blblm

Package Description

This package has functions for running little bag of bootstraps on linear regression models and logistic regression model on datasets.

Bag of little bootstrap (BLB) is a procedure that incorporates bootstrap and resampling to prodce a computationally efficient, yet robust way of estimation.

For the linear regression model (blblm), users can determine the regression coefficients (beta heads), the estimate of variance of errors (sigma), confidence interval and prediction interval on new data.

For logistic regression model(blblr), users can determine the regression coefficients (beta heads), as well as confidence interval on particular coefficient or prediction interval on coefficients.

For both function inside the package, users can also decide if they want to use parallel computing by changing the option in function.

Linear Regression (blblm)

Load the package and test it out:

```
library(blblm)
```

Design

The primary improvement added methods that support parallelization upon users' decision.

Parallelization

Users are able to decide if they want to compute the estimation parallely vs sequentially. They are also welcome to choose the thread numbers to decide how many threads they want to run them.

```
blblm <- function(formula, data, m = 10, B = 5000, parallel = FALSE, threads = 4) {
  # the function split_data put them into m chunk
  data_list <- split_data(data, m)
  if (parallel){
    # if parallel is TRUE, use planner to run multiple threads
    suppressWarnings(plan(multiprocess, workers = threads))
    # calculate the estimate parallely
    estimates <- future_map(
      data_list,
      ~ lm_each_subsample(formula = formula, data = ., n = nrow(data), B = B))
  }else{
    # by default, parallel = FALSE
    # map is computed sequentially
    estimates <- map(
      data_list,
```

```
      ~ lm_each_subsample(formula = formula, data = ., n = nrow(data), B = B))
  }
  res <- list(estimates = estimates, formula = formula)
  class(res) <- "blblm"
  invisible(res)
}
```

plan(multiprocess, workers = threads) enables parallelization and in this case, the default threads/planner is at 4. if the users select parallel = TRUE. plan(multiprocess, workers = threads) is accompanied by future_map from library(furrr) for the computing

Benchmarking

Generating data for the following examples:

```
set.seed(100)
n = 1000
y = rnorm(n, mean = 10, sd = 2)
x = 0.5 * y + rnorm(n, mean = 0, sd = 0.25)
z = 0.3 * y + rnorm(n, mean = 0, sd = 0.25)
data = as.data.frame(cbind(x,y,z))
```

Parallelization

```
#> Loading required package: future
#> # A tibble: 2 x 6
#>   expression                                                          min
#>   <bch:expr>                                                        <bch:t>
#> 1 blblm(y ~ x * z, data, m = 3, B = 100)                           60.93ms
#> 2 blblm(y ~ x * z, data, m = 3, B = 100, parallel = TRUE, threads = 4)   6.71s
#> # … with 4 more variables: median <bch:tm>, `itr/sec` <dbl>,
#> #   mem_alloc <bch:byt>, `gc/sec` <dbl>
```

We can observe that by performing parallelization, the memory allocated is less and the number of iteration is larger using parallelization. The difference can be observe with larger amount of bootstraps and dataset.

Coefficients and Sigma

Users can also decide if they want to compute parallely and define the number of threads themselves when choosing parallel = TRUE.

95 % Confidence Interval for Variable Coefficients

The confidence interval can change the confidence level, decide if they want to compute parallely and define the number of threads themselves when choosing parallel = TRUE.

```
#> # A tibble: 4 x 6
#>   expression                                                    min
#>   <bch:expr>                                                  <bch:t>
#> 1 confint(fit_lm, level = 0.95)                              131.4ms
#> 2 confint(fit_lm_par, level = 0.95)                          120.6ms
```

```
#> 3 confint(fit_lm, level = 0.95, parallel = TRUE, threads = 4)       17.9s
#> 4 confint(fit_lm_par, level = 0.95, parallel = TRUE, threads = 4)   18.4s
#> # … with 4 more variables: median <bch:tm>, `itr/sec` <dbl>,
#> #   mem_alloc <bch:byt>, `gc/sec` <dbl>
```

By computing parallely with the model, we see a lot less memory used compared to without using parallel computing. Time used is slightly faster. By calling the workers multiple times might be what is causing the time to run longer. But that is up to user's choice implement.

95% Prediction Interval

The prediction interval set confidence = FALSE by default. User can change the prediction level, decide if they want to compute parallely and define the number of threads themselves when choosing parallel = TRUE.

```
benchmark3 = bench::mark(
  predict(fit_lm, data.frame(x = c(0.2, 0.5), z = c(0.5, 0.7))),
  predict(fit_lm_par, data.frame(x = c(0.2, 0.5), z = c(0.5, 0.7))),
  predict(fit_lm, data.frame(x = c(0.2, 0.5), z = c(0.5, 0.7)), parallel = TRUE, threads =
        4),
  predict(fit_lm_par, data.frame(x = c(0.2, 0.5), z = c(0.5, 0.7)), parallel = TRUE, threads
        = 4),
  check = FALSE
)
#> Warning: Some expressions had a GC in every iteration; so filtering is disabled.

benchmark3
#> # A tibble: 4 x 6
#> # … with 6 more variables: expression <bch:expr>, min <bch:tm>,
#> #   median <bch:tm>, `itr/sec` <dbl>, mem_alloc <bch:byt>, `gc/sec` <dbl>
```

Logistic Regression (blblr)

Similarly, we have BLB for logistic regression. Users are also allowed to decide if they want to do parallel computing for the estimation. They are also allowed to choose the thread numbers with how many threads they want to run them.

```
blblr <- function(formula, data, m = 10, B = 5000, parallel = FALSE, threads = 4){
  if (parallel){
    # assign workers if parallel = TRUE
    suppressWarnings(plan(multiprocess, workers = threads))
    options(future.rng.onMisuse = "ignore")
    data_list <- split_data(data, m)
    estimates <- future_map(data_list,~lr_each_subsample(formula = formula, data = ., n =
        nrow(data), B = B))
  }else{
    # else parallel = FALSE
    # estimations are running sequentially
    data_list <- split_data(data, m)
    estimates <- map(data_list, ~lr_each_subsample(formula = formula, data = ., n =
        nrow(data), B = B))
  }
  res <- list(estimates = estimates, formula = formula)
  class(res) <- "blblr"
```

```
  invisible(res)
}
```

Benchmarking

Data

First, we generate the sample data needed for running benchmark: (note that the y needs to be in binary form in order to run logistic regression)

```
set.seed(200)
n = 1000
y = rbinom(n, size=1, prob=0.05)
x = 0.5 * y + rnorm(n, mean = 0, sd = 0.25)
z = 0.3 * y + rnorm(n, mean = 0, sd = 0.25)
mydata = as.data.frame(cbind(x,y,z))
```

Parallelization

```
#> # A tibble: 2 x 6
#>   expression
#>   <bch:expr>
#> 1 blblr(y ~ x * z, mydata, m = 3, B = 100)
#> 2 blblr(y ~ x * z, mydata, m = 3, B = 100, parallel = TRUE, threads = 4)
#> # … with 5 more variables: min <bch:tm>, median <bch:tm>, `itr/sec` <dbl>,
#> #   mem_alloc <bch:byt>, `gc/sec` <dbl>
```

We can observe that by performing parallelization, again the memory allocated is comparatively less and the number of iteration is larger by computing parallely. Although the time migh not show much difference, larger gap may be observed with larger amount of bootstraps and dataset.

Coefficients and Probabilities(Odds)

Users can also decide if they want to compute parallely and define the number of threads themselves when choosing parallel = TRUE.

95 % Confidence Interval for Variable Coefficients

Users can change confidence interval by changing the confidence level. They can also decide if they want to compute parallely and define the number of threads themselves when choosing parallel = TRUE. In the case of logistic regression, odds are by default for calculating the confidence interval. If users are interested in computing the coefficient, they can change the parameter odds = FALSE to achieve that.

```
#> # A tibble: 4 x 6
#>   expression                                                    min median
#>   <bch:expr>                                                  <bch:t> <bch:>
#> 1 confint(fit_lr, level = 0.95)                               139.1ms  148ms
#> 2 confint(fit_lr_par, level = 0.95)                           117.7ms  120ms
#> 3 confint(fit_lr, level = 0.95, parallel = TRUE, threads = 4)   19.1s  19.1s
```

```
#> 4 confint(fit_lr_par, level = 0.95, parallel = TRUE, threads = 4)     18s    18s
#> # … with 3 more variables: `itr/sec` <dbl>, mem_alloc <bch:byt>, `gc/sec` <dbl>
```

With parallel computing the confidence interval, we see a lot less memory used compared to without using parallel computing. Time used is slightly faster. Here also compares the difference using the different method of computing the model. Although that time is not taken into consideration, we can still observe a slight difference in the usage of time and memory for both function. By calling the workers multiple times might be what is causing the time to run longer. But that is up to user's choice implement.

95% Prediction Interval

The prediction interval set confidence = FALSE by default. User can change the prediction level, decide if they want to compute parallely and define the number of threads themselves when choosing parallel = TRUE. By default, the calculated interval is odds = TRUE, which means that if users want the prediction intervals for coefficient, they will have to set the odds = FALSE in the parameter.

```
#> # A tibble: 4 x 6
#> # … with 6 more variables: expression <bch:expr>, min <bch:tm>,
#> #   median <bch:tm>, `itr/sec` <dbl>, mem_alloc <bch:byt>, `gc/sec` <dbl>
```