`CIPerm`: An R Package for Computationally Efficient Confidence Intervals from Permutation Tests

Jerzy Wieczorek, Emily Tupaj
Department of Statistics, Colby College, Waterville, ME

Abstract

We carry out computationally-efficient construction of confidence intervals from permutation tests for simple differences in means. When using a permutation test to evaluate $H_0 : \mu_Y - \mu_X = 0$, the naive approach to constructing a CI for the $\mu_Y - \mu_X$ parameter would require carrying out many new permutation tests at different values of $\mu_Y - \mu_X$. Instead, our package constructs a CI cheaply using a single set of permutations, making such CIs feasible for much larger datasets.

Our R package implements methods from Nguyen, Minh D., (2009). "Nonparametric Inference Using Randomization and Permutation Reference Distribution and Their Monte-Carlo Approximation" (2009). Portland State University, *Dissertations and Theses.* Paper 5927. https://doi.org/10.15760/etd.7798

How the methodology works

Let *Y* be a vector of *n* observations from one group, and let *X* be *m* observations from the other group. The difference in sample means is

$$t_0 = \frac{\sum_{j=1}^{n} Y_j}{n} - \frac{\sum_{i=1}^{m} X_i}{m}$$

In the standard permutation or randomization test, at each permutation let *k* denote the number of swapped labels (i.e., *k* of the *X*'s are assigned to the *Y* group and vice versa). Then the permuted test statistics have the form

$$t_{k,d} = \frac{\sum_{i=1}^{k} X_i + \sum_{j=k+1}^{n} Y_j}{n} - \frac{\sum_{j=1}^{k} Y_j + \sum_{i=k+1}^{m} X_i}{m}$$

where *d* indexes over different permutations with the same value of *k*.

- Naive approach: For a given mean difference Δ, you could test $H_0 : \mu_Y - \mu_X = \Delta$ vs $H_A : \mu_Y - \mu_X \neq \Delta$ by replacing all Y_j values with $Y_{j,\Delta} = Y_j - \Delta$ and running the usual test for a null difference of 0. If you do this many times for many different Δ values, then the failed-to-reject values form a confidence interval.

- Nguyen's method: His Theorem 1 and its Corollaries show that the quantiles of $w_{k,d}$ give you the CI endpoints directly, where

$$w_{k,d} = \frac{t_0 - t_{k,d}}{k\left(\frac{1}{n} + \frac{1}{m}\right)}$$

So you *don't* have to try out different Δ values or run new permutations! One set of permutations is enough, if you track *k* at each permutation.

Abbreviated R code

```
cint <- function(group1, group2, conf.level = 0.95) {
  n   <- length(group1)
  m   <- length(group2)
  N   <- n + m
  num <- choose(N, n) # nr of possible combs

  # Form a matrix where each column contains indices
  # in new "group1" for that comb or perm
  dcombn1 <- utils::combn(1:N, n)

  # Form the equiv matrix for indices in new "group2"
  dcombn2 <- apply(dcombn1, 2,
                   function(x) setdiff(1:N, x))

  # Form the corresponding matrices of data values
  combined <- c(group1, group2)
  group1_perm <- matrix(combined[dcombn1], nrow = n)
  group2_perm <- matrix(combined[dcombn2], nrow = m)

  # For each comb or perm, compute:
  #   diffmean = difference in sample means
  #   k = nr of values swapped from group1 to group2
  #   wkd = Nguyen (2009) statistic whose quantiles
  #         are used for CI endpoints
  diffmean <- colMeans(group1_perm) -
                colMeans(group2_perm)
  k <- colSums(matrix(dcombn1 %in% ((n+1):N),
                      nrow = n))
  wkd <- (diffmean[1] - diffmean) / (k * (1/n + 1/m))

  # Sort wkd values and find desired quantiles
  w.i <- sort(wkd,
              decreasing = FALSE, na.last = FALSE)
  siglevel <- (1 - conf.level)/2
  index <- ceiling(siglevel*num) - 1
  # Start counting up from 2nd element of w.i
  # (the 1st will always be 'NaN' since k[1] is 0)
  LB <- w.i[2 + index]
  UB <- w.i[(num - index)]

  return(list(CI = c(LB, UB),
         conf.achieved = 1 - (2*(index+1) / num)))
}
```

Comparing computation times

We also wrote a "naive" version of the code, where the calculations of `k` and `wkd` quantiles were replaced with a for-loop calculating p-values from tests of a particular difference at various null values `deltas`.

Code profiling showed that initial `combn()` & `apply(setdiff())` setup steps account for ~80% of the time of Nguyen's method, and calculations on each permutation add up to ~20%. But for the naive method, that "~20%" grows much larger, because these per-permutation steps are repeated for each candidate value of `deltas`.

In this example, we compare 2 large datasets and only take a sample of all possible permutations. Here `nmc` = number of Monte Carlo draws.

```
x <- rnorm(5000, mean = 0, sd = 1)
y <- rnorm(5000, mean = 1, sd = 1)
cint.nguyen(x, y, nmc = 2000)
#> [1] -1.0219087 -0.9445286

# For naive approach with for-loops,
# use a grid of around 20 steps (of size 0.01)
deltas <- ((-110):(-90))/100

bench::mark(cint.nguyen(x, y, nmc = 2000),
            cint.naive(x, y, deltas, nmc = 2000),
            min_iterations = 10)
#>   expression      median_time   mem_alloc
#> 1 cint.nguyen()        4.2sec       1.6GB
#> 2 cint.naive()        12.2sec       7.4GB
```

Takeaways

Nguyen's method takes much less time and less memory than the naive approach (unless you already know the CI endpoints!)

If your dataset isn't trivially small, use our R package (it's on CRAN!) …or read our code (it's on GitHub!) and implement it in other languages.

Tupaj, E. and Wieczorek, J., "`CIPerm`: Computationally-Efficient Confidence Intervals for Mean Shift from Permutation Methods" (2022). R package version 0.2.1.
https://cran.r-project.org/package=CIPerm
https://github.com/ColbyStatSvyRsch/CIPerm/