Chapter 6 Integrity Checks

6.1 Introduction

Upon completion of data acquisition and integration, the first task the analyst should do is to make sure that the data passes through the first gate: integrity checks. Integrity checks should reveal errors or inconsistencies in the data because of either sampling or errors in the data acquisition procedures. When such errors are present, data cleansing procedures are necessary before proceeding.

Data warehouses and OLAP applications are currently commonplace. A fundamental step in building these systems is the process of data cleansing. This usually means removing errors and inconsistencies from the data. Although these procedures are beneficial to data mining modeling, they are not usually sufficient. The reason lies in the meaning of the terms *error* and *consistency.* The concepts of data errors and data consistency are heavily used in data warehousing, OLAP, and data mining. However, there is no unique or rigorous definition of these terms. Therefore, before we elaborate about data mining requirements of data cleansing, let us discuss these two concepts: *data errors* and *data consistency.*

One common definition of *errors* is an impossible or unacceptable value in a data field. For example, a gender field should accept values representing "Male," "Female," and "missing" (do not forget the "missing"!). However, due to possible errors in data entry, a value of "Tale" may be found. The data cleansing process in this case is defined as removing such values and replacing them with the most probable acceptable value, which is "Male" in this particular case.

Data *consistency*, on the other hand, is a more subtle concept and, therefore, more difficult to define. It may be defined in terms of consistent values among the different fields, as well as in the values of the same field over time. For example, consider the problem of checking the consistency of the values among the fields of "Transaction Value" and "Product Type" in typical retail data. Ideally, we would have a reasonable match between the two fields such that all records with high transaction values are generated from transactions involving expensive products. It is not reasonable to discover some transactions involving expensive product types with low transaction value.

The second common type of inconsistency within a field often occurs as a result of "redefining" the field representation over different periods. For example, over a certain period of time, the gender field is coded as `Male=1`, `Female=2`, and `Missing=0`, while in a later period the same field is coded as `Male=M`, `Female=F`, and `Missing=[blank]`.

Typical data cleaning procedures targeted to data warehousing–oriented cleansing procedures that address data consistency issues are mostly based on replacing "inconsistent" values with the acceptable values based on a simple set of rules. The rules are extracted from extensive exploration of the data in search of such inconsistencies and errors.

From the point of view of data mining, these checks form a good necessary start. It is therefore usually a blessing when the data mining analyst has access to a cleaned and efficiently managed data warehouse or data repository.

However, additional effort is needed to perform more checks on the data. Simple replacement of errors and inconsistent values would allow the correct summarization and calculation of simple sums and averages, as used by OLAP and reporting applications. Additionally, most data mining algorithms pose their special requirements on the nature of the data to be used in building, validating, and deploying models. For example, many implementations of linear and logistic regression models cannot handle missing values, and they either ignore records with missing values or perform some missing value treatment operation (see Chapter 11 for more details). The point is that, although missing values are legitimate and consistent in terms of business meaning, they may not be acceptable to certain data mining algorithms.

We summarize here some of the most common data cleansing and integrity checks of the simple type:

- Detecting invalid characters and values in different fields. In most cases, these values are replaced by missing values. Invalid values are defined as those that are out of the permitted range. See Section 7.5 for a detailed discussion of outliers.
- Detecting outliers using a simple multiple of the standard deviation or by simply listing and excluding the lowest and highest N values, where N is a user-defined number based on the total number of observations and the understanding of the meaning of the variables. The outliers could also be identified using the *interquartile* ranges.
- Detecting missing and nonnumeric data in a numeric field.
- Detecting data with a missing or invalid row identifier (key).
- Identifying rows with fixed values, such as 999, which is sometimes used by databases and operational systems to code a missing or an invalid value.
- Identifying data with invalid dates. Because of the different standards for writing dates in different countries, dates usually pose their own special problems with data collected from different regions. Invalid dates may arise simply because the

day or the year is not present in the field as a result of data entry problems or errors in loading the data from the database. Fields containing data before the year 2000 may not contain four digits in the year part and thus can cause the known Y2K problem.

- Identifying duplicate records and duplicate IDs.
- Identifying *unique* records. Unique records are those that appear exactly once in the database. For example, in credit card transaction data, if a certain credit card number (the ID) appears only once in the entire database, then most likely there is some error attached to this ID. The reason is that, even if the customer does not activate the card, the customer service center must have attempted to contact the customer to inquire about the reason for not activating it. This would generate at least two entries in the database, one for sending the card and one for each attempted contact.
- Identifying the integrity of date-triggered events. For example, the customer service center in the credit card example should have attempted to contact the customer *after* the date of sending the card.
- Checking sums and ratios when they come from different operational sources. For example, in direct marketing applications, the actual discount value should be equal to the properly calculated value as the purchase amount times the discount rate. This type of check should give allowance for simple numerical round-off errors, such as $0.01 in the case of money calculations.
- Checking on artificial data values entered by the users of operational systems to work around certain known system restrictions. For example, some department stores give special discounts to new subscribers to their charge cards. The salesperson may offer the customer immediate credit approval, thus allowing the customer to use the card with the benefit of a discount rate on the first purchase. To get around the requirement of a credit rating check, the salesperson may assign a temporary credit limit of $1.00 to the customer. However, this value stays in the database.

All these checks represent a limited set of examples where *rule-based procedures* should be employed to check and clean the data. Replacing or removing erroneous or inconsistent records should be a matter of a simple set of IF–THEN–ELSE statements in regular SAS code.

In addition to these checks, which are all based on checking particular fields, we should also check the integrity of (1) sampling and (2) the implicit assumption that the mining view and the scoring view have the same statistical properties. This leads us to consider not only comparing the values of a particular field against a set of rules, but also comparing the values from two or more datasets.

6.2 Comparing Datasets

We now focus our attention on the tests and checks used to ensure that the modeling data (mining view) represent the population (scoring view) and that the training and validation partitions (both parts of the mining view) have similar statistical properties.

There is a wealth of well-established statistical methods to compare two samples of data. Before we discuss some of the details of these methods, let us first create our wish list of the measures of similarity, which would increase our confidence that a sample or two datasets represent the same pool of data.

- *Dataset schema:* Both datasets should contain the same variables using the same data types.
- *Values and statistics:* For each of the continuous variables, both datasets should have similar maximums, minimums, averages, and modes. For each nominal and ordinal variable, both datasets should have the same categories.
- *Distributions:* Variables of both datasets should have similar distributions.
- *Cross-relationships:* All relationships among different variables should be the same in both datasets.

Although these measures are very general in scope, they provide the requirements that form the core of integrity check procedures. Let us discuss each of them in some detail.

One last remark on notation specific to datasets, samples, and population. Since the methods used to compare two datasets are the same whether we are comparing a sample to a population or a sample to a sample, we will not make that distinction anymore. We will simply talk about comparing two datasets.

6.3 Dataset Schema Checks

6.3.1 Dataset Variables

We have borrowed the expression *schema* from the field of relational databases to denote the structure of a dataset in terms of the type definition of its constituent fields. In most applications, modeling data is taken as a sample of the population. This can easily be achieved while preserving the field names and their type definitions. However, in some situations, it is possible that the data used for modeling comes from one source and the data used in scoring or model assessment comes from another. Therefore, one must make sure that the two datasets have the same schema.

Manual comparison of the names and variable types of two datasets is quite an easy task in the case of datasets with few variables. However, for datasets with many variables, it is time consuming and error prone. Also, in environments where a large number of models are being produced and made operational, many of the scoring procedures are automated.

The need to compare the schema of two datasets is rather the exception to the rule. Whenever such a situation arises, however, it becomes a major headache for the analyst. For that reason, we have implemented a simple macro that compares two datasets (A) and (B) to ascertain that *every* variable in B is present in A and is defined using the same data type. This implementation allows the possibility that A is a superset of B in terms of the constituting variables. If a perfect match between A and B were required, the user could invoke the macro twice to test if all the variables in B are in A and vice versa. A positive result of these two checks would then guarantee that both datasets contain the same variable names and are implementing the same type definitions.

Since our implementation is in SAS, we check for only two types of data definition—the two SAS data types, numeric and string. A slight modification of the comparison implementation can include other data types such as time and date. We leave that for you to do as a simple exercise, which can be easily achieved through the use of SAS `FORMAT` and `INFORMAT` specifications.

The presented implementation relies on *SAS Dictionary Tables,* which are a set of read-only tables that are accessible through `PROC SQL` and contain information about all the tables defined in the current SAS session. These tables contain all the required information about the fields and their definitions, which is exactly what we need. An alternative to this SQL-based implementation is to use the capabilities of `PROC CONTENTS` and manipulate the results using standard `DATA` step statements. However, I believe that the SQL-based implementation is easier to understand and modify.

Table 6.1 Parameters of macro `SchCompare()`.

Header	`SchCompare(A,B,Result, I_St);`
Parameter	*Description*
`A`	Dataset A (the base)
`B`	Dataset B
`Result`	A dataset containing the status of the variables of B when they are compared to all the variables in A
`I_St`	A status indicator: 0: B is not part of A; 1: B is a part of A

Step 1

Create a table with the field names and types for datasets A and B.

```
proc sql noprint;
create table TA as
  select name,type from dictionary.columns
  where memname = "%upcase(&A)"
  order by name;
```

```
create table TB as
  select name,type from dictionary.columns
  where memname = "%upcase(&B)"
  order by name;
select count(*) into:N from TB;
run;
quit;
```

Step 2

Loop on each element in TB and attempt to find it in TA. We do that by converting the elements of TB into macro variables.

```
data _Null_;
 set TB;
   call symput('V_'||left(_n_),name);
   call symput('T_'||left(_n_),Type);
run;
```

Step 3

Loop on the N variables in TB and check if they exist in TA and create the entries in a Result dataset.

```
proc sql noprint;
 create table &Result (VarName char(32)
   , VarType char(8)
   , ExistInBase num
   , Comment char(80));
%do i=1 %to &N;
  select count(*) into: Ni from TA  where name="&&V_&i" ;
  %if &Ni eq 0 %then %do ; /* variable does not exist */
    %let Value=0;
    let Comment=Variable does not exist in Base Dataset.;
%goto NextVar;
   %end;
  select count(*) into: Ti from TA
where name="&&V_&i" and Type="&&T_&i";
%if &Ti gt 0 %then %do;  /* perfect match */
%let Value=1;
%let Comment=Variable exits in Base Dataset with the same
data type.;
         %end;
 %else %do; /*different type */
%let Value=0;
%let Comment=Variable exists in Base Dataset
but with the different data type.;
   %end;
```

Step 4

Record the entry in the Result dataset and get the next variable.

```
%NextVar:;
  insert into &Result values ("&&V_&i"
   , "&&T_&i"
   , &Value
   , "&Comment");
%end;
 select min(ExitsInBase) into: I from &Result;
run;
quit;
%let &I_ST=&I;
%mend;
```

Using this macro, we can confirm that dataset B is contained in dataset A with the same variable names and types. However, it does not check on the contents of the fields and their formats. For example, date fields are stored in SAS datasets as numeric fields. It is straightforward to modify the macro to check also on the formats used for fields. This can be accomplished using the field format in the dictionary.columns table.

6.3.2 Variable Types

Since the key question in this set of checks is the variable type, our first task is to identify the type of each variable. In SAS, there are two variable types: string and numeric. It is reasonable to assume that if a field were defined as a string variable, then it would be treated as nominal. There are situations where a string variable does in fact contain only numeric data and should be treated as continuous, or at least as an ordinal variable.

The only reason that a continuous a variable was defined as a string field in the SAS dataset is that it originally came from a database where that field was stored in a text field, or during the transformation process it was decided that using a string field would be easier during data conversion. A simple method to convert a character is to use the function INPUT and store the returned values in a new variable, as in the following example:

```
data SS;
 x='1';output;
 x='a'; output;
run;
data NN;
  set SS;
  y=input(x,best10.);
  drop x;
run;
```

The variable `y` in the dataset `NN` will be of type `numeric` and will have the value of the variable `x` in dataset `SS`. It should be noted, however, that if the string variable contains characters other than numbers, these records will be filled by missing values and an error will be entered in the SAS Log.

We will assume from now on that string variables are in fact nominal variables. On the other hand, numeric fields can represent nominal, ordinal, or continuous variables. This is determined by the analyst based on the understanding of the contents of the data in the field and how to interpret it in terms of the business problem being modeled. For example, a numeric field taking only the values of 0 and 1 could be used as an ordinal variable or as a nominal variable. It could also be used as continuous when intermediate values would also be allowed.

We can summarize this information in a simple rule: The analyst has to specify the use of each numeric variable (nominal, ordinal, or continuous) during modeling.

6.4 Nominal Variables

In the case of nominal variables, we conduct two checks.

1. Check that all the categories present in the first dataset are also present in the second.
2. Check that the proportions of all categories in the two datasets are close or, ideally, identical.

The first check is a simple, though essential, one because ignoring one or more categories of any of the variables may result in unpredictable results, at either the modeling or the deployment stage. For example, some decision tree implementations use the average value of the population for classification and estimation whenever the tree model does not contain a clear method for the treatment of this category. Other tree implementations simply generate a missing score.

To avoid the pitfalls of either case, it is better to make sure that *all* the categories of all the variables in the population can be accounted for and used in the modeling in some way or another. (One way of dealing with some categories is to eliminate them intentionally, but we have to find them and define the method of their elimination or replacement).

The second check is more involved and will determine the degree of similarity between the two datasets with respect to the distribution of one particular variable.

The following two subsections provide the details for the two checks.

6.4.1 Testing the Presence of All Categories

Obtaining the distinct categories of a nominal variable is a relatively simple task. SAS provides several routes to achieving just that. `PROC FREQ` obtains the categories of a variable as well as their frequencies. Alternatively, we can use PROC SQL to issue a SELECT statement and use the keyword DISTINCT to create a list of the distinct categories of a field in a dataset. In the latter case, if the frequencies of the categories are also required to be calculated, we can use the function COUNT and group the selection by the variable in question. The SELECT statement would then be in the following form:

```
SELECT DISTINCT Variable, COUNT(*)
 FROM MyDataSet
 GROUP BY Variable
```

The next task is to make sure that all the categories included in the population do in fact exist in the sample. Note that in the current case we start our checks with the population and not the sample. In the case of two partitions, we have to confirm that they both have the same categories for all nominal variables. The macro in Table 6.2 compares the categories of the variable Var in two datasets Base and Sample to check whether all the categories of this variable in the Base dataset exist in the Sample dataset.

Table 6.2 Parameters of macro CatCompare().

Header	CatCompare (Base, Sample, Var, V_Result, I_St);
Parameter	*Description*
Base	Base dataset
Sample	Sample dataset
Var	The variable for which the categories in the Base dataset are compared to the categories in the Sample dataset
V_result	A dataset containing the comparison summary
I_St	A status indicator set as follows: 0: all categories of Var in Base exist in Sample 1: Not all categories of Var in Base exist in Sample

Step 1

Obtain the categories and their count for both Base and Sample for Var.

```
 Proc SQL noprint;
 create table CatB as select distinct &Var from &Base;
 select count(*) into:NB from CatB;
 create table CatS as select distinct &Var from &Sample;
 select count(*) into:NS from CatS;
 create table &V_Result (Category Char(32)
    , ExistsInSample num
    , Comment char(80));
  run;
  quit;
```

Step 2
Convert all the categories of the dataset Base into macro variables for use in the comparison.

```
data _Null_;
 set CatB;
   call symput('C_'||left(_n_),&Var);
run;
```

Step 3
Loop over the Base categories and find whether all of them exist in Sample.

```
proc SQL ;
%do i=1 %to &NB;
  select count(*) into: Nx
   from CatS where &Var = "%trim(&&C_&i)";
%if &Nx =0 %then %do;
   Insert into &V_Result
    values("%trim(&&C_&i)",0
     ,'Category does not exist in sample.');
          %end;
%if &Nx>0 %then %do;
  Insert into &V_Result
   values("%trim(&&C_&i)",1
          ,'Category exists in sample.');
        %end;
%end;
```

Step 4
Calculate the I_St indicator.

```
select min(ExistsInSample) into: Status from &V_Result;
quit;
%let &I_St = &Status;
```

Step 5
Clean the workspace.

```
proc datasets nodetails  library = work;
delete CatB CatC;
quit;
%mend;
```

This macro can be called as many times as necessary to make sure that the values of all the nominal variables in the population exist in the sample (or in both the training and validation partitions).

6.4.2 Testing the Similarity of Ratios

To test the similarity of the ratios in the sample and those in the population with respect to a particular nominal variable, we will implement the techniques used in the analysis of *contingency tables*. A contingency table is simply the result of cross-tabulating the frequencies of the categories of a variable with respect to different datasets. For example, consider the following two SAS code segments generating two samples: `SampleA` and `SampleB`.

```
data SampleA;
INPUT x $ y @@;
datalines;
A  3  A  2  A  2  B  3  B  2  B  1
B  2  B  3  B  3  C  1  C  1  C  2
C  2  C  2  C  1  A  3  A  2  A  1
A  3  A  2  A  2  B  1  B  2  B  1
B  1  B  3  B  2  C  3  C  1  C  1
C  2  C  1  C  3  A  2  A  2  A  1
A  2  A  2  A  3  B  3  B  2  B  1
B  2  B  3  B  2  C  3  C  1  C  2
C  3  C  2  C  3  A  3  A  3  A  2
A  1  A  1  A  3  B  1  B  3  B  1
;
run;
data SampleB;
INPUT x $ y @@;
datalines;
C  2  C  3  C  1  C  1  C  3  C  1
B  2  B  2  B  2  B  1  B  2  B  2
B  3  B  1  B  3  B  3  B  2  B  1
A  2  A  1  A  3  A  2  A  2  A  1
B  2  B  1  B  3  B  2  B  1  B  1
C  3  C  2  C  3  C  3  C  1  C  2
C  1  C  3  C  2  C  3  C  1  C  3
B  1  B  1  B  2  B  2  B  1  B  1
C  1  C  2  C  1  C  2  C  2  C  1
A  2  A  2  A  2  A  1  A  2  A  3
;
run;
```

We can analyze the frequencies of the categories of the variable X in each of these two datasets using PROC FREQ, as follows.

```
PROC FREQ DATA=Sample A;
 TABLES x;
run;

PROC FREQ DATA=Sample B;
 TABLES x;
run;
```

The results of this analysis are summarized in Table 6.3.

We can rearrange the results of the frequencies of the two datasets in the equivalent but important form as shown in Table 6.4.

Table 6.3 Results of PROC FREQ on SampleA and SampleB datasets.

Sample A			*Sample B*		
X	*Frequency*	*Percent*	*X*	*Frequency*	*Percent*
A	21	35.00	A	12	20.00
B	21	35.00	B	24	40.00
C	18	30.00	C	24	40.00

Table 6.4 SampleA and SampleB frequencies as a contingency table.

X	*Sample A*	*Sample B*
A	21	12
B	21	24
C	18	24

The analysis of contingency Table 6.4 is focused on whether we can confirm that there is a *significant difference* between dataset SampleA and dataset SampleB with respect to the frequencies of the variable X. A detailed description of the mathematics behind this type of analysis is presented in Chapter 13. For now, we will contend with the idea that we are using the Chi-square test to confirm the similarity, or dissimilarity, of the two datasets.

The following macro extracts the comparison variable from the two datasets, adds labels to them, and calls a macro that tests the similarity of the distribution of the comparison variable (see Table 6.5). The heart of the comparison is another macro, ContnAna(), which is described in detail in Section 13.5.

Table 6.5 Parameters of macro `ChiSample()`.

Header	`ChiSample(DS1, DS2, VarC, p, M_Result);`
Parameter	***Description***
`DS1`	First dataset
`DS2`	Second dataset
`VarC`	Nominal variable used for comparison
`p`	*P*-value used to compare to the *p*-value of Chi-square test
`M_Result`	Result of comparison. If two datasets are the same with a probability of *p*, then `M_result` variable will be set to Yes; otherwise, will be set to No.

Step 1

Extract only the comparison variable in both datasets, create a dataset `ID` label variable, and concatenate the two sets in one temporary dataset.

```
DATA Temp_1;
 set &DS1;
 Keep &VarC DSId;
 DSId=1;
RUN;
DATA Temp_2;
 set &DS2;
 Keep &VarC DSId;
 DSId=2;
RUN;
DATA Temp_Both;
 set Temp_1 Temp_2;
RUN;
```

Step 2

Use macro `ContnAna()` to test the association between the dataset `ID` variable and the comparison variable.

```
%ContnAna(temp_Both, DSId, &VarC, Temp_Res);
```

Step 3

Get the *p*-value of the Chi-square test, compare it to the acceptance *p*-value, and set the results message accordingly.

```
data _null_;
 set temp_res;
 if SAS_Name="P_PCHI" then
   call symput ("P_actual", Value);
run;
%if %sysevalf(&P_Actual>=&P) %then %let &M_Result=Yes;
%else %let &M_Result=No;
```

Step 4
Clean the workspace.

```
/* clean workspace */
proc datasets library=work nolist;
 delete Temp_1 Temp_2 Temp_both Temp_res;
quit;
```

To demonstrate the use of this macro, test the similarity of the last two datasets (`SampleA`, `SampleB`).

```
%let DS1=SampleA;
%let DS2=SampleB;
%let VarC=x;
%let p=0.9;
%let Result=;
 %ChiSample(&DS1, &DS2, &VarC, &p, Result);
%put Result of Comparison= &Result;
```

In the preceding code, we used a probability level of 90% as the acceptance level of the similarity of the two datasets. The last `%put` statement will echo the result of comparison to the SAS Log. Run the code and test whether the two datasets are similar! (Answer: No, the two datasets are different).

6.5 Continuous Variables

In the case of continuous variables, there are no distinct categories that can be checked, only a distribution of many values. Therefore, integrity checks attempt to ascertain that the basic distribution measures are similar in the two datasets.

The distribution of a continuous variable can be assessed using several measures, which can be classified into three categories: (1) measures of central tendency, (2) measures of dispersion, and (3) measures of the shape of the distribution. Table 6.6 shows the most common measures used in each category.

The calculation of these values for a continuous variable in a dataset using SAS is straightforward. Both `PROC UNIVARIATE` and `MEANS` calculate all these values and can store them in a new dataset using the statement `OUTPUT`. `PROC MEANS` is in general faster and requires less memory that `PROC UNIVARIATE`. `PROC UNIVARIATE`, on the other hand, provides more detailed tests on the variance, mean, and normality of the data. It also provides the ability to extract and present histograms using different methods.

Table 6.6 Common measures of quantifying the distribution of a continuous variable.

Category	*Common measures*
Central tendency	Mean, mode, median
Dispersion	Variance, range, relative dispersion
Shape of distribution	Maximum, minimum, P1, P5, P10, P50, P90, P95, P99, kurtosis, and skewness

The following code provides a wrapper for PROC UNIVARIATE to facilitate its use. The macro calls PROC UNIVARIATE and stores the values of the preceding measures in variables in an output dataset with the appropriate names.

```
%macro VarUnivar1(ds,varX, StatsDS);
 proc univariate data=&ds noprint;
 var &VarX;
 output out=&StatsDS
  N=Nx
  Mean=Vmean
  min=VMin
  max=VMax
  STD=VStd
  VAR=VVar
  mode=Vmode
  median=Vmedia
  P1=VP1
  P5=VP5
  P10=VP10
  P90=VP90
  P95=VP95
  P99=VP99
;
run;
%mend;
```

Call the macro straightforwardly, as in the following code.

```
%let ds=test;      /* Name of data set */
%let varX=x;       /* Analysis variable */
%let StatsDS=unv;  /* Dataset used to store results */

%VarUnivar1(&ds,&varX, &StatsDS);
```

6.5.1 Comparing Measures from Two Datasets

The macro VarUnivar1() calculates the most common univariate measures of a continuous variable. We can use it to determine these measures for a variable in two different datasets—for example, training and validation partitions. Because the variable in question is continuous, there might be sampling errors, and the two datasets might be very different, the results from these two datasets will almost always be different. But how different? That is the important question that we have to answer.

There are two approaches to assess the difference in the measures of continuous variables. The first one is to use statistical tests, such as Chi-square or *t*-tests, to make a statement about the significance of the differences among calculated values. The second, and simpler, approach is to calculate the percentage of differences between the values from the two datasets. Of course, the second method does not enjoy the sound statistical foundation of the first method, but, in view of the simplicity of implementation, it can be a valuable tool, especially when there are many variables to test. It is particularly useful to compare quantities other than the mean and variance. Although in those cases rigorous statistical measures exist, the interpretation of their results is not easy for nonstatisticians.

In this book, we implement both methods. Table 6.7 shows the plan for comparing the different measures.

Table 6.7 Comparision of measures of continuous variables.

Quantity	*Method of comparison*
Mean	t-test
Variance	χ^2 test
Others	Percentage difference

Note, however: there are well-established methods for computing the statistical distributions of other univariate outputs, such as the mode and the different quantiles (`PROC UNIVARIATE` provides many of these measures). But, to keep the required statistical theory to a minimum, we adopt the simpler approach of using percentage difference.

6.5.2 Comparing the Means, Standard Deviations, and Variances

To compare the previous measures in two datasets, we use the macro `CVLimits()`, which calculates the confidence intervals for the mean, standard deviation, and variance for a continuous variable in a dataset. The results are stored in an output dataset with one row. The macro uses `PROC UNIVARIATE` to calculate the base values and the additional confidence intervals from their original formulas. All the formulas are based on the assumption that the variable in question follows the normal distribution.

Step 1

We first use `PROC UNIVARIATE` to calculate the mean, standard deviation, and variance.

```
proc univariate data=&dsin all noprint;
var &VarX;
```

Table 6.8 Parameters of macro `CVLimits()`.

Header	`CVLimits(DSin, VarX, DSout, Alpha);`
Parameter	*Description*
`DSin`	Input dataset
`VarX`	Calculation variable
`DSout`	Output dataset with the measures
`Alpha`	*p*-value for confidence interval calculations

```
    output out=temp_univ
    Var=&VarX._Var
    STD=&VarX._STD
    mean=&VarX._Mean
    N=n;
run;
```

Step 2

Calculate the confidence interval on the mean, variance, and standard deviation.

```
data &dsout;
set temp_Univ;
/* Alpha */
Alpha=&alpha;
MC=tinv(1-&alpha/2,n-1) * &VarX._Std / sqrt(n);

/* Lower and upper limits on the mean */
&VarX._Mean_U= &VarX._Mean + MC;
&VarX._Mean_L=&VarX._Mean - Mc;

/* Lower and upper limits on the variance */
&VarX._Var_U= &VarX._Var * (n-1)/cinv(1-&alpha/2,n-1);
&VarX._Var_L= &VarX._Var * (n-1)/cinv(  &alpha/2,n-1);

/* Lower and upper limits on Standard Deviation */
&VarX._STD_U = sqrt (&VarX._Var_U);
&VarX._STD_L = sqrt (&VarX._Var_L);

drop MC ;
run;
```

Step 3

Finish the macro.

```
%mend CVLimits;
```

Now that we can calculate the confidence limits on a variable in a dataset, the question is: How can we compare these quantities (mean, standard deviation, and variance) for two datasets? The answer to this question can be discovered by considering three possible scenarios.

- *Case 1:* We have drawn a sample dataset from a population, and we want to make sure that the mean, the standard deviation, and the variance of the sample represent the population. This will be a straightforward application of the results of the macro. We calculate the population mean, standard deviation, and variance and test if they all fall within the limits calculated from the sample.
- *Case 2:* We have two samples—training and validation partitions—drawn from a population. In this case, we repeat the procedures used in Case 1 for each sample.
- *Case 3:* We have two samples—training and validation but we cannot calculate the actual measures for the population, either because of the size of population or because we do not have the data. This is the most difficult case and will require us to resort to a somewhat more complex approach. One simple and efficient approach, although not based on rigorous mathematical foundation, is to calculate the confidence intervals for each measure and then find the *intersection* of these intervals. We then require that the datasets that we denote later as *the population* have their respective values within the intersection of the confidence intervals. This is particularly useful in the case of scoring. We can limit the resulting models as valid only for scoring views with means, standard deviations, and variances that satisfy these conditions.

6.5.3 The Confidence-Level Calculation Assumptions

The macro presented in the previous section calculated the confidence intervals of the variable in question assuming that this variable follows the *normal distribution*.

How realistic is this assumption? In real datasets, very few raw variables have normal distributions. However, this does not mean that we do not use the preceding methods, for two reasons. First, these are the only simple methods we have to compare these variables. Second, there is a statistical theory, called the *central limit theory*, that shows that with a large number of observations, the distribution of continuous variables approaches that of a normal distribution. Therefore, in large datasets we can assume that the variable follows that of a normal distribution without a significant error.

Finally, in Chapter 9 we will present the *Box–Cox* transformations, which can transform variables so that they become almost normally distributed (Johnson and Wichern 2001).

In the preceding discussions, we assumed that we had settled on a reasonable value for the confidence limit. However, the choice of the confidence level parameter (α) is not based on rigorous scientific reasoning. α is commonly set at 0.10, 0.05, or 0.01. These values are only what statisticians and modelers usually use so that they have

90, 95, or 99% confidence, respectively, in the results. The point is, the choice among these three values is somewhat arbitrary.

6.5.4 Comparison of Other Measures

As discussed earlier, there are some rigorous techniques to test other measures using statistical tests, similar to those used for testing the values of the means and the variances. In most cases, we focus most of our attention on the mean and the variance of continuous variables.

Comparing the values obtained for the different measures, such as kurtosis or P95, is a simple task, almost trivial. For example, the following macro describes a simple implementation to compare two such values and provide the difference between then as percentage of the first, second, or average value.

Table 6.9 Parameters of macro `CompareTwo()`.

Header	`CompareTwo(Val1, Val2, Method, Diff);`
Parameter	*Description*
`Val1`	First value
`Val2`	Second value
`Method`	Method of comparison: 1: use `Val1` as the base value 2: use `Val2` as the base value 0: use average of `Val1` and `Val2` as the base value
`Diff`	Percentage difference

We can apply this macro to test the percentage difference between any of these values (Table 6.9). Then we adopt a simple strategy of accepting the hypothesis that the two samples are similar if their respective measures differ no more than, say, 10%.

```
%if &Method=1 %then %do;
%let &diff = %sysevalf(100*(&val2-&val1)/&val1);
%end;
%else %if &Method=2 %then %do;
%let &diff = %sysevalf(100*(&val2-&val1)/&val2);
%end;
%else %if &Method=0 %then %do;
%let &diff = %sysevalf(200*(&val2-&val1)/(&val1+&val2));
%end;

%mend;
```

We should note that in the preceding implementation we allowed use of the average value of the two values as the base for comparison. This is useful when we compare two values from two samples or when the intended base value is zero. The following examples use this macro.

```
/* Examples */
%let x=0.05;
%let y=0.06;
%let d=;
%CompareTwo(&x,&y,1,d);
%put Method 1: difference =&d %;

%CompareTwo(&x,&y,2,d);
%put Method 2: difference =&d %;

%CompareTwo(&x,&y,0,d);
%put Method 0: difference =&d %;
```