Original Research Paper

Exact Permutation Algorithm for Paired Observations: A General and Efficient Version

David T. Morse

Department of Counseling and Educational Psychology, Mississippi State University, Mississippi State, MS 39762, USA

Article history
Received: 2014-12-02
Revised: 2014-12-15
Accepted: 2015-01-03

Abstract: For the better part of a century, methods have been illustrated for the enumeration of all possible permutations of cases from which an exact characterization of the likelihood of obtaining results as or more extreme as that observed may be determined without having to rely on parametric assumptions or schemes that may be only asymptotically correct. The challenge is the computational intensity associated with these methods, which is largely overcome with the wide availability of inexpensive, powerful computational resources. The algorithm presented here is given in two versions, one a general form that can be adapted to a wide variety of permutation tests and a specialized one that is efficient for the exact analog to the dependent-*t* test. The application is illustrated using Charles Darwin's *Zea mays* data, which presents a modest task of accounting for $2^{15} = 32,768$ permutations. The resultant algorithm improves on that of Odiase and Ogbonmwan and is presented in syntax that may be run in R, the open source statistical package.

Keywords: Algorithm, Paired Observations, Permutation, Exact Test, P-Value, R

Introduction

Permutation tests have been explicated for nearly a century, so the premise of being able to escape the bonds of parametric tests that are correct only under strong assumptions-and asymptotically at that-is well-known. The usual arguments against the use of permutation (or "exact") tests are: (a) they are more computationally intensive than ordinary parametric or rank-based nonparametric tests; and (b) they are not always available in standard statistical packages. In the first instance, Fisher (1966) noted that his analysis of Charles Darwin's *Zea mays* data set of 15 paired corn plant heights required $2^{15} = 32,768$ permutations. It has been suggested (Ludbrook and Dudley, 1998) that the effort required may have proved a deterrent to Fisher's further use of permutation tests. That's a conclusion consistent with the first argument. However, since that time, computational power has increased dramatically and large-scale, voluntary operations have been established for tackling massive problem sets, such as the Berkeley Open Infrastructure for Network Computing (BOINC; http://boinc.berkeley.edu/), which currently boasts nearly 300,000 volunteers and exceeds 7.3 peta FLOPS of computing power on a daily average. Even on a modest personal computer, Fisher's analysis can be completed in under 0.2 s, using the R package, which is not uniformly optimized for speed. The availability issue is also fading as an argument.

Towards that end, there have been many publications of algorithms to assist in the computation of permutation tests. As a specific example, Odiase and Ogbonmwan (2007) outlined an algorithm suitable for the matched-pairs case of score comparisons. This article presents an improved algorithm and mildly optimized method for the R statistical package that can be used with data sets of any size.

Permutation Tests

As far back as Pitman (1937a; 1937b), the logic of permutation tests has been well-explicated. More recently, excellent explanations from Edgington and Onghena (2007; Manly, 2007) are available. The basic logic is to compare the observed results ("base") to the sample space of all permissible permutations of the scores-what defines permissible permutations depends on the nature of the test. If the number of permutation instances in which the results are as extreme or more extreme than the base results is a sufficiently small fraction of the sample space (e.g., less than one's threshold for classifying a result as non-chance), then we characterize the result as statistically significant. If not, then the result is declared non-significant. It is a simple framework that can easily be extended.

For comparisons involving two sets of scores, there are three principal sets of permissible permutations. For an independent groups comparison (analogous to the independent *t*-test), the population of permissible permutations is “n choose r” (Edgington and Onghena, 2007; Manly, 2007; Pitman, 1937a), in which all possible combinations of n_1 and n_2 cases are created from the combined set of scores, *N*. For a correlation coefficient, the population of permissible permutations is *n*! (Pitman, 1937b), in which each “*x*” score is systematically paired with a different “*y*” score. Finally, for a dependent or matched-pairs design, the population of permissible permutations is 2^n (Fisher, 1966; Ludbrook and Dudley, 1998). The definition of permissible permutations in some classes of designs, however, may require careful thought (Heyvaert and Onghena, 2014) for the class of designs called single subject design.

Materials and Methods

In this presentation, the goal is to present an improved and general algorithm to serve as the basis for permutation tests. It can be used for either independent or matched pairs (dependent) data sets, though the application presented here is strictly for matched pairs sets. In this section, more detailed information is given about: (a) the algorithm presented; and (b) efficiency considerations for implementation of the algorithm in the circumstance of matched pairs data sets.

Permutation Algorithm

The general form of the algorithm presented below can be used to process cases for the paired cases permutation test, or it can also be easily adapted to independent group tests. The algorithm presented by Odiase and Ogbonmwan (2007) relied on hard coding of *for* loops, one for each case in the data set. The current algorithm may be applied to any sample size without additional coding and is therefore much more portable. The logic is based on the FORTRAN algorithm AS 88 by Gentleman (1975). The result of a single call to the algorithm is that, internally, all subsets of size *r* from the *N* cases are generated in lexicographic order (e.g., for “5 choose 3”, the 10 resulting sets of cases would be {1,2,3} {1,2,4} {1,2,5} {1,3,4} {1,3,5} {1,4,5} {2,3,4} {2,3,5} {2,4,5} and {3,4,5}). These subsets are not saved in memory; rather, after each is generated, the permuted data set is processed and a running tally of results is updated. Other solutions to enumerating the “*n* choose *r*” options have also been published (Nijenhuis and Wilf, 1978), though those often must be called once for each permutation cycle. The general form below assumes that: (a) the data vector being processed, X_i, represents the difference scores (= X_{1i}-X_{2i}) for each of the *N* pairs of cases, $i = 1,2,\ldots,N$ and there are no missing values; (b) for a paired cases (dependent) data set, the algorithm is externally called *N* times, for $r = 1, 2, \ldots, N$ and the results within each cycle recorded appropriately against the base (initially observed) result, which represents the case of $r = 0$; and (c) the user accumulates the appropriate computation or outcome for each cycle (represented by “PROCESS DATA” in the algorithm).

Algorithm allnr: n Choose r Function in R Syntax

```
allnr <- function(n, r, x)
# n is number of cases; r is number to be permuted; x is vector of difference scores
# initialize local variables                    (note: “#” signifies a comment in R)
nmr <- (n-r)# n minus r.  R can use either “=” or “<-“ for assignment statements
i<- 1   # index.
j<- c(1:r)                    # create vector to store the chosen case index numbers for a cycle
# main loop cycles ‘n choose r’ times
while (i > 0) {
# reset the loop indices, then process the cases
if (i != r) {                 # “!=” is the “not equal to” operator in R
ip1 <- i +1             # R does not have an increment function like C (e.g., + =)
for (k in ip1:r)  { j[k] <- j[k-1] +1 }
}                             # end of if loop
# PROCESS DATA call or insert the data processing here, on chosen cases j[1],j[2],…,j[r]
i <- r
while ( j[i] >= nmr + i)  { i <- i-1;  if (i == 0) break } # exit routine when i reaches 0
j[i] <- j[i] +1               # otherwise, increment the index and continue
}                             # end of main while loop
}                                # end of function
```

Efficiency Considerations for Paired Observations

There are several ways that the generation of a permutation test for paired data may be made more efficient. First, the number of permutation cycles that actually have to be generated is only $2^n/2$, not 2^n. The reason is, in the permutation analog to the paired or dependent *t*-test, the exchange under investigation for a given case is only considering the second score of a pair to come first for an instance, instead of the first score. So, if a case’s values are exchanged, the resulting difference is the negative of the original difference (e.g., if a pair of scores was 7, 4 then the original difference = 7-4 = 3 and the exchanged values difference would be 4-7 = -3). Thus, if the original set of differences was all positive (the *r* = 0, or first permutation) then the last possible permutation (or *r* = *N* instance) will necessarily have all *N* cases having exchanged scores, yielding all negative differences (if the permutations are generated in lexicographic order). The second permutation (*r* = 1, reversing only scores for case #1) will therefore be the negative of the 2^n-1th permutation (*r* = *N* – 1, last instance), in which all cases *except* #1 have scores exchanged, and so will have negative differences. For example (-1, 2, 3, 4, 5) sums to 13; (1, -2, -3, -4, -5) sums to -13. The rest of the permutations may be thought of as pairs, each member of which also has a negative counterpart. Thus the sum of differences, or the average difference, or the *t*-statistic for that set of mean differences will be the negative of the corresponding value in the first permutation. In this way, the distribution of permutation test results will be symmetric around zero. Only half of the permutations (which means, calling the allnr routine only for *r* = 1,2,...,*N*/2) need to be generated. When *N* is odd, one simply stops processing after the *r* = *N*/2th call is complete. When *N* is even, we need to track within the *r* = *N*/2th call to function allnr until half of the 2^n permutations have been generated. That will save time, even though we have to build in a check on total permutations. In my comparisons using R, this reliably yields a reduction in processing time of about 34%.

A second consideration for efficiency is that, for the permutation analog to the dependent *t*, the sum of the differences is a sufficient statistic for determining whether the results of a permuted data set would equal, exceed, or be less than the originally observed result for a data set. Thus, all the processing step need involve is computation of the sum of the difference scores for a given permutation. This brings up a third efficiency step. In generating that sum, one can sum the exchanged differences only, then subtract twice that sum from the base sum of differences (e.g., permutation sum of differences = base difference-2*sum of exchanged scores). As an example, let the original differences set be (2, 4, 1.5, 1, 3), summing to 11.5. If we exchange the values for cases # 2 and 5, the resulting differences would be (2, -4, 1.5, 1, -3), summing to -2.5. The short-cut described here is to instead take 11.5-2*(4 + 3) = 11.5-14 = -2.5. In other words, we need only sum the exchanged differences, not the full set. These three considerations have been incorporated into the exact.dep.t function and its corresponding version of function allnr, presented in the Appendix. A sample call, using the Darwin data set, also is included.

Appendix: R Code Implementing the allnr and exact.dep.t Functions

```
allnr <- function(n, r, data, base, outcome) {
 # R implementation of Algorithm AS 88 by J.F.
Gentleman (1975).  Applied Statistics, 24, 374-376.

 # n = number of elements in set
 # r = number of elements to be drawn
 # data = vector of scores from which to draw

 # base  = base statistic from data set as recorded
 # outcome[1] = number of instances wherein permuted
result = base result
 # outcome[2] = number of instances wherein permuted
result > base result
 # outcome[3] = number of instances wherein permuted
result < base result
 # outcome[4] = number of permutations generated thus
far
 # outcome[5] = target number of permutations to
generate ( = 1/2 of 2^n)

 # This procedure generates all subsets of r cases out of
the set of n.
 # Current processing is set up for matched pairs
permutation test.

 # Local variables
 # i     = index
 # ip1   = i plus 1
 # j       = vector to store case number/index of chosen
cases (1..r) for a given cycle
 # nmr  = n minus r
 #
 # initialize local variables
 nmr <- (n - r)
 i <- 1
 j <- c(1:r)
 ndiv2 <- n %/% 2                 # integer division

 # loop processes 'n choose r' times
 while (i > 0) {
  # reset loop indices
  if (i != r) {
   ip1 <- i + 1
   for (k in ip1:r) j[k] <- j[k - 1] + 1
  }
   # for dependent t, sum of difference values is
sufficient statistic for magnitude
   x <- base - 2.0 * sum(data[j])                        #
reverse values for selected cases
   if (abs(x-base) < 1.0e-7) { outcome[1] <- outcome[1]
+ 1   # equal to base value
    } else if (x > base) { outcome[2] <- outcome[2] + 1
# more extreme than base
```

```
        } else outcome[3] <- outcome[3] +1                    #
less extreme than base

     # process symmetric case result
     x <- -x
     if (abs(x-base) < 1.0e-7) {outcome[1] <- outcome[1]
+1   # equal to base value
     } else if (x > base) {outcome[2] <- outcome[2] +1
# more extreme than base
       } else outcome[3] <- outcome[3] +1                   #
less extreme than base

   outcome[4] <- outcome[4] + 1                             #
increment total 'cycles' processed
   if (r == ndiv2)
     {if   (outcome[4]   ==   outcome[5])   break}
# early exit if half of all permutations done
   i = r
   while (j[i] >= nmr + i)  { i = i - 1; if (i == 0) break}
# exit main loop when i = 0
   j[i] <- j[i] + 1

  } # main loop end

  return (outcome)
}  # end function


exact.dep.t <- function (data) {

  n <- length(data)               # number of cases, no check
for missing values
  base <- sum(data)               # total of difference scores;
used as referent (e.g., r = 0th case)
  if (base > 0) {
   outcome <- c(1,0,1,1)          # vector of comparisons.
[1] = results equal to data obtained (base value)
   } else if (base < 0) {                # [2] = results more
extreme than base
      outcome <- c(1,1,0,1)             # [3] = results less
extreme than base
       } else outcome <- c(2,0,0,1)     # [4] = number of
permutations processed.  stop at tot_perm / 2
  outcome[5] <- sum(choose(n,0:n)) %/% 2  # [5] = half
of the total possible permutations; this is number needed.

  for  (i  in  1:(n  %/%  2))                    #  lexicographic
distribution of outcomes is symmetric; only need to
process half
   { outcome <- allnr(n, i, data, base, outcome) } # cycle
allnr for instances r = 1, 2,...,n (0 is base case)

  total <- sum(outcome[1:3])                          # number of
comparisons recorded
  prob1 <- (outcome[1] + outcome[2]) / total        # 1 tail
probability
  prob2 <- 2.0 * prob1                                 # 2 tail
probability
  if (prob2 >1.0) prob2 <- 1.0                           # cap
probability at 1

  # output results
  print (paste0('Observed mean difference ', base / n))
  print (paste0('More extreme instances   ', outcome[2]))
  print (paste0('Equal instances          ', outcome[1]))
  print (paste0('Less extreme instances   ', outcome[3]))
  print ('')
  print (paste0('One-tail probability     ', prob1))
  print (paste0('Two-tail probability     ', prob2))

}  # end function

# Sample call in R, using Darwin data (diff = vector,
Cross-fertilized plant height – Self-fertilized height)
diff = c(6.125, -8.375, 1.0, 2.0, 0.75, 2.875, 3.5, 5.125,
1.75, 3.625, 7.0, 3.0, 9.375, 7.5, -6.0)

exact.dep.t(diff)              # diff is a vector of differences
in matched pairs observations
```

Results

The full implementation of the permutation test was evaluated using the Darwin data on corn plant heights (*N* = 15). As reported by Odiase and Ogbonmwan (2007), there were 835 permutations in which the results were more extreme than those observed in the original data, 28 permutation trials in which the results were equal to the original data and 31,905 instances in which the results were less than those originally observed. Thus, as a directional ("one-tailed") probability under the null hypothesis of no difference, the *p*-value is computed as (28+835)/32,768 = 863/32,768 = 0.026337. The non-directional ("two-tailed") result would evaluate as a *p*-value of twice as much, (2 * 863)/32,768 = 0.0526733. These values agree with Odiase and Ongbonmwan, who correctly reported that the dependent *t* statistic, applied to the same data, yields *p*-values of 0.02485 and 0.04970, respectively, for the directional and non-directional tests. Thus, the dependent *t* test tends to be close, but mis-represents the exact test results.

The performance of the R package is reasonably quick for data sets of moderate size. Some examples of processing speed on an ordinary desktop running Windows 7 (64 bit) via an Intel i7 CPU (3.4 Ghz) are: *N* = 20 (2^n = 1,048,576), approximately 5.7 s; *N* = 25 (2^n = 33,554,432), about 185.8 s (approximately 32.5 times longer, which is consistent with the change in size), which is not unreasonable for an interpreted language interface, as R is. For data sets that are substantially larger, it would well be worth the effort to convert the provided code into a compiled form, which would execute more quickly. There are also some vectorized function operations available in R that could profitably be applied, but one design consideration was to keep the code snippets as simple to convert to a different language or framework as possible.

Discussion

Whenever an algorithm is presented, a pertinent question is that of whether alternative methods or routines already exist. The R package has a built-in function, combn, which will generate a set of indices (as column vectors) for an "n choose r" problem. For example, the call, "x<- combn (15,5)" would generate a 5×3003 matrix containing the indices of all combinations in lexicographic order (i.e., x[,1] would include 1,2,3,4,5; x[,2] would be 1,2,3,4,6; and x[,3003] would be [11,12,13,14,15]). That function could be used in lieu of the allnr function presented here. However, there must be sufficient memory to hold the resultant array. As N increases, this could become problematic; the allnr implementation minimizes memory requirements. The Coin package in R is a more fully developed set of easily-called routines, but these are all for independent group tests. The StatExact package (Cytel company), available in stand-alone or as an add-on to IBM SPSS and Systat, is well-known, but is costly and has just recently added the permutation test for matched pairs cases.

This algorithm was developed to allow researchers a simple, no-cost method for implementing the permutation test for matched pairs data sets. By choice, the code was developed to run in the R statistical package, an open-source software project. However, it can easily be converted to many other languages or platforms. The algorithm by Odiase and Ogbonmwan (2007) is functionally satisfactory, but must be revised, by adding or deleting for loops for each possible sample size that one would encounter, which is a slight nuisance for users. No such modification is needed for the present algorithm.

Conclusion

The method described herein yields an exact representation of the permutation method for judging the equivalence of matched pairs data sets. The memory requirements are minimal, the method runs in the most widely used open-source statistical package on the planet and can easily be made a bit more efficient, as shown in the implementation presented in the Appendix. Resampling methods, while easier to program and quick to run, will always yield an approximate result to an exact distribution.

Acknowledgement

The preparation of this manuscript did not involve any external financial support nor any external technical assistance.

Funding Information

There was no financial support involved in any aspect of this study.

Ethics

There are no financial or personal conflicts of interest involved in any aspect of this study.

References

Edgington, E. and P. Onghena, 2007. Randomization Tests. 4th Edn., Chapman and Hall/CRC, Boca Raton, FL., ISBN-10: 1584885890, pp: 376.

Fisher, R.A., 1966. The Design of Experiments. 8th Edn., Hafner, New York.

Gentleman, J.F., 1975. Algorithm AS 88: Generation of all NCR combinations by simulating nested Fortran DO loops. J. Royal Statistical Society, Series C (Applied Statistics), 24: 374-376. DOI: 10.2307/2347110

Heyvaert, M. and P. Onghena, 2014. Analysis of single-case data: Randomisation tests for measures of effect size. Neuropsychological Rehabilitation, 24: 507-527. DOI: 10.1080/09602011.2013.818564

Ludbrook, J. and H. Dudley, 1998. Why permutation tests are superior to t and F tests in biomedical research. Am. Statist., 52: 127-132. DOI: 10.1080/00031305.1998.10480551

Manly, B.F.J., 2007. Randomization, Bootstrap and Monte Carlo Methods in Biology. 3rd Edn., CRC Press, Boca Raton, FL., ISBN-10: 1584885416, pp: 480.

Nijenhuis, A., and H.S. Wilf, 1978. Combinatorial Algorithms: For Computers and Hard Calculators. 2nd Edn., Academic Press, New York, ISBN-10: 0125192606, pp: 302.

Odiase, J.I. and S.M. Ogbonmwan, 2007. Exact permutation algorithm for paired observations: The challenge of R. A. Fisher. J. Math. Stat., 3: 116-121. DOI: 10.3844/jmssp.2007.116.121

Pitman, E.J.G., 1937a. Significance tests which may be applied to samples from any populations. Supplement J. Royal Stat. Society, 4: 119-130. DOI: 10.2307/2984124

Pitman, E.J.G., 1937b. Significance tests which may be applied to samples from any populations. II. The correlation coefficient test. Supplement J. Royal Stat. Society, 4: 225-232. DOI: 10.2307/2983647