Quality Control: How to Analyze and Verify Financial Data

Michelle Duan, Wharton Research Data Services, Philadelphia, PA

Abstract

As SAS® programmers dealing with massive financial data from a variety of data vendors, one major challenge we are facing is effective and efficient quality control. With data sources ranging from plain text files to SQL databases, from a few megabytes to several hundred gigabytes, it is critical for us to make sure the data we provide for our end users is consistent, reliable and up-to-date. This paper will provide an overview of techniques for analyzing and validating financial data products, including checking for uniqueness of key data items such as Company IDs, verifying if missing values or duplicate data entries were introduced during the converting process, comparing and confirming that any data addition, removal or changes between vintages are within normal range, etc. Different data steps and procedures will be used.

Introduction

Wharton Research Data Services (WRDS) is a web-based business data research service from The Wharton School of the University of Pennsylvania. We support faculties and researchers all over the world by supplying data and necessary tools that are needed in their research.

At WRDS, our data is compiled from independent sources that specialize in specific historical data. Some sources include The Center for Research in Security Prices (CRSP), Standard & Poor's Compustat, Thomson Reuters, the NYSE Europnext Trade and Quotes (TAQ), among many others.

We receive data directly from providers in a variety of formats including plain text and SQL tables, and then post the data to our system in SAS datasets. Our procedure often includes but is not limited to the following steps: read source data into SAS, normalize tables, sort and index, create labels, and merge data sets. A number of potential issues can occur during these processes, such as: spurious duplicate records, either from the source data or from our procedures; invalid date values or date ranges; unexpected missing values; unreasonable changes in data contents. Needless to say, data cleanliness and correctness plays a key role in our services. Hence validating our data effectively and efficiently becomes a critical part of our work.

The goal of this paper is to discuss a few techniques that we use to analyze and validate our data. Most of them are not limited to financial data and can be used for other kind of data. Note that we are not building any quantitative models or calculating ratios for financial reports, thus our focus will stay on checking for abnormalities such as unexpected amount of missing values, and comparing between vintages.

The rest of this paper consists of 5 examples. All these examples are based on the following assumptions: Each update is a full refresh instead of concatenation; the latest vintage is dated March 31, 2010, with the prior one December 31, 2009 (the update frequency is per-quarter); the library will be referred to as “lib” and dataset “dsn”; the primary date variable is called “date”, and primary identification variable “ID”.

Example 1 - Duplicate Records

Imprudent data merging and manipulation, redundant records in data source, and even carefully examined sorting procedure, can sometimes produce unwanted duplicate observations. They can not only cost system resources, but also create unexpected, quite often problematic results for research.

Other than analyzing source data with more caution and designing procedures with more comprehensive consideration, we also deploy some quality checks to make sure no duplicate records would exist in our end products.

```
/* Step1: sort with “NODUPREC” option to remove duplicate records
   and generate a temporary data set to hold the result           */
proc sort data=lib.dsn out=_temp_ noduprecs;
       by id date _all_;
run;
```

```
/* Step2: compare numbers of observations before and after the 1st
   step. If different, then there is duplicate records */
data _null_;
    %let didold=%sysfunc(open(lib.dsn));
    %let n_old=%sysfunc(attrn(&didold,nobs));
    %let rc=%sysfunc(close(&didold));

    %let didnew=%sysfunc(open(work._temp_));
    %let n_new=%sysfunc(attrn(&didnew,nobs));
    %let rc=%sysfunc(close(&didnew));

    %if &n_old ^= &n_new %then %do;
       put “%eval(&n_old-&n_new) duplicate records found in %str(dsn)”;
    %end;
run;
```

Although such checks can be avoided at first hand by using NODUPRECS option for SORT procedure, they are necessary in some cases, for example when we do not sort at all. Note that PROC SORT NODUPRECS by id and date only may not eliminate all duplicates, because it will compares all variable values for each observation to those for the previous observation that was written to the output data set, i.e. it will only eliminate consecutive duplicate records. The best way to achieve the result is to use _all_ auto variable in the BY statement.

Example 2 - Multiple records in the same time period

In most time-sensitive data, we should be able to identify one and only one record with ID and date variables within a certain period. For example, we have monthly security data, which has one record for return on the last trading day of each month. Supposedly, given a security ID and a specific month, only one observation should be retrieved. However, we have found some rare cases where more than one record is found in the same month, for the same security ID. Having different date values (e.g. one with last calendar date and the other last trading date) they cannot be identified using the duplicate records checking procedure shown in Example 1. However, the results are the same – they cause confusions and lead to erroneous research result.

For example, watch the bold lines in the following sample data:

ID	DATE	PRC	VOL
101	**20100129**	**11.5**	**200**
101	**20100130**	**12.5**	**200**
102	20100226	42.1	300
103	20100331	25.3	250

In this case, there should be no transaction on Jan 30, 2010 because it was a Saturday. The following procedure will be used to identify such cases.

```
/* create temporary variables year and month */
data temp;
    set lib.dsn;
    year = year(date);
    month = month(date);
run;

/* prepare it for counting with proc sort by ID, year and month */
proc sort;
    by id year month;
run;

/* create a table to hold records where more than one records have the
   same ID, year and month */
proc sql;
     create table multientry as
         select id, year, month, count(*) as ct from temp
         group by id, year, month
         having ct>1;
quit;
```

For problematic dataset, the above code will generate a non-empty dataset and similar note is expected in the log file:

```
NOTE: Table WORK.MULTIENTRY created, with 1 rows and 3 columns.
```

For the sample data mentioned earlier in this section, the dataset MULTIENTRY will have the following record:

```
ID             YEAR           MONTH          CT
101            2010           1              2
```

From this point, we can then take corresponding actions either to consult with data provider or eliminate the incorrect record.

Example 3 - Dates

In most data products, there is at least one essential date variable, such as report date, trade date, or effective date. They are often used as a key variable to retrieve specific data records from a database. For example, for annual company data the key would be the end of fiscal period, and for security data the date when trades take place.

There are several things we want to check for date variables. First, missing values are usually not acceptable for important variables such as date, so we want to make sure there are no missing dates. Second, the latest update should include not only new date values, but also keep the old ones. Thus, during the common period of the two data vintages, there should be about the same number of unique date values. Third, the correspondence between a certain date value and primary ID variables should remain across different vintages, so we expect to have very few to none changes to the number of ID values per period. For example, if there were 100 unique ID values during October 2009 to December 2009 in the old vintage (oldblib), there should be approximately the same number of unique ID values during the same period of time in the new vintage (newlib).

```
proc sql;
/* Count missing DATE and save the numbers into macro variables     */
        select nmiss(date) into: nmissdate_old  from oldlib.dsn;
        select nmiss(date) into: nmissdate_new from newlib.dsn;

/* Count unique values of the following variables: ID, DATE, (total)
OBSERVATIONs, separately */
        select count(unique date), count(*), count(unique ID)
                  into :newdates, :newobs, :newids
                  from newlib.dsn
                  where datevar between "31DEC2009"d and "31MAR2010"d;

/* Calculate range of DATE variable */
        select max(date) into: max_date from newlib.dsn;
        select min(date) into: min_date from newlib.dsn;
quit;
```

The above code calculates the total number of missing date values first, in the old and new vintages respectively. It then calculates the following counts in the new vintage: number of unique date values, number of observation, and number of unique IDs. It will also calculate the range of date variable in the new vintage. We'll use these numbers to generate report similar to the following:

```
There are        0 missing values for DSN:date in the old vintage
There are        0 missing values for DSN:date in the new vintage
In the new vintage, date is within the range of 19251231 and 20100331
During the period from 31MAR2010 to 31MAR2010, there are:
  421217 observations
      64 unique date values
    6681 unique ID values
```

The following code will create two temporary datasets, date_old and date_new, to hold the number of unique date values, number of unique ID values, and number of observations, for each period of time (quarter in this example). It will then compare these numbers and output anything that does not match between two data vintages.

```
proc sql;
     /* get counting numbers for ID, DATE, and observations
        for each quarter, in the prior data vintage */
     create table date_old as
         select year, qtr,
                count(unique date) as ct_date_old,
                count(*) as ct_obs_old,
                count(unique ID) as ct_id_old
         from oldlib.dsn
         where date <= “31DEC2009"d
         group by year, qtr;
     /* get counting numbers for ID, DATE, and observations
        for each quarter, in the new data vintage */
     create table date_new as
         select year, qtr,
                count(unique date) as ct_date_new,
                count(*) as ct_obs_new,
                count(unique ID) as ct_id_new
         from newlib.dsn
         where date <= “31DEC2009"d
         group by year, qtr;
quit;

/* Output rows that do not match between the two vintages, by either only
   existing in one of the two vintages, or having different values */
data result;
     merge dsn_date_old (in=o) dsn_date_new (in=n);
     by year, qtr;
     if (o=1 and n=0) or (o=0 and n=1) or (ct_date_old^=ct_date_new)or
        (ct_obs_old^=ct_obs_new) or (ct_id_old^=ct_id_new)
     then output;
run;
```

With proper rendering, we can write the results into a file which will look like the following:

```
YEAR/QTR   # OF UNIQUE OBS    # OF UNIQUE DATES     # OF UNIQUE ID
             OLD    NEW          OLD     NEW          OLD     NEW
2007 2       63     63           417     416          417     416
2007 3       61     61           422     420          422     420
2008 1       60     60           246     248          246     248
2009 2       50     50           183     185          183     185
```

Example 5 - Value Changes

In a full-refresh update, most data in the common time period usually remain unchanged. Occasionally data providers will make some edits to adjust or correct data, but the changes should stay within a certain range.

This example shows how we check if a record is added or missed in the new vintage, or has the PRC value changed more than 10%.

```
data diff;
    length type $24;
    merge oldlib.dsn (keep   = id date prc
                      rename = (prc=prc_old)
                      in     = in_old )
          newlib.dsn (keep   = id date prc
                      rename = (prc=prc_new)
                      in     = in_new )
    ;
    by id date;
    if in_new=1 and in_old=0 then do;
        type = "New Record";
        output;
    end;
    else if in_new=0 and in_old=1 then do;
        type = "Removed Record";
        output;
    end;
    else if abs(prc_new-prc_old) > prc_old *0.1 then do;
         type = "Diff PRC";
```

```
                        output;
                    end;
                run;
```

Consider the following two pieces of sample data:

```
Prior Data Vintage (oldlib)                   New Data Vintage (newlib)

OBS     ID       DATE           PRC           OBS     ID       DATE           PRC
  1    1021    20090730       14.35             1    1021    20090730       14.35
  2    1021    20090831       20.82             2    1021    20090831       10.82
  3    1021    20090930       15.25             3    1021    20090930       15.25
  4    1021    20091030       18.31             4    1021    20091030       18.31
  5    1021    20091130       19.97             5    1021    20091130       19.97
  6    1021    20091231       16.25             6    1021    20091231       16.25
  7    1084    20090730        4.35             7    1084    20090730        4.35
  8    1084    20090831        4.82             8    1084    20090831        4.82
  9    1084    20090930        6.25             9    1084    20090930        6.25
 10    1084    20091030        8.31            10    1084    20091030        8.31
 11    1084    20091130       10.75            11    1084    20091130        12.3
 12    1084    20091231       11.58            12    1084    20091231       11.58
                                               13    1084    20100129       14.22
```

For the above sample data, the result dataset may look like the following and we can then create report based on it:

```
OBS      TYPE         ID       DATE        PRC_OLD      PRC_NEW
  1    Diff PRC      1021    20090831       20.82        10.82
  2    Diff PRC      1084    20091130       10.75         12.3
  3    New Record    1084    20100129           .        14.22
```

Conclusion

Data validation and quality checks are essential for financial as well as other kinds of databases. Although each data may have its own feature that requires customized quality checks, the ideas behind these validation procedures discussed in this paper are universal and can be applied to many types of numeric data. Using data validation cuts redundant work, saves time, and improves business efficiency and quality.

References

Cody, Ron. 2008. *Cody's Data Cleaning Techniques Using SAS®*. Cary, NC: SAS Institute Inc.

Satchi, Thiru. "Using the Magical Keyword "INTO:" in PROC SQL" *Proceedings of the Twenty-Seventh Annual SAS® Users Group International Conference*. April 2002.
<http://www2.sas.com/proceedings/sugi27/p071-27.pdf>

Wright, Wendi L. "Checking for Duplicates" *Proceedings of the SAS® Global Forum 2007 Conference. April 2007.*
<http://www2.sas.com/proceedings/forum2007/069-2007.pdf>

Acknowledgements

I would like to thank my colleagues at Wharton Research Data Services for their continuous support and insightful advice.

Contact Information

Michelle Duan
Wharton Research Data Services
3733 Spruce St. Suite 216
Philadelphia, PA 19104
Email: mduan@wharton.upenn.edu