# final-project-vignette

## Introduction

For our final project, we were asked to improve Randy's package for creating a linear model using the Bag of Little Bootstraps (BLB) algorithm. In this vignette, I will summarize the different edits that I made to the package in order to improve the package, mostly with regards to its efficiency. The efficiency of the package can be improved through a few methods, namely, paralellization and implementing C++ code into the package.

## Overview of Package

As previously mentioned, the purpose of this package is to use the BLB method to create a linear model. To familiarize ourselves with the functionality of the package, let us do a quick example where we demonstrate the functions features with the `USArrests` dataset

```
library(blblm)
library(future)
library(bench)
```

We first fit the model using the `blblm` function:

```
fit <- blblm(Murder ~  Assault + UrbanPop, data = USArrests, m = 7, B = 1000)
coef(fit)
#> (Intercept)     Assault    UrbanPop 
#>  3.45827457  0.04922251 -0.02070660
```

The SSE is given by:

```
sigma(fit)
#> [1] 1.505991
```

We can furthermore find the coefficients and make 95% confidence intervals for the coefficients the linear model. Note that the `coef` and `confint` functions have been made within the package to run on blblm class type.

```
confint(fit, c("Assault", "UrbanPop"))
#>                 2.5%      97.5%
#> Assault   0.04152002 0.05608303
#> UrbanPop -0.05443763 0.01886159
```

Let us use our model to now make predictions, for example predicting the average number of Murders in a particular area with 150 or 200 assaults and percent Urban population equal to 50 or 75, respectively

```
predict(fit, data.frame(Assault = c(150, 200), UrbanPop = c(50, 75)), confidence = TRUE)
#>          fit       lwr      upr
#> 1  9.806321  8.884119 10.77149
#> 2 11.749781 11.184631 12.40338
```

Above in the "fit" column are the predticted values along with the lower and upper bounds of their prediction intervals

## Implementing Parallelization

To implement paralellization in the package, I decided to add an argument to the `blblm()` function, `use_parallel`, to indicate whether one would like to parallelize the process. Here is an example of the updated function with paralellization processes.

I implemented paralellization in the following way:

```
blblm <- function(formula, data, m = 10, B = 5000, use_parallel = FALSE, num_workers = detectCores(),
seed = 2020) {
  set.seed(seed)
  data_list <- split_data(data, m)
  if (use_parallel == TRUE){
    suppressWarnings(plan(multiprocess, workers = num_workers))
    options(future.rng.onMisuse = "ignore", seed = seed)
    estimates <- future_map(
      data_list,
      ~ lm_each_subsample(formula = formula, data = ., n = nrow(data), B = B))
  } else{
    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)
}
```

Notice the use of `future_map()` instead of `map()`. The function `future_map()` will use lexical analyisis to assign taks to workers, and will be used if the user indicates the argument `use_parallel = true` in the `blblm` function. The user can also indicate a particular seed.

I will now create a test dataset to show the efficiiency differences when setting `use_parallel = FALSE` and `use_parallel = TRUE`. I will be choosing to split the data into 10 parts and run B = 5000 bootstrap samples.

```
x <- runif(10000)
y <- runif(10000)
df <- data.frame(x = x, y = y)
bench::mark(
  blblm(y ~x, data = df, m = 10, use_parallel = FALSE),
  blblm(y ~ x, data = df, m = 10, use_parallel = TRUE),
  check = FALSE
)
#> Warning: Some expressions had a GC in every iteration; so filtering is disabled.
#> # A tibble: 2 x 6
#>   expression                                              min median `itr/sec`
#>   <bch:expr>                                            <bch> <bch:>     <dbl>
#> 1 blblm(y ~ x, data = df, m = 10, use_parallel = FALSE) 13.9s  13.9s    0.0720
#> 2 blblm(y ~ x, data = df, m = 10, use_parallel = TRUE)   9.4s   9.4s    0.106
#> # ... with 2 more variables: mem_alloc <bch:byt>, `gc/sec` <dbl>
```

The execution of the `blblm` function with `use_parallel = TRUE` is over 2 times faster!

## Fixing Documentation

To fix the documentation of the package, I clicked inside each of the functions then went to Code -> "Insert Roxygen Skeleton" within RStudio. By doing this I was able to provide insight into how each of the functions worked and describe the parameters of the function. Furthermore, I gave samples of how to use each of the functions within the documentations as well. Here is an example of how I documented the `print.blblm` function:

```
#' Print the model
#'
#' Present the model that has been created, including the names of both dependent and independent
variables.
#'
#' @param x blblm model
#'
#' @param ... additional arguments
#'
#' @export
#' @examples fit <- blblm(mpg ~ wt * hp, data = mtcars, m = 3, B = 100, use_parallel = FALSE)
#' print(fit)
#' @method print blblm
print.blblm <- function(x, ...) {
  cat("blblm model:", capture.output(x$formula))
  cat("\n")
}
```

This was done for each of the functions that were exported from the package.

## Writing Tests

To write tests, I implemented the `testthat` package in R, and created "test-function.R" files in the "testthat" folder within the package. I ran the functions beforehand to look at the expected output and created that output from scratch. Ensuring that the seed I ran was fixed, I checked to see if the output was the same as when I ran the`blblm` function with specific parameters. Here is an example of one of the function tests for `confint`:

```
test_that("confint for blblm works", {
  fit <- blblm(Murder ~  Assault + UrbanPop, data = USArrests, m = 7, B = 1000)
  test <- matrix(data = round(c(0.04152002, -0.05443763, 0.05608303, 0.01886159), 8), ncol = 2, nrow
= 2)
  rownames(test) <- c("Assault", "UrbanPop")
  colnames(test) <- c("2.5%", "97.5%")
  expect_equal(test, round(confint(fit, c("Assault", "UrbanPop")), 8))
})
```

I did this for each of the exported functions.

## Implementing C++

I attempted to make the `lm1` function faster by implementing a C++ function into the package. I edited the function given at https://github.com/RcppCore/RcppArmadillo/blob/master/src/fastLm.cpp to account for weighted linear regression. Here is the function:

```
//' Compute the weighted linear regression solution
//'
//' @param X a matrix containing the values predictor variables
//' @param y a vector of response variable data values
//' @param w a vector of weights for the regression
//' @export
// [[Rcpp::export]]
List fastLm_w(const arma::mat& X, const arma::colvec& y, const arma::colvec& w) {
  int n = X.n_rows, k = X.n_cols;
  arma::mat w_mat = arma::diagmat(w);
  arma::colvec coef = arma::inv(arma::trans(X)*w_mat*X) * arma::trans(X) * w_mat * y;  // fit model y
~ X
  arma::colvec res  = y - X*coef;          // residuals

  // std.errors of coefficients
  double s2 = std::inner_product(res.begin(), res.end(), res.begin(), 0.0)/(n - k);

  arma::colvec std_err = arma::sqrt(s2 * arma::diagvec(arma::pinv(arma::trans(X)*X)));

  return List::create(Named("coefficients") = coef,
                      Named("stderr")       = std_err,
                      Named("df.residual")  = n - k);
}
```

Now let us benchmark our original function with `fast = TRUE`, a new parameter I created in the `blblm` function, and compare the efficiency:

```
x <- runif(100)
y <- runif(100)
df <- data.frame(x = x, y = y)
bench::mark(
  blblm(y ~x, data = df, m = 10, fast = FALSE, B = 1000),
  blblm(y ~ x, data = df, m = 10, fast = TRUE, B = 1000),
  check = FALSE
)
#> # A tibble: 2 x 6
#>   expression                                                min median `itr/sec`
#>   <bch:expr>                                           <bch> <bch:>     <dbl>
#> 1 blblm(y ~ x, data = df, m = 10, fast = FALSE, B = 1000) 504ms  504ms      1.98
#> 2 blblm(y ~ x, data = df, m = 10, fast = TRUE, B = 1000)  166ms  180ms      5.55
#> # ... with 2 more variables: mem_alloc <bch:byt>, `gc/sec` <dbl>
```

As we can see, the second function (which is the one which implements the C++ function) is nearly 2 times faster!