Controlling the Familywise Error Rate

I'll begin by saying that the algorithm described in this section and implemented for the optimal-profit-factor test of the prior section is my own semi-rigorous development, largely based on Romano and Wolf (2016) "Efficient Computation of Adjusted p-Values for Resampling-Based Stepdown Multiple Testing", but not rigorously proved by me. I have, however, run many numerical simulations of many experimental conditions, and in every case the simulation results were completely in accord with the expected theoretical results. Thus, I am reasonably confident that this algorithm is mathematically correct. Also, understand that this algorithm could be implemented for any permutation test of selection bias. I do present the algorithm in its most general form, but my implementation here is directly analogous to the indicator selection test of the prior section, making it easy for the reader to do side-by-side comparisons of algorithms and results.

Why consider this alternative algorithm? The traditional selection-bias algorithm, for all its utility, suffers from two annoying weaknesses:

1) The null hypothesis is that *all* competitors are unrelated to the target. This is a significant restriction, at least theoretically. In practice, this restriction seems to create no apparent ill effect when violated, but it makes me uncomfortable. (If needed, see Page 8 for a brief review of hypothesis tests.)

2) The computed probability is strictly correct only for whichever competitor has the greatest relationship with the target. All other selection-bias p-values are upper bounds on the true probabilities. This fact was discussed in the prior section.

The second problem, while troubling, is not devastating, because all competitors for which the computed p-value is less than or equal to the desired alpha level for the test can be considered to be related to the target. That joint statement should satisfy the alpha level because if the least of those that satisfy alpha does so, then certainly all those superior to it do as well. This statement is rather heuristic and could use some rigor, though I am quite confident in its truth in regard to practical applications.

On the other hand, even this result is not ideal because we could easily miss some competitors that are truly related to the target. If their computed p-values overestimate the true probabilities under the null hypothesis to a degree that causes the computed p-value to exceed alpha, despite there being a relationship with the target, then we have missed this competitor. This loss of power is a significant problem, and the algorithm described in this section largely or completely solves it.

This alternative procedure is much better than the traditional one-shot method of the prior section, which pools all candidates into a single batch with the null hypothesis that they are *all* unrelated to the target. This new method tests each null hypothesis *individually*, but with the *familywise error rate* (*FWE*) controlled by our desired alpha. The FWE is the probability of rejecting one or more of the true individual null hypotheses. More loosely speaking, FWE is the probability of making even one mistake in identifying individual null hypotheses to reject.

FWE comes in two forms. An FWE with weak control is one which requires that all null hypotheses be true. This is what we have in the traditional selection-bias test. Far more desirable is an FWE with strong control, which means that it holds regardless of which or how many of the null hypotheses are true. This, of course, corresponds better to real life. In my own professional work I have always acted as if the traditional selection-bias test has strong control even though it does not, and it's never come back to bite me. Much heuristic evidence supports that use. Still, a method with strong control would be more appealing.

An even more desirable property of a selection-bias test is that it have as much power as possible. In the case of multiple null hypotheses there are many possible definitions of power. At one extreme we might want to maximize the probability of rejecting at least one false null hypothesis. At the other extreme we might want to maximize the probability of rejecting all false null hypotheses. Those are both too extreme, one with too little demanded and one with too much demanded. More reasonably we might want to maximize some measure of average rejection probability. This intermediate goal, perhaps maximizing the average probability of rejecting false null hypotheses, is doubtless the best, and is a property that I believe is possessed by my new algorithm.

We must understand this property of maximum power, because it is very important in practice. Recall that the traditional selection-bias algorithm provides only upper bounds for the p-values for all competitors except the best. This makes it possible that it will fail to reject null hypotheses (decide that there is a relationship) for competitors that truly have a relationship with the target. That's the beauty of this new algorithm: it can often flag competitors that would have been missed by the traditional algorithm due to overestimation of p-values, while still maintaining a user-specified familywise error rate.

In summary, we want to be able to test each *individual* competitor's null hypothesis while having strong control of the FWE and maximizing average power. The traditional selection-bias algorithm has only weak control of the FWE and it has excellent power only for whichever competitor is the best (maximum relationship with the target).

I believe that my new algorithm provides these superior properties. The algorithm will be shown soon. But first I want to discuss the general philosophy of the procedure so as to make the algorithm more clear.

This is a stepwise procedure, with hypotheses being rejected one at a time, in order starting with the best competitor (largest target relationship) and working downward until no more null hypotheses can be rejected at the user-specified FWE, alpha level. As each null hypothesis is tested, we approximate the null-hypothesis distribution of that relationship statistic by permuting the target as in the traditional algorithm, but finding the maximum of *only the competitors that have not yet had their null hypotheses rejected*. This is the critical difference between this improved algorithm and the traditional selection-bias algorithm. If, for each step, we were to approximate the null hypothesis distribution by finding the maximum relationship statistic of all permuted competitors we would have an algorithm that is essentially identical to the traditional algorithm, just re-ordered as stepwise instead of all at once. However, in the new algorithm, the number of competitors that go into the computation of the maximum relationship statistic is reduced by one for each step, thus shrinking the null distribution.

In summary, this algorithm is almost identical to the traditional algorithm, except that instead of testing all null hypotheses at once, we test them one at a time, and as we do successive tests we keep shrinking the number of competing distributions that go into approximating the null distribution.

I'll now walk through the algorithm listed on the next page, and continue the walkthrough after the listing. Note that this algorithm is relatively straightforward and easy to understand, but too slow for practical work. An equivalent but much faster version will be shown in the next section.

The user has specified that there are n competing populations (indicators here), and the test will employ m permutations (thousands) to estimate the null hypothesis distributions. A desired alpha level (maximum FWE that the user can accept) for the test has also been specified.

The first step is to compute the relationship statistic for each competitor and store them in the original array. We'll also need to sort them so that the stepwise procedure can proceed from best (largest) to smallest. But we must not disturb the order of original, so we copy that array to a work array and sort it ascending. We also initialize sort_indices to an identity array, and when we do the sorting we simultaneously move the elements of this array. Thus, after sorting, sort_indices[0] will be the index of the competitor having the smallest relationship, sort_indices[1] the next smallest, and so forth. Later, the stepwise procedure will work backwards through this array to test the populations in order from best to worst.

As competitors have their null hypotheses rejected, we keep track of which have been rejected via the passed array, where a TRUE value means that its null hypothesis has been rejected; it passed the test for having a relationship with the target. Prepare for the stepwise accumulation loop by initializing passed to FALSE for all competitors.

```
–> Compute the relationship for each competitor and sort

For i from 0 through n-1
   sort_indices[i] = i ;
   original[i] = relationship of competitor i with Y
   work[i] = original[i] ;

Sort work ascending, moving sort_indices simultaneously

–> Initialize that no competitors have yet passed (null rejected)

For i from 0 through n-1
   passed[i] = FALSE ;

–> The stepwise accumulation loop begins here

For step from n-1 through 0, working backwards (best to worst)
   this_i = sort_indices[step]        Index of best remaining competitor
   count[this_i] = 1 ;                Counts right-tail probability

   –> Permutation loop estimates null distribution of population

   For irep from 1 through m          Do all random replications
      Shuffle Y

      max_f = number smaller than smallest possible relationship
      For i from 0 through n-1
         if (NOT passed[i])           Do only those without null rejected
            this_f = relationship of competitor i with Y
            if (this_f > max_f)       Keep track of maximum
               max_f = this_f ;

      If (max_f >= original[this_i])
         ++count[this_i] ;            Count right-tail probability

   –> See if this new competitor passed (NULL rejected).

   If count[this_i] / (m+1) <= alpha
      passed[this_i] = TRUE
   Else
      Break out of step loop; we are done
```

The stepwise accumulation loop now begins. It moves backwards through the competing indicators because they have been sorted ascending and we want to begin with the best. Recall that sort_indices contains the indices of the sorted competitors, so we place in this_i the index of the competitor that is about to be tested for inclusion in the set of rejected null hypotheses. As we did in the traditional algorithm, we initialize the counter of right-tail probability to 1 before performing the loop that approximates the null hypothesis distribution of the relationship statistic.

The permutation loop is now executed. Shuffle the target and initialize max_f to any number that is smaller than the smallest possible relationship statistic. This variable will keep track of the maximum relationship statistic in this replication. We now come to the part of the algorithm that distinguishes it from the traditional selection-bias algorithm. In that prior algorithm we found the maximum relationship statistic across all competing populations. But in this algorithm we exclude those competitors whose null hypotheses have already been rejected. So inside the loop that passes through all populations we process only those for which passed is FALSE. After we find the maximum we compare it to the original value of the competitor being tested and increment the right-tail probability counter if this null hypothesis value equals or exceeds the original value.

After all permutation replications are complete we have an estimate of the right-tail probability of the relationship statistic for competitor this_i. All we need to do at this point is compare this probability to the user-specified alpha. If it is is less than or equal to alpha we add it to the accumulated collection of passing competitors (those that we conclude have a relationship with the target). But if it did not pass we are done, so break out of the accumulation loop.

Here is a rough overview of my intuition for why this algorithm has an FWE of alpha with strong control, and also maximizes the average probability of rejecting false null hypotheses. My hope is that someone will make this more rigorous. I could have done this myself 40 years ago when I had my freshly minted statistics PhD, and I might still be able to do it, but at this point in my life I have too many other interests to devote significant time to this task.

Consider the best competitor, the one having the greatest relationship statistic and hence the one that we test first. Suppose its null hypothesis is true. By implication its relationship statistic will have the same distribution as that for all permutations (under the usual assumption that the target values are independent and identically distributed). Thus we will erroneously reject this null hypothesis with probability alpha. If we do so, it does not matter what errors we may subsequently make for other populations, because the definition of FWE is the probability that we will make one *or more* rejection errors. On the other hand, if we do not reject this null hypothesis, we are finished testing populations, so there is no more opportunity to make an error.

Now suppose the first null hypothesis is false. I claim that the permutation test as described is asymptotically the most powerful possible test for detecting this false null hypothesis. This should be a no-brainer, because we are testing the observed statistic against an asymptotically exact estimate of its actual distribution. If we declare this null hypothesis true (incorrectly, but not affecting FWE), we are finished testing populations for inclusion, so there is no more opportunity to make an error. If we declare it to be false we are correct and we advance to the next candidate.

When we advance to the next candidate, we are in exactly the same situation we were in with the first candidate, but now that first candidate is entirely removed from further computation. Its relationship statistic is no longer referenced, and that population no longer takes part in estimating the null hypothesis distribution of this next candidate. So if this second candidate's null hypothesis is true, we have probability alpha of incorrectly rejecting it. All other logic is exactly as it was for the first candidate.

This repeats until eventually we do not reject a null hypothesis, at which point we stop. We have alpha probability of having erroneously rejected a true null hypothesis at least once along the way, and thus we have a FWE of alpha, as desired. This fact holds regardless of how many null hypotheses are true, so thus our FWE has strong control, as desired. Finally, each time we encounter a false null hypothesis we employ the asymptotically most powerful test possible to test that hypothesis, and so we have maximum average probability of correctly rejecting false null hypotheses (asymptotically).

These assertions are distressingly heuristic, with little or nothing in the way of rigor to back them up. For my power arguments, I conveniently downplayed the fact that the null hypothesis distributions of the test statistics are only *asymptotically* correct, relying on the fact that if a great number of replications are used, the approximation is excellent. However, the intuition seems sound to me. Moreover, I have run massive quantities of Monte-Carlo simulations, using multiple alpha levels, multiple numbers of cases, multiple numbers of candidate populations, and various proportions of the candidates (from 0 to most) having false null hypotheses. In every case, the FWE came in at almost exactly the specified alpha level, well within normal variation tolerances. And this test has amazing power to detect even minuscule degrees of relationship between candidates and Y. So I am confident enough in its practical utility to use it in my own work and recommend it to others.

Demonstrating the Stepwise Algorithm

On Page 88 we saw a demonstration of indicator selection by optimal profit factor, using the traditional Monte-Carlo permutation test. Please keep handy the table of final results from that test. We now run exactly the same test, except using the new stepwise algorithm just described and with alpha=0.1. Here are those results:

```
       Variable  profit factor  unbiased pval
        CMMA_10      1.597        0.005
          RSI_5      1.544        0.019
         RSI_10      1.475        0.078
         RTVY_6      1.462        0.098

Best remaining p-value=0.1960, so quitting
```

For the best two indicators the p-values are almost the same in both tests. (Theoretically, the first should be the same, because the two versions of the test are identical for the best performer. But these tests use random numbers, so small variation is likely. This is why it's important to use a large number of MCPT replications.) By the third, the p-value has dropped from 0.099 for the traditional test (which is an upper bound for the true value) to the true value of 0.078. For the fourth it drops from 0.123 to 0.098. This makes it just under my specified alpha of 0.1 so we pick up one more indicator at this alpha level, a clear demonstration of the increased power of the stepwise version. The fifth p-value, 0.1960, blows far past my alpha, so inclusion ceases.

It is tempting to use a larger alpha in order to see more computed p-values, but there is a serious potential problem with this if you are not careful. ***You must stop considering candidates as soon as the p-value passes your preset alpha.*** This is because raw p-values may actually decrease later. The Romano-Wolf reference cited at the beginning of this section solves this problem by forcing each successive p-value to be at least equal to the prior p-value, and they explain why this is necessary if p-values beyond that for the best competitor are to be used as actual p-values. I do the same in the code presented later. The explanation is far too complex for this text, so please see that paper for details.

Note that these p-values are computed using random numbers, so if you do not perform a large number of replications (thousands) you may occasionally find that the stepwise test produces a p-value slightly greater than that of the traditional test, which in theory should never happen. This is just random variation, easily fixed by using more replications.

Accelerating the Stepwise Algorithm

The algorithm shown on Page 94 is the best way to present the new stepwise algorithm, because it is a straightforward implementation of the mathematical statement. However, it is unnecessarily slow. This is because the block of permutations does not need to be repeated each time a new competitor is tested for inclusion (null hypothesis rejection). We need to do the set of `m` permutations only once, estimating all null hypothesis distributions simultaneously. Then we can do the stepwise inclusion after the permutations are complete.

To collect all distributions at once, we work from worst to best, updating the 'maximum so far' as each increasingly good competitor is added to the mix. Here is the fast but mathematically identical algorithm:

```
For i from 0 through n-1
  sort_indices[i] = i ;
  original[i] = relationship of competitor i with Y
  work[i] = original[i] ;

Sort work ascending, moving sort_indices simultaneously

–> Step 1 of 2: do the random replications and count right tail

For i from 0 through n-1
  count[i] = 1 ;                 Counts right-tail probability

For irep from 1 through m
  Shuffle Y

  max_f = number smaller than smallest possible relationship
  For i from 0 through n-1       Work from worst to best
    this_i = sort_indices[i]
    this_f = relationship between this_i and Y
    if (this_f > max_f)          Keep track of maximum
      max_f = this_f

    If (max_f >= original[this_i])
      ++count[this_i] ;          Count right-tail probability
    } // For irep

–> Step 2 of 2: Do the stepwise inclusion

  For i from n-1 through 0       Work from best to worst
    this_i = sort_indices[i]     Index of best remaining competitor
    If count[this_i] / (m+1) <= alpha
      Accept this competitor
    Else
      Break out of step loop; we are done
```

The slowest part of the selection-bias test is shuffling, whose time is proportional to the typically large number of cases. By computing all null hypothesis distributions in one replication loop we avoid a new set of shuffles every time we test a new candidate.

Code For the Fast Stepwise Algorithm

We now illustrate the algorithm shown on the prior page. This code is extracted from IND_FAM.CPP, a program that is identical to the IND.CPP program discussed on Page 81 except for two differences:

1) We use the new algorithm, so in addition to the prior parameters the user must also specify alpha, the maximum familywise error rate.

2) To simplify the code to be more understandable to the reader, only the maximum of the long and short profit factors is used as the performance criterion. The IND program also tested the long and short performance separately.

The program is called with five parameters as shown below. Please refer to the IND documentation on Page 81 for details.

```
IND_FAM  Floor  Alpha  Nreps  DataFile  ControlFile
```

We'll skip the code that's identical in both programs, or mundane. Assume that we have already computed the vector of original_crits. We have to sort them ascending and also keep track of the indices that point to the indicator at each performance rank. Then initialize the counters that keep track of the right-tail probabilities.

```
for (i=0 ; i<n_indicators ; i++) {
   sort_indices[i] = i ;
   dwork1[i] = original_crits[i] ;
   }
qsortdsi ( 0 , n_indicators-1 , dwork1 , sort_indices ) ; // Sort ascending

for (ivar=0 ; ivar<n_indicators ; ivar++)
   count[ivar] = 1 ;
```

We're now ready for the first of the two steps: the permutations that compute all null hypothesis distributions simultaneously. Here is this code, and a discussion follows.

```
for (irep=0 ; irep<nreps-1 ; irep++) {

   // Shuffle target
   i = ncases ;            // Number remaining to be shuffled
   while (i > 1) {         // While at least 2 left to shuffle
      j = (int) (unifrand() * i) ;
      if (j >= I)          // Cheap insurance against disaster if unifrand() returns 1.0
         j = i - 1 ;
      dummy = target_work[--i] ;
      target_work[i] = target_work[j] ;
      target_work[j] = dummy ;
      }

   // This loop processes competitors in order from poorest to best
   best_crit = -1.e60 ;
   for (ivar=0 ; ivar<n_indicators ; ivar++) {
      k = ind_index[sort_indices[ivar]] ;    // Column of ivar'th poorest in database
      for (i=0 ; i<ncases ; i++)
         ind_work[i] = data[i*nvars+k] ;      // Fetch it from database
      rho = spearman ( ncases , ind_work , target_work , dwork1 , dwork2 ) ;
      if (rho < 0.0) {  // Make sure that indicator and target are positively correlated
         for (i=0 ; i<ncases ; I++)            // Flip sign of this indicator
            ind_work[i] = -ind_work[i] ;
         }
      opt_thresh ( ncases , (int) (floor * ncases + 0.5) , 0 , ind_work , target_work ,
                &dummy , &dummy , &long_pf , &dummy , &short_pf , dwork1 , dwork2 ) ;
      best_pf = (long_pf > short_pf) ? long_pf : short_pf ;  // Performance criterion

      if (best_pf > best_crit)
         best_crit = best_pf ;

      k = sort_indices[ivar] ;     // Index of ivar'th poorest indicator
      if (best_crit >= original_crits[k])
         ++count[k] ;
      } // For ivar, poorest to best
   } // For irep
```

It would be good to examine this code in conjunction with the algorithm shown on Page 99. The first step is to shuffle the target variable, bar-by-bar return in this application. The variable called *max_f* in the general algorithm is called best_crit in this specific application. We initialize it to a number vastly smaller than the smallest possible profit factor.

Then we work through all competing indicators, starting with the worst (smallest profit factor criterion) and progressing toward the best. Fetch into ind_work this indicator. Compute its nonparametric correlation with the target and flip its sign if the correlation is negative. Then find the optimal long and short profit factors and define our performance criterion as whichever is larger. As each new indicator enters the mix, update the maximum, best_crit. Finally, compare the current maximum to the original criterion for the current indicator. Update the right-tail count if indicated.

After all replications are complete we can perform the second of the two steps, adding indicators (rejecting null hypotheses) as long as we are able to avoid exceeding the user-specified alpha. In this case we work from the best to the worst, going backwards through sort_indices. For each, the p-value is the right-tail count divided by the total number of replications, including the original, with monotonicity enforced. As long as alpha is not exceeded we keep going, but as soon as it is exceeded we have to stop.

```
prior = 0.0 ;
fprintf ( fp_log , "\n\nBest of long/short profit factors and p-values..." ) ;
for (i=n_indicators-1 ; i>=0 ; i--) {
   k = sort_indices[i] ;
   pval = (double) count[k] / (double) nreps ;
   if (pval < prior)       // Enforce monotonicity as explained in Romano-Wolf
      pval = prior ;       // cited at the beginning of this section
   prior = pval ;
   if (pval <= alpha) {
      fprintf ( fp_log , "\n%15s %10.3lf %12.3lf",
              names[ind_index[k]], original_crits[k], pval ) ;
      }
   else {
      fprintf ( fp_log, "\n\nBest remaining p-value=%.4lf, so quitting", pval ) ;
      break ;
      }
   } // For all competitors, working from best to worst
```