blblm

1 Introduction

An issue that often occurs when we attempt to quantify the uncertainty that is associated with a given estimator or statistical learning method is that in many scenarios the distribution of the data may not be known. Without knowing the distribution of our data, quantifying the uncertainty can be quite difficult. This is where bootstrap proves it's worth. It is possible to simulate new data multiple times which can provide us with a good estimate of the distribution of our data. From there we can quantify estimates such as standard error of a coefficient or a confidence interval for that coefficient. But this raises another issue. This method doesn't quite work when our data is separated. This is where bag of little bootstraps comes in. It is possible to split the data into sub-samples of data and run bootstrap multiple times on each subset of that data. This will provide us with statistics which we can then reduce down to our wanted result. This package implements bag of little bootstraps for a well known regression function, lm.

2 Testing Our Package

blblm(formula, data, m = 10, B = 5000, parallel = FALSE)

We will be using a randomly generated data set with one response variable and two predictors. The main function that this package contains is a function called `blblm`. The function can take in 5 different arguments. The first argument is `formula` on which you want to regress your data. The second argument `data`, this should be where our data is coming from. Our third argument is where this function will start to differ from the original lm function. Our third argument, `m` denotes how many times we want to split our data. Each of these splits will be bootstrapped. Our forth argument denotes how many times we want our split data to be bootstrapped. And finally we have our fifth argument, `Parallel`, which gives the users an option to use parallelization. If the user wishes to use more than one core, that user will have to prepare their workers beforehand, and set `parallel = TRUE`. Here we'll show some examples of how to use this function. First we will make use of the function without multiple cores.

```
### RANDOMLY GENERATED DATA
set.seed(121)
n = 500
y = rnorm(n,mean = 10,sd = 2)
x = .5 * y + rnorm(n, mean = 0, sd = .25)
z= 0.3 * y + rnorm(n, mean = 0, sd = .25)
df = data.frame(y =y , x =x , z= z)
```

Below we can see the data following the format explained above. Our y variable is our response, while x and z are our predictors. We split the data into 10 subsections and we bootstrap each of those subsections 1000 times.

```
fit = blblm(y~x+z, data = df, m = 10, B =1000)
```

Below is the same implementation, but instead of using one core, here we will use multiple cores. Note, the number of workers you set is machine dependent. It is not the same for everyone.

```
plan(multiprocess,workers = 12)
options(future.rng.onMisuse = "ignore")
fit2 = blblm(y~x+z, data = df, m = 10, B =1000, parallel = TRUE)
```

sigma.blblm(object, confidence = FALSE, level = 0.95, …)

The main use of this function is to return the average sigma value of your bootstrapped data. The first argument, `object` should be your fit model after running blblm on your data. The second argument, `confidence` is a boolean which lets the users decide if they want a confidence interval of the sigma value rather than just a single estimate of the value. The third argument, `level` is the level for your confidence interval and only applies if you select true for `confidence`. Below we will demonstrate how the function works.

Below we see just a single estimate of our sigma value, which is an average of our bootstraps

```
sigma(fit)
#> [1] 0.408327
```

Below we see rather a confidence interval with a level of 0.95, which gives us a confidence interval of the estimate for our sigma value. The function also provides the estimated sigma value from above.

```
sigma(fit,confidence = TRUE, level = 0.95)
#>     sigma       lwr       upr
#> 0.4083270 0.3852652 0.4312812
```

confint.blblm(object, parm = NULL, level = 0.95, …)

The main use of this function is to return a confidence interval for our estimated coefficients. The first argument `object` is our fit model. The second is `parm`, which allows the users to set a parameter on which predictors they want their confidence interval to return. This parameter can also be left as null if the user does not want specific parameters. The third argument,`level` allows users to edit the level of the confidence interval. Below we will demonstrate how this function works.

Below we use the function with parameter x to get a confidence interval of the estimated coefficients for just x

```
confint(fit, c("x"), level = 0.95)
#>       2.5%    97.5%
#> x 1.341932 1.502729
```

Below we use the function with no parameters to get the confidence interval of all the estimated coefficients.

```
confint(fit, level = 0.95)
#>        2.5%     97.5%
#> x 1.3419322 1.5027295
#> z 0.7011934 0.9594575
```

coef.blblm(object, …)

The main use of this function is to return the estimated coefficients derived from our bootstrapped data. This function takes one parameter, `object`, which is simply our fit model. Below we will demonstrate how this function works.

```
coef(fit)
#> (Intercept)           x           z
#>   0.3879749   1.4213177   0.8304002
```

predict.blblm(object, new_data, confidence = FALSE, level = 0.95, …)

The main use of this function is to predict on new data given the fit model that was produced from the blblm function. This function takes in four parameters. The first parameter, `object` is our fit model. The second parameter, `new_data` is the data on which we want to predict on with our given model. The third parameter is a boolean, `confidence` which allows the user to decide whether they want their predictions to be a confidence interval rather than just a predicted response. The forth argument, `level` is the level for the confidence interval and is only relevant if `confidence` is set to true. Below we will demonstrate how this function works.

```
set.seed(121)
n1 = 100
y1 = rnorm(n,mean = 9,sd = 2)
x1 = .5 * y + rnorm(n, mean = 0, sd = .25)
z1= 0.3 * y + rnorm(n, mean = 0, sd = .25)
df1 = data.frame(y =y1 , x =x1 , z= z1)
```

Below we show the function without the confidence intervals, along with the first 6 results

```
precicted_values = predict(fit,df1)
head(precicted_values)
#>         1         2         3         4         5         6
#> 10.060687  9.911528  8.903987  9.671634  8.812000 12.666596
```

Below we show the functions with `confidence` set to true, along with the first 6 results

```
predicted_values_conf = predict(fit, df1,confidence = TRUE,level = 0.95)
head(predicted_values_conf)
#>         fit       lwr       upr
#> 1 10.060687 10.023295 10.097792
#> 2  9.911528  9.850655  9.972626
#> 3  8.903987  8.861088  8.947474
#> 4  9.671634  9.604877  9.736989
#> 5  8.812000  8.770377  8.853358
#> 6 12.666596 12.599695 12.734986
```

3 Single Core vs Multi-Core

We have now demonstrated how the functions work. We next want to show the difference in efficiency when using multiple cores. Again we are going to run the blblm function with and without multiple cores and we will benchmark their speeds to compare.

```
plan(multiprocess,workers = 12)
options(future.rng.onMisuse = "ignore")
bench::mark(
  blblm(y~x+z,data = df,m = 10, B =5000),
  blblm(y~x+z,data = df,m = 10, B =5000,parallel = TRUE),
  check = FALSE
)
#> Warning: Some expressions had a GC in every iteration; so filtering is disabled.
#> # A tibble: 2 x 6
#>   expression                                                    min
#>   <bch:expr>                                                <bch:tm>
#> 1 blblm(y ~ x + z, data = df, m = 10, B = 5000)               2.42s
#> 2 blblm(y ~ x + z, data = df, m = 10, B = 5000, parallel = TRUE) 941.08ms
#> # ... with 4 more variables: median <bch:tm>, `itr/sec` <dbl>,
#> #   mem_alloc <bch:byt>, `gc/sec` <dbl>
```

As we can see from our results, the use of parallelization greatly improves our efficiency and is very recommended if datasets are of a large size. The use of multiple cores should improve speed. Again we ask that you set up your workers beforehand if you wish to use multiple cores.