Rule-Based Data Validation and Reconciliation of Survey Results

Gunnar Ingle[1], Albert Lee[1]
[1]Summit Consulting LLC, 777 6th St NW, Washington, DC 20001

Abstract

Each year the US Department of Agriculture's National Agricultural Statistics Service (NASS) conducts more than a hundred surveys to understand and enumerate every aspect of agriculture in the United States. The quality of survey responses varies with survey and respondent. Ensuring that survey responses are valid, reliable, and internally consistent is vital to publishing accurate official statistics. NASS is undertaking modernization efforts to detect and edit survey responses through rule validation. These innovations include (1) a review and reconciliation of documented (e.g., written in business rules) and undocumented (e.g., only appearing in programming code) validation specifications, (2) distinguishing validation rules whose errors might be correctable with logical programming code or numeric methods, (3) using numeric methods, such as the Fellegi-Holt algorithm, and R software packages developed by Statistics Netherlands to automate response-level validation checks and error corrections, and (4) flagging instances of automated correction or validation errors for NASS analysts. This paper will describe the processes and procedures used for each step and highlight challenges and solutions to issues commonly encountered.

Key Words: editing, validation rules, automated reconciliation

Disclaimer

The findings and conclusions in this document are those of the authors and should not be construed to represent any official USDA or U.S. government positions or policies.

1. Introduction

Statistical agencies edit and impute survey responses. Thousands of survey responses for each USDA NASS survey are submitted in various forms—on paper, over the telephone, and through the internet. Unsurprisingly, not all responses are complete and internally consistent. As such, NASS uses many editing rules to identify data inconsistencies, imputations to replace missing-but-expected responses, and error rules to cure records. These efforts have one goal: to produce reliable official statistics from complete and internally consistent survey data. Although a considerable amount of logic to ensure consistency of survey responses is incorporated into the telephone and internet survey response platforms, there are still many records that need manual corrections by analysts.

The formal study of editing survey data has a long history, going back at least to Pritzker, Ogus, and Hansen (1965). Winkler (2005) gives an extensive list of articles addressing many aspects of editing and imputation, and de Wal (2009) gives a thorough review of edit rules and procedures. One of the first systematic studies of editing was published by Fellegi and Holt (1976). They proposed three principles:

1. The data in each record should be made to satisfy all edits by changing the fewest possible items of data (fields);
2. As far as possible, the frequency structure of the data file should be maintained;
3. Imputation rules should be derived from the corresponding edit rules without explicit specification.

The USDA NASS edit procedures generally adhere to these principles, although the methods for implementing the principles have greatly expanded since Fellegi and Holt presented them in 1976. Other government agencies and private survey organizations have developed their own editing systems. Systems created by private organizations are generally proprietary, while systems built by government agencies may be publicly available. For example, the U.S. Census Bureau has a public domain software package designed for entering, editing, tabulating, and disseminating census and survey data (U.S. Census Bureau 2024). Statistics Netherlands has written an extensive set of rules for editing (Hoogland, et al. 2011) and developed R packages that implement those rules (van der Loo 2021). We used several of those R packages in creating the editing system for NASS.

1.1 Purpose

This paper describes a system automating corrections from a set of error rules. This automation is executed by a series of R packages that implements the Fellegi-Holt paradigm (see Appendix B0 for more details). The following compelling factors motivate statistical agencies to undertake automated survey corrections.[1]

1.2 Automation saves time

Automating any process can allow for more efficient use of analyst time. Compared to manual interventions, automated error corrections resolve routine errors (much) faster. This substitution allows analysts to focus instead on records that have either residual errors at the end of an automated process or other subtle data quality issues.

1.3 Automation improves consistency

Unlike manual error corrections done by analysts, which are mostly decentralized and undocumented, algorithms create an audit trail, rendering explicit how and which responses are changed and by how much. In addition to transparency, automation also provides edit consistency (e.g., similar errors are corrected similarly). For example, every imputation made on the system for a given type of field (qualitative or quantitative) will be using the same exact algorithm. Consequently, all imputations will be uniform. Finally, an automated system could tabulate correction statistics, pinpointing sections of the survey that frequently require corrections. These survey sections could be candidates for subsequent questionnaire design improvement.

1.4 Automation centralizes rules management

USDA NASS is responsible for administering more than 300 surveys annually. Each of these surveys has its own set of editing rules, embedded in its individual programming files. Each survey comprises tens, if not hundreds, of interrelated programming files.

From the perspective of rules management, this presents two immediate issues. First, rule changes are never one and done. Rather, changes of rules are implemented locally in the programming files in which they reside. If an edit rule appears in 10 separate programming files, a revision would require 10 separate updates. On a related note, edit rule inconsistencies cannot be easily detected, especially across surveys. For example, some deterministic edit rules may take the form of $A+B=C$ in one survey and $A+B \geq C$ in another survey. These subtle inconsistencies are obscured by the decentralized management of rules.

The erosion of institutional knowledge due to staff rotations and retirements further compounds this issue. The large number of rules—and the infrequency of updates, which can be years apart from one version to the next—make it virtually impossible for a complete understanding of the context and rationale of rule updates.

[1] Most edit rules are implemented by computer programs. They are likely to be scattered among a large number of computer programs. Because they are not centrally cataloged, they do not lend themselves to centralized management. Consequently, updating, replacing, and removing edit rules would require analysts remembering where in the computer programs the edit rules reside. This is a nontrivial endeavor for large and complex surveys.

An important objective of this project is to create a centralized and coherent rule catalog object. Rendering the rules explicit allows survey agencies to identify redundant or contradicting rules. As an added benefit, a rule catalog can aid in generalizing rule structures, which further facilitates any updates and future management of rules across surveys that use similar rules.

1.5 Organization

Before delving into the technical details and their implementations, we discuss the organization of this document. This will provide the context of what the rules are, where they come from, and how they fit in the automated system. Types of rules include deterministic edit, routing, warning, and, most importantly for the context of this paper, error rules. These rules come from different sources, both formal (e.g., in computer code) and informal (e.g., institutional habits). Rules are scraped, collected, parsed, and incorporated into R packages.

The next sections discuss how the R packages process the rules, including validating and correcting conflicting and redundant rules. Based on a set of perfected rules, the R packages will identify and implement survey data imputations and corrections.

This paper concludes by reporting the extent to which the automated system has successfully implemented corrections and other performance statistics in a recent application. Other methodological details are collected in the appendixes.

2. USDA survey editing rules

This section describes what comprises an editing rule, including the different types and characteristics of rules. Depending on the sources of rules (i.e., primary programming files and secondary sources, including analysts' habits), this section discusses different methodologies used to gather and incorporate editing rules.

2.1. Sources of rules

The starting point of this project is the complete set of primary programming files, which contains the bulk of editing rules.

2.1.1 Parser

Editing rules are embedded in code, which are scattered among programming files. Because these files are nested and recursive, the invocation of editing rules is based on a series of complex if-then-else logic. To appropriately associate editing rules with their triggering preconditions requires disentangling a nest of computer code. If done by hand, this process is both tedious and error-prone.

A code parser is an automated solution. The parser we built contains a set of Python programs that use the ply.lex package to create tokens based on preset logic to analyze and distribute the logic found in the existing code base, also known as Blaise code, into a comma separated values catalog. Instead of an exhaustive explanation of the parser's function, we describe some of its essential features.

The parser first extracts the code verbatim. This means that no matter how nested or what functions are in the code, the full logic attached to the rules should be extracted. To exemplify this, we will use "else" and "else if" code. When reading the code and seeing that a line starts with "else if," the reader inherently knows that the "if" statements before it had to be false to reach this condition. However, when writing out rules individually, we need to ensure this logic makes it into the catalog. In other words, we must create a stack that holds the conditional statements behind "if" or "else if" statements and place that logic as a "not" statement. The following example illustrates this process.

```
Input:
    IF ( (lUnt) = 'TONS') AND ( (lYUnt) = 'POUNDS') "" THEN
        Production := (Yield * Harvested) / 2000
```

```
        ELSE ""
        Production := Yield * Harvested
        ENDIF

Output:
Rule 1: if ((lUnt) = 'TONS') AND ((lYUnt) = 'POUNDS')
Then Production := (Yield * Harvested) / 2000
Rule 2: if (not((lUnt) = 'TONS') AND ((lYUnt) = 'POUNDS'))
then Production := Yield * Harvested
```

This stack is essential specifically for rules in which the "else" statements have overlapping components within their conditional statement. Something to note is the multiple levels of nesting that can appear within this structure, as an "else if" can be called within an "else" statement and so on. Thus, the created stacks must be able to accurately account for the preconditions and apply the associated actions when the preconditions are met.

The parser also separates logic by types. Breaking apart the editing rules according to the expected action resulting from a line of code involves tasks such as changing a variable, ensuring a variable exists, or displaying an error message to an analyst. More details about the types of rules we encountered and the nature of their application can be found in the *Types of rules* section.

2.1.2 Informal rules

Although it is necessary to extract all the editing rules from the primary programming files, the extracted editing rules by themselves are insufficient to render corrections acceptable to NASS. At the beginning of testing, we discovered that there were additional rules that influence the final edits outside of the primary code. These additional rules specifically address certain administrative variables, such as if the farm is likely to have crops or stocks. To the extent possible, we incorporated these rules, supplementing the primary set of editing rules.

2.1.3 Institutional editing habits

Beyond any rules, formal or otherwise, analysts have developed habits and consensuses correcting certain survey errors by certain methods. Some of these methods involved looking up previously reported values. Others are rules of thumb, relying on regional or local knowledge. These habits are informal and undocumented. To bring these habits to light, we facilitated a series of discussions with NASS analysts, contrasting manual and automated corrections. We distilled their insights into additional edit rules, supplementing code parsed from the primary code files.

2.2. Types of rules

Rules, formal or otherwise, are one of four types:

1. Deterministic editing rules implement deterministic changes—namely, if a condition is met, the data are altered in a predefined way.
2. Routing rules trigger an existence check of a specified variable if a condition is met. Therefore, if a routed variable is missing, it is imputed.
3. Error rules—the focus of this paper—determine if an item is logically inconsistent with other reported items for a given survey response.
4. Warning rules indicate a potential error in the data, i.e., a mathematically possible value that is unlikely to be correct.

Because these rules are interdependent, the order of operations matters. After the data have run through deterministic edit rules and routing rules, the system then conducts imputations, and the data file is finally passed to error resolution. This means that the rules are frequently impacted by the effectiveness of the preceding changes. It should be noted that the error rules are the final step

of the data process to prevent the possibility of recursive corrections, i.e., the corrections of one set of variables trigger error rules for another set of variables.

2.2.1 Deterministic edit rules

Deterministic edit rules, which comprise of most rules in the primary code, fall into one of two categories. One category pertains to survey respondents' responses, e.g., the reported number of acres or the production of a crop variety. The other category pertains to calculated variables, which could be a function of responses such as the sum of all planted acreage.

These rules encompass a range of actions, such as setting a variable to a specific value or setting it to the value of another variable or a calculation. The order of these edits is important to ensure the replication of the existing code and logical development of the variables.

2.2.2 Routing rules

Routing rules identify which variables are expected and if the variable is missing would require an imputation. For example, when a routed variable is missing, a routing rule will set the missing variable to an imputation marker (e.g., -99999999), which will trigger an imputation algorithm replacing the missing variable with an imputed value.

2.2.3 Error rules

Error rules are the focus of this paper. Error rules flash a message when certain illogical conditions (or preconditions) are met. At NASS, these messages start with "ERROR" followed by a number that corresponds to the violated rule. This number describes a generalized rule, such as the amount of planted land cannot be greater than total land for a farm, and thus can be found across files as a rule is repeated for different crops. To clarify which parts of the rule are referenced, we have adopted the term "precondition" for the conditional statement of the error rule while referring to the formula as the validation rule's data requirement.

An example of this is:

```
If ((((Production = RESPONSE) AND (Yield = RESPONSE))
 OR ((Production = NONRESPONSE) AND (Yield = RESPONSE))
 OR ((Production = RESPONSE) AND (Yield = NONRESPONSE)))
 AND (Harvested = RESPONSE) ““) Then

Production = ROUND(Harvested * Yield)

“@R[ERROR 361] CALCULATED YIELD AND REPORTED YIELD NOT EQUAL
 Calculated yield is ^aVYield.”
```

2.2.4 Warning rules

Warning rules are the least serious type of alert an analyst can get. They are triggered for responses that are, on the one hand, mathematically possible but, on the other hand, unlikely to be correct. Like error rules, warning rules evaluate the precondition to see if the rule should be triggered. Because warnings are not critical errors, warning rules currently are excluded from the error-resolution algorithm. That is, warnings could still exist in the absence of critical errors.

3. Validating and correcting rulesets

An important and useful innovation of the R packages is their ability to validate rules themselves. With the number of rules involved in any given survey, some rules could be redundant. Worse yet, some rules could conflict with other rules. The R packages can pare down redundant rules and eliminate rules that conflict with others, resulting in the most parsimonious and internally consistent set of rules suitable for implementation.

At the same time, the R packages are not omnipotent. At its heart, the Fellegi-Holt paradigm relies on some form of linear programming solutions. Depending on the format of the rules, some work better in the automated system than others. In our experience, it works best when the rules are linear and involve continuous variables. Although there are advances toward nonlinear rules as well as categorical and integer variables, the R packages' performance in these scenarios is still far from reliable.

This section presents some of the R packages' capabilities and work-arounds.

3.1 Redundancy and infeasibility

After determining which editing rules are applicable to automated error correction, these functions determine if additional refinements are needed. For example:

Redundant rules:

```
Rule 1: Cropland >= Planted
Rule 2: Planted = Harvested + Other
Rule 3: Cropland >= Harvested + Other
```

Conflicting rules:

```
Rule 1: Cropland >= Planted
Fake Rule 1: Planted > Cropland[2]
```

3.2 Linearization in rules

Although most editing rules are mathematically linear, involving (in)equalities, certain rules are nonlinear. A prime example is a multiplication rule that involves two response variables (e.g., A * B == C). Currently, the R packages handle all linear rules. Without modifications, they will have difficulties generating automated corrections that involve a violation of nonlinear rules. The natural log could be a potential linear transformation of multiplication rules (e.g., log(A) + log(B) == log (C)). There are also other potential workarounds that deserve exploration.

3.2.1 Linearization of rule example:

```
Production = (Harvested * Yield)

Becomes
Log(Production) = Log(Harvested) + Log(Yield)
```

3.3 Scalar replacement in rules

When appropriate, we replace variables with their known value rather than leaving them as a variable. For example, data such as previously reported data or administrative data could be replaced with scalars so the error resolution formula does not attempt to change them. This could simplify rules involving the multiplications of variables. This is done in the same step as linearization.

Scalar replacement example:

```
(Yield <= ERRLIM.YIELD)
```

2 This rule is not found in the USDA NASS existing code base, so an example was created to exemplify the infeasibility check because no current infeasibilities exist.

```
ERRLIM.YIELD = 500
```

Replace the variables you do want changed:

```
Yield <= 500
```

4. Error resolution code

With the refined and corrected rulesets developed, we were able to work on the code that will apply the rules for automated correction. This began the development of an R script we titled "Error Resolution." This script implements the Fellegi-Holt principle of parsimony through the use of the R packages found below. The use of these packages was decided through research and development processes in cooperation with NASS statisticians to find the programs that would be best used in a production environment and could achieve the desired results. This research led us to the developers of the packages, Edwin de Jonge and Mark van der Loo of Statistics Netherlands, who have provided suggestions and supported as we adapted their packages for deployment.

4.1 How error resolution is applied

The automated error-resolution package allows for the determination of which data validation rules are applicable to a record as well as for the identification of any validation rules violated by a record. If there are any rules violated, all rules that were applicable are used to generate a linear model which applies the Fellegi-Holt principle and determines the smallest amount of change to the fewest number of variables needed to fulfill all data validation rules. This then creates output variables that can be used to replace the original invalid variables in the record. There are circumstances in which the data sent to the linear program cannot be solved within a reasonable amount of time, if at all. Running the program code to verify whether the output values adhere to all validation rules involves a second review. If any discrepancies are found, a marker is set on the record to indicate the need for manual intervention.

4.2 Features available

***Parallel processing*—**By using the R package DoParrallel (https://cran.r-project.org/web/packages/doParallel/doParallel.pdf), we are able to use the nodes on a device's CPU to run iterations of data simultaneously, increasing speed.

***Weights*—**The algorithm leaves room for weights to be added to provide more control to the outcomes of the error resolution. Although this has not yet been implemented, we wanted to clarify that this feature exists and will be incorporated in the future.

***Model timeout*—**The linear model can run for hours if given data that are considerably far off from the validation rules. As a result of our efforts to ensure the process can be used in a production environment, we established a 1-minute-per-record timeout on the linear program. If a record times out, it will be given an indicator for analyst intervention. This timeout feature can be adjusted easily by anyone who has access to the process.

5. Use for NASS and interface with analysts

The purpose of this work is to reduce the number and the extent of NASS analysts manual interventions during the data validation process. With a user interface (UI), this tool allows users to monitor every step of the process, from raw data import to data preparations before handing off to the R scripts. These scripts subsequently return the validated dataset along with supplementary information to the UI. Other output from these scripts lives in two files: one file indicates how and for what reasons responses were changed, and another file displays remaining critical errors that scripts failed to resolve. This is all displayed to the analyst, who can then send data back to NASS servers to allow them to either continue their normal process or run the data through the engine again if deemed valuable.

5.1 Performance

Using almost a full survey iteration worth of data, we ran around 30,000 records through the end-to-end process to analyze the tool's effectiveness. This dataset included more than 150 variables and has records from all areas of the United States. Before running through the error resolution process, 78.6% of all records had at least one violated rule. After running the error resolution process, only 43.9% of all records were still considered dirty. This also included an average amount of errors per record going from 2.29 to 1.11 which is a reduction of over one error per record. This was all done with a median processing time of 4.95 seconds per record. We believe that as we implement more features and further refine the ruleset, we can continue to increase the number of records that can be resolved and ease the burden on NASS analysts while ensuring data quality and standards.

6. Conclusion

Rule-based automated data validation and correction has shown promise in a production environment by reasonably resolving errant data while reducing the amount of manual effort required. Through using both documented and undocumented editing processes, we are able to closely replicate the existing editing landscape and apply algorithms to ensure consistency in the edits provided. While initially taxing, the rules development process can become increasingly efficient as the generalized approach accommodates a growing number of surveys.

Appendix A: Data and code examples

Example 1

In Table 1 and Table 2 below, you will find an example of the data changes that will come through the code that will be resolved. This is a very simple example, as it has one rule only applied to the record.

Table 1: Example dataset raw

Year	*Income*	*Spending*	*Deficit*
2023	$10000	$9500	-$500
2023	$30000	$35000	$5000
2023	$42000	$45000	$1500

Data Validation Rule Example : Deficit == Income – Spending

Table 2: Example dataset fixed

Year	*Income*	*Spending*	*Deficit*
2023	$10000	$9500	-$500
2023	$30000	$35000	$5000
2023	$42000	$45000	$3000

Example 2

Figure 1 below is an example of the code dealing with a violated ruleset. When faced with our three example rules and dataset, we can demonstrate the resolution code. By first confronting the data with the rules to determine that there are rules that fail, we can then create our mixed-integer problem object and execute it to create our resolution values. In this case, the R engine knows that Rule 2 is violated, and thus the solution is to simply set Y equal to X.

R packages

DCMODIFY

• CRAN: https://cran.r-project.org/web/packages/dcmodify/vignettes/introduction.html
• Git Repo: https://github.com/data-cleaning/dcmodify
• Key function:

```
Modify()
```

Error Locate

• CRAN: https://cran.r-project.org/web/packages/errorlocate/errorlocate.pdf
• Git Repo: https://github.com/data-cleaning/errorlocate
• Key functions:

```
Inspect_mip()
is_linear()
expand_weights()
```

Validate

• CRAN: https://cran.r-project.org/web/packages/validate/validate.pdf
• Git Repo: https://github.com/data-cleaning/validate

• Key functions:

Confront()
Validator()

```
> library(errorlocate)
> library(dplyr)
> library(validate)
> library(validatetools)
>
>
> rules <- validator(CROPLAND >= PLANTED,
+ PLANTED == HARVESTED + OTHER)
> data <- data.frame(CROPLAND = 410,
+ PLANTED = 500,
+ HARVESTED = 210,
+ OTHER = 200)
> confront(data,rules)
Object of class 'validation'
Call:
 confront(dat = data, x = rules)

Rules confronted: 2
 With fails : 1
 With missings: 0
 Threw warning: 0
 Threw error : 0
>
> mip <- inspect_mip(data, rules)
> res_0 <- mip$execute()
> res_0$values

CROPLAND HARVESTED OTHER PLANTED
 410 210 200 410
```

Figure 1: Resolution code example

Appendix B: Fellegi-Holt

The Fellegi-Holt methodology originates from a 1976 paper titled A Systematic Approach to Automatic Edit and Imputation. This paper illustrated a concept by which a process can detect and correct errant data by comparing them with constraints or rules. Fellegi-Holt can then use the applied relationships between the data to resolve the errors through correcting data with the smallest change possible. This model is influential for its power to face all rules and constraints at the same time and not iteratively as many other validation techniques tend to do. In a production context, Fellegi-Holt is effective through its automation of edits that may be complex and time-consuming manual processes.

We will use the following graph in Figure 2 as an example of the implementation of Fellegi-Holt on a dataset.

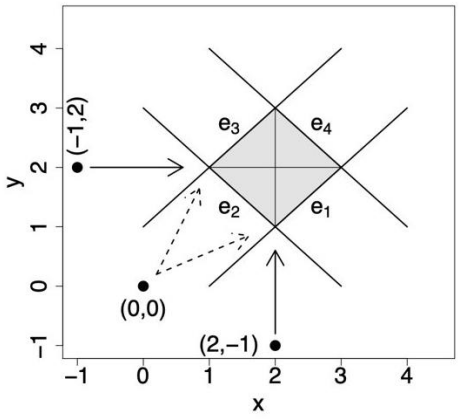

Figure 2: Fellegi-Holt[3]

In this example, the constraints are represented by the lines e1, e2, e3, and e4. These constraints create a geographic area on the graph in which the data are allowed to exist with no rules being violated. As the number of variables and rules increase, the dimensions of the feasibility area increase exponentially. In this simple example, we will use X and Y. Our data point at (2,-1) is out of the allowable range and thus will report an error. Upon finding this error, Fellegi-Holt would conduct the minimum possible change and bring this data point to (2,1), as this changes only one variable and by the smallest amount possible to be within the acceptable range.

The same could be said for our other data points: (-1,2) would get moved to (1,2), as only the X value needs to change for it to be valid. A more realistic and complex example is the data point at (0,0). If variables are whole numbers, they could be moved to either (2,1) or (1,2). For this simple example, it is acceptable to have two positions within reason, but the more rules and variables added, the more likely it becomes that the model would find a single point of acceptable data. Further examples of Fellegi-Holt in action can be found in Appendix A.

3 Statistics Netherlands (2021)

References

Beazley, David. 2022. "PLY (Python Lex-YACC)." October. Accessed June 18, 2024. https://ply.readthedocs.io/en/latest/.

de Wal, T. 2009. "Statistical Data Editing." Chap. 9 in *Sample Surveys: Design, Methods and Applications, Vol. 29A*, by C.R. Rao and D. Pfeffermann, 187–214. Amsterdam: Elsevier B.V. doi:10.1016/50l 69-7 161(08)00009-6.

Fellegi, I.P., and D. Holt. 1976. "A Systematic Approach to Automatic Edit and Imputation." *Journal of the American Statistical Association* 71: 17–35.

Hoogland, J., M. van der Loo, J. Pannekoek, and S. Scholtus. 2011. *Data Editing: Detection and Correction of errors.* The Hague: Statistics Netherlands. https://www.cbs.nl/en-gb/our-services/methods/statistical-methods/throughput/throughput/data-editing-detection-and-correction-of-errors.

Jonge, Edwin de, and Mark P. J. van der Loo. 2021. "Data Validation Infrastructure for R." *Journal of Statistical Software* 97 (10): 1–3. doi:10.18637/jss.v097.i10.

Jonge, Edwin de, and Mark van der Loo. 2023. *dcmodify: Modify Data Using Externally Defined Modification Rules.* https://CRAN.R-project.org/package=dcmodify.

Jonge, Edwin de, and Mark van der Loo. 2022. *errorlocate: Locate Errors with Validation Rules.* https://CRAN.R-project.org/package=errorlocate.

Pritzker, L., Ogus, J., and Hansen, M.H. 1965. "Computer Editing Methods--Some Applications and Results." *Bulletin of the International Statistical Institute, Proceedings of the 35th Session* 395–417.

U.S. Census Bureau. 2024. *Census and Survey Processing System (CSPro).* January 24. Accessed May 28, 2024. https://www.census.gov/data/software/cspro.html.

van der Loo, Mark P.J. and de Jonge, Edwin. 2021. "Data Validation Infrastructure for R." *Journal of Statistical Software* 97 (10): 1–31. doi:10.18637/jss.v097.i10.

Weston, Steve, and Microsoft Corporation. 2022. *doParallel: Foreach Parallel Adaptor for the 'parallel' Package.* https://CRAN.R-project.org/package=doParallel.

Winkler, William. 2005. *Statistical Data Editing References.* Bibliography, Research Triangle Park, NC 27709-4006: National Institute of Statistical Sciences. https://www.niss.org/sites/default/files/WinklerSDE_Ref050203.pdf.