Paper SP13

Don't Be Loopy: Re-Sampling and Simulation 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 better way. A faster way. This paper will show a simpler way to perform bootstrapping, jackknifing, cross-validation, and simulations from established populations. It is simpler and more efficient to get SAS to build all the iterations in one long SAS data set, then use by-processing to do all the computations at once. This lets us use SAS features to gather automatically the information from all the iterations, for simpler computations afterward.

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 2.5^{th} percentile to the 97.5^{th} percentile. We can get both these values from PROC UNIVARIATE, as we will see below.

THE SIMPLE BOOTSTRAP

Before we can look at a more effective approach, we need to look at what is usually done. So we'll start with a worst-case scenario. All too often, the SAS code that we see 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)
```

Now what's wrong with this macro? Let's just mention in passing the less important things. There are no RUN statements, an issue which has potential liabilities in macro programming. (Note: if you don't know why, then you need to be using RUN statements more!) PROC APPEND does not require the BASE data set to exist, so the whole branching construct around the PROC APPEND is unnecessary. The seed is generated internally by SAS, so there is no way to reproduce the results. (Always use your own seed, or else your boss/client/auditor/reviewer will definitely ask you to reproduce your results.) The POINT= method picks through the data set in a non-sequential manner, which may be much slower than the usual SAS sequential reading of data sets, depending on the amount of buffering that SAS is able to do for you. And the NOBS= option does not always work: data set views and sequential data sets (like data off tape drives) do not surface the value of NOBS, because it may not be possible to have that information in the data set header.

The important part 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 file (or your output window, 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.

BETTER BOOTSTRAP CODE

Making a SAS procedure open, run, and close 1000 times is never going to be as fast as executing that SAS procedure once. Even if that SAS procedure uses a BY statement with 1000 levels of the by-variable. So a much faster way to perform this same operation is to create a single data set with all the replicates in it, 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.

Let's re-do this same process now:

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

proc univariate data=outboot;
  var x;
  by Replicate;                                      /* 7 */
  output out=outall kurtosis=curt;
  run;

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

Whoa. Can this be right? An entire bootstrap process in three short steps? Yes. 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 [1], 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 [2], 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 [3], 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 [4], 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 [5], 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 [6], 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 [7], 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.

One point that is not obvious from the above code is that ODS can be used with this approach. If the procedure of interest does not expose the desired statistics with any of the usual output mechanisms (the OUTPUT statement or the OUTEST= option or the OUTSTAT= option, for example), then we may need to use the ODS OUTPUT statement to build a data set that holds the relevant data. If we do this, then the by-variable will automatically be used, and the desired output data set will automatically be built the way we want it by ODS.

Consider a similar example, using ODS to build the data set that we want. This time, we will pull out the mode. (Note that the following code completely ignores the fact that any given random sample from your data might have more than one mode.) This example uses the mode solely because it is one of the descriptive statistics that people do not think of as being available through the OUTPUT statement.

```
proc surveyselect data=YourData out=outboot
      seed=30459584
      method=urs samprate=1 outhits
      rep=1000;
  run;
```

```
ods output Modes=modal;

proc univariate data=outboot modes;
  var YourVariable;
  by Replicate;
  run;

ods output close;

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

Note the ODS OUTPUT statements bracketing the PROC UNIVARIATE code. The first ODS OUTPUT statement assigns the ODS table Modes to a data set that we name. If we have more than one mode for Replicate i, there will be more than one record for Replicate i in the data set MODAL, so in real life we would have to build in rules for handling such a case. Then the second ODS OUTPUT statement closes the output destination and makes the data set available for use in the program. The mode is stored in the variable MODE in the data set, so that we can summarize the behavior of the mode in our final step.

CAN WE IMPROVE ON OUR IMPROVEMENTS?

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 using the %JACKBOOT macro available from the SAS webpages. Or you should consider 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 like so:

```
sasfile YourData load;

proc surveyselect data=YourData out=outboot
      seed=30459584
      method=urs samprate=1 outhits
      rep=1000;
  run;

sasfile YourData close;
```

Also, we found 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).

```
ods listing close;

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

ods listing;
```

ODS LISTING CLOSE turns off the ODS destination that has our list output, while ODS LISTING 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.

So now our bootstrap code for our kurtosis example is:

```
sasfile YourData load;
proc surveyselect data=YourData out=outboot
      seed=30459584
      method=urs samprate=1 outhits
      rep=1000;
  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;
```

Here is a timing result. Now this is a simple case, and PROC UNIVARIATE runs quickly, so this example should make the original macro look better.

REP=1000

	n=500	n=5000	n=50,000
macro	00:00:53	00:01:22	00:07:20
boot2	00:00:04	00:00:20	00:04:15

REP=2000

	n=500	n=5000	n=50,000
macro *	00:01:56	00:02:40	00:14:21
boot2	00:00:08	00:00:33	00:07:40

* for this row, output window fills up every time and has to be cleared

Note that these results show an improvement of a factor of 2 to 10. These factors are a lot smaller than other test results, as we will see later. Also note that these results are totally dependent on available RAM, system settings, hard drive speeds, and even the amount of time needed to run one instance of the procedure that we are using. So don't read too much into these times. Just remember that a procedure which requires an hour to run on your data set would then require far more than 1000 hours for bootstrapping of that same process (at 1000 replicates) if you insist on using that clunky macro code. So time issues need to be considered before we start using re-sampling methodologies without a thought for the consequences.

SAS will do as much buffering in RAM as it can, so that the POINT= option doesn't slow down the macro too much, until we cross a boundary between 5,000 and 50,000 records and we have to start reading from the hard drive more. Note that the last case (n=50,000 and rep=2000) means that we have built a data set of 100 million records for our by-processing bootstrap, and it still runs faster than the macro approach. For even larger data sets or even larger numbers of replicates, we should consider alternatives to this by-processing approach, just because of the size of the OUTBOOT data set.

CASE RESAMPLING

We can take the same idea and apply it to modeling. If we perform a bootstrap when we use regression or a similar multivariate modeling method, and we grab an entire row of the data set for each sample selection, then we call this 'case resampling'. As long as the data set is fairly large, this is usually not a problem.

However, the approach can be criticized. In regression and related modeling problems, the explanatory variables may be fixed. Even if they are not fixed, they are often measured with more precision than the response variables. In addition, the range of the regressors will define the scope of the information we can get from them. This means that, in theory, case resampling can mean that each of our bootstrap samples loses some information. We will talk about alternative bootstrapping approaches after we discuss case resampling.

Let's take a simple regression problem, and let's use a small data set for illustrative purposes.

```
data temp1;
  x=1; y=45; output;
  do x = 2 to 29;
    y = 3*x + 6*rannor(1234);
    output;
    end;
  x=30; y=45; output;
  run;
```

Now we have a data set which (in theory) has a known slope of exactly 3. Of course, the random noise added in means that the estimate of the slope will have noise in it too. That's a fundamental feature of linear regression. The problem comes in with data points X=1 and X=30. These points are not only outliers, they are points of influence, or leverage points if you prefer. If you plot these data, you will see that the data lie on a straight line, except for the two endpoints. These will distort the estimation of the slope, while also affecting the estimates of variability.

We could perform a simple linear regression using PROC REG. If we do, we will find that the estimate of the slope is well under the known parameter value of 3. Or we could use case resampling and our previous bootstrap structure to come up with a non-parametric bootstrap confidence interval for the slope. Alternatively, we could use PROC ROBUSTREG, which is design to handle data which might have outliers and/or leverage points. Let's try all three:

```
/* first we do simple linear regression */
proc reg data=temp1;
  model y=x;
  run;

/* then we do robust regression, in this case, MM-estimation */
proc robustreg data=temp1 method=MM;
  model y=x;
  run;
```

```
/* and finally we use a bootstrap with case resampling */
ods listing close;

proc surveyselect data=temp1 out=boot1
     method=urs samprate=1 outhits rep=1000;
  run;

proc reg data=boot1 outest=est1(drop=_:);
  model y=x;
  by replicate;
  run;

ods listing;

proc univariate data=est1;
  var x;
  output out=final pctlpts=2.5, 97.5 pctlpre=ci;
  run;
```

Here are the results, with the three approaches and the resulting confidence intervals for the slope:

PROC REG	(1.74, 2.80)
PROC ROBUSTREG	(2.39, 3.13)
bootstrap (case resampling)	(1.65, 2.90)

Note that the robust regression did a lot better. It actually captured the correct slope in its 95% confidence interval. If you run the above code, you will also find that the robust regression correctly identified the number of problem cases. The bootstrap code did not find the correct slope, even though it did do a better job of capturing the variability problems caused by the two endpoints.

As an aside, let me warn you ahead of time. The case resampling approach is not perfect, but it is not the problem here. We will soon try this again, using resampling of residuals, and that will not solve the problems either. Just accept that bootstrapping does not solve all problems, at least without putting in the exploratory data analysis and preparatory work that you would need to do with any modeling project.

Not shown above is the use of our original macro code to do the bootstrapping. If we do that, even with this small data set, we get that the macro code takes a factor of 320 times longer to run the bootstrap than our three-step method. That is based on computer-specific and software-specific features, so your mileage may vary. But expect that complex modeling procedures may take 1 to 4 orders of magnitude longer to run using the macro approach to bootstrapping, as opposed to the three-step approach recommended in this paper. That will vary depending on far too many factors to control.

RESAMPLING RESIDUALS INSTEAD OF CASE RESAMPLING

A different way to perform bootstrapping in regression and other modeling problems is to resample the residuals instead of resampling the entire cases. The method is a bit more complex.

[1] Fit the model and retain the fitted values.
[2] For each case (the fitted value and the regressors all together), add a randomly resampled regression residual to the fitted Y-hat. Do this to create B replicates, where B is something like 1000.
[3] Now get the estimate from each of the B replicates and compute the desired bootstrap estimate.

It is arguable as to the choice of residuals. Some people prefer the raw residuals, while others prefer the studentized residuals. Still others suggest doing it both ways and comparing results. Commonly, it makes little difference. In the code below, we will use the raw residuals. We will use the TEMP1 data set we used before.

```
%let regressors = x;
```

```
%let indata = temp1;

/* 1: perform the regression and get the predicted and residual values */
proc reg data= &INDATA;
  model y=&REGRESSORS;
  output out=out1 p=yhat r=res;
  run;

/* 2: split the data: only the residuals will require URS */
data fit(keep=yhat &REGRESSORS order) resid(keep=res);
  set out1;
  order+1;
  run;

/* 3: this doesn't do any sampling - it copies the FIT data set repeatedly */
proc surveyselect data=fit out=outfit method=srs samprate=1 rep=1000; run;

/* 4: this does the WR sampling of residuals for each replicate */
data outres2;
  do replicate = 1 to 1000;
    do order = 1 to numrecs;
      p = ceil(numrecs * ranuni(394747373));
      set resid nobs=numrecs point=p;
      output;
      end;
    end;
  stop;
  run;

/* 5: then the randomized residuals are merged with the unrandomized records */
data prepped;
  merge outfit outres2;
  by replicate order;
  new_y=yhat+res;
  run;

/* 6: the bootstrap process runs on each replicate */
proc reg data=prepped outest=est1(drop=_:);
  model new_y=&REGRESSORS;
  by replicate;
  run;

/* 7: and the sampling distribution is aggregated */
proc univariate data=est1;
  var x;
  output out=final pctlpts=2.5, 97.5 pctlpre=ci;
  run;

proc print; run;
```

This bootstrap code is not as simple as before, but it is still significantly shorter than the typical macro code for resampling residuals. At 8 steps, it is still going to take up less space in your log than the equivalent 5003-step macro to do the same thing. And it will run far faster too.

Let's walk through this process. Step 1 is our regression, performed one time on our base data set. This gives us our fitted values of Y and our residuals. Step 2 then splits the data into the residuals and everything else. Only the residuals need to be randomly selected. This step also adds a counter ORDER to the FIT data set. We will use that later when we merge the data back together.

Step 3 uses PROC SURVEYSELECT to do something with no sampling at all. It appends the FIT data set to itself 1000 times, with no randomization. You are right: PROC SURVEYSELECT wasn't designed for this purpose. But if

you request a sampling method like simple random sampling, and you ask for a 100% sample, the proc is smart enough to simply return the original data set, in original order. So the REP=1000 option gives us a stack of 1000 copies of the FIT data set. This is far faster than 999 PROC APPEND statements.

Step 4 creates 1000 replicates of a URS sample of the residuals. Each replicate has an ORDER variable too, so that we can easily merge this data set with the new data set form step 3. The step creates a random number P which is an integer from 1 to the number of records, and the SET statement then points at this record for the next residual to select. This gives us Unrestricted Random Sampling. We could use PROC SURVEYSELECT here, but then we would also have to randomly permute the residuals within each value of REPLICATE, and we would have to create the ORDER variable too. The data step does this directly. Note the STOP statement. Without that, the data step will naturally loop back to the top of the step, repeat, …, until your disk drive fills up. This is not a good thing. Don't forget the STOP statement when using POINT= .

Step 5 then merges the two data sets, giving us new Y variables made up of the fitted Y and a randomly selected residual. The merge is fairly efficient, since the data are already sorted by REPLICATE and ORDER by their construction. In theory, this could be faster when using a hash instead of the merge, but the code is complex enough for now. PROC SQL with a large enough value for BUFFERSIZE could do the hashing transparently to the user.

Step 6 then performs the 1000 regressions and puts all the results into the EST1 data set. The bootstrap percentile confidence interval is then computed.

Unfortunately, this still does not solve the outlier/leverage problem we have: our 95% bootstrap confidence interval is (1.76, 2.77) . So we are still missing the correct value of the slope parameter.

There are still more forms of the bootstrap. We have already talked about the univariate problem and used the naïve bootstrap for that. But there is a parametric bootstrap which can be used on very small samples. And there is the smooth bootstrap, which adds in some zero-centered random noise (usually normally distributed noise) as part of its sample selection process, to adjust for issues like overly-discrete data. This then becomes equivalent to sampling from a kernel density estimate of the original data. And there are more methods, including the double bootstrap and similar adjusted bootstraps, which are needed for survey sample data. Data from survey samples are not independent, and typically have well-defined correlation structures. These would not be suitable for the types of bootstraps we have discussed above.

THE JACKKNIFE

Like the bootstrap, the jackknife is a non-parametric statistical approach. It is typically used to estimate bias, and to compute non-parametric estimate of standard errors. With the jackknife, we create replicate samples – not by sampling randomly as we did with the bootstrap – but by systematically dropping subsets of the original data. Most commonly, this is done by dropping the kth record from the kth replicate. If we start with a data set of n records this means that we end up with n replicates, each of which has n-1 records in it.

The jackknife is less general than the bootstrap, which has many variations. However, it turns out that it is easier to apply the jackknife to complex sampling schemes. Use of the jackknife in estimation on multi-stage samples is a common technique.

The jackknife will often yield results similar to those of the bootstrap. But there are known cases where the bootstrap performs better, with biases in the jackknife estimates. And you need to remember that the bootstrap is a random resampling method, while the jackknife is not. If you run the bootstrap on the same data set with different initial seeds, you will get (slightly) different results each time. That will not happen with the jackknife. The systematic construction of the replicates ensures that you will get the same replicate data sets, and hence the same result, each time.

It seems like PROC SQL should be a natural way to construct the jackknife data set. What we really want to do is to join the data set with itself, dropping only the pairings where the record number matches itself. So we can create a data step view that holds a record counter REC, and then use that to create our N replicates:

```
data test2 / view=test2;
  set test;
  rec=_n_;
```

```
  run;

proc sql noprint;
  create table outb as
    select a.rec as replicate, b.*
      from test2 a, test2 b
      where a.rec^=b.rec;
  quit;
```

The data step creates a data step view, rather than a regular SAS data step. This creates something which is, in essence, a set of instructions for making a data set, even though SAS treats as a data set. This can save on disk space for large data sets, and it can save time if SAS does not have to read and write a large data set to disk.

The data step creates a record counter, so that we can match records against themselves as we doing the join. Then the PROC SQL step creates the REPLICATE variable and adds it to the set of all the variables in the data view.

It is easy to see that we can use a single DATA step to do the same thing. All we have to do is mimic the techniques that we used in Step #4 in our bootstrap code in which we randomized the residuals of a statistical model. So we can build a jackknife sample like so:

```
data outb;
  do replicate = 1 to numrecs;
    do rec = 1 to numrecs;
      set test nobs=numrecs point=rec;
      if replicate ^= rec then output;
      end;
    end;
  stop;
  run;
```

Again, we use the NOBS= and POINT= options of the SET statement, only this time we do not do any random sampling. In each replicate, we pick every record except one. In the kth replicate, we omit the kth record, thus giving us N replicates of size N-1 each.

And we can also do this using PROC SURVEYSELECT. Assume that we have a data set of size N. Now we request a sample of size N (or a sampling rate of 100%, which is the same thing). If we use a sampling method which cannot select any element twice (like simple random sampling without replacement), then the procedure is smart enough to realize that it is going to be returning the entire data set. So it returns the entire data set, in original (un-randomized) order. This gives us a way of appending the data set to itself K times, where K is going to be specified using the REP= option. (In the previous section, we used this technique to get 1000 copies of the FIT data set stacked in a single data set.)

Once we have this new data set OUTB1, we can create a data step view of this data set, but with the Jth record omitted for replicate J, for J = 1, …, N. That gives us the same bootstrap sample as we had before.

```
proc sql noprint;
  select count(*) into :size from test;
  quit;

proc surveyselect data=test out=outb1
       method=srs samprate=1 rep=&SIZE. ;
  run;

data outb2 / view=outb2;
  set outb1;
  if replicate=mod(_n_,&SIZE.)+1 then delete;
  run;
```

This version seems less intuitive than the previous two. It deliberately uses PROC SQL to get a count of the rows, so that we have an alternative to the DATA step approach, in the case that NOBS is not surfaced in the base data set. Remember that views and sequential data sets (like data sets stored on tape drives) will not surface this attribute.

Let's try these out using a simple test data set of varying size:

	----- Time required (min:sec) -----			
Records	SQL	Data	Surveyselect	final data set size
500	00:02	00:01	00:03	- - -
5000	00:46	00:57	00:40	~1 Gig
15000	07:04	05:39	05:23	10 Gigs
20000	10:33	08:37	08:49	16 Gigs
25000	16:18	13:25	13:21	25 Gigs

The timings are averages of 5 runs each. Differences of less than a few seconds are not significant, as they are smaller than the variation from run to run of the same code. This is a natural feature of code run on modern OSes, because the complexities of the OS require CPU and I/O usage by underlying system processes.

Note that, in general, the data step version and the PROC SURVEYSELECT version take about the same amount of time, while (at least for the larger files) the PROC SQL version takes about 20% longer than the other two. This may only mean that the 'natural' way to build the jackknife data set is not the most efficient way to build it using PROC SQL. Also note that the times required, and the final data set sizes, grow roughly as the square of the number of records. This is logical, because the final data set must have N*(N-1) records in it.

Now let's look at the use of the jackknife. We'll use the same task as for our original (naïve) bootstrap. This time, we will use the data-step version of our jackknife generator, just for simplicity.

```
data outb;
  do replicate = 1 to numrecs;
    do rec = 1 to numrecs;
      set test nobs=numrecs point=rec;
      if replicate ^= rec then output;
      end;
    end;
    stop;
  run;

ods listing close;
proc univariate data=outb;
  var y;
  by replicate;
  output out=outall kurtosis=curt;
  run;
ods listing;

proc univariate data=outall;
  var curt;
  output out=final mean=jmean std=jstd;
  run;
```

As before with the bootstrap, the process is straightforward: we make a single data set that holds all our replicates; then we use y-processing to analyze all the replicates in a single step; and finally we aggregate the results from each replicate into a jackknife statistic.

RANDOMIZATION TESTS

Permutation tests and randomization tests are resampling plans which approach the problem from a slightly different angle. The underlying idea is to compare the results from the real data against the possible results if we re-label the data points, then see how extreme the results form the real data are, when compared against this array of alternatives arrangements of the data. This gives us a non-parametric statistical test of our hypothesis. There is a fuller exposition of this idea in the Cassell paper in the references section of this paper, but below is a short version.

Suppose we have a dataset of biodiversity measures taken from a probability sample of sites across three types of habitat. Unfortunately, some sites have no biota in the taxa of interest, yielding a large number of zeroes in the resulting dataset. The poorer habitats are more likely to have zeroes for diversity measures, so these zeroes have to be preserved in the data. The data are therefore clearly non-normal, and may even have serious issues with heterogeneity of variance.

An alternative is the standard permutation test. This test uses all possible distinct permutations of the dependent variable, holding the independent variables fixed. Performing the original statistical analysis on all such permutations allows one to evaluate how the actual structure of the data compares to random re-arrangements of the data. This gives us a non-parametric way of assessing the significance of the hypothesis test.

Unfortunately, a typical full permutation test is too time-consuming. Consider the above example with a small dataset. Assume there are 34 observations in three habitats, with 12 observations in the first two habitat types and 10 in the third habitat. The number of distinct permutations is thus 34!/(12!12!10!) which is unfortunately a whopping 355 trillion distinct permutations to assess. For a medium-sized dataset, the number of permutations becomes utterly unmanageable.

An alternative to the full permutation test is what is often called a randomization test. It uses a Monte Carlo approach to select a random subset of the total number of permutations, so that the computations can be done in a reasonable amount of time. The randomization test wrapper in the references provides a pair of SAS macros that form a wrapper about the intended analysis. We will skip the macro code just to demonstrate the methodology involved, and to make it clear that this is one more example of resampling, although we re-label the Y values instead of randomly selecting them.

In a permutation test or randomization test, we want to run the regular data to come up with a test statistic or p-value, and then run the randomizations to compare against that base value, to see how ‘unusual’ it is when compared against the data with permuted values of our Y. That means that we will want our long, thin data set of replicates to start with the original data (we will call it replicate 0) before we append the randomized versions.

The first three steps create the permutation data set. The essential trick is that it is easier to store the Y values, randomize the entire records, and then write over the randomized records with the unrandomized Y values. This gives us an equivalent to randomizing the Y values for each replicate and applying them to unrandomized records. So the first step provides replicates, each with a random number RAND_DEP that we use in the PROC SORT to randomize the records within each value of REPLICATE.

The third step appends the original data to the newly-sorted replicate cases. The original data get the value REPLICATE=0. Then the Y values are loaded into a temporary array. When the newly-randomized replicates are read in, their old values of Y are over-written by the values from the array. Since the cases are randomized within each value of REPLICATE, this randomly arranges the Y values in a different order for each replicate. At this point, the data are ready for processing.

```
%let reps=1000;

data __temp_1;
  retain seed 2356464;  drop seed;
  set YourData nobs=numrec;
  do replicate = 1 to &REPS;
    call ranuni(seed,rand_dep);
    output;
    end;
  if _n_=1 then call symputx('numrecs', numrec);
  run;

proc sort data=__temp_1; by replicate rand_dep; run;
```

```
data outrand;
  array deplist{ &NUMRECS } _temporary_ ;
  set YourData(in=in_orig)  __temp_1(drop=rand_dep);
  if in_orig then do;
    replicate=0;
    deplist{_n_} = y ;
    end;
  else y = deplist{ 1+ mod(_n_,&NUMRECS) };
  run;
```

The MIXED procedure uses by-processing to get hypothesis tests for the original data, and also for every new replicate. In this example, we are using ODS to get the comparisons between habitats, so that we can compare Habitat U to Habitat Z.

```
ods output Diffs=diffs;
ods listing close;

proc mixed data=outrand;
  by replicate;
  class habitat;
  model FishS = habitat;
  lsmeans habitat / pdiff ;
  run;

ods output close;
ods listing;
```

The DIFFS data set from PROC MIXED now has multiple comparisons. We will pull only the U vs. Z comparison. That is easy enough to subset out, using a WHERE statement in the following step.

The randomization test is now straightforward. REPLICATE=0 represents the p-value of the test statistic for our original data. We store that in the variable PVALUE, and we compare it against each of the p-values for the following replicates. The non-parametric estimate of the significance level is simply the proportion of p-values that are more significant than that of our original data. (So this could have been done using the test statistics instead of the p-values for the test statistics, if you wanted to.)

```
data _null_;
  retain pvalue numsig numtot 0;
  set diffs end=endofile;
  where Habitat='U' & _Habitat='Z';
  if Replicate=0 then pvalue = probt;
  else do;
    numtot+1;
    numsig + ( probt < pvalue );
    end;
  if endofile then do;
    ratio = numsig/numtot;
    put "Randomization test for habitats U vs Z has sig. level of  " ratio 6.4 ;
    end;
  run;
```

So we end up with a randomization test in only 5 steps. The timing tests done for the referenced paper suggest that a macro to do this processing would take about three orders of magnitude longer (and thousands of steps) to execute.

CROSS-VALIDATION

Cross-validation is another resampling method, which takes yet another approach to model evaluation. When people talk about using hold-out samples, this is not really cross-validation. Cross-validation typically takes K replicate

samples of the data, each one using (K-1)/K of the data to build the model and the remaining 1/K of the data to test the model in some way. This is called a K-fold cross-validation.

For a sample of size N, leave-one-out-cross-validation, or LOOCV, acts a little like our jackknife structure, taking N-1 of the data points to build the model and testing the results against the remaining single data point, in N systematic replicates, with the kth point being dropped in the kth replicate. So this is the K-fold cross-validation taken to its extreme, with K=N.

In addition, the random K-fold cross-validation does not split the data into a partition of K subsets, but takes K independent samples of size N*(K-1)/K instead. This last version readily falls into our PROC SURVEYSELECT feature set. Let's do this one first. We'll once again use our TEMP1 data set from before. Remember that it has two massive leverage points.

```
%let K=3;
%let rate=%sysevalf((&K-1)/&K);

/* generate the cross-validation sample */
proc surveyselect data=temp1 out=xv seed=495857
        samprate=&RATE outall rep=3;
  run;

data xv;
  set xv;
  if selected then new_y=y;
  run;

/* get predicted values for the missing new_y in each replicate */
proc reg data=xv;
  model new_y=x;
  by replicate;
  output out=out1(where=(new_y=.)) p=yhat;
  run;

/* summarize the results of the cross-validations */
data out2;
  set out1;
  d=y-yhat;
  absd=abs(d);
  run;

proc summary data=out2;
  var d absd;
  output out=out3 std(d)=rmse mean(absd)=mae;
  run;
```

The %LET statements at the top provide a computation of the sampling rate: for a K-fold cross-validation, we always want (K-1)/K of the data points in each replicate sample. The %SYSEVALF() macro function lets us perform floating point calculations within the macro code, so here &RATE will be a string that holds the value 0.6666666667 . (Remember that the macro code is just evaluating and then doing string substitution before the SAS code is parsed.)

The key feature of the PROC SURVEYSELECT statement is the OUTALL option. OUTALL tells the procedure to output all of the records from the input data set, but to mark the selected sample records. The way it does this is with a new variable, SELECTED. SELECTED is 1 for the chosen records and 0 for the rest. So in each of the K replicates, all N records appear, with (K-1)/K of them flagged with SELECTED=1 .

The subsequent data step uses SELECTED to prepare for the model validation steps. We will keep our real response value Y, and perform the modeling on NEW_Y, which will be missing for all the values we want to use as our hold-out group in each replicate. The PROC REG step then fits our model, and predicts a Y-hat for each value where NEW_Y is missing. We keep only these records, since they are all we need for our model evaluations.

Two of the popular measures of model performance in cross-validation are the Root Mean Square Error (RMSE) and the mean absolute error (MAE). The RMSE is just the standard deviation of the differences Y – YHAT , while the MAE is just the average of their absolute values. The differences D and their absolute values ABSD are computed in the following data step, and then passed to PROC SUMMARY to get the RMSE and MAE in a straightforward manner.

Now we can use the same sort of code that we used in building the jackknife samples to create the leave-one-out-cross-validation replicates. We then pass the results to our modeling proc in the same way as before:

```
/* do LOOCV */
data xv;
  do replicate = 1 to numrecs;
    do rec = 1 to numrecs;
      set temp1 nobs=numrecs point=rec;
      if replicate ^= rec then new_y=y;  else new_y=.;
      output;
      end;
    end;
    stop;
  run;

/* get predicted values for the missing new_y in each replicate */
proc reg data=xv;
  model new_y=x;
  by replicate;
  output out=out1(where=(new_y=.)) p=yhat;
  run;

/* and summarize the results of the LOOCV */
data out2;
  set out1;
  d=y-yhat;
  absd=abs(d);
  run;

proc summary data=out2;
  var d absd;
  output out=out3
  std(d)=rmse mean(absd)=mae;
  run;
```

Here we use the same code as before, except that the generation system changes. We create N replicates for an N-record data set, and in the kth replicate, we mark observation k as our hold-out to be predicted by our model.

Regular K-fold cross-validation can be done in a similar way. We have to partition the data set into K separate groups of roughly equal size. (Obviously, we can't always make the size of the data set be a multiple of K, so we often have to settle for *nearly* equal sizes.) Then we use that to build a data set of K replicates, each of which uses K-1 of these groups as the modeling set and the remaining group as the hold-out for the replicate. Once we have that data set, the process is exactly as we have shown above.

SIMULATIONS

We can use our basic approach when working with simulations too. In particular, when we want to subset a given population of data points because we cannot afford to perform simulations on all of them, the utility of PROC SURVEYSELECT is immediate. We can pull out a sample of a given size for use in complex simulation analysis in the same way that we have used PROC SURVEYSELECT already. If we only need one record at a time for our simulations, then we can build a data set of records to be processed, and then use by-processing to work through the process.

Here, we will show only a trivial example. We will pull out a simple random sample of size 1000 for use in our simulation. Here, the simulation is done in a data step, loading the variables A1 to A25 in a 5x5 array for further use, perhaps evaluation of the transition states of a 5-state Markov process.

```
proc surveyselect data=largefile out=process_set seed=45884743 method=srs
sampsize=1000;
  run;

data processor;
  array{5,5} a1-a25;
  set process_set;
    . . . . .
  run;
```

Monte Carlo methods are an extremely general class of methodologies. Any kind of simulation that uses probabilities and random numbers to investigate the behavior of a process may be claimed to be a 'Monte Carlo' method.

This means that PROC SURVEYSELECT is but one of many ways to perform Monte Carlo methods in SAS. PROC SURVEYSELECT gives us a way of choosing random records from already-extant data sets. But we have seen that the data step is also a good way to build a stack of replicate samples for further processing. Also, PROC PLAN in the SAS/STAT® module is an excellent way of generating random data.

For example, suppose we need 100 different replicates, each of which has randomly selected 30 of the 200 sites in our study. We can do that using a data step, as we have shown above. But PROC PLAN is designed to generate data sets like this:

```
proc plan seed=4958584;
  factors replicate=100 ordered
          SiteNo = 30 of 200 / noprint;
  output out=plan9;
  run;

proc print data=plan9; run;
```

The ORDERED keyword ensures that the output data set has the 100 replicates in order from 1 to 100. The option for SITENO ensures that we get a random sample of 30 out of the 200 site numbers (which are assumed to be 1 to 200 by default). The default selection method (when no keyword like ORDERED is used) is simple random sampling without replacement.

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 a tool that will help them to do their task.

It is simple and straightforward to write little 'wrappers' around SAS processes to perform tasks like bootstrapping, jackknifing, randomizations tests, and other resampling methodologies. These wrappers generate data sets so that we can use SAS by-processing to solve our problems.

The naïve bootstrap is usually well-suited to univariate problems, and the residual-resampling bootstrap provides a way of working with modeling techniques. Both do make assumptions about the exchangeability of the underlying data, so we cannot use them blindly. Data with correlation structures, unequal weights, etc. may require more complex bootstrapping methodologies. Very small data sets and data sets with severe discreteness may also require additional methodologies.

Tools like bootstrapping and simulation are very useful, and will run very quickly in SAS.. if we just write them in an efficient manner.

REFERENCES

Cassell, David L. “A Randomization-test Wrapper for SAS Procs”. SAS Insitute Inc., 2002. Proceeding of the Twenty-seventh Annual SAS users Group International Conference. Cary, NC: SAS Institute Inc.

Edgington, E. S.(1995). Randomization tests. New York: M. Dekker.

Efron, B. (1979). Bootstrap methods: Another look at the jackknife. The Annals of Statistics, 7, 1-26.

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