Untitled

Marcin Kosinski

25.10.2015

dupa sraka `logitGD()` jasd

```
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))))
}

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
```

```r
  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)
```

```
}
```