Lessons Learned: Deep Cleaning Procedure Design for Name Variables in the Tennessee CSMD

Molly Golladay[1], MEd and Sarah Nechuta, MPH PhD
([1]Correspondence to: molly.golladay@tn.gov)

Summary: Data cleaning is an important part of entity resolution and data linkage strategies for public health analytics. Prescription drug monitoring programs (PDMPs) have inherent data quality limitations created by multiple-user non-standardized data entry. Identification of patient entities in PDMP data is vital to many of the purposes of these databases. In the Controlled Substances Monitoring Database (CSMD), Tennessee's PDMP, patient names are key identification and linkage variables. However, patient name text fields contain many data source-specific errors and require analytics-driven cleaning approaches to maximize use for entity resolution, data linkage, and statistical analyses. Strategies for cleaning patient names including identifying problematic data, record classification, and appropriate variable parsing strategies, are presented below. The results of the cleaning procedure applied to the CSMD are described in support of our proposed methodology. Examples and statistical programs are provided as a resource for analysts addressing similar data quality issues.

Table of Contents

Introduction 2

Key Elements of General Deep Cleaning Procedures

- I. Record Flagging 5
- II. Data Profiling
  - Keyword Search Techniques for Patient Names 6
  - Keyword Specifics: Prefixes and Suffixes 7
  - Keyword Specifics: Numbers in Name Fields 7
  - Keyword Specifics: Patient Notations 7
- III. Record Parsing 9

Special Cases and Unique Procedures 11

Results

- I. Record Classification 11
- II. Verifying Keyword Methods 12
- III. Frequency Distributions of Cleaned Variables 13

Conclusion 15

Additional Resources 16

Appendices

- A. Primary Variables Used in Cleaning the CSMD 17
- B. Basic Overview of SAS Techniques 19
  - I. Standard Cleaning Functions 19
  - II. Arrays 20
  - III. Search Functions 21
  - IV. Strings and Numerics 24
  - V. Words in Strings 25
  - VI. Concatenation 30
  - VII. A Tranwrd Trick 30
  - VIII. No Words in Strings 31
  - IX. Numerics 32
  - X. Identifying Special Characters 33

Introduction

By the nature of their design, prescription drug monitoring programs (PDMPs) are susceptible to data-entry issues at the dispenser level [1]. In Tennessee, our PDMP is the Controlled Substances Monitoring Database (CSMD). Because the CSMD lacks unique identifiers, patient names are a critical component of entity resolution and linking records to other health outcome datasets. Thus, cleaning the patient name fields is the driving focus of this report, and our goals here are to describe key data issues in using PDMP data as well as proposing analytics-driven cleaning solutions for patient names, using what we have discovered in working with the CSMD. We also note here that while we use the CSMD as the framework for this discussion, **all specific examples presented in this report using potentially identifying data such as names or birth dates are fabricated**. Results such as keyword lists or counts of classification types contain no such identifying data and are presented in their own section.

The process of *data cleaning* can be loosely defined as identifying and repairing corrupted or inaccurate records in a dataset. In general, data cleaning processes are concerned primarily with the *standardization* of data: de-duplication, formatting, appearance, and content. The subject of data cleaning has been of increasing interest as digital datasets have become larger and more complex in not just health fields, but a wide variety of computing and business fields, and while there are some general procedures that are useful across these very diverse applications [2-4], there is also wide agreement that discipline-specific cleaning techniques are also necessary. This is also true when dealing with PDMP-style databases.

Patient name data cleaning comes with a more extensive set of challenges in PDMP datasets. This stems primarily from two places. First, we are dealing with records created by multiple dispensers who enter data in non-standard ways. Second, an important component of entity management in a PDMP is linkage within a single database, as it is possible for multiple records to be associated with a single patient. The complexity of balancing these challenges leads to the main issue: we must go beyond standard data cleaning techniques and cannot rely on a cleaning protocol developed for a different type of data source. Because we are dealing with cleaning text fields with no standardization in data entry, we have applied manual data profiling to identify data issues prior to automating the cleaning process via statistical programming.

To illustrate this point and give some insight into the nature of this process, let's consider an example of problematic data in the patient name fields of the CSMD, shown in Table 1. There are three name fields: FirstName, MiddleName, and LastName. The intention is that the dispenser enters the primary first name into FirstName and the full last name into LastName. MiddleName may be used to hold a middle name or initial but may also be left blank. Thus, the overall expectation is that the majority of both FirstName and LastName are composed of single-word entries. For comparison, we turn to the name entries in the database containing TN death records: **less** than 0.01% of these individuals

Table 1: Multi-Word Variables in the Uncleaned CSMD

One-Word First Names	One-Word Last Names	Frequency Count	Percentage of Total
NO	**NO**	156824	0.81772
NO	YES	1685761	8.78993
YES	**NO**	132487	0.69082
YES	YES	17203245	89.70154

have a multi-word first name (223 out of 347,639) and 0.497% of these individuals have a multi-word last name (1,730 out of 347,639).

We count the number of words in these two fields per record in the CSMD in Table 1. We see that 89.7% of the CSMD is indeed formatted in the expected way. A further 0.69% of records containing single-word first names and multi-word last names may also be expected, as persons with multi-word last names are a valid percentage of CSMD records and this percentage is not substantially inflated compared to the death certificate data. However, the remaining 9.59% of the CSMD has questionable formatting. While we may expect a certain degree of inflation in multi-word first names in the CSMD compared to the death data given that the CSMD is more likely to contain a preferred first name than the death certificate (for example, someone with a legal first name of 'Cora' and a legal middle name of 'Ann' may choose to go by 'Cora Ann' in daily life), it is clear that the percentage is high. Profiling these multi-word FirstName records quickly reveals that pharmacies enter notations (i.e., invalid information) in the name fields, which needs to be identified and removed to be able to use the names accurately for analyses.

While this search is merely a rough estimation of the issue, it illustrates our point: the nature of CSMD data entry necessarily leads to problematic data in patient name fields,[1] which means that attempting to create linkages utilizing these fields between databases or to identify patient-level prescription histories without implementing a deep cleaning procedure may result in missed patient information.

Overview of Objectives

We describe our methodology overall in this report and provide an appendix with statistical programming examples as a resource for general use in other settings. We framed our process for name-field cleaning in the CSMD in terms of two overarching data-driven objectives:

Objective #1 **– Data Standardization**

This is the 'typical' goal of a data cleaning procedure, and we have previously described how there are a variety of resources on how to do this [2-4]. Our focus here is not only to standardize format (special characters, typos, etc.) but content as well, which is what makes manual verification an important part of this process.

Objective #2 **– Record Classification**

A unique challenge in the CSMD is that it contains records of multiple types: individual persons, individual pets, business supplies, and dummy records. We use a flag-based approach to classify records into the appropriate categories.

In our cleaning algorithm, we chose to implement these objectives in tandem, using the process flow shown in Figure 1. Prior to employing this algorithm, we first perform 'basic' cleaning procedures: upcasing, trimming trailing and leading blanks, and removing extra spaces between words . The process flow should be used as the overarching visual for the algorithm, described in detail in the next section.

---

[1] A challenge in addressing this issue is that the assumption *cannot* be made that all multi-word entries are problematic. Part of the complexity of the algorithm derives from the necessity that valid entries containing multi-word patient names must be treated differently than problematic entries containing a single name plus patient notations.

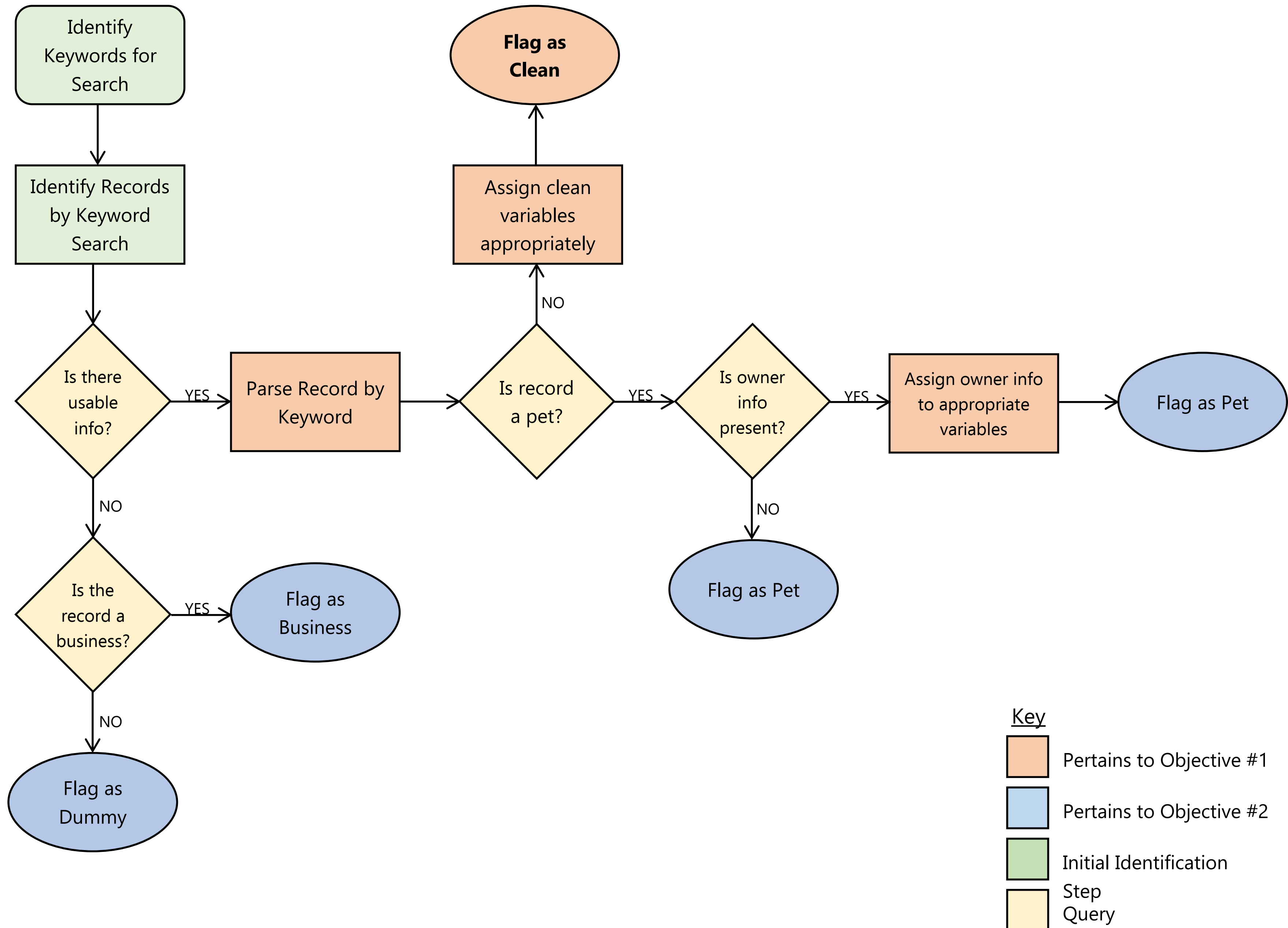
Fig 1: Process Flow for General Deep Cleaning Procedure
Identify Keywords for Search
Identify Records by Keyword Search
Is there usable info?
YES
NO
Parse Record by Keyword
Is record a pet?
NO
Assign clean variables appropriately
Flag as Clean
YES
Is owner info present?
YES
Assign owner info to appropriate variables
Flag as Pet
NO
Flag as Pet
Is the record a business?
YES
Flag as Business
NO
Flag as Dummy
Key
Pertains to Objective #1
Pertains to Objective #2
Initial Identification Step
Query

Key Elements of General Deep Cleaning Procedures

In our cleaning process, we apply three overarching philosophical principles. First, we do not overwrite original variables; all original information in the record must be preserved intact. Second, our procedure is inherently data-driven; making assumptions about data structure without reading the dataset is inefficient. Third, we practice conservative identification; when a record is ambiguous as to classification type, the default assumption is that the record is a person.

I. Record Flagging

In Tennessee, the CSMD tracks both human and non-human records, and one of our cleaning objectives is to classify records as human, animal (pet), business, or dummy records. We endorse the use of record flags in not only this classification process but also in identifying how individual records are being cleaned. This has two main purposes. First, it allows the internal team responsible for data cleaning and management to keep track of which records have already been cleaned while developing their cleaning algorithm. Second, it allows us to track cleaning issues in the database. This is critical for providing feedback to teams developing front-end solutions to data issues; by identifying which dispensers are making which specific errors, training to improve data entry could be implemented for those who have greatest need of these services.

Our procedure makes use of three main CSMD patient record classification flags: *PetFlag*, *BusinessFlag*, and *DummyFlag* (see Appendix A for a full listing of the specific variables used). An 'unflagged' record using this system, then, is a person. Additionally, we use placeholder flags as we are cleaning subsets of the database – for example, when cleaning a record of a suffix or prefix incorrectly placed in FirstName, we use FirstSuf as a flag that this record has been adjusted. These placeholder flags are generally dropped at the end of the cleaning procedure, as they are not particularly useful in the cleaned dataset. The only exception to this is for researchers analyzing the cleaning process itself, the placeholder flags can be used to generate descriptive statistics[2].

II. Data Profiling

The most data-driven piece of the cleaning process is the proper identification of keyword search terms. Because of this, keyword identification is one of the more time-consuming pieces of the procedure as well. As a first step, we recommend approaching the dataset in subsets of roughly 10,000 to 20,000 records at a time. Because reading the dataset is critical in not only identifying terms but also making sure that the algorithm is cleaning as it should, keeping each subset readably small is vital. This isn't the kind of routine where we can look at the first ten or twenty lines and see that it's doing what it should; because each record can be quite unique in its challenges, the whole set must be read.

As a consequence of this subsetting process, we have also observed in our work with the CSMD that it is highly impractical to attempt to write self-contained routines. 'Catching' all of the desired name records on the first try is unreasonable; the database as a whole is just too big and diverse. The data analyst

[2] Further detail on specific flag variables is in Appendix A

must always be prepared either to edit prior routines or to simply add new sections as additional records are detected as the existing cleaning algorithm is implemented.

Keyword Search Techniques for Patient Names

There are several useful initial ways to subset the data to find useful search terms. These include but are not limited to:

- searching for special characters
- searching for entries containing numbers
- searching for multi-word entries
- searching for 'long' entries[3]
- searching for entries that contain no vowels

Many standard sources on data cleaning recommend stripping special characters and numbers out of entries as a matter of course, much as they recommend stripping out multiple spaces between words. We found that this is counterproductive for our patient name fields in the CSMD. Many pet and business records can be identified *using* special characters as a flag, as discussed below, and the numbers are often dates or other identification pieces that may be useful for entity resolution at a later time. While we do eventually remove special characters from the names and move numbers into a separate variable, we initially leave them in the records as searchable terms. Multi-word and 'long' entries are almost all either business records or records that have patient notations, and these searches are particularly useful for identifying further keywords that will be used to parse the records correctly.

A specific and surprisingly useful technique that we identified in the CSMD involved searching for entries which have a second or third word containing no vowels.[4] If the 'word' is longer than two letters, we can conclude that it is not a set of initials and must therefore be an abbreviation (e.g. CHK, MTM, RMR). Since it isn't feasible that we would be aware of every possible abbreviation, running a 'no vowel' search can help us pick up those abbreviations that we weren't aware of previously. We tended to find this search especially useful after parsing the first word of the name into a separate column – more discussion of parsing techniques below – but it's useful as one is finishing a routine and looking for any stray, uncleaned records.

Even though database-specific searches need to be implemented on the 'first pass' of cleaning any large dataset like the CSMD, we can also give some indications of very common keywords found in the CSMD. First, we focus on search words for human records that need parsing. Second, we discuss keywords that are helpful in record classification.

[3] For the CSMD, we defined a 'long' entry as one having more than 20 characters, including spaces

[4] These entries are consistently formatted as *NAME NOTATION*, as in *'MARY MTM'* or *'JOHN RMR.'*

Keyword Specifics: Prefixes and Suffixes

Possibly the most obvious phrases that need to be cleaned from our name variables are common **prefixes** and **suffixes**. These include, but are not limited to:

- Formal titles: Mr, Mrs, Ms, Miss
  - NOTE: Many usages of 'Miss' are in fact pets
- Academic titles and suffixes: Dr, MD, DVM, DDS, PhD, DC
  - NOTE: Some use of these titles can indicate offices instead of persons
- Religious titles: Sister/Sr[5], Brother/Br, Father/Fr, Rev, Rabbi
  - NOTE: 'Sister' is a possible first name, especially in the Southeastern US
- Common suffixes: Jr, Sr, III, IV, etc
  - NOTE: Watch out for all variations of these – we found III, 3rd, and THIRD in the CSMD

Keyword Specifics: Numbers in Name Fields

We have previously discussed the usefulness of searching for **numbers** in our name variables, but here we highlight in particular that the number of digits in the record can be helpful as well. In the CSMD, we found that most entries that only contained one digit were actually a typo, whereas entries with multiple digits usually contained a notation. We used this fact to create appropriate subsets in our cleaning algorithm, and Appendix B contains more detail on the mechanics of this process.

Keyword Specifics: Patient Notations

The final set of important keywords used to parse human records are intended to identify **notations**. Dispensers commonly use FirstName in the CSMD to make record notations, so we see records such as:

- JANE CHECK DOB
- JOHN CID
- MARY MTM 1/3/14

We have decided to retain these notations in a separate variable, but they must be identified first. Our multi-word and long searches have proven useful in finding these terms, and we have also learned two additional techniques to help us find true instances of these notations without having to deal with too much over-identification. First, we use trailing and leading blanks to find terms; for example, we search for '_ID_' instead of just 'ID.' This excludes names like SIDNEY, which happen to contain the letter i beside the letter d. Second, we attempt to identify the most common keyword in several phrases to keep our code efficient. For example, if we look at the following phrases, we see a pattern:

- SEE ID
- CHECK ID
- MUST ID
- ID ONLY

All of these phrases will be identified by searching for 'ID.' So instead of running four separate searches, we only run one and then adjust our parsing algorithm accordingly.

[5] In the CSMD, we found that 'SR' at the beginning of a name tended to be 'Sister,' where 'SR' at the end of a name tended to be 'Senior.' Reading the record can help clarify by providing context.

There are hundreds of possible notation keywords. Here, we list a subset of useful ones, in addition to the previous example:

NOTE	BANNED	FILL	NO
WORKMAN	READ	USE	AKA
DECEASE	PREFER	DOB	SELF
HOSPICE	PAY	TWIN	ONLY

Additionally, there are many pharmacy notations, related either to medication packaging or therapy management that are important. Some of these include:

MTM	LOOSE	DYE	PIP
EZ	PHI	NSC	WC

We also found the following keywords useful in identifying dummy records:

ERROR	VOID	MISTAKE	TEST
DUMMY	DO NOT USE	HOUSE STOCK	PATIENT[6]

Identifying **businesses** can be quite complex, as there are a variety of ways the name can be entered. It is also important to keep in mind that some common business terms, like the word 'CENTER' are also fairly common *last names*, which makes implementing this search more challenging. Again, we present a subset of useful search terms:

HOSPITAL	INSTITUTE	COUNTY	DENTAL
OFFICE	ALLIANCE	PEDIATRIC	METRO
SHELTER	CLINIC	MEDICAL	*State name
RESCUE	SURGERY	OF[7]	

Keyword identification is one of the pieces that makes deep cleaning procedures so complex. The goal of this section has been to give the reader a starting point for running these searches, as well as a set of useful keywords to start with. We end this section with the reminder that it is important to let the data guide these searches; every database will have a unique set of keywords.

[6] When listed as a first name with no other indicators. In the CSMD, a common dummy notation was FirstName: PATIENT, LastName: VOID.

[7] By searching for 'OF' with appropriate leading and trailing spaces, we can find entries like CITY OF, FRIENDS OF, and so on. In the CSMD, this search successfully identified only businesses, but it is possible in another state that this search might also indicate patient notations.

III. Record Parsing

Once a subset of data has been identified by keyword search as being in need of cleaning, the actual *process* of cleaning essentially consists of:

- parsing name information into the appropriate variable(s)
- parsing other retained information, such as notations, into the appropriate variable(s)
- compressing unwanted words or characters

We go back to the first guiding principle stated at the beginning of this section: we strongly recommend retaining all data. To ensure this, our first step is to create a new variable set: UncleanedFirstName, UncleanedMiddleName, UncleanedLastName. This set holds our original information. We have found this to be a good method of tracking our cleaning progress because it may be easier in writing code to overwrite variables for a variety of reasons, and this way our original data remains untouched.

Because of this principle of retaining as much data as possible, the only compressed terms are either non-alphanumeric characters or special cases of suffixes that are written phonetically, such as 'THE 3RD.' In that case, the suffix 'III' would be parsed into the appropriate variable and the original phrasing would not be retained outside of the uncleaned record. In the case of non-alphanumeric characters, we recommend removing all characters except a full stop, parentheses, dashes, and quote marks *before* running any keyword searches[8]. Characters like !@#$ are not useful in the kinds of searches that we run in the CSMD. Periods, parentheses, dashes, and quote marks are useful as search terms, however, and we recommend using PRXPARSE to index the position of any characters that are not either a space or alphabetic. We have found the following uses for these special characters in parsing records:

- periods can be used to separate initialed names such as C.J. if desired
- parentheses can be used to identify nicknames, notations, or pet names
- records beginning with quote marks are almost always pet names
- dashes and quote marks in names can be used either to separate names if desired (MARY-BETH) or can be compressed if names make more sense that way (O'HARA)

When parsing the names, we remind the reader that *placeholder* variables can be invaluable in this process. They are especially useful when an entry contains more than two words or when a subset needs multiple if-statements to sort through its records. Another reason we strongly recommend the usage of placeholder variables is because we have found it easier to clean FirstName and MiddleName simultaneously.

In the vast majority of the CSMD (91.0%), the MiddleName variable is blank. Because of this, it is more efficient to combine the first and middle name variables into a single placeholder and clean that placeholder. Once keyword searches have been used to identify notations, special characters, and

[8] See Appendix B for specific examples, including code, on how to remove characters from a string

prefixes, we use our placeholder variable in combination with an array to parse the names into the correct columns, including sample SAS code here because it makes more sense presented in context[9]:

```
array First(3) $ 50 First1-First3;
placeholder = catx(" ", FirstName, MiddleName);

nameIndex = index(placeholder,' ');
i = 1;

do until(nameIndex <= 1);
    First[i] = substr(placeholder,1,nameIndex-1);
    placeholder = strip(substr(placeholder,nameIndex+1));
    nameIndex = index(placeholder,' ');
    i+1;
end;
```

Placeholders are also especially useful when notations are not uniformly formatted:

- H011 MARY L
- DONNA ROOM211 ANN
- SUSAN 30 DAY SUPPLY

In these cases, we would have identified all of these as records with notations due to the presence of numbers. Since the notations are not in the 'same place' in the string, we would parse the first word, use an if-statement to see which entry contains a number, and put the notation or name in the appropriate place, moving to the next word in the entry and repeating the process until we were out of words.

Parsing variables is a critical part of the data cleaning process. Using placeholder variables helps us not only preserve original data but also optimize our algorithms so that we can manipulate records with differing formats within the same keyword search. Due to the widely varying entry methods in the CSMD, this has proven to be crucial to our success.

[9] Again, see Appendix B for definitions of functions and explanations for why we choose to structure our code in this way

Special Cases and Unique Procedures

We developed methods for two special cases to address unique data features in the CSMD related to identifying non-human records. Due to project resource constraints, these approaches have not been validated and are considered preliminary. Details on these methods for informational and learning purposes are available upon request from the first author.

Results

We now turn our focus from theoretical examples and recommendations to the actual findings of our particular deep cleaning procedure on patient name records in the CSMD. It is worth reminding the reader here that the below counts and percentages are results from the de-identified CSMD, whereas prior examples of potential name fields needing cleaning were fabricated for explanatory purposes. These results are framed in terms of our two objectives: classification and standardization. We also investigate how the recommendations for keyword searches are supported by our results.

I. Record Classification

In Table 2, we see the breakdown of record types contained in the CSMD. We can see that the overall percentage of non-human records is ~0.5%. We have found that animal and business records, due to their highly variable formats, are the most challenging to deal with analytically, and it is useful to be able to remove them from the dataset if desired. It also be of interest to exclude non-human records for specific analyses. It is important to note here that the identification of non-human records was deliberately cautious to reduce the likelihood of incorrectly classifying human records as non-human, meaning that there are likely additional non-human records to be identified. Data cleaning work as regards this classification is an ongoing process.

Table 2: Record Classification in the CSMD After Cleaning

Record Type	N	%
Human	19081768	99.5
Animal	89209	0.465
Business	7137	0.0372
Dummy	203	0.00106

An example of this can be shown by examining the pre-existing SpeciesCode field in the CSMD. SpeciesCode is a non-required variable that dispensers are recommended to code as '1' for any human record and '2' for any non-human record. Only 29.9% of the pet records identified by our cleaning procedure (N = 89209, as per Table 2 above) were previously flagged with SpeciesCode of '2', and only 42.3% of the business records we identified (N = 7137) had a previous SpeciesCode of '2.' If SpeciesCode were utilized properly by all dispensers, it should identify 100% of our non-human records, and indeed make a classification algorithm unnecessary. The lesson to be learned from this is that multiple-user data entry can render coding variables useless if they are not complete – a coding variable such as SpeciesCode should *not* be assumed valid without evaluation of data quality and completeness.

II. Verifying Keyword Methods

For the remainder of our results, we will be presenting counts from the assumed human records only. Non-human records, once classified as such, were not standardized and are thus not included in the subsequent parts of the cleaning procedure.

One of the keyword searches we recommended in the previous section involved special characters. We stated that rather than stripping all such characters out, they could be used for classification and parsing purposes. Table 3 presents a count of the highest frequency special characters; we do not list percentages because some entries have multiple characters, making aggregation less meaningful. As we interpret these results, we must remind ourselves that there are many names that actually have non-alphabetic characters in their proper form. While we must remove these characters, as they make analytics difficult, it's important to bear in mind that high frequencies do not necessarily imply entry errors or extraneous notations. This is supported by the fact that dashes are by far the most common character in our name fields – due to the relative frequency of both hyphenated first and last names, this should not be surprising. We actually see that the top five special character types found are the special characters that we recommend leaving in the name variables to use as search keywords.

Table 3: Special Characters

Character	Frequency
-	141732
( )	31213
" '	21109
0-9	15389
. ,	3133
}	1340
/ \	923
&	380
`	378
_	346
^	286
#	140
;	126
+	98
[	97

In addition to hyphens being part of some proper names, we have also stated that parentheses and numbers often indicate notations. Quotes also show up in names with accents because that is often how the accents translate in a US digital entry format, but can also be useful in pet record classification. Periods can indicate an initialed nickname and commas are often used when a suffix is indicated. For these reasons, we cannot assume that these characters are typos to be stripped out, and the high frequencies in Table 3 support this argument. The rest of Table 3, however, is more indicative of characters resulting either from typos or from dispenser-unique formatting choices. They are not common enough in the database to be useful as search parameters, and our results confirm our initial recommendation of stripping them out of the name variables prior to deep cleaning.

Table 4: Number of Digits Per Entry

Number of Digits	Frequency
1	**10260**
2	**1901**
3	660
4	603
5	**1740**
6	177
7	202
8	402
9	94
10	12
11	4
15	1

Let's focus now on the numbers appearing in name fields in the CSMD. Table 3 showed us the overall count of records containing numbers (N = 15389). Table 4 shows us the *number of digits* present in each of these number-containing entries. We are interested in this count as we have found that low numbers of digits are more indicative of typos to be corrected (eg, DAV1D, J0HN, etc.). We see in Table 4 that single digits are the most common occurrence in name variables containing numbers, being 63.9% of all the entries flagged as containing at least one number prior to cleaning, but because there are multiple sources, we were careful to write our cleaning algorithm so that it accommodates the variety of ways this single digit can appear; in the next section, we will see that single trailing digits can occur as 'notations' as well as single digit typos already shown. The higher number of digits observed in other entries were either ID numbers or part of patient notations and could be treated more uniformly in our cleaning method. Further analysis of the numbers present showed that the most common digit to show up, independent of the total number of digits in a single variable, was the number **1**.

III. Frequency Distributions of Cleaned Variables

First, let's look at the results of the standardization issues that require record parsing: the presence of notations, prefixes/suffixes, or non-alphabetic characters. These results are summarized in Table 5.

The biggest concerns in the human records are non-alphabetic characters and suffixes. This makes logical sense, as some names contain special characters such as apostrophes or dashes when entered correctly, and generational suffixes are another relatively common non-standard name addition.

Table 5: Standardization Issues in the CSMD Name Fields

	N	%
Contains Notations	54955	0.29
Contains Suffix	91735	0.48
Contains Prefix	6622	0.03
Contains Non-Alphabetic	210035	1.10
Other Human Records	18718519	98.10

Let's focus on prefixes and suffixes first. While the CSMD does have pre-established variables for NameSuffix and NamePrefix, because they are not required to be completed, we see a pattern of name fields being used for suffixes and prefixes instead of these variables – Table 6 compares usage of NameSuffix and NamePrefix to the suffixes and prefixes cleaned from the name variables. Cleaning for prefix keywords identified an additional 6658 records containing prefixes – this means that 14.8% of all identified prefixes were only found after cleaning. The pre-established suffix variable is less properly used: cleaning identified an additional 89411 records, meaning that 59.6% of all identified suffixes were only found after cleaning. There were 1186 additional cases where NameSuffix was utilized, but name field cleaning identified a second redundant suffix in a name variable. We recall, however, that the goal of cleaning is not so much to identify suffixes and prefixes as it is to *remove* them from name fields to facilitate the use of the names for analytic purposes.

Table 6: Results of Prefix/Suffix Cleaning in Name Fields

	Before Cleaning	After Cleaning
Has Prefix	38239	44897
Has Suffix	60637	150048

While keeping this objective in mind, we still believe it to be useful to look at the most common prefix and suffix occurrences, to verify the keyword search terms proposed in previous sections. Table 7 looks at the cleaned prefix/suffix distributions in the human records of the CSMD. The percentages calculated are in relation to the total number of cleaned prefixes (N = 6622) and cleaned suffixes (N = 91735) identified by our deep cleaning procedure.

Table 7: Top 3 Prefix/Suffix Frequencies After Cleaning

		N	%
Prefixes	MR	3516	53.10
	MS	1283	19.37
	DR	1149	17.35
Suffixes	JR	59440	64.80
	SR	14030	15.29
	III	9235	10.07

Another keyword search requiring parsing to remove extraneous information from name fields concerns the identification of notations, and in previous sections, we presented several potential search terms for this procedure. Table 8 below shows the most common notations identified by our cleaning procedure and calculates the percentage out of the total number of records with notations (N = 54955, from Table 5). While these are literal counts, we recall that there are also sets of notations that are 'grouped' – identified by a common search term.

Table 8: Most Common Notations Identified By Cleaning

Notation	N	%
CRH	3044	5.51
1	2515	4.56
CC	1794	3.25
2	1634	2.96
MOT	1289	2.33
ID	**1034**	**1.87**
3	904	1.64
SEE ID	**851**	**1.54**
DEL	771	1.40
4	736	1.33
WC	712	1.29

It turns out that the single most common type of notation in the name fields result from searching for variants on 'ID.' If we search these notation records for any entry containing a reference to 'ID,' we find 4728 records, which is 8.60% of all of our notations. The top two 'ID'-containing entries are *ID* and *SEE ID*, shown in bold in Table 8. We also noted in our prior discussion of name fields containing numbers that records containing a single digit are the most common, and we see from Table 8 that some of those single-digit records are single trailing digits, shown as notations rather than typos.

As a final note of interest, we return to Table 1, where we observed that approximately 9.5% of the uncleaned name fields had non-standard formatting in the FirstName variable. In our results, we have shown that record classification and record parsing do not account for anywhere near that percentage of the CSMD records – only 0.5% of the database is classified as non-human based on our procedures, and only about 2% of the database contains errors such as notations, suffixes, or non-alphabetic characters. So what else is going on?

Table 9: Middle Name Length After Cleaning

Name Length	N	%
1	1162338	76.55
4	91896	6.05
5	87580	5.77
3	77268	5.09
6	50448	3.32

We have observed in a previous section that approximately 91% of all records have an empty MiddleName variable. The overwhelming majority of cleaning that needs to take place in the CSMD is simply to look at records that need their name variables parsed properly. To explore this, we looked at all records in the CSMD that have multiple words in FirstName (N = 1,661,491), then we subsetted this to only assumed human records (N = 1,590,488), and then we subsetted further to only the records that did not have notations, suffixes, or non-alphabetic characters, leaving us with 1,518,424 records, which is 91.4% of our original subset. These records *do* need cleaning, but not the deep cleaning we've described in the previous sections. Essentially, these records have middle names that need to be moved into the correct variable, which is confirmed in Table 9 by looking at the length of the 'second word' contained in these FirstName entries – the vast majority of them are just one letter long; they are simply middle initials.

Conclusion

Data cleaning is a key component of entity resolution and data linkage strategies for public health analytics. In Tennessee, we use our PDMP data to conduct analyses to address the opioid crisis by providing answers to questions regarding use of opioids and adverse outcomes, and to identify populations most at risk. These analyses require accurate patient entity resolution and linkage to health outcomes data. Because the CSMD doesn't require systematic collection of unique identifiers, patient names play a key role in entity management. In this report, we've suggested a potential methodology to apply when cleaning patient name fields to maximize use of available information to identify patient

entities, using the results from the CSMD to illustrate the usefulness of this methodology. We hope that these procedures will be helpful to readers who are looking at cleaning their own data.

One limitation to note is that our cleaning process did not systematically address misspellings of names. Due to the highly variable nature of names and potential for atypical spellings and nicknames, this was beyond the scope of this paper. We are implementing processes that account for name misspellings and common variations in our ongoing entity resolution work using algorithms that incorporate fuzzy matching techniques and using SAS DataFlux Data Management Studio.

Acknowledgements

Thank you to Prescription Drug Overdose team members who contributed to discussions, ideas, and attended many meetings on data cleaning strategies for the CSMD patient name and related variables in the summer of 2017, in particular Dr. Ben Tyndall and Dr. Sutapa Mukhopadhyay. Thank you to Dr. Melissa McPheeters for providing comments to improve this report.

Additional Resources

Because of the specific challenges in cleaning the CSMD, the references listed below are essentially the documents we found useful in constructing SAS code. We are not treating this as a formal reference section for this reason, and this is by no means a comprehensive listing of literature on data cleaning.

[1] Prescription Drug Monitoring Program Training and Technical Assistance Center. (April 2015). *Technical Assistance Guide: PDMP Suggested Practices to Ensure Pharmacy Compliance and Improve Data Integrity.* Brandeis University. Retrieved from http://www.pdmpassist.org/pdf/Resources/Pharmacy_compliance_data_quality_TAG__FINAL_20150615_A.pdf

[2] Dusetzina, S., Tyree, S., & Meyer, A. (2014, September 4). Appendix 4.1, Useful SAS Functions and Procedures. In *Linking Data for Health Services Research: A Framework and Instructional Guide.* Rockville, MD, US. Retrieved from https://www.ncbi.nlm.nih.gov/books/NBK253312/

[3] Christen, P. (2012). *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer Publishing Company, Incorporated.

[4] Cody, R. (2017). *Cody's Data Cleaning Techniques Using SAS®, Third Edition*. Cary, NC: SAS Institute, Incorporated.

We also found the following SUGI papers to be extremely useful:

129-29, The Perks of PRX: http://www2.sas.com/proceedings/sugi29/129-29.pdf

247-31, An Introduction to Character Functions: http://www2.sas.com/proceedings/sugi31/247-31.pdf

059-30, A Clever Demonstration of the SAS SUBSTR Function: http://www2.sas.com/proceedings/sugi30/059-30.pdf

Appendices

A. Primary Variables Used in Name Cleaning Procedures for the CSMD

Below, we present a brief summary of the relevant variable fields that exist in the CSMD before and after name cleaning.

Pre-Existing Variables		
Field Name	Description	Role in Cleaning
FirstName	Contains primary first name	Needs to be cleaned; can contain notations, unwanted special characters, or other extra information such as prefix/suffix
MiddleName	Contains middle name or initial *(optional)*	Mostly empty; needs to be cleaned, but can be done in tandem with FirstName; subject to same cleaning issues as above
LastName	Contains full last name	Needs to be cleaned; subject to same cleaning issues as FirstName
DateOfBirth	Contains date of birth in SAS time-date format	Can be used as a secondary 'screening' variable for initially identifying potential non-human records
NamePrefix	Contains preferred prefix *(optional)*	Used only after cleaning for comparison in descriptive analysis
NameSuffix	Contains preferred suffix *(optional)*	
SpeciesCode	'1' for human prescriptions, '2' for non-human prescriptions *(optional)*	
Cleaned Variables Created By Deep Cleaning Procedure		
Field Name	Description	Notes on Cleaning
UncleanedFirstName	Contains contents of original FirstName	Retains original information for comparison or after-cleaning analysis
UncleanedMiddleName	Contains contents of original MiddleName	
UncleanedLastName	Contains contents of original LastName	
First1-First4	Contains cleaned first and middle names; each variable only contains a single word	Cleaned name fields are parsed into separate variables because it's easier to concatenate strings later than to parse them a second time
Last1-Last4	Contains cleaned last names; each variable only contains a single word	
BirthYear	Created from DateOfBirth to contain only the birth year as a numeric variable	Like DateOfBirth, can be useful in identifying initially identifying potential non-human records
PetFlag	'1' for identified pet/animal prescriptions, '0' otherwise	By flagging non-human records as they are identified, they can be removed from the algorithm for efficiency; additionally, flags can be used to
BusinessFlag	'1' for identified business	

	records, ‘0’ otherwise	identify records of interest after cleaning
DummyFlag	‘1’ for voided records, ‘0’ otherwise	
PetName	Contains cleaned pet names from animal prescriptions	Name of pet is less important than name of owner for analytics; if pet name is present in animal prescription, it’s moved here. Owner name (if given) goes into First/Last variables
SpeciesName	If cleaned animal prescription contains species information, it is placed here	Variable exists mostly for descriptives or a potential future analysis; current CSMD projects are not focused on non-human records. Species is preserved in the interest of not deleting any information
CleanNamePrefix	Contains cleaned prefixes	Cleaned prefix/suffix are put in their own fields for comparison to original NamePrefix/NameSuffix after cleaning
CleanNameSuffix	Contains cleaned suffixes	
ChildFirst	Contains child first name information	A few records are worded with mother’s name and child’s name (JANE BABY JOHN); these variables contain the cleaned child’s name. Mother’s name is in First/Last variables
ChildLast	Contains child name information if last name differs from LastName	
PatientNotes	Contains cleaned notations	All non-name information in any name field is placed here; majority of these appear to be notations or some sort of business-specific ID numbers. Retained for potential later use in analytics projects
PrefFlag	‘1’ if cleaning process identified a prefix in a name field, ‘0’ otherwise	Post-cleaning flags simply used to generate internal statistics to see what the algorithm is doing; also used to generate descriptives if desired (see Results section)
SufFlag	‘1’ if cleaning process identified a suffix in a name field, ‘0’ otherwise	
NotesFlag	‘1’ if cleaning process identified a notation in a name field, ‘0’ otherwise	
CharFlag	‘1’ if cleaning process identified a special character in a name field, ‘0’ otherwise	
NumFlag	‘1’ if cleaning process identified a number in a name field, ‘0’ otherwise	

B. A Basic Overview Of SAS Techniques

SAS is an excellent tool for cleaning large datasets and has a number of functions that are extremely useful for text cleaning in particular. There are several websites and SUGI papers that discuss the basic cleaning functions, and a summary of these functions will be presented below. But for the kind of deep cleaning that we propose on PDMP-type databases, there are several non-standard routines that are extremely useful and are not found by a simple documentation search. To illustrate this, let's look at a dataset containing three irregular records:

	var1	var2	var3
1	Susie	Jane Smit	
2	Steve L		Jones
3	Paul++++++++	Te-ddy	Anderson

This will be the main sample that we work from throughout this section.

I. Standard Cleaning Functions

There are several basic functions that any cleaning procedure is going to utilize. These basic functions are detailed in many resources and we simply list them below, noting any potential issues.

- **left**

  Syntax:
  ```
  varName = left(varName);
  ```

- **trim**

  Syntax:
  ```
  varName = trim(varName);
  ```

  NOTE: There is a function called *STRIP* that will simultaneously remove trailing and leading spaces. Depending on your data requirements, it's slightly faster to run. If you know that you don't need these spaces, then I'd recommend running *strip* instead of *left* and *trim*.

- **upcase**

  Syntax:
  ```
  varName = upcase(varName);
  ```

- **compbl**

  Syntax:
  ```
  varName = compbl(varName);
  ```

- **compress**

  Syntax:
  ```
  varName = compress(varName,'$');
  ```
  where *$* is a placeholder for any character or characters that you are looking to remove.

  Some cleaning references recommend using *compress* to strip all special characters from variables at the outset, but we have found in our work on the CSMD that some special characters are useful searching parameters and therefore do not recommend stripping non-alphanumeric characters until later in the cleaning process.

- **tranwrd**

  Syntax:

```
varName = tranwrd(varName, '$', 'R');
```

where $ is a placeholder for the character string you want to change and R is a placeholder for the string you want to replace it with

II. Arrays

Some readers may already be familiar with arrays, but they are so useful in cleaning that we give a brief overview here just in case. An array in SAS is basically a variable that consists of a collection of variables. We can declare an array using the following syntax:

```
array vary(3) var1-var3;
```

We now have a variable called **vary** that contains the three variables var1, var2, and var3.

This is important because an array can run on an internal loop *within* a datastep. From a pure programming perspective, a SAS datastep is a big 'do-loop.' When you run a datastep, what you're telling SAS is, "Look at each record in my file and do these things to it."[10] Do-loops are great when you want to run a repetitive process on a large number of records or variables, and setting up variable arrays allow you to do to variables what SAS is already doing to the records. Here's some code:

```
data cleaning2;
   set foo;

   array vary(3) var1-var3;

   do i=1 to 3;
      vary[i] = left(trim(compbl(upcase(vary[i]))));
      vary[i] = compress(vary[i], '+-');
   end;

   drop i;
run;
```

This code efficiently performs a basic cleaning routine on our example dataset, giving us the following output:

	var1	var2	var3
1	SUSIE	JANE SMIT	
2	STEVE L		JONES
3	PAUL	TEDDY	ANDERSON

Let's break it down. After declaring our array, we set up our do-loop with '`do i=1 to 3;` ', where we're telling SAS to "look at each variable in our array and do the following stuff to it." The first step performs all of our formatting functions in a nested function, removing trailing and leading spaces, as well as internal extra spaces and converting the string to uppercase characters. The second step removes the unwanted special characters, '+' sign and '-'. We always have to tell SAS when the do-loop is over by using an end-statement. For personal preference and space considerations, we drop the variable 'i' at the end of our datastep because we only needed it for the do-loop.

[10] A major drawback to this construction is that during a datastep, SAS can only 'look' at one record at a time. This is problematic if you want to compare record values to each other in a single datastep creation, but that's not necessarily pertinent to data cleaning.

Arrays are not only an incredibly useful tool in basic cleaning, like the above functions, but a powerhouse in all programming applications, whether working in SAS or not, and it is definitely worth the time it takes to master their use.

III. Search Functions

There are really two main ways to search string variables. The preferred method depends on what you're trying to find.

One method is to use the SAS function **find**. Its syntax is:

```
find(varName, 'search string', 'i');
```

The first two parameters are required, and the third is optional. What this function does is scans the variable of interest for whatever string is given. It actually returns a number – the position of the first letter of the identified string – but we're usually not interested in that number because we're using *find* as a condition in an if-statement.

The third parameter, 'i' is an indicator to SAS to make the search case-insensitive. If you're implementing proper cleaning technique, you've already upcased your dataset, in which case you *could* just run:

```
find(varName, 'SEARCH STRING');
```

and it should work fine. ***But!*** If you make a single case mistake, your search won't work. It is our recommendation that you always include the case-insensitive parameter as a guard against errors.

The other search method is to look for explicit equalities:

```
varName = 'SEARCH STRING';
```

There's no way to make this case-insensitive, so it will always be error-prone in that regard. This method is a more strict technique, as it will disregard any variable that isn't an exact match, but depending on what you need, that may be useful. Let's look at some code to see how these work.

```
data searchExample;
  /* Let's use the cleaned dataset formed in our 'Array' section */
  set cleaning2;

  result = find(var1,'sus', 'i');
  result2 = find(var1, 'sie', 'i');

  put '   ' result= result2=;
  put '';

  if find(var2, 'ddy', 'i') then put '   ' var1 var2 var3;
  put '';

  if var3 = 'JONES' then put '   ' var1 var2 var3;
  put '';

  if var3 = 'jones' then put '   Found it!';
  else put '   Not found!';
  put '   End of record';
  put '';
run;
```

Some of these statements are used to make the output log readable. Here it is:

```
  result=1 result2=3
;
;
;
  Not found!
  End of record
;
  result=0 result2=0
;
;
  STEVE L   JONES
;
  Not found!
  End of record
;
  result=0 result2=0
;
  PAUL TEDDY ANDERSON
;
;
  Not found!
  End of record
```

First, we can see that the datastep goes through one record at a time. So the first five lines pertain to the 'Susie' record, the next section to the 'Steve' record, and the last section to the 'Paul' record.

In the Susie record, our code for *result* and *result2* will return the position of the first letter of 'SUS' for *result* and the position of the first letter of 'SIE' for *result2*, which is why we correctly see *result=1* and *result2=3*. Since neither of those strings are present in the last two records, *find* will return zeroes, which is exactly what we see.

In the next section of code, we used a *find* as the conditional in an if-statement. What will happen here is that if *find* returns a value larger than 0, meaning that the string is present in the record, the if-statement will treat it as TRUE[11]. Since the only var2 that has a 'ddy' in it is in the 'Paul' record, where *find* will return a value of 3, that's the only record we see for the output in that line.

The next section has an if-statement with a stronger conditional. var3 must actually equal the string 'JONES,' and we know that in the 'Steve' record, it does. Notice two things here. First, if the data had not been previously formatted using left and trim, this statement would not have worked. ' JONES ' is not the same string as 'JONES.' We can see how this is a problem by looking at the next statement, where SAS searches for the string 'jones.' Since equality can't be case-insensitive, it doesn't find the lowercase 'jones' string anywhere and returns *Not found* for all three records, even though we do have Jones as a last name.

Thus, when using search functions, we need to be aware of how strong we need a search to be and how SAS will behave in each case. While we have presented the most simple search options above, we will see that there are more sophisticated ways to search strings for information as we move forward. There's a useful trick we can use when looking at records by using *find* that bears mentioning here. Let's look at a 'messier' dataset than the example we've been working with:

[11] In computer logic, we generally treat '0' as FALSE and '1' as TRUE – the value for 2+2=4 would be 1 and the value for 2+5=4 would be 0 in this sense – and if-statements run basically as '*if 1 then do*' or '*if 0 then don't do*.' But SAS has a strange little quirk where its if-statements read any number bigger than zero as TRUE. When we use *find* as a conditional, we are exploiting this quirk. But recall that it's SAS-specific. If you try this trick in other scripts or languages, it won't work!

	var1	var2	var3
1	Jane c id		Doe need dob
2	David	M	Dobbler
3	Rocket DO	Jones OWN	Glen

A big part of entity management in PDMP-style databases concerns the fact that dispensers often use the name categories to make patient notations. To clean these, we first have to find the records that have these notes. So here's some code that does that:

```
data identify;
   set foo;

   array variables(3) var1-var3;

   do i=1 to 3;
      if find(variables[i],'id', 'i') then ID_Flag = 1;

      if find(variables[i], 'dob', 'i') then DOB_Flag = 1;
   end;

   if ID_Flag ^= 1 then ID_Flag = 0;
   if DOB_Flag ^= 1 then DOB_Flag = 0;

   drop i;
run;
```

A note concerning arrays before we look at the output: we used an array to run our *find* search so that we could look at each variable in the record at once. We can't use an if-else statement here because our flagged values would be overwritten by later variables. Now, here's the created dataset:

	var1	var2	var3	ID_Flag	DOB_Flag
1	Jane c id		Doe need dob	1	1
2	David	M	Dobbler	1	1
3	Rocket DO	Jones OWN	Glen	0	0

It looks like SAS has made a mistake! It flagged record 2 as having notations when it doesn't.

Remember that *find* is searching without context – the name Dav**id** ends in 'id,' so it will be flagged by our search, as will the name **Dob**bler, which contains 'dob.'

Here's the nifty trick: we can use leading and trailing spaces in our search term to narrow it down. So let's adjust our code a little:

```
data identify;
   set foo;

   array variables(3) var1-var3;

   do i=1 to 3;
      if find(variables[i],' id ', 'i') then ID_Flag = 1;

      if find(variables[i], ' dob ', 'i') then DOB_Flag = 1;
   end;

   if ID_Flag ^= 1 then ID_Flag = 0;
   if DOB_Flag ^= 1 then DOB_Flag = 0;

   drop i;
run;
```

Our new dataset:

	var1	var2	var3	ID_Flag	DOB_Flag
1	Jane c id		Doe need dob	1	1
2	David	M	Dobbler	0	0
3	Rocket DO	Jones OWN	Glen	0	0

When we use spaces around our search term of interest, it can help weed out similar terms. If SAS is looking for _id_, then David won't be flagged. Two drawbacks to note here. The first one is usually less of an issue and has an easy fix: if your search term occurs at the very *end* of a string, then putting a trailing space on your *find* function will exclude that record. But when we say end of the string, we mean the end of the declared string length – notice that our search above still picked up the 'dob' at the end of the first record's var3. This string looks like it's 12 characters long, but when creating the dataset, we preset all of our variable lengths at 20 characters using the following syntax:

```
length var1-var3 $ 20;
informat var1-var3  $char20.;
input var1 var2 var3 $;
```

This way, the trailing spaces on a *find* are accommodated by the fact that each variable has 'extra space' at the end. We recommend that when importing your dataset, you should include explicit length statements for each name category (we used 50 characters) so that you can avoid this issue.

The second issue with our *find* trick is less fixable: using leading spaces automatically excludes instances at the beginning of the term. If we're searching for 'ID,' and the notation is 'ID JANE,' then our *find* function above won't see it. The recommended workaround is to use:

```
find(var1, ' id ', 'i') | (substr(var1,1,3) = 'ID ')
```

A detailed description of the **substr** function is in the next.

IV. Strings and Numerics

For the kind of data cleaning discussed here, we deal mostly with strings. As we do so, it's critical to bear in mind that SAS treats strings and numbers very differently. For many operations, this doesn't matter much, but when we're looking at conditional statements, it can be quite important. Especially when using comparison statements in our logic (equal to, greater than, etc).

We've already discussed one string comparison: 'ABC' = 'abc'. Here, SAS would return a 0, meaning FALSE, because its equality is case-sensitive. It's exactly comparing the whole string along its length, which is why 'ABCDE' = 'ABCDF' would also result in a 0. If you're trying to find exact strings, there's no problem here.

String operators can be modified in SAS using a colon however. If you submit 'ABC' := 'ABCDE', then SAS will return a 1, meaning TRUE!

The colon modifier is used when comparing strings of differing *lengths*. What it essentially does is chops off the end of the longer string to make them equal. So by putting := in the above statement, what SAS is comparing is '**ABC**' = '**ABC**DE'. The first three characters *are* the same, so SAS returns TRUE.

The colon modifier can be used on any comparison operation for strings and it more or less does the same thing each time. We note here that while SAS will use the ASCII collating sequence to report such

statements as 'G' is greater than 'A,' we don't recommend using this in cleaning. It's just too easy to lose track[12].

Plus, there's an easier way to compare parts of strings than mastering colon modifiers, and that is to use **substr**.

```
substr(varName,start,length);
```

Substr looks at the portion of the string specified by the starting positon *start* and length *length*. So we could instead type:

```
varName = 'ABCDE';
substr(varName,1,3) = 'ABC';
```

This logic statement simply asks if the first three characters of *varName* are equal to the string 'ABC'. It does the same thing as the colon statement, but it does so in a more readable way. This method is admittedly slightly less efficient, but we recommend it as a more straightforward process, especially for someone less familiar with programming logic.

The other key difference between strings and numerics that will be of use here concerns the NOT EQUAL comparison. Depending on how your statement is constructed, SAS will sometimes handle a phrase like:

```
varName ~= 'ABCDE';
```

in a strange manner. We have found that it's more effective to use the equivalent statement:

```
~(varName = 'ABCDE');
```

It behaves identically and never gives errors.

One last point regarding conditional comparison – it can be used in SAS to construct *subsets*.

Let's say we want to use a datastep to construct a subset of our example data:

```
data subset;
   set cleaning2;

   if (find(var1,'sus','i') | find(var2,'ddy', 'i'));
run;
```

If we do this, the set *subset* contains:

	var1	var2	var3
1	SUSIE	JANE SMIT	
2	PAUL	TEDDY	ANDERSON

It has identified only the records with var1 containing 'sus' and var2 containing 'ddy.'

Two things about why this works: first of all, we're exploiting the fact that SAS considers 0 a logical FALSE and *any positive number* a logical TRUE. So we're setting up a truth table-type scenario here with our OR statement – any evaluation that isn't F | F will return a 1 for TRUE. So when we look at our if-statement, we're taking our OR statement and conditioning for when that statement is greater than or equal to 1.

Second of all, by not putting anything after the conditional on that line, we're telling SAS to *discard* any record that doesn't meet the conditional. From the documentation:

[12] Want to know more about colon modifier functions? It's all here: http://www2.sas.com/proceedings/sugi26/p073-26.pdf

> The subsetting IF statement can be used only in a DATA step. A subsetting IF statement tests the condition after an observation is read into the Program Data Vector (PDV). If the condition is true, SAS continues processing the current observation. Otherwise, the observation is discarded, and processing continues with the next observation[13].

So the short answer to 'why does this work?' is that SAS is specifically set up to work this way! Bear in mind, however, that many other programming or scripting languages are not structured like this, so if working in something other than SAS, this trick may give an error message.

V. Words in Strings

We must remember that SAS doesn't read characters as text in the same way that humans do. This may seem like an overly obvious observation, but it's worth mentioning for one main reason: SAS doesn't read words. It *seems* like it reads words because of a built-in function called **scan**.

```
scan(varName,num,'$');
```

What *scan* returns is a string containing the characters occurring between the **num**th instance of '$' and the previous one. If that's too abstract to make sense, let's look at an example:

```
data _null_;
   variable = 'THIS IS A SENTENCE, CONTAINING A COMMA, SEVERAL PHRASES, AND A
      PERIOD.';

   result = scan(variable,1," ");
   result2 = scan(variable,3," ");

   result3 = scan(variable,1,",");
   result4 = scan(variable,3,",");

   put result=;
   put result2=;
   put '   ';
   put result3=;
   put result4=;
run;
```

The output log looks like:

```
result=THIS
result2=A

result3=THIS IS A SENTENCE
result4=SEVERAL PHRASES
```

When we use a space as the delimiter, what *scan* effectively does is count words. That's what's happening in *result* and *result2* – we're looking at the first word and the third word. But SAS isn't actually reading words; we've just told it to count spaces. If we use a comma as the delimiter, *scan* counts differently, as we see in *result3* and *result4*.

As long as we keep in mind what scan is actually doing, it's a very useful. One of its most important uses in data cleaning is to 'count words.' Specifically, to isolate multi-word records from single-word records. Let's combine the two examples we've been working with, looking at a properly formatted version:

---

[13] Quoted from: https://support.sas.com/documentation/cdl/en/lrcon/65287/HTML/default/viewer.htm#p04fy20d8il3nfn1ssywe4f25k27.htm

	var1	var2	var3
1	JANE C ID		DOE NEED DOB
2	DAVID	M	DOBBLER
3	ROCKET DO	JONES OWN	GLEN
4	SUSIE	JANE SMITH	
5	STEVE L		JONES
6	PAUL	TEDDY	ANDERSON

In the world of *name* cleaning, we know that multi-word entries are going to be the most problematic. So let's use *scan* to flag multi-word *var3* entries:

```
data multiWord;
   set cleaned;

   if ~(scan(var3,1," ") = var3) then multiFlag = 1;
   else multiFlag = 0;
run;
```

The resulting set looks like:

	var1	var2	var3	multiFlag
1	JANE C ID		DOE NEED DOB	1
2	DAVID	M	DOBBLER	0
3	ROCKET DO	JONES OWN	GLEN	0
4	SUSIE	JANE SMITH		0
5	STEVE L		JONES	0
6	PAUL	TEDDY	ANDERSON	0

What we did was simply looked for the records where the *first word* of var3 was equal to the whole var3 entry and flagged the entries where that was not the case. By far and away, this is the single most useful application of *scan* in name cleaning that we have discovered to date.

Some literature recommends using *scan* to 'break up' names. If we look at record 5, we see that the middle initial 'L' in var1 really belongs in var2. We could move it:

```
data moveName;
   set cleaned;

   if find(var1,'steve', 'i') then do;
      var2 = scan(var1,2," ");
      var1 = scan(var1,1," ");
   end;
run;
```

Gives us as output:

	var1	var2	var3
1	JANE C ID		DOE NEED DOB
2	DAVID	M	DOBBLER
3	ROCKET DO	JONES OWN	GLEN
4	SUSIE	JANE SMITH	
5	STEVE	L	JONES
6	PAUL	TEDDY	ANDERSON

And that's fine – as long as the *identical* mistake has been made in each entry. Consider records 1 and 3 as well. What if we applied the same cleaning routine here to these without reading them – if we just thought that all var1 multi-word entries must be an accidental middle name?

```
data moveName;
   set cleaned;

   if ~(scan(var1,1," ") = var1) then do;
      var2 = scan(var1,2," ");
      var1 = scan(var1,1," ");
   end;
run;
```

Our result would be:

	var1	var2	var3
1	JANE	C	DOE NEED DOB
2	DAVID	M	DOBBLER
3	ROCKET	DO	GLEN
4	SUSIE	JANE SMITH	
5	STEVE	L	JONES
6	PAUL	TEDDY	ANDERSON

Well, this is terrible. Record 5 was fixed correctly, but in both records 1 and 3, we've lost information! Record 1 had a notation that has been lost, and record 3 is clearly a pet if we read it. The main point here – unrelated to SAS tricks – is that name cleaning is a delicate process that must be undertaken carefully. But we introduce this issue here to show that there is a better way to parse multi-word entries than using *scan*.

To discuss this 'better way,' we need to introduce a new function, **index**.

Syntax:

```
number = index(varName,'search string');
```

Index returns the position of the first character of the search string found in the variable varName. If the search string is not found, a zero will be returned.

Let's see how this works:

```
data moveName;
   set cleaned;

   if ~(scan(var1,1," ") = var1) then do;
      number = index(var1, " ");

      placeholder1 = substr(var1,1,number-1);
      placeholder2 = substr(var1,number+1);
   end;
run;
```

Our resulting dataset looks like:

	var1	var2	var3	number	placeholder1	placeholder2
1	JANE C ID		DOE NEED DOB	5	JANE	C ID
2	DAVID	M	DOBBLER	.		
3	ROCKET DO	JONES OWN	GLEN	7	ROCKET	DO
4	SUSIE	JANE SMITH		.		
5	STEVE L		JONES	6	STEVE	L
6	PAUL	TEDDY	ANDERSON	.		

We chose not to overwrite our previous var1 and var2 so that we can see exactly what's going on. This code only runs for multi-word var1 entries – notice that records 2, 4, and 6 are missing values for our new

variables. It indexes the position of the first *space* in each multi-word entry – *number* is 5 for record 1 because the fifth character in the string 'JANE C ID' is a space. We then use substr to cut the entry into the characters before the space (i.e., the first word) and then everything after the space. This way, the notation for record 1 is fully preserved, even though it's still two words long.

The goal, then, is to write a routine using these tricks that essentially turns our non-standardized data into this:

	var1	var2	var3	pet_name	notations
1	JANE		DOE		C ID NEED DOB
2	DAVID	M	DOBBLER		
3	GLEN		JONES	ROCKET	
4	SUSIE	JANE	SMITH		
5	STEVE	L	JONES		
6	PAUL	TEDDY	ANDERSON		

The code that does this is:

```
data moveName;
   set cleaned;

   if ~(scan(var1,1," ") = var1) then do;
      number = index(var1, " ");

      placeholder1 = substr(var1,1,number-1);
      placeholder2 = substr(var1,number+1);

      var1 = placeholder1;

      if find(placeholder2, 'do', 'i') then do;
         var1 = var3;
         var3 = scan(var2,1," ");
         var2 = " ";

         pet_name = placeholder1;
      end;
      else if ~(scan(placeholder2,1," ")=placeholder2) then notations =
         placeholder2;
      else if missing(var2) then var2 = placeholder2;
   end;

   if ~(scan(var2,1," ") = var2) & missing(var3) then do;
      number = index(var2, " ");

      var3 = substr(var2,number+1);
      var2 = substr(var2,1,number-1);
   end;

   if ~(scan(var3,1," ") = var3) then do;
      number = index(var3, " ");

      notations = catx(" ",notations,substr(var3,number+1));
      var3 = substr(var3,1,number-1);
   end;

   drop number placeholder1 placeholder2;
run;
```

Almost every routine used here has previously been discussed in this document. Put together in this context, we can see how they form the foundation of text cleaning.

VI. Concatenation

The one function used in the previous section that we haven't discussed yet is a *concatenation* function. What concatenation functions do is put strings together. SAS has three main concatenation functions, depending on what you need:

- **cat**

  Syntax:
  ```
  varName = cat('string1','string2');

  varName = 'string1' || 'string2';
  ```
  Either phrase does the exact same thing – combines the two (or more) strings without removing anything at all.

- **cats**

  Syntax:
  ```
  varName = cats('string1','string2');
  ```
  This function combines the two (or more) strings while removing any trailing or leading spaces.

- **catx**

  Syntax:
  ```
  varName = catx('$','string1','string2');
  ```
  This function combines the two (or more) strings, removes trailing and leading spaces, and adds the separator '$.' We usually use a space as the separator for the CSMD.

VII. A Tranwrd Trick

Probably the most obvious application of *tranwrd* is to fix typos or standardize phrasing that you're trying to use in a search:

```
FirstName = tranwrd(FirstName, "K 9", "CANINE");
FirstName = tranwrd(FirstName, " CID ", " SEE ID ");
```

But there's another useful thing we can do with this function.

A relatively common indication in a pet prescription record is the presence of some form of the word 'OWNER.' But this word doesn't contribute anything to the record – it's not a formal title or a name – so there's little point in retaining it. At the discretion of the analytics team, there may be more words like this.

You *could* use *compress* in a step-by-step process to remove these words, but *compress* is a tricky function, and barring unwanted special characters like @$%, etc., it is not particularly recommended as a function to run on the overall set. What we recommend doing is once you've used words like 'OWNER' in a search to isolate records, simply run something like:

```
varName = tranwrd(varName, "OWNER", "$");
```

In fact, make sure you change every word to the *same* special character. That way, at the end of your datastep, you only need one line to get rid of them all simultaneously:

```
varName = compress(varName, "$");
```

This way you know that nothing important has been removed, because you're using a character that doesn't show up anywhere else, and you can simply add *tranwrd* statements as they become necessary after search functions are done. Using *compress* too early means that you've lost the searching ability.

VIII. No Words in Strings

When searching multi-word records, we rely on the presence of spaces between words quite heavily. What if there are no spaces?

Consider the following example:

	var1	var2	var3
1	CHERYLMTM		SMITH
2	FREDERICKPHI		JONES
3	PATRICKCHK		SILVER
4	ANNADBL		PALMER

These are all common pharmacy notations that have been frequently observed in the CSMD. Some of them may end up being more useful than others, but they all need to be cleaned from the first name variable in order to be of any potential use.

The first problem is that our previous technique involves looking for words or spaces. There are no spaces here.

One choice would be to use *index* and simply index the phrases 'MTM,' 'PHI,' and so on, but there's a more efficient solution here. Notice that each phrase occurs at the end of the name and is exactly three letters long. The technique we're about to describe only works when cleaning phrases of known length, so it's important to verify that the variables have the correct format before implementing this routine. Here's some code:

```
data cleaned;
   set foo;

   placeholder = left(reverse(var1));
   notations = left(reverse(substr(placeholder,1,3)));
   var1 = left(reverse(substr(placeholder,4)));

   drop placeholder;
run;
```

The resulting dataset:

	var1	var2	var3	notations
1	CHERYL		SMITH	MTM
2	FREDERICK		JONES	PHI
3	PATRICK		SILVER	CHK
4	ANNA		PALMER	DBL

The essence here is that we *reverse* the name, use *substr* to cut the first three characters, and then put the re-*reverse*d pieces in their appropriate locations. We do this because *substr* only works from left-to-right; i.e., it only trims characters from the front. By reversing the words, we're getting around the fact that the names are not the same length.

Also notice that we apply a *left* function every time we *reverse*. This is because there are potentially extra blank characters at the ends of these variables that will also get reversed. If you have already cleaned your name variables using *trim* or *strip*, you may be able to omit the *left* here.

IX. Numerics

It should be relatively obvious that if a number is present in a name variable, that record needs cleaning. But there are two ways a number can be present:

- an unnecessary number that is part of a notation or record ID
- a typo ('0' for 'O', '1' for 'I', etc.)

These are cleaned very differently. To identify them, however, we discovered a relatively simple trick. If you *count* the number of digits in a record, that can help you identify algorithmically which category a record belongs to. Low numbers of digits – 1 or 2 – are more indicative of typos, like J0NES or JON3S, whereas more digits are likely an ID, like a phone number or a room number.

We wrote the following code to count digits:

```
...;
   /* We use an array to count the number of digits in each string */
   array newCounts{10} newCounts1-newCounts10;

   /* Array is initialized to zeros */
   do j = 1 to 10;
      newCounts(j) = 0;
   end;

   /* For each digit, 0-9, count how many appear in varName and put it in the
   appropriate array position */
   do j = 1 to 10;
      numChar = put(j-1,1.);
      newCounts(j) = countc(varName, numChar);
   end;

   /* Add up the total number of digits */
   sumCounts = sum(of newCounts{*});
...;
```

To see how this works, let's run this code on a dataset. We will just look at the output:

var1	newCounts1	newCounts2	newCounts3	newCounts4	newCounts5	newCounts6	newCounts7	newCounts8	newCounts9	newCounts10	sumCounts
T0MMY	1	0	0	0	0	0	0	0	0	0	1
EDWARD RM 4004	2	0	0	0	2	0	0	0	0	0	4
SANDRA 4045559213	1	1	1	1	2	3	0	0	0	1	10
MA00561RTIN	2	1	0	0	0	1	1	0	0	0	5

The final column tells us that the first record has 1 number, the next has 4, and so on. We have left the array variables newCounts1—newCounts10 to illustrate that if you need them, you have also generated a record of how many zeros, ones, twos, etc. each record has. If for some reason, you needed to find all records that only have one or two zeros in them, you could use newCounts1 as a search parameter.

There are two other useful functions to discuss when dealing with numerics. We already know that *index* records the position of the first character of a search string, but SAS has two other useful indexing functions:

- **anydigit**

  Syntax:

  ```
  number = anydigit(varName);
  ```

  This function returns the position of the first numeric character in the variable.

- **anyalpha**

  Syntax:

  ```
  number = anyalpha(varName);
  ```

  This function returns the position of the first alphabetic character in the variable.

When dealing with 'split' names, these functions are invaluable. Consider the final record in our output above – clearly there's an ID number splitting the name 'Martin.' To clean this record, we would use the following code:

```
...;
   nameIndex = anydigit(var1);
   placeholder1 = substr(var1,1,nameIndex-1);
   placeholder2 = substr(var1,nameIndex);

   nameIndex = anyalpha(placeholder2);
   placeholder3 = substr(placeholder2,nameIndex);

   notations = substr(placeholder2,1,nameIndex-1);
   var1 = cats(placeholder1,placeholder3);
...;
```

When we run this routine, we end up with:

	var1	notations
1	TOMMY	
2	EDWARD RM 4004	
3	SANDRA 4045559213	
4	MARTIN	00561

The nice thing about both of these indexing functions is that they are nonspecific – we don't need to know what number or letter to search for; we're looking for *any* occurrence.

X. Identifying Special Characters

In our report, we stated that we found a useful technique in the CSMD involving searching for entries which have a word containing no vowels. If the 'word' is longer than two letters, we can conclude that it is not a set of initials and must therefore be an abbreviation (e.g. CHK, MTM, RMR). Since it isn't feasible that we would be aware of every possible abbreviation, running a 'no vowel' search can help us pick up those abbreviations that we weren't aware of previously. We include the code here:

```
prxSpace = prxparse("/[AEIOUY]/i");
position = prxmatch(prxSpace, varName);
if position ^= 0 then
   vowelFlag = 1;
else vowelFlag = 0;
```

prxparse is a Perl regular expression function. In our case, it returns a pattern identifier when it finds a vowel (the /i at the end makes it a case-insensitive search). We use **prxmatch** to translate that pattern identifier into a position location within the string *varName* and simply flag the record with a 1 if it contains a vowel and a 0 if it does not. We tended to find this search useful after parsing the first word

of the name into a separate column, but it's incredibly useful as one is finishing a routine and looking for any stray, uncleaned records.

If searching for special characters, we also use *prxparse*. Here, we're looking to identify periods, parentheses, dashes, and quote marks after other special characters have been removed:

```
prxSpace = prxparse("/[^A-Z \s]/i");
position = prxmatch(prxSpace, varName);
if position ^= 0 then
   varFlag = 1;
else varFlag = 0;
```

While this search would also flag numbers, we recommend using *anydigit* as described above to deal with numerics.