MTH 511a - 2020: Lecture 23

Instructor: Dootika Vats

The instructor of this course owns the copyright of all the course materials. This lecture material was distributed only to the students attending the course MTH511a: "Statistical Simulation and Data Analysis" of IIT Kanpur, and should not be distributed in print or through electronic media without the consent of the instructor. Students can make their own copies of the course materials for their use.

1 Stochastic optimization methods

We go back to optimization this week. The reason we took a break from optimization is because we will focus on stochastic optimization methods, which will lead the discussion into other stochastic methods.

We will cover two topics:

1. Stochastic gradient ascent
2. Simulated annealing

Our goal is the same as before: for an objective function $f(\theta)$, our goal is to find

$$\theta^* = \arg\max_{\theta} f(\theta)\,.$$

1.1 Stochastic gradient ascent

Recall, in order to maximize the objective function the gradient ascent algorithm does the following update:

$$\theta_{(k+1)} = \theta_{(k)} + t\nabla f(\theta_{(k)})\,,$$

where $\nabla f(\theta_{(k)})$ is the gradient vector. Now, since in many statistics problems, the objective function is the log-likelihood (for some density $\tilde{f}$),

$$f(\theta) = \frac{1}{n}\sum_{i=1}^{n} n\log \tilde{f}(\theta|x_i) \Rightarrow \nabla f(\theta) = \frac{1}{n}\sum_{i=1}^{n} \nabla\left[n\log \tilde{f}(\theta|x_i)\right]\,.$$

That is, in order to implement a gradient ascent step, the gradient of the log-likelihood is calculated for the whole data. However, consider the following two situations

- the data size n and/or dimension of θ are prohibitively large so that calculating the full gradient multiple times is infeasible
- the data is not available at once! In many online data situations, the full data set is not available, but comes in sequentially. Then, the full data gradient vector is not available.

In such situations, when the full gradient vector is unavailable, our goal is to *estimate* the gradient. Suppose i_k is a randomly chosen index in $\{1, \dots, n\}$. Then

$$\mathbb{E}\left[\nabla\left[n \log \tilde{f}(\theta|x_{i_k})\right]\right] = \frac{1}{n}\sum_{i=1}^{n}\left[n\nabla \log \tilde{f}(\theta|x_i)\right] .$$

Thus, $\nabla n \log\left[\tilde{f}(\theta|x_{i_k})\right]$ is an unbiased estimator of the complete gradient, but uses *only one data point*. Replacing the complete gradient with this estimate yields the *stochastic gradient ascent* update:

$$\theta_{(k+1)} = \theta_{(k)} + t\left[\nabla\left[n \log \tilde{f}(\theta_{(k)}|x_{i_k})\right]\right] ,$$

where i_k is a randomly chosen index. This randomness in choosing the index makes this a *stochastic algorithm*.

- advantage: it is much cheaper to implement since only one-data point is required for gradient evaluation
- disadvantage it may require larger k for convergence to the optimal solution
- disadvantage as k increases, $\theta_{(k+1)} \not\to \theta^*$. Rather, after some initial steps, $\theta_{(k+1)}$ oscillates around θ^*.

After K iterations, the final estimate of θ^* is

$$\hat{\theta}^* = \frac{1}{K}\sum_{k=1}^{K}\theta_{(k+1)} .$$

However, since each step involves evaluating only data gradient, variability in subsequent updates of $\theta_{(k+)}$ increases. To stabilize this behavior, often `mini-batch` stochastic gradient is used.

1.1.1 Mini-batch stochastic gradient ascent

Let I_k be a random subset of $\{1, \dots, n\}$ of size b. Then, the mini-batch stochastic gradient ascent algorithm implements the following update:

$$\theta_{(k+1)} = \theta_{(k)} + t \left[\frac{1}{b} \sum_{i \in I_k} \nabla \left[n \log \tilde{f}(\theta_{(k)} | x_i) \right] \right] .$$

The mini-batch stochastic gradient estimate of θ^* after K updates is

$$\hat{\theta}^* = \frac{1}{K} \sum_{k=1}^{K} \theta_{(k)} .$$

There are not a lot of clear rules about terminating the algorithm in stochastic gradient. Typically, the number of iterations $K = n$, so that one full pass at the data is implemented.

1.1.2 Logistic regression

Recall the logistic regression setup where for a response Y and a covariate matrix X,

$$Y_i \sim \text{Bern} \left(\frac{e^{x_i^T \beta}}{1 + e^{x_i^T \beta}} \right) .$$

In order to find the MLE for β, we obtain the log-likelihood.

$$L(\beta | Y) = \prod_{i=1}^{n} (p_i)^{y_i} (1 - p_i)^{1 - y_i}$$

$$\Rightarrow n \log \tilde{f}(\beta) = -n \sum_{i=1}^{n} \log \left(1 + \exp(x_i^T \beta) \right) + n \sum_{i=1}^{n} y_i x_i^T \beta$$

Taking derivative:

$$\nabla \left[n \log \tilde{f}(\beta) \right] = n \sum_{i=1}^{n} x_i \left[y_i - \frac{e^{x_i^T \beta}}{1 + e^{x_i^T \beta}} \right] .$$

As noted earlier, the target objective is concave, thus a global optima exists and the gradient ascent algorithm will converge to the MLE. We will implement the stochastic gradient ascent algorithm here.

```r
#################################################
## MLE for logistic regression
## Using stochastic gradient ascent
#################################################

f.gradient <- function(y, X, beta)
{
  n <- dim(X)[1]
  beta <- matrix(beta, ncol = 1)
  pi <- exp(X %*% beta) / (1 + exp(X%*%beta))
  rtn <- colSums(X* as.numeric(y - pi))
  return(n*rtn)
}


#################################################
## The following is a general function that
## implements the regular gradient ascent
## the stochastic gradient ascent and
## mini-batch stochastic gradient ascent
#################################################
SGA <- function(y, X, batch.size = dim(X)[1], t = .1, max.iter = dim(X)[1],
    adapt = FALSE)
{
  p <- dim(X)[2]
  n <- dim(X)[1]

  # create the mini-batches
  permutation <- sample(1:n, replace = FALSE)
  K <- floor(n/batch.size)
  batch.index <- split(permutation, rep(1:K, length = n, each = n/K))

  # index for choosing the mini-batch
  count <- 1

  beta_k <- rep(0, p) # start at all 0s
  track.gradient <- matrix(0, nrow = max.iter, ncol = p)
  track.gradient[1,] <- f.gradient(y = y, X= X, beta = beta_k)

  # saving the running mean of the estimates of theta^*
  mean_beta <- rep(0,p)

  # tk: in case we want t_k
  tk <- t

  # ideally, we will have a while loop here, but
  # I have written this to always complete some max.iter steps
```

```
  for(iter in 1:max.iter)
  {
    count <- count+1

    if(adapt) tk <- t/(sqrt(iter)) # in case t_k
    if(count %% K == 0) count <- count%%K +1 # when all batches finish,
        restart the batches
    if(iter %% (max.iter/10) == 0) print(iter) #feedback

    # batch of data
    y.batch <- y[batch.index[[count]] ]
    X.batch <- matrix(X[batch.index[[count]], ], nrow = batch.size)

    # SGA step
    beta_k = beta_k + tk* f.gradient(y = y.batch, X = X.batch, beta =
        beta_k)/batch.size

    # saving overall estimates and running gradients for demonstration
    mean_beta <- (beta_k + mean_beta*(iter - 1))/(iter)
    if(batch.size == n)
    {
      est <- beta_k
    }else{
      est <- mean_beta
    }
    track.gradient[iter,] <- f.gradient(y = y, X = X, beta = est)/n
  }
  rtn <- list("iter" = iter, "est" = est, "grad" = track.gradient[1:iter,])
  return(rtn)
}
```

Next, I will generate data from the logistic regression model in order to demonstrate the performance of the stochastic gradient ascent algorithm.

```
# Generating data for demonstration
set.seed(10)
p <- 5
n <- 1e4
X <- matrix(rnorm(n*(p-1)), nrow = n, ncol = p-1)
X <- cbind(1, X)
beta <- matrix(rnorm(p, 0, sd = 1), ncol = 1)
p <- exp(X %*% beta)/(1 + exp(X%*%beta))
y <- rbinom(n, size = 1, prob = p)
```

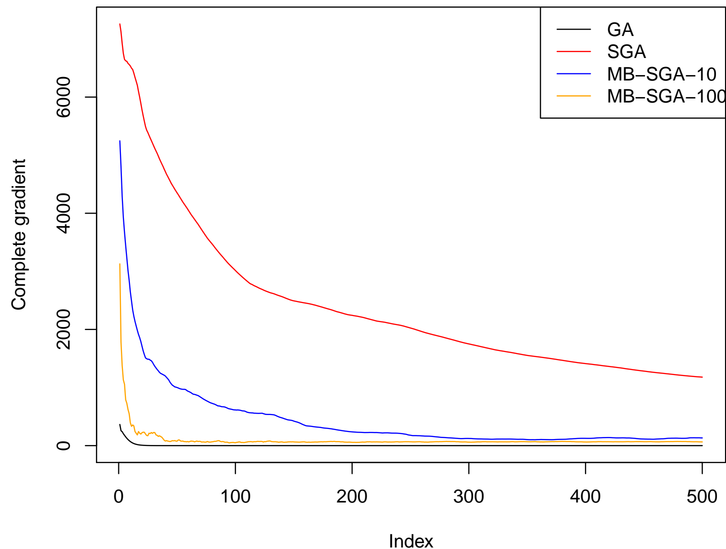
We implement the stochastic gradient ascent algorithm and keep a track of the running average of the estimate of θ^*: $\hat{\theta}^*_k$ and track the value of the complete gradient at that

value $\|\nabla f(\hat{\theta}_k^*$. A running plot of these should tend to 0 as k increases. We will implement the original gradient ascent, stochastic gradient ascent, and mini-batch gradient ascent algorithm.

I first run this for tuned values of t.

```
# Tuned value of t
ga <- SGA(y, X, batch.size = 1e4, t = .0015, max.iter = 1e3)
b1 <- SGA(y, X, batch.size = 1, t = .1, max.iter = ga$iter)
b10 <- SGA(y, X, batch.size = 10, t = .1, max.iter = ga$iter)
b100 <- SGA(y, X, batch.size = 100, t = .1, max.iter = ga$iter)

index <- 1:500
plot(apply(ga$grad[index,], 1, function(t) sum(abs(t))), type = 'l', ylim =
    c(0,max(apply(b1$grad[,], 1, function(t) sum(abs(t))))), ylab =
    "Complete gradient")
lines(apply(b1$grad[index,], 1, function(t) sum(abs(t))), col = "red")
lines(apply(b10$grad[index,], 1, function(t) sum(abs(t))), col = "blue")
lines(apply(b100$grad[index,], 1, function(t) sum(abs(t))), col = "orange")
legend("topright", col = c("black", "red", "blue", "orange"), legend =
    c("GA", "SGA", "MB-SGA-10", "MB-SGA-100"), lty = 1)
```



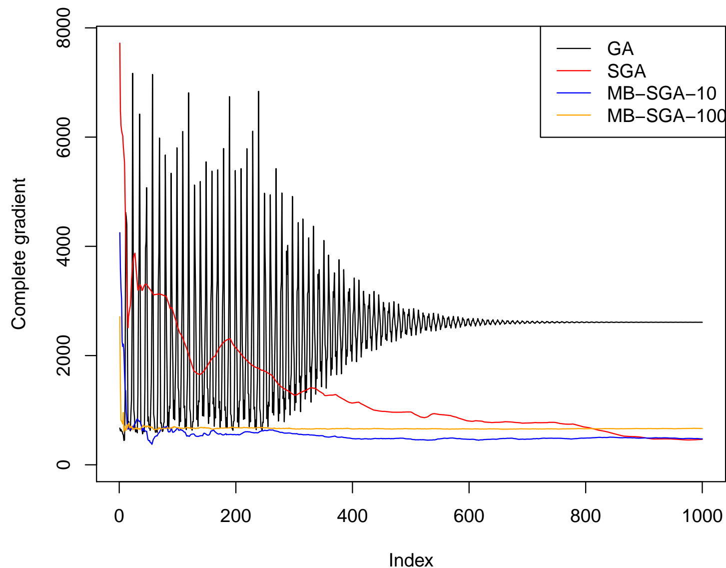
We see that the original stochastic gradient ascent algorithm is slow to converge although it is the cheapest. The mini-batches are far more stable.

Next, we repeat the same algorithm with different choices of t. These chosen choices of t are too large so that all the algorithm no longer perform well and essentially oscillate

locally.

```r
# all bad values of t
ga <- SGA(y, X, batch.size = n, t = .005, max.iter = 1e3)
b1 <- SGA(y, X, batch.size = 1, t = 1, max.iter = ga$iter)
b10 <- SGA(y, X, batch.size = 10, t = 1, max.iter = ga$iter)
b100 <- SGA(y, X, batch.size = 100, t = 1, max.iter = ga$iter)

index <- 1:1000
plot(apply(ga$grad[index,], 1, function(t) sum(abs(t))), type = 'l', ylim =
    c(0,max(apply(b1$grad[,], 1, function(t) sum(abs(t))))), ylab =
    "Complete gradient")
lines(apply(b1$grad[index,], 1, function(t) sum(abs(t))), col = "red")
lines(apply(b10$grad[index,], 1, function(t) sum(abs(t))), col = "blue")
lines(apply(b100$grad[index,], 1, function(t) sum(abs(t))), col = "orange")
legend("topright", col = c("black", "red", "blue", "orange"), legend =
    c("GA", "SGA", "MB-SGA-10", "MB-SGA-100"), lty = 1)
```

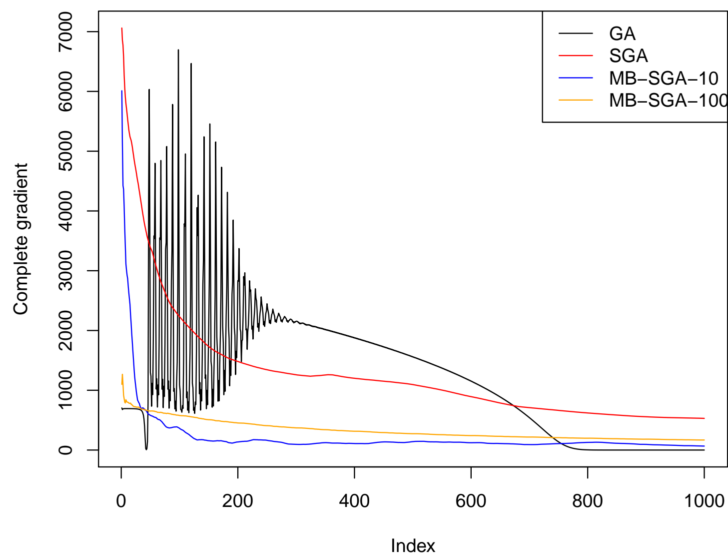


The original gradient ascent algorithm oscillates drastically. The stochastic versions seem to be converging away from 0 as well. Thus, the value of t is critical to implementing the (stochastic) gradient ascent algorithms in a stable way.

Note that the oscillations occurs due to a large value of t. But which value of t which be large and which will be small is difficult to assess in the beginning. It is thus useful to choose a decreasing sequence t_k that reduces the step size and avoids long durations of getting stuck in an oscillation. Here we will use $t_k = t/\log(k)$.

```
ga <- SGA(y, X, batch.size = n, t = .05, max.iter = 1e3, adapt = TRUE)
b1 <- SGA(y, X, batch.size = 1, t = 1, max.iter = ga$iter, adapt = TRUE)
b10 <- SGA(y, X, batch.size = 10, t = 1, max.iter = ga$iter, adapt = TRUE)
b100 <- SGA(y, X, batch.size = 100, t = 1, max.iter = ga$iter, adapt = TRUE)

index <- 1:1000
plot(apply(ga$grad[index,], 1, function(t) sum(abs(t))), type = 'l', ylim =
    c(0,max(apply(b1$grad[,], 1, function(t) sum(abs(t))))), ylab =
    "Complete gradient")
lines(apply(b1$grad[index,], 1, function(t) sum(abs(t))), col = "red")
lines(apply(b10$grad[index,], 1, function(t) sum(abs(t))), col = "blue")
lines(apply(b100$grad[index,], 1, function(t) sum(abs(t))), col = "orange")
legend("topright", col = c("black", "red", "blue", "orange"), legend =
    c("GA", "SGA", "MB-SGA-10", "MB-SGA-100"), lty = 1)
```



Note here that although the algorithm begins oscillate, due to decreasing step-sizes, the algorithm escapes out of local oscillations.

2 Questions to think about

1. Can we change t_k adaptively as the algorithm goes along?
2. Try and implement a stochastic Newton-Raphson algorithm following the same

lines of reasoning as discussed here.