Untitled

Marcin Kosinski

25.10.2015

```
logitGD <- function(y, x, optim.method = "GDI", eps = 10e-4,
                    max.iter = 100, alpha = function(t){1/t}, beta_0 = c(0,0)){
  stopifnot(length(y) == length(x) & optim.method %in% c("GDI", "GDII", "SGDI")
            & is.numeric(c(max.iter, eps, x)) & all(c(eps, max.iter) > 0) &
              is.function(alpha))
  iter <- 0
  err <- list()
  err[[iter+1]] <- eps+1
  w_old <- beta_0

  res <-list()
  while(iter < max.iter && (abs(err[[ifelse(iter==0,1,iter)]]) > eps)){

    iter <- iter + 1
    if (optim.method == "GDI"){
      w_new <- w_old + alpha(iter)*updateWeightsGDI(y, x, w_old)
    }
    if (optim.method == "GDII"){
      w_new <- w_old + as.vector(inverseHessianGDII(x, w_old)%*%
                                   updateWeightsGDI(y, x, w_old))
    }
    if (optim.method == "SGDI"){
      w_new <- w_old + alpha(iter)*updateWeightsSGDI(y[iter], x[iter], w_old)
    }
    res[[iter]] <- w_new
    err[[iter]] <- sqrt(sum((w_new - w_old)^2))

    w_old <- w_new

  }
  return(list(steps = c(list(beta_0),res), errors = c(list(c(0,0)),err)))
}

updateWeightsGDI <- function(y, x, w_old){
  (1/length(y))*c(sum(y-p(w_old, x)), sum(x*(y-p(w_old, x))))
  #c(sum(y-p(w_old, x)), sum(x*(y-p(w_old, x))))
}

updateWeightsSGDI <- function(y_i, x_i, w_old){
  c(y_i-p(w_old, x_i), x_i*(y_i-p(w_old, x_i)))
}

p <- function(w_old, x_i){
  1/(1+exp(-w_old[1]-w_old[2]*x_i))
}

inverseHessianGDII <- function(x, w_old){
```

```
  solve(
    matrix(c(
      sum(p(w_old, x)*(1-p(w_old, x))),
      sum(x*p(w_old, x)*(1-p(w_old, x))),
      sum(x*p(w_old, x)*(1-p(w_old, x))),
      sum(x*x*p(w_old, x)*(1-p(w_old, x)))
    ),
    nrow =2 )
  )
}
```

```
                    # wstępna inicjalizacja parametrów
eps = 1e-5                              # warunek stopu.

n = length(data)                        # data jest listą ramek danych.

diff = eps + 1                          # różnice w oszacowaniach parametrów
                                        # między kolejnymi krokami.

learningRates = function(x) 1/x         # długości kroku algorytmu.

beta_old = numeric(0, length = k)       # punkt startowy dlugosci k,
                                        # gdzie k to liczba zmiennych
                                        # objaśniających w modelu.

                                # estymacja
i = 1                                   # iterator kroku algorytmu
while(i <= n | diff < eps) do           # do zbieżności lub wyczerpania zbiorów
  batch = data[[i]]

  beta_new = beta_old - learningRates(i) * U_Batch(batch)
                                        # U_Batch to częściowa funkcja
                                        # log-wiarogdności dla zaobserwowanego
                                        # zbioru `batch`

  diff = euclidean_dist(beta_new, beta_old) # odległość euklidesowa

  beta_old = beta_new

  i = i + 1
end while
return beta_new
```

```
coxphSGD <- function(formula, data,
                     learningRates = function(x){1/x},
                     beta_0 = 0, epsilon = 1e-5 ) {
  checkArguments(formula, data, learningRates,
                 beta_0, epsilon) -> beta_old # check arguments

  n <- length(data)
  diff <- epsilon + 1
  i <- 1
  beta_new <- list() # steps are saved in a list so that they can
```

```
                    # be tracked in the future
  # estimate
  while(i <= n & diff > epsilon) {
    #tryCatch({
    beta_new[[i]] <- coxphSGD_batch(formula = formula, data = data[[i]],
                                    learningRate = learningRates(i),
                                    beta = beta_old)

    diff <- sqrt(sum((beta_new[[i]] - beta_old)^2))
    beta_old <- beta_new[[i]]
    i <- i + 1
    #}, error = function(cond) {i <<- n + 1})
  }

  # return results
  fit <- list()
  fit$Call <- match.call()
  fit$coefficients <- beta_new
  fit$epsilon <- epsilon
  fit$learningRates <- learningRates
  fit$steps <- i
  class(fit) <- "coxphSGD"
  fit
}

coxphSGD_batch <- function(formula, data, learningRate, beta){

  # Parameter identification as in  `survival::coxph()`.
  Call <- match.call()
  indx <- match(c("formula", "data"),
                names(Call), nomatch = 0)
  if (indx[1] == 0)
      stop("A formula argument is required")
  temp <- Call[c(1, indx)]
  temp[[1]] <- as.name("model.frame")

  mf <- eval(temp, parent.frame())
  Y <- model.extract(mf, "response")

  if (!inherits(Y, "Surv"))
      stop("Response must be a survival object")
  type <- attr(Y, "type")

  if (type != "right" && type != "counting")
      stop(paste("Cox model doesn't support \"", type, "\" survival data",
          sep = ""))

  # collect times, status, variables and reorder samples
  # to make the algorithm more clear to read and track
  cbind(not_censored = 1 - unclass(Y)[,2],
        times = unclass(Y)[,1],
        mf[, -1]) %>%
    arrange(times) -> batchData
```

```
  # calculate the log-likelihood for this batch sample
  partial_sum <- list()

  for(k in 1:nrow(batchData)) {

    # risk set for current time/observation
    risk_set <- batchData %>%
      filter(times <= batchData$times[k])

    nominator <- apply(risk_set[, -c(1,2)], MARGIN = 1, function(element){
      element * exp(element * beta)
    }) %>%
      t %>%
      colSums()

    denominator <- apply(risk_set[, -c(1,2)], MARGIN = 1, function(element){
      exp(element * beta)
    }) %>%
      t %>%
      colSums()

    partial_sum[[k]] <-
      batchData[k, "not_censored"] * (batchData[k, -c(1,2)] - nominator/denominator)

  }

  do.call(rbind, partial_sum) %>%
    colSums() -> U_batch

  beta_out <- beta + learningRate * U_batch

  return(beta_out)
}

checkArguments <- function(formula, data, learningRates,
                           beta_0, epsilon) {
  assert_that(is.list(data) & length(data) > 0)
  assert_that(length(unique(unlist(lapply(data, ncol)))) == 1)
  # + check names and types for every variables
  assert_that(is.function(learningRates))
  assert_that(is.numeric(epsilon))
  assert_that(is.numeric(beta_0))

    # check length of the start parameter
  if (length(beta_0) == 1) {
    beta_0 <- rep(beta_0, as.character(formula)[3] %>%
                    strsplit("\\+") %>%
                    unlist %>%
                    length)
  }
  return(beta_0)
}
```

```
x <- runif(1000)
z <- 2 + 3*x
pr <- 1/(1+exp(-z))
y <- rbinom(1000,1,pr)


logitGD(y, x, optim.method = "GDI", eps = 10e-5, max.iter = 500)$steps -> GDI
logitGD(y, x, optim.method = "GDII", eps = 10e-5, max.iter = 500)$steps -> GDII

ind <- sample(length(y))
logitGD(y[ind], x[ind], optim.method = "SGDI",
        max.iter = 500, eps = 10e-5)$steps -> SGDI.1
ind2 <- sample(length(y))
logitGD(y[ind2], x[ind2], optim.method = "SGDI",
        max.iter = 500, eps = 10e-5)$steps -> SGDI.2
ind3 <- sample(length(y))
logitGD(y[ind3], x[ind3], optim.method = "SGDI",
        max.iter = 500, eps = 10e-5)$steps -> SGDI.3
ind4 <- sample(length(y))
logitGD(y[ind4], x[ind4], optim.method = "SGDI",
        max.iter = 500, eps = 10e-5)$steps -> SGDI.4
ind5 <- sample(length(y))
logitGD(y[ind5], x[ind5], optim.method = "SGDI",
        max.iter = 500, eps = 10e-5)$steps -> SGDI.5

do.call(rbind, c(GDI, GDII, SGDI.1, SGDI.2, SGDI.3, SGDI.4, SGDI.5)) -> coeffs
unlist(lapply(list(GDI, GDII, SGDI.1, SGDI.2, SGDI.3, SGDI.4, SGDI.5), length)) -> algorithm
data2viz <- cbind(as.data.frame(coeffs),
      algorithm = unlist(mapply(rep, c("GDI", "GDII", "SGDI.1", "SGDI.2", "SGDI.3", "SGDI.4", "SGDI.5")
names(data2viz)[1:2] <- c("Intercept", "X")
library(ggplot2); library(ggthemes)
ggplot(data2viz) +
  geom_point(aes(x = X, y = Intercept, col = algorithm)) +
  geom_line(aes(x = X, y = Intercept, col = algorithm,
                group = algorithm)) +
  theme_tufte(base_size = 20)
```

logitGD() asda graphSGD()

```
graphSGD(c(0,0), y, x)
graphSGD(c(3.1,2.1), y, x)
graphSGD(c(4,3), y, x)
graphSGD(c(1,2), y, x)
```

```
dataCox <- function(N, lambda, rho, x, beta, censRate){

  # real Weibull times
  u <- runif(N)
  Treal <- (- log(u) / (lambda * exp(x %*% beta)))^(1 / rho)

  # censoring times
  Censoring <- rexp(N, censRate)
```

```
  # follow-up times and event indicators
  time <- pmin(Treal, Censoring)
  status <- as.numeric(Treal <= Censoring)

  # data set
  data.frame(id=1:N, time=time, status=status, x=x)
}

x <- matrix(sample(0:1, size = 40, replace = TRUE), ncol = 2)

head(dataCox(20, 3, 2, x, beta = c(2,3), 5))
```

```
  id       time status x.1 x.2
1  1 0.04737041      1   1   1
2  2 0.03925718      1   1   1
3  3 0.04814980      1   1   1
4  4 0.03243838      0   0   0
5  5 0.01085835      0   0   1
6  6 0.02817386      1   1   1
```

```
graphSGD(c(0,0), y, x, 4561);graphSGD(c(0,0), y, x, 456)
graphSGD(c(2,1), y, x, 4561);graphSGD(c(2,1), y, x, 456);
graphSGD(c(1,0), y, x, 4561);graphSGD(c(1,0), y, x, 456);
graphSGD(c(2.1,3.1), y, x, 4561)graphSGD(c(2.1,3.1), y, x, 456)
```

```
x <- matrix(sample(0:1, size = 20000, replace = TRUE), ncol = 2)
dCox <- dataCox(10^4, lambda = 3, rho = 2, x, beta = c(1,3), censRate = 5)
vizCoxSGD(dCox)
```

```r
coxphSGD <- function(formula, data, learningRates = function(x){1/x},
                     beta_0 = 0, epsilon = 1e-5 ) {
  checkArguments(formula, data, learningRates,
                 beta_0, epsilon) -> beta_old # check arguments
  n <- length(data)
  diff <- epsilon + 1
  i <- 1
  beta_new <- list() # steps are saved in a list so that they can
                     # be tracked in the future
  # estimate
  while(i <= n & diff > epsilon) {
    beta_new[[i]] <- coxphSGD_batch(formula = formula, data = data[[i]],
                      learningRate = learningRates(i), beta = beta_old)

    diff <- sqrt(sum((beta_new[[i]] - beta_old)^2))
    beta_old <- beta_new[[i]]
    i <- i + 1
  }
  # return results
  list(Call = match.call(), coefficients = beta_new, epsilon = epsilon,
       learningRates = learningRates, steps = i)
}

coxphSGD_batch <- function(formula, data, learningRate, beta){
  # collect times, status, variables and reorder samples
  # to make the algorithm more clear to read and track
  batchData <- prepareBatch(formula = formula, data = data)

  # calculate the log-likelihood for this batch sample
  partial_sum <- list()

  for(k in 1:nrow(batchData)) {
    # risk set for current time/observation
    risk_set <- batchData %>% filter(times <= batchData$times[k])

    nominator <- apply(risk_set[, -c(1,2)], MARGIN = 1, function(element){
      element * exp(element * beta)
    }) %>% rowSums()

    denominator <- apply(risk_set[, -c(1,2)], MARGIN = 1, function(element){
      exp(element * beta)
    }) %>% rowSums()

    partial_sum[[k]] <-
      batchData[k, "event"] * (batchData[k, -c(1,2)] - nominator/denominator)
  }
  do.call(rbind, partial_sum) %>%
    colSums() -> U_batch

  return(beta + learningRate * U_batch)
}
```

```r
prepareBatch <- function(formula, data) {
  # Parameter identification as in  `survival::coxph()`.
  Call <- match.call()
  indx <- match(c("formula", "data"),
                names(Call), nomatch = 0)
  if (indx[1] == 0)
      stop("A formula argument is required")
  temp <- Call[c(1, indx)]
  temp[[1]] <- as.name("model.frame")

  mf <- eval(temp, parent.frame())
  Y <- model.extract(mf, "response")

  if (!inherits(Y, "Surv"))
      stop("Response must be a survival object")
  type <- attr(Y, "type")

  if (type != "right" && type != "counting")
      stop(paste("Cox model doesn't support \"", type, "\" survival data",
          sep = ""))

  # collect times, status, variables and reorder samples
  # to make the algorithm more clear to read and track
  cbind(event = unclass(Y)[,2], # 1 indicates event, 0 indicates cens
        times = unclass(Y)[,1],
        mf[, -1]) %>%
    arrange(times)
}
```