Large-p-small-n-linear-regression

Wei Deng

Problem setup

```
options(digits=3)
set.seed(0)
```

Load (Install) the required libraries.

```
# install packages if you haven't installed them before:
# install.packages("invgamma")
# install.packages("rmutil")
# install.packages("scales")
library(MASS)
library(scales)
library(rmutil)
library(invgamma)
```

Problem setup for the large-p-small-n problem.

```
# total number of samples
n = 100
# total number of parameters
p = 1000
# initialize covariance matrix
cov = matrix(rep(0, p), nrow=p, ncol=p)
for (i in 1:p)
    for (j in 1:p)
        cov[i, j] = 0.6^(abs(i-j))

# initialize X
X = mvrnorm(n, rep(0, p), cov)
# true beta
beta = rep(0, p); beta[1:3] = c(3, 2, 1)
# noise to the output
epsilon = rnorm(n, 0, sqrt(3))
sd_par = 0.2
betas = beta
# generate response values
for (i in 1:n) {
    beta[1:3] = c(rnorm(1, 3, sd_par), rnorm(1, 2, sd_par), rnorm(1, 1, sd_par))
    Y = as.vector(X %*% beta + epsilon)
    betas = rbind(betas, beta)
}
x_prime_x = t(X) %*% X
bias = as.vector(t(X) %*% Y)
sample_init = rnorm(p, 0, 0.1)
```

Set hyperparameters, where most of them are non-informative.

```
# Set non-informative hyperparameter
nu = 1; lamda = 1
v1 = 10; a = 1; b = p
# Test different hyperparameters
v0 = 0.1; sd_default = 1;
```

Set the thinning factor, total number of iterations and parameters to store all the samples. We need to run long enough to make sure the algorithm is robust to different setups.

```
thinning = 100
iterations = 500000
total_parameters = iterations / thinning * p
```

Optimize hyperparameters via SGLD-SA

Initialize hyperparameters and starting points for SGLD-SA

```
theta = 0.5; d_star1 = rep(1, p); d_star0 = rep(1, p); p_star = rep(0.5, p);
sample = sample_init
sd = sd_default
x = X
y = Y
```

Build the log posterior function, and it is not used if we derive the closed-form gradient rather than the numerical gradient.

```
log_posterior = function(par) {
    sum(dnorm(Y, mean=as.vector(X %*% par), sd=sd, log=T)) +
    sum(dnorm(par, mean=0, sd=sd/sqrt(d_star1), log=T)) +
    sum(dlaplace(par, m=0, s=sd/d_star0, log=T)) +
    dinvgamma(sd^2, nu / 2, rate=lamda*nu/ 2, log=T)
}
```

Other setups

```
output_sgld_sa = matrix(rep(0, len=total_parameters), ncol=p)
```

Start training ...

```
for (ii in  1: iterations) {
    eta = ii^(-1/3) * 0.001
    decay = min(0.1, 1000 / (100 + ii)^0.7)
    # rand_idx = sample(n, size=50, replace=TRUE)
    # x = X[rand_idx, ]
    # y = Y[rand_idx]
    # x_prime_x = t(x) %*% x
    # bias = as.vector(t(x) %*% y)
    # gradient = numDeriv::grad(log_posterior, sample)
    # closed-formed gradient is faster than numerical gradient
    gradient = (-as.vector(x_prime_x %*% sample) + bias -
```

```
                  d_star1 * sample)/sd^2 - d_star0 * sign(sample) / sd
    gradient = gradient + rnorm(p, 0, 0.01)
    # sampling via SGLD
    sample = sample + eta * gradient + sqrt(2*eta) * rnorm(p, 0, 1)
    # hyperparameter optimization via stochastic approximation
    a_star = (dnorm(sample, mean=0, sd=sqrt(v1*sd^2)) * theta)
    b_star = (dlaplace(sample, m=0, s=v0*sd) * (1 - theta))
    p_star = (1 - decay) * p_star + decay * a_star / (a_star + b_star)
    d_star1 = (1 - decay) * d_star1 + decay * (p_star / v1)
    d_star0 = (1 - decay) * d_star0 + decay * ((1 - p_star) / v0)
    theta = (1 - decay) * theta + decay * (sum(p_star) + a - 1) / (a + b + p - 2)
    neg_B = sum(abs(sample) * d_star0)
    neg_four_AC = 4*(n+p+nu)*(sum((y - as.vector(x %*% sample))^2)
                              +sum(sample^2 * d_star1)+nu*lamda)
    root = (neg_B + sqrt(neg_B^2 + neg_four_AC))/(n+p+nu)/2
    sd = sqrt((1 - decay) * sd^2 + decay * root^2)
    if (ii %% thinning != 0)
        next
    if (ii %% 50000 == 0)
        print(paste0(ii*100.0/iterations, "% completed"))
    output_sgld_sa[ii / thinning, ] = sample
}
```

```
## [1] "10% completed"
## [1] "20% completed"
## [1] "30% completed"
## [1] "40% completed"
## [1] "50% completed"
## [1] "60% completed"
## [1] "70% completed"
## [1] "80% completed"
## [1] "90% completed"
## [1] "100% completed"
```

Optimize hyperparameters via SGLD-EM

Use the same hyperparameters and starting points for SGLD-EM

```
theta = 0.5; d_star1 = rep(1, p); d_star0 = rep(1, p); p_star = rep(0.5, p);
sample = sample_init
sd = sd_default
x = X
y = Y
output_sgld_em = matrix(rep(0, len=total_parameters), ncol=p)
```

Start training ...

```
for (ii in  1: iterations) {
    eta = ii^(-1/3) * 0.001
    # no-longer use the adaptive step size
    decay = 1
    gradient = (-as.vector(x_prime_x %*% sample) + bias -
```

```
                  d_star1 * sample)/sd^2 - d_star0 * sign(sample) / sd
    gradient = gradient + rnorm(p, 0, 0.01)
    # sampling via SGLD
    sample = sample + eta * gradient + sqrt(2*eta) * rnorm(p, 0, 1)
    # hyperparameter optimization via EM
    a_star = (dnorm(sample, mean=0, sd=sqrt(v1*sd^2)) * theta)
    b_star = (dlaplace(sample, m=0, s=v0*sd) * (1 - theta))
    p_star = (1 - decay) * p_star + decay * a_star / (a_star + b_star)
    d_star1 = (1 - decay) * d_star1 + decay * (p_star / v1)
    d_star0 = (1 - decay) * d_star0 + decay * ((1 - p_star) / v0)
    theta = (1 - decay) * theta + decay * (sum(p_star) + a - 1) / (a + b + p - 2)
    neg_B = sum(abs(sample) * d_star0)
    neg_four_AC = 4*(n+p+nu)*(sum((y - as.vector(x %*% sample))^2)
                              +sum(sample^2 * d_star1)+nu*lamda)
    root = (neg_B + sqrt(neg_B^2 + neg_four_AC))/(n+p+nu)/2
    sd = sqrt((1 - decay) * sd^2 + decay * root^2)
    if (ii %% thinning != 0)
        next
    if (ii %% 50000 == 0)
        print(paste0(ii*100.0/iterations, "% completed"))
    output_sgld_em[ii / thinning, ] = sample
}
```

```
## [1] "10% completed"
## [1] "20% completed"
## [1] "30% completed"
## [1] "40% completed"
## [1] "50% completed"
## [1] "60% completed"
## [1] "70% completed"
## [1] "80% completed"
## [1] "90% completed"
## [1] "100% completed"
```

Optimize hyperparameters via vanilla SGLD

Use the same hyperparameters and starting points for SGLD.

```
theta = 0.5; d_star1 = rep(1, p); d_star0 = rep(1, p); p_star = rep(0.5, p);
sample = sample_init
sd = sd_default
x = X
y = Y
output_sgld = matrix(rep(0, len=total_parameters), ncol=p)
```

Start training ...

```
for (ii in  1: iterations) {
    eta = ii^(-1/3) * 0.001
    gradient = (-as.vector(x_prime_x %*% sample) + bias -  d_star1 * sample)/sd^2
                  - d_star0 * sign(sample) / sd
    gradient = gradient + rnorm(p, 0, 0.01)
```

```
    # sampling via SGLD
    sample = sample + eta * gradient + sqrt(2*eta) * rnorm(p, 0, 1)
    if (ii %% thinning != 0)
        next
    if (ii %% 50000 == 0)
        print(paste0(ii*100.0/iterations, "% completed"))
    output_sgld[ii / thinning, ] = sample
}
```

```
## [1] "10% completed"
## [1] "20% completed"
## [1] "30% completed"
## [1] "40% completed"
## [1] "50% completed"
## [1] "60% completed"
## [1] "70% completed"
## [1] "80% completed"
## [1] "90% completed"
## [1] "100% completed"
```

#The third variable with true value 1 is hard to estimate. From the plot below, we see that SGLD-SA is more powerful than SGLD-EM to capture the third variable. You can try different seeds and #hyperparameters to compare it. #{r} #plot(output_sgld_sa[100:5000,3], pch=16, cex = .5, col="blue", # ylab="Parameter value", main="SGLD-SA v.s. SGLD-EM") #lines(output_sgld_em[100:5000,3], pch=16, cex = .5, col="yellow") #

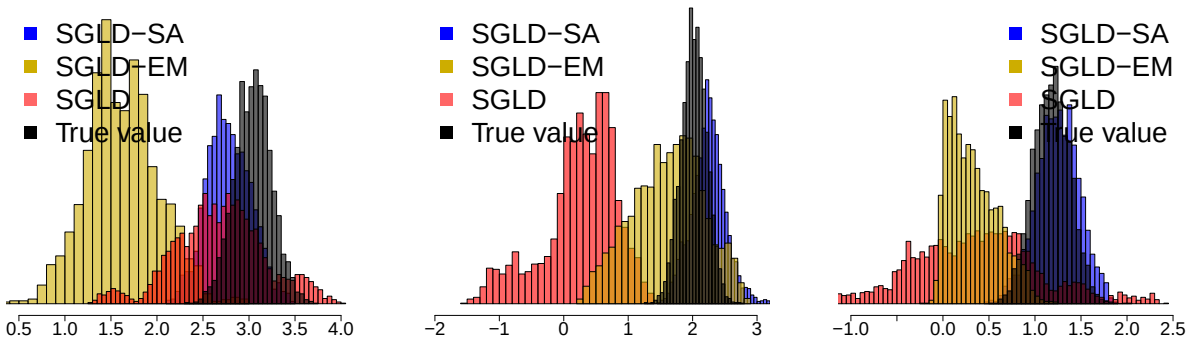
Burn in step

```
output_sgld_sa = output_sgld_sa[1000:5000,]
output_sgld_em = output_sgld_em[1000:5000,]
output_sgld = output_sgld[1000:5000,]
```

Evaluate the results

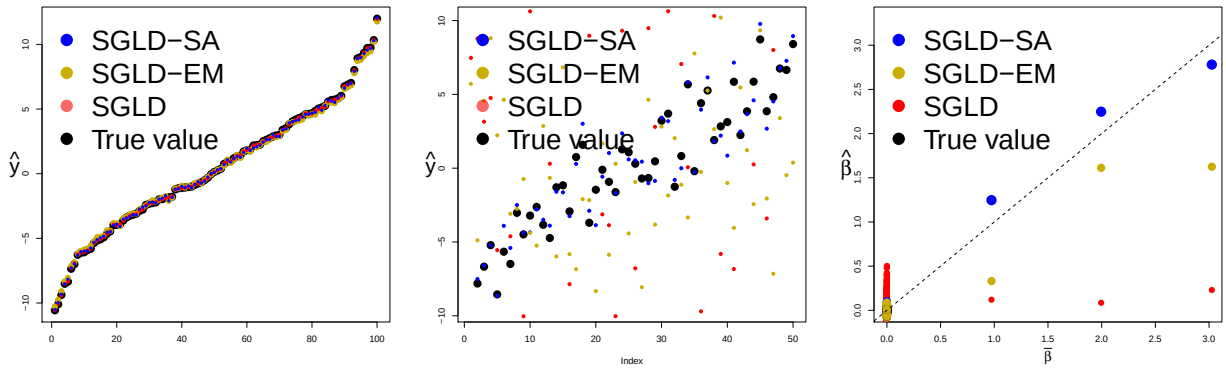
```
ltys = c(1,1,1)
trans = 0.6 # transpanrancy level
cols = c("blue", "gold3", alpha("red", trans), "black")
main_cex = 1.6
lagend_cex = 3
par(mfrow=c(1, 3))

hist(output_sgld_sa[, 1], 30, main="", xlim=c(0.5, 4), cex=1.5, ylim=c(0, 500),
     cex.main=main_cex, col=alpha("blue", trans), yaxt='n',
     ylab=NULL, xlab=NULL, cex.axis=2)
hist(rep(output_sgld_em[, 1], 1), 30, col=alpha("gold3", trans), add=TRUE)
hist(rep(output_sgld[,1], 1) + 2, 60, col=alpha("red", trans), add=TRUE)
hist(rnorm(dim(output_sgld_sa)[1]*1.0, beta[1], sd_par), 30,
     col=alpha("black", trans), add=TRUE)
legend("topleft", legend = c("SGLD-SA", "SGLD-EM", "SGLD", "True value"),
       pch=c(15, 15, 15, 15), col=cols, bty = "n", cex=lagend_cex)
```

```
hist(output_sgld_sa[, 2], 30, main="", xlim=c(-2, 3), cex=1.5, ylim=c(0, 500),
     cex.main=main_cex, col=alpha("blue", trans), yaxt='n',
     ylab=NULL, xlab=NULL, cex.axis=2)
hist(rep(output_sgld[,2], 1), 30, col=alpha("red", trans), add=TRUE)
hist(rep(output_sgld_em[, 2], 1), 30, col=alpha("gold3", trans), add=TRUE)
hist(rnorm(dim(output_sgld_sa)[1]*1.2, beta[2], sd_par), 30,
     col=alpha("black", trans), add=TRUE)
legend("topleft", legend = c("SGLD-SA", "SGLD-EM", "SGLD", "True value"),
       pch=c(15, 15, 15, 15), col=cols, bty = "n", cex=lagend_cex)

hist(output_sgld_sa[, 3], 30, main="", xlim=c(-1, 2.5), cex=1.5, ylim=c(0, 500),
     cex.main=main_cex, col=alpha("blue", trans), yaxt='n',
     ylab=NULL, xlab=NULL, cex.axis=2)
hist(rep(output_sgld[,3], 1), 60, col=alpha("red", trans), add=TRUE)
hist(rep(output_sgld_em[, 3], 1), 35, col=alpha("gold3", trans), add=TRUE)
hist(rnorm(dim(output_sgld_sa)[1], beta[3], sd_par), 30,
     col=alpha("black", trans), add=TRUE)
legend("topright", legend = c("SGLD-SA", "SGLD-EM", "SGLD", "True value"),
       pch=c(15, 15, 15, 15), col=cols, bty = "n", cex=lagend_cex)
```




```
par(mfrow=c(1, 3))

beta = colMeans(betas) # for convinience of plotting
# predictive distribution plot based on testing set
# generate testing set
point_size = 1.5
pred_optim = as.vector(x %*% beta)
pred_mcmc = as.vector(x %*% colMeans(output_sgld_sa))
pred_mcmc_em = as.vector(x %*% colMeans(output_sgld_em))
pred_non_sparse = as.vector(x %*% colMeans(output_sgld))
temp = rbind(y, seq(1, length(pred_optim)))
new_idx = temp[, order(temp[1, ])][2,]
# which in is to find the location of xx in a vector

plot(y[new_idx], main="", ylab="", type='p', pch=16, xlab="", cex=0,
     cex.main=point_size)
points(y[new_idx], col="black", type='p', pch=16, cex=2)
points(pred_mcmc_em[new_idx], col="gold3", type='p', pch=16, cex=1.5)
points(pred_non_sparse[new_idx], col="red", type='p', pch=16, cex=1)
```

```
points(pred_mcmc[new_idx], col="blue", type='p', pch=16, cex=0.8)
mtext(expression(hat(y)),side=2,las=1,line=2, cex=1.5)
legend("topleft", legend = c("SGLD-SA", "SGLD-EM", "SGLD", "True value"),
       pch=c(16,16,16,16), pt.cex=c(lagend_cex,lagend_cex,lagend_cex,lagend_cex),
       col=cols, bty = "n", cex=lagend_cex)


# predictive distribution plot based on testing set
# generate testing set
test_x = mvrnorm(n/2, rep(0, p), cov)
test_epsilon = rnorm(n/2, 0, sqrt(3))
test_y = as.vector(test_x %*% beta + test_epsilon)
# test_y = as.vector(test_x %*% beta)
pred_optim = as.vector(test_x %*% beta)
pred_mcmc = as.vector(test_x %*% colMeans(output_sgld_sa))
pred_mcmc_em = as.vector(x %*% colMeans(output_sgld_em))
pred_non_sparse = as.vector(test_x %*% colMeans(output_sgld))
temp = rbind(test_y, seq(1, length(pred_optim)))
new_idx = temp[, order(temp[1, ])][2,]

plot(test_y[new_idx], main="", type='p', ylab="", pch=16, cex=0, cex.main=point_size)
points(pred_non_sparse[new_idx], col="red", type='p', pch=16, cex=1)
points(pred_optim[new_idx], col="black", type='p', pch=16, cex=2)
points(pred_mcmc[new_idx], col="blue", type='p', pch=16, cex=1)
points(pred_mcmc_em[new_idx], col="gold3", type='p', pch=16, cex=1)
mtext(expression(hat(y)),side=2,las=1,line=2, cex=1.5)
legend("topleft", legend = c("SGLD-SA", "SGLD-EM", "SGLD", "True value"),
       pch=c(16,16,16,16), pt.cex=c(lagend_cex,lagend_cex,lagend_cex,lagend_cex),
       col=cols, bty = "n", cex=lagend_cex)


plot(beta[1:3], colMeans(output_sgld_sa[,1:3]), main="",
    ylab="", xlab=expression(bar(beta)), ylim=c(0, 3.3), xlim=c(0, 3),
    col="blue", type='p', pch=16, cex=2.5, cex.main=main_cex, cex.lab=1.5)
points(beta, colMeans(output_sgld)/3, col=alpha("red", 1), cex=1.5, type='p', pch=16)

points(beta, colMeans(output_sgld_sa), cex=2, type='p', pch=16, col="blue")
points(beta, colMeans(output_sgld_em), cex=2, type='p', pch=16, col="gold3")
mtext(expression(hat(beta)),side=2,las=1,line=2, cex=1.5)
for (i in 2:(p/2-1)) {
    points(beta[(2*i):(2*i+1)], colMeans(output_sgld_sa[, (2*i):(2*i+1)]),
           cex=2, type='p', pch=16, col="blue")
    points(beta[(2*i):(2*i+1)], colMeans(output_sgld_em[, (2*i):(2*i+1)]),
           cex=2, type='p', pch=16, col="gold3")
}

lines(seq(-1, 4, 0.5), seq(-1, 4, 0.5), type="l", lty=2)
legend("topleft", legend = c("SGLD-SA", "SGLD-EM", "SGLD", "True value"),
       pch = c(16, 16, 16, 16),
       pt.cex=c(lagend_cex,lagend_cex,lagend_cex),
       col=c("blue", "gold3", "red", "black"),
       text.col = c(1, 1), bty = "n", cex=lagend_cex)
```



Evaluate the testing performance based on MAE/MSE

```
round(c(mean(abs(test_y - test_x %*% colMeans(output_sgld_sa))),
        mean(abs(test_y - test_x %*% colMeans(output_sgld_em))),
        mean(abs(test_y - test_x %*% colMeans(output_sgld)))), 3)
```

```
## [1]  1.71  2.20 12.14
```

```
round(c(mean((test_y - test_x %*% colMeans(output_sgld_sa))^2),
        mean((test_y - test_x %*% colMeans(output_sgld_em))^2),
        mean((test_y - test_x %*% colMeans(output_sgld))^2)), 4)
```

```
## [1]   4.08   7.21 219.15
```