Paper 4638-2020

Cracking Cryptic Doctors' Notes with SAS® PRX Functions

Amy Alabaster, Kaiser Permanente; Mary Anne Armstrong, Kaiser Permanente

ABSTRACT

SAS® programmers analyzing complex free text data, like that present in the electronic medical record, likely find basic string functions insufficient or unwieldy for extracting meaningful data points from highly variable text sources. In addition to coded data like diagnoses, a patient's medical record consists of doctors' notes, test results, and reports with immense provider variation in text—cryptic abbreviations and typos are only the beginning. Perl regular expressions tackle many text problems like those encountered by programmers analyzing the medical record, and SAS uses this powerful tool with a suite of functions and call routines. This paper reviews the basics of implementing Perl regular expressions in SAS using SAS functions like PRXPARSE, CALL PRXSUBSTR, PRXMATCH, PRXCHANGE, and CALL PRXNEXT. Text examples draw from several medical specialties. Topics covered include managing variation within single words (like typos), locating multiple keywords with varying distance between words, finding multiple iterations of a target phrase, breaking up text by words or sentences, dealing with negation, and managing very long regular expressions. Practical considerations for getting started with a natural language processing project and balancing false positives versus false negatives are discussed.

INTRODUCTION

This paper introduces one of many SAS tools available to analyze free text variables – the PRX functions which use Perl regular expressions. PRX functions are enormously powerful and flexible and can be used seamlessly within your DATA step – no special software required. We cover the basics of the required syntax – forewarning, there is a small learning curve – and move on to simple and intermediate examples. The examples draw from medical research but can be easily applied to other industry-specific analysis of large text fields.

QUICK START GUIDE TO USING PERL REGULAR EXPRESSIONS IN SAS

The PRX in SAS PRX- functions stands for Perl Regular eXpressions. Regular expressions (or RegEx) are used across many platforms, including SAS, and are composed using a sequence or pattern of characters. This pattern can then be searched for within a body of text. Perl is one popular syntax used to write regular expressions. RegEx can use typical characters (A-Z, 1-9) or special metacharacters that define ideas like:

- Wild card characters… e.g. `\d` = any digit
- Iterations of a character or character class… e.g. `\d+` = 1 or more digit
- Position within a text string… e.g. `\bdigit` = 'digit' at the beginning of a word boundary
- Grouping and alternation… e.g. `digit(s|al)` = 'digits' OR 'digital'

A RegEx is the pattern of characters you need to locate within a string. They have their own syntax. In SAS, PRX functions and call routines do the actual locating. They are the packaging that wraps around the RegEx, and as with any SAS function, they *also* have their

own syntax. Before we discuss the packaging though, we'll review the Perl syntax used to define a RegEx.

METACHARACTERS

Regular expressions take advantage of several different kinds of special characters.

Error! Reference source not found. shows a (non-comprehensive) list of common metacharacters, their descriptions, and examples of what can be found and not found while using them. Note that all metacharacters are case sensitive.

Metacharacter	Definition	Example
Wildcards		
\d	any digit (0-9)	\d\d matches two digits in *temp of 98* but not the single digit in *age 5 years*
\w	any word character (A-Za-z0-9_)	\w\w\w\w\w matches *Smith* but not *O'hare*
\W	any non-word character	\w\w\W\w\w matches *he is* and *11:32*
\s	any whitespace character	\s matches a space, a tab, but not ¶ (paragraph symbol)
.(period)	any character	p.k.m.n matches *pokemon, pokiman* and *pok%m$n* but not *pokemn*
\b	word boundary	toxic\b is found in *toxic looking* or *looks toxic*[even at end of string] but not *toxicology*
Grouping and alternation		
|	alternate choices	temp|fever matches the beginning of *temp of 102* or *fever of 102*
()	groups characters*	combin(ed|ing)matches *combined* and *combining*, but not *combination*
[xyz]	match any character in brackets	m[ie]r[ea]na matches *mirena* or *merana*
[^xyz]	match any character not in brackets	temp[^:] is found in *temperature of 101* but not *temp: 101*
Repetition		
?	match *character or group* 1 or 0 times (greedy) †	mirr?ena matches *mirena* or *mirrena* gen(eral)? matches *gen* or *general*
*	match 0 or more times (greedy)†	temp.*\d\d\d matches *temp of 101* or *temperature measured to be 101* or *temp101*
+	match 1 or more times (greedy) †	temp.+\d\d\d matches *temp of 101* or *temperature measured to be 101* but not *temp101*
{n,m}	match at least n, but not more than m times (greedy) †	temp.{0.20}\d\d finds matches of *two digits* between 0 and 20 characters away from *temp*
*?	Match 0 or more times (lazy) ‡	temp.*?\d\d\d matches *temp of 101*, from the phrase 'temp of 101 or 102'

Using special characters literally		
\	escape character: match a literal { } [ ] ( ) ^ $. | * + ? \	\d\d\.\d only matches *98.6* while \d\d.\d matches 98.6 or 98#6
Delimiters		
/.../	delimit the start and end of the RegEx	/temp.*\d\d\d?/
/.../i	the i after delimiters makes RegEx case insensitive	/uppercase/i matches *uppercase* and *UPPERCASE*

Table 1. A non-comprehensive list of special characters used in regular expressions

A note about parenthesis

* Grouping characters is useful for alternating choices, as shown in the Table 1 example. In addition to grouping, parenthesis also define what's known in RegEx as a capture buffer. For example, in the RegEx /(temp.*)(\d\d\d?)/ capture buffer – first () – 1 includes the characters temp through the 0 or more characters .*. Capture buffer 2 – second () – includes the two or three digits, presumably the temperature data that we're most interested in capturing for analysis purposes. Some PRX functions act specifically on capture buffers to extract pertinent characters (see pages 7-8 for an example).

A note about greedy and lazy repetition factors

† Greedy repetition factors return the longest match possible. So in /temp.*\d\d/ - where you are searching for the word 'temp' followed by the greedy .* (0 or more of any character) followed by \d\d (2 numbers): even if 2 numbers are found within a few characters, if 2 numbers are found again 100 characters later, the larger match will apply. E.g. it matches the entirety of '*temp of 98.2, age of 27'.*

‡ Lazy repetition factors return the shortest match possible. So, using /temp.*?\d\d/ the function would identify a match of only the phrase '*temp of 98',* instead of '*temp of 98.2, age of 27'* in the above example. This is important when you need to pull out a specific value or determine the length of a key phrase. All repetition factors can be either greedy or lazy. Adding ? to any repetition factor makes it lazy.

You can imagine from the examples above that there are often many solutions to a RegEx problem. Those new to building regular expressions may find the tool available at https://regex101.com useful for seeing how RegEx work in action and for troubleshooting tricky patterns.

PRX FUNCTIONS AND CALL ROUTINES

As mentioned earlier, PRX functions and call routines are specific to SAS and do the work to locate a RegEx of interest and output information such as position and length. This paper will cover a selection of (but not all) PRX functions that have been particularly useful in health research.

Table 2 summarizes the syntax and role of the PRX functions and call routines that will be discussed in this paper. Much of this information is borrowed from the SAS 9 Perl Regular Expressions tip sheet on the SAS support website. Additional details for each function will be described in the examples in the next section.

Function or Call Routine	Syntax	What it does
PRXPARSE	*regex-id* = prxparse(*perl-regex*)	Compile Perl regular expression *perl-regex* and return *regex-id* to be used by other PRX functions. RegEx is stored in *regex-id* variable, but note if you print out *regex-id* it'll just print '1'.
CALL PRXSUBSTR	call prxsubstr(*regex-id*, *source*, *pos*, *len*)	After using prxparse, searches in *source* for *regex-id*, and if found returns starting position *pos* and length *len* of previously defined *perl-regex*. Else returns *pos*=0 and *len*=0.
PRXMATCH	*pos* = prxmatch(*regex-id* | *perl-regex*, *source*)	Searches in *source* for *perl-regex* or previously defined *regex-id* and returns starting *pos* if found, else 0.
PRXPOSN	text = prxposn(*regex-id*, *n*, *source*)	After a call to prxmatch or prxchange, prxposn returns the text of capture buffer *n**. If *n* is 0, prxposn returns the entire match.
CALL PRXNEXT	call prxnext(*regex-id*, *start*, *stop*, *source*, *pos*, *len*)	Use this call routine iteratively to find multiple iterations of your *perl-regex*. Searches in *source* between positions *start* and *stop* (initially set to 1, and length(*source*) respectively). Returns *pos* and *len*. Also resets *start* to *pos*+*len*+1 so another search can easily begin where the last search left off.
PRXCHANGE	*new-string* = prxchange(*regex-id* | *perl-regex*, *times*, *old-string*)	Search and replace *times* number of times in *old-string* and return modified string in *new-string*. If the value of *times* is -1, then matching patterns continue to be replaced until the end of *old-string* is reached. See example for special *perl-regex* syntax.

Table 2. Syntax and description of PRX Functions and Call Routines, color coded for clarity

* See note under Table 1 about capture buffers.

A BASIC SAS PROGRAM FOR DEFINING AND USING A REGULAR EXPRESSION

The following program uses PRXPARSE to define a regular expression that will be used to search for a phrase describing temperature or fever. It then uses CALL PRXSUBSTR to return the starting position and length of the matched phrase. The results are printed and shown in Output 1.

A few notes:

- In the first step, prxparse defines the RegEx and assigns the result to the *regex-id* re. Since the search pattern is the same for every observation, best programming practice suggests only defining the RegEx once, and using retain to fill in the result through the entire dataset. It is in the next step, when we use call prxsubstr that the *source* note_text is searched for a match in each observation.

- Since the *perl-regex* is a character string, remember to surround it in quotations. Also remember to use the forward slash delimiters at the beginning and end of *perl-regex*, and inside the quotations.
- Placing an `i` between the ending delimiter and the ending quotation makes the entire *perl-regex* case insensitive.
- The RegEx described here looks for the keyword fever or temp, followed by three digits. The period . is used to denote that from 1-30 of *any* character can be between the fever/temp keyword and the three digits. The ? denotes that the repetition factor {1,30} is a lazy repetition factor – it finds the shortest match possible.

Here is the full code:

```
data notes;
  input note_text $80.;
  datalines;
  Fever up to 101 today as well increased fussiness x 2days
  Parents came into ED after temp this morning at home 101
  Vomit x 1 yesterday. No known fever at home
  ;
run;

data parse;
  set notes;
  if _N_ = 1 then do;
       re=prxparse("/(fever|temp).{1,30}?\d\d\d/i");
     end;
  retain re;
  call prxsubstr(re, note_text, pos, len);
run;

proc print data=parse; run;
```

Obs	note_text	re	pos	len
1	Fever up to 101 today as well increased fussiness x 2days	1	1	15
2	Parents came into ED after temp this morning at home 101	1	28	29
3	Vomit x 1 yesterday. No known fever at home	1	0	0

Output 1. Output from PROC PRINT following PRXPARSE and PRXSUBSTR

The material in this 'Quick Start Guide' is mostly borrowed from our previous SAS proceedings paper (Alabaster 2018). Other excellent guides on regular expressions and PRX functions have been written by SAS programmers, including Ron Cody (2004), David Cassell (2005), Richard Pless (2005), and Kenneth Borowiak (2007). Check out the reference section of this paper for more information about their papers.

MORE REAL RESEARCH EXAMPLES USING PRX FUNCTIONS

Below are examples drawn from real health research projects covering topics like managing variation within single words (like typos), locating multiple keywords with varying distance between words, finding multiple iterations of a target phrase, breaking up text by words or sentences, dealing with negation, and managing very long regular expressions.

DEFINING SEVERAL TARGET REGEX USING PRXPARSE

For a recent research project, our client wanted to look for evidence within operative notes that an intrauterine device (IUD) perforated a patient's uterus. The term IUD could be used in the note, or the specific type of IUD, like Mirena or Paragard. The study ultimately required many RegEx, but below we describe two which capture the following concepts:

1. Discussion of embedded IUD
2. The word IUD discussed in close context with anatomy not normally associated with an IUD (e.g. abdomen, peritoneal cavity, fimbra)

The following regular expressions (and those throughout this section) use common syntactic tools. We allowed for a lot of flexibility in our keywords to allow for variation and typos.

- Variation in keywords (e.g. IUD or Intrauterine device or Mirena) captured using grouping `()` and alternation `|` metacharacters
- Differences in spelling captured using brackets [AB] (e.g `iu[ds]` for IUD or IUS), wildcards (especially for vowels known to be big sources of typos), and optional characters ? (e.g. `m.rr?.nn?a` for Mirena, Merana, or Mirrenna).
- Pairs of keywords (e.g. IUD and embed) allowed to be separated using a min and max distance, . wildcard with `{#,#}`
- We broke up several long RegEx for ease of reading with return and tab keys. Since the RegEx syntax reads these literally, the filler `|xxx` is inserted to mark where breaks are to occur, and the alternation `|` metacharacter is placed at the start of the new line. The use of the `|` alternation metacharacter helps to ensure that fillers and tabs are inconsequential (it's just a choice that will not ever be matched in the text)

The following program parses a concatenated note dataset for two different regular expressions:

```
data parse;
  set concat;
  if _N_ = 1 then do;
    re1=prxparse("/(iu[ds]|int.{0,3}ut.r.n.{1,2}d|m.rr?.nn?a|xxx
       |p.r.gau?rd).{1,100}[ie]mbed/i");
    re2=prxparse("/(iu[ds]|int.{0,3}ut.r.n.{1,2}d|m.rr?.nn?a|xxx
       |p.r.gau?rd).{1,50}(abdom[ei]n|oment(um|al)|pelvi[sc]|fimbra)/i");
  end;
  retain re1 re2;
run;
```

USE CALL PRXSUBSTR AND SUBSTR FUNCTIONS TO EXTRACT A TARGET PHRASE

Taking advantage of templated notes

For another research project, our client needed to gather information about young infants presenting to the emergency department with fever. Among the many variables they wished to collect were items that could only be found in free text notes. For instance, they wanted to know whether a patient was well-appearing or ill-appearing at presentation. While doctors' notes are highly variable, they often use note templates with common headings and subheadings. An infant's appearance is often found under the subheading 'General:' or 'Constitutional:'. Below are three notes that contain the needed text:

Note 1: General Appearance: Alert and non-toxic. Appears in no acute distress. ?Eyes: PERRL, lids and conjunctiva normal, EOM intact ?ENT: Hearing grossly normal, external ears and nose normal ? Fever of 100.5 at home rectally

Note 2: GENERAL: well appearing, no acute distress, well hydrated ?HEAD: normocephalic and anterior fontanelle open, flat ?EYES: normal eye exam ?EAR: TMs clear bilaterally ?NOSE: normal external appearance ? Fever 2 days

Note 3: Const: dehydrated, ill-appearing ? Eyes: pupils equal and reactive, lids, conjunctiva nl ?ENT: bilateral TM's and external ear canals normal, normal and patent, mucous membranes moist, pharynx normal without lesions ? Temperature 99 at home

The client requested all text from the start of the general/constitutional subheading until the next subheading. Since in our system paragraph breaks show up as a '?', we wrote a RegEx that pulls all text from the specified keyword until the next paragraph break, or '?'.

Here are some features of the RegEx re3 below:

- The first capturing group in `()` matches the keywords general or constitutional. However, the `(eral)` and `(itutional)` are optional, by use of the ? repetition factor.
- There is a single optional whitespace (again, using ?).
- The next capturing group `()` matches a colon :, the word appear, or a dash -. Remember that the | indicates alternate choices.
- A ? in a RegEx typically indicates a zero or one repetition factor. However, using the escape character \ (see table 1), it denotes a literal '?'. The brackets `[]` combined with a carrot `[^]` indicate that any character *except* the character in brackets should be matched. Combined with a * repetition factor this denotes that we want zero or more characters until we reach a literal ?.

Here is the code:

```
data parse;
  set concat;
  if _N_ = 1 then do;
    re3=prxparse("/(gen(eral)?|const(itutional)?) ?(:|appear|-)[^\?]*/i");
  end;
  retain re3;
run;
```

Now that our RegEx for the general/constitutional subheading text is defined using PRXPARSE, we now want to locate the desired text in the body of the notes and do something with the match. For this project, we are going to extract the text using the position and length of the match and print out the results:

```
data substring;
 set parse;
 call prxsubstr(re3, notes, pos, len);
 length gen_string $ 500;
 if pos NE 0 then gen_string=substr(notes, pos, len);
run;

proc print data=substring(where=(gen_string NE '')); var gen_string; run;
```

```
Obs gen_string
  1 General Appearance: Alert and non-toxic.  Appears in no acute distress.
  2 GENERAL: well appearing, no acute distress, well hydrated
  3 Const: dehydrated, ill-appearing
```

Output 2. Output from PROC PRINT following CALL PRXSUBSTR and SUBSTR

EXTRACT A NUMBER FROM A TARGET PHRASE USING PRXMATCH AND PRXPOSN

Temperature measured in a doctors office or at the hospital is usually recorded in a structured data field in the medical record and is easily extractable. However, for our research project studying young febrile patients, the researchers also wanted to know if the patient had a fever at home before presenting to the medical center. This information is only available in free text notes. The notes used for the last example above on page 6

contain information about temperatures measured at home. Our goal in the next example is to extract the numeric temperature into a numeric variable.

In the following code, the RegEx is first defined:

- It looks for the keyword `fever` or `temp`. Temperature will also be matched as it contains the keyword 'temp'.
- The matched pattern also must contain two or three digits. The `?` makes the third digit optional.
- To allow for one digit following a decimal, we have to use the escape `\` character to match a literal period `\.`. This and a final digit `\d` are wrapped in parentheses `()` and followed by a `?` to make the decimal point and last digit optional.
- The period `.`, the any character wildcard, followed by the `{1,30}` repetition factor allows for between 1 and 30 characters between the fever/temp keyword and the two or three digits.
- `?` is used again, this time to denote that the repetition factor is lazy. We want the shortest match possible.

```
data parse;
  set notes;
  if _N_ = 1 then do;
       re4=prxparse("/(fever|temp).{1,30}?(\d\d\d?(\.\d)?)/i");
     end;
  retain re4;
run;
```

In the next code block, we first find the starting position of the match (if any) using PRXMATCH. If the starting position is greater than 0 (in other words, there is a match), we use PRXPOSN to extract the text containing the number. The second argument to PRXPOSN specifies which capture buffer should be extracted. We want the whole number, which is all the text contained in the second set of parenthesis `()` above. The optional decimal and digit are a third capture buffer because they are also enclosed in parenthesis, but these are also contained within the second capture buffer. See the note in the quick start guide about capture buffers for more info. Finally, we convert the text into a numeric variable:

```
data extract;
  set parse;
  pos=prxmatch(re4, notes);
  if pos >0 then num_string=prxposn(re4, 2, notes);
  temp_num=input(num_string, 5.1);
run;

proc print data=extract; var re4 pos num_string temp_num; run;
```

Obs	re	pos	num_string	temp_num
1	1	189	100.5	100.5
2	1	0		.
3	1	221	99	99

Output 3. Output from PROC PRINT following PRXPARSE, PRXMATCH, AND PRXPOSN

DEFINING WORDS WITHIN A REGEX

There are times that instead of looking for a target phrase within a certain range of *characters*, you want to look within a certain number of *words*. For example, when we were looking for evidence of removal of an IUD within notes, we searched for keywords related to the word 'removed'. However, finding the word 'removed' was insufficient, because right before the word removed, could be the word 'not'. We needed to find negation words near the target keywords.

Find negation within five words before or after a keyword

The goal for the next example is to find evidence of IUD removal while avoiding negation terms. In this example, we'll assume negation terms usually come before our keyword (e.g. 'without removal' or 'not actually removed'). We will first capture keywords for 'removal' and five words preceding that keyword. Then, within the matched text, we will look for a negation term.

There are many ways to define a word with RegEx. The code you use will depend on if whether your word is made up of only alphabet characters `[a-z]`, alphanumeric characters `[a-z0-9]` or `\w`, or alphanumeric characters and some punctuation `[\w'-]`. A word will have one or more of the desired characters, so you need the one or more repetition factor, `+`, e.g. `\w+`. To capture multiple words, you need to decide what you will allow between the words. `(\w+\s+){1,5}` captures 1-5 words each followed by one or more whitespace characters. `(\w+\W+){1,5}` captures 1-5 words each followed by one or more non-word characters. The former example would avoid punctuation and is good if you want a group of words within a sentence. The latter example is more flexible and will capture a group of words containing punctuation, quotation marks, etc.

In the following code, we first define our RegEx as in previous examples. We are searching for the word removed or removal `remov(ed|al)`, and capturing up to five preceding words. A second RegEx defines negation words separated by the alternation character `|`:

```
data iud_notes;
  input notes $80.;
  datalines;
  IUD was removed intact
  After discussing we proceeded without IUD removal
  IUD strings not seen, so could not easily be removed today
  Mirena IUD removed without difficulty
  Here to discuss her IUD
  ;
run;

data parse;
  set iud_notes;
  if _N_ = 1 then do;
    re5=prxparse("/(\w+\W+){0,5}remov(ed|al)/i");
    re6=prxparse("/not|without/i");
  end;
  retain re5 re6;
run;
```

We again use CALL PRXSUBSTR and the SUBSTR function to extract text containing our RegEx for removal `re5`. We then search within the extracted text contained in the `remov_str` variable for the second RegEx `re6` with negation words. An indicator variable `iud_matched` is created if it meets the Boolean criteria of no match found (position returned from PRXMATCH=0).

```
data match;
 set parse;
 call prxsubstr(re5, notes, pos, len);
 length remov_str $ 100;
 if pos NE 0 then do;
   remov_str=substr(notes, pos, len);
   iud_removed=(prxmatch(re6, remov_str)=0);
  end;
run;

proc print data=match; var remov_str iud_removed; run;
```

Obs	remov_str	iud_removed
1	IUD was removed	1
2	discussing we proceeded without IUD removal	0
3	so could not easily be removed	0
4	Mirena IUD removed	1
5		0

Output 4. Output from PROC PRINT following PRXPARSE and PRXMATCH

MATCH MULTIPLE ITERATIONS OF A TARGET PHRASE WITH CALL PRXNEXT

Let's return to the example on page 7, where a clinician wanted to extract all text on an infant's appearance starting from the subheading 'General:' or 'Constitutional:' until the next paragraph break. We used CALL PRXSUBSTR to locate the position and length of the *first* match of our RegEx, if any, found within a note. There are times when a single note may contain multiple iterations of this target phrase − for instance if a patient received an exam by more than one physician, or a physician uses a note template containing both a 'General:' and 'Constitutional:' subheading. To ensure all target text is collected, we want to match *all* instances of the target phrase. We do this using the CALL PRXNEXT routine.

For the purpose of this example, we combine Note 2 and Note 3 above − this is text of one observation stored in the notes field of our theoretical dataset:

GENERAL: well appearing, no acute distress, well hydrated ?HEAD: normocephalic and anterior fontanelle open, flat ?EYES: normal eye exam ?EAR: TMs clear bilaterally ?NOSE: normal external appearance ? Fever 2 days Const: dehydrated, ill-appearing ? Eyes: pupils equal and reactive, lids, conjunctiva nl ?ENT: bilateral TM's and external ear canals normal, normal and patent, mucous membranes moist, pharynx normal without lesions ? Temperature 99 at home

As before, PRXPARSE is first used to define the RegEx. The following code uses the same RegEx defined in the previous example:

```
data parse;
  set concat;
  if _N_ = 1 then do;
    re3=prxparse("/(gen(eral)?|const(itutional)?) ?(:|appear|-)[^?]*/i");
  end;
  retain re3;
run;
```

In the next code block, CALL PRXNEXT uses the defined RegEx to search within the note for a match and returns the position and length of the match − just like we did with CALL PRXSUBSTR. This time, instead of searching within the entire note for the *first* match, we

search for the first match within the defined start and stop positions. At the beginning of the code block, start and stop are set to 1 and length(note) respectively. At the end of the CALL PRXNEXT call routine, the function resets the start position to position + length of the previous match. We use a DO WHILE loop to loop through the following program steps until a match is no longer found, causing pos =0:

- Assign matching text to variable 'gen_string' using SUBSTR function
- Output an observation containing the text chunk
- Continue searching in text at new start position using CALL PRXNEXT

```
data all_substrings;
 set parse;
 length gen_string $ 500;
 start=1;
 stop=length(notes);
 call prxnext(re3, start, stop, notes, pos, len);
 do while pos>0;
   gen_string=substr(notes, pos, len);
   output;
   call prxnext(re3, start, stop, notes, pos, len);
 end;
run;

proc print data=all_substrings; var gen_string pos len start stop; run;
```

```
Obs gen_string
  1 General Appearance: Alert and non-toxic.  Appears in no acute distress.
  2 Const: dehydrated, ill-appearing

Obs pos   len   start   stop
  1 1     58    59      457
  2 215   39    254     457
```

Output 5. Output from PROC PRINT following CALL PRXNEXT and SUBSTR

REMOVE VARIABILITY FROM A TEXT BLOCK WITH PRXCHANGE

As part of note data preparation, it may be useful to simplify the data set to reduce some amount of variability. We can do this by removing all non-alphanumeric (a-z, A-Z, 0-9), spaces, and phrase-ending (.,?) characters. Using the code below, *non-toxic* and *nontoxic* will be read similarly. If these distinctions or symbols are unimportant for the task at hand, their removal can quite helpful.

The syntax for PRXCHANGE is different from the other PRX functions because you need to specify both a search and substitution regular expression.

- In the following code, the *perl-regex* has three delimiters (/), separating the RegEx into two sections, a search section and substitute section. The s before the first delimiter indicates that the first RegEx should be substituted for the second RegEx.
- The first RegEx has alphanumeric characters, whitespace, a period, a comma, and a question mark enclosed in square brackets with a leading carrot (^). This represents a single character that is not (because of the ^) within the brackets (remember brackets indicate choice).

- The second delimited RegEx is null (no characters between //).
- Any character not within the square brackets in the previous RegEx will be removed and replaced with nothing (in other words, substituted with the null RegEx).
- In the prxchange syntax, *n-times=-1* indicates that the function will search and substitute until the end of the *source* text is reached.

Here's the code:

```
data cleanconcattext;
  set concattext;
  length cleantext $10000.;
  cleantext=prxchange("s/[^a-zA-Z0-9?., ]//", -1, note_text);
run;
```

PRACTICAL CONSIDERATIONS FOR GETTING STARTED WITH A NATURAL LANGUAGE PROCESSING PROJECT

PREPPING NOTE DATA

As notes can be thousands of characters long, they are broken into several lines within a table in the EMR database. For most projects, since we need to find phrases that could span multiple lines of a note, we first concatenate all lines within a single note. There are many ways to do this, but here is one suggested method:

3. Determine the distribution and maximum number of lines for a single note within the dataset.
4. Convert your note line dataset from long form to wide form using your favorite long to wide program (e.g. PROC TRANSPOSE).
5. Use the CATX function to concatenate lines and create a single character variable, after first specifying the length of the new variable. Note that the maximum size of a single variable created within a data step is 32,767 characters. A delimiter, such as a whitespace, can be specified in CATX to separate concatenated lines within the new variable.

Interestingly, in our note data set, paragraph breaks are saved in the EMR as a question mark, ?. Though they make the electronic note fields painful to look at, they are useful for determining contiguous phrases. They can be processed within a regular expression like any other character.

DESIGNING A REGULAR EXPRESSION

Regular expressions might look like they were written by a robot, but, at least using the methodology covered in this paper, they are entirely human crafted. Knowledge of the syntax and metacharacters and an eye for patterns is hugely helpful, but even the best programmer needs exposure to subject area expertise and anecdotal experience to write a good regular expression.

For us, crafting regular expressions starts with a conversation with a clinical expert and review of several example notes. After the notes have been concatenated and cleaned, we export a sample of notes into a spreadsheet. The clinical expert helps to identify target phrases and discusses other possible variations based on their experience. As we brainstorm target phrases, we also identify sources of potential false positives – negated phrases, keywords used out of context, etc.

In any variable defining algorithm, false positives and false negatives are inevitable, and the key is finding a balance. If there's a budget for additional chart review following electronic

case identification, with the goal to manually remove false positives, a more inclusive RegEx strategy could be best – inclusive RegEx=avoid missing cases or false negatives. If missing some cases is OK, as long as cases identified are most certainly true positives, then a more conservative RegEx strategy may be better suited.

CONCLUSION

SAS PRX functions and call routines can assist with complex natural language processing problems in health research.

REFERENCES

Alabaster A. Armstrong MA. 2018. "Interpreting electronic health data using SAS PRX functions." WUSS 2018 Proceedings. Paper 92-2018.

Backman T. 2004. "Benefit-risk assessment of the levonorgestrel intrauterine system in contraception." Drug Safety. 27(15):1185-204.

Borowiak KW. 2007. "PERL Regular Expressions 102." SAS Global Forum 2007 Proceedings. Paper 135-2007.

Cassell DL. 2005. "PRX Functions and Call Routines." SUGI 30 Proceedings. Paper 138-30.

Cody R. 2004. "An Introduction to Perl Regular Expressions in SAS 9." SUGI 29 Proceedings. Paper 265-29.

Heinemann K, Reed S, Moehner S, Minh TD. 2015. "Risk of uterine perforation with levonorgestrel-releasing and copper intrauterine devices in the European Active Surveillance Study on Intrauterine Devices." Contraception. 91:274-9.

Pless R. 2005. "An Introduction to Regular Expressions with Examples from Clinical Data." PharmaSUG 2005 Proceedings. Paper TU02.

"Regular Expressions 101." Available at: https://Regex101.com

SAS RegEx Tip Sheet. Available at: https://support.sas.com/rnd/base/datastep/perl_regexp/regexp-tip-sheet.pdf

van Grootheest K, Sachs B, Harrison-Woolrych M, Caduff-Janosa P, van Puijenbroek E. 2011. "Uterine perforation with the levonorgestrel-releasing intrauterine device: analysis of reports from four national pharmacovigilance centres." Drug Safety. Jan 1;34(1):83-8

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Amy Alabaster
Kaiser Permanente Division of Research
amy.alabaster@kp.org