Chapter 9 Data Transformations

9.1 Raw and Analytical Variables

Data transformation is perhaps the most important step in the data preparation process for the development and deployment of data mining models. Proper transformations can make the difference between powerful and useless models. In almost all practical cases, it is necessary to perform some transformations on extracted data before it can be used in modeling. Data transformation has two objectives.

1. Generate new variables, called analytical variables.
2. Fix any potential problems such as missing values and skewed variable distributions.

The mining view, which is extracted from a database or a data warehouse, usually contains *raw variables.* Raw variables are those that have a direct business meaning and exist in the same format and have the same content as in the original operational system. In most cases, these variables do not suffice to render the best predictive models. Therefore, a set of data manipulation steps is used to transform some or all of these variables into new variables with better predictive power, or better behavior with respect to the model formulations. The new variables are usually called *modeling variables* or *analytical variables.* An important feature of analytical variables is that some of them may not have a clear business meaning. Let us illustrate this point with a simple example.

When the database contains the data field of `Date Of Birth`, we can use this field as the raw variable in the mining view. However, using dates is not convenient because of the complexity of their arithmetic. Therefore, it is more appropriate to create a new field representing `Age` instead. Note that `Age` is a transformation on the date, which is calculated by subtracting the actual raw `Date Of Birth` from a reference date, which is given the default value of the current date. The value representing the number of years in the result is what we use as `Age`. However, further investigations into the modeling process may indicate that it is better to use the logarithm of `Age` instead of the raw value. This simple transformation usually spreads the histogram of a variable into a more favorable form, as detailed later in this chapter. The new analytical variable `Log(Age)` does not really have an easy-to-explain business meaning. For example, a customer with an age of 26 will have a `Log(Age)` of `3.258` (using natural logarithms). This is not a common way to represent a person's age.

In contrast, many transformations can yield analytical variables that have a clear meaningful and even useful business use. The following are examples of such variables commonly used in marketing and credit risk scoring applications.

- Average transaction value over a certain period (month, quarter, year)
- Average number of transactions over a certain period (month, week, etc.)
- Ratio of purchases from a specific product or product category to total purchases
- Ratio of total purchases in a time period to available credit in the same period
- Ratio of each purchase to the cost of shipment (specially when shipping costs are paid by the customer)
- Ratio of number/value of purchased products to size of household
- Ratio of price of most expensive product to least expensive product
- Ratio of largest to smallest purchase in a time period
- Ratio of total household debt to total household annual income
- Ratio of largest income in a household to total household income
- Ratio of available credit to household income
- Average annual income per household member
- Ratio of debt to available credit
- Average percentage of used portion of credit over a certain period

9.2 Scope of Data Transformations

The number of possible data transformations is indeed large and limited only by imagination. In this book, we focus on the most common and useful transformations that can be easily implemented and automated using SAS. However, before we start exploring some of these transformations, it is necessary to note the following general principles that need to be considered when deciding on implementing certain data transformations.

1. *Efficiency of Transformations and Data Size*
   We need to take into consideration the size of the data to avoid creating a bottleneck in the data preparation process. This is particularly important in the case of using a transformation during the scoring process. One should always try to avoid computationally demanding transformations unless absolutely necessary. For example, in a credit scoring model one may normalize a payment amount by the maximum payment within a subgroup of the data. This requires the calculation of the maximum payment within the subgroups, then the normalization of the payment amounts by these values for the different groups. When both the number of subgroups and the number of records are large, this type of transformation becomes computationally demanding.

2. *Scope of Modeling and Scoring Datasets*
   Some, and in rare cases all, of the analytical variables resulting from data transformations are used in the construction of the data mining model. Because the resulting model depends on these variables, the raw variables must also be subjected to the same transformations at the time of scoring. This fact should always be remembered to make sure that any transformation performed during modeling, and its result being used in the construction of the model, should also be *possible* during scoring. Sometimes this can be tricky. For example, assume that we have the seven values of income (in dollars) shown in Table 9.1.

   One can normalize these values to make them take values between 0 and 1, as shown in Table 9.2. In doing so, we used the maximum and minimum values and used the simple formula:

$$\text{Normalized value} = (x - x_{\min})/(x_{\max} - x_{\min}), \qquad (9.1)$$

   where, x, $x_{\min}$, and $x_{\max}$ are income, minimum income, and maximum income, respectively.

   The problem with such efficient, but naive, transformation is that we *cannot* assume that it will lead to values between 0 and 1 at the time of scoring. There

Table 9.1 Seven income values.

15,000	56,000	22,000	26,000	34,000	44,000	46,000

Table 9.2 Normalized income values.

0	1.00	0.17	0.27	0.46	0.71	0.76

could be records with income values higher than $56,000 or lower than $15,000. Also, we should not attempt to find the new maximum and minimum using the scoring data and then use them to apply the transformation because the model would have already been constructed on the basis of the minimum and maximum found in the modeling data. The only solution in this case, if we really want to use this transformation, is to keep in mind that this variable can take values outside the 0–1 range at the time of scoring.

3. *Modes of Scoring*
There are two modes of scoring: batch scoring and real-time scoring. In the case of batch scoring, the scoring view is prepared with all the required transformations and then the model is used to score the entire set of records at one time. On the other hand, in the case of real-time scoring the scoring view contains only one record. The transformations are performed on that record; then the resulting values are fed to the model to render the score. Therefore, any data transformations that require information about other records may not be possible or technically allowable in terms of the time required to perform them. Realistically, at least at the time of writing this book, the majority of data mining model deployment scenarios employ batch scoring. However, with the increasing power of Internet-based applications and distributed databases, it is anticipated that the number of data mining models used in real-time scoring will increase significantly.

4. *Interpretation of Results*
In many cases, the resulting scores of models need to be interpreted in terms of the variables used in developing them. A typical example is credit scoring. In this case, consumers in many countries have the right to know the justification behind their credit scores. This in turn requires that these variables themselves be meaningful in terms of the business problem being modeled. However, it is sometimes hard to explain the meaning of some of the transformations that prove to result in variables leading to good models.

For example, it is difficult to assign an easy meaning, or even a label, to a variable that has the following definition:

$$\text{Variable}X = (T * B_6)^{1.6} / \log(\text{TCC6} * \text{IRate}), \tag{9.2}$$

where T is customer tenure, B_6 is the average balance of the checking account in the last six months, TCC6 is the number of transactions on the customer's credit card in the last month, and IRate is the maximum of a constant of ten or the average interest paid by the customer in the last three months.

Although the variable X, as defined, may be a good predictor of a certain behavior, and therefore can be used in the model, it is challenging to attempt to provide an easy-to-understand explanation for it.

The remaining sections of this chapter discuss the implementation of the most common variable transformation methods:

- Creation of new variables
- Mapping of categorical variables
- Reduction of cardinality for categorical variables
- Normalization of continuous variables
- Binning of continuous variables
- Changing the variable distribution

9.3 Creation of New Variables

The most obvious type of data transformation is the creation of new variables by combining raw variables using different mathematical formulas. There are two main methods of generating these formulas: (1) business-related formulas and (2) pure mathematical expressions.

The first approach relies on the analyst's understanding of the business problem, where the new variables would represent meaningful, or potentially meaningful, quantities that can be deduced from the raw variables. These quantities are usually based on using averages, sums, ratios, or other similar simple formulas. This category of variables has two advantages.

1. They are usually derived using simple formulas that can be easily programmed and implemented at both the modeling and the scoring stage.
2. They have definite business interpretations that can give an insight into the meaning of the results of the model.

The second method that can be used to create new variables relies on use of mathematical expressions that do not necessarily have a certain interpretation, but may prove beneficial to the model. The challenge with this type of variable is finding the appropriate mathematical expressions to define it. Experience with the data plays a significant role in helping the analyst to select appropriate candidate formulas. However, there exist many methods to generate these variables by relying on some heuristic that generates a large number of potentially useful transformations, then reduces this set of candidates into a much smaller set. The reduction is performed by the selection of those leading to a better model. The different algorithms implementing this heuristic share the following aspects.

1. They implement a predefined set of allowable forms to generate the new variables. Each algorithm offers the use of a set of equation forms, such as polynomials, products and divisions of database variables, and so on.

2. They adopt a criterion, or several of them, to filter the newly generated variables. Such criteria include correlation coefficients, entropy variance, the Pearson Chi-squared statistic, and the Gini variance, among others. The filtering criteria are used to formulate a merit function to score each of the new variables. The merit function could simply be the raw value of the filtering criteria, such as the correlation coefficient, or a complex function of several of them.
3. They use mechanism for reducing the variables using the merit function. One approach is to generate *all* the possible variables, then calculate their merit function value and select the best ones. Variations on this approach include the setup of a critical cutoff value for the merit function below which a variable would be dropped. The reduction phase of the analytical variables can also include the raw variables as candidates for reduction. The reduction procedures implement several iterative or stepwise reduction phases. For example, the correlation coefficient may be used to remove some variables in the first round; then the Gini variance could be used in a second run. Other heuristics attempt to *group* the variables in subsets that are either correlated to each other or have been derived from the same set of raw variables. Then the reduction process operates on the best variable from each group.

The last step of the algorithm, dealing with the reduction of automatically generated analytical variables, can be postponed until all the other analytical variables have been generated through other transformations, or it can be applied twice. The first time is when new variables are created to reduce their numbers and the second time when all the other transformations have been completed.

9.3.1 Renaming Variables

In order to develop a good implementation of variable generation and selection, it is usually easier to work with a set of *normalized* variable names. By that we mean that the variable names are mapped from their original values, such as age, income, and *TXC*, to a simple sequence of symbols such as X_1, X_2, and X_3. The use of such sets of normalized variable names in the generation of new variables allows us to implement a simple coding convention for new variables. Table 9.3 shows a simple coding scheme for naming new variables using the different operations.

The use of variables with the normalized names X_1, X_2, $\cdots$ in such a scheme is obviously easier than using the original raw names and will result in a SAS code that is smaller and less error prone. The following is a description of two macros that do this mapping and store the mapping scheme in a dataset. The first macro extracts the variable names from a list and generates a list of normalized names. It also saves the mapping scheme in a dataset. The second macro uses the mapping scheme to rename the variables in a new dataset, which is useful at the time of scoring (see Table 9.4).

Table 9.3 A convention for naming new variables.

Operation	*Coding scheme*	*Example*	*Meaning*
Log	`L_Var`	`L_`X_1	Log of variable x_1
To power y	`P_`y`_Var`	`P_255_X1`	$x_1^{(2.55)}$
Division	`D_Var1_Var2`	`D_X1_X2`	x_1/x_2
Multiplication	`M_Var1_Var2`	`M_X1_X2`	$x_1 * x_2$
Addition	`A_Var1_Var2`	`A_X1_X2`	$x_1 + x_2$

Table 9.4 Parameters of macro `NorList()`.

Header	`NorList(ListIn, VPrefix, M_ListOut, MapDS);`
Parameter	*Description*
`ListIn`	Input list of original variable names
`VPrefix`	Prefix used to generate the normalized variable names
`M_ListOut`	Output list of normalized variable names
`MapDs`	Dataset of old and new variables (mapping scheme)

Step 1

We decompose the input list to the variable names and store them in a dataset. We use the macro `ListToCol()`, included in this book's appendix, which converts a list of variable names to the rows of a string variable in a dataset. We store the results in a temporary dataset, `Temp_Vars`.

```
%ListToCol(&ListIn, Temp_Vars, OldVarName, VIndex);
```

Step 2

Using a simple loop on this dataset, we create the mapping dataset and generate the new normalized variable names using the given prefix.

```
data &MapDS;
set Temp_Vars;
NewVarName=compress("&VPrefix" ||'_'|| VIndex);
run;
```

Step 3

Finally, we extract the new names into the output list. This step is only for checking the integrity of the results because we already know the names of the new variables, and we could have simply written them in such a list. But this way we will guarantee that we have completed the mapping properly.

```
Data _Null_;
  set &MapDS;
   call symput ("x_" || left(_N_), NewVarName);
   call symput ("N" , _N_);
 run;
%let LOut=;
%do i=1 %to &N;
  %let LOut=&lOut &&x_&i;
%end;
%let &M_ListOut=&Lout;
```

Step 4

Clean the workspace and finish the macro.

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

The following code segment shows how to use the preceding macro.

```
%let Listin=a b c d e X1 X2 X45 Y8 H99 Age;
%let ListOut=;
%let MapDs=MapVars;

%NorList(&ListIn, Y, ListOut, &MapDS);
%put Output List = &ListOut;
proc print data=MapVars;
run;
```

Now the next macro implements the name-map dataset generated by the previous macro to normalize variable names in an input dataset.

Table 9.5 Parameters of macro `NorVars()`.

Header	`NorVars(DSin, MapDS, DSout);`
Parameter	*Description*
`DSin`	Input dataset with the original variable names
`MapDS`	Dataset with name map generated by macro `NorList` above
`DSout`	Output dataset with normalized variable names

Step 1

Extract the old and new variable names from the name-map dataset to the macro variables.

```
Data _Null_;
  set &MapDS;
   call symput ("N_" || left(_N_), NewVarName);
   call symput ("O_" || left(_N_), OldVarName);
   call symput ("N" , _N_);
 run;
```

Step 2

Compile a variable-name list for use in a (RENAME) statement to generate the final dataset.

```
%let RenList=;
%do i=1 %to &N;
  %let RenList=&RenList %left(&&O_&i) = %left(&&N_&i);
%end;
```

Step 3

Apply the new name changes to a new dataset using the RENAME option of a simple DATA step, which does nothing other than store the output dataset with the new variable names.

```
DATA &DSout;
  SET &DSin  (RENAME=(&RenList));
RUN;
```

Step 4

Finish the macro.

```
%mend;
```

The next code shows the use of the macro to rename the variables Age and Income in the following simple dataset to Y_1 and Y_2, respectively.

```
Data Test;
 input CustName$ Age Income;
Datalines;
 Tom   20 53000
 Al    30 40000
 Jane  35 60000
;
run;
```

```
%let DSin=Test;
%let DSout=test_Nor;
%let ListIn=Age Income;
%let ListOut=;
%let MapDS=MapVar;

%NorList(&ListIn, Y, ListOut, &MapDS);
%NorVars(&DSin, &MapDS, &DSout);

proc print data=Test_Nor;
run;
```

9.3.2 Automatic Generation of Simple Analytical Variables

Automatic generation of new analytical variables is a powerful technique that can be used when the required mathematical form is known. For example, the use of *interaction terms* is popular in regression analysis because it allows the introduction of nonlinear effects in the modeling data in a simple but effective way. Similarly, but less popular, the introduction of *ratio terms* between variables could also reveal nonlinearities. However, with the ratio terms extra care should be taken because of the possibility of dividing by zero. Other forms that can be introduced include taking the logarithm of continuous variables to spread their histogram.

We now provide the details of the macro that will generate automatic variables to provide the interaction terms of a list of variables.

Table 9.6 Parameters of macro `AutoInter()`.

Header	`AutoInter(ListIn, DSin, M_ListInter, DSout, DSMap);`
Parameter	*Description*
`ListIn`	Input list of variables to be used in the generation of interaction terms
`DSin`	Input dataset
`M_ListInter`	Output list of new variables
`DSOut`	Output dataset with interaction terms
`DSMap`	Dataset with mapping rules

Step 1

Begin by decomposing the input dataset using the utility macro `ListToCol()`.

```
%ListToCol(&ListIn, Temp_Vars, BaseVars, VIndex);
```

Step 2

Extract the names of the variables from the dataset `Temp_vars` to a set of macro variables, $x_1, x_2, \ldots$.

```
Data _Null_;
  set Temp_Vars;
   call symput ("x_" || left(_N_), BaseVars);
   call symput ("N", _N_);
 run;
```

Step 3

Create the dataset `DSMap` to hold the variables and their transformations. Start by storing the raw variables and their names. In the same dataset, generate the new variables and the formulas used to generate them. In this case, the formulas describe the interaction terms of the variables in the list.

```
Data &DSMap;
 format VarName $80.;
 format Expression $200.;
/* the first base ones are simply themselves */
 %do i=1 %to &N;
   VarName = "&&x_&i";
   Expression = VarName;
   Output;
 %end;
/* Generate the new variable transformation equations
   using the following two loops:  */
%do i=1 %to %eval(&N-1);    /* first loop: variable i */
  %do j=%eval(&i+1) %to &N;     /* Second loop: variable j */
    /* compose the new variable name */

VarName = "%sysfunc(compress(M_ &&x_&i  _ &&x_&j))" ;

    /* Compose the equation LHS */
    %let RHS_&i._&j= &&x_&i * &&x_&j ;
    Expression = compress("&&RHS_&i._&j");
    output;
  %end;
 %end;
run;
```

Step 4

Finally, apply these transformations and create the new variables in the output dataset.

```
Data &DSout;
 set &DSin;
%do i=1 %to %eval(&N-1);    /* Loop: variable i */
```

```
  %do j=%eval(&i+1) %to &N; /* Loop: variable j */
  /* the new variable equation */
  %sysfunc(compress(M_ &&x_&i _ &&x_&j)) = &&RHS_&i._&j;
  %end;
 %end;
run;
```

Step 5

Clean the workspace and end the macro.

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

The following code shows how to implement the preceding macro for an input dataset.

```
Data Test;
 input ID x1 x2 X3 TCmx;
 Datalines;
1 4 4 5  6.6
2 8 9 9  2.0
3 1 3 4  3.1
4 2 9 3 -1.8
5 3 3 4 -2.8
;
run;

%let ListIn=x1 x2 x3 TCmx;
%let DSin=Test;
%let ListInter=;
%let DSout=test_Inter;
%let DSMap=MapVar;

%AutoInter(&ListIn, &DSin, ListInter, &DSout, &DSMap);
```

9.4 Mapping of Nominal Variables

By mapping, we mean the generation of a set of *indicator,* or *dummy,* variables to represent the different values of a categorical variable. For example, suppose we have a variable representing the type of customer residence, taking the values Rent (R), Own (O), Live with family (WF), Shared accommodation (SA), and Unknown (U). One could generate five indicator variables that take a value of either 1 or 0, depending on the value of the original variable, as shown in Table 9.7.

Table 9.7 Mapping of residence types.

Residence type	*R*	*O*	*WF*	*SA*	*U*
Rent	1	0	0	0	0
Own	0	1	0	0	0
With family	0	0	1	0	0
Shared accommodation	0	0	0	1	0
Unknown	0	0	0	0	1

The mapping in Table 9.7 is known as 1-to-N mapping, which means that one variable resulted in N new indicator variables, with N being the total number of categories of the original variable. The number of categories in a variable, N, is known as its *cardinality.*

A closer examination of the values reveals that we could infer the value of one of the new indicator variables by knowing the values of the others. In mathematical jargon, these variables are linearly dependent. Therefore, one of these variables can be dropped without loss of information. This is known as 1-to-$N-1$ mapping. In this case, one of the values is dropped and a total of $N-1$ indicator variables are generated in place of the original variable. The dropped value could be either the last new indicator variable, say alphabetically, or the one equivalent to the category with the lowest frequency.

A word of warning about mapping nominal variables: This should not be performed for variables with *high cardinality.* Attempting to do so will result in a large number of indicator variables, which in turn will lead to overly sparse data. Such sparse datasets very often lead to numerical and performance problems with many modeling algorithms. Therefore, mapping nominal variables with the macros described next should be performed only after reducing the cardinality of target variables to acceptable limits.

There are no scientific rules for determining an acceptable value for the maximum number of categories. We can only suggest, as a rule of thumb derived from experience, that categorical variables with a cardinality higher than about 10 to 12 are good candidates for cardinality reduction algorithms. These algorithms are discussed in Section 10.2.

As in the case of continuous variables, where we recommended mapping their names to a set of normalized names, it is recommended that the mapped nominal variables be given names using a systematic procedure. One such scheme is to append the variable name with the category index or with the value of the category itself. In the following SAS implementation, we allow for both methods. Of course, before deciding to map a nominal variable, we recommend that the different categories and their frequencies be first calculated and examined. Therefore, we start with the macro `CalcCats()`, which finds the categories and their frequencies, as shown in Table 9.8. The macro is a simple wrapper to `PROC FREQ` and is self-explanatory.

Table 9.8 Parameters of macro `CalcCats()`.

Header	`CalcCats(DSin, Var, DSCats);`
Parameter	*Description*
`DSin`	Input dataset
`Var`	Variable to calculate categories for
`DSCats`	Output dataset with the variable categories

```
%macro CalcCats(DSin, Var, DSCats);
proc freq data=&DSin noprint;
tables &Var /missing out=&DSCats;
run;
%mend;
```

The next macro is `MappCats()`, which does the actual mapping and uses `PROC FREQ` as one of its steps to calculate the different categories and their frequencies.

Table 9.9 Parameters of macro `MappCats()`.

Header	`MappCats(DSin, var, Method, DSout, DSCats);`
Parameter	*Description*
`DSin`	Input dataset
`Var`	Variable to calculate categories for
`Method`	Method of extending the variable name: Method =1 uses the index of categories, Method $\neq$ 1 uses the value of the category
`DSout`	The output dataset with the mapped variables
`DSCats`	Dataset with the variable categories

Step 1

Start by finding the different categories of the variable and their frequencies and store them in the dataset `DSCats`.

```
proc freq data=&dsin noprint;
tables &var /missing out=&DSCats;
run;
```

Step 2

Convert the categories to macro variables in order to use them, or their indexes, in the generation of the new indicator variables. Note that if the values of the categories are to be appended to the variable names, the resulting variable names must constitute valid SAS variable names. Furthermore, they should not contain special characters, such as +, -, and &, or spaces and will be within the permissible length of SAS variables. Missing values are assumed to have been replaced with some other value to allow such appending.

```
data &DSCats;
 set &DSCats;
 call symput("y"||left(_n_),&var);
 CatNo=_N_;
run;
```

Step 3

Count the categories and use a loop to generate the new variables in the output dataset, using either of the methods given in the macro parameter `Method`.

```
proc sql noprint;
 select count(*) into :Nc from &dscats;
run;
quit;

data &dsout;
 set &dsin;
  %do i=1 %to &Nc;

%if &Method=1 %then %do;
IF &var = left("&&y&i") then &var._&i=1;
ELSE    &var._&i=0;
%end;
                %else %do;
  IF &Var = left("&&y&i")  then &Var._&&y&i =1;
  ELSE    &Var._&&y&i =0;
                %end;

  %end;
run;
```

Step 4

Finish the macro.

```
%mend;
```

Let us demonstrate the use of the macro through a simple example. The following code generates the dataset Test and maps the variable X to new dummy variables.

```
data Test;
 input X $ y $;
datalines;
 A B
 C A
 A A
 B B
 B .
 B B
 D A
 ;
 RUN;

%let dsin=test;
%let var=X;
%let dsout=test_m;
%let dscats=Test_cats;

%MappCats(&dsin, &var,2, &dsout, &dscats);
```

The macro MappCats() performs 1-to-N mapping. If 1-to-$N-1$ mapping is required, we need only change step 3 where we limit the loop to only $N-1$ of the categories. Another option is to decide which category to remove by inspecting the results of the frequency distribution and remove, say, the category with the smallest frequency. This can be easily performed by doing a 1-to-N mapping and then dropping the unnecessary dummy variable.

9.5 Normalization of Continuous Variables

Normalization of continuous variables is performed by rescaling the values such that the variables have a mean value of zero and a standard deviation of one. This is usually termed *standard normalization.* There are two possible situations where the normalization of continuous variables is beneficial in the data mining process.

The first situation arises in several modeling algorithms that need the variables to be properly scaled in order to avoid numerical ill conditions. This is particularly true in the cases of linear and logistic regression, principal component analysis, and factor analysis. On the other hand, the implementation of these algorithms in most modern software packages, such as SAS, includes this normalization behind the scene.

The other situation is that of a business meaning of the normalized variable, especially when the values of the variable in the normalized form will be used for comparison and interpretation. In this case, rescaling does not necessarily create a mean of zero and a standard deviation of one. A typical example is to normalize the credit risk of a customer between zero and 1000. In such cases, the values are scaled such that they fall into a certain range. This type of normalization is termed *mid-range normalization.*

PROC STDIZE allows the user to standardize a set of variables using several criteria, including the mid-range normalization.

As for range standardization, the implementation only requires the computation of the minimum and maximum values and uses the following simple equation:

$$y = y_{\min} + R\left(\frac{x - x_{\min}}{x_{\max} - x_{\min}}\right), \tag{9.3}$$

where,

x	The current value
y	The new standardized value
$x_{\min}, x_{\max}$	Minimum and maximum original values
$y_{\min}$	Minimum value for the standardized range
R	The width of the standardized range

The SAS implementation of the transformation is straightforward.

9.6 Changing the Variable Distribution

Changing the distribution of a variable, independent or dependent, can result in a significant change in model performance. This may reveal relationships that were masked by the variable distribution. This section discusses three methods for transforming the distribution of a continuous variable.

The first one is based on the variable ranks, and we show how the ranks could lead to the transformation of the variable to an almost normal distribution. The second method, using the set of transformations known as *Box–Cox transformations*, also attempts to transform the variable to a normally distributed one. The last method is based on the idea of spreading the histogram to expose the details of the variable distribution.

9.6.1 Rank Transformations

The simplest method, which is valid only for continuous variables, is to replace the values with their *ranks.* The ranks are easily calculated using PROC RANK. Let us demonstrate this through an example. The following code creates the dataset T with the variable X having 50 observations.

```
data T;
input X @@;
datalines;
597.38  882.30  285.47  462.36  958.77
339.72  741.44  808.50   37.99  503.89
232.78  568.02  476.57  308.69  420.15
418.09  302.45  630.27  827.49   94.21
141.66  989.73  100.90   25.19  215.44
 73.71  910.78  181.60  632.28  387.84
608.17   85.20  706.30  834.28  435.41
 51.97  993.29  797.13  718.52   52.36
630.37  289.68  814.71  794.85  315.43
314.37  156.37  486.23  430.90  286.54
;
run;
```

We invoke PROC RANK as follows:

```
PROC RANK DATA = T OUT=TRanks;
       VAR X;
   RANKS X_Rank;
RUN;
```

This will create a new dataset, TRanks, and the ranks of the variable X are stored in the new variable X_Rank. The ranks are represented by integer values in ascending order from 1 to 50 (the total number of observations).

We can then transform these ranks into their normal scores (i.e., their probabilities on the normal distribution by spreading these values on the bell curve) using the simple transformation:

$$y_i = \Phi^{-1}\left(\frac{r_i - \frac{3}{8}}{n + \frac{1}{4}}\right), \tag{9.4}$$

with r_i being the rank of observation i, n being the number of observations, and Φ^{-1} being the inverse cumulative normal (PROBIT) function. The transformation will give a new variable that is very close to the normal distribution. The implementation of this transformation is very easy, as shown in the following code.

```
data TNormals;
 set Tranks nobs=nn;
 X_Normal=probit((X_Rank-0.375)/(nn+0.25));
run;
```

Furthermore, PROC RANK itself can provide this transformation directly by using the option NORMAL=BLOM in the PROC RANK statement.

```
PROC RANK DATA = T NORMAL=BLOM OUT=TRanks;
       VAR X;
   RANKS X_Normal;
RUN;
```

The modification will create the variable `X_Normal` directly.

The preceding method looks like a very easy approach to transform any variable to a normally distributed one. However, the difficulty in using this process is that it is possible to do it with the training data. At the time of scoring, the basis of ranking the training data is no longer available. Therefore, we are not able to deduce the equivalent transformation for the scoring values, and any model developed with a variable that was normalized using this method is not usable for scoring purposes.

We recommend that this approach be used only for cases where scores are not to be produced using a dataset other than the one used for training the model. Such cases exist in exploratory data analysis and in cases of cluster analysis when all the data is used to build the model. Another option is to normalize the variable in the entire population before sampling for the mining view.

9.6.2 Box–Cox Transformations

The difficulty in selecting the optimal transformation for a variable lies in the fact that it is not possible to know in advance the best transformation to improve the model's performance. One of the most common transformations is the Box–Cox transformation, which attempts to transform a continuous variable into an *almost* normal distribution. This is achieved by mapping the values using the following set of transformations:

$$y = \begin{cases} x^{\lambda-1}/\lambda & \lambda \neq 0 \\ \log(x) & \lambda = 0 \end{cases} . \tag{9.5}$$

Linear, square root, inverse, quadratic, cubic, and similar transformations are all special cases of Box–Cox formulations. Because of the use of power and log relationships, the values of the variable x must all be positive. In order to handle the possibilities of negative values, we use the following more general form:

$$y = \begin{cases} (x+c)^{\lambda-1}/g\lambda & \lambda \neq 0 \\ \log(x+c)/g & \lambda = 0 \end{cases} . \tag{9.6}$$

The parameter c is used to appropriately offset negative values, and the parameter g is used to scale the resulting values. Parameter g is often taken as the *geometric mean* of the data. The procedure of finding the optimal value of the parameter λ simply iterates through different values, from the range of −3.0 to 3.0 in small steps, until the resulting transformed variable is as close as possible to the normal distribution.

The following implementation of the Box–Cox transformation attempts to maximize the likelihood function by testing different values of the parameter λ until the maximum is attained. The likelihood function is defined in Johnson and Wichern (2001):

$$L(\lambda) = -\frac{n}{2} \ln \left[\frac{1}{n} \sum_{j=1}^{n} (y_j - \bar{y})^2 \right] + (\lambda - 1) \sum_{j=1}^{n} \ln x_j, \qquad (9.7)$$

with y_j being the transformation of the observation x_j according to Equation 9.5. The average term $\bar{y}$ refers to the mean value of the transformed values, or

$$\bar{y} = \frac{1}{n} \sum_{j=1}^{n} y_j. \qquad (9.8)$$

Note that the term in the square brackets in Equation 9.7 is the biased variance of the transformed value. This fact makes the calculation of the likelihood function much easier.

Typically, λ lies between −3.0 and 3.0. The implementation allows the user to specify the width of the search step to adjust the accuracy of the deduced parameter. The code is divided into two macros; the first one calculates the likelihood function $L(\lambda)$, as in Equation 9.7, and the main macro loops over the values of λ and finds the maximum value of the likelihood. Once the maximum value of the likelihood is found, the equivalent λ is used to transform the variable and write the results into an output dataset.

We first present the likelihood function calculation, then the main macro.

Table 9.10 Parameters of macro `CalcLL()`.

Header	`CalcLL(DS,N,L,X,M_LL);`
Parameter	*Description*
`DS`	Input dataset
`N`	Number of observations in `DS`
`L`	Test value of parameter λ
`X`	Name of variable
`M_LL`	Likelihood function defined in Equation 9.7

Step 1

Calculate the transformed values according to Equation 9.5. In doing so, we also calculate the logarithm of the variable to use it in the likelihood function expression.

```
data temp;
 set &ds;
  %if %sysevalf(&L=0) %then %do;
y_temp=log(&x);
```

```
%end;
  %else %do;
y_Temp=((&x.**&L)-1)/&L;
%end;
  lx_Temp=log(&x);
run;
```

Step 2
Use the variance function, VAR, to calculate the first term of the likelihood function and the sum of the Log transform of the raw values of the second term.

```
proc sql noprint;
select var(y_Temp)*(&N-1)/&N,
       sum(lx_Temp) into :vy, :slx from temp;
quit;
%let &M_LL = %sysevalf(
              -&N * %sysfunc(log(&vy))/2
               + (&L-1)*&slx);
```

Step 3
Finish the macro.

```
%mend;
```

The second part of the implementation presents the macro, which loops over the different values of the λ and finds the optimal one.

Table 9.11 Parameters of macro BoxCox().

Header	BoxCox(DSin, XVar, XVarT, DSout, W, M_Lambda, M_LL, M_C)
Parameter	*Description*
DSin	Input dataset
XVar	Name of variable to be transformed
XVarT	Name of variable after transformation
DSout	Output dataset
W	Value used as increment in the evaluation of λ
M_Lambda	Value of optimal λ
M_LL	Value of likelihood function at optimal Λ
M_C	Value of needed offset to make all raw values positive

Step 1

Evaluate the value of the offset parameter `C` needed to make all values positive.

```
proc sql noprint;
 select max(&Xvar) into :C from &dsin;
 select count(&Xvar) into :N from &dsin;
quit;
%if %sysevalf(&C <1e-8) %then
%let C=%sysevalf(-&C + 1e-8);
%else %Let C=0;
```

Step 2

Create a dataset with the transformed positive values.

```
data &dsout;
 set &dsin;
  &Xvar = &Xvar + &C;
run;
```

Step 3

Start the optimization loop of the likelihood function starting with a λ value of 3.0 and decrementing it to a minimum of -3.0 with steps of the given parameter `W`. To guarantee that each step increases the likelihood function, start with a current maximum of a large negative value, -1×10^{50}.

```
%let L_best =3;
%let LL_Max=-1e50;

%do i=1 %to %sysevalf(6/&W+1);
 %let L=%sysevalf(3-(&i-1)*&W);
```

Step 4

In each iteration, evaluate the likelihood function by calling the macro `CalcLL`.

```
 %let LL_Now=;
 %CalcLL(&dsout,&N,&L,&XVar,LL_Now);
```

Step 5

If the current value is higher than the previous maximum, exchange the values until the last increment (end of the loop).

```
 %if %sysevalf(&LL_Now > &LL_max) %then %do;
      %let LL_max=&LL_now;
  %let L_best=&L;
```

```
   %end;
 /* Loop */
%end;
```

Step 6

Now set the values of the optimal λ and the likelihood function.

```
%let &mLL=&LL_Max;
%let &mLambda=&L_best;
%let &mC=&C;
```

Step 7

Finally, apply the optimal transformation and finish the macro.

```
data &dsout;
 set &dsout;
  %if %sysevalf(&L_Best=0) %then %do;
&XVarT=log(&XVar);
%end;
  %else %do;
&XVarT=((&XVar.**&L_best)-1)/&L_best;
%end;
run;
%mend;
```

To demonstrate the use of the macro `BoxCox()`, we generate a dataset with a variable that is *not* normally distributed and apply the transformation on it. The following code generates the dataset `Test` with such properties.

```
data Test;
 do x = 1 to 10 by 0.01;
   y = x*x+exp(x + normal(0));
   output;
 end;
run;
```

By calling the macro `BoxCox()`, we generate a new dataset `Test_BC` with the variable `Ty` being *close to* normal.

```
%let dsin=Test;
%let Xvar=y;
%let XvarT=Ty;
%let dsout=Test_BC;
%let W=0.2;
%let Lambda=;
%let LL=;
%let C=;
```

```
%BoxCox(&dsin, &Xvar, &XVarT, &dsout, &W, Lambda, LL,  C);
%put &Lambda &LL &C;
```

We may examine the distribution of the resulting variable Ty in comparison to the original Y by using the equal width binning macro: BinEqW() (see Section 10.3.1). The results are displayed in Figure 9.1, which shows that the distribution of the transformed values resemble that of a normal distribution more than the original data.

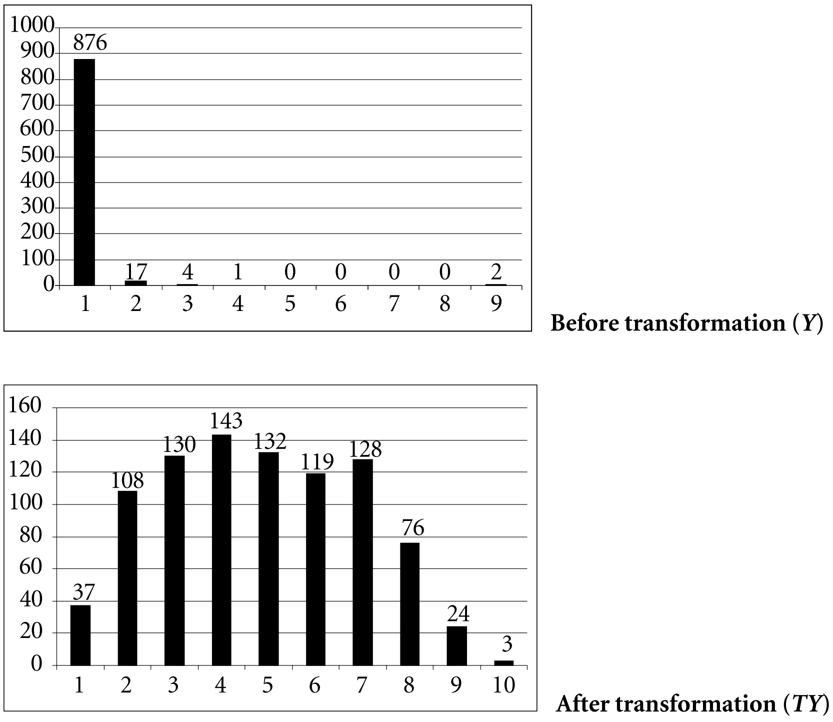

Figure 9.1 Effect of Box–Cox transformation.

9.6.3 Spreading the Histogram

Spreading the histogram can be considered a special case of Box–Cox transformations. As Figure 9.1 shows, the Box–Cox transformation does spread the histogram. However, in other cases we may not be interested in actually converting the distribution to a normal one but merely in spreading it to allow the variable to exhibit some variance and thus be useful for modeling. This can be achieved by using two special cases of Box–Cox transformations.

The first transformation is using the logarithm of the variable (after adding a positive constant if necessary). This has the effect of spreading the distribution to the *right* side of the histogram. For example, using the same TEST dataset as in the case of the Box–Cox example just described, we create a new dataset with the transformed variable Log_Y as follows:

```
DATA Test_T;
 SET Test;
  Log_Y=LOG(y);
RUN;
```

Using equal-width histograms, Figure 9.2 shows the distribution of the variable Y before and after the transformations.

The opposite effect, that is, spreading the histogram to the *left*, can be achieved using a simple power transformation. This is displayed in Figure 9.3, which shows

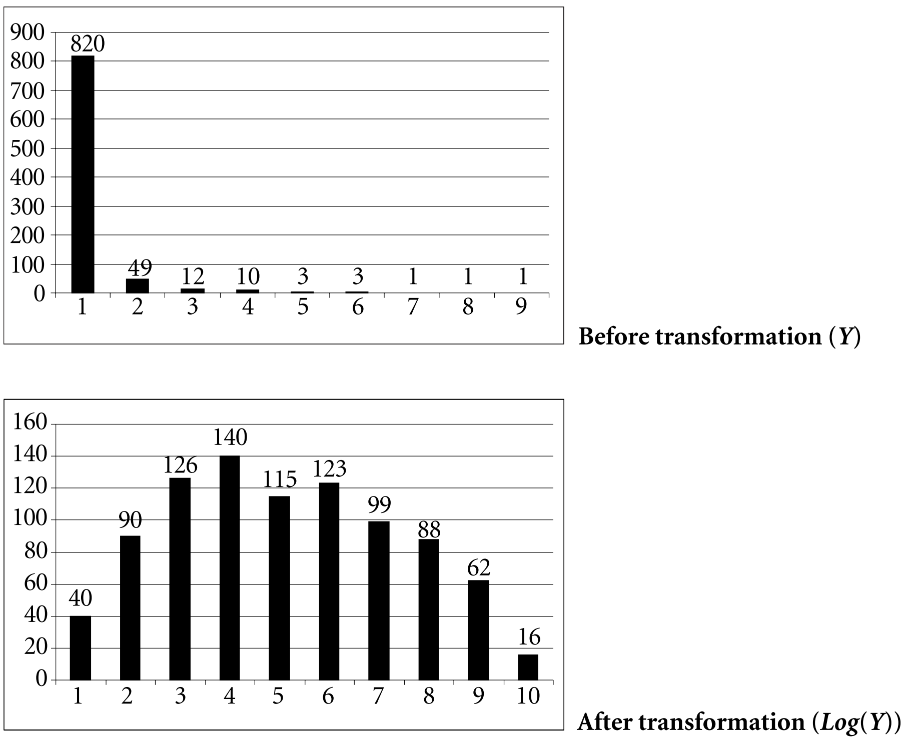

Figure 9.2 Effect of Log transformation.

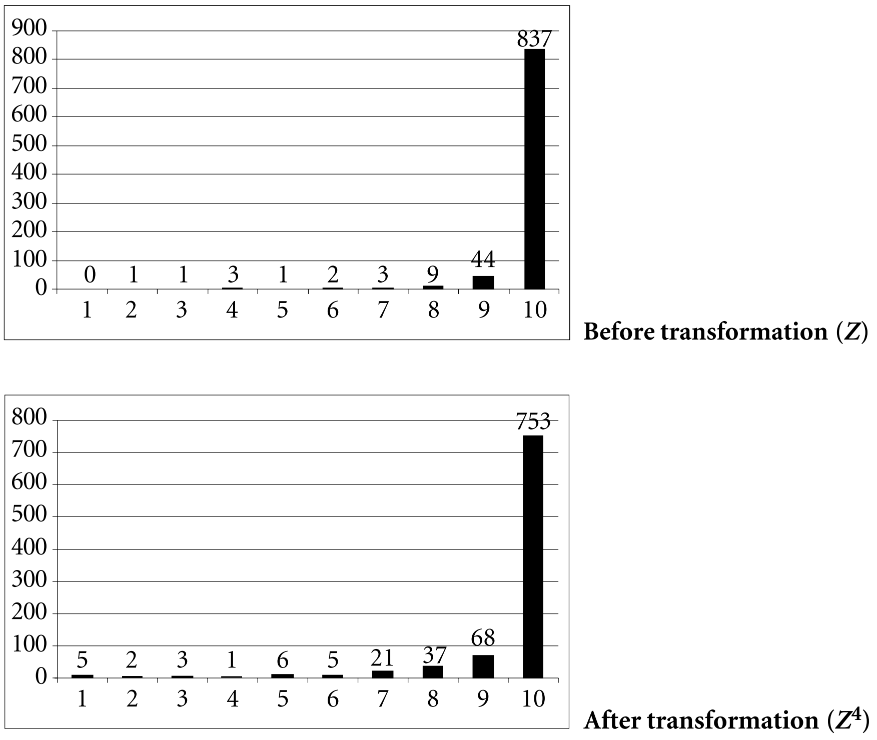

Figure 9.3 Effect of power transformation.

that the power transformation, a fourth order in this case, does spread the histogram, but not as successfully as the Log transformation.

Other similar and popular transformations are summarized in Table 9.12.

Table 9.12 Useful Transformations.

Original scale	*Transformed scale*
Counts, y	$\sqrt{y}$
Proportion, p	$\text{logit}(p) = \frac{1}{2}\log\left(\frac{p}{1+p}\right)$
Correlation, r	Fisher's $z(r) = \frac{1}{2}\log\left(\frac{1+r}{1-r}\right)$