MWSUG 2016 - Paper RF11

Tips and Tricks for Producing Time-Series Cohort Data

Nate Derby, Stakana Analytics, Seattle, WA

ABSTRACT

Time-Series cohort data are needed for many applications where we're tracking performance over time. We show some simple coding techniques for effectively and efficiently producing time-series cohort data with SAS®.

INTRODUCTION

A *time series cohort* is a group of people defined by some time-related event, such as *all customers who opened a checking account in January 2016*. The goal is to track (and eventually analyze or forecast) the activity of these people over time.

Typically we'll define other cohorts and compare them to each other, so for example we'll have *all customers who opened a checking account in February 2016*, *all customers who opened a checking account in March 2016*, and so on. The dates of the activity we want to track are all relative to the event in question. So in our example, we might track what a customer does the first, second, and third month after opening a checking account. To collect this data, we'll need to find the date each customer opened the account (which will be different for every customer), then collect the data for that customer relative to that specific date the account was opened.

Clearly there's a fair amount of programming involved to do this correctly. However, with a few tips and tricks described in this paper, this can be a fairly simple task. All of these tips and tricks have the same general idea: *Let our program do the work for us!*

USE MACROS

The most important concept when working with time series cohorts is consistency. We need to apply *the exact same process* to each cohort. The best and easiest way to do this is to *use macros*! They are designed for this exact situation.

A detailed and thorough introduction to SAS macros is Carpenter (2004). As an example, we would define the macro `%collectCohortData` like this:

```
%MACRO collectCohortData( cohortDate );

  [CODE FOR COLLECTING DATA FOR THIS COHORT]

%MEND collectCohortData;
```

Then we could collect data for January and February 2016 like this:

```
%collectCohortData( 16m1 ) *for January;
%collectCohortData( 16m2 ) *for February;
```

There are of course many options for how we could code the different cohorts, but using `YYmM` for month `M` and year `YY` makes it easy to calculate the start date of our cohort:

```
%LET month = %SCAN( &cohortDate, 2, m );
%LET year = %EVAL( 2000 + %SCAN( &cohortDate, 1, m ) );
%LET cohortStart = %SYSFUNC( MDY( &month, 1, &year ) );
```

Following best practices, be sure to designate `month`, `year`, and `cohortStart` as *local* macro variables.

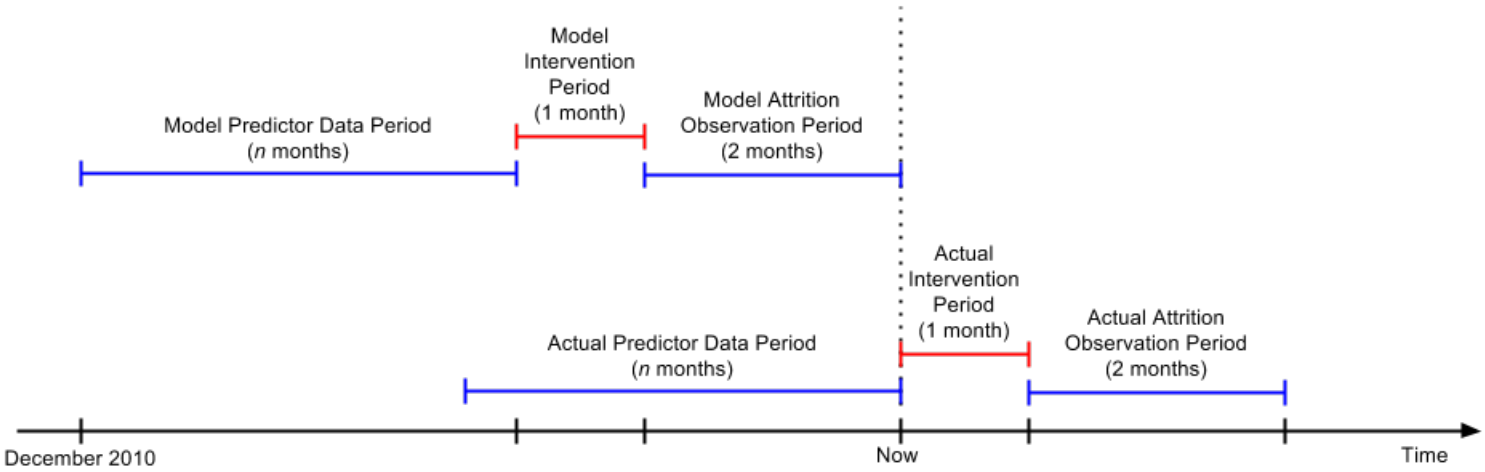

Figure 1: A diagram illustrating overlapping time-series intervals, from Derby and Keintz (2016).

DIVIDE THE CODE INTO MACRO UNITS

For effective, flexible code organization, we can divide our code into *macro units*, which are macros that that describe a specific function or phase of the analysis, as explained in Derby (2008). Here's an example:

```
%collectData;
%graphData;
%makeForecasts;
%graphForecasts;
```

where we can define `%collectData` as a series of calls to our `%collectCohortdata` macro defined earlier to collect data for four consecutive months:[1]

```
%MACRO collectData;

  %collectCohortData( 16m1 );
  %collectCohortData( 16m2 );
  %collectCohortData( 16m3 );
  %collectCohortData( 16m4 );

%MEND collectData;
```

The other macros can be defined similarly.

DEFINE MANY DATES IN TERMS OF ONE OR TWO SPECIFIC DATES

Time-series cohorts typically use dates that are defined relative to one or two *specific* dates. For example, in Figure 1 we have two overlapping sets of three time-series intervals, one of which is the other one shifted by three months. Both of them are defined by the dates *Start Date* (December 2010) and *Now*, which is the upper end of one set of intervals and three month before the upper end of the second set of intervals. *Start Date* (the date we started collecting data) will never change, but *Now* will always change.

We never want to hard-code our dates (i.e., put their actual value into the code) because we'll most likely change those dates at some point later on (e.g., when *Now* changes in Figure 1). If we hard-code those dates, we would need to change every single one of those dates later on. If we define all these dates relative to these two definitive dates (especially *Now*, which will always change), we only need to change the value of that one date.

[1] For more elegant code, we could rename the `%collectCohortdata` macro as `%collectData` and define it so that it applies to one specific cohort when there's an argument in parentheses, but applies to *all* cohorts when it stands alone without an argument. This underlying structure is called a *recursion* and is described in Derby (2010).

On page 1, we already illustrated this by defining the cohort start date as the macro variable `cohortStart`. For a more complicated example (with overlapping time intervals), look at Figure 1 again. The full context of this example is in Derby and Keintz (2016), but the main point is that we have two sets of time intervals: The *modeling* data set (with predictor data starting on *Start Date*, ending on *Now* minus 3 months) and the *scoring* data set (starting on *Start Date* plus 3 months, ending on *Now*). Here's how we would code that within our macro, called `%prepareData`:

```
%MACRO prepareData( dataSet );

  %IF &dataSet = modeling %THEN %DO;

      %LET predictorStartDate = &startDate;
        %* starting at the earliest transaction date ;
      %LET predictorEndDate = %EVAL( &now - 84 );
        %* ending three months ago ;

      %END;
    %ELSE %IF &dataSet = scoring %THEN %DO;

      %LET predictorStartDate = %EVAL( &startDate + 84 );
        % starting at the earliest transaction date plus three months ;
      %LET predictorEndDate = &now;
        % ending now ;

      %END;

      [SAS CODE FOR PULLING/PROCESSING THE DATA, USING THE MACRO VARIABLES ABOVE]

%MEND prepareData;
```

DEFINE ONE MONTH AS 4 WEEKS

For time series programming, months are messy. They have an inconsistent number of days, and that number of days is almost never a multiple of 7. Some months will have more Saturdays (or other days of the week) than others, which creates many inherent problems. For example, if we're analyzing retail data and comparing one month with 4 Saturdays to another month with 5 Saturdays, the one with 5 Saturdays will inherently have more sales just because it has more Saturdays. The same applies to Sundays with church attendance, or Mondays at work. These are examples of a *weekly seasonality effect*, meaning that we'll have some patterns over the days of the week. This almost always happens.

An easy way to avoid this is to simple define a month as four successive weeks. It doesn't matter how we define a week; it can be Sunday to Saturday, Tuesday to Monday, or Thursday to Wednesday. The point is that every "month" will have the same number of each weekday so that these weekly seasonality effects can average out. We'll still have monthly seasonality effects, such as Independence Day in July, kids returning to school in September, and Christmas in December. But those are easier to deal with.

However, there are three caveats to this rule:

- This only applies if we're looking at *generic* months, most commonly as the number of preceding months.
- When we going back 10 or more months, we'll have some inconsistency issues, since 10 months does *not* equal 40 weeks. As a rule of thumb, this trick only applies when looking back over less than 10 months, with the fewer the better.
- Somewhere in the output, clearly mention that we're defining a month as a series of 4 weeks. Otherwise we're misrepresenting the data.

We use this approach in the `%prepareData` macro above, where we define three months as $3 \times 4 \times 7 = 84$ days.

USE DATE AND TIME FUNCTIONS AND FORMATS

Calculating dates and times can be complicated. Which years are leap years? Which months have 31 days? What is the day of the week of a given date? If we use times, do we use different time zones? When we're programming for a specific time period, it may be tempting to simply hard code a number or use a simplified calculation for it. For example, March has 31 days and the number of years is the number of days divided by 365. But what happens when we want to move on to a different month? Dividing the days by 365 is always a little off because of leaps years.

More generally, a one-time project for one specific time period can easily become a more permanent project for many different time periods. Why not spend a little extra time with our code and use the proper function that calculates these quantities the right way? We'll thank ourselves later when we suddenly have to change the parameters of our code.

Here are two especially useful date/time functions:

- `INTCK`: Calculates the number of intervals between two date/datetime values. For example, how many months are there between two dates?
- `INTNX`: Calculates the date/datetime of the start of the interval a specified number of intervals from the interval that contains a given date/datetime. For example, what is the date of the beginning of the month three months from now?

Formats for date or time variables may seem unimportant until we're one day faced with looking at a date and seeing the number of days since January 1, 1960. We could format these values at the end of the analysis, but why not format them at the beginning? With one short statement, we can format our variables properly and have them correctly formatted everywhere in our analysis, including when we want to quickly look at a data set midway through our process.

A short but thorough introduction to date and time functions is Karp (2010). A comprehensive list of SAS date and time formats (and informats) is at SAS Institute (1999) (newer sources aren't as useful as this one).

To see this in action, let's return to our `%collectCohortData` macro from page 1. Using the `MDY` function, we defined the macro variable `cohortStart` as the first day of the month defined by `&month` and `&year`:[2]

```
%LET cohortStart = %SYSFUNC( MDY( &month, 1, &year ) );
```

How do we calculate the *last* day of the month? Since we don't know which month this is, we can't simply designate the 28th, 29th, 30th or 31st day of the month. We could try to be clever and take the first of the next month and subtract 1:[3]

```
%LET cohortEnd = %EVAL( %SYSFUNC( MDY( &month+1, 1, &year ) ) - 1 );
```

However, this doesn't work if the month is December, since that would create a 13th month and cause an error. Happily, the `INTNX` function takes care of this:[4]

```
%LET cohortEnd = %SYSFUNC( INTNX( month, &cohortStart, 0, end ) );
```

In fact, we could even use `INTNX` to fix our clever solution above, going to the first of the next month and subtracting 1 day:

```
%LET cohortEnd = %EVAL( %SYSFUNC( INTNX( month, &cohortStart, 1, beginning ) ) - 1 );
```

Both of these two above solutions with `INTNX` work equally well.

[2] Since we're in the macro language, we must use the `%SYSFUNC` function to execute the `MDY` function.

[3] The `%EVAL` macro function is used here to do integer arithmetic, which otherwise wouldn't be done in the macro language.

[4] Since we're (again) in the macro language, we must use the `%SYSFUNC` function to execute the `INTNX` function, plus remove the single quotes from the first and fourth arguments.

EMPIRICALLY CALCULATE END DATES

Sometimes we'll define a cohort (or time interval connected to a cohort) by the end date of a data set. Even if we know this data set well, *don't assume what this date is*. If this date doesn't match the actual end date of our data set, our analysis could easily be done completely wrong.

Likewise, if we have multiple data sets (such as customers, accounts, and transactions) that are supposed to have the same end date, don't believe it. Empirically calculate the end dates of each one, then use the *lowest* one for our analysis, because otherwise at least one data set will have no data for the most recent dates. This would cause major problems, since for time series analysis, the most recent data are our most important data. So **our most important data would be completely wrong**, thus seriously compromising our statistical models.

We can do this through a set of simple SQL statements with macro variables:

```
PROC SQL NOPRINT;
  SELECT MAX( effectiveDate )
    INTO :end1
    FROM accounts;
  SELECT MAX( transactionDate )
    INTO :end2
    FROM transactions;
QUIT;

%LET end = %SYSFUNC( MIN( &end1, &end2 ) );
```

CONCLUSIONS

Although there's an inherently complex process to process data for time series cohorts, these tips and tricks can make it much easier to do that by taking full advantage of SAS functions and the SAS macro language. We'll inherently make our programs do much of the work for us.

REFERENCES

Carpenter, A. (2004), *Carpenter's Complete Guide to the SAS Macro Language*, second edn, SAS Institute, Inc., Cary, NC.

Derby, N. (2008), Guidelines for organizing SAS project files, *Proceedings of the 2008 SAS Global Forum*, paper 083-2008.
http://www2.sas.com/proceedings/forum2008/083-2008.pdf

Derby, N. (2010), Using recursion for more convenient macros, *Proceedings of the 2010 Western Users of SAS Software Conference*.
http://www.wuss.org/proceedings10/coders/2953_7_COD-Derby.pdf

Derby, N. and Keintz, M. (2016), Reducing credit union member attrition with predictive analytics, *Proceedings of the 2016 SAS Global Forum*.
http://support.sas.com/resources/papers/proceedings16/11882-2016.pdf

Karp, A. (2010), Working with SAS date and time functions: Foundations & Fundamentals, *Proceedings of the 2010 SAS Global Forum*, paper 134-2010.
http://support.sas.com/resources/papers/proceedings10/134-2010.pdf

SAS Institute (1999), SAS date, time, and datetime values.
https://v8doc.sas.com/sashtml/lrcon/zenid-63.htm

ACKNOWLEDGEMENTS

I'd like to thank Evan Kass for giving me the idea of writing this paper. Thanks also to Art Carpenter who has, over many years, always inspired me to write better and more reusable code.

CONTACT INFORMATION

Comments and questions are valued and encouraged! Please don't hesitate to contact me:

Nate Derby
Stakana Analytics
815 First Ave., Suite 287
Seattle, WA 98104-1404
nderby@stakana.com
http://nderby.org
http://stakana.com