Chapter 10 Binning and Reduction of Cardinality

10.1 Introduction

Binning is the process of converting a continuous variable into a set of ranges and then treating these ranges as categories, with the option of imposing order on them. We say *option* because sometimes it may not be necessary, or even desirable, to keep the order relationship of the ranges. For example, if we bin the variable representing the age of a customer into ranges of 10 years (0–10, 11–20, 21–30, . . . , etc.), a further analysis of the business problem may reveal that customers in the ranges 0 to 10 and 81 to 90 have similar behavior with respect to a certain issue, such as the cost of health services, and may be combined into a single group. Therefore, keeping the strict order of the bins is not always necessary. Similarly, classical scorecards built using regression models, in credit risk and marketing applications, employ binning to convert continuous variables to a set of indicator variables. The use of these variables in a regression model results in easy-to-interpret and easy-to-use scorecards.

Cardinality reduction of nominal and ordinal variables is the process of combining two or more categories into one new category. Nominal variables with high cardinality are a definite source of problems. If we attempted to map all their values to indicator variables, we would end up with a large number of variables, many of them almost full of zeros. On the other hand, if we do not convert them to indicator variables and use them with algorithms that can tolerate this problem, such as decision tree algorithms, we run into the problem of overfitting the model. Therefore, whenever possible one must consider reducing the number of categories of such variables.

Binning of continuous variables and cardinality reduction of nominal variables are two common transformations used to achieve two objectives:

1. Reduce the complexity of independent, and possibly (but less frequently) dependent, variables. For example, in the case of a variable representing the

balance of a checking account, instead of having a very large number of distinct values, we may bin the balance into, say, ranges of $500.00.

2. Improve the predictive power of the variable. By careful binning and grouping of categories, we can increase the predictive power of a variable with respect to a dependent variable for both estimation and classification problems. This is done by the selection of the binning and grouping scheme that will increase a certain measure of association between the DV and the variable at hand.

Binning and cardinality reduction are very similar procedures. They differ only in the fact that in the case of binning the order of the values is taken into account.

Methods of binning and cardinality reduction range from simple ad hoc methods based on the understanding of the data and the experience of the analyst to more sophisticated methods relying on mathematical criteria to assess the goodness of the grouping scheme. This chapter presents some of these methods. We start with the methods of cardinality reduction, and we will show later that cardinality reduction is the general case of binning.

10.2 Cardinality Reduction

10.2.1 The Main Questions

The following are the three main questions that guide exposition of this subject.

1. What is the maximum acceptable number of categories?
2. How do we reduce the number of categories?
3. What are the potential issues at deployment time?

The answer to the first question is again based only on experience and can be the subject of debate. It also depends on the nature of the data and the meaning of the categories themselves.

For example, when considering the categories of a variable that represent marital status, it may be logical to group the category of `Single` together with those of `Divorced` and `Separated`. This is based on the understanding of the meaning of the values and assigning new groups on that basis. A general rule of thumb is that a nominal variable with more than about a *dozen* categories should be considered for cardinality reduction.

As for the second question, there are two main strategies for reducing the number of categories. The first approach, which is the easiest and most tempting, is to ignore some of the categories with small frequencies. However, this simplistic approach does not take into account the multivariate nature of all real-life datasets; that is, we are *not* dealing only with *this* variable, but rather with a large number of variables with interaction relationships. Therefore, the simple removal of some records may make it easier to handle this particular variable, but it will definitely affect the distribution of the other variables too, with unknown results.

The second approach is to devise some scheme to group the categories into a smaller set of new *super* categories. The grouping, or cardinality reduction, scheme could be based on investigating the contents of each category and grouping them on the basis of their meaning with respect to the current problem being modeled. This is the best and most recommended option. The SAS implementation of these grouping schemes is straightforward through the use of IF–THEN–ELSE statements, as illustrated using the following example.

Example 10.1 Consider the following dataset with the two variables Age and Marital_Status.

```
DATA Test;
 FORMAT Marital_Status $10.;
 INPUT ID Age Marital_Status $ @@;

 CARDS;
   1 24 Unknown     2 29 Single
   3 24 Divorced    4 29 Divorced
   5 30 Single      6 30 Separated
   7 39 Unknown     8 35 Divorced
   9 42 Separated  10 41 Separated
  11 44 Separated  12 47 Unknown
  13 45 Divorced   14 47 Married
  15 48 Widow      16 49 Married
  17 51 Single     18 54 Single
  19 53 Divorced   20 55 Single
  21 55 Married    22 56 Divorced
  23 56 Single     24 60 Widow
  25 59 Married    26 62 Widow
  27 61 Widow      28 68 Unknown
  29 66 Married    30 69 Married
;
run;
```

We now apply a set of ad hoc rules to reduce the cardinality of the variable Marital_Status to two categories, A and B, as follows:

```
Data Grouped;
  set Test;
   if   Marital_Status="Separated"
     OR Marital_Status="Widow"
     OR Marital_Status="Unknown" THEN Marital_Status="A";
   if   Marital_Status="Single"
     OR Marital_Status="Divorced"
     OR Marital_Status="Married" THEN Marital_Status="B";
run;
PROC FREQ DATA=Grouped;
  tables Marital_Status;
run;
```

The use of `PROC FREQ` should confirm that there are only two categories. The purpose of this almost trivial example is to show how programming this type of transformation is easy and problem dependent.
◆

However, it is not always possible to do that, especially when the number of categories is large, say 50, and there is no meaningful way of combining them. In this case, we have to combine the categories such that we increase the overall contribution of the variable in question to the model. This approach is the basis of decision tree models. In this case, the tree splits are designed such that different categories are grouped in order to increase the predictability of the dependent variable in terms of the current independent variable.

Methods based on decision trees splits in cardinality reduction are called *structured grouping methods*. We devote the next section to these methods and their implementation in SAS.

The last question with regard to cardianlity reduction is related to deployment procedures. Reducing the cardinality of a nominal variable leads to the definition of a new variable with new categories based on the grouping of the original categories. The mapping between the old and the new categories should be done efficiently in anticipation of preparation of the scoring view. One of the efficient statements in SAS is the `IF-THEN` statement. Therefore, in our implementation, we will make sure that the final output of the cardinality reduction macros is a set of `IF-THEN` statements.

10.2.2 Structured Grouping Methods

These methods are based on using decision tree splits to find the optimal grouping of the categories of a nominal variable into a smaller number of categories. Note that these methods also work for continuous and ordinal variables, in which case we seek to reduce the variable into a smaller number of bins.

10.2.3 Splitting a Dataset

The idea of grouping some categories together relies on the concept of finding a dataset split into subgroups. Figure 10.1 shows such a split with a root node and three child nodes on a nominal variable X. In this split, the variable X has been grouped into three nodes containing categories A; B,C; and D,E. The splitting algorithm determines these groups such that the total information gained about a certain dependent variable Y is maximized after the split. The term *information* is used to denote some measure of the desired property of the dependent variable Y. If we use $I(\cdot)$ to denote such an information function, then we say that the required split is the one that maximizes the expression

$$\sum_{j=1}^{m} I(C_j) - I(P), \tag{10.1}$$

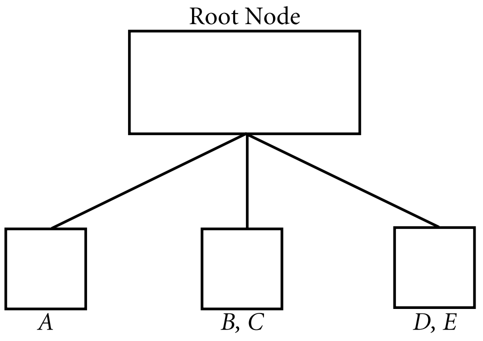

Figure 10.1 A dataset split.

where P denotes the parent node, and C_j, $j = 1, \cdots, m$ are the m child splits. By restricting the splits to binary, the right side of Expression 10.1 becomes

$$I(C_r) + I(C_l) - I(P), \tag{10.2}$$

where C_r and C_l are the left and right child splits, respectively. Although Expression 10.2 is useful, it is more appropriate to represent the power of a split using the percentage increase of the information. This is defined as

$$\hat{I} = [I(C_r) + I(C_l) - I(P)]\,/I(P). \tag{10.3}$$

Structured cardinality reduction algorithms are based on Equation 10.3. Various algorithms differ only in the definition of the information function $I(\cdot)$. The following are three common definitions of the information function.

- Entropy variance
- χ^2 test
- Gini measure of diversity

We will implement the algorithm that uses the Gini measure as an example.

10.2.4 The Main Algorithm

Figure 10.2 shows how the main algorithm works. The algorithm starts with a root node, which is split into two nodes: 1 and 2, as shown in Figure 10.2(a). Now that we have a tree to work with, we focus only on the terminal nodes, which are nodes 1 and 2 in this case. Then we attempt to split each of these terminal nodes, as in Figure 10.2(b). We use the appropriate definition of the information function to find the best split. In the case of the displayed tree, it was better to split node 1, as in Figure 10.2(c).

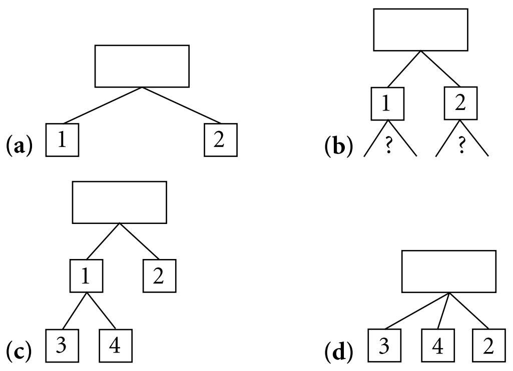

Figure 10.2 Splitting algorithm.

Now that we have a new list of terminal nodes (3, 4, 2), we can delete node 1 and replace it with its two children 3, 4, as shown in Figure 10.2(d). The procedure is then repeated until one of two conditions is satisfied: (1) The number of terminal nodes (new groups) reaches the required maximum number of categories, or (2) we cannot split any of the terminal nodes because the information function could not be increased by a significant amount. The limit on the significance of the information function and the maximum number of final groups are the two parameters that control the final result.

Note that the procedure is only a heuristic approach to solving the optimization problem at hand. It is not guaranteed to find the *best* grouping. However, it is guaranteed that the resulting grouping has a higher information content, with respect to the designated dependent variable, than the starting set of categories in the original dataset. This *theoretical* limitation is an unsolved issue in *standard* implementations of splitting algorithms. We say standard because some research has been done to find splitting algorithms that will find the global maximum information gain, but these methods are outside the scope of this book.

The algorithm just described can be formally stated as follows.

1. Start with an empty list of terminal nodes.
2. Find a split. This will generate right and left nodes R and L.
3. Store these nodes in the terminal node list.
4. The list now holds n terminal nodes.
5. If n is equal to the maximum required number of categories, stop.
6. Else, loop over *all* n nodes in the terminal node list.
7. Attempt to split each node and measure the information gained from that split.
8. Find the best split among the n candidate splits. Call this node C_x.

9. If the information gain is less than the minimum required gain for a split, then stop.
10. Else, split node C_x into right and left nodes R and L. Delete C_x from the terminal node list and add R and L.
11. Set $n = n + 1$.
12. Go to step 5.

The exact mathematical expression for the information function depends on three factors:

- The type of the information function, that is, Gini, Entropy or χ^2-based
- The nature of the independent variable being binned or reduced: nominal, ordinal, or continuous
- The nature of the dependent variable used to guide the splitting: nominal or continuous

10.2.5 Reduction of Cardinality Using Gini Measure

This section presents an algorithm for the reduction of cardinality of a nominal (independent) variable with the aid of another nominal dependent variable using the Gini diversity measure. As mentioned in Chapter 2, most data mining projects involve a classification task with the dependent variable being binary. In our implementation we represent the dependent variable using a strict binary representation, either 0 or 1.

Figure 10.3 shows the details of a split of a binary dependent variable into a total of *m* splits. In addition to Figure 10.3, we will adopt the notation shown in Table 10.1.

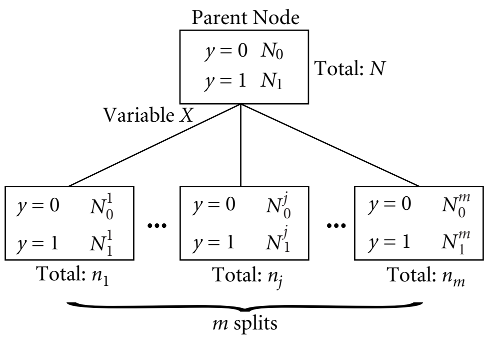

Figure 10.3 Details of binary DV splits.

Table 10.1 Notation of splits with binary DV.

Symbol	*Description*
y	Dependent variable (DV)
N	Total number of records in the dataset
X	The independent variable
N_1, N_0	Number of records for the categories 1 and 0 of y
N_1^j, N_0^j	Number of records in the j^{th} child node with $y = 1$ and $y = 0$, respectively
n_j	Total number of records in the j^{th} child node
m	Number of splits of the variable X

It is easy to confirm the following relationships:

$$n_j = N_1^j + N_0^j, \quad j = 1, \cdots, m. \tag{10.4}$$

$$N_k = \sum_{j=1}^{m} N_k^j, \quad k = 0, 1, \tag{10.5}$$

$$N = \sum_{k=0}^{1} N_k = \sum_{j=1}^{m} n_j. \tag{10.6}$$

We define the Gini measure of the parent node as follows:

$$G_p = 1 - \frac{\sum_{k=0}^{1} N_k^2}{N^2}. \tag{10.7}$$

We define the Gini measure for the jth child node as

$$G_j = 1 - \frac{\sum_{k=0}^{1} (N_k^j)^2}{(n_j)^2}. \tag{10.8}$$

The weighted Gini measure, $\hat{G}$, for the current split on X is then given by

$$\hat{G} = \sum_{j=1}^{m} \frac{n_j}{N} G_j. \tag{10.9}$$

Finally, the Gini ratio is defined as

$$G_r = 1 - \frac{\hat{G}}{G_p}. \tag{10.10}$$

The value of the Gini ratio, G_r, represents the percentage increase in the information function defined in Equation 10.3 using Gini diversity measure. This function is the basis of our implementation of the algorithm described in Section 10.2.4.

Our SAS implementation is divided into three macros.

1. `GSplit()` returns the best split among a set of terminal nodes that are candidates for further splitting.
2. `GRedCats()` is the main macro. It creates the list of terminal nodes and calls `GSplit` to iteratively split them into more binary splits until the termination condition is reached. The result of the splits is stored in a mapping dataset. The mapping dataset contains the mapping rules for the generation of the new categories.
3. `AppCatRed()` is used to apply the mapping rules to new datasets, either at the time of modeling or during scoring.

We start with the main macro: `GRedCats` (see Table 10.2).

Table 10.2 Parameters of macro `GRedCats()`.

Header	`GRedCats(DSin, IVVar, DVVar, Mmax, DSGroups, DSVarMap);`
Parameter	*Description*
`DSin`	Input dataset
`IVVar`	Variable to be recategorized
`DVVar`	Dependent variable
`MMax`	Maximum number of categories
`DSGroups`	Dataset to store the new groups of categories
`DSVarMap`	Dataset to store the new group maps

Step 1

Determine the distinct categories of the `X` variable. This is achieved using the service macro `CalCats`. The results are stored in a temporary dataset, `Temp_cats`.

```
%CalcCats(&DSin, &IVVar, Temp_Cats);
```

Step 2

Convert the categories and their frequencies into macro variables using a simple `DATA` step.

```
Data _null_;
set Temp_Cats;
call symput ("C_" || left(_N_), compress(&IVVar));
call symput ("n_" || left(_N_), left(count));
call symput ("M", left(_N_));
Run;
```

Step 3

Calculate the count and percentages of $y = 1$ and $y = 0$ for each category using `PROC SQL` and store the results in a temporary dataset, `Temp_Freqs`.

```
proc sql noprint;
create table Temp_Freqs
 (Category char(100), DV1 num,
  DV0 num, Ni num, P1 num );
   %do i=1 %to &M;
select count(&IVVar) into :n1 from &DSin
where &IVVar = "&&C_&i" and &DVVar=1;
select count(&IVVar) into :n0 from &DSin
where &IVVar = "&&C_&i" and &DVVar=0;
%let p=%sysevalf(&n1 / &&n_&i);
insert into Temp_Freqs
values("&&C_&i", &n1, &n0, &&n_&i, &p);
  %end;
quit;
```

Step 4

Sort the categories according to the percentage of $y = 1$ and store this order into a set of macro variables.

```
proc sort data=temp_Freqs;
 by p1;
run;

data _Null_;
set temp_Freqs;
call symput("Cat_"||left(_N_), compress(Category));
run;
```

Step 5

Create a dataset to store the terminal nodes and the list of categories in each of them. This dataset is `Temp_TERM`. To initialize this dataset we insert *all* the categories. The number of terminal nodes now is only 1, which is the root node.

```
data Temp_TERM;
length node $1000;

Node='';
 %do j=1 %to &M;
   Node = Node ||" &&C_&j";
 %end;
run;
%let NNodes=1;
```

Step 6

Start the splitting loop. Convert all the rows of the dataset `Temp_TERM`, which are equivalent to terminal nodes, into macro variables representing the list categories to attempt to split in each node.

```
%DO %WHILE (&NNodes <&MMax);
Data _Null_;
set Temp_TERM;
call symput ("L_" || left(_N_), Node );
run;
```

Step 7

Loop on all these terminal nodes, attempt to split them by calling the macro `GSplit(.)`, and find the best split by maximizing the largest Gini measure (`BestRatio`).

```
%let BestRatio =0;
%DO inode=1 %to &NNodes;
 %let List_L=; %let List_R=; %Let GiniRatio=;
%GSplit(&&L_&inode, Temp_Freqs, List_L,
List_R, GiniRatio);
%if %sysevalf(&GiniRatio > &BestRatio)
%then %do;
%let BestRatio=&GiniRatio;
%let BestLeft=&List_L;
%let BestRight=&List_R;
%let BestNode=&Inode;
     %end;
%End;
```

Step 8

Add this best split to the list of terminal nodes `Temp_TERM` by adding its right and left children and by removing it from the list. Also increment the node count by +1. This completes the main splitting loop.

```
Data Temp_TERM;
  Set Temp_TERM;
    if _N_ = &BestNode Then delete;
run;
proc sql noprint;
 insert into Temp_TERM values ("&BestLeft");
   insert into Temp_TERM values ("&BestRight");
quit;

 %let NNodes=%Eval(&NNodes +1);
%END;   /* End of the splitting loop */
```

Step 9

We should now have a set of groups of categories that we need to map to the original ones. We will adopt a simple method for generating new names for the categories by subscripting the variable name by `_1`, `_2`, and so on. These new names will be stored in the output dataset `DSGroups`.

```
data &DSGroups;
set Temp_TERM (REname=(Node=OldCategory));
length NewCategory $40;
NewCategory="&IVVar._"||left(_N_);
run;
```

Step 10

We are now in a position to create the mapping dataset. The mapping dataset (`DSVarMap`) contains simply two columns, the old category name and the new name. This is done in two steps.

First convert the list of terminal nodes and the categories listed in them into a set of macro variables.

```
data _NULL_;
Set Temp_TERM;
call symput("List_"||left(_N_), Node);
call symput("NSplits",compress(_N_));
run;
```

Then create the table `DSVarMap` and loop on each of the macro variables, representing the list of categories in terminal nodes. Decompose each list and register the entries in the dataset `DSVarMap`. The decomposition of the list is performed using the service macro `Decompose(.)`.

```
proc sql noprint;
create table &DSVarMap
(OldCategory char(30), NewCategory char(30));
quit;
%DO ix=1 %to &NSplits;
  /* decompose each list */
%Decompose(&&List_&ix, Temp_list);
  /* convert the new list dataset to macro variables */
data _Null_;
set Temp_list;
call symput("CC_"||left(_N_),compress(Category));
call symput("CM", compress(_N_));
run;
/* insert all these values into the mapping dataset */
proc sql noprint;
%do j=1 %to &CM;
  insert into &DSVarMap values
       ("&&CC_&j", "&IVVar._&ix");
```

```
%end;
quit;
%END;
```

Step 11
Clean up and finish the macro.

```
proc datasets library = work nolist;
  delete temp_cats Temp_freqs Temp_term
         temp_list temp_gcats;
run;quit;
%mend;
```

The second macro, `GSplit()`, performs the actual splitting using the Gini measure (See Table 10.3).

Table 10.3 Parameters of macro `GSplit()`.

Header	`GSplit(Listin, DSFreqs, M_ListL, M_ListR, M_GiniRatio);`
Parameter	*Description*
`Listin`	Input list of categories
`DSFreqs`	Dataset containing frequencies of categories
`M_ListL`	List of categories in the *left* child node
`M_ListR`	List of categories in the *right* child node
`M_GiniRatio`	Resulting Gini ratio of this split

Step 1
Decompose the list into the categories and put them in a temporary dataset `Temp_GCats`, using the macro `Decompose(.)`. Then store these categories in a set of macro variables.

```
%Decompose(&Listin, Temp_GCats);
data _null_;
set Temp_GCats;
call symput("M",compress(_N_));
call symput("C_"||left(_N_), compress(Category));
run;
```

Step 2
Start a loop to go over the categories in the list and calculate the Gini term for the parent node.

```
proc sql noprint;
%let NL=0; %let N1=0; %let N0=0;
%do j=1 %to &M;
```

```
%let NL=%Eval(&NL+&&Ni_&j);
%let N1=%eval(&N1+&&DV1_&j);
%let N0=%eval(&N0+&&DV0_&j);
%end;
%let GL = %sysevalf(1 - (&N1 * &N1 + &N0 * &N0)
/(&NL * &NL));
quit;
```

Step 3

Loop on each possible split, calculate the Gini ratio for that split, and monitor the maximum value and the split representing that current best split. This will be used later as the best split for this list (terminal node).

```
%let MaxRatio=0;
%let BestSplit=0;
%do Split=1 %to %eval(&M-1);
/* the left node contains nodes from 1 to Split */
%let DV1_L=0;
%let DV0_L=0;
%let N_L=0;
%do i=1 %to &Split;
%let DV1_L = %eval(&DV1_L + &&DV1_&i);
%let DV0_L =  %eval(&DV0_L + &&DV0_&i);
%let N_L =  %eval(&N_L + &&Ni_&i);
%end;
/* The right node contains nodes from Split+1 to M */
%let DV1_R=0;
%let DV0_R=0;
%let N_R=0;
%do i=%eval(&Split+1) %to &M;
%let DV1_R = %eval(&DV1_R + &&DV1_&i);
%let DV0_R =  %eval(&DV0_R + &&DV0_&i);
%let N_R =  %eval(&N_R + &&Ni_&i);
%end;
%let G_L  = %sysevalf(1 - (&DV1_L*&DV1_L+&DV0_L*&DV0_L)
/(&N_L*&N_L));
%let G_R = %sysevalf(1 - (&DV1_R*&DV1_R+&DV0_R*&DV0_R)
/(&N_R*&N_R));
%let G_s= %sysevalf( (&N_L * &G_L + &N_R * &G_R)/&NL);

%let GRatio = %sysevalf(1-&G_s/&GL);
%if %sysevalf(&GRatio >&MaxRatio)
%then %do;
%let BestSplit = &Split;
%let MaxRatio= &Gratio;
%end;
    %end;
```

Step 4

Now that we know the location of the split, we compose the right and left lists of categories.

```
/* The left list is: */
 %let ListL =;
%do i=1 %to &BestSplit;
  %let ListL = &ListL &&C_&i;
 %end;
 /* and the right list is: */
%let ListR=;
%do i=%eval(&BestSplit+1) %to &M;
  %let ListR = &ListR &&C_&i;
 %end;
```

Step 5

Store the results in the output macro variables and finish the macro.

```
/* return the output values */
%let &M_GiniRatio=&MaxRatio;
%let &M_ListL=&ListL;
%let &M_ListR = &ListR;
%mend;
```

The final macro, `AppCatRed(.)`, is used to apply the mapping rules to a dataset. This macro would be invoked on both the mining and scoring views (see Table 10.4).

Table 10.4 Parameters of macro `AppCatRed()`.

Header	`AppCatRed(DSin, Xvar, DSVarMap, XTVar, DSout);`
Parameter	*Description*
`DSin`	Input dataset
`XVar`	Variable to be recategorized
`DSVarMap`	Mapping rules dataset obtained by macro `GRedCats(.)`
`XTVar`	New name of transformed variable
`DSout`	Output dataset with the new grouped categories

Step 1

Extract the old and new categories into macro variables.

```
Data _NULL_;
set &DSVarMap;
call symput ("OldC_"||left(_N_), compress(OldCategory));
call symput ("NewC_"||left(_N_), compress(NewCategory));
call symput ("Nc", compress(_N_));
run;
```

Step 2

Generate the rules and create the output dataset with the transformed variable. Finish the macro.

```
DATA &DSout;
length &XTVar $40;
set &DSin;
%do i=1 %to &Nc;
    IF &XVar = "&&OldC_&i" THEN &XTVar = "&&NewC_&i" ;
%end;
RUN;
%mend;
```

10.2.6 Limitations and Modifications

In the SAS code of the previous subsection, we implemented the algorithm to reduce the number of categories using a nominal dependent variable using the Gini measure. The implementation is subject to the following limitations.

DV Type

The SAS code included the calculations only for the case of a binary dependent variable. We can extend this code to include the use of two more types: nominal dependent variables with more than two categories and continuous DVs.

In the first case, more than two categories, there are two modifications to the algorithms. The first one deals with the extension of the equations defining the Gini ratio to more than two categories. The Gini term for the parent node is defined as

$$G_l = 1 - \frac{\sum_{k=1}^{c} N_k^2}{N^2}, \qquad (10.11)$$

with c being the number of categories of the DV. The Gini measure for the jth node is then given as

$$G_j = 1 - \frac{\sum_{k=1}^{c} (N_j^k)^2}{(n_j)^2}. \qquad (10.12)$$

The weighted Gini measure, $\hat{G}$, for the current split on X is similarly defined as

$$\hat{G} = \sum_{j=1}^{m} \frac{n_j}{N} G_j. \qquad (10.13)$$

Finally, the Gini ratio is calculated as

$$G_r = 1 - \frac{\hat{G}}{G_l}. \tag{10.14}$$

The second modification is the search for the split point in the list of ordered categories according to the content of the DV categories. In the case of a binary DV, we sorted the categories using their percentage of the category $y = 1$. In the case of a multicategory DV, we need to perform this search for each category. Therefore, we would be working with several lists simultaneously, each representing one of the categories being equivalent to the binary case of $y = 1$. Although it is straightforward to modify the SAS implementation to accommodate this situation, in most practical applications the DV is binary.

In the case of using a continuous DV, we will have to convert the definition of the information function to use the average value of the DV instead of the count of one of the categories. This is detailed in the next section.

Type of Information Function

So far, we have defined the information function using the Gini diversity measure. However, we can also define the information function using χ^2 or the Entropy variance.

The extension of the presented code to account for the type of dependent variable, and of the information functions, should be straightforward by modifying the relevant section of the code.

10.3 Binning of Continuous Variables

The main issue in binning is the method used to assign the bounds on the bins. These bounds can be assigned by dividing the range of the variable into an equal number of ranges; thus the bins are simply bins of equal width. This method is called *equal-width binning*. Other binning methods involve more sophisticated criteria to set the boundaries of such bins. In this section, we present three methods: *equal-width*, *equal-height*, and *optimal binning*.

10.3.1 Equal-Width Binning

Equal-width binning is implemented using the following steps.

1. Calculate the range of the variable to be binned.
2. Using the specified number of bins, calculate the boundary values for each bin.
3. Using the specified boundaries, assign a bin number to each record.

These steps are implemented in the macro `BinEqW2()`. This macro generates a new variable with the postfix `_Bin` containing the bin number. It also stores the mapping scheme in a dataset. A simpler version of the macro is `BinEqW()`, which does not store the binning scheme (see Table 10.5). As discussed, storing the binning scheme is important to allow the use of the same bin limits during scoring.

Table 10.5 Parameters of macro `BinEqW2()`.

Header	`BinEqW2(DSin, Var, Nb, DSOut, Map);`
Parameter	*Description*
`DSin`	Input dataset
`Var`	Variable to be binned
`Nb`	Number of bins
`DSOut`	Output dataset
`Map`	Dataset to store the binning maps

Step 1

Calculate the maximum and minimum of the variable `Var`, and store these values in the macro variables `Vmax` and `Vmin`. Then calcluate the bin size `BS`.

```
/* Get max and min values */
proc sql  noprint;
 select  max(&var) into :Vmax from &dsin;
 select  min(&Var) into :Vmin from &dsin;
run;
quit;

 /* calcualte the bin size */
%let Bs = %sysevalf((&Vmax - &Vmin)/&Nb);
```

Step 2

Loop on each value, using a `DATA` step, and assign the appropriate bin number to the new variable `&Var._Bin`, that is, the original variable name postfixed `Bin`.

```
/* Now, loop on each of the values and create the bin
   limits and count the number of values in each bin
*/
data &dsout;
 set &dsin;
  %do i=1 %to &Nb;
  %let Bin_U=%sysevalf(&Vmin+&i*&Bs);
  %let Bin_L=%sysevalf(&Bin_U - &Bs);
  %if &i=1 %then  %do;
```

```
IF &var >= &Bin_L and &var <= &Bin_U THEN &var._Bin=&i;
  %end;
  %else %if &i>1 %then %do;
IF &var > &Bin_L and &var <= &Bin_U THEN &var._Bin=&i;
 %end;
  %end;
run;
```

Step 3

Finally, create a dataset to store the bin limits to use later to bin scoring views or other datasets using the same strategy.

```
/* Create the &Map dataset to store the bin limits */
proc sql noprint;
 create table &Map (BinMin num, BinMax num, BinNo num);
  %do i=1 %to &Nb;
  %let Bin_U=%sysevalf(&Vmin+&i*&Bs);
  %let Bin_L=%sysevalf(&Bin_U - &Bs);
  insert into &Map values(&Bin_L, &Bin_U, &i);
  %end;
quit;
```

Step 4

Finish the macro.

```
%mend;
```

The only remaining issue in equal-width binning is how to assign the values if they happen to be exactly equal to the boundary value of the bins. This issue becomes a real problem in cases where the values of the variable take integer values and the binning boundaries are also integers. Consider a bin with upper and lower limits of b_u and b_l, respectively. We can set the rule to assign the value x to this bin using one of two rules:

A. Assign x to bin b if $b_l < x \leq b_u$.

B. Assign x to bin b if $b_l \leq x < b_u$.

Rule A will cause a problem only at the minimum value, and rule B will cause a problem at the maximum value. Intermediate integer values that coincide with b_l and b_u will be assigned to different bins depending on the rule we use. This can sometimes dramatically change the resulting distribution of the binned variable. One solution for values on the boundaries is to divide their frequencies to the two bins. However, this results in the counterintuitive noninteger frequency counts.

There is no standard solution to this problem. Different vendors of commercial analysis software use different methods for assigning records to the different bins.

10.3.2 Equal-Height Binning

In equal-height binning, we attempt to select the upper and lower limits of each bin such that the final histogram is as close to uniform as possible. Let us demonstrate this concept by a simple example.

Assume we have the following 20 values for a variable, x, which is sorted in ascending order: 20, 22, 23, 24, 25, 25, 25, 25, 25, 27, 28, 28, 29, 30, 32, 40, 42, 44, 48, 70. Assume further that we wish to bin this variable into 5 bins of equal height. In the case of equal-width binning, we find the bin size as

$$\text{Bin size} = (70 - 20)/5 = 10 \tag{10.15}$$

and the frequencies for each bin would be as shown in Table 10.6. (Note that in Table 10.6 the use of brackets follows the convention: ("(" means *larger than*, and "]", means *less then or equal to*).

Table 10.6 Equal-width bins.

Bin	*Bin limits*	*Frequency*
1	[20 , 30]	14
2	(30 , 40]	2
3	(40 , 50]	3
4	(50 , 60]	0
5	(60 , 70]	1

If we decided to change the bin limits such that each bin would contain more or less the same number of records, we would simply attempt to assign about 4 observations to each bin. This can be done using the scheme shown in Table 10.7.

Table 10.7 Equal-height bins.

Bin	*Bin limits*	*Frequency*
1	[20 , 24]	4
2	(24 , 25]	5
3	(25 , 28]	4
4	(28 , 42]	4
5	(42 , 70]	3

The frequency distribution of the new bins is almost uniform. However, the bins do not have exactly equal size but are more or less of equal height.

The algorithm adopted to set the values in equal height is a simple one. It works as follows:

1. Determine the approximate number of observations to be allocated to each bin. This number is calculated from the simple formula:

$$N_b = N/B, \tag{10.16}$$

   where

   N_b = Number of observations per bin

   N = Total number of observations

   B = Number of bins

   The calculation is rounded to the nearest integer.

2. Sort the values in ascending order.
3. Start with the first bin: Allocate the first `Nb` values to the first bin.
4. If repeated values fall on the bin boundary, there could be two types of solutions for this bin count: (1) we could carry the repeated values over to the next bin and leave this bin with less than the average frequency, or (2) we could allow the bin to have more than average frequency by taking all the repeated values into the current bin. We always take the option that would result in the minimum deviation from the required frequency.
5. Calculate the accumulated error in terms of the accumulated frequency of the bins compared with the expected cumulative frequency.
6. We may attempt to correct for this error immediately in the next bin or attempt to spread the correction over some or all of the remaining bins.

The implementation of this algorithm is straightforward, with the exception of the last step, which may lead to complications. The simplest method to handle the last step is to attempt to compensate for the accumulated error immediately in the very next bin.

The algorithm is implemented in the following macro.

Step 1

Find the unique values and their frequencies using `PROC FREQ`. Also calculate the cumulative frequencies (see Table 10.8).

```
proc freq data = &dsin noprint;
 tables &var / out=&var._Freqs outcum;
run;

/* Get sum of frequencies */
proc sql  noprint;
 select  max(cum_freq) into :SumFreq from &Var._Freqs;
```

Table 10.8 Parameters of macro `BinEqH()`.

Header	`BinEqH(DSin, Var, Nb, DSOut, Map);`
Parameter	*Description*
`DSin`	Input dataset
`Var`	Variable to be binned
`Nb`	Number of bins
`DSOut`	Output dataset
`Map`	Dataset to store the binning maps

```
run;
quit;
```

Step 2
Calculate the average bin size.

```
 /* calculate the bin size */
%let Bs = %sysevalf(&SumFreq/&Nb);
```

Step 3
Assign the bin number for each unique value by adjusting the width of the bins to account for possible spillovers.

```
data &Var._Freqs;
 set &var._freqs;

/* starting values */
retain bin 1 used 0 past 0 spill 0 binSize &Bs;

/* If a spillover occurred in the last iteration
   then increment the bin number, adjust the bin width,
   and save the used frequencies as "past" */
if spill=1 then do;
past = used;
BinSize=(&sumFreq-used)/(&Nb-bin);
Bin=Bin+1;
Spill=0;
end;
used = used + count;

/* check if this results in a spillover */
if used >= past +BinSize then spill =1;
else spill=0;

run;
```

Step 4

Find the upper and lower limits and the frequency in each bin.

```
proc sort data=&Var._Freqs;
 by bin;
run;

data &Var._Freqs;
 set &Var._Freqs;
  by bin;
 retain i 1;
 if first.bin then call symput ("Bin_L" || left(i), &var);
 if last.bin then do;
call symput ("Bin_U" || Left (i), &var);
i=i+1;
 end;
run;

/* Check the actual number of bins and get
   the frequency within each bin */
proc sql noprint;
 select max(bin) into :Nb from &Var._Freqs;
 %do i=1 %to &Nb;
  select sum(count) into :Bc&i from &Var._Freqs
                              where bin=&i;
 %end;
 run;
quit;
```

Step 5

Add the bin number to the data and store the bin limits in the `Map` dataset.

```
data &dsout;
 set &dsin;
  %do i=1 %to &Nb;
IF &var >= &&Bin_L&i and &var <= &&Bin_U&i
  THEN &var._Bin=&i;
  %end;
run;

/* and the binning limits to the dataset VarBins */

data &Map;
 do BinNo=1 to &Nb;
   &var._Lower_Limit=symget("Bin_L" || left(BinNo));
   &var._Upper_Limit=symget("Bin_U" || left(BinNO));
   Frequency = 1.* symget("Bc" || left(BinNo));
   output;
  end;
run;
```

Step 6
Clean the workspace and finish the macro.

```
/* clean workspace */
proc datasets library=work nolist;
delete &Var._Freqs;
quit;

%mend;
```

10.3.3 Optimal Binning

Equal-height binning attempts to make the frequencies of the records in the new bins equal as much as possible. Another way to look at it is that it attempts to *minimize* the difference between the frequencies of the new categories (bin numbers). Therefore, we consider it a special case of *optimal binning*.

However, in this section, we use the term *optimal* to mean that the new categories will be the *best* set of binning strategies so that the variable has the *highest predictive power* with respect to a specific dependent variable. This is similar to the case of reducing the cardinality of categorical variables, discussed earlier in this chapter, using structured grouping methods.

In fact, the method presented here depends on transforming the continuous variable to a large number of categories, using simple equal-width binning, and then using the Gini measure category reduction method to reduce these bins into the required number of bins. That is the reason we stated in Section 10.1 that binning could be treated as a special case of cardinality reduction.

The method described in the previous paragraph is actually the basis of decision tree algorithms with a continuous independent variable. As with cardinality reduction, we could have employed a number of alternative splitting criteria to find the optimal grouping, such as the Entropy variance, or χ^2, of the resulting grouping.

The macro `GBinBDV()` is the SAS implementation using the Gini measure, which assumes that the dependent variable is binary (1/0). (See Table 10.9)

The parameter `NW` is used to initially divide the range of the variable `IVVar` to `NW` equal divisions before grouping them to the final bins. The larger this number, the more accurate the binning, but the more it becomes demanding in terms of computing time. Also note that this macro does not result in an output dataset with the bin numbers assigned to the records. We will implement this separately in an another macro.

The implementation assumes the following.

- The dependent variable is binary (1/0).
- The variable to be binned is continuous. If it contains only integer values, the resulting bin limits may not be integers.
- No missing values are accounted for in either the dependent variable or the binning variable.

The macro is very similar to that used to reduce the cardinality of nominal variables, `GRedCats()`, with the exception of the initial binning, using the equal-width method, and the calculation of the generated mapping rules.

Table 10.9 Parameters of macro `GBinBDV()`.

Header	`GBinBDV(DSin, IVVar, DVVar, NW, Mmax, DSGroups, DSVarMap);`
Parameter	*Description*
`DSin`	Input dataset
`IVVar`	Continuous variable to be binned
`DVVar`	Dependent variable used to bin `IVVar`
`NW`	Number of divisions used for initial equal-width binning
`MMax`	Maximum number of bins
`DSGroups`	Dataset with final groups of bins (splits)
`DSVarMap`	Dataset with mapping rules

Step 1
Start by binning the variable `IVVar` into a total of `NW` equal-width bins using the macro `BinEqW2()`. But first copy the `IVVar` and the `DVVar` into a temporary dataset to make sure that the original dataset is not disturbed.

```
   Data Temp_D;
     set &DSin (rename=(&IVVar=IV &DVVAR=DV));
     keep IV DV;
   run;

%BinEqW2(Temp_D, IV, &NW, Temp_B, Temp_BMap);
```

Step 2
Determine the count of each bin and the percentage of the `DV=1` and `DV=0` in each of them. We know that the bin numbers are from 1 to `NW` , but some bins may be empty. Therefore, we use macro `CalcCats()` to get the count of only those in the `Temp_GB` dataset.

```
%CalcCats(TEMP_B, IV_Bin, Temp_Cats);
```

Step 3

Next, sort `Temp_Cats` using `IV_Bin` and convert the bins to macro variables.

```
proc sort data=Temp_Cats;
   by IV_Bin;
 run;

Data _null_;
  set Temp_Cats;
       call symput ("C_" || left(_N_), compress(IV_Bin));
         call symput ("n_" || left(_N_), left(count));
       call symput ("M", left(_N_));
  Run;
```

Step 4

Calculate the count (and percentage) of `DV=1` and `DV=0` in each category using `PROC SQL` and store the results values in the dataset `Temp_Freqs`.

```
proc sql noprint;
  create table Temp_Freqs (Category char(50),
                             DV1 num, DV0 num,
                             Ni num,  P1 num );

%do i=1 %to &M;
  select count(IV) into :n1 from Temp_B
                where IV_Bin = &&C_&i and DV=1;
  select count(IV) into :n0 from Temp_B
                where IV_Bin = &&C_&i and DV=0;
  %let p=%sysevalf(&n1 / &&n_&i);
  insert into Temp_Freqs
              values("&&C_&i", &n1, &n0, &&n_&i, &p);
%end;
quit;
```

Step 5

Create the `TERM` dataset to keep the terminal nodes and their category list, and initialize the node counter. Store all the categories as a starting point.

```
data Temp_TERM;
  length node $1000;

  Node='';
  %do j=1 %to &M;
    Node = Node ||" &&C_&j";
  %end;
run;
```

Step 6
Start the splitting loop.

```
%let NNodes=1;

%DO %WHILE (&NNodes <&MMax);
/* Convert all the rows of the splits to macro variables,
   we should have exactly NNodes of them. */
   Data _Null_;
      set Temp_TERM;
       call symput ("L_" || left(_N_), Node );
   run;
/* Loop on each of these lists, generate possible splits
   of terminal nodes, and select the best split using
   the GiniRatio. */
%let BestRatio =0;

%DO inode=1 %to &NNodes;
/* The current node list is &&L_&i
   Using this list, get the LEFT and RIGHT categories
   representing the current best split, and the
   Gini measure of these children. */
   %let List_L=; %let List_R=; %Let GiniRatio=;
%GSplit(&&L_&inode, Temp_Freqs, List_L, List_R, GiniRatio);

 /* Compare the GiniRatio, if this one is better,
    and keep a record of it. */
%if %sysevalf(&GiniRatio > &BestRatio) %then %do;
%let BestRatio=&GiniRatio;
%let BestLeft=&List_L;
%let BestRight=&List_R;
%let BestNode=&Inode;
     %end;

%End; /* end of the current node list */

/* Add this split to the Temp_TERM by removing the
   current node, and adding two new nodes. The contents of the new
   nodes are the right and left parts of the current node. */
Data Temp_TERM;
  Set Temp_TERM;
    if _N_ = &BestNode Then delete;
run;
proc sql noprint;
 insert into Temp_TERM values ("&BestLeft");
   insert into Temp_TERM values ("&BestRight");
quit;
```

```
 /* increment NNodes */
 %let NNodes=%Eval(&NNodes +1);

%END;   /* End of the splitting loop */
```

Step 7

Now we should have a set of bin groups, which we need to map to a new set of ranges for final output and transformation of the input dataset. These new ranges will be obtained by getting the lower and upper bounds on the smallest and largest bins in each node. The results are stored in the output dataset `DSVarMap`.

```
/* we get all the final lists from the splits */
data _NULL_;
 Set Temp_TERM;
   call symput("List_"||left(_N_), Node);
   call symput("NSplits",compress(_N_));
run;

/* And we create the new explicit mapping dataset */
proc sql noprint;
 create table &DSVarMap (BinMin num, BinMax num,
                        BinNo num);
quit;
%DO ix=1 %to &NSplits;
/* Get the first and last bin number from each list. */
 %let First_bin=;%let Last_bin=;
 %FirstLast(&&List_&ix, First_bin, Last_bin);

 /* get the outer limits  (minimum first, maximum last)
    for these bins */
 proc sql noprint;
   select BinMin into :Bmin_F from Temp_BMap
                      where BinNo=&First_bin;
   select BinMax into :Bmax_L from Temp_BMap
                      where BinNo=&Last_bin;

   /* Store these values in DSVarMap under the
      new bin number: ix */
   insert into &DSVarMap values (&Bmin_F, &Bmax_L, &ix);
 quit;
%END;
/* Generate DSGroups */
 data Temp_TERM;
     set Temp_TERM;
first=input(scan(Node,1),F10.0);
 run;
 proc sort data=Temp_TERM;
```

```
  by first;
 run;

 Data &DSGroups;
  set Temp_TERM (Rename=(Node=OldBin));
  NewBin=_N_;
  drop first;
 run;

    /* Because the split number is not representative
       of any specific order, we should sort them on
       the basis of their values */
     proc sort data=&DSVarMap;
      by BinMin;
     run;
     /* and regenerate the values of BinNo accordingly. */
     data &DSVarMap;
      Set &DsVarMap;
       BinNo=_N_;
     run;
```

Step 8

Clean the workspace and finish the macro.

```
/* clean up and finish */
 proc datasets library = work nolist;
 delete temp_b Temp_bmap Temp_cats temp_d
        temp_freqs temp_gcats temp_Term;
    run;quit;

%mend;
```

To apply the determined bins, we present the following macro, which reads the bin limits from the maps dataset and generates the new bin number in the input dataset (see Table 10.10).

The macro is implemented in two steps.

Step 1

The binning conditions are extracted.

```
%macro AppBins(DSin, XVar, DSVarMap, XTVar, DSout);

/* extract the conditions */
data _Null_;
 set &DSVarMap;
  call symput ("Cmin_"||left(_N_), compress(BinMin));
```

Table 10.10 Parameters of macro `AppBins()`.

Header	`AppBins(DSin, XVar, DSVarMap, XTVar, DSout);`
Parameter	*Description*
`DSin`	Input dataset
`XVar`	Continuous variable to be binned
`DSVarMap`	Dataset with mapping rules
`XTVar`	The new transformed variable (bin numbers)
`DSOut`	Output dataset

```
  call symput ("Cmax_"||left(_N_), compress(Binmax));
  call symput ("M", compress(_N_));
run;
```

Step 2

Then the new transformed variable is added to the output dataset.

```
/* now we generate the output dataset */
Data &DSout;
  SET &DSin;
   /* The condition loop */
   IF &Xvar <= &Cmax_1  THEN  &XTVar = 1;
   %do i=2 %to %eval(&M-1);
     IF &XVar > &&Cmin_&i AND &XVar <= &&Cmax_&i
      THEN  &XTVar = &i ;
    %end;
   IF &Xvar > &&Cmin_&M  THEN  &XTVar = &M;
 run;

%mend;
```