Hints for Lab 8 (GD and SGD)

Yanfei Kang

3/28/2020

Optimize α in GD

1. Improve the R function graddesc.lm(), try to optimize α in each step, instead of setting it to a constant.

In the following function, for each iteration, we can use optimization methods to optimize α.

```
gd.lm <- function(X, y, beta.init, alpha, tol = 1e-05, max.iter = 100) {
    beta.old <- beta.init
    J <- betas <- list()
    if (alpha == "auto") {
        alpha <- optim(0.1, function(alpha) {
            lm.cost(X, y, beta.old - alpha * lm.cost.grad(X, y, beta.old))
        }, method = "L-BFGS-B", lower = 0, upper = 1)
        if (alpha$convergence == 0) {
            alpha <- alpha$par
        } else {
            alpha <- 0.1
        }
    }
    betas[[1]] <- beta.old
    J[[1]] <- lm.cost(X, y, beta.old)
    beta.new <- beta.old - alpha * lm.cost.grad(X, y, beta.old)
    betas[[2]] <- beta.new
    J[[2]] <- lm.cost(X, y, beta.new)
    iter <- 0
    while ((abs(lm.cost(X, y, beta.new) - lm.cost(X, y, beta.old)) > tol) & (iter <
        max.iter)) {
        beta.old <- beta.new
        if (alpha == "auto") {
            alpha <- optim(0.1, function(alpha) {
                lm.cost(X, y, beta.old - alpha * lm.cost.grad(X, y, beta.old))
            }, method = "L-BFGS-B", lower = 0, upper = 1)
            if (alpha$convergence == 0) {
                alpha <- alpha$par
            } else {
                alpha <- 0.1
            }
        }
        beta.new <- beta.old - alpha * lm.cost.grad(X, y, beta.old)
        iter <- iter + 1
        betas[[iter + 2]] <- beta.new
        J[[iter + 2]] <- lm.cost(X, y, beta.new)
```

```
    }
    if (abs(lm.cost(X, y, beta.new) - lm.cost(X, y, beta.old)) > tol) {
        cat("Could not converge. \n")
    } else {
        cat("Converged. \n")
        cat("Iterated", iter + 1, "times.", "\n")
        cat("Coef: ", beta.new, "\n")
        return(list(coef = betas, cost = J, niter = iter + 1))
    }
}
## Make the cost function
lm.cost <- function(X, y, beta) {
    n <- length(y)
    loss <- sum((X %*% beta - y)^2)/(2 * n)
    return(loss)
}
## Calculate the gradient
lm.cost.grad <- function(X, y, beta) {
    n <- length(y)
    (1/n) * (t(X) %*% (X %*% beta - y))
}
```

Let us now generate some data and compare functions with and without optimized α.

```
## Generate some data
set.seed(20200401)
beta0 <- 1
beta1 <- 3
sigma <- 1
n <- 10000
x <- rnorm(n, 0, 1)
y <- beta0 + x * beta1 + rnorm(n, mean = 0, sd = sigma)
X <- cbind(1, x)
gd.auto <- gd.lm(X, y, beta.init = c(0, 0), alpha = "auto", tol = 1e-05, max.iter = 10000)
```

```
## Converged.
## Iterated 3 times.
## Coef:  0.995839 2.998762
```

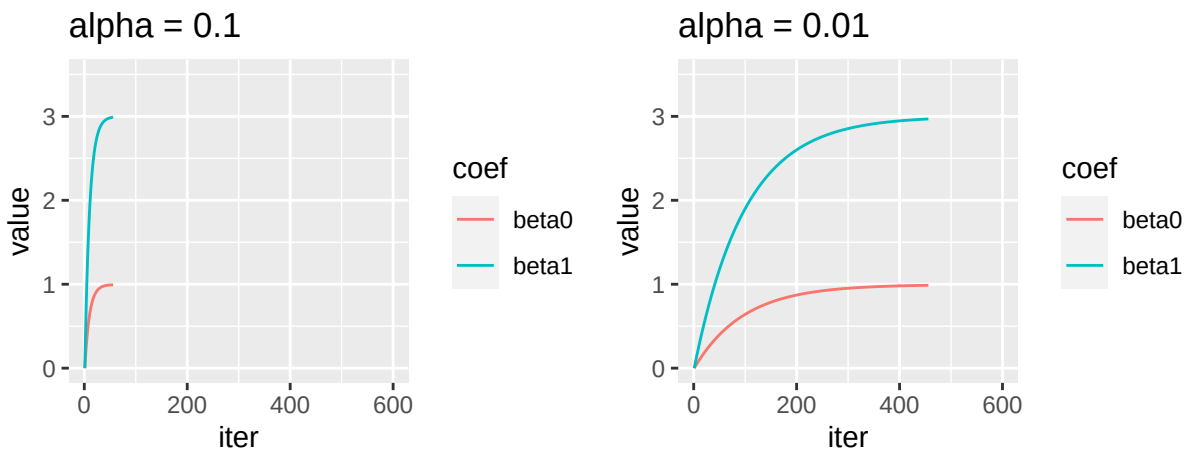
```
gd1 <- gd.lm(X, y, beta.init = c(0, 0), alpha = 0.1, tol = 1e-05, max.iter = 10000)
```

```
## Converged.
## Iterated 55 times.
## Coef:  0.9934615 2.99014
```

```
betas <- as.data.frame(t(do.call(cbind, gd1$coef)))
colnames(betas) <- c("beta0", "beta1")
betas <- betas %>% mutate(iter = 1:nrow(betas))
betas <- melt(betas, id.vars = "iter", variable.name = "coef")
p1 <- ggplot(betas, aes(iter, value)) + geom_line(aes(colour = coef)) + ylim(c(0,
    3.5)) + ggtitle("alpha = 0.1") + xlim(c(0, 600))
gd2 <- gd.lm(X, y, beta.init = c(0, 0), alpha = 0.01, tol = 1e-05, max.iter = 10000)
```

```
## Converged.
## Iterated 455 times.
## Coef:  0.9872457 2.969085
```

```
betas <- as.data.frame(t(do.call(cbind, gd2$coef)))
colnames(betas) <- c("beta0", "beta1")
betas <- betas %>% mutate(iter = 1:nrow(betas))
betas <- melt(betas, id.vars = "iter", variable.name = "coef")
p2 <- ggplot(betas, aes(iter, value)) + geom_line(aes(colour = coef)) + ylim(c(0,
    3.5)) + ggtitle("alpha = 0.01") + xlim(c(0, 600))
niter <- data.frame(alpha = c("auto", 0.1, 0.01), niter = c(gd.auto$niter, gd1$niter,
    gd2$niter))
p1 + p2
```




```
knitr::kable(niter) %>% kable_styling(bootstrap_options = "striped", full_width = F)
```

alpha	niter
auto	3
0.1	55
0.01	455

We find that using optimized α improves iteration efficiency significantly, while using smaller learning rates yields larger number of iterations.

SGD in R

2. Write a function in R for Stochastic Gradient Descent for linear regression, and test your function in the Bodyfat data.

Recall the SGD iteration. Repeat the following until convergence {

$$\beta := \beta - \alpha \nabla J(\beta)_i.$$

}

```
sgd.lm <- function(X, y, beta.init, alpha = 0.5, n.samples = 1, tol = 1e-05, max.iter = 100) {
    n <- length(y)
    beta.old <- beta.init
    J <- betas <- list()
    sto.sample <- sample(1:n, n.samples, replace = TRUE)
    betas[[1]] <- beta.old
    J[[1]] <- lm.cost(X, y, beta.old)
```

```
    beta.new <- beta.old - alpha * sgd.lm.cost.grad(X[sto.sample, ], y[sto.sample],
        beta.old)
    betas[[2]] <- beta.new
    J[[2]] <- lm.cost(X, y, beta.new)
    iter <- 0
    n.best <- 0
    while ((abs(lm.cost(X, y, beta.new) - lm.cost(X, y, beta.old)) > tol) & (iter +
        2 < max.iter)) {
        beta.old <- beta.new
        sto.sample <- sample(1:n, n.samples, replace = TRUE)
        beta.new <- beta.old - alpha * sgd.lm.cost.grad(X[sto.sample, ], y[sto.sample],
            beta.old)
        iter <- iter + 1
        betas[[iter + 2]] <- beta.new
        J[[iter + 2]] <- lm.cost(X, y, beta.new)
    }
    if (abs(lm.cost(X, y, beta.new) - lm.cost(X, y, beta.old)) > tol) {
        cat("Could not converge. \n")
    } else {
        cat("Converged. \n")
        cat("Iterated", iter + 1, "times.", "\n")
        cat("Coef: ", beta.new, "\n")
        return(list(coef = betas, cost = J, niter = iter + 1))
    }
}

## Make the cost function
sgd.lm.cost <- function(X, y, beta) {
    n <- length(y)
    if (!is.matrix(X)) {
        X <- matrix(X, nrow = 1)
    }
    loss <- sum((X %*% beta - y)^2)/(2 * n)
    return(loss)
}
## Calculate the gradient
sgd.lm.cost.grad <- function(X, y, beta) {
    n <- length(y)
    if (!is.matrix(X)) {
        X <- matrix(X, nrow = 1)
    }
    t(X) %*% (X %*% beta - y)/n
}
```

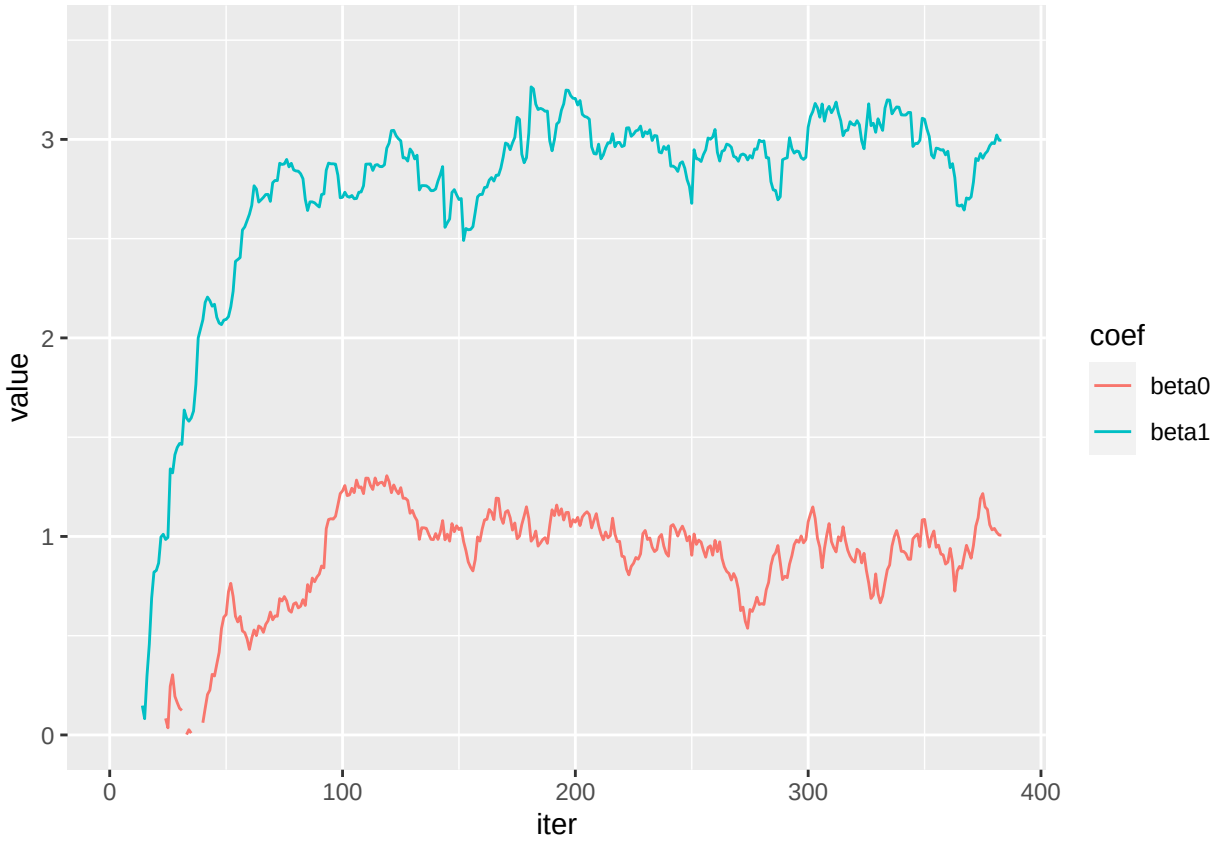
Let us test SGD on generated data.

```
# test on the generated data
sgd.est <- sgd.lm(X, y, beta.init = c(-4, -5), alpha = 0.05, tol = 1e-05, max.iter = 10000)
```

```
## Converged.
## Iterated 382 times.
## Coef:  1.006272 2.995898
```

```
betas <- as.data.frame(t(do.call(cbind, sgd.est$coef)))
colnames(betas) <- c("beta0", "beta1")
```

```
betas <- betas %>% mutate(iter = 1:nrow(betas))
betas <- melt(betas, id.vars = "iter", variable.name = "coef")
ggplot(betas, aes(iter, value)) + geom_line(aes(colour = coef)) + ylim(c(0, 3.5))
```



We find that the trace plots of SGD are not as smooth as GD.

Now let us test on bodyfat data.

```
bodyfat <- read.csv("../BSC-L10-sgd/Bodyfat.csv")
# Note the importance of scaling
bodyfat.X <- bodyfat %>% select(Abdomen, Weight) %>% scale() %>% as.matrix()
bodyfat.X <- cbind(1, bodyfat.X)
bodyfat.y <- bodyfat %>% select(bodyfat) %>% as.matrix()
sgd.bodyfat <- sgd.lm(bodyfat.X, bodyfat.y, beta.init = c(0, 0, 0), alpha = 0.05,
    tol = 1e-05, max.iter = 10000)
```

```
## Converged.
## Iterated 1362 times.
## Coef:  19.78609 10.81002 -5.165585
```

```
betas <- as.data.frame(t(do.call(cbind, sgd.bodyfat$coef)))
colnames(betas) <- c("beta0", "beta1", "beta2")
betas <- betas %>% mutate(iter = 1:nrow(betas))
betas <- melt(betas, id.vars = "iter", variable.name = "coef")
ggplot(betas, aes(iter, value)) + geom_line(aes(colour = coef)) + ylim(c(-10, 25))
```

Note that you need re-scale your results.

Comparison of GD, SGD and Newton

3. Compare Gradient Descent, Stochastic Gradient Descent and Newton method.

We focus on the generated data. First recall Newton Method.

```
# newton method
func = function(beta) {
    sum((y - beta[1] - beta[2] * x)^2)/2/length(y)
}
grad = function(beta) {
    matrix(c(sum(-2 * (y - beta[1] - beta[2] * x)), sum(-2 * x * (y - beta[1] - beta[2] *
        x))), 2, 1)/length(y)
}
hess = function(beta) {
    matrix(c(2 * length(x), 2 * sum(x), 2 * sum(x), 2 * sum(x^2)), 2, 2)/length(y)
}
newton <- function(f3, x0, tol = 1e-09, n.max = 100) {
    # Newton's method for optimisation, starting at x0 f3 is a function that given x
    # returns the list {f(x), grad f(x), Hessian f(x)}

    x <- x0
    f3.x <- f3(x)
    xs <- list()
    xs[[1]] <- x
    n <- 0
```

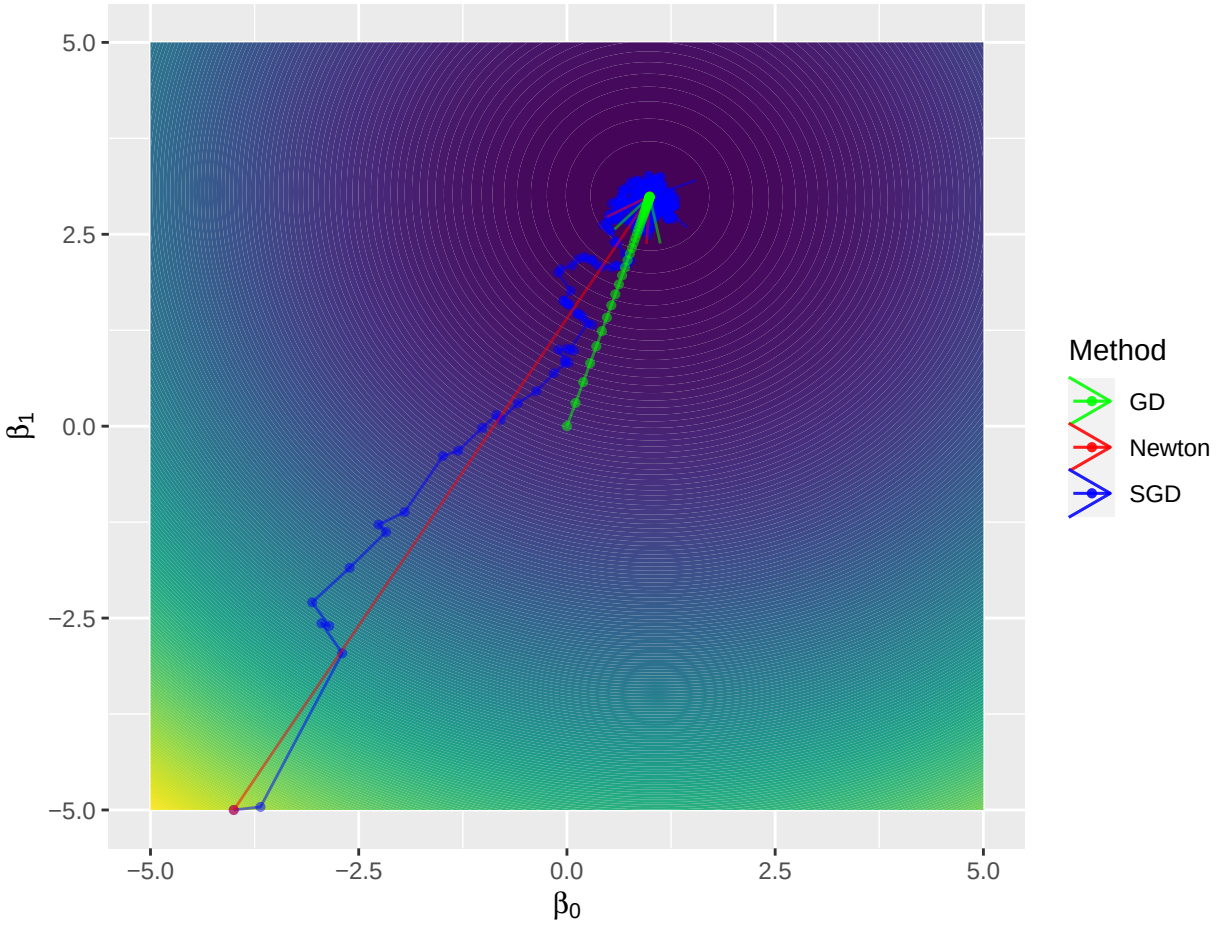
```
    while ((max(abs(f3.x[[2]])) > tol) & (n < n.max)) {
        x <- x - solve(f3.x[[3]], f3.x[[2]])
        f3.x <- f3(x)
        n <- n + 1
        xs[[n + 1]] <- x
    }
    if (n == n.max) {
        cat("newton failed to converge. \n")
    } else {
        cat("iterated", n + 1, "times. \n")
        return(xs)
    }
}
## Newton optimization
optimOut <- newton(function(beta) {
    list(func(beta), grad(beta), hess(beta))
}, c(-4, -5))
```

```
## iterated 2 times.
```

Now we plot the traces of Gradient Descent, Stochastic Gradient Descent and Newton method.

```
trace.newton <- data.frame(x = sapply(optimOut, `[`, 1), y = sapply(optimOut, `[`, 2))
trace.gd <- data.frame(x = sapply(gd1$coef, `[`, 1), y = sapply(gd1$coef, `[`, 2))
trace.sgd <- data.frame(x = sapply(sgd.est$coef, `[`, 1), y = sapply(sgd.est$coef, `[`, 2))
xs <- seq(-5, 5, length = 100)
ys <- seq(-5, 5, length = 100)
g <- expand.grid(xs, ys)
z <- sapply(1:dim(g)[1], function(i){lm.cost(X, y, c(g[i,1],g[i,2]))})
f_long <- data.frame(x = g[,1], y = g[,2], z = z)
colors <- c("SGD" = "blue", "Newton" = "red", "GD" = "green")

ggplot(f_long, aes(x, y, z = z)) +
  geom_contour_filled(aes(fill = stat(level)), bins = 200) +
  guides(fill = FALSE) +
  geom_path(data = trace.sgd, aes(x, y, z=0, color = 'SGD'), arrow = arrow(), alpha = 0.5) +
  geom_point(data = trace.sgd, aes(x, y, z=0, color = 'SGD'), size = 1.1, alpha = 0.5) +
  geom_path(data = trace.newton, aes(x, y, z=0, color = 'Newton'), arrow = arrow(), alpha = 0.5) +
  geom_point(data = trace.newton, aes(x, y, z=0, color = 'Newton'), size = 1.1, alpha = 0.5) +
  geom_path(data = trace.gd, aes(x, y, z=0, color = 'GD'), arrow = arrow(), alpha = 0.5) +
  geom_point(data = trace.gd, aes(x, y, z=0, color = 'GD'), size = 1.1, alpha = 0.5) +
  ggtitle('') +
  labs(x = expression(beta[0]),
         y = expression(beta[1]),
         color = "Method") +
    scale_color_manual(values = colors)
```



Our findings

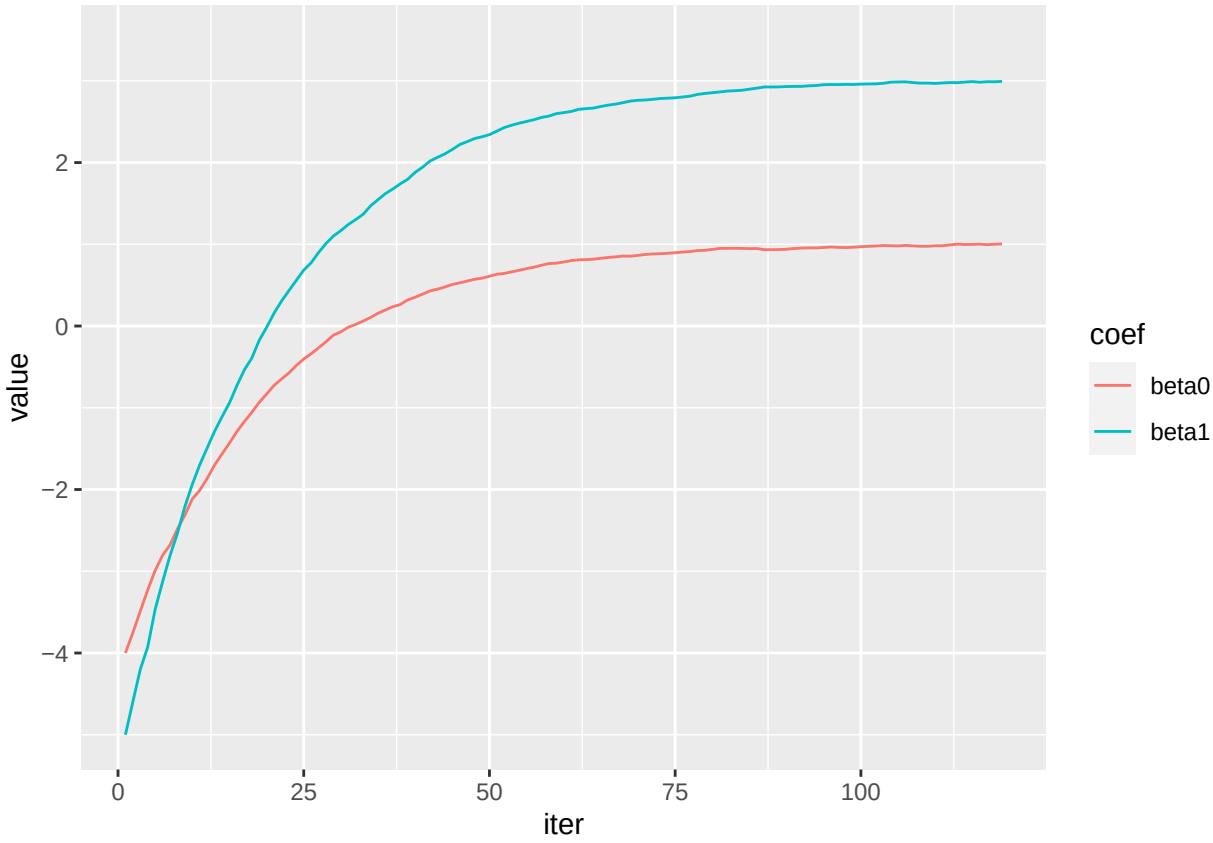
- Newton method converges very fast (if the second derivative exists).
- However, the analytic expression for the second derivative is often complicated or intractable, requiring a lot of computation. Therefore, **Newton method is not widely used in machine learning.**
- If the number of training samples is **very** large, GD may take too long, while using SGD will be faster because you use only one training sample and it starts improving itself right away from the first sample.
- In practice, computing error on every single example leads to large variance in the parameter update. You have found that the trace plots of SGD are much noisier. We can balance the complexity of computing on all training samples and large errors on single samples by computing on some samples (i.e., mini-batch). This would leads to more stable convergence (see the section below for an example).
- When the training set is very large, stochastic gradient descent is often preferred over batch gradient descent.
- Another issue is the choice of learning rate. See Bottou (2012), Zeiler (2012) and the SGD modules of `scikit-learn` in Python for more details.

Mini-batch SGD

```r
# test on the generated data
gd.mini <- sgd.lm(X, y, beta.init = c(-4, -5), alpha = 0.05, tol = 1e-05, max.iter = 10000,
    n.samples = 100)
```

```
## Converged.
## Iterated 118 times.
## Coef:  1.005626 2.992949
```

```r
betas <- as.data.frame(t(do.call(cbind, gd.mini$coef)))
colnames(betas) <- c("beta0", "beta1")
betas <- betas %>% mutate(iter = 1:nrow(betas))
betas <- melt(betas, id.vars = "iter", variable.name = "coef")
ggplot(betas, aes(iter, value)) + geom_line(aes(colour = coef)) + ylim(c(-5, 3.5))
```



Now you can compare mini-batch GD with SGD.

References

Bottou, Léon. 2012. "Stochastic Gradient Descent Tricks." In *Neural Networks: Tricks of the Trade*, 421–36. Springer.

Zeiler, Matthew D. 2012. "Adadelta: An Adaptive Learning Rate Method." *arXiv Preprint arXiv:1212.5701*.