n}$: plus

PI: for Y when $x = x^*$

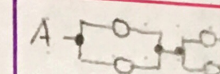
$$\beta_0 + \beta_1 x^* \pm t_{\alpha/2} \cdot S \cdot \sqrt{1 + \frac{1}{n} + \frac{(x^* - \bar{x})^2}{S_{xx}}}$$

Ho-Testing (for LR)

We have two approaches:
T-Testing or CI-Testing

CI-Testing
- Calculate CI with formula
- Check if real value lies within

$$a^T\beta \pm t_{\alpha/2} \cdot S \cdot \sqrt{a^T (X^TX)^{-1} a}$$



$\hat{p} = (0.9 + 0.9 - (0.9)^2) \cdot (0.9 + 0.9 - (0.9)^2)$

Introduction
Computer Arithmetics

732A90
Computational Statistics

Krzysztof Bartoszek
(krzysztof.bartoszek@liu.se)

22 I 2019 (P42)
Department of Computer and Information Science
Linköping University

Computational Statistics — In brief

- Even simple data analysis (mean, variance) by hand is tedious
- Today: huge datasets, models capturing system complexity, interactions (between variables **and** observations)
- We will discuss:
  - Being careful with calculations—overflow
  - Generation of random variables including correlated ones
  - Numerically optimizing functions, esp. maximum likelihood
  - Computing confidence (credible) intervals for distributions when analytical ones are unobtainable

Lesson structure

- Lectures
- Computer Labs
- Seminars
- Examination: Reports, seminars, final exam
- Final exam: computer based
- Answer in English.
- Electronic reports as **.PDF**.
- Disclose **ALL** collaborations and sources.
- Provide source code (if used).
- E–mail contact: krzysztof.bartoszek@liu.se

Course materials, software

- Lecture slides
- 2016 lecture slides (732A38)
- Handouts, R code
- Various suggested www pages or articles
- **Googling**
- James E. Gentle “Computational Statistics”, Springer, 2009
- Geof H. Givens, Jennifer A. Hoeting “Computational Statistics”, Wiley, 2013

- R

Course contents

- Recap: R
- Recap: Basic Statistics
- Computer Arithmetics (JG pages 85–105)
- Optimization (JG pages 241–272, handouts)
- Random Number Generation (JG pages 305–312, 325–328, handouts)
- Monte Carlo Methods (JG pages 312–318, 328 417–429, handouts)
- Numerical Model Selection and Hypothesis Testing (JG pages 52–56, 424, 435–467, handouts)
- Expectation Maximization Algorithm and Stochastic Optimization (JG pages 275–284, 296–298, 480–483, handout)

Pages are recommended reading for each lecture, **NOT** exact lecture content. The lectures will build up on this material.

Examination

Computer labs (need to be passed)

Presentation or opposition and attendance at seminars (see 732A90_ComputationalStatisticsVT2019_CourseInformation.pdf).

Computer exam points
A: $[18, \infty)$, B: $[16, 18)$, C: $[14, 16)$, D: $[12, 14)$, E: $[10, 12)$, F: $[0, 10)$

Allowed aids for exam: printed books and own PDF document containing max 100 pages (see 732A90_ComputationalStatisticsVT2019_CourseInformation.pdf).

Computer Arithmetics

SHOULD YOU CARE?

For a detailed mathematical treatment see Ch. 4.2 in
D. E. Knuth (1998). The Art of Computer Programming, Volume 2, Addison–Wesley.

Computer Arithmetics: Examples

Computations can be affected by magnitudes of numbers.

```
x<-0.5^10000;y<-0.4^10000;x/(x+y)+y/(x+y)
x<-0.5^1000;y<-0.4^1000;x/(x+y)+y/(x+y)
x<-0.1^1000;y<-0.2^1000;x/(x+y)+y/(x+y)
```

```
t<-rnorm(5,10^18,1);t[3]-t[4];t[1]-t[2]
```

```
x<-10^800;sd<-10^400;y<-x/sd;y
```

And your are doing estimation under a nice, fancy model ...

Data presentation

- Computers store information in binary form
  0 1 0 1 1 0 0 1
- 1Byte=8bits (typical counting unit)
- 1Word=32 or 64bits (depending on architecture)
- 1KB=1024bytes
- 1MB=1024KB
- and so on

QUESTION: Why binary form?

Character encoding

- ASCII (American Standard Code for Information Interchange)
  - 1 byte per character, 7 bits coding, 1 parity check or 0
  - $2^7 = 128$ characters can be encoded
  - "Usual" English letters, Arabic numerals, punctuation, i.e. "standard" keyboard
  - 1–31 control characters, 0: NULL **WHY?**
  - Design influenced by contemporary (1960) hardware
  - Extended ASCII: all 8 bits, 256 characters
- Unicode
  - 8, 16 or 32 bits encoding
  - "of more than 128,000 characters covering 135 modern and historic scripts, as well as multiple symbol sets" (Wikipedia)
- `read.csv()`, `read.table()` have `fileEncoding` argument

Fixed–point system (integers)

- We use the base–10 (decimal) system, e.g.
  $1234 = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$
- We could use base–m system for any m
- Computers: base–2 (binary) system
- Each integer represented as:
  $A = a_0 \cdot 2^0 + a_1 \cdot 2^1 + a_2 \cdot 2^2 + \ldots$

EXERCISE: 5, 16, 17, 31, 32, 33, 255, 256 in binary
QUESTION: What is the range of a byte, word, double word?

- Negative numbers
  - **Leading bit:** first bit 0 if positive, 1 if negative
  - **Twos–complement:** sign bit 1, remaining bits to **opposite** value and then +1
    e.g. $5 = 00000101$, $-5 = 11111011$
    QUESTIONS $5 + (-5) =$?, try 12 and -12
  - Range: $[-2^{k-1}, 2^{k-1} - 1]$ on k bits **WHY?**

Arithmetic operations

R operations on binary

- Addition, multiplication: base–2 instead of base–10
- Subtraction: $A - B = A + (-B)$ (*twos–complement*)
- Division: tedious, rounded towards 0 `as.integer(17/3)`
- **Overflow:** adding two large numbers the sign bit can be treated as a high order bit and on some architectures results in a negative number

Floating–point system (rational, “real”)

- How can we represent fractions (rational numbers)?
- Sign
- Exponent (**signed**, read standards if interested)
- Mantissa or Significand
- on 64bits:

sign (1bit)	Exponent (11bits)	Mantissa (52 bits)

$$\pm 0.d_1 d_2 \dots d_p \cdot b^e \qquad b = 2, \quad p = 52$$

- **Range:** $\approx [-10^{300}, 10^{300}] \approx [-b^{e_{max}}, b^{e_{max}}]$

Floating–point system

- Rationals rounded towards the nearest computer float

```
options(digits=22) #max possible
0.1
[1] 0.10000000000000000055511
```

- **EXAMPLE:** Assume base $b = 10$ and mantissa has 5 digits $p = 5$:
  $1.2345 = +0.12345 \cdot 10^1$
  $4.0000567 = +0.40000 \cdot 10^1$
- Problem remains whatever base (b) is chosen
- **EXERCISE:** Try to convert some numbers

Floating–point system

- Distribution of computer floats

- Dense from -1 to 1
- Density decreases
- same number of points for each exponent:

$$\dots, \ \cdot 10^{-3}, \ \cdot 10^{-2}, \ \cdot 10^{-1}, \ \cdot 10^{1}, \ \cdot 10^{2}, \ \cdot 10^{3}, \ \dots$$

- What about integers?
  $5 = +0.50000 \cdot 10^1$

```
options(digits=22)
9007199254740992
9007199254740993
9007199254740994
```

Floating–point system, special “numbers”

- We do not discuss how the exponent is actually coded.
- Usually the maximum allowed number in the exponent is one unit less than possible.
- $\pm$Inf: exponent is $\exp_{\max} + 1$, mantissa is 0
- NaN: exponent is $\exp_{\max} + 1$, mantissa is $\neq 0$
- 0 **WHY?**

Overflow: number larger than can be represented
Underflow: loss of significant digits

```
10^200*10^200 =  Inf
10^400/10^400 =  NaN
10^-200/10^200 =  0
10^-200*10^200 =
0*10^400 =
x<-10^300;while(1){x<-x+1}
```

Arithmetic operations

- Floats are rounded so usual mathematical laws do not hold — floating point arithmetic
- **Examples**

```
1/3+1/3 =  0.6666667
options(digits=22)
1/3+1/3 =  0.66666666666666296592
10^(-200)/(10^(-200)+10^(-200)) =  0.5
10^(-200)/(10^(-200)+20^(-200)) = 1
```

- Software is designed to make operations as correct as possible
- **Do we need to work with such extreme numbers?**

Arithmetic operations

- $X + Y$, $X \cdot Y$ can display overflow, underflow
- $A \neq B$ but $X + A = B + X$
- $A + X = X$ but $A + Y \neq Y$
- $A + X = X$ but $X - X \neq A$
- **COMPARING FLOATS IS TRICKY!**

```
options(digits=22)
x<-sqrt(2)
x*x
[1] 2.0000000000000000444089
(x*x)==2
[1] FALSE
isTRUE(all.equal(x*x,2))
[1] TRUE
```

Summation

Underflow problems can occur with any summation (`x<-x+1`)

```
options(digits=22)
x<-1:1000000; sum(1/x); sum(1/rev(x))
[1] 14.392726722865722521 76
[1] 14.392726722865724298 12
```

- **WHICH ONE IS CORRECT?**
- **WHICH ONE IS MORE ACCURATE?**

Potential solutions

Solution A:

1. Sort the numbers ascending **CAN BE EXPENSIVE**
2. Sum in this order

Solution B:

1. Sum numbers pairwise, from n obtain $n/2$ numbers **HOW TO CHOOSE PAIRS?**
2. Continue until 1 number left

More on summing

Example

- Computing exponent using Taylor series

$$e^x = 1 + x + x^2/2 + x^3/6 + \ldots = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

The exponential

```
options(digits=22)
fTaylor<-function(x,N){1+sum(sapply(1:N,
    function(i,x){x^i/prod(1:i)},x=x,simplify=
    TRUE))}
exp(20) #fine
[1] 485165195.4097902774811
fTaylor(20,100)
[1] 485165195.4097902774811
fTaylor(20,100)-exp(20)
[1] 0
exp(-20) #problem
[1] 2.061153622438557869942e-09
fTaylor(-20,100)
[1] -3.853877217352419393137e-10
fTaylor(-20,200)
[1] -3.853877217352419393137e-10
```

More on summing

Example

- Computing exponent using Taylor series

$$e^x = 1 + x + x^2/2 + x^3/6 + \ldots = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

- **WHY?**

- Varying sign of terms
- **CANCELLATION** adding two numbers of almost equal magnitude but of opposite sign
- Effects of cancellations accumulate
- **SOLUTION:** Different algorithm ...

Can you explain why?

```
## Example due to Thomas Ericsson in his
    Numerical Analysis course at Chalmers
f1<-function(x){(x-1)^6}
f2<-function(x){1-6*x+15*x^2-20*x^3+15*x^4-6*x
    ^5+x^6}

x<-seq(from=0.995,to=1.005,by=0.0001)
y1<-f1(x);y2<-f2(x)

plot(x,y1,pch=19,cex=0.5,ylim=c(-5*10^(-15),20
    *10^(-15)),main="Two ways to calculate (x
    -1)^6",xlab="x",ylab="y")
points(x,y2,pch=18,cex=0.8)
```

Matrix Calculations

- Many problems (in Statistics, Numerical methods, e.t.c) can be reduced to solving

$$\mathbf{A}\vec{x} = \vec{b}$$

$\mathbf{A}$ (design) matrix
$\vec{x}$ vector of unknowns
$\vec{b}$ vector of scalars (data)

- Algorithm should be **numerically stable**

(small changes in $\mathbf{A}$ or $\vec{b}$ imply in small changes in $\vec{x}$)

Example: Linear regression models

Minimize

$$RSS(\beta_0, \beta_1, \ldots, \beta_m) = \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_{i1} - \ldots - \beta_m x_{im})^2$$

The system of equations

$$\frac{\partial RSS}{\beta_0} = \frac{\partial RSS}{\beta_1} = \ldots \frac{\partial RSS}{\beta_m} = 0$$

can be written as

$$\mathbf{X}^T\mathbf{X}\vec{\beta} = \mathbf{X}\vec{y}$$

where

$$\mathbf{X} = \begin{bmatrix} 1 & x_{11} & \ldots & x_{1m} \\ 1 & x_{21} & \ldots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \ldots & x_{nm} \end{bmatrix}$$

Solving a linear system

- $\mathbf{A}\vec{x} = \vec{b}$ needs a stable numerical solution
- **Computer arithmetics**

- - Original system: $\mathbf{A}\vec{x} = \vec{b}$
  - Perturbed system: $\mathbf{A}\vec{x'} = \vec{b'}$, $\vec{x'} = \vec{x} + \delta\vec{x}$, $\vec{b'} = \vec{b} + \delta\vec{b}$
- Stable: small perturbation in $\vec{b}$, small perturbations in $\vec{x}$
- $\|\vec{b}\| = \|\mathbf{A}\vec{x}\| \leq \|\mathbf{A}\|\|\vec{x}\|$ implies $\|\vec{x}\|^{-1} \leq \|\mathbf{A}\|\|\vec{b}\|^{-1}$
  $\|\delta\vec{x}\| = \|\mathbf{A}^{-1}(\delta\vec{b})\| \leq \|\mathbf{A}^{-1}\|\|\delta\vec{b}\|$
  together

$$\frac{\|\delta\vec{x}\|}{\|\vec{x}\|} \leq \|\mathbf{A}^{-1}\|\|\mathbf{A}\|\frac{\|\delta\vec{b}\|}{\|\vec{b}\|}$$

Solving a linear system

Condition number of a matrix

$$\kappa(\mathbf{A}) = \|\mathbf{A}^{-1}\|\|\mathbf{A}\|$$

- Large $\kappa(\mathbf{A})$ is a bad sign, but does not imply ill–conditioning
- L_2 norm: $\kappa(\mathbf{A})$ is the ratio of the maximum and minimum eigenvalues of $\mathbf{A}$
- Under L_2 norm

$$\kappa(\mathbf{A}^T\mathbf{A}) \geq \kappa(\mathbf{A})^2 \geq \kappa(\mathbf{A})$$

(regression setting)

Solving a linear system

Dealing with ill–conditioning

- Rescale the variables (columns)
- Use a different algorithm for solving
  e.g. QR, Cholesky, SVD

Cholesky: **A** symmetric–positive–definite
$\mathbf{A}\vec{x} = \vec{b}$ is equivalent to $\mathbf{L}\mathbf{L}^T\vec{x} = \vec{b}$ **WHY?**

1. Solve $\mathbf{L}^T\vec{y} = \vec{b}$
2. Solve $\mathbf{L}\vec{x} = \vec{y}$

Optimization

732A90
Computational Statistics

Krzysztof Bartoszek
(krzysztof.bartoszek@liu.se)

24 I 2019 (P42)
Department of Computer and Information Science
Linköping University

Summary

- Computations can behave "differently" at different numerical ranges.
- Floating point system.
- Computer arithmetics is not the same as "usual" arithmetic.
- Summing series, solving linear systems (inversion?)

Plan for today

- Introduction
- Mathematical definition of problem
- 1D optimization
- kD optimization
- `R` code examples

Optimization

Nearly everything is optimization **!**

- Chemistry
- Physics
- Economics, **Industry**
- Engineering

BUT EVEN

- Your mobile price plan
- Course scheduling
- Your lunch choice

STATISTICS

- Fit parameters to data
- Propose optimal decision

Optimization: Example

ANY BIOLOGICAL

ORGANISM

YOU

Optimization: Example

Industry

How to produce a cylindrical (**WHY?**) $0.5L$ beer can so it requires minimum material?

Given a certain product minimize e.g. material usage, production effort while still meeting consumer requirements.

Optimization: Example

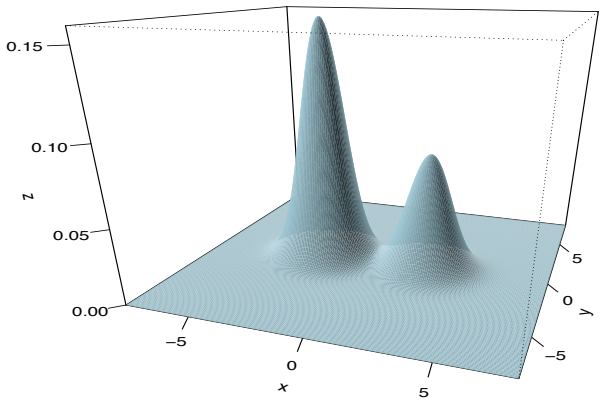
Economics/Logistics

- Travelling Salesman Problem
- Windmills
- Flight schedule (especially “cheap” airlines)

Optimization: Example

Statistics

Maximize likelihood, model fitting



Mathematical formulation

The goal is to minimize (maximize)

Objective function: $f(\theta)$
(reproduction, chances of survival, quality of life, cost, profit, likelihood, fit to data)

depending on

Parameters or **Unknowns** θ
(reproduction strategy, resource utilization, consumer choices, height & diameter, production, raw material choice, service times, route, flight routes/times ,parameters)

Maximal likelihood

An i.i.d. sample $(X_1, \ldots, X_n)$ is drawn from a probability distribution $P(X|\Theta)$, where Θ is an unknown parameter set.

The joint probability of all the observations is

$$P(X_1, \ldots, X_n|\Theta) = \prod_{i=1}^{n} P(X_i|\Theta).$$

Find Θ that maximizes $P(X_1, \ldots, X_n|\Theta)$.

Mathematical formulation

$$\min_{\theta \in \Theta} f(\theta) \quad \text{subject to} \quad \begin{array}{ll} c_i(\theta) = 0, & i \in E \\ c_i(\theta) \geq 0, & i \in I \end{array}$$

QUESTION: What should we do if we are interested in maximization instead of minimization?

QUESTION: What should we do if the constraints are $c_i(x) \leq 0,\ i \in I$?

Constraints examples

- Available environment
- Volume: 0.5l of can
- Production: Factories (F_1, F_2), retail outlets (R_1, R_2, R_3), cost of shipping $i \to j$: c_{ij}, production a_i per week, requirement b_j per week **to optimize:** x_{ij} amount shipped $i \to j$ per week

$$\begin{aligned} &\min_{x \in \mathrm{R}^3} \sum_{ij} c_{ij} x_{ij} && \text{minimize shipping costs} \\ &\sum_{j=1}^{3} x_{ij} \leq a_i, i = 1, 2 && \text{production capacity} \\ &\sum_{i=1}^{3} x_{ij} \geq b_j, j = 1, 2, 3 && \text{demand} \\ &\forall_{i,j} x_{ij} \geq 0 \end{aligned}$$

Question: What would happen if we drop demand constraint?

- ML: often no constraints

Optimization approaches

- Constrained optimization
  - Lagrange multipliers, linear programming
  - E.g. LASSO
  - **Not this lecture!**
- Unconstrained optimization
  - Steepest descent
  - Newton method
  - Quasi–Newton–Methods
  - Conjugate gradients

Why are there different methods?

Exercise

- Split into pairs/triplets/quadruples
- Think of some human anatomy part/organ:
  - What is its function?
  - What could it have been optimized for over the course of time?
  - Is it still under selection?
  - What constraints was and is it under?
- Think of a situation where optimization is needed in your own student/professional/personal/financial situation.
- State the problem in terms of
  - Objective function
  - Parameters
  - Constraints
  - Does it have a trivial solution?
- **10** minutes

1D Optimization

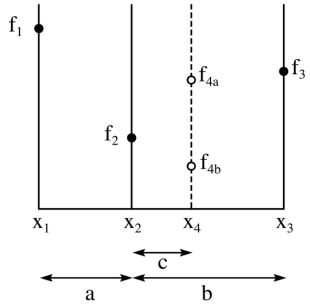
- Function of a single parameter, find minimum
- *What algorithm would you suggest?*
- **Golden–section search**
  local minimum on $[A, B]$ interval (constraint)
- Works by narrowing down the search interval with a constant reduction factor

$$1 - \alpha = \frac{\sqrt{5} - 1}{2} \approx 0.62$$

Question: Does α remind you of something?

Golden section (minimization)

1: $x_1 = A$, $x_3 = B$,
2: **while** $x_1 - x_3 > \epsilon$ **do**
3: $a = \alpha(x_3 - x_1)$
4: $x_2 = x_1 + a$, $x_4 = x_3 - a$
5: **if** $f(x_4) > f(x_2)$ **then**
6: $x_1 = x_1$, $x_3 = x_4$
7: **else**
8: $x_1 = x_2$, $x_3 = x_3$
9: **end if**{We know the value at **3** points!}
10: **end while**



Wikipedia, Golden–section search

f has to be **UNIMODAL**

1D Optimization: Example

`732A90_ComputationalStatisticsVT2019_Lecture02codeSlide16.R`

Multi–dimensional optimization

Find

$$\min_{\vec{x} \in \mathbb{R}^n} f(\vec{x})$$

Using (known, or numerically evaluated)

$$\textbf{Gradient} \quad \bigtriangledown f(\vec{x}) = \left(\frac{\partial f(\vec{x})}{\partial x_1}, \ldots, \frac{\partial f(\vec{x})}{\partial x_n} \right)^T$$

$$\textbf{Hessian} \quad \bigtriangledown^2 f(\vec{x}) = \left[\frac{\partial^2 f(\vec{x})}{\partial x_i \partial x_j} \right]_{i,j=1}^{n}$$

General strategy

1. Provide a **(good)** starting point $\vec{x}_0$, $\vec{x} = \vec{x}_0$
2. Choose a direction $\vec{p}$ ($\|p\| = 1$) and step size a
3. Move to $\vec{x} := \vec{x} + \alpha\vec{p}$
4. Repeat step 2 until convergence

How to choose the direction?

Taylor's theorem

$$f(\vec{x}+a\vec{p}) = f(\vec{x}) + \boxed{\alpha\vec{p}^T \cdot \bigtriangledown f(\vec{x})} + o(\alpha^2)$$

$\vec{p}$ s.t. $\vec{p}^T \cdot \bigtriangledown f(\vec{x}) < 0$ is a *descent* direction.

Steepest descent is

$$\vec{p} = -(\bigtriangledown f(\vec{x}))/\| \bigtriangledown f(\vec{x})\|$$

How to choose the step size?

- **Expensive way**: find the global minimum in direction $\vec{p}$
- **Trade–off way**: find a decrease which is *sufficient*

BACKTRACKING

1: Choose (large) $\alpha_0 > 0$, $\rho \in (0,1)$, $c \in (0,1)$,
2: $\alpha = \alpha_0$
3: **repeat**
4: $\quad \alpha = \rho\alpha$
5: **until** $f(\vec{x}+\alpha\vec{p}) \leq f(\vec{x}) + c\alpha\vec{p}^T \bigtriangledown f(\vec{x})$

Newton's method

- Newton–Raphson method
- Hessian ignored in steepest descent
- If f is quadratic

$$f(\vec{p}) = \frac{1}{2}\vec{p}^T\mathbf{A}\vec{p} + \vec{b}^T\vec{p} + c,$$

then minimum

$$\vec{p}^* = \mathbf{A}^{-1}\vec{b}.$$

- Taylor expansion of f

$$f(\vec{x}+a\vec{p}) = f(\vec{x}) + \alpha\vec{p}^T \cdot \bigtriangledown f(\vec{x}) + \frac{\alpha^2}{2}\vec{p}^T \bigtriangledown^2 f(\vec{x})\vec{p} + o(\alpha^3)$$

- $x := x + \alpha\vec{p}$ where

$$\boxed{\vec{p} = -\left(\bigtriangledown^2 f(\vec{x})\right)^{-1} \bigtriangledown f(\vec{x})}$$

Newton's method

- $\left(\bigtriangledown^2 f(\vec{x})\right)^{-1}$ is expensive to compute, there are quicker approaches, e.g. Cholesky decomposition
- Hessian should be **positive definite** for $\vec{p}$ to be a descent direction (if not see book)
- Memory expensive — need to store $O(n^2)$ elements

BUT

- Method converges quickly esp. near optimum

Quasi–Newton methods

- k iteration number
- Compute an approximation to the Hessian, $\mathbf{B}$, that will allow for efficient choice of $\vec{p}$.
- **SECANT CONDITION:** (quasi–Newton condition)

$$\mathbf{B}_{k+1}\left(\vec{x}_{k+1} - \vec{x}_k\right) = \bigtriangledown f(\vec{x}_{k+1}) - \bigtriangledown f(\vec{x}_k)$$

BFGS Algorithm

1: Choose $\mathbf{B}_0 > 0$, $\vec{x}_0$, $k = 0$
2: **repeat**
3: $\quad \vec{p}_k$ is solution of $\mathbf{B}_k\vec{p}_k = \bigtriangledown f(\vec{x}_k)$
4: $\quad$ find suitable α_k
5: $\quad \vec{x}_{k+1} = \vec{x}_k + \alpha_k\vec{p}_k$
6: $\quad$ calculate $\mathbf{B}_{k+1}$ {next slide}
7: $\quad k = k + 1$
8: **until** convergence of $\vec{x}_k$ at minimum

How to compute $\mathbf{B}_{k+1}$?

- We want $\mathbf{B}_{k+1}$ and $\mathbf{B}_k$ to be close to each other
- $$\min_{\mathbf{B}} \|\mathbf{B} - \mathbf{B}_k\|$$
  $$s.t.\ \mathbf{B} = \mathbf{B}^T,\ \text{secant condition}$$
- $\vec{y}_k = \bigtriangledown f(\vec{x}_{k+1}) - \bigtriangledown f(\vec{x}_k)$, $\vec{s}_k = \vec{x}_{k+1} - \vec{x}_k$
- $$\mathbf{B}_{k+1} = \mathbf{B}_k - \frac{\mathbf{B}_k\vec{y}_k\vec{y}_k^T\mathbf{B}_k}{\vec{y}_k^T\mathbf{B}_k\vec{y}_k} + \frac{\vec{s}_k\vec{s}_k^T}{\vec{y}_k\vec{s}_k^T}$$
- Closed form Sherman–Morrison formula for $\mathbf{B}_{k+1}^{-1}$
- We have to store $\mathbf{B}_k^{-1}$

BFGS

- BGFS: Broyden–Fletcher–Goldfarb–Shanno
- More iterations than Newton's method (uses approximation)
- Each iteration quicker, no numeric inversion
- Good for large scale problems
- Choice of $\mathbf{B}_0$?

Conjugate Gradient method—quadratic case

Minimize

$$f(\vec{x}) = \frac{1}{2}\vec{x}^T\mathbf{A}\vec{x} - \vec{b}^T\vec{x}$$

for $\mathbf{A}$ symmetric positive definite.
Gradient:

$$\bigtriangledown f(\vec{x}) = \mathbf{A}\vec{x} - \vec{b} = r(\vec{x})$$

Two vectors $\vec{p}$ and $\vec{q}$ are **conjugate** with respect to $\mathbf{A}$ if

$$\vec{p}^T\mathbf{A}\vec{q} = 0.$$

IDEA: $\vec{p}$ and $\vec{q}$ are orthogonal w.r.t. to an inner product associated with $\mathbf{A}$. Use this to find a basis that will allow for easy finding of $\vec{x}$.

Conjugate Gradient method

- $\vec{p}_0 = \vec{r}_0$
- $\vec{p}_{k+1} = -\vec{r}_k + \beta_{k+1}\vec{p}_k$

- Conjugate condition has to be satisfied so

$$\beta_{k+1} = \frac{\vec{r}_k^T \mathbf{A} \vec{p}_{k-1}}{\vec{p}_k^T \mathbf{A} \vec{p}_k}$$

Exercise: check this

- Convergence in $\dim(\mathbf{A})$ steps
(or unless cutoff for $\vec{r}_k$)

Nonlinear CG method

- If $f(\cdot)$ general, use $\bigtriangledown f(\cdot)$ instead of $r(\cdot)$

1: Choose $\vec{x}_0$, $\vec{p}_0 = -\bigtriangledown f(\vec{x}_0)$, $k = 0$
2: **while** $\bigtriangledown f(x_k) \neq \vec{0}$ **do**
3: find suitable α_k
4: $\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{p}_k$ {and now update step}
5: $\beta_{k+1} = \left(\bigtriangledown^T f(\vec{x}_{k+1}) \bigtriangledown f(\vec{x}_{k+1})\right) / \left(\bigtriangledown^T f(\vec{x}_k) \bigtriangledown f(\vec{x}_k)\right)$
{Fletcher–Reeves update, other possible}
6: $\vec{p}_{k+1} = -\bigtriangledown f(\vec{x}_{k+1}) + \beta_{k+1}\vec{p}_k$
7: $k = k + 1$
8: **end while**

Nonlinear CG method

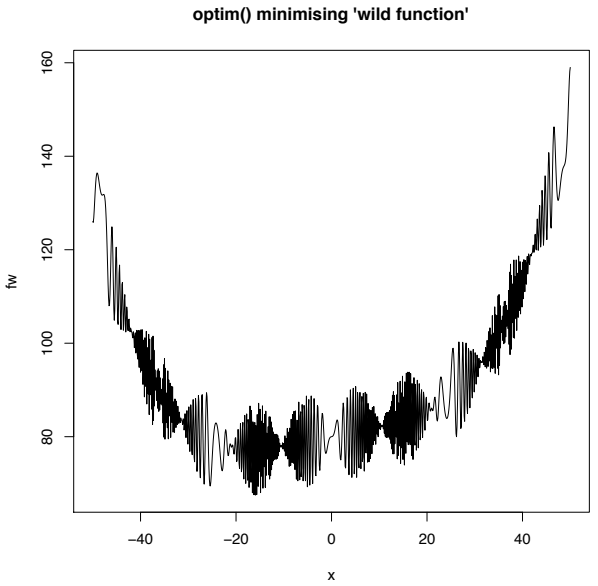
- Local minimum convergence

- But this is true of all methods that cannot "jump out" of descent path

- Faster than steepest descent

- Slower than Newton and Quasi–Newton but significantly less memory

kD Optimization: Example



kD Optimization: Example

`732A90_ComputationalStatisticsVT2019_Lecture02codeSlide31.R`

Random Number Generation

732A90
Computational Statistics

Krzysztof Bartoszek
(krzysztof.bartoszek@liu.se)

30 I 2019 (P42)
Department of Computer and Information Science
Linköping University

Summary

- Optimization is everywhere
- Numerical methods for finding minimum
- 1D: Golden section (unimodal), `optimize()`
- kD: choose step size and direction (gradient), `optim()`

Pseudorandom numbers

- **A computer is a deterministic machine**
- *Congruential generators*
- Functions of time
- Be careful with respect to application

First step: Generating Unif$[0, 1]$

Linear congruential generator

Define a sequence of integers according to

$$x_{k+1} = (a \cdot x_k + c) \mod m, \quad k \geq 0$$

x_0 is **seed**, e.g. based on time

mod m: remainder after division by m

- $x_k \in \{0, \ldots, m-1\}$ and integer
- $x_k/m \sim \text{Unif}[0, 1]$
- $a, c \in [0, m)$ need to be carefully selected

First step: Generating Unif$[0, 1]$

Generated numbers will get into a loop with a certain **period**

$$x_{k+1} = (a \cdot x_k + c) \mod m, \quad k \geq 0$$

$x_0 = a = c = 7$, $m = 10$

1. $x_1 = (7 \cdot 7 + 7) \mod 10 = 56 \mod 10 = 6$
2. $x_1 = (7 \cdot 6 + 7) \mod 10 = 49 \mod 10 = 9$
3. $x_1 = (7 \cdot 9 + 7) \mod 10 = 70 \mod 10 = 0$
4. $x_1 = (7 \cdot 0 + 7) \mod 10 = 7 \mod 10 = 7$
5. $x_1 = (7 \cdot 7 + 7) \mod 10 = 56 \mod 10 = 6$
6. ...

First step: Generating Unif$[0, 1]$

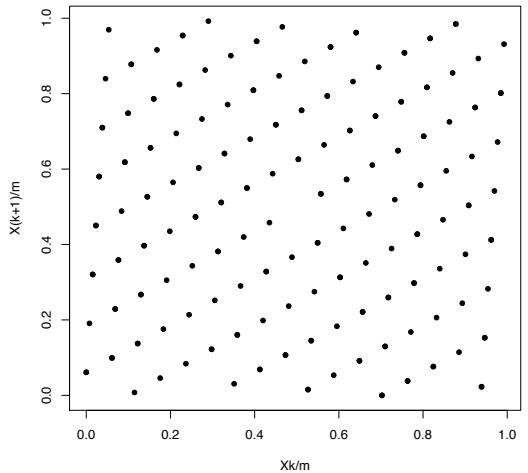
```
fthreebits<-function(k,s,L,N){
    X0<-4*s+1;a<-8*k+5;m<-2^L;X<-X0
    for (i in 1:N){
        print(c(X,rev(intToBits(X)[1:5])))
        X<-(a*X)%%m ##c=0
    }
}
```

```
> source("CongGen.R");fthreebits(k=2,s=3,L=8,N=10)
[1] 13  0  1  1  0  1
[1] 17  1  0  0  0  1
[1] 101   0   0   1   0   1
[1] 73  0  1  0  0  1
[1] 253   1   1   1   0   1
[1] 193   0   0   0   0   1
[1] 213   1   0   1   0   1
[1] 121   1   1   0   0   1
[1] 237   0   1   1   0   1
[1] 113   1   0   0   0   1
```

Last three bits change between 001 and 101
Discard less significant bits

First step: Generating Unif$[0, 1]$

See also D. E. Knuth (1998). The Art of Computer Programming, Volume 2, Addison–Wesley. Ch. 3.3.4



732A90_ComputationalStatisticsVT2019_Lecture03codeSlide06.R

First step: Generating Unif$[0,1]$

- Period is $\leq m$ by definition
- a, c, m (**large**) have to be chosen carefully
  1. c and m have to be relatively prime (no common divisors bar 1)
  2. $a = 1 \mod p$ for every prime divisor p of m
  3. $a = 1 \mod 4$ if 4 divides m
  4. Then full period m reached **(what about $a = c = 1$?)**
- Seed defines the random sequence — same seed, same sequence
  Be careful when re–opening an R workspace
- Other methods (not in this course)

Second step: Generating Unif$[a,b]$

- $U \sim \text{Unif}[0,1]$ can be transformed into $X \sim \text{Unif}[a,b]$ as

$$X = a + U \cdot (b - a)$$

- U can also be transformed into **discrete** uniform distribution on integers $\in \{1, \ldots, n\}$ as ($[\cdot]$, integer part)

$$X = [nU] + 1$$

Questions

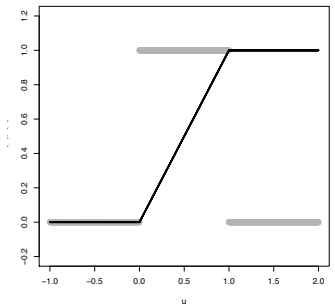
1. Why +1?
2. How can U be transformed into Y, where Y is discrete uniform on integers $(50, 55, 60)$?

Second step: Generating nonuniform random numbers

- $U \sim \text{Unif}(0,1)$
- Let F_U be the *cumulative distribution function* (CDF) of U

$$F_U(u) = P(U \leq u) = \begin{cases} 0 & u \leq 0 \\ u & 0 < u \leq 1 \\ 1 & 1 < u \end{cases}$$

- The *probability distribution function* (PDF) of U



$$f_U(u) = \begin{cases} 1 & 0 < u < 1 \\ 0 & u \notin (0,1) \end{cases}$$

Inverse CDF method

Let X be a random variable with CDF $X \sim F_X$ (F_X strictly increasing)

Consider $Y = F_X^{-1}(U)$, where $U \sim \text{Unif}(0,1)$

$$\begin{aligned} F_Y(y) &= P(Y \leq y) = P(F_X^{-1}(U) \leq y) \\ &= P(F_X(F_X^{-1}(U)) \leq F_X(y)) \\ &= P(U \leq F_X(y)) = F_U(F_X(y)) = F_X(y) \end{aligned}$$

Y has same probability distribution as X

Inverse CDF method

If we can generate $U \sim \text{Unif}(0,1)$, then

we can generate $X \sim F_X$ as

$$X = F_X^{-1}(U)$$

Provided we can calculate $F_X^{-1} \ldots$

Inverse CDF method: Example

Let $X \sim \exp(\lambda)$, i.e. with pdf

$$f_X(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

implying (**SHOW THIS**)

$$F_X(x) = \int_{-\infty}^{x} f_X(s)\mathrm{d}s = 1 - e^{-\lambda x}, \quad x \geq 0$$

QUESTIONS:
What is $F_X(x)$ for $x < 0$?
What is $\mathrm{E}[X]$?

Inverse CDF method: Example

Find F_X^{-1}

$$\begin{aligned} y &= 1 - e^{-\lambda x} \\ e^{-\lambda x} &= 1 - y \\ x &= -\tfrac{1}{\lambda}\ln(1-y) \\ F_X^{-1}(y) &= -\tfrac{1}{\lambda}\ln(1-y) \end{aligned}$$

Hence, if $U \sim \mathrm{U}(0,1)$, then

$$-\frac{1}{\lambda}\ln(1-U) = X \sim \exp(\lambda)$$

Inverse CDF method

1. When F_X^{-1} can be derived: **EASY**
2. When **NOT**: numerical solution

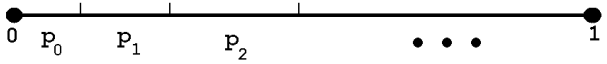
   time–consuming

   numerical errors ?

Situation 2 is common ... e.g. $\mathcal{N}(0,1)$

Generating discrete RVs

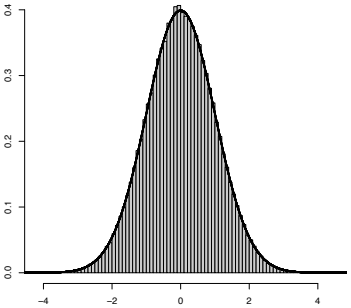
1. Define distribution $P(X = x_i) = p_i$
2. Generate $U \sim \text{Unif}(0, 1)$
3. If $U \leq p_0$, set $X = x_0$
4. Else if $U \leq p_0 + p_1$, set $X = x_1$
5. ...



Generating $\mathcal{N}(0, 1)$

Assume

- $\theta \in \text{Unif}(0, 2\pi)$
- $D \in \text{Unif}(0, 1)$



1: Generate θ, D
2: Generate X_1 and X_2 as

$$X_1 = \sqrt{-2 \ln D} \cos\theta$$
$$X_2 = \sqrt{-2 \ln D} \sin\theta$$

X_1 and X_2 are independent and normally distributed

But finding such transformations is not easy

Acceptance/rejection methods

- IDEA: generate $Y \sim f_Y$ similar to some known PDF f_X
- IDEA: f_Y is easy to generate from
- REQUIREMENT: there exists a constant c

$$\forall_x c f_Y(x) \geq f_X(x)$$

- f_Y: majorizing density, proposal density
- f_X: target density
- c: majorizing constant

Acceptance/rejection methods

1: **while** X not generated **do**
2: Generate $Y \sim f_Y$
3: Generate $U \sim \text{Unif}(0, 1)$
4: **if** $U \leq f_X(Y)/(c f_Y(Y))$ **then**
5: $X = Y$
6: Set X is generated
7: **end if**
8: **end while**

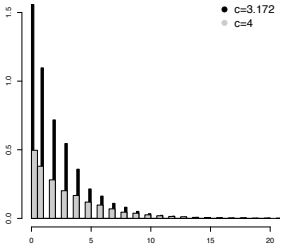
- $X \sim f_X$ **CHECK THIS**
- Larger c: larger rejection rates—c as small as possible
  number of draws $\sim$ Geometric$(1/c)$ mean: c
- Can work in higher dimensions—**but** high rejection rate

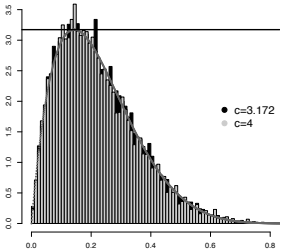
Acceptance/rejection methods: Example

Generate beta(2,7)

```
y<-dbeta(seq(0,2,0.0001),2,7)
c<-max(y);c
[1] 3.172554
```

1: **while** X not generated **do**
2: Generate $Y \sim \text{Unif}(0,1)$
3: Generate $U \sim \text{Unif}(0,1)$
4: **if** $U \leq dbeta(Y,2,7)/(c \cdot 1)$ **then**
5: $X = Y$
6: Set X is generated
7: **end if**
8: **end while**





QUESTION:
Compare acceptance and rejection regions (of Y) for different c.

Acceptance/rejection methods:

- Acceptance/rejection is difficult to apply
- Difficult to find majorizing density
  - can always take $\sup(f_X) \cdot \text{Unif}(0,1)$
  - but what is the problem?

Generating multivariate normal

Generate $\mathcal{N}(\vec{\mu}, \mathbf{\Sigma}) \in \mathbb{R}^n$

1: Generate n i.i.d. $\mathcal{N}(0,1)$ r.vs. $\vec{X} = (X_1, \ldots, X_n)$
{We know how to do this, see slide 16}
2: Compute Cholesky decomposition (a.k.a. matrix square root) of $\mathbf{\Sigma}$, i.e. find $\mathbf{A}$, lower triangular s.t. $\mathbf{A}\mathbf{A}^T = \mathbf{\Sigma}$,
{in R: `chol()` }
3: $\vec{Y} = \mu + \mathbf{A}\vec{X}$

QUESTION:
what is the expectation and variance–covariance of $\vec{Y}$?

Random numbers in R

1. `d`distribution name`()`: density of distribution
2. `p`distribution name`()`: CDF of distribution
3. `q`distribution name`()`: quantiles of distribution
4. `r`distribution name`()`: simulate from distribution

Summary

- Computers generate pseudo–random numbers
- We draw from pseudo–uniform and transform to desired distribution
- Analytical methods for transforming exist but are distribution specific

Monte Carlo Methods

732A90
Computational Statistics

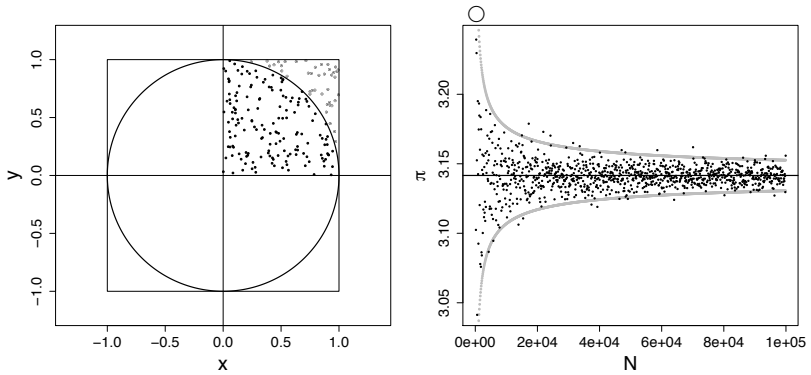
Krzysztof Bartoszek
(krzysztof.bartoszek@liu.se)

5 II 2019 (P42)
Department of Computer and Information Science
Linköping University

What is the area of the unit circle?

```
f.circArea<-function(N){
    m.xy<-cbind(runif(N),runif(N))
    4*sum(apply(m.xy,1,function(xy){xy[1]^2+xy
        [2]^2<1}))/N
}
```

$$3.141 \approx \pi = \int \mathbb{1} \mathrm{d}x$$



Monte Carlo methods: outline

- **Monte Carlo methods** are a class of computational algorithms that use repeated random sampling to compute their results.
- Monte Carlo methods for random number generation
  - Metropolis–Hastings algorithm
  - Gibbs sampler
- Monte Carlo methods for statistical inference
  - Estimate integrals (we already did!)
  - Variance estimation
  - Variance reduction: importance sampling, control variates

Markov Chain Monte Carlo

Previous lecture: Generate

- univariate distributions (inverse CDF, acceptance/rejection)
- multivariate normal

but general multivariate distribution?

MCMC

Bayesian inference: Recap

A dataset D is obtained by sampling from a distribution $f(\cdot|\theta)$. How to estimate θ?

- *Frequentists*: θ is an unknown but fixed parameter, compose likelihood $\mathcal{L}(D|\theta)$ and find θ that maximizes it.
- *Bayesians*: θ is a random variable with **prior** probability law $p(\theta)$ before observing D
- After observing D, Bayes' theorem gives

$$p(\theta|D) = \frac{p(D|\theta)p(\theta)}{p(D)} = \frac{p(D|\theta)p(\theta)}{\int p(D|\theta)p(\theta)\mathrm{d}\theta}$$

Bayesian inference: Recap

$$p(\theta|D) = \frac{p(D|\theta)p(\theta)}{p(D)} = \frac{p(D|\theta)p(\theta)}{\int p(D|\theta)p(\theta)\mathrm{d}\theta}$$

We know: $p(D|\theta)$ (the model), $p(\theta)$ (the prior)
We need: simulate from $p(\theta|D)$ (the posterior)

1. General (multivariate) type distribution
2. Integral can be impossible to compute

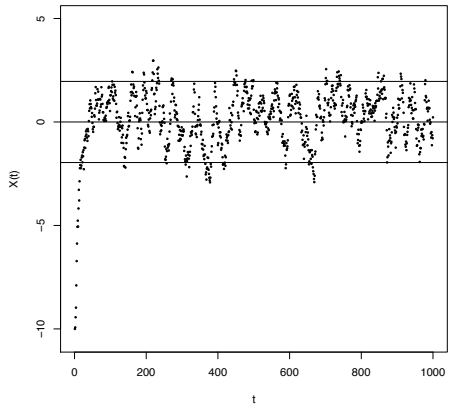
1. MCMC solves this
2. Not needed (given D it is constant)

Markov Chains: Recap

- A Markov chain is a sequence X_0, X_1, ... of random variables such that the distribution of the next value depends only on the current one (and parameters).
- $P(X_{t+1}|X_t)$ is called a **transition kernel**. Assume it does not depend on t (**time homogeneous**).
- A Markov chain is **stationary**, with stationary distribution Φ, if $\forall_k\ X_k \sim \Phi$
- One shows (not trivial in general) that under *certain* conditions a Markov chain will converge to the stationary distribution in the limit.

Markov Chains: Example

$$X(t+1) = e^{-1}X(t) + \epsilon \ , \epsilon \sim \mathcal{N}(0, \tfrac{5}{2} \cdot (1 - e^{-2}))$$



Discard first $K - 1$ samples: **burn–in period**

MCMC: Example

Linear regression with residual normally/student/etc. distributed

$$Y = \beta X + \epsilon$$

How to find credible interval for β if we know $\mathrm{Var}\,[\epsilon] = \sigma^2$?

1. $$P(Y|X, \beta) = \prod_{i=1}^{N} f(Y_i|\text{mean} = \beta X_i, \text{var} = \sigma^2)$$
2. Obtain $P(\beta|Y, X)$ by drawing from $P(Y|X, \beta)P(\beta)$ **in a clever way**.
3. The prior ?
4. Use the MCMC sample to obtain quantiles.

Normal residual: analytical solution

Metropolis–Hastings algorithm

We have

- A PDF $\pi(x)$ that we want to sample from.
- A **proposal distribution** $q(\cdot|X_t)$ that has a **regular** form w.r.t. to $\pi(\cdot)$
  E.g. $q(\cdot|X_t)$ is normal with mean X_t and given variance
- *Regular* form: suffices that the proposal has the same support as π.

Metropolis–Hastings Sampler

$$\alpha(X_t, Y) = \min\left\{1, \frac{\pi(Y)q(X_t|Y)}{\pi(X_t)q(Y|X_t)}\right\}$$

Initialize chain to X_0, $t = 0$
while $t < t_{\max}$ **do**
  Generate a candidate point $Y \sim q(\cdot|X_t)$
  Generate $U \sim Unif(0, 1)$
  if $U < \alpha(X_t, Y)$ **then**
    $X_{t+1} = Y$
  else
    $X_{t+1} = X_t$
  end if
  $t = t + 1$
end while

Metropolis–Hastings Sampler: Properties

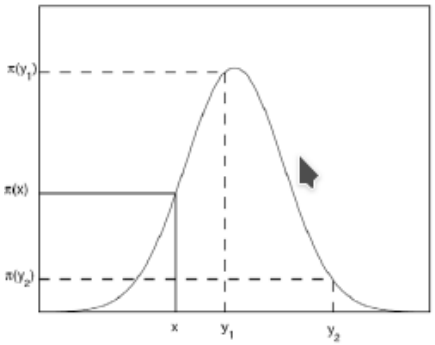
- Informally: "The chain $(X_t)_{t=0}^{\infty}$ will converge to $\pi(\cdot)$."
- The chain might not move sometimes.
- The values of the chain are dependent.
- If $q(X_t|Y) = q(Y|X_t)$ (i.e. symmetric proposal) we get **Random–walk Monte Carlo**:

$$\alpha(X_t, Y) = \min\left\{1, \frac{\pi(Y)}{\pi(X_t)}\right\}$$

Choice of proposal distribution

- In Random–Walk Monte Carlo

  If $\pi(Y) \geq \pi(X)$, the chain moves to the next point, otherwise only with some probability.

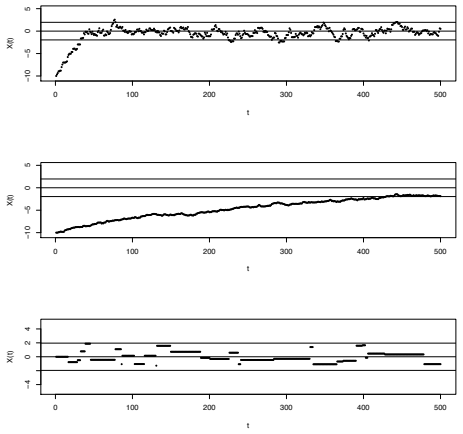


Choice of proposal dist.: **target**: $\pi(\cdot) = \mathcal{N}(0, 1)$

`732A90_ComputationalStatisticsVT2019_Lecture04codeSlide14.R`

Choice of proposal distribution

q normal with sd: `props=` 0.5, 0.1 and 20



Gibbs sampler: alternative to Metropolis–Hastings

We want to generate from a distribution on $\mathbb{R}^d$.

1: Initialize chain to $X_0 = (X_{0,1}, \ldots, X_{0,d})$, $t = 0$
2: **while** $t < t_{\max}$ **do**
3: **for** $i = 1, \ldots, d$ **do**
4: Generate

$$X_{t+1,i} \sim f(\cdot | X_{t+1,1}, \ldots, \mathbf{X_{t+1,i-1}}, \mathbf{X_{t,i+1}}, \ldots, X_{t,d})$$

5: **end for**
6: $t = t + 1$
7: **end while**

Gibbs sampler

- At each iteration inside the `for` loop univariate random numbers are generated.
- Only one element is updated.
- **WE NEED TO KNOW THE CONDITIONAL MARGINAL DISTRIBUTIONS**.
- Convergence may be slow.
- Can be useful in high dimensions (i.e. proposal density may be difficult to find in another way).

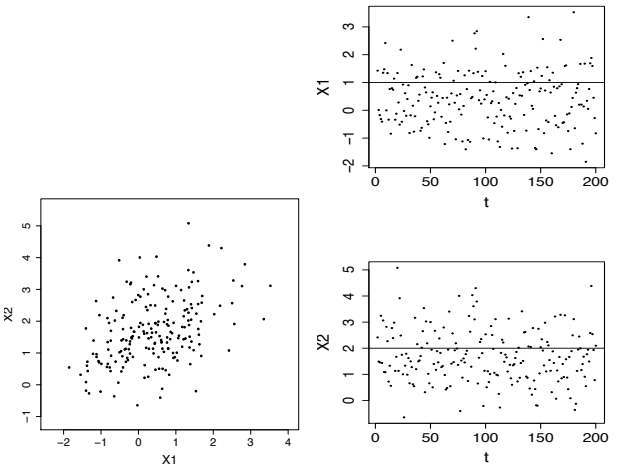
Gibbs sampler: target: d–dim $\mathcal{N}(\mu, \mathbf{\Sigma})$

`732A90_ComputationalStatisticsVT2019_Lecture04codeSlide18.R`

Gibbs sampler: Example (code: see R scripts)

Generate from

$$\mathcal{N}([1\ 2]^T, \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix})$$



Convergence monitoring

- When should we stop the chain? When are we (nearly) at the stationary distribution?
- Typically such a sample is generated to make further inference.

Convergence monitoring: Gelman–Rubin method

We want to estimate $v(\theta)$.

1. Generate k sequences of length n with different starting points.
2. Compute between– and within– sequence variances:

$$B = \frac{n}{k-1}\sum_{i=1}^{k}(\overline{v}_{i\cdot} - \overline{v}_{\cdot\cdot})^2 \quad W = \sum_{i=1}^{k}\frac{s_i^2}{k} \quad s_i^2 = \sum_{j=1}^{n}\frac{(\overline{v}_{ij} - \overline{v}_{i\cdot})^2}{n-1}$$

3. Overall variance estimate: $\hat{\mathrm{Var}}[v] = \frac{n-1}{n}W + \frac{1}{n}B$
4. Gelman–Rubin factor:

$$\sqrt{R} = \sqrt{\frac{\hat{\mathrm{Var}}[v]}{W}}$$

5. Values much larger than 1 indicate lack of convergence
6. See `?coda::gelman.diag`

Gibbs sampler

```
library(coda)
f1<-mcmc.list();f2<-mcmc.list();n<-100;k<-20
X1<-matrix(rnorm(n*k),ncol=k,nrow=n)
X2<-X1+(apply(X1,2,cumsum)*(matrix(rep(1:n,k),ncol=
    k)^2))
for (i in 1:k){f1[[i]]<-as.mcmc(X1[,i]);f2[[i]]<-as
    .mcmc(X2[,i])}
print(gelman.diag(f1))
# Potential scale reduction factors:
#      Point est. Upper C.I.
#[1,]       0.999       1.01

print(gelman.diag(f2))
# Potential scale reduction factors:
#      Point est. Upper C.I.
#[1,]       1.82       2.38
```

MC for inference

- Estimation of a definite integral

$$\theta = \int_D f(x)\mathrm{d}x \quad \left(\text{recall } \pi = \int_{\bigcirc} 1\mathrm{d}x\right)$$

- Decompose into:

$$f(x) = g(x)p(x) \quad \text{where} \quad \int_D p(x)\mathrm{d}x = 1$$

- Then, if $X \sim p(\cdot)$

$$\theta = \mathrm{E}[g(X)] = \int_D g(x)p(x)\mathrm{d}x$$

-

$$\hat{\theta} = \frac{1}{n}\sum_{i=1}^{n} g(x_i), \quad \forall_i x_i \sim p(\cdot)$$

MC for inference

- Decomposition is not unique, some will be better (lower variance) others worse. $p(x) \propto |f(x)|$: minimal
- Can we easily generate from $p(\cdot)$?
- Bayesian inference: use MCMC samples from $p(\theta|D)$ to obtain a point estimator

$$\theta^* = \int \theta p(\theta|D) \approx \frac{1}{n}\sum_{i=1}^{m} \theta_i$$

- $\hat{\theta}$ depends on n and $g(X)$, how variable will it be?

$$\widehat{\text{Var}}\left[\hat{\theta}\right] = \frac{1}{n(n-1)}\sum_{i=1}^{n}\left(g(x_i) - \overline{g(x)}\right)^2$$

- MCMC: estimator biases as chain correlated, use longer chain and batch mean instead of x_i.

Model Selection and Hypothesis Testing

732A90
Computational Statistics

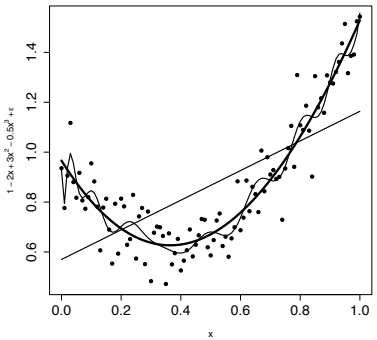
Krzysztof Bartoszek
(krzysztof.bartoszek@liu.se)

II 2019 ()
Department of Computer and Information Science
Linköping University

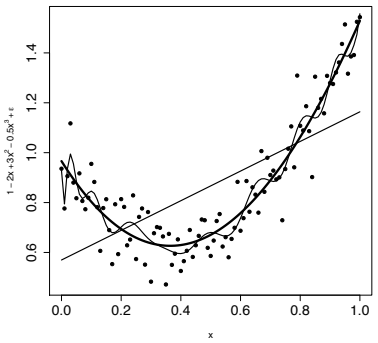
Summary

1. Generating data from a general multivariate distribution
2. Markov Chain Monte Carlo: Metropolis–Hastings algorithm, Gibbs sampling
3. Convergence: Gelman–Rubin method
4. Estimation of integral

Model selection



Model selection



Tools for model selection

- **Comparing different models**
- Information criteria (not this course)
- Cross–validation
- Hypothesis testing

- **Uncertainty estimation**
- Confidence intervals

Hypothesis testing: **Recap**

1. Assume a probabilistic model
   State a null hypothesis (H_0 e.g. no difference) and alternative (H_1 difference)

2. Observe data X
3. Calculate a test statistic e.g. $T(X) = (\overline{X})/(\widehat{\text{sd}(X)})$
   (different statistics will have different **efficiency** (power, ability to distinguish between hypotheses) associated with them)

4. Under H_0 $T(X)$ has "known" distribution
5. Decision: Is the value of $T(X)$ *surprising* (in the **critical region**)? If so reject H_0 in favour of H_1.

Hypothesis testing: Example

```
x<-rnorm(10,mean=4,sd=1)
```

Hypotheses:

$H_0 : \mu = 4, X \sim \mathcal{N}(\mu, \sigma^2)$
$H_1 : \mu \neq 4, X \sim \mathcal{N}(\mu, \sigma^2)$

Hypothesis testing: Example

```
x<-rnorm(10,mean=4,sd=1)
```

Hypotheses:

$H_0 : \mu = 4, X \sim \mathcal{N}(\mu, \sigma^2)$
$H_1 : \mu \neq 4, X \sim \mathcal{N}(\mu, \sigma^2)$

Test statistic

$T(x) = \frac{\overline{x}-\mu}{s/\sqrt{n}} \sim t(n-1)$

```
tx<-(mean(x)-4)/(sqrt(var(x)/length(x)))
t0<-qt(0.975,df=length(x)-1)
(tx>t0)||(tx<(-t0)) ## reject if TRUE
```

Hypothesis testing: Example

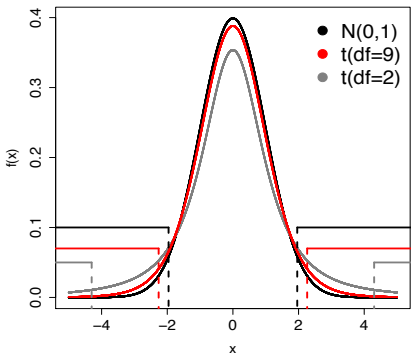
```
x<-rnorm(10,mean=4,sd=1)
```

Hypotheses:

$H_0 : \mu = 4, X \sim \mathcal{N}(\mu, \sigma^2)$
$H_1 : \mu \neq 4, X \sim \mathcal{N}(\mu, \sigma^2)$

Test statistic

$T(x) = \frac{\overline{x}-\mu}{s/\sqrt{n}} \sim t(n-1)$




```
tx<-(mean(x)-4)/(sqrt(var(x)/length(x)))
t0<-qt(0.975,df=length(x)-1)
(tx>t0)||(tx<(-t0)) ## reject if TRUE
```

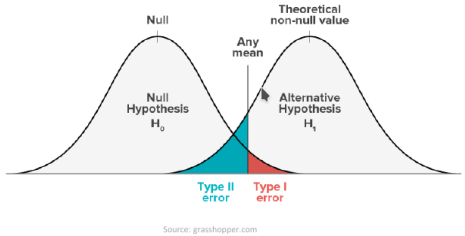
Hypothesis testing: Power

How does one compares different statistics?
POWER
Power = 1 − Type II error

Ability to correctly identify *surprise*,
i.e. indicate H_1.

How to compute power?

- Analytically (?)



- Generate data samples that satisfy H_1
  Compute percent of correct rejections

Monte Carlo Hypothesis testing

We may use "any" test statistic.
We do **not** need to know its distribution.

$H_0 : \mu = 4, X \sim \mathcal{N}(\mu, \sigma^2)$
$H_1 : \mu \neq 4, X \sim \mathcal{N}(\mu, \sigma^2)$

Monte Carlo Hypothesis testing

We may use "any" test statistic.
We do **not** need to know its distribution.

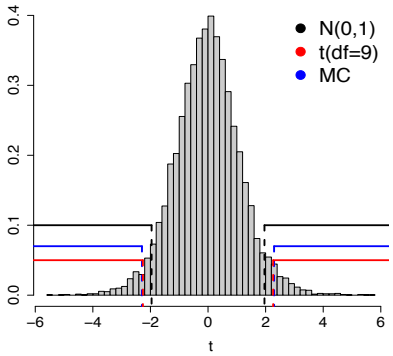
$H_0 : \mu = 4, X \sim \mathcal{N}(\mu, \sigma^2)$
$H_1 : \mu \neq 4, X \sim \mathcal{N}(\mu, \sigma^2)$

Test statistic

$T(x) = \frac{\overline{x}-\mu}{s/\sqrt{n}} \sim t(n-1)$

1: **for** $i = 1$ to B **do**
2: Generate $Y_1, \ldots, Y_n$ i.i.d. from H_0, i.e. $\mathcal{N}(\mathbf{4}, \sigma^2)$
3: Compute t_i from $Y_1, \ldots, Y_n$
4: **end for**
5: Use $t_1, \ldots, t_B$ to construct a histogram
6: Use the histogram as the distribution of $T(x)$ under H_0

Monte Carlo Hypothesis testing




```
x<-rnorm(10,4,1)
s<-var(x)
B<-10000
n<-length(x)
tsamp<-rep(NA,B)
for (i in 1:B){
    Y<-rnorm(n,4,s)
    tsamp[i]<-(mean(Y)-4)/(sd(Y)/sqrt(length(Y)))
}
hist(tsamp,breaks=50,col=gray(0.8),main="",xlab="t"
    ,ylab="",freq=FALSE,cex.axis=1.5,cex.lab=1.5)
```

Permutation tests

- A. k. a. randomization tests
- One solution if we do not know the distribution under H_0
- Computationally expensive
- Any sample size
- Two sample problem:
  - Population 1 distributed as F
  - Population 2 distributed as G
  - $H_0 : F = G$
    $H_1 : F \neq G$

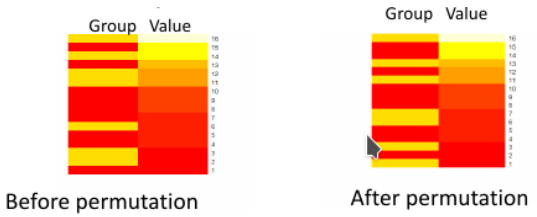
Permutation tests: mouse data

```
> t(mouse)
      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9]  [,10] [,11] [,12]
Group "y"   "z"   "z"   "y"   "y"   "z"   "y"   "y"   "y"   "y"   "z"   "z"
Value " 10" " 16" " 23" " 27" " 31" " 38" " 40" " 46" " 50" " 52" " 94" " 99"
      [,13] [,14] [,15] [,16]
Group "y"   "z"   "y"   "z"
Value "104" "141" "146" "197"
```

Do the values differ significantly between control and treatment groups?

Permutation tests

IDEA: If $F = G$ then **group label does not matter**
We may permute labels and still have a sample from F (or G)



Test statistic:

$$T(X) = \text{mean}(\text{values}|\text{group} = z) - \text{mean}(\text{values}|\text{group} = y)$$

Permutation test: scheme

1: $T(X)$ value of statistic from observed data

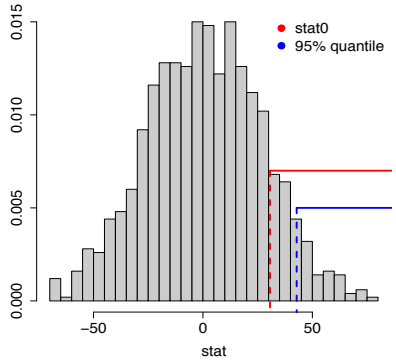
2: Create permutations $g_1^*, \ldots, g_B^*$ of group variable
{If the number of permutations is too large, sample B randomly **without** replacement. E.g. generate random permutations and keep only unique ones.}

3: Evaluate test statistic on each permutation

4: Estimate p–value: $\hat{p} = \#\{T(X_{g_b^*}) \geq T(X)\}/B$
5: If test is two–sided: $\hat{p} = \#\{|T(X_{g_b^*})| \geq |T(X)|\}/B$

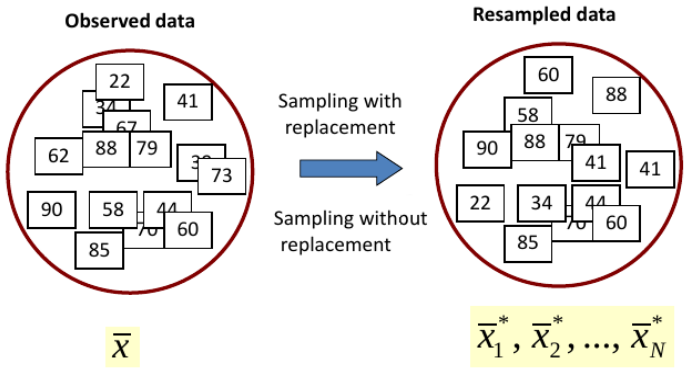
Permutation tests

Do we reject the null?




```
B=1000
stat=numeric(B)
n=dim(mouse)[1]
for(b in 1:B){
  Gb=sample(mouse$Group, n)
  stat[b]=mean(mouse$Value[Gb=='z'])-mean(mouse$
      Value[Gb=='y'])
}
stat0=mean(mouse$Value[mouse$Group=='z'])-mean(
    mouse$Value[mouse$Group=='y'])
print(c(stat0,mean(stat>stat0)))
## [1] 30.63492  0.12700
```

Resampling methods



Jackknife and bootstrap

Theory **different**, coding **similar**
Data (**i.i.d.**) $X \sim F(\cdot, w)$

1: Observed data: $D = (X_1, \ldots, X_n)$, estimator $\widehat{w} = T(D)$
2: **for** $i = 1, \ldots, B$ { Jackknife $B \leq n$} **do**
3: Generate

$D_i^* = (X_1^*, \ldots, X_n^*)$ by sampling with replacement
{**Nonparametric Bootstrap**, F unknown}

$D_i^* = X[-i]$ {**Jackknife**, F unknown}

$D_i^* = (X_1^*, \ldots, X_n^*)$ by generating from $F(\cdot, \hat{w})$
{**Parametric Bootstrap**, F known}

4: **end for**
5: Distribution of $\widehat{w}$ is estimated by $T(D_1^*), \ldots, T(D_B^*)$
{The histogram based on resampled values is used in place of the true density.}

Uncertainty estimation: confidence intervals

Estimate $100(1-\alpha)\%$ percentile confidence interval for w
$se(\cdot)$ is the square root of estimated variance (computationally heavy)
NOT by jackknife **TOO DEPENDENT!!**

1: Compute $T(D_1^*), \ldots, T(D_B^*)$
2: Sort in ascending order, obtaining $y_1, \ldots, y_B$
   {**percentile** method} **OR**
   Compute $y_i = (T(D_i^*) - T(D))/(se(T(D_i^*)))$ $i = 1, \ldots, B$
   {**t** method}
3: Define $A_1 = \lceil (B\alpha/2) \rceil$, $A_2 = \lfloor (B - B\alpha/2) \rfloor$
4: Confidence interval is given by
   (y_{A_1}, y_{A_2}) {**percentile** method} **OR**
   $(T(D) - se(T(D^*)) \cdot y_{A_1}, T(D) + se(T(D^*)) \cdot y_{A_2})$
   {**t** method}

Hypothesis testing: does statistic from observed data fall into CI (H_0) or not (H_1)

Uncertainty estimation: variance of estimator

Bootstrap

$$\widehat{\mathrm{Var}\,[T(\cdot)]} = \frac{1}{B-1}\sum_{i=1}^{B}\left(T(D_i^*) - \overline{T(D^*)}\right)^2$$

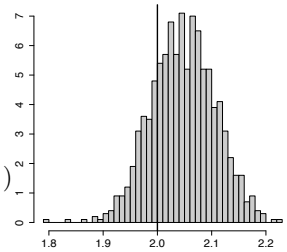
Jackknife ($n = B$)

$$\widehat{\mathrm{Var}\,[T(\cdot)]} = \frac{1}{n(n-1)}\sum_{i=1}^{n}(T_i^*) - J(T))^2\,,$$

where

$$T_i^* = nT(D) - (n-1)T(D_i^*) \qquad J(T) = \frac{1}{n}\sum_{i=1}^{n} T_i^*$$

Bootstrap in R




```
library("boot")
stat1<-function(data,vn){
    data<-as.data.frame(data[vn,])
    res<-lm(Response~Predictor,data)
    res$coefficients[2]
}
x<-rnorm(100);data<-cbind(Predictor=x,Response=3+2*
    x+rnorm(length(x),sd=0.5))
res<-boot(data,stat1,R=1000)
print(boot.ci(res))
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
##Based on 1000 bootstrap replicates
#Intervals :
#Level      Normal              Basic
#95%   ( 1.933,  2.164 )   ( 1.935,  2.162 )
# Level      Percentile            BCa
#95%   ( 1.934,  2.161 )   ( 1.936,  2.166 )
```

Bootstrap bias correction

1: Observed data: $D = (X_1, \ldots, X_n)$, estimator $\widehat{w} = T(D)$
2: **for** $i = 1, \ldots, B$ **do**
3: Generate

   $D_i^* = (X_1^*, \ldots, X_n^*)$ by sampling with replacement.
4: Calculate $T_i^* = T(D_i^*)$.
5: **end for**
6: Bias corrected estimator is

$$T_1 := 2T(D) - \frac{1}{B}\sum_{i=1}^{B} T_i^*.$$

Jackknife also has a bias correction method (see 2016 slides).

Comments

- Jackknife overestimate variance
- Bootstrap–t method is more accurate than percentile
- Permutations: sampling **without** replacement, bootstrap **with**
- Permutation p–value exact if all permutations used, bootstrap always approximate
- Bootstrap may be used for a wider class of problems
- Nonparametric bootstrap works badly for small samples ($n < 40$)
- Parametric bootstrap can work for small samples
- Bias corrections
- Methods do not require distributional assumptions

Summary

- Why are some models better than others?
- Hypothesis testing
- Monte Carlo hypothesis testing
- Resampling methods (permutations, jackknife, bootstrap)
- Simulation methods (parametric bootstrap)

Permutation tests for model selection

Data predictors: `X[,c(V1,V2)]`, response: `Y`
Model M relating Y and X

Competing models
H_0 variables `V1` should not be in M (smaller model)
H_1 all variables are significant
Test statistic: $T(M)$

Permutation test

1: **for** $i = 1 \dots B$ **do**
2: Obtain `V1`* by permuting order of columns in `V1`, fit model `Y=M(X[,c(V1`*`,V2)])`
3: Compute test statistic T_i for this model
4: **end for**
5: Compute p–value using above distribution of T

EM Algorithm, Stochastic Optimization

732A90
Computational Statistics

Krzysztof Bartoszek
(krzysztof.bartoszek@liu.se)

28 II 2018 (A37)
Department of Computer and Information Science
Linköping University

Stochastic and combinatorial optimization

- So far: Unconstrained optimization
  - Predictor variables are continuous
  - Response function is differentiable
- We discussed Steepest descent, Newton, BFGS, CG
- But: predictors can be discrete
  (scheduling problems, travelling salesman)
- But: outcome can be discrete, noisy or multi–modal

Stochastic and combinatorial optimization

Given a (large) set of states S, find

$$\min_{s \in S} f(s)$$

- Exhaustive search (shortest path algorithm)
- Often exhaustive search is NP–hard (TSP)
- Alternative: stochastic methods
  random search

Simulated annealing

Motivation from physics: cooling of metal

- Parameters:
  Energy of metal
  (decreasing, but not strictly monotonic)
  Temperature (decreasing)
- Aim: find global minimum energy

Simulated annealing

0. Set $k = 1$ and initialize state s.
1. Compute the temperature $T(k)$.
2. Set $i = 0$ and $j = 0$.
3. Generate a new state r and compute $\delta f = f(r) - f(s)$.
4. Based on δf, decide whether to move from state s to state r.
   If $\delta f \leq 0$,
   accept state r;
   otherwise,
   accept state r with a probability $P(\delta f, T(k))$.
   If state r is accepted, set $s = r$ and $i = i + 1$.
5. If i is equal to the limit for the number of successes at a given temperature, go to step 1.
6. Set $j = j + 1$. If j is less than the limit for the number of iterations at given temperature, go to step 3.

7. If $i = 0$,
   deliver s as the optimum; otherwise,
   if $k < k_{\max}$,
   set $k = k + 1$ and go to step 1;
   otherwise,
   issue message that
   'algorithm did not converge in $k_{\max}$ iterations'.

Simulated annealing

- https://www.youtube.com/watch?v=iaq_Fpr4KZc
- Generating new state:
  - Continuous: choose a new point a (random) distance from the current one
  - Discrete: similar or some rearrangement
- Selection probability: e.g $\exp(-\delta f(x)/T)$: decreasing with $f(x)$, increasing with T
- Temperature function: constant, proportional to k, or

$$T(k+1) = b(k)T(k), \quad b(k) = (\log(k))^{-1}$$

Remember: A smaller value is better than one on the path to the global minimum! Always keep track of smallest found.

Genetic algorithm

- Inspiration from evolutionary theory: survival of the fittest
- Variables=genotypes
- Observation=organism, characterized by genetic code
- State space=population of organisms
- Objective function=fitness of organism

New points are obtained from old points by crossover and mutation, the population only retains the fittest organisms (with better objective function).

https://en.wikipedia.org/wiki/List_of_genetic_algorithm_applications

Simulated annealing: TSP example

Assume constant temperature

1: Choose initial configuration $(Town_1, \ldots, Town_n)$
2: $k = 1$
3: **while** $k < k_{max} + 1$ **do**
4: Generate new configuration by rearrangement,

$$(1,2,3,4,5,6,7,8,9) \rightarrow (1,6,5,4,3,2,7,8,9)$$
$$(1,2,3,4,5,6,7,8,9) \rightarrow (1,7,8,2,3,4,5,6,9)$$

5: Measure difference in path length (δf) between old and new configuration
6: **if** shorter path found **then**
7: accept it
8: **else**
9: accept it with probability $P(\delta f)$
10: **end if**
11: $k++$
12: **end while**

Genetic algorithm

Encoding points

1. Enumerate each element of the state space, S
2. Code for observation i is binary representation of i (or something else)

Mutation and recombination rules

Generation k	Generation $k+1$
Crossover	
$x_i^{(k)}$ 11001001	$\rightarrow x_i^{(k+1)}$ 11011010
$x_j^{(k)}$ 00111010	
Inversion	
$x_i^{(k)}$ 11101011	$\rightarrow x_i^{(k+1)}$ 11010111
Mutation	
$x_i^{(k)}$ 11101011	$\rightarrow x_i^{(k+1)}$ 10111011
Clone	
$x_i^{(k)}$ 11101011	$\rightarrow x_i^{(k+1)}$ 11101011

Genetic algorithm

0. Determine a representation of the problem, and define an initial population, $x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}$. Set $k = 0$.
1. Compute the objective function (the "fitness") for each member of the population, $f(x_i^{(k)})$ and assign probabilities p_i to each item in the population, perhaps proportional to its fitness.
2. Choose (with replacement) a probability sample of size $m \leq n$. This is the reproducing population.
3. Randomly form a new population $x_1^{(k+1)}, x_2^{(k+1)}, \ldots, x_n^{(k+1)}$ from the reproducing population, using various mutation and recombination rules (see Table 6.2). This may be done using random selection of the rule for each individual of pair of individuals.
4. If convergence criteria are met, stop, and deliver $\arg\min_{x_i^{(k+1)}} f(x_i^{(k+1)})$ as the optimum; otherwise, set $k = k + 1$ and go to step 1.

Genetic algorithm: TSP example

Encoding and crossover

- Encode tours as $A_1, \ldots, A_n$ but

  Parent 1: FAB|ECGD Parent 2: DEA|CGBF
  Child: FAB|CGBF Child: DEA|ECGD

Instead

1. Remove FAB from DEACGBF $\longrightarrow$ DECG.
   Child becomes FABDECG.
2. Second child will be by taking prefix from Parent 2:
   DEAFBCG

Genetic algorithm: Mutations

- If a population is small and only crossover: the input domain becomes limited and may converge to a local minimum.
- Large initial populations are computationally heavy.
- Mutations allow one to explore more of S: jump out of local minimum.
- In TSP: mutation move a city in the tour to another position.
- Reproduction: Among m tours selected at step 2, two best are selected for reproduction, two worst replaced by children.
- If m is large, some tours might never be parents, global solution may be missed. Random chance of reproduction?
- Mutation probability is usually small (unless you want to jump wildly)

EM algorithm

Fundamental algorithm of computational statistics!

Model depends on the data which are observed (known) **Y** and **latent** (unobserved) data **Z**.

The data's (**both Y**'s and **Z**'s) distribution depends on some parameters θ.

AIM: Find MLE of θ.

- All data is known: Apply unconstrained optimization (discussed in Lecture 2)
- Unobserved data
  - **Sometimes** it is possible to look at the marginal distribution of the observed data.
  - Otherwise: **EM algorithm**

EM algorithm

Let

$$Q(\theta, \theta^k) = \int \log p(\mathbf{Y}, \mathbf{z}|\theta) p(\mathbf{z}|\mathbf{Y}, \theta^k) \mathrm{d}\mathbf{z} = \mathrm{E}\left[\mathrm{loglik}(\theta|\mathbf{Y}, \mathbf{Z})|\theta^k, \mathbf{Y}\right]$$

1: $k = 0$, $\theta^0 = \theta^0$
2: **while** Convergence not attained **and** $k < k_{max} + 1$ **do**
3: **E–step**: Derive $Q(\theta, \theta^k)$
4: **M–step**: $\theta^{k+1} = \mathrm{argmax}_\theta \ \ Q(\theta, \theta^k)$
5: $k++$
6: **end while**

Example: Normal data with missing values (but here analytical approach is also possible)

EM algorithm: R

```
> Y<-rnorm(100)
> Y[sample(1:length(Y),20,replace=FALSE)]<-NA
> EM.Norm(Y,0.0001,100)
[1]    1.0000    0.1000 -997.5705
[1]    0.1341894    1.3227095 -128.2789837
[1]   -0.03897274    1.38734070 -126.86036252
[1]   -0.07360517    1.39307050 -126.80801589
[1]   -0.08053165    1.39392861 -126.80593837
[1]   -0.08191695    1.39408871 -126.80585537
> mean(Y,na.rm=TRUE)
[1] -0.08226328
> var(Y,na.rm=TRUE)
[1] 1.411775
```

Notice: can be done by studying marginal distribution of observed data.

EM algorithm: R

`732A90_ComputationalStatisticsVT2019_Lecture06codeSlide15.R`

EM algorithm: Applications

Mixture models Z is a latent variable, $P(Z = k) = \pi_k$

- Mixed data comes from different sources (e.g. for regression, classification)
- Clustering
  1. Density in each cluster is normally distributed.
  2. Cluster label is latent (we do not know what are the chances an observation is from the given cluster)

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\vec{\mu}_k, \mathbf{\Sigma}_k) \quad \text{(informally)}$$

Direct MLE leads to numerical problems.
Introduce latent class variables and use EM.

EM algorithm: Gaussian mixtures

1. Initialize the means $\boldsymbol{\mu}_k$, covariances $\boldsymbol{\Sigma}_k$ and mixing coefficients π_k, and evaluate the initial value of the log likelihood.
2. **E step**. Evaluate the responsibilities using the current parameter values

$$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}. \tag{9.23}$$

3. **M step**. Re-estimate the parameters using the current responsibilities

$$\boldsymbol{\mu}_k^{\text{new}} = \frac{1}{N_k}\sum_{n=1}^{N} \gamma(z_{nk})\mathbf{x}_n \tag{9.24}$$

$$\boldsymbol{\Sigma}_k^{\text{new}} = \frac{1}{N_k}\sum_{n=1}^{N} \gamma(z_{nk})(\mathbf{x}_n - \boldsymbol{\mu}_k^{\text{new}})(\mathbf{x}_n - \boldsymbol{\mu}_k^{\text{new}})^{\text{T}} \tag{9.25}$$

$$\pi_k^{\text{new}} = \frac{N_k}{N} \tag{9.26}$$

where

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk}). \tag{9.27}$$

4. Evaluate the log likelihood

$$\ln p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\} \tag{9.28}$$

and check for convergence of either the parameters or the log likelihood. If the convergence criterion is not satisfied return to step 2.

$$Ez_{nk} = \gamma(z_{nk})$$

Source: Pattern recognition by Bishop

Summary

Random walk over the state space in search of minimum

1. Follow decreasing path
2. **BUT** with a certain probability go to higher values, to avoid local minima traps.
3. **Never forget** best found conformation!

4. Simulated annealing, Genetic algorithm, **EM algorithm**,
   Stochastic gradient descent (see 2016 slides)