Paper CT09

Perl Regular Expressions in SDTM and ADaM Programming: Live Study Examples

Pavol Valovic, Premier Research, Inc., Bratislava, Slovakia

ABSTRACT

In clinical research, free text fields are often used when the clinical site staff wants to include important information regarding adverse events, concomitant medications, or any other result deemed important for the investigator. Extracting useful data from free text fields is often difficult, however, Perl Regular Expressions (PRX) present a unique solution to this problem. PRX are a powerful tool for determining patterns of text within a string. SAS supports the usage of PRX and provides a great opportunity to tackle day-to-day tasks in determining data patterns in free text fields. Many papers have already been published on the general topics of PRX. In our submission, we present actual examples of PRX in live studies and discuss cases in which the usage of very complex PRX should be used cautiously and other programming techniques might be more suitable.

INTRODUCTION

Free text fields can be found in any clinical trial submission. Site staff usually provide descriptive explanations or descriptions of adverse events, concomitant medications, and abnormal findings during assessments. However, free text fields might contain information which was incorrectly submitted in free text field and would be better captured in a pre-formatted field. Therefore, programmers must implement complex strategies to either extract information from the free text fields or check the free text fields for the strings used in downstream conditions. A number of papers have already explained technical background of Perl Regular Expressions (PRX) in SAS. While this knowledge is important, in this paper, we will explore the real-world use of PRX in clinical trial submission. All the examples presented in this paper were either used by the author or colleagues in actual clinical trials. For readers unfamiliar with PRX in SAS, we recommend to first read referenced papers.

The PRX used in this paper provide useful insights for statistical programmers regarding its potential use in their day-to-day work.

FINDING TEXT STRINGS IN FREE TEXT USING PRXMATCH

In the first example, sponsor wanted to capture adverse events of special interest occurring in the treated (index) knee or associated tissues near the knee. Unfortunately, anatomical location was not captured in the EDC, and we had to extract the conditions from the free text field AETERM. First, sponsor asked us to check whether an adverse event of special interest occurred in the knee using the following terms:

A = Event that AETERM contains the strings
("KNEE", "PATELLA", "MENISCUS", "FEMUR", "FEMORAL", "LIGAMENT", "FIBULA", "FIBULAR", "TIBIA", "TIBIAL", "QUADRICEP TENDON", "INJECTED")

To find the occurrence of these terms in variable AETERM, we have created a simple PRX using the function `prxmatch`:

```
if prxmatch("m/KNEE|PATELLA|MENISCUS|FEMUR|FEMORAL|LIGAMENT|FIBULA|FIBULAR|TIBIA|
TIBIAL|QUADRICEP TENDON|INJECTED/i", aeterm) > 0
  then condition_A = 1;
  else condition_A = 0;
```

The ‘m’ tag at the beginning of PRX is used so the procedure PRXMATCH tries to find the exact match of search string in the AETERM. The ‘i’ tag at the end forces a case insensitive match (“KNEE” is the same as “knee” or “Knee” or “knEe”).

After the condition_A is met (equals to 1), we needed to eliminate the possibility that an adverse event occurred in the non-treated (non-index) knee:

B = Event that AETERM contains the strings
("NON-INDEX KNEE", "NON- INDEX", "NON INDEX" "NOT INDEX KNEE", "NOT ON INDEX KNEE" "NO INDEX KNEE", "NO INDEX", "NOT INJECT", "NON INJECT", "NOT ON INJEC", "NOT IN INJEC", "NON-INJECT")

Again, a very simple PRX took care of this task:

```
if prxmatch("m/NON-INDEX KNEE|NON- INDEX|NON INDEX|NOT INDEX KNEE|NOT ON INDEX KNEE|
NO INDEX KNEE|NO INDEX|NOT INJECT|NON INJECT|NOT ON INJEC|
NOT IN INJEC|NON-INJECT/i", aeterm) > 0
  then condition_B = 1;
  else condition_B = 0;
```

Finally, a third condition was considered. We had to look for a laterality of adverse event (left, right, bilateral) and compare it with the subject-level index knee variable from ADSL as follows:

C = Event that AETERM contains strings ("Right", "Left", "Bilateral")

With following PRX to match the strings:

```
if prxmatch("m/RIGHT|LEFT|BILATERAL/i", aeterm) > 0
  then condition_C = 1;
  else condition_C = 0;
```

These three conditions (condition_A, condition_B, condition_C) were then used in downstream derivations that are out of scope of this paper.

Our first example demonstrates how a PRX can be used in place of multiple `index() or find()` functions, which are often lengthy (requiring a unique `index() or find()` function to be used for each term) and difficult to navigate. Instead, PRX is efficient and can be easily updated as required.

REPLACING A PATTERN OF TEXT USING PRXCHANGE

In concomitant medication listing, indications for medications were collected in the database as free text. If the indication was an adverse event or medical history, the event was referenced using the string “AEnnn_” or “MHnnn_” where *nnn* is the row number of adverse event/medical history collected on the appropriate CRF page. In addition, multiple strings were concatenated and printed as follows:

MH003_STAGE III
CHRONIC KIDNEY
DISEASE
MH004_HYPERTENSION

This means subject had 2 different indications for concomitant medication (“STAGE III CHRONIC KIDNEY DISEASE” and “HYPERTENSION”). A request has been made to remove the text strings referring to medical history records and to keep only sensible text in the listing.

The first part of the problem is to remove the string pattern from the start of the indication string. A function `prxchange` is used as follows:

```
indication = prxchange('s/^[MHAE]{2}\d{3}_//', -1, indication);
```

Let’s break down the PRX. The `prxchange` function replaces string pattern with any user input. The ‘^’ symbol matches the position at the beginning of the input string, which means that `prxchange` will be trying to match the search pattern only at the start of the indication text (as intended in this step). String `[MHAE]{2}` tries to match any two-letter combination of MHAE letters (remember, we are looking for either ‘MH’ or ‘AE’). Next, `d{3}` matches any three digits followed by the underscore ‘_’. The `//` part at the end of the PRX substitutes the matching search pattern with blank space. The ‘`-1`’ statement ensures that matching patterns continue to be replaced until the end of the indication string is reached. To summarize, PRX attempts to find the match at the beginning of the indication text consisting of two letters, followed by three digits and an underscore character and replaces it with blank space. However, we still want to deal with similar string in the middle of the text.

If we were to omit the ‘^’ symbol at the beginning of the PRX, all patterns in the text would be replaced by blank spaces. Final text string from the example would then look like “STAGE III CHRONIC KIDNEY DISEASE

HYPERTENSION" which is a little bit ambiguous. Therefore, this is not a solution. Instead, we employ a second `prxchange` function as follows:

```
indication = prxchange('s/[MHAE]{2}\d{3}_/+ /', -1, indication);
```

After basic inspection, this PRX appears very similar to the first PRX with few differences. This PRX attempts to find the matching pattern anywhere in the string but, more importantly, substitutes the pattern with string '+ '. Because we have already used the first PRX to replace the starting pattern with blank space, we do not have to worry about changing the starting 'MH003_' for '+ '. After both PRX were used, this was the final indication text:

STAGE III CHRONIC KIDNEY
DISEASE + HYPERTENSION

Meaningful text is preserved, and indications are concatenated with '+' sign for increased clarity of the text.

USING BOTH PRXMATCH AND PRXCHANGE COMBINED WITH OTHER DATA STEP FUNCTIONS

In the third and last example, we explore a free text issue that is prevalent in statistical programming. We are referring to submitting the subject ID in the protocol deviation logs. Usually, the subject ID in protocol deviation reporting systems is a free text field, where study staff can either fill out a correct single subject ID (the most ideal case), list multiple subjects per deviation, or provide free text. In SDTM, a protocol deviation domain (SDTM.DV) must be submitted as one record per protocol deviation per subject. This creates a challenging task to restructure the raw protocol deviation log into tabulated version.

In the study example provided below, subject IDs were collected in the '01-XXX' format. In the protocol deviation log, site staff sometimes submitted subject IDs in the '1XXX' format, which is incorrect. Six different types of subject ID texts were submitted: blank subject ID, single subject ID in correct format, subject ID submitted in incorrect format, free text without any subject ID, multiple subject IDs in correct format, and multiple subject IDs in incorrect format.

SUBJID	PD_TEXT
	Sample Protocol Deviation text.
01-053	Sample Protocol Deviation text.
01-053	Sample Protocol Deviation text.
1104	Sample Protocol Deviation text.
1104	Sample Protocol Deviation text.
1101	Sample Protocol Deviation text.
See details in actions	Sample Protocol Deviation text.
01-122	Sample Protocol Deviation text.
1103, 1104, 1107, 1206, 1208	Sample Protocol Deviation text.
01-122	Sample Protocol Deviation text.
01-134	Sample Protocol Deviation text.
	Sample Protocol Deviation text.
01-137	Sample Protocol Deviation text.
1106	Sample Protocol Deviation text.
01-021, 01-071, 01-077	Sample Protocol Deviation text.
01-118	Sample Protocol Deviation text.
01-025	Sample Protocol Deviation text.
01-039, 01-046, 01-049, 01-070, 01-097	Sample Protocol Deviation text.
01-053, 01-055, 01-062, 01-110, 01-122, 01-128, 01-132	Sample Protocol Deviation text.
01-056	Sample Protocol Deviation text.
01-062	Sample Protocol Deviation text.
01-062	Sample Protocol Deviation text.
01-034	Sample Protocol Deviation text.

First, we must remove rows where the subject ID is either missing or not starting in the subject ID format (either correct or incorrect). We can use PRX or other programming techniques.

With PRX:

```
if missing(subjid) or (prxmatch("m/^01\-\d{3}/i", subjid) = 0 and
prxmatch("m/^1\d{3}/i", subjid) = 0) then delete;
```

The first `prxmatch` attempts to find the '01-XXX' pattern (correct subject ID format) at the beginning of the string, and the second `prxmatch` attempts to find the '1XXX' pattern (incorrect subject ID format). If the subject ID is missing or the string does not start with '01-XXX' or '1XXX' pattern, then this row is deleted.

Alternatively, since subject ID can start only with '1' or '0', simpler PRX can be constructed:

```
if missing(subjid) or (prxmatch("m/^0/i", subjid) = 0 and prxmatch("m/^1/i", subjid) =
0) then delete;
```

Without PRX:

```
if missing(subjid) or (subjid ^=: '0' and subjid ^=: '1') then delete;
```

All three sections of code have the same outcome (three rows are deleted). Not using complex PRX is favorable in this case because subject ID strings can only start with either '0' and '1'. Simple example of PRX is on the same level of complexity as not using PRX at all. However, in the case of more variable subject ID patterns, a potential more complex PRX is more suitable.

Now that we have removed unwanted rows from the dataset, let's move on to the next step, which is transforming incorrectly formatted ('1XXX') subject IDs into correct format ('01-XXX'). For this purpose, we implement a slightly more difficult PRX:

```
if prxmatch("m/1\d{3}/i", subjid) = 1 then subjid = prxchange("s/1(?=\d{3})/01-/", -1,
subjid);
```

The `prxmatch` finds '1XXX' string anywhere in the text (doesn't have to be at the beginning, since we can have multiple subject IDs delimited with comma). If a match is found, then in `subjid` variable, we want the number '1' to be replaced with '01-'. However, we must ensure that only the first '1' in the pattern is replaced (for example '1104'), unless we end up with subject ID as '01-01-04' which is not correct. Because of this, we use (?=) metacharacter, which represents a look-ahead assertion. The PRX `"s/1(?=\d{3})/01-/` replaces string '1' which is followed by any three digits (`\d{3}`) and replaces the '1' with '01-'. The look-ahead metacharacter ensures that any three digits are used only to check the condition but are not included in the final match. Without the look-ahead metacharacter, this PRX would replace whole '1XXX' pattern with '01-' which is not what we want.

After this step, all subject IDs are in the same correct format:

SUBJID	PD_TEXT
01-053	Sample Protocol Deviation text.
01-053	Sample Protocol Deviation text.
01-104	Sample Protocol Deviation text.
01-104	Sample Protocol Deviation text.
01-101	Sample Protocol Deviation text.
01-122	Sample Protocol Deviation text.
01-103, 01-104, 01-107, 01-206, 01-208	Sample Protocol Deviation text.
01-122	Sample Protocol Deviation text.
01-134	Sample Protocol Deviation text.
01-137	Sample Protocol Deviation text.
01-106	Sample Protocol Deviation text.
01-021, 01-071, 01-077	Sample Protocol Deviation text.
01-118	Sample Protocol Deviation text.
01-025	Sample Protocol Deviation text.
01-039, 01-046, 01-049, 01-070, 01-097	Sample Protocol Deviation text.
01-053, 01-055, 01-062, 01-110, 01-122, 01-128, 01-132	Sample Protocol Deviation text.

Finally, the last step is to find the subject ID pattern and output one row per subject ID found. First, we use PRX to determine the count of '01-XXX' patterns in the SUBJID string, which gives us the number of subject IDs. We begin by defining the basic setup:

```
expression = prxparse("/01\-\d{3}/");
start = 1;
finish = length(subjid);
count = 0;
```

The `expression` variable contains the identification number of the target pattern ('01-XXX'). The `finish` is simply the length of the whole subject ID string, and we initialize `count` variable, which holds the actual number of subject IDs in the string.

After this setup, we run the first CALL PRXNEXT routine. This routine returns the position and length of a pattern match located between the `start` and `stop` positions in the source string (subject ID). Because the value of the start parameter is updated to be the position of the next character that follows a match, CALL PRXNEXT enables us to search a string for a pattern multiple times in succession.

```
call prxnext(expression, start, finish, subjid, position, length);
```

The first CALL PRXNEXT routine returns `position = 1`, because each SUBJID string contains at least one matching pattern. The variable `position` is used as the foundation of an iterative do-loop.

```
do while (position > 0);
  count+1;
  call prxnext(expression, start, finish, subjid, position, length);
end;
```

This do-loop runs while `position` is non-zero, or in other words while CALL PRXNEXT can still find a matching pattern in the subject ID string. For each found pattern, `count` is incremented by 1, giving the final number of subject IDs in the string.

SUBJID	PD_TEXT	count
01-053	Sample Protocol Deviation text.	1
01-053	Sample Protocol Deviation text.	1
01-104	Sample Protocol Deviation text.	1
01-104	Sample Protocol Deviation text.	1
01-101	Sample Protocol Deviation text.	1
01-122	Sample Protocol Deviation text.	1
01-103, 01-104, 01-107, 01-206, 01-208	Sample Protocol Deviation text.	5
01-122	Sample Protocol Deviation text.	1
01-134	Sample Protocol Deviation text.	1
01-137	Sample Protocol Deviation text.	1
01-106	Sample Protocol Deviation text.	1
01-021, 01-071, 01-077	Sample Protocol Deviation text.	3
01-118	Sample Protocol Deviation text.	1
01-025	Sample Protocol Deviation text.	1
01-039, 01-046, 01-049, 01-070, 01-097	Sample Protocol Deviation text.	5
01-053, 01-055, 01-062, 01-110, 01-122, 01-128, 01-132	Sample Protocol Deviation text.	7
01-056	Sample Protocol Deviation text.	1
01-062	Sample Protocol Deviation text.	1
01-062	Sample Protocol Deviation text.	1
01-034	Sample Protocol Deviation text.	1
01-068	Sample Protocol Deviation text.	1
01-062, 01-069	Sample Protocol Deviation text.	2
01-070, 01-074	Sample Protocol Deviation text.	2

Finally, a simple combination of do-loop, SCAN function, and OUTPUT statement creates one record per protocol deviation per subject:

```
do i = 1 to count;
  subjid_new = strip(scan(subjid, i, ','));
  output;
end;
```

SUBJID	PD_TEXT	count	SUBJID_NEW
01-053	Sample Protocol Deviation text.	1	01-053
01-053	Sample Protocol Deviation text.	1	01-053
01-104	Sample Protocol Deviation text.	1	01-104
01-104	Sample Protocol Deviation text.	1	01-104
01-101	Sample Protocol Deviation text.	1	01-101
01-122	Sample Protocol Deviation text.	1	01-122
01-103, 01-104, 01-107, 01-206, 01-208	Sample Protocol Deviation text.	5	01-103
01-103, 01-104, 01-107, 01-206, 01-208	Sample Protocol Deviation text.	5	01-104
01-103, 01-104, 01-107, 01-206, 01-208	Sample Protocol Deviation text.	5	01-107
01-103, 01-104, 01-107, 01-206, 01-208	Sample Protocol Deviation text.	5	01-206
01-103, 01-104, 01-107, 01-206, 01-208	Sample Protocol Deviation text.	5	01-208
01-122	Sample Protocol Deviation text.	1	01-122
01-134	Sample Protocol Deviation text.	1	01-134
01-137	Sample Protocol Deviation text.	1	01-137
01-106	Sample Protocol Deviation text.	1	01-106
01-021, 01-071, 01-077	Sample Protocol Deviation text.	3	01-021
01-021, 01-071, 01-077	Sample Protocol Deviation text.	3	01-071
01-021, 01-071, 01-077	Sample Protocol Deviation text.	3	01-077
01-118	Sample Protocol Deviation text.	1	01-118
01-025	Sample Protocol Deviation text.	1	01-025

CONCLUSION

Free text fields cause many headaches for beginners and experienced statistical programmers. In particular, free text fields contain patterns of information that should be better collected in pre-formatted fields. We present a pattern matching routine using PRX in SAS that effectively handles the patterns of text in the strings.

In three examples, we examined common problems encountered by programmers in live studies and provided fundamental knowledge about the use of PRX. Of course, there might be unique problems with very complex patterns and PRX, which are not in the scope of this paper. However, the proposed paper provides a good starting point for exploring the complex and beautiful world of PRX.

For purposes of testing and debugging the PRX, we provide a useful link to a simple online tool in Recommended Reading section.

REFERENCES

Pattern Matching Using Perl Regular Expressions (PRX)
https://documentation.sas.com/doc/en/pgmsascdc/v_055/lefunctionsref/n13as9vjfj7aokn1syvfyrpaj7z5.htm

Using Perl Regular Expressions in the DATA Step
https://documentation.sas.com/doc/en/pgmsascdc/v_055/lefunctionsref/p1vz3ljudbd756n19502acxazevk.htm

Tables of Perl Regular Expression (PRX) Metacharacters
https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lefunctionsref/p0s9ilagexmjl8n1u7e1t1jfnzlk.htm

ACKNOWLEDGMENTS

I would like to acknowledge my colleagues Peter Slezak and Knut Muller for providing useful insight and support during the preparation of this paper.

RECOMMENDED READING

regex101: build, test and debug regex
https://regex101.com/

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:
Author Name: Pavol Valovic
Company: Premier Research, Inc.
Address: Sevcenkova 3370/34, Bratislava, Slovakia
Work Phone: +421905425792
Email: pavol.valovic@premier-research.com
Website: https://www.linkedin.com/in/pavolvalovic/