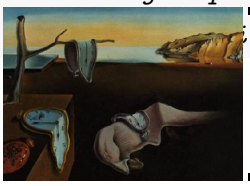


Journal of Statistical Software
MMMMMM YYYY, Volume VV, Issue II. doi: 10.18637/jss.v000.i00

fastcpd: Fast Change Point Detection in R

First Author
Texas A&M University

Second Author
Texas A&M University

Abstract

TBD

Keywords: change point detection, gradient descent, quasi-newton, incremental gradient, R.

1. Introduction: Fast change point detection in R

fastcpd (**FAST** **C**hange **P**oint **D**etection) is an R package for fast offline change-point detection. Change-point detection is an important problem in many fields, including signal processing, finance, and biology. Given a sequence of data points, a change-point detection algorithm identifies points where the underlying statistical properties of the data change. **fastcpd** implements a novel approach for change-point detection that is orders of magnitude faster than existing methods, without sacrificing accuracy.

The **fastcpd** method is based on a sequential optimization algorithm that uses gradient descent to efficiently search for change-points. The algorithm is designed to minimize a cost function that captures the likelihood of observing the data given a particular segmentation. The key idea is to update the cost function using information from previous steps, rather than re-optimizing the objective function at each step. This allows **fastcpd** to quickly identify change-points in long sequences of data, even when the cost function involves solving a non-trivial optimization problem.

fastcpd supports change-point detection in a variety of settings, including generalized linear models and penalized regression. The package provides a simple and intuitive interface for users to specify their data and desired parameters, and produces output in a convenient format for further analysis. In addition, **fastcpd** includes a suite of visualization tools for exploring the results of the change-point detection algorithm.

We believe that **fastcpd** will be a valuable tool for researchers and practitioners working with change-point detection problems. By providing fast and accurate change-point detection in a user-friendly package, **fastcpd** will enable users to more easily analyze and understand complex datasets.

In R (R Core Team 2017), Killick and Eckley (2014) has implemented **changepoint** to

```
  family = NULL,
  epsilon = 1e-10,
  min_prob = 10^10,
  winsorise_minval = -20,
  winsorise_maxval = 20,
  p = NULL,
  cost = negative_log_likelihood,
  cost_gradient = cost_update_gradient,
  cost_hessian = cost_update_hessian
)
```

where each parameters have the following usages:

- `data`: A data frame containing the data to be segmented where each row denotes each data point. In regression settings, the first column is the response variable while the rest are covariates.
- `beta`: Initial cost value specified in Algorithm 1 in Zhang and Dawn (2022).
- `segment_count`: Number of segments for initial guess. If not specified, the initial guess on the number of segments is 10.
- `trim` Trimming for the boundary change points so that a change point close to the boundary will not be counted as a change point. This parameter also specifies the minimum distance between two change points. If several change points have mutual distances smaller than `trim * nrow(data)`, those change points will be merged into one single change point.
- `momentum_coef` Momentum coefficient to be applied to each update. This parameter is used when the loss function is bad-shaped so that maintaining a momentum from previous update is desired. Default value is 0, meaning the algorithm doesn't maintain a momentum by default.
- `sgd_k` Number of epochs in for each update whenever the algorithm takes a new data point, in the sense that the data are analyzed sequentially according to the nature of Pruned Exact Linear Time (PELT) algorithm (Killick, Fearnhead, and Eckley 2012).
- `family` Family of the model. Can be "`binomial`", "`poisson`", "`lasso`", "`gaussian`" or "`custom`". For simplicity, user can also omit this parameter, indicating that they will be using their own cost functions. If specified as "`custom`" or "`NULL`", the user must specify the cost function, gradient and corresponding Hessian matrix. Hessian is preferred when the user want to specify their own cost function, but not analytically available, the user should provide a single number (diagonal matrix) to replace the Hessian matrix. Should be left as `NULL` if the user would like to use their own cost functions.

- `epsilon` Epsilon to avoid numerical issues. Only used for Logistic Regression and Poisson Regression.
- `min_prob` Minimum probability to avoid numerical issues. Only used for Poisson Regression.
- `winsorise_minval` Minimum value for the parameter in Poisson Regression to be winsorised.
- `winsorise_maxval` Maximum value for the parameter in Poisson Regression to be winsorised.
- `lambda` Lambda for L1 regularization. Only used in "lasso".
- `cost` Cost function to be used. This and the following two parameters should not be specified at the same time with `family`. If not specified, the default is the negative log-likelihood for the corresponding family.
- `cost_gradient` Gradient function for the custom cost function.
- `cost_hessian` Hessian function for the custom cost function.

Return of the function is a list containing two elements:

- `cp_set`: A vector containing the change points.
- `cost_value`: Values of the cost function for each data segments separated by the change points.

A '`fastcpd`' object is returned by the function. This object can be used to plot the change points and the cost function values. Compatible functions include `plot()`, `print()` and `summary()`.

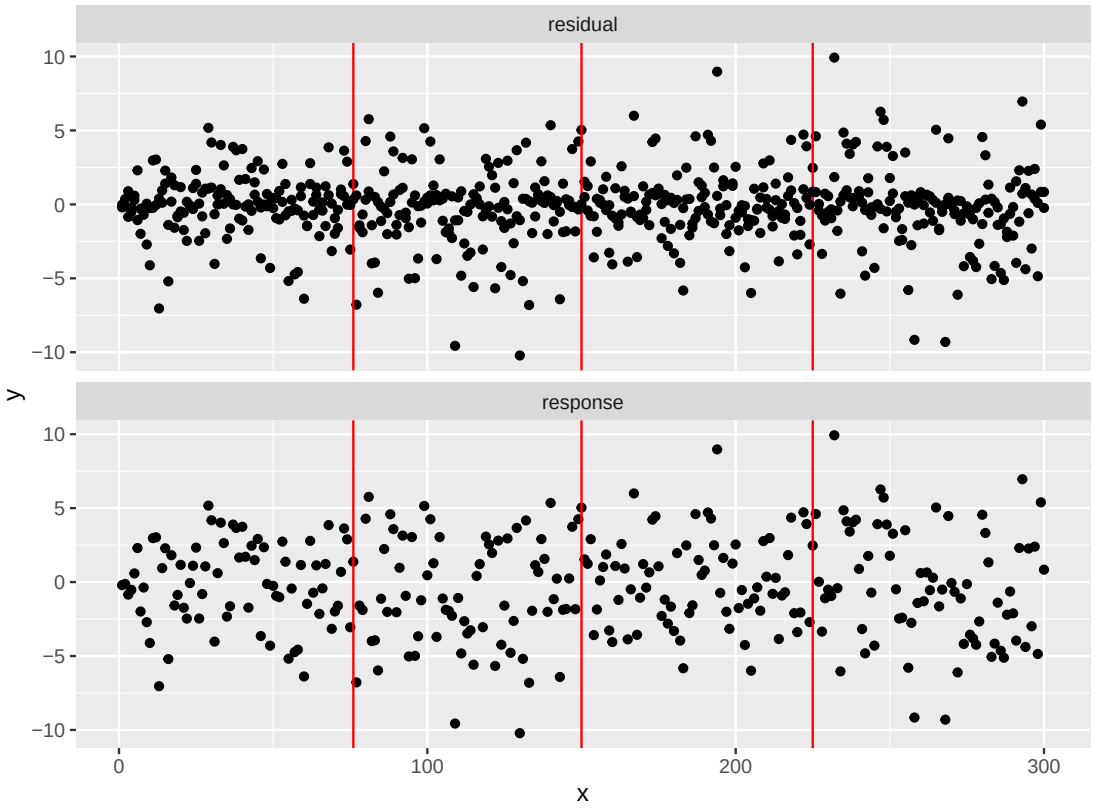
3. Changes in linear models

```
summary(linear_regression_result)

##
## Call:
## fastcpd(data = data, beta = beta, k = function(x) 0, family = "gaussian",
##     epsilon = 1e-05)
##
## [1] "Residuals:"
##        Min         1Q     Median         3Q        Max
## -1.9500129 -0.4839276  0.0068760  0.4708422  1.8456402
```

Model	`family`	Description
Linear Regression	`gaussian`	Will be using pre-defined cost functions, details can be found in Section 3
Logistic Regression	`binomial`	Will be using pre-defined cost functions, details can be found in Section 4
Poisson Regression	`poisson`	Will be using pre-defined cost functions, details can be found in Section 5
LASSO	`lasso`	Will be using pre-defined cost functions, details can be found in Section 3
User-defined Model	`custom`	If `family` is specified as "custom" or `NULL`, `cost`, `cost_gradient` and `cost_hessian` must be provided.
Huber Regression	`custom`	Instead of providing Huber Regression as a built-in model, we provide an example of how to use `custom` model. Details can be found in Section 7.2
Quantile Regression	`custom`	Instead of providing Quantile Regression as a built-in model, we provide an example of how to use `custom` model. Details can be found in Section 7.3
Mean Shift	`custom`	Instead of providing Quantile Regression as a built-in model, we provide an example of how to use `custom` model. Details can be found in Section 7.4

Table 1: All the models that have been implemented in the package, including an empty model so that the users are able to provide their own.

```
##
## Change points:
## [1]  76 150 225

plot(linear_regression_result)
```


residual
response
y
x
10
5
0
−5
−10
0
100
200
300

	p=1			p=3			p=5		
	n=300	n=600	n=1500	n=300	n=600	n=1500	n=300	n=600	n=1500
cpc=0	1.00, 1.48s	1.00, 3.88s	1.00, 12.12s	1.00, 1.59s	1.00, 3.36s	1.00, 9.72s	1.00, 1.81s	1.00, 3.87s	1.00, 12.16s
	1.00, 103.61s	1.00, 341.06s	1.00, 1178.56s	1.00, 61.17s	1.00, 226.55s	1.00, 1283.73s	1.00, 96.40s	1.00, 282.26s	1.00, 913.02s
	1.00, 156.23s	1.00, 481.71s	1.00, 1376.99s	1.00, 78.88s	1.00, 228.14s	1.00, 3373.77s	1.00, 109.70s	1.00, 346.53s	1.00, 3554.79s
	0.90, 40.71s	1.00, 194.13s	1.00, 1160.40s	1.00, 37.18s	0.90, 168.97s	0.91, 603.55s	0.56, 28.89s	0.40, 93.09s	0.77, 473.10s
cpc=1	0.98, 2.40s	0.98, 6.25s	0.98, 14.13s	0.98, 1.72s	0.99, 4.53s	0.99, 16.93s	0.99, 2.18s	0.99, 5.41s	1.00, 14.97s
	0.91, 93.22s	0.98, 428.46s	1.00, 2755.46s	0.97, 92.85s	0.98, 348.92s	1.00, 2078.98s	1.00, 69.20s	1.00, 288.67s	1.00, 1975.72s
	0.96, 186.53s	0.99, 931.36s	0.99, 4333.45s	0.99, 158.12s	0.99, 602.64s	1.00, 4723.81s	0.98, 210.51s	1.00, 440.30s	1.00, 2915.44s
	0.96, 31.05s	1.00, 110.85s	0.97, 650.02s	0.92, 19.16s	0.97, 87.13s	0.97, 574.22s	0.98, 21.14s	0.88, 73.06s	0.92, 417.99s
cpc=3	0.75, 2.58s	0.99, 7.00s	0.99, 16.35s	0.83, 2.01s	0.99, 4.20s	1.00, 10.48s	0.84, 2.14s	0.99, 4.95s	1.00, 14.83s
	0.83, 116.19s	0.97, 586.45s	0.99, 2609.28s	0.82, 74.45s	0.99, 218.70s	1.00, 1940.88s	0.92, 77.67s	0.99, 323.52s	1.00, 1697.94s
	0.82, 227.43s	0.97, 903.17s	0.99, 3847.89s	0.81, 199.59s	0.99, 517.29s	1.00, 2144.98s	0.92, 113.10s	0.92, 618.55s	1.00, 3090.91s
	0.92, 23.83s	0.96, 70.75s	0.98, 373.69s	0.98, 14.70s	0.98, 45.15s	1.00, 289.93s	0.95, 10.81s	0.94, 44.93s	0.98, 242.29s
cpc=5	0.67, 2.82s	0.82, 7.87s	0.99, 15.63s	0.55, 2.32s	0.88, 7.17s	0.99, 16.55s	0.75, 1.86s	0.96, 4.67s	1.00, 10.74s
	0.79, 111.85s	0.83, 455.71s	0.99, 2270.08s	0.83, 46.30s	0.93, 311.22s	0.99, 1719.32s	0.79, 72.71s	1.00, 115.01s	1.00, 1357.81s
	0.52, 200.35s	0.89, 848.59s	0.99, 2710.85s	0.94, 92.61s	0.99, 294.48s	0.99, 2647.00s	0.91, 138.84s	1.00, 366.56s	0.99, 2266.15s
	0.87, 17.31s	0.97, 59.14s	0.99, 225.48s	0.97, 10.93s	0.99, 42.71s	0.99, 203.91s	0.96, 9.39s	0.98, 30.35s	0.98, 150.41s

Table 2: Comparison of the algorithms for the linear regression model in mean.

4. Changes in generalized linear models: Logistic regression

We first consider the change-point detection problem in the generalized linear models (GLM). Suppose we have a data set containing a set of predictors/covariates and corresponding response variables, i.e. each data point `data[i, ]` contains a set of predictors/covariates `data[i, -1]` and a response `data[i, 1]`. The reason we set the first column to be the response variables are that in R, we can use `-1` to denote all columns other than the first column without the need to consider number of columns altogether.

Suppose the response variables follows a binomial distribution with specifics defined as

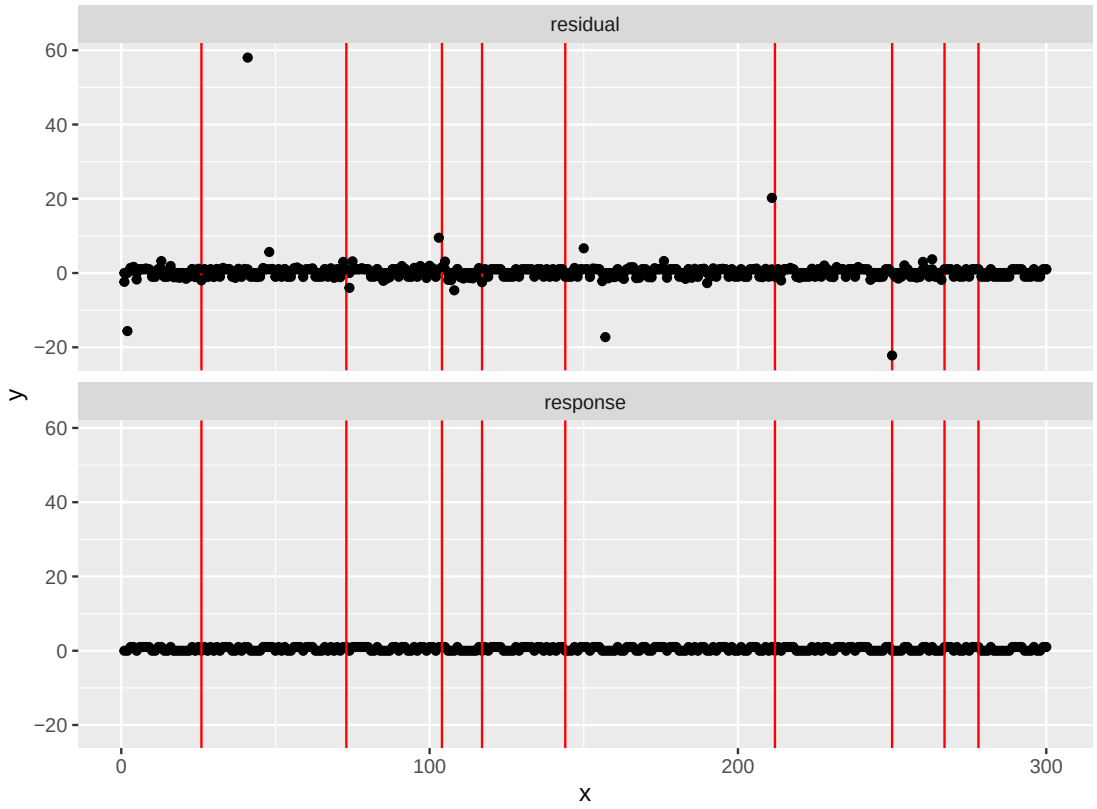
$$y_i \sim \text{Bernoulli}\left(\frac{1}{1+e^{-x_i^\top \theta_i}}\right), \quad x_i \sim \mathcal{N}_p(0,\Sigma) \text{ with } \Sigma = (0.9^{|i-j|})_{p\times p}, \quad 1 \le i \le n, \quad (1)$$

where $\{y_i\}$ is the response variable, $\{x_i\}$ is the covariate vector, Σ is the covariance matrix for the sampling of $\{x_i\}$, p is the number of predictors, n is the total number of data points.

We now use the **fastcpd** package to detect the change points in the data set `data`.

```
## Warning:  fit_glm:  fitted probabilities numerically 0 or 1 occurred
## Warning:  fit_glm:  fitted probabilities numerically 0 or 1 occurred
## Warning:  fit_glm:  fitted probabilities numerically 0 or 1 occurred
## Warning:  fit_glm:  fitted probabilities numerically 0 or 1 occurred
## Warning:  fit_glm:  fitted probabilities numerically 0 or 1 occurred
## Warning:  fit_glm:  fitted probabilities numerically 0 or 1 occurred
## Warning:  fit_glm:  fitted probabilities numerically 0 or 1 occurred
```

```
summary(logistic_regression_result)

##
## Call:
## fastcpd(data = data, beta = beta, k = function(x) 0, family = "binomial",
##     epsilon = 1e-05)
##
## [1] "Residuals:"
##       Min        1Q    Median        3Q       Max
## -22.20031  -1.01676  -0.00002   1.02329  58.02099
##
## Change points:
## [1]  26  73 104 117 144 212 250 267 278

plot(logistic_regression_result)
```




```
# n <- 500
# p <- 8
# x <- cbind(1, mvtnorm::rmvnorm(n, rep(0, p - 1), diag(p - 1)))
# xb <- c(
#   x[1:225, ] %*% c(-1, rnorm(p - 1, 0.5, 0.1)),
#   x[226:500, ] %*% c(1, rnorm(p - 1, 0.5, 0.1))
# )
# y <- rbinom(n, size = 1, prob = 1 / (1 + exp(-xb)))
# data <- cbind(y, x)
# logistic_regression_result <- fastcpd(
#   data = data,
#   beta = (p + 1) * log(n) / 2,
#   family = "binomial"
# )
```

	p=1			p=3			p=5		
	n=300	n=600	n=1500	n=300	n=600	n=1500	n=300	n=600	n=1500
cpc=0	0.61, 1.34s	0.64, 2.96s	0.61, 8.70s	0.22, 1.01s	0.21, 2.16s	0.30, 6.24s	0.12, 0.96s	0.10, 1.93s	0.11, 5.35s
	0.67, 23.88s	0.66, 101.76s	0.88, 405.55s	0.15, 6.41s	0.11, 14.61s	0.15, 91.25s	0.07, 4.94s	0.07, 10.64s	0.15, 31.18s
	0.77, 45.09s	0.85, 185.40s	0.82, 784.56s	0.09, 6.70s	0.08, 18.90s	0.15, 101.19s	0.07, 7.75s	0.07, 16.13s	0.16, 60.53s
	0.48, 23.04s	0.53, 79.22s	0.65, 238.85s	0.06, 10.93s	0.09, 21.36s	0.20, 78.12s	0.07, 10.18s	0.06, 21.70s	0.27, 64.66s
cpc=1	0.69, 1.47s	0.78, 3.30s	0.85, 10.43s	0.67, 0.91s	0.65, 2.19s	0.68, 6.29s	0.58, 0.94s	0.60, 1.88s	0.61, 5.68s
	0.64, 22.21s	0.75, 76.50s	0.81, 452.43s	0.63, 4.55s	0.60, 15.78s	0.61, 80.27s	0.57, 4.42s	0.57, 10.38s	0.67, 30.22s
	0.84, 33.11s	0.69, 143.66s	0.68, 681.70s	0.57, 6.26s	0.59, 17.69s	0.64, 128.01s	0.57, 7.94s	0.57, 18.31s	0.61, 81.24s
	0.71, 22.97s	0.70, 73.92s	0.83, 253.55s	0.56, 9.30s	0.59, 22.18s	0.65, 88.01s	0.57, 9.86s	0.56, 24.33s	0.61, 74.32s
cpc=3	0.76, 1.28s	0.76, 2.79s	0.87, 10.19s	0.85, 0.94s	0.87, 2.27s	0.82, 5.94s	0.83, 0.97s	0.83, 1.92s	0.82, 5.16s
	0.71, 18.69s	0.78, 90.92s	0.75, 405.34s	0.83, 5.63s	0.83, 16.40s	0.86, 86.75s	0.82, 4.81s	0.81, 10.35s	0.82, 37.93s
	0.62, 35.17s	0.76, 99.60s	0.67, 609.95s	0.82, 7.69s	0.82, 25.64s	0.85, 104.59s	0.81, 8.07s	0.81, 15.89s	0.83, 64.66s
	0.58, 28.55s	0.74, 66.22s	0.84, 225.26s	0.82, 12.20s	0.83, 22.50s	0.81, 83.54s	0.82, 11.57s	0.80, 23.02s	0.81, 71.49s
cpc=5	0.67, 1.13s	0.80, 3.01s	0.78, 9.63s	0.86, 0.88s	0.90, 2.02s	0.90, 6.32s	0.89, 0.89s	0.89, 1.79s	0.88, 5.04s
	0.71, 19.48s	0.72, 59.59s	0.81, 349.84s	0.89, 4.97s	0.88, 12.10s	0.90, 55.76s	0.89, 5.49s	0.88, 9.70s	0.89, 33.49s
	0.83, 38.36s	0.63, 142.46s	0.80, 592.42s	0.89, 8.16s	0.90, 23.92s	0.90, 99.00s	0.90, 8.36s	0.89, 19.43s	0.87, 60.18s
	0.51, 24.90s	0.67, 70.35s	0.83, 221.87s	0.88, 9.02s	0.89, 25.20s	0.87, 68.76s	0.89, 9.85s	0.88, 20.84s	0.80, 49.09s

Table 3: Comparison of the algorithms for the logistic regression model in mean.

5. Changes in generalized linear models: Poisson regression

We now consider the change-point detection problem in the Poisson Regression setting.

```
data_gen_poisson <- function(n, p, true.coef, true.cp.loc, Sigma, evar) {
  loc <- unique(c(0, true.cp.loc, n))
  if(dim(true.coef)[2] != length(loc)-1) stop("true.coef and true.cp.loc do not match
  x <- mvtnorm::rmvnorm(n, mean = rep(0, p), sigma = Sigma)
  y <- NULL
  for(i in 1:(length(loc)-1))
  {
    Xb <- x[(loc[i] + 1):loc[i + 1], ,drop=FALSE]%*%true.coef[,i,drop=FALSE]
    y <- c(y, rpois(loc[i + 1] - loc[i], exp(Xb)))
  }
  data <- cbind(y, x)
  true_cluster <- rep(1:(length(loc)-1), diff(loc))
  result <- list(data, true_cluster)
  return(result)
}

cost_glm_poisson <- function(data, family="poisson")
{
data <- as.matrix(data)
p <- dim(data)[2] - 1
out <- fastglm(as.matrix(data[,1:p]), data[,p+1], family=family)
return(out$deviance/2)
}

CP_vanilla_poisson <- function(data, beta, cost=cost_glm_poisson, trim = 0.025)
{
n <- dim(data)[1]
p <- dim(data)[2] - 1
Fobj <- c(-beta, 0)
cp_set <- list(NULL,0)
set <- c(0,1)
 for(t in 2:n)
 {
 m <- length(set)
 cval <- rep(NA, m)
  for(i in 1:m)
  {
  k <- set[i] + 1
  if(t-k>=p-1) cval[i] <- suppressWarnings(cost(data[k:t,, drop = FALSE])) else cval[
```

```
  }
 obj <- cval + Fobj[set+1] + beta
 min_val <- min(obj)
 ind <- which(obj==min_val)[1]
 cp_set_add <- c(cp_set[[set[ind]+1]], set[ind])
 cp_set <- append(cp_set,list(cp_set_add))
 ind2 <- (cval + Fobj[set+1]) <= min_val
 set <- c(set[ind2], t)
 Fobj <- c(Fobj, min_val)
 }
cp <- cp_set[[n+1]]
nLL <- 0

cp <- cp[(cp >= trim * n) & (cp <= (1 - trim) * n)]
cp <- sort(unique(c(0, cp)))

segment_indices <- which((diff(cp) < trim * n) == TRUE)
if (length(segment_indices) > 0) {
  cp <- floor(
    (cp[-(segment_indices + 1)] + cp[-segment_indices]) / 2
  )
}
cp_loc <- unique(c(0,cp,n))
for(i in 1:(length(cp_loc)-1))
 {
 seg <- (cp_loc[i]+1):cp_loc[i+1]
 data_seg <- data[seg,]
 out <- fastglm(as.matrix(data_seg[, 1:p]), data_seg[, p+1], family="poisson")
 nLL <- out$deviance/2 + nLL
 }

output <- list(cp, nLL)
names(output) <- c("cp", "nLL")
return(output)
}
experiment_setup <- purrr::cross_df(
  list(
    n = c(300, 600, 1500),
    p = c(1, 3, 5),
    change_points_count = c(0, 1, 3, 5),
    sgd_k = c(1, 3, 5, 7)
  )
)
```

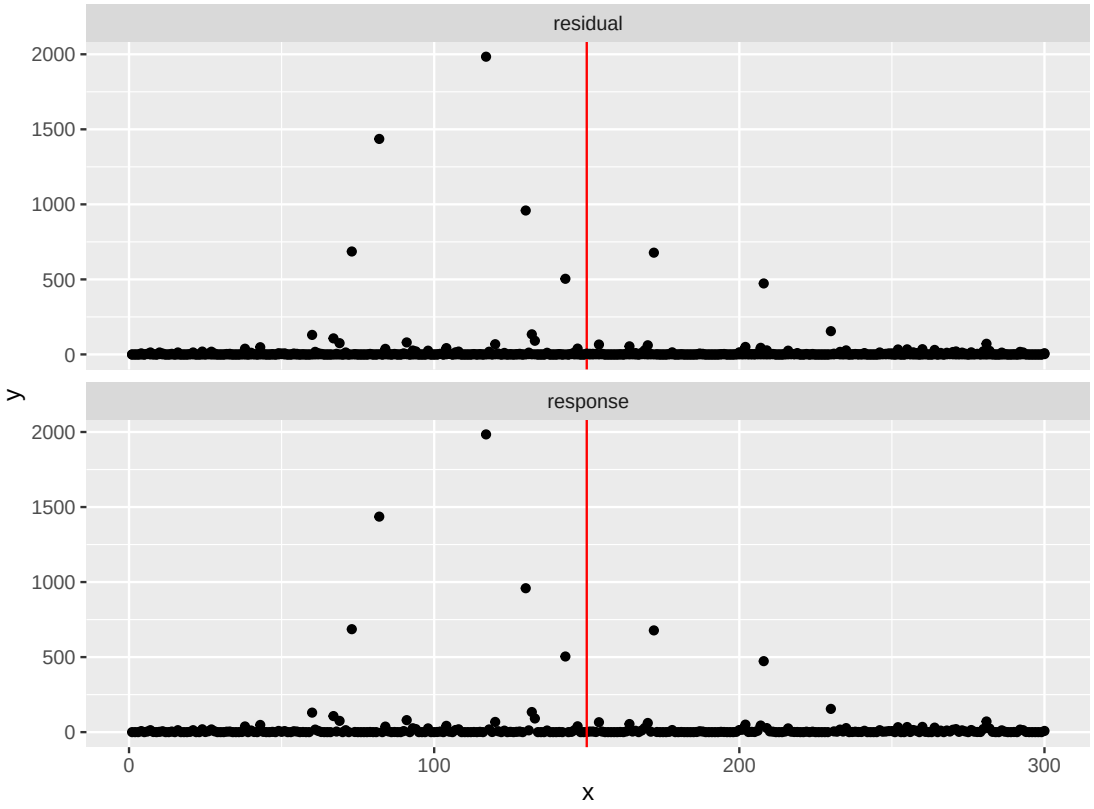
```
experiment_num <- 10

rand_gd <- time_gd <- rep(list(rep(NA, experiment_num)), nrow(experiment_setup))
rng <- RNGseq(experiment_num * nrow(experiment_setup), 3)

cl <- parallel::makeCluster(parallel::detectCores(), outfile = paste0("log/poisson.log
doParallel::registerDoParallel(cl)
experiment_result <- foreach::foreach(experiment_setup_index = seq_len(nrow(experimen
  foreach::foreach(experiment_index = seq_len(5), rrng = rng[(experiment_setup_index -
    rngtools::setRNG(rrng)
    experiment_setup_row <- experiment_setup[experiment_setup_index, ]
    n <- experiment_setup_row$n
    p <- experiment_setup_row$p
    change_points_count <- experiment_setup_row$change_points_count
    sgd_k <- experiment_setup_row$sgd_k

    if (p == 1) {
      theta_0 <- 1.2
    } else {
      theta_0 <- c(1, 1.2, -1, 0.5, -2)[seq_len(p)]
    }

    true.coef <- matrix(rep(theta_0, change_points_count + 1), nrow = p, ncol = change

    theta_norm <- c(0.36, NA, 0.81, NA, 1.96)[p]
    delta <- rnorm(p)
    sigma_ <- 0.9**abs(row(diag(p)) - col(diag(p)))
    delta_coef <- sqrt(theta_norm / t(delta) %*% sigma_ %*% delta)

    if (change_points_count == 0) {
      true.cp.loc <- NULL
    } else if (change_points_count == 1) {
      true.cp.loc <- n / 2 * seq_len(change_points_count)
      true.coef[, 2] <- theta_0 + c(delta_coef) * delta
    } else if (change_points_count == 3) {
      true.cp.loc <- n / 4 * seq_len(change_points_count)
      true.coef[, 2] <- theta_0 + c(delta_coef) * delta
      true.coef[, 4] <- theta_0 - c(delta_coef) * delta
    } else if (change_points_count == 5) {
      true.cp.loc <- n / 6 * seq_len(change_points_count)
      true.coef[, 2] <- theta_0 + c(delta_coef) * delta
      true.coef[, 4] <- theta_0 - c(delta_coef) * delta
      true.coef[, 6] <- theta_0 + c(delta_coef) * delta
```

```
    }

    Sigma <- diag(1, p)
    evar <- 0.5
    out <- data_gen_poisson(n, p, true.coef, true.cp.loc, Sigma, evar)
    data <- out[[1]]
    g_tr <- out[[2]]
    beta <- log(n)/2

    if (file.exists(paste0("cache/poisson", experiment_setup_index, "_", experiment_index, '
      rds_file <- readRDS(paste0("cache/poisson", experiment_setup_index, "_", experiment_in
      cp_set <- rds_file$cp_set
      time_used <- rds_file$time_used
    } else {
      start <- proc.time()
      if (sgd_k == 7) {
        cp_set <- CP_vanilla_poisson(data[, c(1:p + 1, 1)], beta)$cp
      } else {
        poisson_regression_result <- fastcpd(
          data = data,
          beta = beta,
          family = "poisson",
          k = function(x) sgd_k - 1,
          epsilon = 1e-5
        )
        cp_set <- poisson_regression_result@cp_set
      }
      time_used <- unname((proc.time() - start)[3])
      saveRDS(list(data = data, cp_set = cp_set, time_used = time_used), paste0("cache/poiss
    }
    cp_gd <- cp_set[!(cp_set==0)]
    K_est <- length(cp_gd) + 1
    cp_un <- unique(c(0, cp_gd, n))
    g_est <- rep(1:K_est, diff(cp_un))
    c(fossil::rand.index(g_tr, g_est), time_used)
  }
parallel::stopCluster(cl)

experiment_setup_index <- 25
experiment_setup_row <- experiment_setup[experiment_setup_index, ]
n <- experiment_setup_row$n
p <- experiment_setup_row$p
change_points_count <- experiment_setup_row$change_points_count
```

```r
sgd_k <- experiment_setup_row$sgd_k

if (p == 1) {
  theta_0 <- 1.2
} else {
  theta_0 <- c(1, 1.2, -1, 0.5, -2)[seq_len(p)]
}

true.coef <- matrix(rep(theta_0, change_points_count + 1), nrow = p, ncol = change_po

theta_norm <- c(0.36, NA, 0.81, NA, 1.96)[p]
delta <- rnorm(p)
sigma_ <- 0.9**abs(row(diag(p)) - col(diag(p)))
delta_coef <- sqrt(theta_norm / t(delta) %*% sigma_ %*% delta)

if (change_points_count == 0) {
  true.cp.loc <- NULL
} else if (change_points_count == 1) {
  true.cp.loc <- n / 2 * seq_len(change_points_count)
  true.coef[, 2] <- theta_0 + c(delta_coef) * delta
} else if (change_points_count == 3) {
  true.cp.loc <- n / 4 * seq_len(change_points_count)
  true.coef[, 2] <- theta_0 + c(delta_coef) * delta
  true.coef[, 4] <- theta_0 - c(delta_coef) * delta
} else if (change_points_count == 5) {
  true.cp.loc <- n / 6 * seq_len(change_points_count)
  true.coef[, 2] <- theta_0 + c(delta_coef) * delta
  true.coef[, 4] <- theta_0 - c(delta_coef) * delta
  true.coef[, 6] <- theta_0 + c(delta_coef) * delta
}

Sigma <- diag(1, p)
evar <- 0.5
out <- data_gen_poisson(n, p, true.coef, true.cp.loc, Sigma, evar)
data <- out[[1]]
g_tr <- out[[2]]
beta <- log(n)/2
poisson_regression_result <- fastcpd(
  data = data,
  beta = beta,
  family = "poisson",
  k = function(x) 0,
  epsilon = 1e-4
```

```
)
```

```
summary(poisson_regression_result)

##
## Call:
## fastcpd(data = data, beta = beta, k = function(x) 0, family = "poisson",
##     epsilon = 1e-04)
##
## [1] "Residuals:"
##       Min        1Q    Median        3Q       Max
## -1.000000 -1.000000 -0.577591  0.372314 12.995381
##
## Change points:
## [1] 150

plot(poisson_regression_result)
```



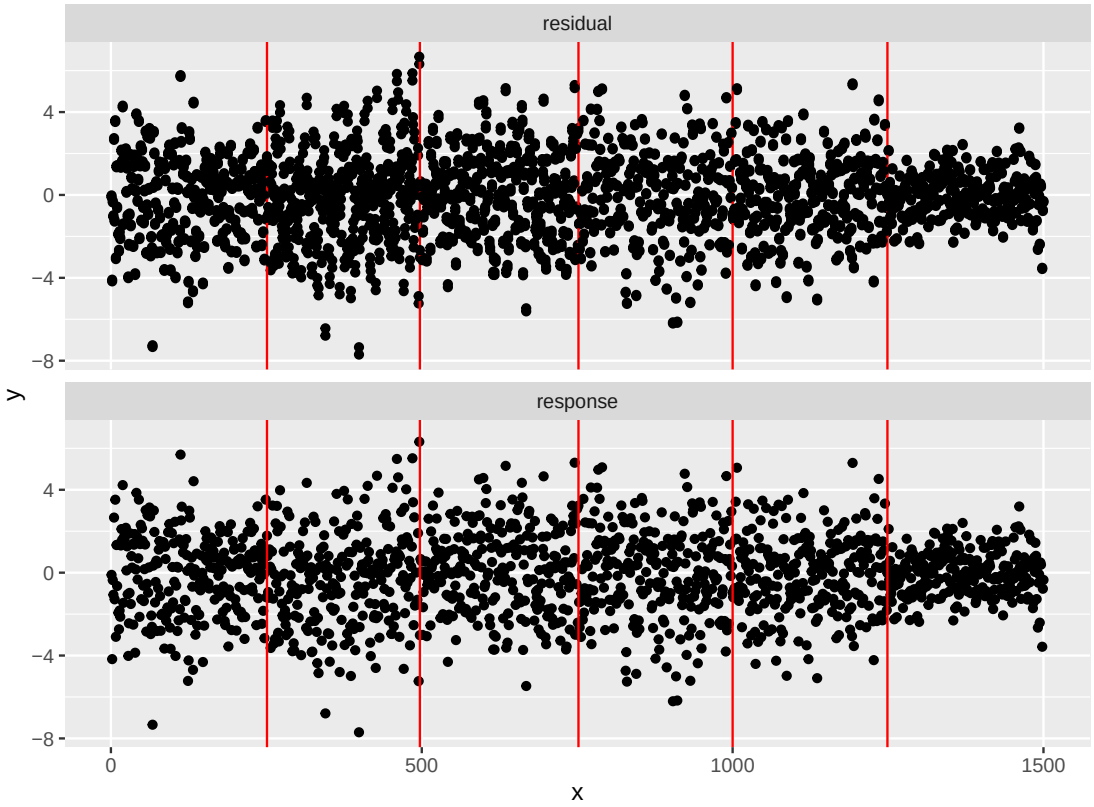
	p=1			p=3			p=5		
	n=300	n=600	n=1500	n=300	n=600	n=1500	n=300	n=600	n=1500
cpc=0	0.57, 4.11s	0.69, 5.24s	0.65, 9.85s	0.14, 3.62s	0.12, 4.08s	0.12, 6.50s	0.46, 7.12s	0.50, 10.33s	0.50, 14.94s
	0.71, 24.17s	0.64, 62.72s	0.64, 413.16s	0.21, 2.49s	0.17, 10.75s	0.15, 45.75s	0.47, 8.53s	0.50, 15.08s	0.50, 33.78s
	0.58, 41.26s	0.41, 145.26s	0.51, 739.57s	0.20, 4.70s	0.16, 16.23s	0.18, 61.29s	0.44, 10.64s	0.50, 15.78s	0.50, 36.28s
	0.74, 38.22s	0.64, 102.88s	0.80, 541.88s	0.15, 20.08s	0.17, 66.15s	0.12, 212.00s	0.08, 16.57s	0.09, 48.26s	0.09, 159.42s
cpc=1	0.88, 3.87s	0.86, 5.36s	0.80, 7.44s	0.64, 0.62s	0.62, 1.30s	0.63, 3.76s	0.98, 6.45s	0.93, 8.36s	1.00, 14.05s
	0.83, 18.51s	0.92, 71.90s	0.95, 442.40s	0.63, 3.56s	0.64, 10.45s	0.63, 42.62s	0.90, 9.38s	0.97, 17.97s	1.00, 32.82s
	0.84, 26.55s	0.86, 117.91s	0.84, 739.52s	0.63, 6.38s	0.62, 18.79s	0.67, 67.00s	0.93, 7.69s	1.00, 14.66s	1.00, 34.34s
	0.90, 26.55s	0.92, 88.83s	0.90, 440.15s	0.69, 17.74s	0.67, 58.88s	0.62, 207.08s	0.58, 19.86s	0.59, 56.14s	0.60, 156.02s
cpc=3	0.92, 0.94s	0.94, 2.19s	0.94, 6.29s	0.83, 0.59s	0.84, 1.25s	0.82, 3.73s	0.76, 7.97s	0.75, 9.53s	0.76, 13.27s
	0.88, 12.19s	0.92, 43.29s	0.94, 219.21s	0.83, 2.69s	0.83, 5.79s	0.85, 38.09s	0.74, 10.30s	0.75, 14.61s	0.75, 33.68s
	0.91, 29.64s	0.95, 110.69s	0.95, 575.92s	0.82, 5.40s	0.81, 12.36s	0.83, 51.83s	0.75, 7.57s	0.75, 16.40s	0.75, 11.59s
	0.96, 26.71s	0.94, 77.56s	0.99, 309.25s	0.87, 15.29s	0.85, 50.04s	0.88, 214.72s	0.81, 16.48s	0.84, 50.63s	0.82, 150.00s
cpc=5	0.92, 1.18s	0.96, 2.33s	0.97, 5.91s	0.87, 0.60s	0.88, 1.34s	0.90, 5.44s	0.67, 6.54s	0.67, 9.73s	0.67, 11.56s
	0.89, 10.81s	0.95, 45.44s	0.96, 204.24s	0.88, 2.68s	0.87, 10.02s	0.86, 36.15s	0.69, 12.01s	0.68, 15.26s	0.67, 30.63s
	0.94, 21.44s	0.96, 76.84s	0.97, 403.26s	0.89, 5.56s	0.87, 11.39s	0.87, 46.96s	0.68, 8.50s	0.67, 13.21s	0.67, 11.73s
	0.96, 22.01s	0.97, 61.44s	0.98, 220.29s	0.91, 16.81s	0.92, 44.63s	0.93, 153.21s	0.89, 14.07s	0.90, 37.67s	0.90, 107.41s

Table 4: Comparison of the algorithms for the poisson regression model in mean.

6. Changes in penalized linear models

We now consider the change-point detection problem in the penalized linear models.

```
summary(lasso_regression_result)

##
## Call:
## fastcpd(data = data, beta = beta, k = function(x) 0, family = "lasso",
##     epsilon = 1e-04)
##
## [1] "Residuals:"
##       Min        1Q    Median        3Q       Max
## -7.348485 -1.209673  0.094231  1.271128  6.672489
##
## Change points:
## [1]  251  497  752 1000 1249

plot(lasso_regression_result)
```



	s=3			s=6	s=10
cpc=0	1.00, 65.24s 0.89, 2496.32s 1.00, 4922.51s 0.00, 0.00s	1.00, 35.13s 1.00, 4380.03s 1.00, 3153.90s 0.00, 0.00s	1.00, 29.76s 1.00, 1849.64s 1.00, 2737.14s 0.00, 0.00s		
cpc=1	1.00, 48.68s 0.98, 2571.30s 0.98, 3551.50s 0.00, 0.00s	0.99, 32.39s 0.99, 3523.90s 1.00, 2679.05s 0.00, 0.00s	0.99, 33.05s 1.00, 2049.49s 0.99, 2218.52s 0.00, 0.00s		
cpc=3	0.96, 53.23s 0.98, 1720.88s 0.98, 2608.36s 0.00, 0.00s	1.00, 40.23s 0.99, 1187.46s 0.99, 2164.48s 0.00, 0.00s	0.99, 32.94s 1.00, 858.13s 1.00, 1440.42s 0.00, 0.00s		
cpc=5	1.00, 33.08s 0.97, 1152.88s 0.98, 1492.22s 0.00, 0.00s	0.99, 26.20s 1.00, 852.29s 1.00, 1267.07s 0.00, 0.00s	1.00, 21.56s 0.99, 587.59s 1.00, 698.19s 0.00, 0.00s		

Table 5: Comparison of the algorithms for the poisson regression model in mean.

7. Changes in user specified models with custom cost functions

7.1. Reproducing built-in models

```
logistic_regression_result <- fastcpd(
  data = data,
  beta = beta,
  family = "binomial",
  k = function(x) 20,
  epsilon = 1e-5
)
logistic_loss <- function(data, theta) {
  u <- c(data[, -1, drop = FALSE] %*% theta)
  sum(-data[, 1] * u + log(1 + exp(u)))
}

logistic_loss_gradient <- function(data, theta) {
  x <- data[-1]
  y <- data[1]
  c(-(y - 1 / (1 + exp(-x %*% theta)))) * x
```

```
}

logistic_loss_hessian <- function(data, theta, hessian) {
  data_x <- data[-1]
  prob <- 1 / (1 + exp(-data_x %*% theta))
  hessian + (data_x %o% data_x) * c((1 - prob) * prob)
}
logistic_regression_result_custom <- fastcpd(
  data = data,
  beta = beta,
  k = function(x) 20,
  epsilon = 1e-5,
  cost = logistic_loss,
  cost_gradient = logistic_loss_gradient,
  cost_hessian = logistic_loss_hessian
)
logistic_regression_result@cp_set

## [1]   64  301  681 1130 1444

logistic_regression_result_custom@cp_set

## [1]  291  685 1163
```

7.2. Changes in Huber regression models

```
n <- 400 + 300 + 400
p <- 3
x <- mvtnorm::rmvnorm(n, mean = rep(0, p), sigma = diag(p))
theta <- rbind(
  mvtnorm::rmvnorm(1, mean = rep(0, p), sigma = diag(p)),
  mvtnorm::rmvnorm(1, mean = rep(3, p), sigma = diag(p)),
  mvtnorm::rmvnorm(1, mean = rep(5, p), sigma = diag(p))
)
theta <- theta[rep(seq_len(3), c(400, 300, 400)), ]
y_true <- rowSums(x * theta)
factor <- c(
  2 * stats::rbinom(400, size = 1, prob = 0.95) - 1,
  2 * stats::rbinom(300, size = 1, prob = 0.95) - 1,
  2 * stats::rbinom(400, size = 1, prob = 0.95) - 1
)
```

```
y <- factor * y_true + stats::rnorm(n)
data <- cbind(y, x)

huber_loss <- function(
  data,
  theta,
  threshold = 1
) {
  residual <- data[, 1] - data[, -1, drop = FALSE] %*% theta
  indicator <- abs(residual) <= threshold
  sum(residual ^ 2 / 2 * indicator + threshold * (abs(residual) - threshold / 2) * (1
}

huber_loss_gradient <- function(data, theta, threshold = 1) {
  residual <- c(data[1] - data[-1] %*% theta)
  if (abs(residual) <= threshold) {
    - residual * data[-1]
  } else {
    - threshold * sign(residual) * data[-1]
  }
}

huber_loss_hessian <- function(
  data,
  theta,
  hessian,
  threshold = 1
) {
  residual <- c(data[1] - data[-1] %*% theta)
  if (abs(residual) <= threshold) {
    hessian + outer(data[-1], data[-1])
  } else {
    hessian + 0.01 * diag(length(theta))
  }
}

huber_regression_result <- fastcpd(
  data = data,
  beta = (p + 1) * log(n),
  cost = huber_loss,
  cost_gradient = huber_loss_gradient,
  cost_hessian = huber_loss_hessian
)
```

Algorithm	Data	Coefficients
Quantile Regression	1st 150	-1.3890396, 1.8959482, -0.5951861
Quantile Regression	2nd 150	-1.3890396, 1.8959482, -0.5951861
Quantile Regression	3rd 150	0.4124297, -0.7807476, -1.481476
Linear Regression	1st 150	-1.3890396, 1.8959482, -0.5951861
Linear Regression	2nd 150	-2.5457644, 2.9560383, 0.5427844
Linear Regression	3rd 150	0.4124297, -0.7807476, -1.481476

Table 6: Comparison of the algorithms for the logistic regression model.

```
summary(huber_regression_result)

##
## Call:
## fastcpd(data = data, beta = (p + 1) * log(n), cost = huber_loss,
##     cost_gradient = huber_loss_gradient, cost_hessian = huber_loss_hessian)
##
## Change points:
## [1] 396 715
```

7.3. Changes in quantile regression models

```
n <- 450
p <- 3
data <- mvtnorm::rmvnorm(n / 5, mean = rep(0, p), sigma = diag(p))[rep(1:(n / 5), each = 5),
theta <- matrix(rnorm(p * 2), 2, p)[rep(1:2, c(n * 2 / 3, n / 3)), ]
xb <- rowSums(data * theta)
xb[1:(n / 3 / 5) * 5 + (n / 3) - 4] <- xb[1:(n / 3 / 5) * 5 + (n / 3) - 4] + 50
xb[1:(n / 3 / 5) * 5 + (n / 3) - 3] <- xb[1:(n / 3 / 5) * 5 + (n / 3) - 3] + 40
xb[1:(n / 3 / 5) * 5 + (n / 3) - 1] <- xb[1:(n / 3 / 5) * 5 + (n / 3) - 1] - 5
xb[1:(n / 3 / 5) * 5 + (n / 3)] <- xb[1:(n / 3 / 5) * 5 + (n / 3)] - 10
data <- cbind(xb, data)
```

Verify that the data contains three parts with two segments. The first segment is from 1 to 300 and the second segment is from 301 to 450. By doing piecewise quantile regression and linear regression, we can verify that coefficients of quantile regression in the first 150 observations are the same as those in the second 150 observations. The coefficients of linear regression in the first 150 observations are different from those in the second 150 observations.

Let see what is the output if we misspecify the model.

```
# linear_regression_result <- fastcpd(
#   data = data,
#   beta = log(n) / 2,
#   segment_count = 10,
#   trim = 0.025,
#   k = function(x) 0,
#   family = "gaussian",
#   epsilon = 1e-5
# )

# summary(linear_regression_result)
```

The output is not correct. Now let see what is the output if we specify the model to be a quantile regression model using the custom cost function.

```
quantile_loss <- function(
  data,
  theta,
  quant = 0.5,
  smoothing = 0.25
) {
  residual <- data[, 1] - data[, -1, drop = FALSE] %*% theta
  mean(residual * (quant - (residual < 0)))
}

quantile_loss_gradient <- function(
  data, theta, quant = 0.5, smoothing = 0.25
) {
  c(quant - 1 / (1 + exp(c(data[1] - data[-1] %*% theta) / smoothing))) * data[-1]
}

quantile_loss_hessian <- function(
  data,
  theta,
  hessian,
  quant = 0.5,
  smoothing = 0.25
) {
  # hessian - exp(c(data[1] - data[-1] %*% theta) / smoothing) / (smoothing * (1 + ex
  hessian + 0.01 * diag(length(theta))
}

# print(fastcpd(
```

```
#   data = data,
#   beta = log(n) / 2,
#   segment_count = 5,
#   trim = 0.025,
#   k = function(x) 0,
#   cost = quantile_loss,
#   cost_gradient = quantile_loss_gradient,
#   cost_hessian = quantile_loss_hessian
# ))

# print(fastcpd(
#   data = data,
#   beta = log(n) / 2,
#   segment_count = 5,
#   trim = 0.025,
#   k = function(x) 4,
#   cost = quantile_loss,
#   cost_gradient = quantile_loss_gradient,
#   cost_hessian = quantile_loss_hessian
# ))

# print(fastcpd(
#   data = data,
#   beta = log(n) / 2,
#   segment_count = 5,
#   trim = 0.025,
#   k = function(x) 9,
#   cost = quantile_loss,
#   cost_gradient = quantile_loss_gradient,
#   cost_hessian = quantile_loss_hessian
# ))

# benchmarked <- microbenchmark::microbenchmark(
#   "k = 1" = fastcpd(
#     data = data,
#     beta = log(n) / 2,
#     segment_count = 5,
#     trim = 0.025,
#     k = function(x) 0,
#     cost = quantile_loss,
#     cost_gradient = quantile_loss_gradient,
#     cost_hessian = quantile_loss_hessian
#   ),
```

Figure 1: Benchmarking results for the quantile regression model.

```
#   "k = 5" = fastcpd(
#     data = data,
#     beta = log(n) / 2,
#     segment_count = 5,
#     trim = 0.025,
#     k = function(x) 4,
#     cost = quantile_loss,
#     cost_gradient = quantile_loss_gradient,
#     cost_hessian = quantile_loss_hessian
#   ),
#   "k = 10" = fastcpd(
#     data = data,
#     beta = log(n) / 2,
#     segment_count = 5,
#     trim = 0.025,
#     k = function(x) 9,
#     cost = quantile_loss,
#     cost_gradient = quantile_loss_gradient,
#     cost_hessian = quantile_loss_hessian
#   ),
#   times = 2
# )
# print(benchmarked)
```

7.4. Changes in mean shift models

Now let's consider the mean shift model using custom cost function. Since a mean shift model is a special case of a linear regression model, by specifying a response variable we can use the linear regression model to detect the change points.

```
p <- 1
data <- rbind(
  mvtnorm::rmvnorm(100, mean = rep(0, p), sigma = diag(1, p)),
  mvtnorm::rmvnorm(100, mean = rep(5, p), sigma = diag(1, p))
)
data <- cbind(c(data[1:100, , drop = FALSE] %*% colMeans(data[1:100, , drop = FALSE]))

mean_shift_linear_result <- fastcpd(
  data = data,
```

```
  beta = log(200) / 2,
  segment_count = 3,
  trim = 0.025,
  k = function(x) 0,
  family = "gaussian",
  epsilon = 1e-5
)
summary(mean_shift_linear_result)

##
## Call:
## fastcpd(data = data, beta = log(200)/2, segment_count = 3, trim = 0.025,
##     k = function(x) 0, family = "gaussian", epsilon = 1e-05)
##
## [1] "Residuals:"
##           Min            1Q        Median            3Q           Max
## -1.387779e-17 -4.336810e-19  0.000000e+00  0.000000e+00  2.775558e-17
##
## Change points:
## [1] 100
```

Now let's see what would be the output if we use the custom cost function.

```
data <- data[, -1, drop = FALSE]

mean_loss <- function(data, theta) {
  norm(sweep(data, 2, theta), type = "F") ^ 2 / 2
}

mean_loss_gradient <- function(data, theta) {
  theta - data
}

mean_loss_hessian <- function(data, theta, hessian) {
  hessian + diag(p)
}

mean_loss_result <- fastcpd(
  data = data,
  beta = 10,
  segment_count = 4,
  trim = 0.025,
  k = function(x) 0,
```

```
  p = p,
  cost = mean_loss,
  cost_gradient = mean_loss_gradient,
  cost_hessian = mean_loss_hessian
)

summary(mean_loss_result)

##
## Call:
## fastcpd(data = data, beta = 10, segment_count = 4, trim = 0.025,
##     k = function(x) 0, p = p, cost = mean_loss, cost_gradient = mean_loss_gradient
##     cost_hessian = mean_loss_hessian)
##
## Change points:
## [1] 100
```

8. Adaptive number of epochs

In this section, we will discuss how to use the adaptive number of epochs feature in **fastcpd**. The adaptive number of epochs feature is designed to automatically determine the number of epochs to run the algorithm. The criteria for determining the number of epochs is based on the following function:

$$K = \min(\max(\boldsymbol{k}, \lceil \frac{1000}{\text{segment length}} \rceil), 10) \tag{2}$$

where K is the number of epochs to run the algorithm, $\boldsymbol{k}$ is the number of epochs specified by the user parameter `sgd_k`, and `segment length` is the length of the segment. The goal is to upper limit the number of epochs to 10 when the segment length is small and to lower limit the number of epochs to $\boldsymbol{k}$. The transition between the upper limit and lower limit is calculated by $\lceil 1000/\text{segment length} \rceil$.

```
k <- function(x, n = 1500) {
  if (x < n / 10 * 1 / 3) 3
  else if (x < n / 10 * 2 / 3) 2
  else if (x < n / 10) 1
  else 0
}
```

9. Summary and discussion

Our package **fastcpd** provides a fast and flexible implementation of change point detection algorithms. The package is designed to be easy to use and to provide a wide range of options for the user. The package is implemented in R and is available on GitHub.

Computational details

If necessary or useful, information about certain computational details such as version numbers, operating systems, or compilers could be included in an unnumbered section. Also, auxiliary packages (say, for visualizations, maps, tables, ...) that are not cited in the main text can be credited here.

The results in this paper were obtained using R 4.1.3 with the **MASS** 7.3.58.1 package. R itself and all packages used are available from the Comprehensive R Archive Network (CRAN) at https://CRAN.R-project.org/.

Acknowledgments

All acknowledgments (note the AE spelling) should be collected in this unnumbered section before the references. It may contain the usual information about funding and feedback from colleagues/reviewers/etc. Furthermore, information such as relative contributions of the authors may be added here (if any).

References

Killick R, Eckley I (2014). "changepoint: An R package for changepoint analysis." *Journal of statistical software*, **58**(3), 1–19.

Killick R, Fearnhead P, Eckley IA (2012). "Optimal detection of changepoints with a linear computational cost." *Journal of the American Statistical Association*, **107**(500), 1590–1598.

R Core Team (2017). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Zhang X, Dawn T (2022). "Sequential Gradient Descent and Quasi-Newton's Method for Change-Point Analysis." *arXiv preprint arXiv:2210.12235*.

A. More technical details

Appendices can be included after the bibliography (with a page break). Each section within the appendix should have a proper section title (rather than just *Appendix*).

For more technical style details, please check out JSS's style FAQ at https://www.jstatsoft.org/pages/view/style#frequently-asked-questions which includes the following topics:

- Title vs. sentence case.
- Graphics formatting.
- Naming conventions.
- Turning JSS manuscripts into R package vignettes.
- Trouble shooting.
- Many other potentially helpful details...

B. Using BibTeX

References need to be provided in a BibTeX file (`.bib`). All references should be made with `\cite`, `\citet`, `\citep`, `\citealp` etc. (and never hard-coded). This commands yield different formats of author-year citations and allow to include additional details (e.g., pages, chapters, ...) in brackets. In case you are not familiar with these commands see the JSS style FAQ for details.

Cleaning up BibTeX files is a somewhat tedious task – especially when acquiring the entries automatically from mixed online sources. However, it is important that informations are complete and presented in a consistent style to avoid confusions. JSS requires the following format.

- JSS-specific markup (`\proglang`, `\pkg`, `\code`) should be used in the references.
- Titles should be in title case.
- Journal titles should not be abbreviated and in title case.
- DOIs should be included where available.
- Software should be properly cited as well. For R packages `citation("pkgname")` typically provides a good starting point.

Affiliation:

Contact Author
Journal of Statistical Software
and
Department of Statistics
Faculty of Economics and Statistics
Universität Innsbruck
Universitätsstr. 15
6020 Innsbruck, Austria
E-mail: Achim.Zeileis@R-project.org
URL: https://www.zeileis.org/

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd