SESUG Paper 80-2021

Coarsening Continuous Variables: An Automated Method to Create Categorical Version of Continuous Variables While Mitigating Loss of Precision

Zach Acuff, RTI International

ABSTRACT

Researchers often want to create a categorical version of a continuous variable, perhaps for analytical purposes or to make potentially identifying information more secure for public consumption. Although there are already several ways to categorize continuous data, going from a continuous to a categorical variable necessarily involve some loss of precision that may not be desirable. This paper describes a SAS® macro which presents a novel method for automatically categorizing continuous data with minimal user input. This method creates a categorical version of a continuous variable such that 1) each categorical value occurs at least as often as the user desires, and 2) the total difference between the actual continuous values and their categorical counterparts is minimized to the extent the macro is able.

INTRODUCTION

Continuous variables can, theoretically, take on an infinite number of values. In practice, of course, many variables we call continuous have practical limits regarding the number of distinct values they can take on, but still: this ability to capture the tiniest measurable differences of some attribute of interest is exactly why continuous variables are so useful. On the other hand, dealing with an excessively large number of distinct values of a continuous variable may provide too much detail from an analysis perspective. Additionally, a distinct value that occurs infrequently in a large dataset might risk identifying an individual sample member if that data is made publicly available. For situations like these, we must create a categorical version of this continuous variable that suits our needs.

Categorizing a continuous variable requires grouping the continuous data into some number of intervals based on their rank. Some common methods for doing so include grouping the data based on quantiles (e.g., four categories for each quartile of data), grouping the data into aesthetically pleasing intervals (e.g., “0—9”, “10—19”, “20—29”, “30—39”, and “40+”), or grouping the data into clinically meaningful categories (e.g., “Underweight”, “Normal”, “Overweight”, and “Obese”). Although these methods of categorizing continuous data are often sufficient, each involves a loss in precision that may not be desirable. For example, you may want the categorical values to relate to the actual corresponding continuous values and not just their rank in the data. Further, making the data available to the public might mean that the categorical version must abide by some rule, such as the requirement that the frequency for any individual value must be at least 30 (which was the impetus for the creation of the macro in this paper). Grouping continuous values in this way is not as straightforward.

The macro “%coarsen” was developed specifically to transform a continuous variable into a categorical variable such that all categorical values occur at least 30 times in the data, although this minimum frequency can be any whole number. The macro first groups the continuous values into intervals such that each interval contains at least as many data points as the user desires. Then the mean of the continuous data points in each interval is calculated and this mean becomes the new categorical value for all continuous data points in that interval. Then the sum of the absolute differences between all the continuous and categorical values is calculated. Finally, the macro attempts to adjust the endpoints of these intervals with the goal of decreasing the overall difference between the continuous and categorical values. The macro terminates when it can no longer find adjustments to the intervals which would increase the fidelity to the continuous data.

MACRO PARAMTERS AND GUIDELINES

Using %coarsen is simple, as only a small number of simple parameters is required:

```
%coarsen (ds= , orig_var= , new_var= , round_val= , group_size= );
```

ds: The name of the dataset with the continuous variable to be categorized.

orig_var: The name of the continuous variable on the dataset &ds.

new_var: The name of the categorical variable to be created.

round_val: The amount of precision desired in the categorical values. The default is 1, meaning that the new categorical values will be rounded to the nearest integer if no other value is provided.

group_size: The smallest frequency which is permitted for any individual value of the categorical variable.

A few things to note:

•The macro will automatically categorize all values of the continuous variable in the dataset, so if the user only wishes to categorize positive values and leave negative values alone, the input dataset must be subset to those records beforehand.

•It may be that the user is not concerned with the size of the categorical groups being created, but rather the number of groups created. The macro is not equipped to handle this situation directly, but taking the number of records in the input dataset and dividing by the desired number of groups should give the user a rough estimate of the required value of "group_size."

•The end result of the macro is to create the global macro variable "coarsen_code" which contains the SAS code needed to actually create the categorical variable in a DATA step. The user can simply invoke "&coarsen_code" in their own DATA step when the macro has finished.

MACRO OVERVIEW

Details on the specifics of the macro can be found in the comments of the macro code in Appendix A, so this section is meant to provide a conceptual overview of the macro's logic without focusing too much on the code itself.

Consider the SAS example dataset "bmimen", which has the age and body mass index (BMI) of 3,264 males ranging in age from 2 years to 80 years, and in BMI from 12.1 to 60.9. The variable BMI takes on 301 distinct values in this dataset, of which 52 values only occur once in the dataset. Suppose that the variable BMI needs to be made publicly available, but the publicly available version must not have any specific BMI value that occurs fewer than 10 times to prevent an individual sample member being singled out. Since many values of BMI occur fewer than 10 times (in fact, about 17% of the records on "bmimen" have values that occur fewer than 10 times in the dataset), this is a good opportunity to use %coarsen. The initial dataset has BMI rounded to the nearest tenth, but we'll let the categorical version be rounded the nearest hundredth to increase its precision. Here is the macro call for this example:

```
%coarsen (ds=SAShelp.bmimen, orig_var=BMI, new_var=coarse_BMI, round_val=.01,
group_size=10);
```

To begin deciding how to group values into intervals, first consider the distinct values of BMI in the dataset and their corresponding frequency. Here are the top rows of that output from the FREQ procedure:

	BMI	COUNT	CUM_FREQ
1	12.1	1	1
2	12.2	1	2
3	12.3	1	3
4	12.7	1	4
5	12.8	3	7
6	12.9	1	8
7	13	3	11
8	13.1	1	12
9	13.2	3	15
10	13.3	3	18
11	13.4	4	22
12	13.5	7	29
13	13.6	4	33
14	13.7	5	38
15	13.8	8	46
16	13.9	10	56
17	14	5	61
18	14.1	16	77
19	14.2	16	93
20	14.3	11	104

Figure 1. The First 20 Records in the PROC FREQ Output Dataset for "SASHELP.BMIMEN"

The first few BMI values are the lowest BMI values in "bmimen" and they only occur a single time. It isn't until the BMI value of 13 that the cumulative frequency exceeds 10, which means that the most compact way to construct this first interval is to include all values in the range [12.1, 13.0]. If we take the weighted mean of the data points in this interval rounded to the nearest hundredth, that gives us 12.69. Therefore, the first value of the categorical version of BMI will be 12.69 and have a frequency of 11. This is the code that the macro will generate for this first interval:

```
if (12.1<=BMI<=13) then coarse_BMI=12.69;
```

Continuing this trend, the next interval would then begin with 13.1 and go up to 13.4. The weighted mean of the second interval is 13.29 and also has a frequency of 11. The macro generates this code for the second interval:

```
if (13.1<=BMI<=13.4) then coarse_BMI=13.29;
```

This method seems reasonable, so we could just keep going: start with the minimum value of the interval, then (if the minimum values occurs fewer than 10 times) extend the interval to fit the next largest value until the interval covers at least 10 data points, then let that value be the endpoint of the interval, get the weighted mean of the interval, and repeat for the next interval until all the values of the original BMI variable have a new categorical value. The only other constraint to consider is that an interval should not be constructed if there are fewer than 10 data points remaining which have not been grouped into an interval (otherwise the minimum frequency requirement is not met). When this happens, all remaining values must be grouped into the final interval (which %coarsen accounts for).

However, there is a catch. Consider the first two intervals constructed above: [12.1, 13.0] and [13.1, 13.4]. Both intervals cover exactly 11 data points of the original BMI variable so the minimum frequency requirement is satisfied, but notice that the BMI value of 13.1 only occurs once in the original variable. This means that 13.1 could be moved into the first interval so that the new intervals are [12.1, 13.1] and [13.2, 13.4] and their corresponding frequencies would be 12 and 10, so the minimum frequency requirement is still met, and new means would be calculated for these intervals. This opens up the

possibility of adjusting our intervals to find ones which better fit the data, which is great, but it does add a new layer of complexity to what was previously a fairly straightforward algorithm.

Assessing which pair of intervals fits the data better is simple: compare both interval options and select the one which results in the lowest overall difference between the original continuous values and their categorical counterparts. After doing the math, the first pair of intervals (i.e., [12.1, 13.0] and [13.1, 13.4]) results in an overall difference of 3.88 units from the continuous values for the first 22 data points, and the second pair of intervals (i.e., [12.1, 13.1] and [13.2, 13.4]) results in an overall difference of 3.90 units from the continuous values of those same data points. Therefore, the first pair of intervals results in a more faithful transformation than the second pair of intervals, so the value 13.1 should indeed stay in its initial interval. Although the algorithm happened to select the best intervals on the first try in this example, there are instances where intervals should be adjusted to get a better fit. For instance, the algorithm will initially create the intervals [13.9] and [14.0, 14.1] with frequencies of 10 and 21, respectively, but a better option is to use the intervals [13.9, 14.0] and [14.1] instead (which have frequencies of 15 and 16, respectively).

Comparing the effects of different intervals and choosing the better one is easy enough, but actually identifying which intervals could be adjusted in the first place is trickier. The main difficulty is in the fact that SAS processes the dataset one record at a time, so when an interval is being constructed SAS can't simply "look ahead" to see whether additional values should be included in the interval or not; SAS can only "see" whether or not the value of the current record it's processing can be the endpoint for the interval being constructed. Therefore, although the initial algorithm can produce suitable intervals in a single DATA step, giving it the ability to dynamically adjust these intervals means the DATA step must be wrapped in a DO loop that repeats itself every time it finds an alternative to the intervals it's constructing. Upon each iteration of this DO loop, the macro constructs intervals until it identifies a value which could be placed in the interval on either side of it, at which point it terminates the DATA step and assesses the effect of both options. Whether the value in question should be "shifted" into a new interval or not is recorded by the macro and then the DO loop repeats itself, constructing the intervals from scratch again but this time remembering which "shifts" work and which ones didn't so it doesn't assess the same value twice. This DATA step terminates when it can no longer find any more values to shift into different intervals.

At this point in the "bmimen" example, the main DO loop has constructed a total of 181 intervals after undergoing 30 iterations; 30 iterations means that 29 opportunities to shift values into different intervals were identified and a decision was made on each (the 30th iteration occurred when there were no more values to shift and the macro could construct its final intervals). Of these 29 opportunities to shift values into new intervals, 16 were found to be "good" (i.e., shifting these values into different intervals results in a better fit) and 13 were found to be "bad" (i.e., shifting these values into different intervals results in a fit worse than or equal to the alternative). In many cases, %coarsen could essentially be complete at this point: the macro has considered a variety of options for grouping the continuous values into intervals and settled on the options which result in the closest fit to the original data. Even at this point, however, there is still the potential for an even better fit that the process described so far does not account for.

So far, the DO loop which contains the heart of the macro has been comparing two pairs of intervals at a time, where the difference between the pairs of intervals is that a particular value is either the maximum of the lesser interval or the minimum of the greater interval. What has not been taken into consideration, however, is the possibility that this greater interval might later be adjusted when taking greater values into account which have not yet been grouped into an interval. On the first completion of the DATA step, %coarsen constructs the intervals [31.3], [31.4, 31.7]. However, a better fit for this range would be [31.3, 31.4] and [31.5, 31.7]. The DATA step does not "see" this better alternative on its first completion because when deciding how to group the value 31.4, it was comparing the pair of intervals [31.3] and [31.4, 31.6] with the pair [31.3, 31.4] and [31.5, 31.6]. The former pair is the better fit, so the macro places 31.4 into the greater interval and "remembers" this decision on subsequent iterations of the DO loop. During one of these subsequent iterations, though, it adjusts the interval [31.4, 31.6] to become [31.4, 31.7]. Because the macro already decided to place 31.4 into the greater interval, it needs to be able to "go back" and see that [31.3, 31.4] and [31.5, 31.7] is best and that 31.4 should actually be placed in the lesser interval. Therefore, once the DATA step finishes constructing intervals for the entire variable, it must go back at least one more time to look for more opportunities to shift values from one interval to the

other, by “forgetting” the bad decisions and “remembering” the good ones since what was previously a bad value to shift into a lower interval could actually be a good value to shift upon revisiting.

Finally, once the main DATA step has constructed the same set of intervals twice in a row, this means that it can truly find no more values to shift around and the macro is completed with this final set of intervals. The “if-then” statements which create the categorical variable are assigned to the global macro variable “coarsen_code” which the user can use in their own DATA step to create this “coarsened” variable.

CONCLUSION

Although there are several well-known methods of creating a categorical version of a continuous variable, the macro %coarsen has several advantages which may make it the most suitable method for certain situations. In particular, the macro’s ability to accommodate the user’s specifications for the minimum allowed frequency of the categorical variable is a distinct feature, as figuring out suitable intervals manually would be extremely tedious and overwhelming for even moderately large sets of data. Further, the macro doesn’t just blindly construct intervals by grouping the data into compact intervals and stopping there (even though that could still result in a respectable solution); it repeatedly attempts to find better ways of constructing the intervals and each adjustment to the intervals results in a better and better fit. Finally, the new categorical variable that is created can actually be treated as continuous for analysis—even though the original continuous variable was literally categorized into a discrete number of values, there is no reason the new variable cannot be treated as continuous (provided that the underlying data is sufficiently large and has sufficient variation).

There are, however, a few caveats to keep in mind when using %coarsen. First, the method of constructing intervals for the continuous variable begins with the lowest values of the variable and gradually works its way up until it has grouped the maximum value into the final interval. The fact that the intervals are constructed in ascending order is arbitrary: it would be possible to construct the intervals in descending order and the final intervals would likely have some differences (and possibly be a better fit). This brings up the fact that the macro does not necessarily consider all possible ways of constructing intervals with the constraints given: it only considers some alternatives to the method of creating compact intervals in ascending order. The number of possibilities for grouping the variable into intervals is potentially extremely large depending on the minimum frequency requirement and the size and variation of the underlying data, so the macro does not necessarily consider all possible solutions, which means that the solution it settles on is not necessarily the optimal one. Lastly, the macro could take a long time to complete depending on the size and variation of the data and the user-supplied parameters, so the main DO loop may go through many iterations before landing on a final solution. Increasing the efficiency of the macro and coming up with a method to explore all interval options more thoroughly to potentially find a better fit are both opportunities for improving the macro in the future.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Zach Acuff
RTI International
zacuff@rti.org

APPENDIX A: CODE TO CREATE AND CALL THE %COARSEN MACRO

```
%macro coarsen (ds=, orig_var=, new_var=, round_val=1, group_size=);

/*Create the dataset "frequencies" that lists all individual values of the
continuous variable in the original dataset in ascending order along with
those counts and cumulative frequencies.*/
proc freq data=&ds. noprint;
tables &orig_var. / list missing outcum out=frequencies (keep=&orig_var.
count cum_freq);
run;

/*Record the total number of records in the original dataset and the maximum
number of records that share the same value. The former is needed for the
macro to determine how many records remain to be grouped while constructing
intervals, and both are needed to determine whether the desired recoding is
even possible in the first place.*/
data _null_;
retain max_count 0;
set frequencies end=eof;
if count>max_count then max_count=count;
if eof then do;
      call symput("nobs", strip(cum_freq));
      call symput("max_count", strip(max_count));
end;
run;

/*Terminate macro if there are not enough records to make even 2 groups of
the desired size.*/
%if %sysevalf(&nobs.<=((&group_size.*2)-1)) %then %do;
      %put ERROR: Insufficient number of records to recode to more than 1
value for variable &orig_var.. Macro will terminate.;
      %abort;
%end;

/*Terminate macro if there are not enough different values to make even 2
groups of the desired size.*/
%else %if %sysevalf((&nobs.-&max_count.)<(&group_size.)) %then %do;
      %put ERROR: Insufficient variation in data to recode to more than 1
value for variable &orig_var.. Macro will terminate.;
      %abort;
%end;

/*Initialize macro variables

good_shift_vals: "good shift values"; a string of values which should not be
the maximum of an interval because using a greater value will result in a
better fit

bad_shift_vals: "bad shift values"; a string of values which are the minimum
of intervals and should not be shifted to a lower interval because it will
not improve fit

iteration: the number of times the main DATA step has been processed
```

```
cycle: the number of times the main DATA step has produced a "final" set of
intervals; after completing the first cycle, the DATA STEP will start over,
remembering the "good shift values" but forgetting the "bad shift values" so
that those can be re-assessed after shifting established "good shift values"
since those could make a difference; the cycle ends when the intervals
produced result in the same fit as the prior intervals produced

complete: a flag variable to indicate the process is complete and the DO loop
can be exited; the process is complete when the intervals produced result in
the same fit as the prior intervals produced

prior_avg_dist: "prior average distance"; the average distance between the
original and coarsened values from the prior set of intervals produced; this
is needed so that the DO loop will terminate when the intervals by cycle N
are no better than the intervals produced by cycle N-1 (i.e., when they are
equal and no further improvement is possible*/

%let good_shift_vals=.;
%let bad_shift_vals=.;
%let iteration=0;
%let cycle=0;
%let complete=0;
%let prior_avg_dist=.;

/*This DO loop is the heart of the macro. The coarsening process is complete
when the latest set of intervals produced are equal to the previous set of
intervals produced (i.e. when no further improvement is possible by this
process)*/
%do %until (&complete.=1);

/*The "shift candidate" is a paricular value of the original variable which
is eligible to be the maximum of a particular interval, but could also
accommodate the next-largest value as the maximum of this interval with the
possible benefit of an improved fit.*/
      %let shift_candidate=.;
      %let iteration=%eval(&iteration.+1);

      data &orig_var.;
      set frequencies end=eof;

      length assess_base_code assess_shift_code $32767;

/*These retained variables are needed to compute the mean of each interval,
including the potential intervals constructed by grouping values
differently*/
      retain min min_count   group_count 0 val_sum 0 new_val second_val .
      assess_base_code "" assess_shift_code ""
      prior_max . prior_group_count . prior_val_sum . prior_new_val .;

      by &orig_var.;

/*Group_count will be 0 only when an interval is beginning to be constructed
(i.e. for the first record in the frequencies dataset, or immediately after
an interval has finished being constructed). When this happens, the minimum
of this interval will be the current value and the frequency of this value
needs to be recorded to assess whether it can be shifted*/
      if (group_count=0) then do;
```

```
            min=&orig_var.;
            min_count=count;
      end;
      group_count+count;
      val_sum+(&orig_var.*count);
      if group_count-min_count=count then second_val=&orig_var.;
      if last.&orig_var. then do;
/*This DO loop is activated when a "shift candidate" is encountered in the
data. If so, code is generated to assess the fit of both interval choices for
the variable and the data step is terminated since the remaing values of the
original variable do not need to be considered until a decision is made about
how to construct the current intervals.*/
            if (group_count>=&group_size.) and ((&nobs.- cum_freq)
            >=&group_size.) and (&orig_var. ^in (&good_shift_vals.)) then do;
                  max=&orig_var.;
                  new_val=round(val_sum/group_count, &round_val.);
                  assess_base_code=strip(assess_base_code) || " if (" ||
                  strip(min) || "<=&orig_var.<=" || strip(max) || ") then
                  base_diff=abs(&orig_var.-" || strip(new_val) || ");";
                  assess_shift_code=tranwrd(assess_base_code, "base_diff",
                  "shift_diff");
                  if prior_max ^in (&bad_shift_vals. &good_shift_vals.) and
                  (group_count-min_count)>=&group_size. then do;
                        mod_prior_new_val=round((prior_val_sum +
                        (min*min_count)) / (prior_group_count + min_count),
                        &round_val.);
                        mod_new_val=round((val_sum - (min*min_count)) /
                        (group_count - min_count), &round_val.);
                        assess_shift_code=tranwrd(assess_shift_code, "<=" ||
                        strip(prior_max) || ")", "<=" || strip(min) || ")");
                        assess_shift_code=tranwrd(assess_shift_code, "-" ||
                        strip(prior_new_val) || ")", "-" ||
                        strip(mod_prior_new_val) || ")");
                        assess_shift_code=tranwrd(assess_shift_code, "(" ||
                        strip(min) || "<=", "(" || strip(second_val) ||
                        "<=");
                        assess_shift_code=tranwrd(assess_shift_code, "-" ||
                        strip(new_val) || ")", "-" || strip(mod_new_val) ||
                        ")");
                        call symput("assess_base_code",
                        strip(assess_base_code));
                        call symput("assess_shift_code",
                        strip(assess_shift_code));
                        call symput("ceiling", strip(&orig_var.));
                        call symput("shift_candidate", strip(prior_max));
                        stop;
                  end;
                  output;
                  prior_group_count=group_count;
                  prior_val_sum=val_sum;
                  prior_new_val=new_val;
                  group_count=0;
                  val_sum=0;
                  prior_max=max;
                  second_val=.;
            end;
/*If no more shift candidates are found, construct the last interval*/
```

```
            else if eof then do;
                  max=&orig_var.;
                  new_val=round(val_sum/group_count, &round_val.);
                  assess_base_code=strip(assess_base_code) || " if (" ||
                  strip(min) || "<=&orig_var.<=" || strip(max) || ") then
                  base_diff=abs(&orig_var.-" || strip(new_val) || ");";
                  call symput("assess_base_code", strip(assess_base_code));
                  output;
            end;
      end;
      keep min  max new_val group_count;
      run;

/*Compare the effect of both interval groupings being considered and select
the one which better fits the original variable. The macro variables
"good_shift_vals" and "bad_shift_vals" will be repeatedly used by the macro
to remember these decisions on subsequent iterations of the DO loop*/
      data _null_;
      set &ds. (where=(&orig_var.<=&ceiling.)) end=eof;
      retain base_cum_diff 0 shift_cum_diff 0;
      &assess_base_code.
      base_cum_diff + base_diff;
      &assess_shift_code.;
      shift_cum_diff + shift_diff;
      if eof then do;
            if shift_cum_diff<base_cum_diff then call symput
            ("good_shift_vals", "&good_shift_vals. " || strip
            (&shift_candidate.));
            else call symput("bad_shift_vals", "&bad_shift_vals. " ||
            strip(&shift_candidate.));
      end;
      run;

/*When no more "shift candidates" can be found, the macro has found a
suitable way of coarsening the original variable. Still, the macro must
consider additional "shift candidates" that may not have been identifiable on
the first "cycle" of completing the macro. The entire DO loop must repeat at
least once more to look for these new "shift_candidates" so they can be
considered*/
      %if (&shift_candidate.=.) %then %do;

            %let prior_cycle=&cycle.;
            %let cycle=%eval(&cycle.+1);

            data _null_;
            set &ds. end=eof;
            retain base_cum_diff 0;
            &assess_base_code.
            base_cum_diff + base_diff;
            if eof then do;
                  call symput("avg_dist_cycle&cycle.", base_cum_diff/&nobs.);
            end;
            run;

            %if (&prior_cycle.^=0) %then %do;
```

```
                  %if %sysevalf(&&avg_dist_cycle&cycle. >
                  &&avg_dist_cycle&prior_cycle.) %then %put ATTENTION:
                  Average distance increased, something is wrong.;
                  %else %if %sysevalf(&&avg_dist_cycle&cycle. <
                  &&avg_dist_cycle&prior_cycle.) %then %let bad_shift_vals
                  =.;
                  %else %if %sysevalf(&&avg_dist_cycle&cycle.=
                  &&avg_dist_cycle&prior_cycle.) %then %let complete=1;
            %end;
            %else %let bad_shift_vals = .;

      %end;

/*Finally, create the macro variable "coarsen_code" which contains the code
to create the new, categorical version of the continuous variable*/
      %global coarsen_code;

      data _null_;
      set &orig_var. end=eof;
      length coarsen_code $32767;
      retain coarsen_code "";
      coarsen_code=strip(coarsen_code) || " if (" || strip(min) ||
      "<=&orig_var.<=" || strip(max) || ") then &new_var.=" || strip(new_val)
      || ";";
      if eof then do;
            call symput("coarsen_code", strip(coarsen_code));
      end;
      run;

%end;

%mend coarsen;

%coarsen (ds=SAShelp.bmimen, orig_var=BMI, new_var=coarse_BMI, round_val=.01,
group_size=10);
```