Paper 373-2008

BFF: Binary Flags Forever (Or, Boole's Paradise)

David J. Pasta, ICON Clinical Research, San Francisco, CA
Eric Elkin, ICON Clinical Research, San Francisco, CA

ABSTRACT

Binary flags, also called indicator variables or dummy variables, take on only the values 0 and 1. Manipulating these variables with arithmetic operations and calculating descriptive statistics on them can be a very efficient way to summarize data in different ways. Some arithmetic operations correspond to logical operations (Boolean algebra). For example, the product of two binary flags corresponds to AND, the maximum corresponds to OR, the sum of binary flags gives the count of values that are 1, and the mean of binary flags gives the proportion of values that are 1. One common use of binary flags is to indicate which inclusion and exclusion criteria an observation satisfies. This allows construction of a "subject flow chart" from the original sample to the final analysis sample that can be changed easily to apply the criteria in a different order. This paper gives general information on the use of binary flags as well as several practical examples of their use in PROC FREQ and PROC MEANS to achieve various goals very efficiently.

INTRODUCTION

Most of you are probably familiar with the idea of creating variables that take on only the values 0 and 1. Sometimes called indicator variables or dummy variables, in the context of this paper we will term them binary flags: binary because they take on two values and flags because they can be used to signal when a condition holds.

Creating lots of binary flags – sometimes more than you think would be needed – can make it easy to write (and read!) code that implements complex logical expressions. The manipulation of logical expressions using AND, OR, and NOT follow a set of rules referred to as Boolean algebra. Named after English mathematician George Boole (1815-1864), this refers to the system of logic he described in detail in 1854. Boole's work is considered the foundation of computer science and the design of digital circuits. Boolean algebra works with any two symbols, but it is customary (and most convenient) for those to be the numbers 0 and 1. There are equivalencies between Boolean algebra and arithmetic manipulations of binary numbers, and that makes them especially useful in the context of SAS programming.

In addition to being useful for manipulating and documenting logical conditions, binary flags are conveniently used to summarize information across observations of a dataset. Examining sets of flags in PREC FREQ with the LIST option is especially helpful to understand how many cases meet which logical conditions. PROC MEANS can be used to summarize binary flags in various ways. For example, the mean of a binary flag is the proportion of values that is 1. The sum gives the count of the number of times the value is 1. The maximum over a set of binary flags is 1 if any of the flags are 1; the minimum is 1 only if all of the flags are 1. This allows a convenient way to use arithmetic to ask "is it ever true that …" or "is it always true that …" questions. It is often especially effective to summarize multilevel level data to higher level data using binary flags in PROC MEANS with the BY and/or CLASS statement.

CREATING SIMPLE BINARY FLAGS

It is easy to create binary flags using assignment statements of logical expressions or using if-then statements. For example, suppose you have a variable RACE that takes on the values . (missing), 1 (for White/Caucasian), 2 (for Black/African American), and 3 (for Asian/Pacific Islander). Then one way to create binary flags would be as follows:

```
b_white=(race eq 1);
b_black=(race eq 2);
b_asian=(race eq 3);
```

In SAS, a logical expression such as (race eq 1) takes on the value 1 if it evaluates to "true" and 0 if it evaluates to "false." This code fragment exemplifies several conventional practices that you may find helpful in working with binary flags. The first is that the names for the binary flag variables all begin b_ (for binary). This makes it easy to remember which variables are binary flags and they will sort together in an alphabetical list of variables. You might use f_ (for flag), t_ (for true or truth), l_ (for logical), or d_ (for dummy) if you prefer. (Reserving d_ for SAS date variables may be preferable than using it to represent dummy variables.) The point is to adopt a convention that works for you. Also, the part of the variable name after the b_ is designed to describe what the variable means when it is true. This leads to code that reads more naturally. Another convention is the use of "eq" instead of "=" in logical expressions, and enclosing the entire logical expression in parentheses. Neither is strictly necessary, but it does tend to make the code easier to read. Consider the confusion if the code fragment read as follows (yes, it works):

```
b_white=race=1;
b_black=race=2;
b_asian=race=3;
```

BE CAREFUL ABOUT MISSING VALUES

One advantage of binary flags is that it forces you to think explicitly about missing values as you create the flags and as you manipulate them. We all know we should be careful about missing values, but sometimes we get a little sloppy and assume that missing values will be handled the way we would want them to be handled if we were thinking about how they should be handled which we aren't because that's annoying and besides we need to get on with the writing of the code and can't be bothered with every little thing when we have these good ideas we need to get into code right away before we forget.

Which leads to errors and unintended consequences that can be really hard to find and figure out.

So a good habit to get into is creating binary flags to indicate the "missingness" status of key variables such as demographic or other "classification" variables, predictors, or outcomes. The ability to test for missingness quickly and easily through the use of binary flags will more than pay back the minor effort to create the flags. It may be natural to code the flag as true (or 1) if the value is missing, but our experience is that it is better to code the flag as true (or 1) when the value is present. So, when creating flags for the RACE variable we would also create one more:

```
b_race=not missing(race);
```

This line of code exemplifies additional conventions (and an exception). First, the name of the resulting binary variable contains the name of the variable (or an abbreviation). Second, the code uses the MISSING function rather than engaging in the dangerous practice of testing for only the standard missing value

```
b_race=(race ne .);
```

or the more accurate but somewhat obscure

```
b_race=(race gt .z);
```

in part because it tends to lead to code that is easier to read. Although generally it looks best to have the entire logical expression in parentheses for clarity, we find that in the case of "not" it is acceptable to omit the outer parentheses.

CREATING BINARY FLAGS FOR VARIABLES WITH ONLY TWO VALUES

You might think it would be redundant to create binary flags for variables that take on only two values, but it is not. If the two-valued variable is character, it is of course convenient to have a numeric version. If a numeric variable takes on two values other than 0 and 1, it is convenient to have a binary version. Finally, if the two-valued variable can be missing, it is especially helpful to create binary flags to handle that explicitly.

For example, suppose you have a variable SEX that takes on the value 1 (for males) and 2 (for females). We would create three binary flags:

```
b_sex=not missing(sex);
b_male=(sex eq 1);
b_female=(sex eq 2);
```

These variables would tell you that you know the subject's sex (b_sex), that you know the subject is male (b_male), or that you know the subject is female (b_female).

COMBINING BINARY FLAGS INTO NEW FLAGS

Once you have binary flags for several variables or conditions, you may find it useful to combine them into new flags. This is easily done either with Boolean algebra (logical expressions) or with arithmetic expressions. The one key thing to remember is to think about how you want to handle missing.

The logical operator AND (known as *conjunction* in formal logic) combines two logical values so that the result is true only if each input is true. This corresponds to multiplying two 0/1 variables: the product is 1 only if both of the inputs are 1. You can get the same result if you use the MIN function. The logical operator OR (known as *disjunction*) combines two logical values so that the result is true if either input it true. This corresponds to taking the maximum of two 0/1 variables: the maximum is 1 if either input is 1. The logical operator NOT (known as *negation*) reverses the truth value. This corresponds to subtracting a value from 1. If X is 0, then 1-X is 1; similarly, if X is 1, then 1-X is 0. This is reasonably straightforward … if it were not for missing values.

BE CAREFUL ABOUT MISSING VALUES

When SAS calculates a product of two numbers, the result is missing if either value is missing. However, the MIN function will calculate the minimum of the non-missing values. Thus if X and Y are binary flags, X*Y is not necessarily the same as MIN(X,Y) if missing values are present. The way SAS handles logical expressions that include missing values is a potential trap for the unwary. In numerical operations, missing values are simply treated as the very large negative values used to represent them; this is different from the conventions of some other programming languages. In SPSS, for example, "three-valued logic" is used where missing is a third possible result of a logical expression. This deviates from Boolean logic, but in some cases provides an answer that is more useful to the programmer. In SAS, the rule in logical expressions is this: *1 is true and anything else (including missing) is false*. Here is a summary table worth pondering:

X	y	NOT x	1 - x	x AND y (3-valued)	x AND y (SAS)	x * y	MIN(x.y)	x OR y (3-valued)	x OR y (SAS)	MAX(x,y)
0	0	1	1	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	1	1	1
1	0	0	0	0	0	0	0	1	1	1
1	1	0	0	1	1	1	1	1	1	1
.	0	1	.	0	0	.	0	.	0	0
.	1	1	.	.	0	.	1	1	1	1
.	.	1	.	.	0	.	.	.	0	.

Three-valued logic says that if you can tell whether the answer must be true or false regardless of the missing value, that should be the outcome; otherwise the outcome is missing. SAS treats missing values as false. That means when X is missing, "NOT(X)" takes on the value 1, but of course the arithmetic equivalent (1-X) will be missing.

DE MORGAN'S LAWS

De Morgan's laws are named after Augustus De Morgan (1806-1871). Although they make perfect sense, they are not always known by that name. One brief way to express them is, "the negation of a conjunction is the disjunction of the negations, and the negation of a disjunction is the conjunction of the negations." Perhaps they are easier to follow as "not(P **and** Q) = (not P) **or** (not Q)" and "not(P **or** Q) = (not P) **and** (not Q)." These are useful tools for formally rewriting logical expressions. But always remember to consider the implications of missing values.

EXAMPLE OF COMBINING BINARY FLAGS

Suppose we want to find the non-white males in our sample. There are a number of ways we could do this using the binary flags we have created. In the code below, the letter L in the variable names refers to using a logical expression and A to using an arithmetic expression, the M stands for Male, and AB refers to Asian/Black and NW to NonWhite.

```
lmab=(b_male and (b_asian or b_black));
amab=bmale*max(basian,b_black);
lmnw=(b_male and not b_white);
amnw=bmale*(1-b_white);
```

In the presence of missing data, these different approaches give slightly different answers:

sex	race	b_male	b_white	b_asian	b_black	lmab	amab	lmnw	amnw
.	.	0	0	0	0	0	0	0	0
.	1	0	1	0	0	0	0	0	0
.	2	0	0	0	1	0	0	0	0
.	3	0	0	1	0	0	0	0	0
1	.	1	0	0	0	0	0	1	1
1	1	1	1	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1
1	3	1	0	1	0	1	1	1	1
2	.	0	0	0	0	0	0	0	0
2	1	0	1	0	0	0	0	0	0
2	2	0	0	0	1	0	0	0	0
2	3	0	0	1	0	0	0	0	0

DESCRIPTIVE STATISTICS ON BINARY FLAGS USING FREQ AND MEANS

Binary flags lend themselves to summarization as frequency distributions (especially with the LIST option in PROC FREQ) and as descriptive statistics. Consider some simulated data containing SEX and RACE variables for each of 309 subjects. A small fraction of subjects have missing values. Simple frequency distributions on the variables yield the following:

sex	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	178	58.75	178	58.75
2	125	41.25	303	100.00

Frequency Missing = 6

race	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	248	86.11	248	86.11
2	30	10.42	278	96.53
3	10	3.47	288	100.00

Frequency Missing = 21

Creating the binary flags as described above, we use the following code to produce a LIST mode frequency distribution and summary statistics:

```
proc freq data=p02;
   tables sex*b_sex*b_male*b_female
      race*b_race*b_white*b_black*b_asian
      / list missing;
run;
```

```
proc means data=p02 maxdec=3 n nmiss min max mean sum;
   var race sex b_sex–b_asian;
run;
```

Notice the use of the MISSING option on the TABLES statement in PROC FREQ. This is not required but is strongly recommended because it provides details on how the missing values are being handled.

sex	b_sex	b_male	b_female	Frequency	Percent	Cumulative Frequency	Cumulative Percent
.	0	0	0	6	1.94	6	1.94
1	1	1	0	178	57.61	184	59.55
2	1	0	1	125	40.45	309	100.00

race	b_race	b_white	b_black	b_asian	Frequency	Percent	Cumulative Frequency	Cumulative Percent
.	0	0	0	0	21	6.80	21	6.80
1	1	1	0	0	248	80.26	269	87.06
2	1	0	1	0	30	9.71	299	96.76
3	1	0	0	1	10	3.24	309	100.00

Variable	N	N Miss	Minimum	Maximum	Mean	Sum
race	288	21	1.000	3.000	1.174	338.000
sex	303	6	1.000	2.000	1.413	428.000
b_sex	309	0	0.000	1.000	0.981	303.000
b_male	309	0	0.000	1.000	0.576	178.000
b_female	309	0	0.000	1.000	0.405	125.000
b_race	309	0	0.000	1.000	0.932	288.000
b_white	309	0	0.000	1.000	0.803	248.000
b_black	309	0	0.000	1.000	0.097	30.000
b_asian	309	0	0.000	1.000	0.032	10.000

Notice that the PROC MEANS output provides much of the same information as the frequency distributions in compact form.

USING BINARY FLAGS WITH MULTILEVEL DATA

One of the most valuable uses of binary flags is when processing multilevel data. The levels might be site, patients within sites, and observations within patients. Or they might be schools, classrooms, students, courses, and tests. Whatever the nature of the levels, binary flags can help characterize the values at the lower level. This is especially helpful when applying inclusion-exclusion criteria.

To provide a concrete example, we have created simulated data with 20 sites, each of which has 11 to 20 patients for a total of 309 patients. Each patient has patient-level characteristics (SEX and RACE) but also has one or more visit records. The visits are erratically spaced over a two-year time period, with VDATE ranging from 1 to 730. We have been asked to find out how many patients we would include if we required that each patient meet the following criteria: first visit within 60 days of enrollment, at least 5 visits total, and the visits span at least 6 months. But at the meeting where the study team came up with those criteria, we heard some discussion that maybe the first visit could be within 90 or even 120 days; that maybe as few as 3 visits would be enough; and that perhaps the data should span 9 or 12 months. So we decide to create a series of flags.

After exploring the data, we decide to summarize the visit date VDATE to the patient level using PROC MEANS:

```
proc means data=v01 noprint nway;
   class site patid sex race;
   var vdate;
   output out=p01(drop=_type_) min=minv max=maxv range=rangev n=numv;
run;
```

The use of the NOPRINT and NWAY options means that there will be no printed output and only the full combinations of the CLASS variables will be output. Notice that the CLASS statement could have included just the SITE and PATID. However, it is convenient to carry along the variables that are at the level of the patient or higher, such as SEX and RACE. This avoids having to merge those values onto the patient-level dataset later. The OUTPUT statement is used to create various summary statistics for the VDATE variable and specify the associated variable names.

From the patient-level dataset P01, we can create a series of binary flags for minimum visit date, MINV:

```
b_lovle2=(0 le minv le 60);
b_lovle3=(0 le minv le 90);
b_lovle4=(0 le minv le 120);
b_loveq3=(60 lt minv le 90);
b_loveq4=(90 lt minv le 120);
```

We use LO in the variable name instead of MIN to keep the name short and use a number referring to the approximate number of months to help us remember the variable. Notice that we create both a binary flag for the range 0 to the end value and also for the incremental part of the range. Which is more useful will depend on our purpose. Similarly, we create variables for NUMV, the number of visits, and RANGEV, the range of the visit dates in days, as follows:

```
b_nvge3=(numv ge 3);
b_nvge4=(numv ge 4);
b_nvge5=(numv ge 5);
b_nveq3=(numv eq 3);
b_nveq4=(numv eq 4);

b_rvge06=(rangev ge 183);
b_rvge09=(rangev ge 274);
b_rvge12=(rangev ge 365);
b_rv0609=(183 le rangev lt 274);
b_rv0912=(274 le rangev lt 365);
```

Now we can create a variable for the specified criteria:

```
b_v2506=(b_lovle2 and b_nvge5 and b_rvge06);
```

We have our answer to the question we were asked, and we can provide additional detail about how many patients met which of the criteria by looking at a LIST mode frequency distribution of the component variables:

b_v2506	b_lovle2	b_nvge5	b_rvge06	Frequency	Percent	Cumulative Frequency
0	0	0	0	30	9.71	30
0	0	0	1	84	27.18	114
0	0	1	0	2	0.65	116
0	0	1	1	105	33.98	221
0	1	0	0	3	0.97	224
0	1	0	1	12	3.88	236
1	1	1	1	73	23.62	309

But in fact the researchers forgot to mention that they also need to have SEX and RACE nonmissing. Because SEX and RACE were carried along in the CLASS statement in PROC MEANS, those patient-

level variables are available without having to do a MERGE to add them to the dataset. That makes it easy to create the necessary flags. An alternative would have been to create B_SEX and B_RACE in the visit-level dataset and add them to the CLASS statement in the PROC MEANS. Adding B_SEX and B_RACE to the LIST mode frequency distribution adds them to the inclusion-exclusion criteria:

b_sex	b_race	b_v2506	b_lovle2	b_nvge5	b_rvge06	Frequency	Cumulative Percent	Cumulative Frequency
0	1	0	0	0	0	2	0.65	2
0	1	0	0	0	1	2	0.65	4
0	1	0	0	1	1	2	0.65	6
1	0	0	0	0	0	4	1.29	10
1	0	0	0	0	1	3	0.97	13
1	0	0	0	1	0	1	0.32	14
1	0	0	0	1	1	4	1.29	18
1	0	1	1	1	1	9	2.91	27
1	1	0	0	0	0	24	7.77	51
1	1	0	0	0	1	79	25.57	130
1	1	0	0	1	0	1	0.32	131
1	1	0	0	1	1	99	32.04	230
1	1	0	1	0	0	3	0.97	233
1	1	0	1	0	1	12	3.88	245
1	1	1	1	1	1	64	20.71	309

Taking this example a step further, one could explore how many patients would be added if we were to adjust the criteria. For example, we could consider looking at `b_v2506*b_lovle3*b_lovle4*b_nvge3*b_nvge4`. In one short table we can see how many patients we would add using various combinations of the criteria:

b_v2506	b_lovle3	b_lovle4	b_nvge3	b_nvge4	Frequency	Percent
0	0	0	0	0	36	11.65
0	0	0	1	0	26	8.41
0	0	0	1	1	80	25.89
0	0	1	0	0	3	0.97
0	0	1	1	0	10	3.24
0	0	1	1	1	21	6.80
0	1	1	0	0	2	0.65
0	1	1	1	0	13	4.21
0	1	1	1	1	45	14.56
1	1	1	1	1	73	23.62

CUMULATIVE FLAGS VERSUS INCREMENTAL FLAGS

As mentioned in connection with creating the flags, sometimes it is convenient to use cumulative flags such as b_nvge3, b_nvge4, and b_nvge5 and other times it is convenient to use incremental flags for intermediate values, so that the set of flags might be b_nveq3, b_nveq4, and b_nvge5. Some people are more comfortable with one version than another, but generally we find the cumulative flags are more useful more often. To see how the two sets work together, consider the following frequency distributions:

b_nvge3	b_nvge4	b_nvge5	Frequency	Percent	Cumulative Frequency	Cumulative Percent
0	0	0	41	13.27	41	13.27
1	0	0	49	15.86	90	29.13
1	1	0	39	12.62	129	41.75
1	1	1	180	58.25	309	100.00

b_nvge5	b_nvge4	b_nvge3	Frequency	Percent	Cumulative Frequency	Cumulative Percent
0	0	0	41	13.27	41	13.27
0	0	1	49	15.86	90	29.13
0	1	1	39	12.62	129	41.75
1	1	1	180	58.25	309	100.00

b_nveq3	b_nveq4	b_nvge5	Frequency	Percent	Cumulative Frequency	Cumulative Percent
0	0	0	41	13.27	41	13.27
0	0	1	180	58.25	221	71.52
0	1	0	39	12.62	260	84.14
1	0	0	49	15.86	309	100.00

b_nvge5	b_nveq4	b_nveq3	Frequency	Percent	Cumulative Frequency	Cumulative Percent
0	0	0	41	13.27	41	13.27
0	0	1	49	15.86	90	29.13
0	1	0	39	12.62	129	41.75
1	0	0	180	58.25	309	100.00

All four distributions provide essentially the same information but the third one gives it in a different order. The first two versions use the cumulative flags and the resulting order of categories is the same. The third distribution uses the incremental flags and the order is somewhat unnatural: the patients who meet none of the criteria are first, then those who meet the most stringent criterion, then the next most stringent criterion, and so on. By specifying the flags in the other order, a more natural ordering results.

OTHER WAYS TO SUMMARIZE MULTILEVEL INFORMATION

It is natural to summarize visits to the patient level, but it may also be useful to summarize information to the site level. Returning to the examples of binary flags for RACE, one might get the SUM of B_BLACK to get the number of Blacks at the site, the MAX of B_ASIAN to see if there are any Asian patients, and the MIN of B_WHITE to see if all the patients are White. Before interpreting the results, though, be sure to think about how a missing RACE would be handled. Consider the following distribution:

maxasian	maxblack	minwhite	Frequency	Percent	Cumulative Frequency	Cumulative Percent
0	0	0	2	10.00	2	10.00
0	0	1	1	5.00	3	15.00
0	1	0	11	55.00	14	70.00
1	0	0	1	5.00	15	75.00
1	1	0	5	25.00	20	100.00

The first line shows us there are two sites that have no Black patients, no Asian patients, but not every patient was White … because at least one patient had missing RACE.

Sometimes it is helpful to summarize lower levels of data as percentages rather than as proportions. This can be handled by multiplying the mean of a binary flag by 100 or by creating a flag that takes on the values 0 and 100. The disadvantage of creating a flag taking values 0 or 100 is that logical manipulation no longer works; both values are treated as False in SAS.

In some situations, creating flags that are missing or 1 can be useful as an alternative. Before venturing into this territory, be sure you will be able to accomplish what you intend.

CONCLUSION

Binary flags – variables that take on only the values 0 and 1 – can be extremely useful tools. They can be manipulated with logical expressions (Boolean algebra) or with ordinary arithmetic and have the virtue of forcing you to think about how to handle missing values. They are especially useful for summarizing multilevel data and for characterizing a complex set of logical conditions. Once you get comfortable with the basic uses of binary flags, it is easy to see many uses for them. The one thing to remember: *BE CAREFUL ABOUT MISSING VALUES*.

CONTACT INFORMATION

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

David J. Pasta
VP, Statistics & Data Operations
ICON Clinical Research
188 Embarcadero, Suite 200
San Francisco, CA 94105
(415) 371-2111
dpasta@ovation.org
www.iconclinical.com

Eric Elkin
Senior Research Manager
ICON Clinical Research
188 Embarcadero, Suite 200
San Francisco CA 94105
(415) 371-2153
eelkin@ovation.org
www.iconclinical.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.