Chapter 11
Treatment of Missing Values

11.1 Introduction

Missing values exist in abundance in databases, and procedures for treating them are by far the most recurring theme in data mining modeling. Therefore, I allowed this chapter to be the longest in the book!

One may ask two questions regarding missing values: (1) What do they mean, and (2) where do they come from? To answer these questions, we have to trace the origins of missing values. A missing data may represent, or be a product of, any of the following.

- An unknown value. For example, during surveys, responders may not answer certain questions.
- A result of the design of the operational system, such as the activation date of a credit card. Before the card is activated, the activation date is `NULL`—a missing value.
- An error while transforming data from different platforms and systems. For example, while porting data from a flat file to a relational database, we may attempt to feed a numeric field in a table with string values. In most cases, this raises an error and a missing value is generated.
- An undefined value resulting from mathematical operations either on other missing values or as a result of prohibited mathematical operations, such as division by zero or taking the log of a nonpositive number. The result of such ill-defined operations depends strongly on the data processing engine. For example, in SAS operations on missing values return missing values, in most cases without an error message (only a note in the SAS Log).

The first two cases in the list represent situations where the missing value has some specific business meanings, and the last two represent suppressed or ignored errors.

The problem is that the analyst is usually confronted with the mining view as an extract from the database and may not know the reason for the missing values. It is, therefore, advisable to attempt to know, as much as technically and economically feasible, the source of missing values before embarking on replacing them or dealing with them using one of the techniques described in this chapter.

The next important questions are: Do we need to do anything? Can we use the variables with the missing values? In other words, when should we resort to replacing missing values with something else *at all?* To answer this question, we need to know the meaning of the missing values, that is, are they errors that need correction or do they have a business meaning? If they have a meaning, then we have to make sure that any scheme used to replace these values will retain or properly represent this meaning. In view of this, the most difficult situation is when some missing values of a certain variable have a meaning while some are just errors that need correction.

The last factor to consider when attempting to treat missing values is to examine the specific requirements of the data mining technique to be used with the data, as well as the implementation software. Some data mining algorithms tolerate missing values and can treat them as a separate category, such as decision trees. Other methods need to have all the values represented by numeric values, such as neural networks, regression models, and clustering algorithms. In addition, some software implementations allow the user to treat missing values at the time of modeling, thus making the task of explicitly replacing or imputing them seem redundant or unnecessary. However, explicit treatment of missing values provides the analyst with better control over the results and ensures that the same procedures and values used in modeling are applied at the time of scoring. In addition, using cleaner variables in the final mining view with any modeling package allows the analyst to focus on taking advantage of their features in the modeling phase by solving the data problems beforehand.

The following are three basic strategies to treat missing values.

1. *Ignore the Record*

   In this case, we elect to remove the record that contains the missing value from the mining view. The advantage of this option is that we will use a record only when it is complete and therefore the model will be based on actual data and not on guessed values. There are two disadvantages to this method. First, most real datasets are so sparse that only a relatively small proportion of records is free of missing values. This small proportion may not be adequate to describe the behavior the model is supposed to capture and represent. The second reason is that the scoring data will also contain missing values. Therefore, if the model is based on the assumption that the values of all variables are known, those records with missing values will not be scored correctly, or perhaps not scored at all.

   Hence, this approach is recommended when the proportion of the data containing missing values is relatively small in both the training and the scoring datasets. It can be effectively used, however, in cases where the data collection procedures should not allow missing values as an option. For example, in scientific and engineering applications of data mining, such as weather prediction and industrial process control, the data is gathered from measurement equipment that

does not normally result in missing values. When records do contain missing values, it is an indication that something went wrong with the data collection equipment or software, and these records should not be used at all.

2. *Substitute a Value*

In this strategy, we substitute missing values with some value of our choosing. This value could be based on the general characteristics of the variable, such as the mode or the mean, or on a user-defined value, such as zero or any given value. In this scheme, the following options are typically used.

- Nominal variables
  - The mode: most occurring value
  - Any of the categories (user-defined value)
  - A new category to indicate a missing value
- Ordinal and continuous variables
  - The mean (only continuous)
  - The median
  - The mode
  - Maximum value
  - Minimum value
  - Zero or any other user-defined value

In all these options, we must be aware that we are creating a *bias* in the data toward the selected value (i.e., mode, mean, etc). In the case of continuous variables, using the median usually creates less bias than does using the mean.

One advantage of this approach is that the implementation of the substitution routines is simple and straightforward. Also, the justification for selecting the specific replacement option is often easy. For example, in the case of nominal variables, the mode represents the most probable value.

We will provide two macros, one for nominal variables and another for continuous variables, for the replacement of missing values with the preceding options.

3. *Impute the Values*

In this case, we attempt to *re-create* the data and simulate the process of data generation by fitting the data to some model and using the results of that model to predict the values that appeared as missing. Therefore, we use the nonmissing values of some variables to predict the missing values in other variables. The Multiple Imputation (MI) procedure of SAS/STAT implements several methods to impute missing values for continuous, ordinal, and nominal variables. Imputation algorithms depend on the *pattern* of missing values. We will elaborate more on this subject shortly and show how to identify what we call the *missingness* pattern and how to select the appropriate algorithm. Because `PROC MI` contains all the procedures needed for imputing missing values, our exposition focuses on understanding the background of the subject matter, writing macros that allow

us to use PROC MI properly by identifying the missingness pattern, and wrapping up the procedure with an easy-to-use macro.

11.2 Simple Replacement

11.2.1 Nominal Variables

We will assume that nominal variables are defined in SAS datasets as string variables. Accordingly, the following macro calculates the nonmissing value of the mode of a string variable in a dataset. We exclude the possibility of the mode being the missing value because we plan to use this macro to substitute for the missing values themselves. The macro relies on PROC FREQ to calculate the frequencies of the categories of the variable XVar in the dataset DSin and selects the largest value as the output M_Mode.

```
%macro ModeCat(DSin, Xvar, M_Mode);
/* Finding the mode of a string variable in a dataset */
%local _mode;
proc freq data=&DSin noprint order=freq;
 tables &Xvar/out=Temp_Freqs;
run;
/* Remove the missing category if found */
data Temp_freqs;
 set Temp_freqs;
  if &Xvar='' then delete;
run;
/* Set the value of the macro variable _mode */
data Temp_Freqs;
 set Temp_Freqs;
if _N_=1 then call symput('_mode',trim(&xvar));
run;

  /* Set the output variable M_mode and clean the workspace */
%let &M_Mode=&_mode;

proc datasets library=work nodetails nolist;
 delete Temp_Freqs;
quit;

%mend;
```

The next macro provides three methods for the substitution of missing values in a string variable: (1) the mode, (2) a user-defined value, or (3) deletion of the record. To calculate the mode, we use the macro ModCat().

```
%macro SubCat(DSin, Xvar, Method, Value, DSout);
/*
Substitution of missing values in a nominal (String) variable
DSin:   input dataset
Xvar:   the string variable
Method: Method to be used:
1=Substitute mode
2=Substitute Value
3=Delete the record
Value:  Used with Method
DSout:  output dataset with the variable Xvar free of missing
        values
*/
/* Option 1: Substitute the Mode */
%if &Method=1 %then %do;
  /* calculate the mode using macro ModeCat */
       %let mode=;
%ModeCat(&DSin, &Xvar, Mode);
 /* substitute the mode whenever Xvar=missing */
 Data &DSout;
  Set &DSin;
   if &Xvar='' Then &Xvar="&mode";
  run;
    %end;
/* Option 2: Substitute a user-defined value */
%else %if &Method=2 %then %do;
 /* substitute the Value whenever Xvar=missing */
 Data &DSout;
  Set &DSin;
   if &Xvar='' Then &Xvar="&Value";
  run;
     %end;

/* Option 3: (anything else) delete the record */
%else %do;
 /* Delete record whenever Xvar=missing */
 Data &DSout;
  Set &DSin;
   if &Xvar='' Then delete;
  run;
   %end;
%mend;
```

The following sample code shows how to use the three methods.

```
data WithMiss;
length Gender $6.;
input ID Age Gender $ @@;
datalines;
```

```
 1   . Male      2  28 Male
 3  35 Male      4  27 Male
 5  41 Male      6  21 Female
 7  30 .         8   . Male
 9  28 Female   10  32 Male
11  38 Female   12  42 .
13  37 .        14  28 Female
15  38 Male     16  18 Male
17  46 Female   18   . Female
19  32 Male     20  27 Female
;
run;

/* Substitute the mode */
%let Dsin=WithMiss;
%let Xvar=Gender;
%let Method=1;
%let Value=;
%let DSout=Full_Mode;
%SubCat(&DSin, &Xvar, &Method, &Value, &DSout);

/* Substitute the value "Miss" */
%let Dsin=WithMiss;
%let Xvar=Gender;
%let Method=2;
%let Value=Miss;
%let DSout=Full_Value;
%SubCat(&DSin, &Xvar, &Method, &Value, &DSout);

/* Delete the record */
%let Dsin=WithMiss;
%let Xvar=Gender;
%let Method=3;
%let Value=;
%let DSout=Full_Del;
%SubCat(&DSin, &Xvar, &Method, &Value, &DSout);
```

11.2.2 Continuous and Ordinal Variables

In the case of continuous variables, the options for substituting missing values are more than those for nominal variables. In this case the following are the options.

- The mean (only continuous)
- The median
- The mode
- Maximum value

- Minimum value
- Zero or any other user-defined value

Section 6.5 presented macro VarUnivar1(), which calculates all the univariate statistics of a continuous variable in a dataset. We will use this macro to calculate these statistics, and then use them to substitute for the missing values. In addition to the list of possible values, the macro allows the use of the values of the standard deviation, P1, P5, P10, P90, P95, P99, and it allows deletion of the record. All these options are accessible by defining the value of the parameter Method.

```
%macro SubCont(DSin, Xvar, Method, Value, DSout);
/* Calculate the univariate measures */
 %VarUnivar1(&DSin, &Xvar, Temp_univ);
/* Convert them into macro variables */
data _null_;
 set Temp_univ;
    Call symput('Mean'  ,Vmean);
Call symput('min'   ,VMin);
Call symput('max'   ,VMax);
Call symput('STD'   ,VStd);
Call symput('mode'  ,Vmode);
Call symput('median',Vmedian);
Call symput('P1'    ,VP1);
Call symput('P5'    ,VP5);
Call symput('P10'   ,VP10);
Call symput('P90'   ,VP90);
Call symput('P95'   ,VP95);
Call symput('P99'   ,VP99);
run;

/* Substitute the appropriate value using the
   specified option in the parameter 'Method' */
Data &DSout;
 set &DSin;
%if %upcase(&Method)=DELETE %then %do;
 if &Xvar=. then Delete;
  %end;
%else %do;
 if &Xvar=. then &Xvar=
   %if %upcase(&Method)=MEAN %then &mean;
   %if %upcase(&Method)=MIN %then &min;
   %if %upcase(&Method)=MAX %then &max;
   %if %upcase(&Method)=STD %then &std;
   %if %upcase(&Method)=MODE %then &mode;
   %if %upcase(&Method)=MEDIAN %then &median;
   %if %upcase(&Method)=P1 %then &p1;
   %if %upcase(&Method)=P5 %then &P5;
   %if %upcase(&Method)=P10 %then &P10;
```

```
   %if %upcase(&Method)=P90 %then &P90;
   %if %upcase(&Method)=P95 %then &P95;
   %if %upcase(&Method)=P99 %then &P99;
   %if %upcase(&Method)=VALUE %then &Value;
%end;
run;

/* Finally, clean the workspace */
proc datasets library=work nolist nodetails;
 delete temp_univ;
run; quit;
%mend;
```

The following code demonstrates the use of the macro with few options using the last dataset, `WithMiss`, for the continuous variable `Age`.

```
/* Use the mean */
%let DSin=WithMiss;
%let Xvar=Age;
%let Method=mean;
%let value=;
%let DSout=Full_mean;
%SubCont(&DSin, &Xvar, &Method,&value, &DSout);

/* Use the median */
%let DSin=WithMiss;
%let Xvar=Age;
%let Method=median;
%let value=;
%let DSout=Full_median;
%SubCont(&DSin, &Xvar, &Method,&value, &DSout);

/* Use a user-defined value */
%let DSin=WithMiss;
%let Xvar=Age;
%let Method=value;
%let value=25.8;
%let DSout=Full_value;
%SubCont(&DSin, &Xvar, &Method,&value, &DSout);

/* Delete the record */
%let DSin=WithMiss;
%let Xvar=Age;
%let Method=delete;
%let value=-560;
%let DSout=Full_del;
%SubCont(&DSin, &Xvar, &Method,&value, &DSout);
```

11.3 Imputing Missing Values

We start this section by examining the most used method for data imputation: the regression method. Assume that we have five continuous variables: x_1, x_2, x_3, x_4, and x_5. Assume further that the variables x_1, x_2, x_3, and x_4 are all made of complete observations, that is, they do not contain missing values. Only variable x_5 contains some missing values. Additionally, we have business reasons to believe that there is a correlation between the variable x_5 and all the other variables x_1, x_2, x_3, x_4. An obvious way of inferring the missing values is to fit a regression model using x_5 as a dependent variable and the variables x_1, x_2, x_3, x_4 as independent variables, using all the nonmissing observations of the dataset. We then apply this model to the observations where x_5 is missing and predict those missing values. This process will yield one fitted value for each observation where x_5 is missing.

11.3.1 Basic Issues in Multiple Imputation

Multiple imputation methods add a twist on the preceding procedure. Instead of fitting only one regression model, we could generate several models by allowing the model parameters themselves to follow some probability distribution and add some random error to the fitted value. In this way, we *re-create* the original distribution from which the values of the variable x_5 were drawn. Therefore, we end up with *multiple imputed* values for each missing value of x_5. The number of these imputed values need not be large. In fact, it can be shown that only a small number of imputed values can lead to accurate results (Rubin 1987). In practice the number of imputed values is taken between 3 and 10.

Multiple imputation theory also provides the methods to perform statistical inference regarding the imputed variable, x_5 in our example, by combining the imputed values with the original nonmissing records.

The main features of multiple imputation can be summarized as follows.

- The imputation process is divided into three steps: (1) fitting a set of models that represent the nonmissing values of the variable in question, (2) using these models to impute several values for each missing value, and (3) combining these values to produce statistical inferences about the imputed variables.
- The final result of the imputation procedure is not one *best* or *better* estimate of the missing value, but rather a set of values representing the distribution from which this value was drawn.
- The focus of the procedure is to provide better estimates of the statistical properties of the variables with missing values.

The last feature strikes at the heart of the problem.

In data mining, we have two distinct procedures: modeling and scoring. During both of these procedures, we encounter missing values that we need to replace with

something. Although the procedures MI and MIANALYZE of SAT/STAT do a good job of fitting the imputation models and combining them, the multiple imputation procedure just described leaves us with the following tasks and questions.

- We need to devise a method to combine the imputed values into *one appropriate* value to use.
- If we fit an imputation model using the modeling data, how do we replicate that model for the scoring data? Do we use the same fitted imputation models to score the scoring data and produce the imputed values, and then combine them? Or do we use the scoring data to fit new models, and use those instead to calculate the imputed values? And if the latter is the choice, does that mean that the relationship among variables in the scoring data is different from that among variables in the modeling data? Is that allowed?
- Some imputation models require the data to have a certain distribution of their missing values, their *missingness pattern*. How do we guarantee that this pattern will be the same in both the modeling and scoring datasets?

Now that we have asked the difficult questions, let us address the easy ones: What models are there in PROC MI? What are their assumptions? How do we make the best of them? Let us start with some background about the patterns of missing values, or *missingness patterns*.

11.3.2 Patterns of Missingness

There are three types of missing data patterns: (1) monotone missing data pattern, (2) non-monotone missing data pattern, and (3) arbitrary missing data pattern.

Consider a set of variables, V_1, V_2, . . ., V_n, with some missing values across the different records. If we consider the variables in this strict order (V_1, . . ., V_n), then these variables are said to have a *monotone missing data pattern* if, in all records where the value of the variable V_i is missing, the values of the variables V_j, $j > i$ are all missing. If this condition does not apply, the pattern of missing data is called *non-monotone.* Finally, if we do not observe or we ignore the pattern of the missing data, we say that the dataset has an *arbitrary* missing data pattern. Note that the term *data with an arbitrary missing pattern* refers to a position taken by the analyst and not an actual state of the data.

Let us demonstrate these definitions using an example of the five variables $x_1, \cdots, x_5$. The values of these five variables are shown in Table 11.1, which shows that the fourth record contains missing values for variables x_1, x_2, and x_3 but nonmissing values for both x_4 and x_5. Therefore, this dataset has a non-monotone missing data pattern. However, we can reorder the variables, as shown in Table 11.2, where the data has a monotone missing pattern.

When working with PROC MI of SAS/STAT, one has more flexibility in selecting the imputation method. Therefore, starting with a dataset with a non-monotone missing pattern, it is desirable to attempt to convert it to have a monotone pattern. This may

Table 11.1 Five variables with missing values.

Record	x_1	x_2	x_3	x_4	x_5
1	.	.	.	.	.
2	13	10	18	12	13
3	18	16	12	12	11
4	.	.	.	12	4
5	18	7	5	16	11
6	1	17	6	19	4
7	.	.	.	.	7
8	.	8	.	13	11
9	14	8	.	16	18
10	.	.	.	9	7

Table 11.2 Reordered five variables with missing values.

Record	x_5	x_4	x_2	x_1	x_3
1	.	.	.	.	.
2	13	12	10	13	18
3	11	12	16	18	12
4	4	12	.	.	.
5	11	16	7	18	5
6	4	19	17	1	6
7	7	.	.	.	.
8	11	13	8	.	.
9	18	16	8	14	.
10	7	9	.	.	.

be possible by simply reordering the variables, but this is not always guaranteed to succeed. In this case, we can impute just enough values to make the missing pattern monotone, and then apply one of the methods of monotone pattern. We will adopt this two-stage methodology to impute missing values.

11.4 Imputation Methods and Strategy

We now take a look at the imputation methods available in `PROC MI` of SAS/STAT (version 9.1). The methods that `PROC MI` offers depend on the type of the imputed

variable as well as on the pattern of missing values. Table 11.3 summarizes these methods.

Table 11.3 shows that when the missing pattern is monotone, `PROC MI` offers more flexibility in the choice of algorithm to use, especially for imputing continuous variables. We do not go into the details of the theory behind different methods because the SAS online documentation covers them in full. Additional details on algorithms of multiple imputation can also be found in Rubin (1987) and Schafer (1997).

Table 11.3 Imputation methods available in `PROC MI`.

Pattern of missingness	*Type of imputed variable*	*Recommended methods*
Monotone	Continuous	Regression Predicted mean matching Propensity score
Monotone	Ordinal	Logistic regression
Monotone	Nominal	Discriminant function method
Arbitrary	Continuous	MCMC full-data imputation MCMC monotone-data imputation

Before presenting our imputation scheme, we make the following comments:

- The MCMC (Markov Chain Monte Carlo) method can impute missing values for continuous variables to make the pattern monotone. There is no similar option for either nominal or ordinal variables at the moment in `PROC MI`. Therefore, when the pattern of missing data in a set of nominal variables is not monotone, we can only attempt to either reorder the variables, in hopes that this would reveal that they are monotone, or replace some values with known values (such as the mode) and test the missingness pattern again.
- The default method for imputing a continuous variable is regression.
- When imputing continuous variables, we can specify several interaction or nonlinear terms to be used during the imputation process. This is a big advantage that often leads to better results. Although the exact nonlinear form that will lead to such improvement is not known in advance, typical interaction and second-order terms of continuous variables usually lead to some improvement.

Now, our general imputation strategy is to attempt to make the missing pattern monotone by rearranging the variables. If that does not work, we resort to the MCMC method to impute enough observations to make the pattern monotone. But, as shown in Table 11.3, this option is available only for continuous variables. Therefore, when the rearranged variables contain ordinal or nominal variables and the missing pattern is not monotone, we have to deal with ordinal and nominal variables separately. We can summarize this process in the following steps.

1. Given the list of variables that are believed to be related to each other and that contain missing values, we first identify the pattern of missingness with the current order of variables.
2. If the missingness pattern is not monotone, we attempt to arrange the variables to make it monotone.
3. If the rearranged variables still do not have a monotone missing pattern, we separate the continuous variables from the others and use MCMC to make them follow a monotone pattern, and then we use one of the methods of continuous variable imputation to impute *all* the continuous variables and combine the imputed variables to form *one* set with *nonmissing* continuous variables.
4. Now that all the continuous variables have nonmissing values, we add the nominal and ordinal variables and check again whether we could arrange the variables to form a set with a monotone missing pattern.
5. If the resulting set still does not have a monotone pattern, we treat each group of nominal or ordinal variables using a separate imputation model.

Figure 11.1 shows schematically these steps as a comprehensive process that could be adopted to impute the missing values.

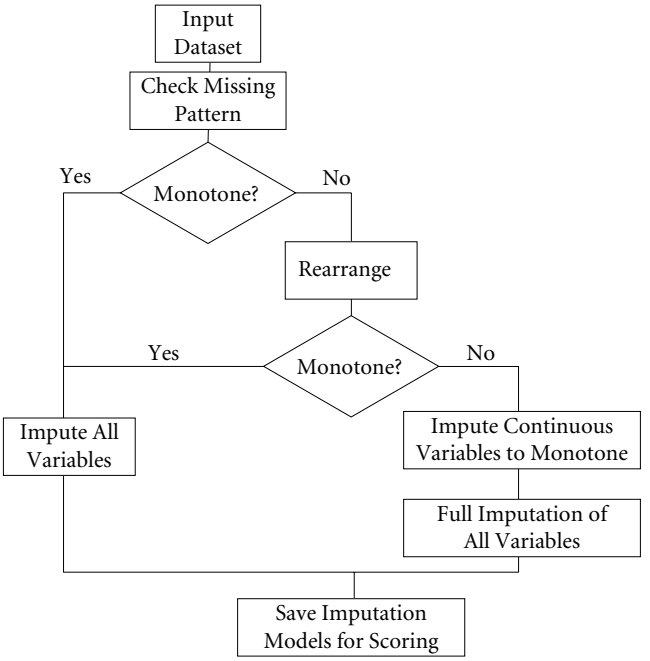

Figure 11.1 Multiple imputation strategy.

Implementing this process with a large number of variables is not a trivial task. However, in most practical cases with large number of variables, we are usually able to divide the variables into a number of smaller groups of variables. Each group usually contains 5 to 10 variables. The groups are usually formed by considering the meaning of the variables and the likely interdependence or correlation between them. In cases where we do not have enough insight into the problem domain, we may resort to calculating the correlation matrix among all the variables and using its entries to divide the variables into smaller sets of correlated variables. The application of the preceding process to each of these groups then becomes a much easier task.

The last issue we need to consider in planning our imputation strategy is how to deal with the fact that we are dealing with not one but at least two datasets: the modeling and scoring views. This setting poses an important question: If we impute the values of the modeling view, how do we use the same set of models to impute similar values for the scoring view?

To address this issue, we have to make the following assumption: The missing data pattern observed in the modeling dataset is identical to that in the scoring dataset.

Using this assumption, we have all the freedom to find the most suitable arrangement to impute the missing values in the modeling dataset. We then translate that into a simple model to use for replacing the missing values in the scoring dataset. This approach is very practical because most real-life scoring datasets have a large number of records; therefore, they require simple and efficient algorithms. We will use a logistic regression model to impute values of nominal and ordinal variables and a linear regression model to impute values of continuous variables.

Considering the previously listed steps of implementing multiple imputation, we note that we need macros to achieve the following tasks.

1. A macro to extract the pattern of missing values given a dataset and a list of variables in a certain order.
2. A macro to rearrange a list of variables such that their pattern of missing values becomes either monotone or as close as possible to monotone pattern. The inputs to this macro will be the initial order of these variables and their initial missing data pattern.
3. A macro to check that a given pattern of missing values is monotone.
4. A macro to apply the MCMC method to a set of continuous variables to impute just enough values to give them a monotone missing pattern.
5. A macro to impute a set of continuous variables with a monotone missing pattern.
6. A macro to combine the results of imputation of a set of continuous variables into one complete dataset.
7. A macro to apply the logistic regression method to impute one nominal or ordinal variable using a set of nonmissing variables (of any type).
8. A macro to combine the imputed values of ordinal or nominal variables.

9. A macro to use the results of imputation of a modeling view to extract a model that can be used to substitute missing values for a scoring dataset. This model will be a simple linear regression model for continuous variables and a logistic regression model for ordinal and nominal variables.

11.5 SAS Macros for Multiple Imputation

11.5.1 Extracting the Pattern of Missing Values

The first macro we present extracts the pattern of missing values of a set of variables in a dataset. The variables are given as an ordered list. We should note that `PROC MI` extracts such a pattern, but it does not provide an easy way to output that pattern into a dataset for further processing.

The macro works by counting the number of variables in the list and generating all missing patterns that could be present in a list of such length (see Table 11.4). It then tests the presence of all of these patterns and eliminates all but those that prove to be present in the data. The number of such combinations increases rapidly with the number of variables in the list. The exact number of these combinations is 2^n where n is the number of variables in the list. For example, if we have a list of 10 variables, we would be testing 1028 patterns. However, the implementation is a simple set of `IF-THEN` rules, which makes it fast for most practical cases with 10 to 15 variables in one list.

Table 11.4 Parameters of macro `MissPatt()`.

Header	`MissPatt(DSin, VarList, MissPat);`
Parameter	*Description*
`DSin`	Input dataset
`VarList`	List of variables in the given order
`MissPat`	Output dataset with the pattern of missing values

Step 1

First count the variables and store their names in macro variables.

```
%let i=1;
%let condition = 0;
%do %until (&condition =1);
   %let Var&i=%scan(&VarList,&i);
   %if "&&Var&i" ="" %then %let condition =1;
   %else    %let i = %Eval(&i+1);
%end;
%let Nvars=%eval(&i-1);
```

Step 2

Read one observation at a time, obtain the variable types, and store them in the macro variables `T1, ... TNvars`.

```
Data _Null_;
 set &DSin (Obs=1);
 %do i=1 %to &Nvars;
  call symput("T&i", VType(&&Var&i));
 %end;
run;
```

Step 3

Now that we have `Nvars` variables, the number of possible patterns is 2^{Nvars}. Therefore, construct a dataset to contain these patterns. This dataset will be the output of the macro `MissPatt()`. Create it first using `PROC SQL`.

```
proc sql noprint;
 create table &MissPat ( %do i=1 %to &Nvars;
                           &&Var&i  num,
                         %end;
                         PatNo num);
quit;
```

Step 4

To facilitate finding the patterns, enumerate the 2^{Nvars} patterns such that they all fit into just one loop. First generate all possible patterns, store them in a dataset, use them to build testing conditions, and then search for these conditions in the input dataset.

To generate the patterns, observe that for the first variable, we need two rows: a row with all 1's and a second row with a 0 in the first variable. For the second variable, we need to copy the two rows of the first variable and replace the 1 for the second variable with 0. We then repeat this process for all the other variables. Therefore, for variable i, we insert $2^{(i-1)}$ rows and replace the i^{th} location with 0 for the variable i. In the implementaton, we use the variable `PatNo` as temporary storage for the variable index `i` for replacement.

```
proc SQL noprint;
/* first row, insert all X's and PatNo=0 */
 insert into &MissPat values (%do i=1 %to &Nvars;
                                 1,
                              %end;
                                 0);
quit;
```

```
/* then start the loop on the variables */
%do i=1 %to &Nvars;
  /* select all the rows to a temporary table,
  change Var&i to zero and set PatNo to i */
   data temp;
     set &MissPat;
     &&Var&i =0;
     PatNo=&i;
   run;

  /* Append &MissPat with the temp dataset */
   data &MissPat;
     set &MissPat temp;
    run;
%end;

/* finally, we renumber the patterns from 1 to 2^Nvars,
   and create a new field called Present */
data &MissPat;
 set &MissPat;
  PatNo= _N_;
  Present =0;
 run;
```

Step 5

Now confirm that these patterns really exist in the dataset `DSin`. Loop on these patterns, extract the equivalent condition of the missing pattern, and then mark that pattern as present in the data (in the `Present` field).

```
/* Number of patterns */
%let NPats=%sysevalf(2**&Nvars);

%do ipat=1 %to &Npats;
  /* extract the condition by extracting the values
     of the variables */
 proc sql noprint;
  select %do i=1 %to &Nvars;  &&Var&i , %end; PatNo
    into %do i=1 %to &Nvars; :V&i, %end; :P
          from &MissPat where PatNo=&ipat;
 quit;
```

Step 6

Compile the condition and shape it depending on the variable type `N` or `C`.

```
%let Cond=;
%do i=1 %to &Nvars;
 %if (&i>1) %then %let Cond=&Cond and;
   %if &&V&i=O %then %do;
    %if &&T&i=C %then   /* String C */
         %let Cond=&Cond &&Var&i = '';
    %if &&T&i=N %then   /* Numeric N */
         %let Cond=&Cond &&Var&i=. ;
                    %end;
 %else %do;
    %if &&T&i=C %then
        %let Cond=&Cond &&Var&i ne '';
    %if &&T&i=N %then
        %let Cond=&Cond &&Var&i ne . ;
       %end;
 %end;
```

Step 7

Now, read the records from the dataset and check whether the condition exists. If it exists in the dataset, mark the record by setting `Present=1`. This concludes the pattern loop.

```
%let Nc=O;
  data _null_;
    set &Dsin;
      IF &Cond then do;
         call symput('Nc',1);
       stop;
                  end;
  run;

    %if &Nc=1 %then %do;
    proc sql noprint;
       update &MissPat set Present=1 where PatNo=&ipat;
    quit;
                   %end;

%end;  /* End of the pattern loop */
```

Step 8

Now that all the `Present` flags have been updated, remove all patterns but those with `Present=1`, and drop all `PatNo` and `Present` variables from the dataset `MissPat`.

```
data &MissPat;
 set &MissPat;
 if Present =O then delete;
```

```
 drop Present PatNo;
run;
```

Step 9

Clean the workspace and finish the macro.

```
proc datasets library =work nodetails nolist;
 delete temp;
run; quit;

%mend;
```

The following code demonstrates the use of the macro `MissPatt()` in a simple case.

```
data test;
 input x1 x2 x3 x4$ x5;
 datalines;
 . 1  . Abc 1
 . 1  1 .   1
 1 .  1 B   .
 . 1  . C   1
 . 1  1 .   1
 1 .  1 D   1
 . 1  . E   1
 . 1  . .   1
 1 .  1 F   1
 . 1  . A   .
 . 1  1 .   1
 1 .  . A   1
 . 1  . A   1
 . 1  1 .   1
 1 .  . DD  .
;
run;

%let DSin=test;
%let VarList=x1 x2 x3 x4 x5;
%let MissPat=MP;
%MissPatt(&DSin, &VarList, &MissPat)

proc print data=mp;
run;
```

The SAS output window should display the contents of the dataset `MP`, as shown in Table 11.5, where each observation in the dataset now represents an existing missing data pattern. The table shows that this dataset has a non-monotone missing pattern.

Table 11.5 The dataset `MP` as calculated by macro `MissPatt()`.

Obs	*x*1	*x*2	*x*3	*x*4	*x*5
1	1	0	1	1	1
2	0	1	0	1	1
3	1	0	0	1	1
4	0	1	1	0	1
5	0	1	0	0	1
6	1	0	1	1	0
7	0	1	0	1	0
8	1	0	0	1	0

11.5.2 Reordering Variables

Given a list of variables in a certain order along with their missing patterns, this macro attempts to reorder these variables such that they have either a monotone missing pattern or as close to a monotone pattern as possible. This macro should be used before attempting to impute values to facilitate the use of the different methods for datasets with monotone missing patterns. It requires the invocation of the macro `MissPatt()` of Section 11.5.1 to calculate the missing pattern dataset.

The macro works by reordering the columns of the missing pattern dataset, and not the actual dataset, such that the resulting pattern is monotone. A monotone pattern will be achieved if the columns are such that their sum is in decreasing order from left to right. A true monotone pattern also has the sum of the rows in increasing order from top to bottom. And this is exactly the algorithm used in this macro (see Table 11.6). The macro, however, cannot guarantee that the resulting order would be a monotone missing pattern because such patterns may not exist.

Table 11.6 Parameters of macro `ReMissPat()`.

Header	`ReMissPat(VListIn, MissIn, MissOut, M_VListOut);`
Parameter	*Description*
`VListIn`	Input list of variables
`MissIn`	Input missing pattern dataset
`MissOut`	Output missing pattern dataset
`M_VListOut`	Output list of variables in the new order

Step 1
Start by extracting the variable names from the input variable list `VlistIn` and count their number.

```
%let i=1;
%let condition = 0;
%do %until (&condition =1);
  %let Var&i=%scan(&VListIn,&i);
  %if "&&Var&i" ="" %then %let condition =1;
  %else   %let i = %Eval(&i+1);
%end;
%let Nvars=%eval(&i-1);
```

Step 2

Now we have `Nvars` variables. Create the first sum of the missing pattern matrix in each record to use it later in sorting. Assume that the missing pattern dataset contains at least one variable. That simplifies the code and starts the summation loop from the index 2. Then use this summation variable to sort the rows in descending order.

```
/* create the summation variable _Total_V. */
data Temp_MP;
  set &MissIn;
   _Total_V=sum(&Var1
    %do i=2 %to &Nvars;
    , &&Var&i
    %end;);
run;
/* Sort using the row totals,
   and then drop the field _Total_V.*/
proc sort data=Temp_mp;
 by descending _total_v;
run;
Data Temp_MP;
 set Temp_MP;
 drop _Total_V;
run;
```

Step 3

Transpose the missing pattern to prepare for sorting using the column sum.

```
proc transpose data=Temp_mp out=Temp_mpt Prefix=P;
 run;
```

Step 4

Similar to the case of the rows, find the summation of the original columns and sort on that summation variable and then drop it.

```
data temp_mpt;
 set temp_mpt;
 _total_P=sum (P1 %do i=2 %to &Nvars; , p&i  %end; );
```

```
  _index=_N_;
run;

proc sort data=temp_mpt;
 by descending _total_p;
run;
data temp_MPT;
 set temP_MPT;
 drop _total_P;
run;
```

Step 5

In preparation for the final extraction of the new variable order, transpose the missing pattern to its original orientation again.

```
proc transpose data=temp_mpt out=temp_mptt prefix=v;
run;
```

Step 6

By now, the dataset `temp_mptt` should contain the closest pattern to a monotone as possible, with the last row containing the order of variables that leads to that pattern. Therefore, extract the new order from the variable names stored in the macro variables $v_1, \ldots, v_{Nvars}$ and compile the output list.

```
proc sql noprint;
 select v1
  %do i=1 %to &Nvars;
   , v&i
  %end;
  into    :P1
       %do i=1 %to &Nvars;
        , :P&i
       %end;
    from temp_mptt
      where _Name_ ='_index';
quit;
```

Step 7

In case some of the variables did not exist in the missing pattern, remove all rows containing missing values in the resulting new missing pattern. Also delete unnecessary rows that were used in the calculations.

```
data &MissOut;
  set temp_Mptt;
  if _Name_='_index' then delete;
/* delete missing rows */
  %do i=1 %to &Nvars;
```

```
    if v&i = . then delete;
  %end;
  drop _Name_;
run;
```

Step 8

We can now restore the names of the original variables and compile the output variable list in the new order.

```
data &MissOut (Rename =
               (%do i=1 %to &Nvars;
                %let j=&&P&i;
                V&i=&&Var&j
                %end;)
                        );
 set &Missout;
run;

%let ListOut=;
%do i=1 %to &Nvars;
  %let j=&&P&i;
  %let ListOut= &ListOut &&Var&j;
%end;
%let &M_VListOut=&ListOut;
```

Step 9

Clean the workspace and finish the macro.

```
proc datasets library=work nodetails nolist;
delete temp_mp temp_mpt temp_mptt;
quit;
%mend;
```

The following code continues from the sample code in Section 11.5.1 to demonstrate the use of the macro `ReMisspat`.

```
%let LOut=;
%let Lin= x1 x2 x3 x4 x5;
%let Missin=MP;
%let MissOut=Re_MP;

%ReMissPat(&Lin, &MissIn, &MissOut, Lout);
%put Output List = &Lout;
```

The `%put` statement at the end of the code will display the following line in the SAS Log:

```
Output List = x4 x5 x1 x3 x2
```

Table 11.7 The dataset `Re_MP` as calculated by macro `ReMissPat()`.

Obs	*x*4	*x*5	*x*1	*x*3	*x*2
1	1	1	1	1	0
2	1	1	0	0	1
3	1	1	1	0	0
4	0	1	0	1	1
5	1	0	1	1	0
6	0	1	0	0	1
7	1	0	0	0	1
8	1	0	1	0	0

In addition, the new dataset representing the pattern of missing values, `Re_MP`, will be as shown in Table 11.7. The table shows that these variables are now closer to having a monotone missing pattern, but the dataset could not be made completely monotone.

11.5.3 Checking Missing Pattern Status

This macro checks whether a given missing pattern is monotone or not. We achieve that by applying the definition of a monotone missing pattern. Recall that for a missing pattern to be monotone, we require that, in any given observation, if a variable in the variable list is missing, then all subsequent variables in that list are also missing. To prove that a pattern is monotone is not a simple task. But to prove that it is *not* is relatively easy. All we need to find is *one* record that violates the definition. This is how this macro works.

We check the rows of the missing pattern dataset and loop over the variables in order. If any variable has a missing value in that pattern, then we check if any of the subsequent variables is *not* missing. When this condition is satisfied, the missing pattern is non-monotone. If we cannot find violations to the definition, the dataset is monotone.

The following are the details of macro `Check Mono()` (see Table 11.8).

Table 11.8 Parameters of macro `CheckMono()`.

Header	`CheckMono(MissPat, VarList, M_Result);`
Parameter	*Description*
`MissPat`	Input missing pattern dataset
`VarList`	Input variable list
`M_Result`	Output result

Step 1
Load the variable names into macro variables, count them, and count the number of the missing patterns.

```
%let i=1;
%let condition = 0;
%do %until (&condition =1);
   %let Var&i=%scan(&VarList,&i);
   %if "&&Var&i" =""  %then %let condition =1;
   %else    %let i = %Eval(&i+1);
%end;
%let Nvars=%eval(&i-1);

/* add a pattern number variable
   and count the patterns. */
data temp_MP;
 set &MissPat;
 _PatNo=_N_;
 call symput ('Np', trim(_N_));
run;
```

Step 2
Assume that the pattern is monotone.

```
%let Mono=1;   /*  The default assumption */
```

Step 3
Loop over the patterns, extract the values of the pattern flags corresponding to each of the variables, and store them in macro variables.

```
/* Pattern loop */
%do ipat=1 %to &Np;
 proc sql noprint;
   select &Var1 %do i=2 %to &Nvars;
                , &&Var&i
                %end;
          INTO  :P1 %do i=2 %to &Nvars;
                        , :P&i
                    %end;
      FROM temp_MP where _PatNo=&ipat;
  quit;
```

Step 4
Check the assumption of monotone missingness for each pattern by finding the first variable that has a zero in that pattern. If a zero is found, then attempt to find a

subsequent variable with a one in the same pattern. If a one is found, then this pattern violates the monotone pattern assumption and we reset the status flag Mono.

```
/* Find the lowest index jmin
   with a zero in the pattern */
%let jmin=%eval(&Nvars+1);
%do i=1 %to &Nvars;
 %if (&&P&i = 0) %then %do;
     %if (&jmin > &i) %then %let jmin = &i;
                        %end;
%end;
/* jmin should be smaller than Nvars */
%if &jmin < &Nvars %then %do;
 /* Search for any value of 1 above this index */
  %do i=%eval(&jmin+1) %to &Nvars;
    %if (&&P&i =1) %then %let Mono=0; /* Violation */
  %end;
                    %end;
```

Step 5

Finish the pattern loop and store the result in the output macro parameter M_Result.

```
%end;/* end of pattern loop */
%let &M_Result=%trim(&mono);
```

Step 6

Clean the workspace and finish the macro.

```
proc datasets library=work nodetails nolist;
delete temp_mp;
quit;
%mend;
```

The following code shows how to use the macro CheckMono(). The output macro variable M_Result will return a value of one if the pattern is monotone and a value of zero otherwise. The code generates an artificial simple pattern for five variables.

```
data Miss;
input x1 x2 x3 x4 x5;
datalines;
0 0 0 0 0
1 1 1 1 0
1 1 1 1 1
1 0 0 0 0
1 1 0 0 0
1 1 1 0 0
```

```
;
run;

%let VarList=x1 x2 x3 x4 x5;
%let MissPat=Miss;
%let Result=;

%CheckMono(&MissPat, &VarList, Result);
%put &Result;
```

11.5.4 Imputing to a Monotone Missing Pattern

This macro wraps PROC MI to implement the MCMC method to impute just enough missing values for the input dataset to have a monotone missing pattern. It is valid only for the imputation of continuous variables. Before we proceed to the implementation of the macro, we discuss two key parameters of PROC MI, namely, the number of imputed values for each missing value and the random seed.

The number of imputations performed by PROC MI is determined by the parameter NIMPUTE. Its default value is 5. For most datasets, the number of imputations should be between 3 and 10. If we denote the value of the NIMPUTE parameter by m, and the number of records in the original dataset by N, then the resulting dataset with the imputed values will have mN records. In cases of very large datasets, we may be forced to use lower values of NIMPUTE (e.g., 3); with smaller datasets we could use larger values (e.g., 10). In the following macro, we use the default value of 5.

In addition, PROC MI uses a random seed to implement the random variants in the different algorithms. One should always set the SEED parameter to a fixed value (e.g., 1000) to guarantee the repeatability of the results.

```
%macro MakeMono(DSin, VarList, DSout);
proc mi data=&DSin nimpute=5 seed=1000 out=&DSout noprint;
 mcmc impute=monotone;
 var &VarList;
 run;
%mend;
```

The output of the macro MakeMono() is the output dataset DSout containing five new imputed values for each missing value such that the new pattern of missing values is monotone. A variable, called _Imputation_, is added to the dataset, indicating the imputation index (from 1 to 5 in this case). Further invocation of PROC MI should include a BY statement, as follows:

```
BY _Imputation_;
```

This will guarantee that PROC MI uses all the imputed values to estimate the new final imputed values for the variables included in the variable list VarList.

11.5.5 Imputing Continuous Variables

PROC MI offers the following four methods for imputing missing values of continuous variables.

1. Regression (monotone)
2. Predicted mean matching (monotone)
3. Propensity scores (monotone)
4. Markov Chain Monte Carlo (arbitrary)

The first three methods are available only for datasets with the monotone missing pattern, and the MCMC method is available for the arbitrary missing pattern. We have adopted the strategy of converting the missing pattern to a monotone pattern before the full imputation.

Implementing PROC MI to impute the missing values of a set of continuous variables is straightforward. Therefore, our macro is a wrapper of PROC MI. The only consideration is that, as discussed in Section 11.4, we need to keep in mind that we are not interested only in imputing the missing values (and combining them somehow to replace the missing values), but also in fitting a model that could be applied to the scoring data. The method that best meets these requirements is the regression method.

The regression method of PROC MI attempts to fit a linear model to predict the values of the variable in question in terms of the other variables with nonmissing values. Hence, it is subject to all the limitations of linear models and their assumptions. The linearity assumptions could be relaxed sometimes by the introduction of *interaction terms*. For example, if we attempt to fit the values of the variable y in terms of the variables x_1, x_2, x_3, x_4, the standard linear model would fit the following equation:

$$y = a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4. \tag{11.1}$$

This model may not represent all possible nonlinear relationships between the variable y and x_1, x_2, x_3, x_4. Therefore, we attempt to improve the model by fitting the values of y to the following more complex form:

$$\begin{aligned} y = {} & a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 \\ & + b_1x_1^2 + b_2x_2^2 + b_3x_3^2 + b_4x_4^2 \\ & + c_{12}x_1x_2 + c_{13}x_1x_3 + c_{14}x_1x_4 + c_{23}x_2x_3 + c_{24}x_2x_4 + c_{34}x_3x_4. \end{aligned} \tag{11.2}$$

In this last model, the equation of y includes all possible second-order terms of the variables x_1, x_2, x_3, x_4. The term *interaction terms* is usually reserved for the terms on the third line of Equation 11.2.

Chapter 9 introduced a macro for the automatic generation of such interaction terms and for calculating them as new variables in a dataset. PROC MI allows the specification of such interaction terms *during modeling*. This means that the variables are

not actually stored in a dataset, but generated on the fly while modeling. Although this approach saves the disk space used to store the additional variables in these datasets, we prefer to generate these interaction terms explicitly as new variables in an intermediate dataset and include them in the imputation model. This facilitates the imputation of missing values during scoring.

Adopting this approach for the generation of interaction terms makes the implementation of PROC MI a simple task. The following macro wraps PROC MI to impute a set of continuous variables with monotone missing pattern using the regression method. To use this macro for both cases of direct imputation, or imputation after making the dataset have a monotone missing pattern using partial imputation, implement a set of %if conditions. These conditions impute only one set of values using the (BY_Imputation_;) command in the case of a dataset that has been processed by macro MakeMono() of Section 11.5.4.

If the input variable IfMono is set to N, five imputed values will be generated for each missing value of the variable specified in the MonoCMD variable. The format of the MonoCMD command describes the MONOTONE statement of PROC MI, as in the following example:

```
MONOTONE REG (Y = X1 X2 X3 M_X1_X3);
```

In the preceding code, the variable Y will be imputed in terms of the variables X1, X2, X3, and M_X1_X3. The MonoCMD variable may contain many such commands separated by semicolons. For example, we can define the variable MonoCMD as follows:

```
%let MonoCMD= MONOTONE REG (Y1 = X1 X2 X3 M_X1_X3) %str(;)
MONOTONE REG (Y2 = X1 X2 X3 M_X1_X3) %str(;)
MONOTONE REG (Y3 = X1 X2 X3 M_X1_X3) %str(;) ;
```

The function %str() has been used to write a semicolon in the macro variable MonoCMD without terminating it prematurely. The command will generate three imputation models for the variables Y1, Y2, and Y3.

```
%macro ImpReg(DSin, VarList, MonoCMD, IFMono, NImp, DSout);
/*
 Imputing a set of variables using regression models.
 The variables are assumed to have a monotone missing pattern.
 The variable MonoCMD contains the details of the monotone
 command. The IFMono (Y/N) indicates whether the input
 dataset has been made monotone using the MCMC method before,
 and, therefore, contains the variable _Imputation_,
 which then should be used in the BY statement.
 Also, if the variable _Imputation_ exists, we impute
 only 1 value, and in this case the parameter Nimp is
 ignored (but must be specified in the macro call).
*/
 proc mi data=&DSin
        seed=1000  out=&DSout noprint
```

```
    nimpute=%if %upcase(&IfMono)=Y %then 1;
            %else &Nimp   ;
    ;
&MonoCMD;
var &VarList;
%If %upcase(&IfMono)=Y %then by _Imputation_;;
 run;
%mend;
```

As in previous `PROC MI` implementations, we specified the seed, using `SEED=1000`, to guarantee the repeatability of the results of different runs. The specification of the seed number is arbitrary; any integer value could be used.

11.5.6 Combining Imputed Values of Continuous Variables

Now that we have imputed a few values for each missing value, we need to combine them into one dataset to use in either modeling or scoring. In the case of continuous variables, we could use the average of the imputed values as the estimate of the missing value. The imputed values for each missing value are labeled using the automatic variable `_Imputation_`. The following macro performs this averaging for one continuous variable in a dataset.

In this macro, the variable `_Imputation_` is termed the *imputation index* variable because it defines the label for the imputation set. The macro requires the definition of a unique identification variable, `IDVar`. This should not be a problem because all datasets used in practical data mining applications have a unique identifier, as discussed in Section 1.3.

The macro `AvgImp()` works by rolling up the imputed values and finding the average value of the variable in question, as shown in Figure 11.2.

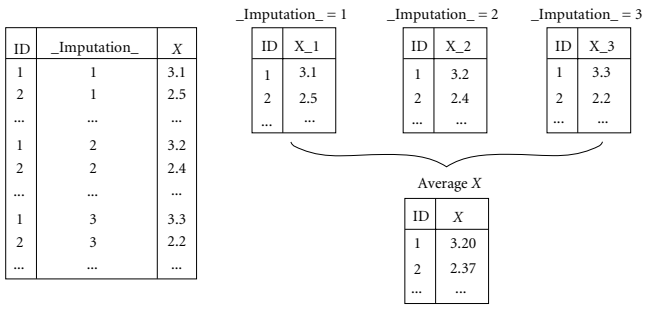

Figure 11.2 Combining imputed values for a continuous variable.

Table 11.9 Parameters of macro `AvgImp()`.

Header	`AvgImp(DSin, Nimp, IDVar, IDImp, Xvar, DSout)`
Parameter	*Description*
`DSin`	Input dataset
`Nimp`	Number of imputed values
`IDVar`	Unique ID variable
`IDImp`	Variable used to label imputations
`XVar`	Imputed variable
`DSout`	Output dataset with the average of imputed values

Step 1

Roll up the different imputed values into separate datasets named `Temp_1`, `Temp_2`, and so on. Each of these datasets contains the imputed values of the variable `XVar` in a new variable using a suitable subscript. Sort each dataset by the unique data identifier `IDVar`.

```
%do i=1 %to &nimp;
Data temp_&i;
  set &dsin;
  if &IDImp=&i;
  x_&i=&Xvar;
  keep &Xvar._&i &IDvar;
run;
   proc sort data=temp_&i;
   by &IDvar;
   run;
%end;
```

Step 2

Merge the datasets `Temp_1`, `Temp_2`, ... to a single dataset using the ID variable `IDVar`.

```
   data temp_all;
    merge %do i=1 %to &Nimp; temp_&i %end; ;
by &IDvar;
   run;
```

Step 3

The output dataset should contain the average of all the imputed values merged from the rollup datasets.

```
   Data &DSout;
    set temp_all;
```

```
  &Xvar =MEAN( x_1 %do i=2 %to &Nimp; , x_&i %end;) ;
 keep &IDvar &Xvar;
    run;
```

Step 4

Clean the workspace and finish the macro.

```
proc datasets library=work nolist nodetails;
delete temp_all %do i=1 %to &Nimp; temp_&i %end; ;
run; quit;
%mend;
```

The following code shows how to use the preceding macro. The dataset `Test` simulates the imputed values by creating the variable `_Imputation_` directly. As the macro syntax shows, we do not need to keep the default name of this variable, but it is recommended that we keep it unchanged because other SAS procedures, such as `PROC MIANALYZE`, use this variable for summary statistics.

```
data test;
 input _Imputation_  ID X;
 datalines;
1 1 2  1 2 3  1 3 5  1 4 7
2 1 3  2 2 5  2 3 5  2 4 7
3 1 4  3 2 9  3 3 5  3 4 7
;
run;
options mprint;
%let nimp=3;
%let dsin=test;
%let IDImp=_Imputation_;
%let IDVar=id;
%let Xvar=x;
%let DSout=out;
%AvgImp(&DSin, &Nimp, &IDvar, &IDImp, &xvar, &dsout);
```

Note on Using the Average

Could we have used a summary statistic other than the average—for example, the mode or the median? The answer comes from the theory of the imputation itself. The theory does not attempt to create *one* new value to replace the unknown missing value. It provides a set of values that represent the *distribution* from which the unknown value came. It is up to us to extract from this distribution what we need. Because we only need one value, we may use the average as the best unbiased estimate. In addition, in view of the small number of imputed values, fewer than 10 in general, there should not be a large difference between the average and the median. `PROC MIANALYZE` uses

all the imputed values to accurately calculate univariate statistics of the *entire* dataset, with the imputed values included.

11.5.7 Imputing Nominal and Ordinal Variables

Table 11.3 earlier in this chapter showed that PROC MI implements logistic regression for imputing ordinal variables. It also shows that imputing nominal variables could be achieved by the discriminant function method. In the case of binary nominal variables only, logistic regression could also be used for nominal variables. In both cases, PROC MI requires a CLASS statement with the variable(s) being modeled as nominal or ordinal variables. Also, in both cases, PROC MI requires a monotone missing values pattern.

It is known that logistic regression, as well as discriminant function, models perform better when all the predictors are continuous. In addition, our imputation strategy relies on imputing all the missing values of the continuous variables first. In view of this, we assume that all the predictors in the imputation of nominal and ordinal variables are continuous and have no missing values.

In this case, the imputation of ordinal and nominal variables is reduced to wrapping PROC MI, as in the following macro.

```
%macro NORDImp(DSin, Nimp, Method,CVar, VarList,DSout);
/* Imputing Nominal and Ordinal variables
 DSin:    input dataset.
 DSout:   output dataset.
 Nimp:    Number of imputations
 Method:  either Logistic or Discrim
 CVAr:    variable to be imputed. If CVar is ordinal,
          only logistic method could be used. If CVar is
          nominal and binary, both Logistic and Discrim
          methods could be used. If CVar is nominal and NOT
          binary, then only Discrim method could be used.
 VarList: list of NONMISSING variables to be used as
          predictors in the imputation models.
*/
proc mi data=&DSin out=&DSout nimpute=&Nimp seed=1000 noprint;
 class &CVar;
 monotone &Method (&Cvar);
 var &VarList &Cvar;
 run;
%mend;
```

11.5.8 Combining Imputed Values of Ordinal and Nominal Variables

Imputed values for *ordinal* variables could be combined in a similar fashion to that of continuous variables, that is, by taking averages. However, unlike continuous values,

ordinal variables, although preserving order relationship, admit a certain finite set of values. Therefore, the use of the average is inappropriate, because it will render new values that may not exist in the original categories. We could select an existing value that is nearest to the computed mean value or select the *median* and require that the number of imputed values be odd to guarantee that the median value will always be one of the original values. The macro `AvgImp()`, described in Section 11.5.6, could be modified by changing the function in step 3 from `MEAN` to `MEDIAN`.

The case of *nominal* variables is a bit more complex. The logical option is to substitute the *mode,* the most frequent value, from the imputed values for each missing value. Writing a macro to find the mode of a set of values is straightforward. (Review Section 5.6.) Unfortunately, the computational cost of implementing such a scheme is not acceptable in most cases.

For example, consider the case where we have a scoring dataset of one million records with one nominal variable of which only 5% of the values are missing (i.e., 50,000 missing values). And assume that we imputed five values for each missing value. Then we will attempt to invoke the mode finding macro 50,000 times. Therefore, we may use this macro to calculate the mode and use it to combine the imputed values for a nominal variable only when the number of missing values is small. To overcome this difficulty with nominal variables, we may pursue one of the following two alternative strategies.

- During imputation, decide to impute only one value and use it to substitute for the missing value.
- Avoid multiple imputation altogether. This approach is explained in the last section of this chapter.

Finally, we would like stress that combining the values of nominal variables is not a theoretical difficulty, but rather a computational limitation imposed by the special needs of data mining procedures for working with large scoring datasets. If the computational time is not an issue, or if the analyst has access to powerful computers, then computing the mode should not be a problem.

11.6 Predicting Missing Values

By *predicting* missing values, we mean that we develop a predictive model that describes the relationship between the variables that are correlated to the variable in question. For example, for each continuous variable with some missing values, we may develop a linear regression model in the other variables that are free of missing values. Similarly, we may develop a logistic regression for each binary nominal or ordinal variable in terms of a set of continuous variables free of missing values (or that have been completed through imputation).

The fundamental difference between this approach and that of multiple imputation is that by fitting a model, we do not attempt to generate the original distribution

from which the variable is assumed to be drawn. We rather fit *one* value based on the dependence between the variable and other variables in the dataset.

The most appealing advantage to this approach is the ease of implementation, especially at the time of scoring. Fitting a logistic regression model using the nonmissing training data, and then using it to substitute the missing values in both the training and scoring datasets, should be straightforward. What makes things even easier is that SAS implementation of both linear and logistic regression, `PROC REG` and `PROC LOGISTIC`, use only nonmissing observations by default.

Using `PROC REG` and `PROC LOGISTIC` is thoroughly documented in the SAS/STAT documentation and many textbooks on the subject. But for the purpose of completeness, we present the following example code.

We start with the dataset `WithMiss`.

```
Data WithMiss;
input x1 x2 x3 y$ @@;
datalines;
4   2   3   .   1   4   1   A
0   1   0   B   2   3   3   A
3   1   0   A   2   3   1   .
2   3   1   B   0   3   2   A
1   0   0   B   3   3   1   B
1   3   0   .   1   1   4   A
3   2   1   B   2   3   3   B
1   0   1   B   3   3   3   A
1   3   3   .   2   3   2   A
3   0   4   A   1   0   4   .
2   1   4   A   0   1   0   B
4   2   3   A   1   1   3   A
;
run;
```

We create a logistic regression model that fits the values of the variable `y` in terms of the nonmissing variables `x1`, `x2`, and `x3`.

```
proc logistic data=WithMiss outest=LogModel noprint;
model y (Event='A')= x1 x2 x3;
run;
```

Before we score the data, we label the missing observations with a new variable `Miss`. We also rename the variable `y` as `y1`.

```
data Modeling;
 set WithMiss;
 Y1=Y;
 if Y1 ='' then Miss=1;
 else Miss=0;
 drop Y;
 run;
```

PROC LOGISTIC offers the option of generating the scores from a saved model. But we will use PROC SCORE and substitute in the logit equation to explore the full details of the model. Therefore, we start by using PROC SCORE to substitute in the linear form needed for the logit function. It results in a new variable y.

```
proc score data=Modeling
           out=WithoutMiss1
           score=LogModel
           type=parms;
Var x1 x2 x3;
run;
```

Now we use the logit function formula to calculate the score of the variable. If the score is larger than 0.5, we score it A, otherwise B. We perform this scoring only for the missing observations marked with the variable Miss.

```
data WithoutMiss;
 set WithoutMiss1;
 if Miss=1 then do;
   PorbA=exp(-y)/(1+exp(-y));
    if ProbA>0.5 Then Y1='A';
else Y1='B';
end;
keep x1 x2 x3 Y1;
run;
```

The final dataset, WithoutMiss, should have all the missing values substituted by the appropriate score of the logistic regression model.

In the preceding code, we should keep the dataset LogModel to score the scoring dataset in a similar way and substitute for missing values of the variable y.