Paper 92-2018

Interpreting electronic health data using SAS PRX functions

Amy Alabaster, Kaiser Permanente Division of Research; Mary Anne Armstrong, Kaiser Permanente Division of Research

ABSTRACT

Analysts of electronic medical record data take advantage of structured data fields used for patient tracking and claims purposes. In the absence of codes, analysts can use unstructured text data, such as those found in doctors' notes. Though notes are rich in information, they are also full of inconsistencies – cryptic abbreviations, typos, and immense provider variation – that often make structured analysis using basic string functions inadequate. Alternatively, Perl regular expressions can be used to tackle many text problems in health research. This paper reviews the basics of regular expressions and their implementation in SAS® using the functions and call routines: PRXPARSE, CALL PRXSUBSTR, PRXMATCH, CALL PRXNEXT, PRXPOSN, and PRXCHANGE. To illustrate how these functions are applied in the health research setting, a post-marketing pharmaceutical study is discussed- in which, the FDA asked a team of research partners to show how un-coded intrauterine device (IUD) outcomes could be found in unstructured health record data. Even with a powerful tool like regular expressions, a balance must be achieved between the risk of false positives and false negatives. For this purpose, a random sample of 100 charts was closely reviewed by a clinical expert for data accuracy. For the primary outcome, uterine perforation following IUD insertion, 77 of 100 events found using text searching capabilities were confirmed to be true uterine perforations. While advanced text mining tools are available with other platforms, and even now with SAS, a basic understanding of regular expressions and the PRX functions are sufficient to achieve efficient and valid results.

INTRODUCTION

The electronic medical record (EMR) is comprised of structured and unstructured data. Structured data includes demographics, diagnosis codes, drug codes, vital sign measurements, laboratory values, etc. Unstructured data includes all the many notes and free-text entries from office visits, hospital encounters, secure messages, radiology scans, etc. In health research, we often need to define a data element from unstructured note data, which can be challenging due to the variability in the way information is reported.

For example, a child's temperature taken in clinic may be available in structured data, but what if a researcher needs to know whether a child had a fever measured at home before coming to the clinic? While this information is often available in the clinical note, it may be recorded as 'temp of 102 by ear' or 'fever recorded at home – 102F' or '>101 temperature this morning', etc. A tool is needed that can locate keywords (temp/fever) and the numeric values closest to that keyword, without pulling in one of many unrelated numeric values that may be discussed within the same note.

Perl regular expressions, and the PRX functions in SAS®, are one such tool available to help capture the many possible flavors of a targeted phrase, without pulling in similar but not quite correct variations. Though PRX functions have been available in SAS version 9 and up for more than ten years, many analysts find the syntax intimidating. This paper hopes to show that once the basic syntax is mastered, slogging through the slashes and stars and random characters can add a great deal of value to the analysis of unstructured text data.

The first section of this paper discusses examples from health research as it walks through a quick start guide to regular expressions in SAS. PRX functions PRXPARSE, CALL PRXSUBSTR, PRXMATCH, CALL PRXNEXT, PRXPOSN, and PRXCHANGE make an appearance.

MOTIVATING EXAMPLE

A recent study conducted in Europe found a potential relationship between risk of uterine perforation and breastfeeding at the time of intrauterine device (IUD) insertion (Heinemann 2015). This prompted interest in conducting a similar study in the United States. While the European study was a prospective enrollment study, researchers in the U.S. hoped to use retrospective data in the electronic record of large regional health systems. Before the study could be approved, the FDA required that researchers show the main outcome, uterine perforation, could be obtained from note data, since it was predicted (accurately) that diagnostic codes would not be helpful in most cases.

In the second section, we will show that the task at hand, parsing operating room and clinic notes for phrases related to uterine perforation, can be accomplished with SAS PRX functions.

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. `^\d` = a digit at the beginning of a line
- 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.

Table 1 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. A link to the complete list of metacharacters is provided in the reference section.

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* and *pokiman* and *pok%m$n*
Position		
`\b`	word boundary	`toxic\b` is found in *toxic looking* or *looks toxic*[even at end of string] but not *toxicology*
`^`	beginning of line	`^cat` is found in *cat in the hat* but not *hat on the cat*
`$`	end of line	`hat$` is found in *cat in the hat* but not *hat on the cat*

Grouping and alternation		
`()`	groups characters†	`combin(e|in)` matches the beginning of *combined, combining*, and *combination*, but not *combo*
`|`	alternate choices	`temp|fever` matches the beginning of *temp of 102* or *fever of 102*
`[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 *temp of 101* but not *temp: 101*
Repetition		
`?`	match *character or group* 1 or 0 times (greedy) ‡	`mirr?ena` matches *mirena* or *mirrena*
`*`	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 at 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 from *temp*
`*?`	match 0 or more times (lazy) §	`temp.*?\d\d\d` matches *temp of 101*, etc
Using special characters literally		
`\`	escape character: match a literal { } [ ] ( ) ^ $. | * + ? \	`\d\d\.\d` matches *98.6* but not *986*
Delimiters		
`/…/`	delimit the start and end of the RegEx	`/temp.*\d\d\d?/`

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, parentheses also define what's known in RegEx as a capture buffer. For example, in the RegEx `/(temp.*)(\d\d\d?)/` capture buffer 1 includes the characters *temp* through the 0 or more characters (.*) before the first of two or three digits. 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.

A note about greedy and lazy repetition factors

† Greedy repetition factors match as many times as 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. It matches the entirety of *temp of 98.2, age of 27.*

§Lazy repetition factors match as few times as possible. So using `/temp.*?\d\d/` the function would identify a match of only the phrase *temp of 98* 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 the health research setting.

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 research case 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* variable 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 this one 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.

Table 2. Syntax and description of PRX Functions and Call Routines

† 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 `re` (the *regex-id*). 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 data set. It is in the next step, when we use **prxsubstr,** that `note_text` (the *source*) 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 `i`nsensitive.

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

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.

VALIDATING AN IUD PERFORATION SEARCH ALGORITHM USING PRX FUNCTIONS

In this section we'll show several examples from a single study that utilized natural language processing extensively to define a research outcome. You can use the subheadings to jump ahead to specific topics or functions of interest.

THE PROJECT

In 2016, several health research organizations across the U.S. partnered to conduct a retrospective, data-only study of adverse intrauterine device (IUD) outcomes among postpartum women. This followed a European study which found that, while rare, uterine perforation was associated with breastfeeding status at time of IUD insertion (Heinemann 2015). Given possible differences in IUD insertion and breastfeeding practices in the U.S. compared to Europe, the FDA required the IUD manufacturer to conduct a follow-up study. To reduce cost and time needed to conduct the study, the pharmaceutical company proposed the use of retrospective electronic medical record data. But first, the FDA asked the research partners to show that the outcomes of interest are obtainable within the medical record.

While there are diagnosis, procedure, and pharmaceutical product codes for IUD insertions, IUD outcomes are more difficult to pinpoint. Most women keep their IUD until the recommended removal date (3-10 years) or they no longer need or want to use an IUD as their birth control method. IUD removals are often coded using diagnosis and procedure codes, but not always. Some women spontaneously expel their IUD, an event with no specific code, and which may or may not even be discussed by a provider in a clinical note. In rare instances the IUD may become embedded in the uterine wall or pass through the uterus into the abdominal cavity. For some women, uterine perforations are asymptomatic and only discovered when they have an appointment with a provider for IUD removal. Others experience discomfort or more serious symptoms, prompting them to visit their provider. Depending on it's location, perforated IUDs may be removed non-surgically in the medical office, in the operating room, or not at all (Backman 2004). Some non-specific codes, such as 'IUD complication', may help in the identification of IUD perforations, but they lack the sensitivity and specificity to be used alone for this purpose. Clinic notes, radiology reports, and operating room notes are essential for identifying all IUD perforations in an EMR database, and natural language processing is required to handle the large variability in case presentation and provider language.

Each of the four research organizations providing retrospective data was asked to develop their own algorithm for identifying IUD perforations and to review a random sample of 100 suspected perforations to measure their algorithm's validity. Each organization reported their positive predictive value (# actual perforations/# suspected perforations). Negative predictive value was not obtained due to the very rare nature of the outcome (1 to 3 per 1000: van Grootheest 2011). At our institution, Kaiser Permanente Northern California (KPNC), we used SAS PRX functions for all our natural processing needs throughout our part of this ongoing pharmaceutical study.

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 this project, since we needed to find phrases that could span multiple lines of a note, we first concatenated all lines within a single note. There are many ways to do this, but here is one suggested method:

1. Determine the distribution and maximum number of lines for a single note within the data set
2. Convert your note line data set from long form to wide form using your favorite long to wide program (e.g. PROC TRANSPOSE).
3. 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 pilcrows, ¶. 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 (e.g. just like % or #).

Example: Remove variability from the outset 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, as would *temp: 101* and *temp 101*. As long as these distinctions or symbols are unimportant for the task at hand, their removal is 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. Be sure to read on for more examples of PRX functions if substitutions or string manipulation are not your intended goal.

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

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 crafted by humans. 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=avoiding 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. We discuss the results of validating our own algorithm later.

PARSING OPERATING ROOM NOTES FOR SURGICALLY MANAGED IUD PERFORATIONS

In this subsection, we’ll build on the basic code from the first section to parse several target regular expressions. We then use PRXSUBSTR to create substrings that are printed as part of the RegEx building and troubleshooting process.

The data set used in this section represents a cohort of women with a known IUD insertion who subsequently had an operating room record that mentioned ‘IUD’, ‘Intrauterine device’, ‘Intrauterine contraceptive device’, or ‘Uterus perforation’ in the standardized diagnosis or procedure list. We excluded women whose operating room event happened after a known IUD removal or if the operating room event indicated an IUD insertion following an un-coded removal. We also flagged operating room events where laparoscopy or pelviscopy procedures were performed – the initial review process found a strong correlation between these procedures and removal of perforated IUDs.

Example: Parsing several target RegEx using PRXPARSE

In consultation with our clinical experts, we decided that most uterine perforations discussed in operating notes could be captured using the following concepts:

1. Discussion of IUD embedded or embedded IUD
2. Literal mention of IUD perforation
3. The word IUD discussed in close context with anatomy not normally associated with an IUD (e.g. abdomen, peritoneal cavity, fimbra)
4. IUD followed by words such as buried, stuck, encased, adhesed
5. Verbs not normally associated with IUD removal such as free, transect, tease

In the next coding example, we show how different matches can be selected and then printed to verify their individual validity. Though capturing different clinical concepts, the regular expressions below use common syntactic tools:

- Variation in keywords (e.g. IUD or IUS or Intrauterine device or Mirena or Paragard) captured using grouping `()` and alternation `|` metacharacters
- Differences in spelling captured using brackets `[AB]`, wildcards (especially for vowels known to be big sources of typos), and optional characters `?`
- Pairs of keywords (e.g. IUD and embed) allowed to be separated using a min and max distance `{#,#}`
- Two different regular expressions written if keyword order can be alternated (e.g. IUD embed vs. embed IUD)
- 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
- The concept IUD is captured in each `re` below with the RegEx snip: `(iu[ds]|int.{0,3}ut.r.n.{1,2}d|m.rr?.nn?a|p.r.g.{1,2}rd)`

The following program parses a concatenated note data set for several different regular expressions:

```
data parse;
  set concat;
  if _N_ = 1 then do;
    /*1*/re1=prxparse("/(iu[ds]|int.{0,3}ut.r.n.{1,2}d|m.rr?.nn?a|p.r.g.{1,2}rd).{1,100}[ie]mbed/i");
    /*1*/re2=prxparse("/([ie]mbed|p[or]{1,2}trud).{1,100}(iu[ds]|int.{0,3}ut.r.n.{1,2}d|m.rr?.nn?a|p.r.g.{1,2}rd)/i");
    /*2*/re3=prxparse("/(iu[ds]|int.{0,3}ut.r.n.{1,2}d|m.rr?.nn?a|p.r.g.{1,2}rd).{1,100}perf.r.t/i");
    /*3*/re4=prxparse("/(iu[ds]|int.{0,3}ut.r.n.{1,2}d|m.rr?.nn?a|p.r.g.{1,2}rd).{1,50}(abdom[ei]n|oment(um|al)|xxx
          |pelvi[sc]|paracolic gutter|cul.{0,2}de.{0,2}sac|ovarian fossa|[lr][ul]q|xxx
          |(left|right) (upper|lower) quad|epiploica|fimbra|perit.ne(um|al) cav|sacrum)/i");
    /*3*/re5=prxparse("/(abdom[ei]n|extra.?uterine).{1,30}(iu[ds]|int.{0,3}ut.r.n.{1,2}d|m.rr?.nn?a|p.r.g.{1,2}rd)/i");
    /*4*/re6=prxparse("/(iu[ds]|int.{0,3}ut.r.n.{1,2}d|m.rr?.nn?a|p.r.g.{1,2}rd).{1,30}(p[or]{1,2}trud|buried|xxx
          |stuck|encase|encaps.lat|adhe[sr]e[dn])/i");
    /*5*/re7=prxparse("/(iu[ds]|int.{0,3}ut.r.n.{1,2}d|m.rr?.nn?a|p.r.g.{1,2}rd).{1,15}(free|transect|teased)/i");
  end;
  retain re1-re7;
run;
```

Example: Iterating through RegEx design using CALL PRXSUBSTR and SUBSTRN function

Ultimately, only a single binary variable ‘uterine_perf’ was created, but in the process of creating the NLP algorithm, each RegEx `re` was inspected to see if we were capturing what we expected to capture. Remember, only a sample of notes was reviewed to help define the initial RegEx. The following program locates the position and length of each of our seven previously defined RegEx using CALL PRXSUBSTR. The SUBSTRN function is used to snip out the RegEx match with surrounding text, which is then printed (Output 2):

```
data substring;
  set parse;
  /*1*/call prxsubstr(re1, fullopnote, pos, len);
  if pos NE 0 then embed_str1=substrn(fullopnote, pos-60, len+90);

  /*1*/call prxsubstr(re2, fullopnote, pos, len);
  if pos NE 0 then embed_str2=substrn(fullopnote, pos-60, len+90);

  /*2*/call prxsubstr(re3, fullopnote, pos, len);
  if pos NE 0 then perf_str=substrn(fullopnote, pos-60, len+90);
  /*...etc...*/
run;

proc print data=substring(where=(embed_str1 NE '')); var embed_str1;
proc print data=substring(where=(embed_str2 NE '')); var embed_str2;
proc print data=substring(where=(perf_str   NE '')); var perf_str;
/*...etc...*/
run;
```

```
Obs embed_str1
 68 Paraguard IUD without strings embedded in the posterior uterine w
 91 IUD imbedded to adhesive comeplex with
124 Mirena IUD not embedded, string in cavity, easily

...

Obs perf_str
  4 ¶Findings ¶Perforated IUS with lateral arms perforating through fundus
 15 iud removed. There was no evidence indicating perforation, and a nor
200 nctional cyst on left ovary.  IUD in posterior cul de sac, perforating
```

Output 2. Partial output from PROC PRINT following CALL PRXSUBSTR and SUBSTRN

Example: Avoiding negated phrases using PRXPARSE, PRXMATCH, and PRXPOSN

One of the biggest challenges encountered in this project was dealing with negated phrases (obs 124 above 'not embedded' and obs 15 'no evidence indicating perforation'). Depending on the size of the job, these could either be removed manually or through additional language processing. The code below uses several iterative steps to flag negated phrases (`negflag`) within 6 words before 'embed'. The results are printed and shown in Output 3.

- First, regular expressions needed for each step of this process are parsed using the now familiar **prxparse** function
- In `re1` the code `[^.¶]` indicates that all characters that are not a period or pilcrow should be captured. Note that inside brackets, the period (.) does not need to be escaped with \. The goal is pulling all characters between the end of the last sentence and the keyword embed or imbed. This particular RegEx is computationally expensive, so we search the source `embed_str1` rather than `fullopnote`
- In this example, **prxmatch** is used as a Boolean function (true/false: is `re1` found in `embed_str1`), followed by **prxposn**, which snips out the substring that matches the specified capture buffer from the RegEx. If capture buffer=0, the full RegEx is assigned to the snip, e.g. above RegEx as `sentence`
- `re2` uses the any word character `/w` wildcard and any non-word `/W` wildcards, in combination with repetition factors, to identify 0 to 6 words in the sentence preceding the word embed or imbed. In the second line beginning with **prxmatch**, 0 to 6 words are snipped from `sentence` using **prxposn** and assigned to the variable `sixwords`. Note that **prxposn** only works after first using **prxmatch** or **prxchange**
- Finally, `sixwords` is searched for `re3`, which identifies negation terms, and `negflag` is assigned the Boolean result of **prxmatch**

Here's the code:

```
data parse_neg;
  set substring;
  if _N_ = 1 then do;
    re1=prxparse("/[^.¶]*[ie]mbed/i");
    re2=prxparse("/((\w+\W*){0,6})[ie]mbed/i");
    re3=prxparse("/\bno |not |negative|neg |history|h\/o/i");
  end;
  retain re1 re2 re3;
  if embed_str1 NE '' then do;
    sentence=prxposn(re1, 0, embed_str1);
    if prxmatch(re2, sentence)>0 then sixwords=prxposn(re2, 1, sentence);
    negflag=(prxmatch(re3, sixwords)>0;
  end;
run;

proc print data=parse(where=(sentence NE ''));
var sentence sixwords negflag; run;
```

```
Obs sentence                               sixwords                        negflag
 68 Paraguard IUD without strings embed    Paraguard IUD without strings         0
 91 IUD imbed                              IUD                                   0
124 Mirena IUD not embed                   Mirena IUD not                        1
```

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

NEEDLE IN A HAYSTACK: FINDING IUD PERFORATIONS DISCOVERED WITHOUT SURGICAL MANAGEMENT

As mentioned at the start of this section, not all IUD perforations are surgically managed. If an IUD is only partially embedded in the uterus, it can sometimes be removed in an outpatient clinic. The above strategy was repeated using clinic notes. First, clinic notes were identified that had some coded suggestion of an IUD issue. RegEx related to the concepts ‘embed’ and ‘perforation’ were borrowed from the previous sub-section and searched for among flagged clinic notes. At first, when we executed this plan, we found many more IUD perforations than expected. On inspection, most of these notes contained a standardized warning, which preceded the IUD insertion itself – e.g. ‘Discussed with patient risk of IUD perforation…’.

In the examples above, the PRX functions return the first match found in the source of the specified RegEx. In our notes with a standard warning, the first match is the warning itself. But what if we want to know if a perforation is described further along in the note? **Call prxnext** is one tool available which easily handles repetition of a target phrase.

Example: Skipping the standard warning with CALL PRXNEXT and PRXMATCH

Call prxnext searches in source between positions start and stop (initially set to 1, and length(source) respectively). It returns pos and len, and also resets start to pos+ len+1, so another search can easily begin where this one left off.

- The first RegEx captures the concept *IUD perforation*, with up to 50 characters between the first and second word and variations/typos in the keywords. Brackets `[]` and `^` indicate that there should not be a comma after the perforation keyword as initial review indicated that this is an almost sure indicator of listed risks in a standard warning
- The first time **call prxnext** is used, it acts identically to **call prxsubstr**, searching from the beginning to end of `fullclinicnote`. The matched text and surrounding characters are snipped out using the **substrn** function. `start` is automatically reset to `pos+len+1`
- The snip `perf_str` is then searched for warning keywords, as specified in the second RegEx `re2`, and, if found, `perf_str` is reset to null
- The second time **call prxnext** is used, the call routine begins searching for `re1` at the new `start` position. If a second match is found, it is snipped out using the **substrn** function as before

```
 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|p.r.g.{1,2}rd).{0,50}perf.r.t(ed?|ion)[^,]/i");
     re2=prxparse("/(risk|precaution|r\Wo||warning/i");
  end;
  retain re1 re2;
  start=1;
  stop =length(fullclinicnote);
  call prxnext(re1, start, stop, fullclinicnote , pos, len);
  if pos>0 then do;
    perf_str=substrn(fullclinicnote, pos-50, len+70);
    if prxmatch(re3, perf_str)>0 then perf_str='';
    call prxnext(re1, start, stop, fullclinicnote , pos, len);
    if pos>0 then perf_str=substrn(fullclinicnote, pos-30, len+60);
  end;
run;
```

RESULTS: POSITIVE PREDICTIVE VALUE

Among a cohort of 168,744 IUD insertions, 444 potential IUD perforations were identified using the strategy detailed here. A clinical expert reviewed a random sample of 100 patients using the full electronic record. The positive predictive value was 77%, or 77 of 100 potential perforations reviewed were true perforations; 9 of 100 could not be confirmed due to ambiguity in the medical record and/or in the clinical presentation of the case.

Noteworthy, 54 of 58 suspected perforations identified from operating notes were true perforations (PPV=93%). Only 23 of 42 suspected perforations that were not surgically managed could be confirmed as true perforations. Most perforations that are managed in the operating room are complete perforations, where the IUD passed completely through the uterus. Whereas perforations managed outside the operating room are almost entirely partial perforations. From both a clinical and programming point of view, the language used for partial perforations is much less clear.

The data reported here is part of an ongoing study. Ultimately, the final project will combine text search with chart review confirmation, especially for non-surgical cases.

CONCLUSION

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

REFERENCES

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

“Tables of Perl Regular Expression (PRX) Metacharacters.” *SAS 9.2 Language Reference: Dictionary, Fourth Edition*. Available at: http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a003288497.htm

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.

ACKNOWLEDGMENTS

The research team at KPNC that worked on this project included Maqdooda Merchant, Debbie Posthlethwaite, and Tina Raine-Bennett. In addition, we’d like to acknowledge Liyan Liu, who has contributed extensively to natural language processing education at the KPNC Division of Research.

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

Monday Tips:

1. Use PRXPARSE to define a text string or pattern (think wildcards and optional characters, etc). Then use CALL PRXSUBSTR to locate the starting position and length of your target pattern within a text source variable.
2. Wildcards (\d for any digit, \w for any word character, . for *any* character) make SAS PRX functions wildly fun to use and are great for dealing with typos in text string analysis.
3. A single line of code can remove extraneous characters from a text block, making further string analysis using SAS PRX functions even easier: `clean-text=PRXCHANGE("s/[^a-zA-Z0-9¶., ]//", -1, note-text);`