Paper 268-2010

BootstrapMania!: Re-Sampling the SAS® Way

David L. Cassell, Design Pathways, Corvallis, OR

ABSTRACT

The most common way that people do simulations and re-sampling plans in SAS® is, in fact, the slow and awkward way. People tend to think in terms of a huge macro loop wrapped around a piece of SAS code, with additional chunks of code to get the outputs of interest and then to weld together the pieces from each iteration. But SAS is designed to work with by-processing, so there is a faster, cleaner, simpler way. The basics of this approach have appeared in previous papers, but this paper focuses on new innovations in bootstrapping, and code for bootstrap methodologies which have not been shown before.

INTRODUCTION

The most common way that people bootstrap parameter estimates in SAS is the way that causes many people to abandon SAS for this sort of task. So first, what is a bootstrap, and how do we perform it?

For our purposes, resampling covers a range of ideas. All of them involve taking our sample and using it as our basis for drawing new samples. Hence the name 'resampling'. We sample from our sample, in one way or another.

There are several ways in which we can 'resample'. We can draw 'randomly' from our sample, or we can draw designed subsets from our sample. (See our discussions on bootstrapping and jackknifing and cross-validation.) Also, we can pull the data but exchange the labels on the data points. (See our discussion on randomization tests.)

In bootstrapping, we want to approximate the entire sampling distribution of some estimator. We can do that by resampling (simple random sampling with replacement) from our original sample. Typically in bootstrapping, we want to estimate the bias of the estimator, or its standard error, or a confidence interval for the parameter. Bootstrapping is a non-parametric method, since we are not making specific assumptions about the distribution(s) from which the data arise. But that does not mean that it has no underlying assumptions at all!

This non-parametric methodology means that bootstrapping can be a useful alternative to inference based on 'parametric' assumptions, for example, inference based on the normality of the distribution of the parameter estimates. If the traditional underlying assumptions are not met, or are suspect, then a non-parametric approach can be more reliable. This approach also helps when parametric inference is simply not workable: the distribution of the tri-mean, for example, is so complex that a bootstrap estimate of its error is a standard approach.

We will start out with the naïve bootstrap, which works well when looking at the behavior of a single variable. We need to take our original sample, and draw many samples from it. Typically, we will be drawing something on the order of a thousand samples, so this gets computer-intensive rather quickly.

The theory behind the bootstrap leads us to draw many independent samples, each of which is a simple random sample WITH replacement, of the same size as the original sample. A simple random sample with replacement means that we draw one observation at random, record it, then place it back in our sample so that it can be drawn again if chance dictates. This means that it is theoretically possible (although really ridiculously unlikely) that our simple random sample with replacement of size n could consist of just one record, drawn n times in a row. (Don't worry, this isn't going to happen to you. The probability that this happens to you when you work with your sample of size 50 is on the order of 10**-85 .) But it does mean that we need to make sure that we get all the times when we pick record i, not just the first time. This will matter in a bit. Simple Random Sampling With Replacement is often abbreviated as SRS-WR, WR (the 'With Replacement' part), or URS (Unrestricted Random Sampling). The 'Unrestricted' part comes from the concept that there is no restriction on the legal values you can get on draw number 2, or 3, or …, while in Simple Random Sampling Without Replacement, you cannot select any of the values that have already been chosen.

We then draw a large number of these samples. As a general guideline, 1000 samples are usually enough for a good look. However, if the results really matter, you should consider drawing as many of these samples as you can afford, given your time and computing resources. So a fast, efficient approach is going to be important.

Once we get our 1000 (or however many) independent samples, we compute our estimate or statistic for each sample, and then look at the collection of values as our estimate of the overall sampling distribution of the real estimator. Under common assumptions, this works pretty well. It doesn't work all the time, so we have to pay attention to our data and our data sources and our meta-data before we take this sort of approach. We often describe the underlying requirement as 'exchangeability', meaning (roughly) that it wouldn't matter which data point came from which record in the data. That means that time series data, repeated measures data, survey sample data, and data with analytic weights may be inappropriate for the naïve bootstrap we will be talking about first. There are more complex bootstrapping methods to deal with many of these issues, but they require some thought. Despite the convenience of the naïve bootstrap, it should not be used naively.

Once we get all the 1000 (or however many) values from our samples, we can compute a mean, or a standard deviation, or a confidence interval. One way to estimate 100(1-alpha)% confidence intervals from bootstrap samples is to take the alpha/2 and 1 – alpha/2 quantiles of the estimated values. These are called bootstrap percentile intervals. For example, a bootstrap 95% percentile interval would be the interval from the 2.5th percentile to the 97.5th percentile. We can get both these values from PROC UNIVARIATE, as we will see below.

THE SIMPLE BOOTSTRAP - YOU SHOULD KNOW THIS

As we have discussed before, the usual 'bad' SAS code for bootstrapping is one humongous macro loop, with complex code lodged inside to handle all the pieces: a chunk to do the sampling from the data set, some statistical process that uses the just-built data, some way of welding the new information to the already-collected information, and then finally a procedure to take the information and calculate the desired bootstrap estimates. One example of this type of code might be this macro to generate a non-parametric bootstrap confidence interval for the kurtosis of a variable X.

```
%macro bootie ( input=, reps= );

%DANGER DANGER WILL ROBINSON! ;
%WARNING: do not try this at home!! ;

  %do i = 1 %to &REPS ;
    data gen;
      do i=1 to nobs;
        rec = ceil(nobs * ranuni(0));
        set &INPUT nobs=nobs point=rec;
        output;
        end;
      stop;

    proc univariate data=gen;
      var x;
      output out=outx kurtosis=curt;

    %if &I = 1 %then %do;
      data outall;
        set outx;
      %end;
    %else %do;
      proc append base=outall data=outx;
      %end;
    %end;  /* i=1 to &REPS loop */
  proc univariate data=outall;
    var curt;
    output out=final pctlpts=2.5, 97.5 pctlpre=ci;
%mend;

%bootie(input=YourData, reps=1000)
```

There are lots of small problems with code like this. But the important issue is that the macro achieves its goal in the slowest, most painful way possible. For 1000 reps, the macro runs in 3001 steps. Just think about what that will do to your log and output files (or your log and output windows, if you run this from the Display Manager, which I really do NOT recommend). In fact, the most common way that I find that people are writing code like this is when they complain about this last problem. *"My program runs all weekend and is filling my log (and/or output) window. I have to hang around for hours and deal with the little pop-up windows telling me that my window is full. How do I turn off that pop-up?"* The real problem is not that pop-up windows occurred; the real problem is the underlying code.

As we have demonstrated in previous papers, building all the iterations in a single data set and then using BY-processing will reduce the number of steps. This will drastically reduce the required time and CPU usage. We will create a single data set with all the replicates in it, and then run the procedure on this new data set to get all the results in one output data set. This will also have the advantage that we will not need a separate procedure to do the appending operations, as they will be done automatically through the by-processing feature.

Here is code as we have demonstrated before:

```
sasfile YourData load;                              /*  1 */
proc surveyselect data=YourData out=outboot         /*  2 */
      seed=30459584                                 /*  3 */
      method=urs                                    /*  4 */
      samprate=1                                    /*  5 */
      outhits                                       /*  6 */
      rep=1000;                                     /*  7 */
  run;
sasfile YourData close;                             /*  8 */

ods listing close;                                  /*  9 */
proc univariate data=outboot;
  var x;
  by Replicate;                                     /* 10 */
  output out=outall kurtosis=curt;
  run;
ods listing;                                        /* 11 */

proc univariate data=outall;
  var curt;
  output out=final pctlpts=2.5, 97.5 pctlpre=ci;
  run;
```

We in essence have a 'wrapper' around our procedure of interest, in much the same way that PROC MI and PROC MIANALYZE create a wrapper around a procedure in order to provide an analysis of the effect of multiple imputation on the analysis of a data set.

Let's look at this more carefully. PROC SURVEYSELECT appeared in SAS 8.2, with the survey analysis procedures PROC SURVEYMEANS and PROC SURVEYREG. It allows one to generate random samples of many kinds from an input data set. The REP= feature was introduced to allow samplers to create what are known as replicate samples in sampling theory. But the same feature can be used to create bootstrap samples and simulation data sets.

In line [2], we invoke PROC SURVEYSELECT and tell it the input and output data set names, through the traditional SAS options DATA= and OUT= . At this point, the code looks little different from a PROC SORT statement, or any other common SAS procedure.

In line [3], we specify a random seed. If we specify a seed of 0, PROC SURVEYSELECT will generate a random seed on its own, and use that. Unlike the RANUNI() function in our macro above, PROC SURVEYSELECT will politely tell us in the output what seed it actually used. This is subtle but important. The information about the starting seed gives us enough information that we can reproduce our results, even if we choose to use a seed of zero, or if we leave the SEED= option out (the default is to use SEED=0). The alternative to a seed of 0 is to specify an integer seed between 1 and 2**31 - 1. Using a random seed of our own choosing gives us code which documents the seed for the pseudo-random number generator under the hood. Remember, you will need to know which seed was used. And you will need that information just as soon as you can no longer find that information in your files. Without the seed, you cannot reproduce your results for your boss (or your client, or your major professor, or your review board, or your journal editors, or…).

In line [4], we use METHOD=URS. The METHOD= option lets us specify the type of random sampling. For a bootstrap, we need a simple random sample with replacement, and we need the sample to be of the same size as the original data set. Remember that simple random sampling with replacement is also called Unrestricted Random Sampling, which is why it is abbreviated as URS in the METHOD= option.

In order to get a sample of the same size as our original data set, we could find the data set size and put that in a macro variable for use in the procedure call. But all we really need is a 100% sample. So, in line [5], we can use the SAMPRATE= option to get that without having to figure out the data set size first. SAMPRATE= accepts either whole numbers (as percents) or proportions up to one. Either SAMPRATE=1 or SAMPRATE=100 will give us that 100% sample.

In line [6], we use the keyword OUTHITS. This is for use when we ask for samples which could return a record more than once – like URS samples. OUTHITS makes sure that the procedure generates an output record every time it hits a given record, rather than only the first time. This gives us the bootstrap sample that we need in the next step.

In line [7], we specify the number of bootstrap samples that we want to generate. This automatically generates a variable called REPLICATE, which keeps track of the replicate number for the samples. This variable is then ideal for use as a by-variable. It increases by 1 each time we start a new sample. This makes it ideal for by-processing.

So, in the line labeled [10], we use the variable REPLICATE as the by-variable in our procedure. This is an important point. All we have to do in order to use this bootstrapping method for any procedure is to take the already-written code that we have been using, and insert that BY statement to get the necessary bootstrap realizations.

Now, we still have some of the same problems we saw before. PROC SURVEYSELECT is very fast, but it runs through the original data set once for every replicate. It could be faster, if there were a way to load the whole data set into RAM first, and then only access the RAM. Prior to SAS 9, that was only possible using SAS/SHARE. (Although SAS will internally buffer as much of the disk I/O as it can, to speed up processes like the POINT= non-sequential read.)

As of SAS 9, there is the SASFILE statement. The SASFILE statement lets us load the data set into buffers in RAM, and hold it there until we again use another SASFILE statement to release those buffers. Note that if your data set is too large to fit in your RAM, this is not a good thing. Of course, if your data set is too large to fit in your available RAM, 1000 replicates of it using this bootstrapping approach may be too large to fit on your hard drive. If so, then you should consider alternatives. One option is using the %JACKBOOT macro available from the SAS webpages, while another will be discussed later in this paper. For that matter, you should consider whether you should be using bootstrapping at all, or whether alternative approaches are more appropriate for your analysis problem. Or you should consider whether you need to subset your data before beginning your analyses. Remember: a bootstrap is only a linearization of a surface; it is not appropriate in every circumstance, and it does not fix every problem even when it is appropriate.

The SASFILE statement has three options: OPEN, LOAD, and CLOSE. LOAD opens the file, allocates the buffers in RAM which will hold the file, and then reads the data into memory. OPEN does all but reading the data in, which is left until the DATA or proc step reads the data file. For our purposes, the difference between the two is minimal. CLOSE frees up the RAM buffers when we're done with the file. So we can use the SASFILE statement in lines [1] and [8] above, to load the data into RAM and then free up the RAM afterward. Clearly, if the data set is too large to fit into RAM, this is not an option. But if your data set is really that lare, you should probably reconsider the use of the naïve bootstrap.

Furthermore, a lot of output was generated by that PROC UNIVARIATE step with the by-variable REPLICATE. Did we really want to see all 1000 individual computations of the kurtosis, when the resulting data set OUTALL holds all the information that we want? Well, no. The information is summarized much more conveniently in the OUTALL data set, so if we want to see the details we would be better off suppressing the output from the procedure and printing the OUTALL data set as a nicely-formatted table.

So we want to simplify the contents of our list file (or our output window if we are insistent on running this in Display Manager). The way to do this is with ODS. If, in the middle of our code, we bracket the procedure with ODS LISTING statements, then we can turn off the output to the Output window (or the .lst file if we are in batch mode). The ODS LISTING CLOSE statement in line [9] turns off the ODS destination that has our list output, while the ODS LISTING statement in line [11] turns it back on. This approach is much better than the old NOPRINT option, for one simple reason. While NOPRINT is shorter, and clearly only applies to a single procedure step, it also turns off every single one of the ODS destinations. Using the NOPRINT option will make it impossible to get data sets for our bootstrap estimates when we need to use the ODS OUTPUT statement to create the output data set.

THE SIMPLE BOOTSTRAP - A NEW METHOD

There is an alternative approach that is roughly as fast, but a little more complex to use. This uses the POINT= option and nested DO-loops in a data step. This method assumes that the NOBS= option will surface the number of observations in the YOURDATA input set. Remember that the NOBS value is not always available, for example, if your input data are in a view instead of a data set, or exist in an external data table instead of a SAS file. If the number of observations is not surfaced via the NOBS= option, then you have to compute it first, and insert it yourself. On the other hand, this method is available even if you do not have SAS/STAT licensed, since it does not use PROC SURVEYSELECT. Now let's take a look at some sample code:

```
sasfile YourData load;
data outboot(drop=__i);
  do Replicate = 1 to 1000;                          /* 1 */
    do __i = 1 to numrecs;                           /* 2 */
      p = int(1 + numrecs*(ranuni(39573293)));       /* 3 */
      set YourData point=p nobs=numrecs;             /* 4 */
      output;                                        /* 5 */
      end;
    end;
  run;
sasfile YourData close;

ods listing close;
proc univariate data=outboot;
  var x;
  by Replicate;
  output out=outall kurtosis=curt;
  run;
ods listing;

proc univariate data=outall;
  var curt;
  output out=final pctlpts=2.5, 97.5 pctlpre=ci;
  run;
```

Note that we have replaced the PROC SURVEYSELECT inside the SASFILE statements with a DATA step. This is the only change we need to make. (Remember: if the input data set is too large to fit in RAM, we don't want to use the SAFILE statements anyway. Sometimes this can only be determined by trying it.) We are still building a tall-

and-thin data set with a BY-variable named REPLICATE for feeding into our middle step. As can be seen in statement [1], we once again create 1000 separate BY-groups, with the BY-variable REPLICATE counting from 1 to 1000. As before, there is no need to set this count at 1000. It is simple enough to make this a lot smaller for test cases, or much larger when extensive replication is required.

Within each replicate, we need to draw an unrestricted random sample (simple random sampling with replacement) of size *N*, where *N* is the size of our input data set. That gets constructed with the inner DO-loop in statement [2]. The variable __I is used so that there is a much smaller chance of over-writing a variable from the input data set. The value NUMRECS is created by the DATA step and available when the do-loop runs, because the SET statement [4] is processed before the other statements are executed.

In order to obtain an unrestricted random sample for each replicate, we have to choose a random record of the input data set *N* ties within each replicate. The simplest way to do that is to use the POINT= option to repetitively pull a random integer between 1 and *N*. In line [3], NUMRECS times a random number between 0 and 1 (not including the endpoints) gives us a random number between 0 and NUMRECS (not including 0 or the value of NUMRECS). Adding 1 and truncating gives us a random integer between 1 and NUMRECS, *including* the endpoints 1 and NUMRECS. So our variable *P* has the value we want to feed back to the SET statement [4] so that we can pick out the *P*th record of the input file. Then we only need to output that value in statement [5] and repeat the process. Also note that we used a hard-coded random number as the seed for RANUNI(), rather than using a seed of zero: a seed of zero will give a random start which is not reproducible, while any integer between 1 and (2**31)-1 will yield a reproducible random start. I usually use a random number generator written in Perl to generate my seeds.

And, just in case you are interested, you can build a similar bootstrap data set using PROC SQL, and it will run at roughly the same speed.

WORKING WITH A REALLY LARGE DATA SET

Right about now, you should be asking yourself why we have taken the time to demonstrate yet another way of building the same old bootstrap data set to feed into our middle step. This newer method is going to give us ways of bootstrapping our data with large data sets, as well as giving us a way of computing other types of bootstraps. (Yes, there is more than one kind of bootstrap.)

More people are worknig with larger and larger data sets. Plenty of people want to bootstrap statistics pulled from such data. I often tell people that if they have such a large data set, they should reconsider bootstrapping at all. But people seldom listen to me. Hence, our next example. What do we do if the input data set is so large that we *cannot* fit the OUTBOOT data set into our disk space?

The first person who mentioned this to me was Bob Virgile, the well-known SAS author. He and I agreed that there was no way to get a data view out of PROC SURVEYSELECT for this purpose, but that it would be simple given the previous code. The only changes required are removing the SASFILE statements and turning the OUTBOOT data set into a data view:

```
data outboot(drop=__i) / view = outboot;                    /* 1 */
  do Replicate = 1 to 1000;
    do __i = 1 to numrecs;
      p = int(1 + numrecs*(ranuni(39573293)));
      set YourData point=p nobs=numrecs;
      output;
      end;
    end;
  run;

ods listing close;
proc univariate data=outboot;
  var x;
  by Replicate;
  output out=outall kurtosis=curt;
```

```
    run;
  ods listing;

  proc univariate data=outall;
    var curt;
    output out=final pctlpts=2.5, 97.5 pctlpre=ci;
    run;
```

If the data set is so large that the OUTBOOT file won't fit on your hard drives, then expect that the input data set is not going to fit in available RAM, and the SAFILE statements won't help you. Other than removing the two SASFILE statements, the only change needed is in statement [1], where we use the VIEW= option to create a data view instead of a data set. This means that the *description* of the file is stored on disk, and extra CPU is used to build the data file from that description. Thus, the huge OUTBOOT data file is constructed on the fly, instead of being stored on disk. The file is then piped into the middle step, where SAS does all the heavy lifting for you. This may take significantly longer than our previous code (although orders of magnitude less than the macro code at the beginning of the paper), but has the benefit that it can be run when drive space is tight.

However, you really need to ask yourself the following question. If you have this many observations, should you be doing bootstrapping at all? Bootstrapping is a useful tool, but it does not solve all problems.

OTHER BOOTSTRAP METHODS USING THE DATA STEP APPROACH

The code above - every example of code above - only does one kind of bootstrap. But there is not a single bootstrap. There are many tools which use the fundamental boostrap re-sampling methodology. So there is not **a** bootstrap, but many bootstraps. The above code only shows how to code up what is known as the 'simple bootstrap' or 'naïve boostrap'. Below we will see some other bootstraps, and show how to modify our basic DATA step bootstrap code to create them.

THE SMOOTH BOOTSTRAP

The smooth bootstrap uses the basic bootstrapping methodology, but adds a new wrinkle. Instead of having far too many observations, we have the case where we have far too *few* distinct observations. If we have a data set with only, say, ten or twenty distinct values for X, then no simple resampling protocol is going to give us a normal-looking density. We will always end up with a density that looks like a sample from a discrete population. So how can we get a resampling process that will yield the long-term behavior we need for bootstrap theory?

One way is to add a little bit of mean-zero random noise to each of the resampled observations. This becomes equivalent to sampling from a kernel density estimate of the data. The code to perform this is very simple, once we have the previous code to work from.

```
  sasfile YourData load;
  data outboot(drop=__i);
    do Replicate = 1 to 1000;
      do __i = 1 to numrecs;
        p = int(1 + numrecs*(ranuni(39573293)));
        set YourData point=p nobs=numrecs;
        x = x + rannor(457383)/sqrt(numrecs);             /* 1 */
        output;
        end;
      end;
    run;
  sasfile YourData close;

  ods listing close;
  proc univariate data=outboot;
```

```
    var x;
    by Replicate;
    output out=outall kurtosis=curt;
    run;
  ods listing;

  proc univariate data=outall;
    var curt;
    output out=final pctlpts=2.5, 97.5 pctlpre=ci;
    run;
```

The only change we make is adding line [1]. Here we take our variable X and add some random noise from a normal distribution with mean zero and variance 1/sqrt(N). If plots of a few of the realizations do not look like a normal curve, then we can simply increase the size of the error term.

If this bothers you because the variable X occurs several times and you'll need to replace X with a variable name, simply add a macro variable, using a %LET statement to encapsulate the changes you'll be making:

```
  %let bvar = X;

  sasfile YourData load;
  data outboot(drop=__i);
    do Replicate = 1 to 1000;
      do __i = 1 to numrecs;
        p = int(1 + numrecs*(ranuni(39573293)));
        set YourData point=p nobs=numrecs;
        &BVAR = &BVAR + rannor(457383)/sqrt(numrecs);
        output;
        end;
      end;
    run;
  sasfile YourData close;

  ods listing close;
  proc univariate data=outboot;
    var &BVAR ;
    by Replicate;
    output out=outall kurtosis=curt_&BVAR ;
    run;
  ods listing;

  proc univariate data=outall;
    var curt_&BVAR ;
    output out=final pctlpts=2.5, 97.5 pctlpre=ci;
    run;
```

This code can be further tweaked, if you want to, by the use of the CALL STREAMINIT() routine and the RAND() function to generate both uniform and normally distributed pseudo-random numbers. Tests on basic cases have not shown any significant improvements in the speed of the code, though.

THE PARAMETRIC BOOTSTRAP

People usually view the bootstrap as a non-parametric approach, so many see this as something of an oxymoron. But the concept is simple. Instead of re-sampling from the observed sample, we use information from the observed sample to generate numbers from a defined distribution, and we do the re-sampling from that.

The following code assumes that the number of records in the input data set YourData is already known, and held in the macro variable &NUMRECS . If you don't have this ahead of time, it can be captured using a quick PROC SQL step or DATA step (or macro code or SCL code that gets the number of records).

```
/* make sure the macro variable is left-aligned */
%let NUMRECS = %left(&NUMRECS);

data outboot (drop = o1-o&NUMRECS);
   array obs{&NUMRECS} o1-o&NUMRECS ;
   array param{&NUMRECS} _temporary_;

   /* 1: load the OBS array with the observations of the variable X */
   do until(eof);
      set YourData end=eof;
      obs{_n_} = x;
      end;

   /* 2: compute any statistics needed for building the distribution */
   ave = mean(of o1-o&NUMRECS);
   std = std (of o1-o&NUMRECS);

   /* 3: load PARAM with values from the theoretical distribution */
   do _n_ = 1 to dim(param);
      param(_n_) = ave + std * rannor(39473725);
      end;

   /* 4: now re-sample from the theoretical distribution */
   do Replicate = 1 to 1000;
      do _n_ = 1 to &NUMRECS;
         p = int(1 + &NUMRECS *(ranuni(19573293)));
         xNew = param{p};
         output;
         end;
       end;
   run;

ods listing close;
proc univariate data=outboot;
  var xNew;
  by Replicate;
  output out=outall kurtosis=curt;
  run;
ods listing;

proc univariate data=outall;
  var curt;
  output out=final pctlpts=2.5, 97.5 pctlpre=ci;
  run;
```

The single data step can be revised in several ways. Most importantly, there is no need to restrict this to a case where we compute a normal distribution using a simple mean and standard deviation.

In part [1], we use a Whitlock do-loop (also known as a DOW-loop) to load the array OBS with the values of our variable X. The explicit do-loop lets us run through the records of the data set YourData without needing to iterate through the entire DATA step. This lets us use the DATA step like a do-block in a regular procedural language. The variable _n_ is automatically created by the DATA step as a record counter, so that makes it convenient to use when loading the array also. Note that _n_ is automatically deleted at the completion of the DATA step, so we continue to use it as a DO-variable in steps [3] and [4] merely because it is convenient (and we know there is no variable in the original data set with that name).

In part [2], we are computing the sample mean and sample standard deviation for the theoretical distribution in part [3]. But you don't have to use them. There are lots of other statistical functions available in the DATA step, and if you want particular quantities which are not available as functions, you can compute them yourself (the numbers are sitting there in an array for you) or you can compute them using a tool like PROC STDIZE and then use those results. For example, we could compute a median with the MEDIAN() function and an interquartile range with the IQR() function, instead of using the mean and standard deviation. (A normal distribution has an IQR of 1.35 standard deviations, so dividing the IQR by 1.35 gives a more robust estimate of the standard deviation.) But this method is a lot more flexible than that. You can gather whatever descriptive statistics you want at this point, and then use them to generate any distribution at all in step [3].

Step [4] uses the same ideas we developed earlier, only instead of sampling repeatedly from the input data set using the POINT= option, we simply do the same kind of URS sampling from the array of theoretical values. This builds the B bootstrap samples to be fed into the middle step.

CONCLUSIONS

The main point you should take away from this paper is that you don't have to turn resampling and simulation processes into huge, painful, time-consuming chunks of SAS macro code. All too often, the pain of writing bad code leads people in the wrong direction. Instead of finding a better way of writing the code, people abandon the code altogether and go elsewhere. But people often resort to SAS macro code before they learn that SAS usually has several tools that will help them to do their task. You no longer need to do that. In particular, you now have tools for working with extremely large data sets, as well as tools for performing smooth bootstraps and parametric bootstraps.

Tools like bootstrapping and simulation are very useful statistical tools, and will run very quickly in SAS.. if we just write them in an efficient manner. The code snippets presented in this paper provide new ways of performing some common kinds of bootstraps that will run far more efficiently than a huge macro loop to perform the same analyses.

REFERENCES

Cassell, David L. "Don't Be Loopy: Re-Sampling and Simulation the SAS® Way". SAS Institute Inc., 2007. Proceedings of the 2007 SAS Global Forum. Cary, NC: SAS Institute Inc.

Efron, B. (1981). Nonparametric estimates of standard error: The jackknife, the bootstrap and other methods. Biometrika, 68, 589-599.

Efron, B. (1982). The jackknife, the bootstrap, and other resampling plans. Society of Industrial and Applied Mathematics CBMS-NSF Monographs, 38.

Efron, B., & Tibshirani, R. J. (1993). An introduction to the bootstrap. New York: Chapman & Hall, software.

SAS OnlineDoc® 9.1.3, Copyright © 2002-2005, SAS Institute Inc., Cary, NC, USA; All rights reserved. Produced in the United States of America.

Thompson, Paul A. "A Tutorial on Bootstrapping in the SAS System". SAS Institute Inc., 1996. Proceedings of the Twenty-first Annual SAS Users Group International Conference. Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

CONTACT INFORMATION

The author welcomes questions and comments. He can be reached at his private consulting company, Design Pathways:

David L. Cassell
Design Pathways
3115 NW Norwood Pl.
Corvallis, OR 97330
DavidLCassell@msn.com
541-754-1304