Perl Regular Expression in SAS Macro Programming

Yanwei Zhang, CNA Insurance Company, Chicago, IL

Abstract

In this paper, the Perl regular expression facility that provides a concise and flexible means for matching strings of text is extended to the macro environment using three new macro functions. [1] Consequently, this allows direct pattern matching and replacement in macro variables, facilitating the construction of flexible and customized functions through internally parsing patterned inputs. This is illustrated with a specific example where we outline the key steps to build an automated variable selection process in the generalized linear models, with additional capability to perform joint test of several variables such as spline bases.

1 Introduction

Perl regular expression [PRX] is a powerful string manipulation tool, which provides a concise and flexible means for developing complex pattern-matching and textual search-and-replace algorithms. SAS 9 incorporates this facility into the data step and some procedures. As a result, many labor intensive and sophisticated tasks that involve complicated text manipulation can be readily handled using the built-in functions and call routines that rely on Perl regular expression. A comprehensive overview of the set of SAS PRX functions can be found in Cody (2004). The current paper presents a natural extension of this functionality to the macro environment via three new macro functions that employ the power of the data-step PRX functions, along with enhancements and additions. It is therefore possible to implement flexible pattern search and replacement within macro variables, which could essentially benefit the general macro programming in terms of efficiency, robustness and usability. Specifically, employing Perl regular expression in the macro environment can

- Assist in the tedious but critical work of validating macro arguments;
- Help construct user-friendly macros that support internal parsing of patterned inputs.

After a brief review of Perl regular expression in the next section, we will introduce and illustrate the use of the three new macro functions in creating a customized and user-friendly program to implement an automated variable selection process in the generalized linear models in Section 3. Section 4 will provide closing comments.

2 Perl regular expression

Regular expression is a tiny and highly specialized programming language in its own right. It is a series of characters that defines a pattern to search for within a target string. Indeed, regular expressions have appeared in SAS since version 6.12. They are generally referred to as SAS regular expression, to be distinguished from the Perl regular expression [PRX] that was made accessible in SAS 9. The two groups of regular expressions have different syntax, and this paper will only focus on Perl regular expression. The following gives a brief introduction of the syntax of Perl regular expression, and for a comprehensive overview, see Schwartz *et al.*(2005). Since regular expression is often called pattern in Perl, these two names are used interchangeably in the following discussions.

[1] The source code along with a help documentation can be downloaded from `http://www.actuaryzhang.com/software.html`.

Perl regular expression offers two operations on strings, one being the match of a pattern using the operator `m//`, and the other being the search and substitution of patterns using the operator `s///`. Regular expression (pattern) needs to be inserted between the delimiters `//`. The `m` in the matching operator `m//` is almost always left out for simplicity. The substitution operator `s///` replaces whichever part of a string matches the first pattern with a replacement string specified in the second pattern.

The simplest form of a pattern match is on literal strings. For example,

```
/cat/ is matched in "cat" and "wildcat", but not in "car".
```

By default, a pattern can be matched anywhere in a string, but this behavior can be changed using a number of anchors to specify the position of the pattern in relation to the text. The caret `^` and the dollar sign `$` specify that the match must occur at the beginning or the end of a string, respectively. The word-boundary anchor `\b` matches either end of a word made up of ordinary letters, digits and underscores. For example,

```
/^cat/ is matched in "cater" but not in "wildcat";
/cat$/ is matched in "wildcat" but not in "cater";
/cat\b/ is matched in "wildcat" but not in "cats".
```

The anchors `^` and `$` in the above case are examples of metacharacters, a number of special characters that are assigned special meanings. It is through the use of these metacharacters that regular expressions express real power in textual search and replacement. For example, the dot `.` matches any single character except a newline. Other metacharacters such as `* + ? ( )` and so on will be discussed shortly.

Another important concept is the character class, which allows a set of possible characters, rather than just a single character, to match at a particular point in a pattern. Character classes are denoted by brackets `[]`, with the set of characters to be possibly matched inside. For example, `[aenwy]` may match any one of those five characters. Within character classes, the special character `-` can act as an operator to write a contiguous set of character as a range. As such, the class `[abcde]` can also be written as `[a-e]`. The special character `^`, when used in a character class, denotes a negated character class, which matches any character but those in the brackets. That is, `[^012]` matches any single character but those three. In addition, some frequently used character classes also have shortcuts:

- `\d` matches any digit, the character class `[0-9]` ;
- `\w` matches any word character, the character class `[A-Za-z0-9_]` ;
- `\s` matches any whitespace, the character class of five whitespace characters: form-feed, tab, newline, carriage return and the space character.

It often happens that the same sequence of characters needs to be matched multiple times. This can be accomplished using the quantifier which specifies the quantity of the preceding item. Specifically, the special characters `*`, `+` and `?` mean to match the preceding character zero or more times, one or more times and zero or one time, respectively. For example,

```
/cat\s+/ is matched in "cat  dog" but not in "catalog";
/cat\s*/ is matched in both "cat  dog" and "catalog";
/cats?/ is matched in both "cat" and "cats".
```

It is to be noted that all these quantifiers are greedy, meaning that they will find as many characters as possible to match the regular expression. For example, given the string `"concatenate"`, the pattern `/.+t/` matches `"concatenat"` rather than `"concat"`. For each greedy quantifier, there is a non-greedy alternative (greedy quantifier appended with a question mark `?`), which will find the least number of characters as possible to match the regular expression. Thus, the pattern `/.+?t/` will match `"concat"` in the previous example.

Parentheses `()` can be used for grouping of characters, and `|` can be used for alternation, meaning that either side of the specified pattern may match. For example,

```
/(cat)+/ matches strings like "catcatcat" ;
/cat|dog/ is matched in both "cat" and "dog".
```

Aside from their ability to group characters, parentheses can also trigger the capture buffer, which will hold the part of the string matched by the part of the pattern enclosed in parentheses. A capture buffer is particularly useful when the desired character data can not be identified without using some of the surrounding characters. For example, given the string `"2+3=5"`, the pattern `/=(\d+)/` allows us to extract just the result of the summation `"5"` without `"="`. Multiple buffers can be designated in one pattern and a number is assigned to each according to the order of the opening parentheses.

Since metacharacters (`. \ * ( [ $` and so on) have special meanings, to match them literally, we have to "escape" it by adding a backslash in front. For example, `\*` matches `*`. Also, option modifiers may be appended after the ending delimiter of a pattern to change its behavior from the default. For example, the particularly useful modifier `/i` enables a case-insensitive match. As such,

```
/cat/ does not match "cAt", but /caT/i does.
```

String replacement can be achieved using the substitution operator `s///`. It has two specifications: on the left, the matching regular expression, and on the right, the substitution value. For example, given the string `"cats are not dogs"`, `s/cat/pig/` returns `"pigs are not dogs"`. Also, it is often useful to remember patterns that have been matched on the left so that they can be used again on the right. In the substitution operator, anything matched in a capture buffer gets remembered in the variables `$1,...,$9`, and can be used in the substitution value on the right. For example, `s/(cat|dog)s/$1z/` returns `"catz are not dogz"` in the previous example.

3 Perl regular expression in the macro environment

Perl regular expression can be parsed and utilized to manipulate textual data in the SAS data step using a family of SAS PRX functions. It is particularly effective when dealing with patterned data such as addresses and phone numbers. In fact, Perl regular expression can be applied to manipulate macro variables as the content of a macro variable is always string. In light of that, three new PRX macro functions, `%PRXSEARCH`, `%PRXSEARCHALL` and `%PRXREPLACE`, are built to enable the use of Perl regular expression for pattern matching and replacement in the macro environment. These macros are based upon the data-step PRX functions, along with enhancements and additions. The syntax and the usage of them are summarized in Table 1.

In order to demonstrate how Perl regular expression and the new PRX macros can effectively accomplish and enhance some sophisticated work of macro programming, we will consider the problem of variable selection in the generalized linear models. Unlike the REG procedure, the GENMOD procedure in SAS is not equipped with automated variable selection functionality. Furthermore, even the built-in automated procedures may not meet the needs of practitioners in certain circumstances. For instance, the selection process is sometimes based on groups of variables where the desire is to determine statistically significant groups (e.g., groups representing financial information, demographics and so on). In such a case, focus is on the group and thus variables within one group should be tested jointly to assess the group effect. Another example is when there appears regression splines in the model for smoothing purposes (e.g., see Hastie *et al.* 2008). Splines are represented by a number of basis functions, and hence, it would be more reasonable to analyze the spline term jointly than to test individual basis functions. In the following, we will outline a concise solution to all these problems through the use of Perl regular expression and the new PRX macros. We first highlight the steps to build an automated variable selection process with the GENMOD procedure, and then extend this to

Table 1: Three new macro functions to facilitate the use of Perl regular expression

Macro name	Usage details
%PRXSEARCH(string=, pattern=, buffer=0)	It searches for pattern in string and returns the value of a capture buffer in the first matched occurrence. buffer determines which capture buffer to be returned. The default value of 0 means the whole pattern will be returned. Two global macro variables _pre_buffer and _post_buffer are generated automatically, returning whatever is before and after the buffer, respectively. Multiple blanks are compressed to single blank in the result.
%PRXSEARCHALL(string=, pattern=, buffer=0, collapse=, count=, sep=|)	It is similar to %PRXSEARCH except that it returns all the occurrences of the specified buffer, concatenated with the elements being separated by sep. This result and the total number of matches are stored in two global macro variables with the name in collapse and count, respectively. This function employs %PRXSEARCH internally and keeps calling it until no specified pattern is found. The default delimiter to collapse the stings defaults to |.
%PRXREPLACE(string=, pattern=, times=-1)	It implements pattern replacement in string for a number of times specified in times. pattern must be a substitution operator of the form s///. times defaults to -1, meaning replacement of all matched patterns.

more sophisticated programs that support patterned user inputs for joint test of variables and regression splines. Throughout the discussion, the use of Perl regular expression to manipulate statistical model specifications is emphasized, and details of model estimation is left out.

3.1 Automated variable selection in the generalized linear models

To fix idea, we consider a hypothetical example where a forward variable selection procedure is to be implemented in a Poisson regression with response variable Y and predictors X1-X4 in the original model:

```
%let model= %str(Y= X1 X2 X3 X4 / dist=poisson link=log);
```

The list of candidate variables to be added is stored in the macro variable test. For example,

```
%let test= Z1 Z2 Z3;
```

For simplicity, stopping criterion is chosen to be that if no variable has P-value less than 0.05 then the test procedure terminates. In such a case, the automated variable selection process contains the following logical steps:

1. Fit the current model;
2. Add one variable at a time from the candidate list, fit the new model and calculate P-value of the added variable;
3. If all the P-values are greater than 0.05, stop. Otherwise, update the model by adding the variable with the smallest P-value. Repeat steps 1-3 until the stopping criterion is reached.

These steps indicate that there is an inner loop that goes through the candidate variables, adding one variable at a time to the model statement, and an outer loop that iterates the inner loop until the stopping criterion is satisfied. The outer loop is fairly straightforward (e.g., with a call of %do %while), and we will concentrate on the inner loop in the following discussion.

To proceed, we first pre-process the variable test using the following:

```
%PRXSEARCHALL(string=&test,pattern=%str(/(\w+)/i), buffer=1,
                        count=nTest, collapse=testVars)
%put &nTest **** &testVars;  /* The following appears in the log */
3 **** Z1|Z2|Z3
```

%PRXSEARCHALL searches and extracts all the words (\w+) in the content of test, and creates the macro variable nTest to store the number of variables and testVars to concatenate the variables with each term separated by |. The benefit of delimiting the variables by symbols other than space will become clear in later examples.

Then, the model statement is parsed to extract important information in order to facilitate the addition of the candidate variable:

```
%let indepVars=%PRXSEARCH(string=&model, pattern= %str(/=([^\/]+)\/?/i), buffer=1);
%let indepVarsBefore= &_pre_buffer;
%let indepVarsAfter= &_post_buffer;
%put &indepVars ** &indepVarsBefore ** &indepVarsAfter;  /*The below is in the log*/
X1 X2 X3 X4  ** Y= ** / dist=poisson link=log
```

We see that with one call of %PRXSEARCH, the content of model is broken it into several pieces, i.e., all independent variables and the parts including the response variable and model options. In the pattern, ([^\/]+) defines a buffer that matches any character but / one or more times, and \/? matches the model option delimiter / zero or one time. Strings before and after the specified capture buffer are stored in the two automatic macro variables _pre_buffer and _post_buffer, respectively.

Now, the inner loop can be executed using the following:

```
%macro loop ;
%do i=1 %to &nTest ;
  %let testVar = %scan(&testVars,&i,|);
  %let model_update=&indepVarsBefore &indepVars &testVar &indepVarsAfter;
  %put &i: &model_update ;
  /*  Extra code goes here to fit the GLM and store the p-values  */
%end ;
%mend;
%loop   /* The following appears in the log  */
1: Y=  X1 X2 X3 X4  Z1 / dist=poisson link=log
2: Y=  X1 X2 X3 X4  Z2 / dist=poisson link=log
3: Y=  X1 X2 X3 X4  Z3 / dist=poisson link=log
```

We see that the three variables Z1-Z3 are added to the model statement sequentially and corresponding model can be fitted to obtain the P-value.

Suppose, after all the candidate variables have been tested, Z2 is selected for inclusion in the first iteration. We can update variables model, nTest and testVars for use in the outer loop using

```
%let model=&indepVarsBefore &indepVars Z2 &indepVarsAfter;
%let nTest= %eval(&nTest-1);
%let testVars= %PRXREPLACE(string=&testVars, pattern=%str(s/(\|\s*Z2|Z2\s*\|)//i));
%put &testVars;  /* The following appears in the log */
Z1|Z3
```

The call of %PRXREPLACE simply replaces |Z2 or Z2| in testVars with a white space. Now another iteration can be executed: parse the updated model, call %loop with the new nTest and testVars, and

update `model`, `nTest` and `testVars` again if any variable is selected. The outer loop continues until the stopping criterion is reached or all the candidate variables are added.

3.2 More sophisticated program

As said in the beginning of this section, variable selections sometimes need to be conducted on group of variables or regression splines. It is therefore necessary to customize the above automated program to enable joint test of variables, and in particular, easy specification of spline effects. Perl regular expressions and the PRX macro functions begin to show real power in these more complicated situations.

Suppose, in the first scenario of extension for joint test of variables, one would like to specify variables to be grouped together with a syntax like `(A B C)`. And further suppose the candidate list is:

```
%let test= (Z1 Z2) Z3 (Z4 Z5 Z6);
```

Then, we can count the number of variables/terms to be used in the inner loop as

```
%PRXSEARCHALL(string=&test,pattern=%str(/(\w+|\(.+?\))/i), buffer=1,
                count=nTest, collapse=testVars)
%put &nTest **** &testVars;  /* The following appears in the log */
3 **** (Z1 Z2)|Z3|(Z4 Z5 Z6)
```

The pattern `(\w+|\(.+?\))` matches either a word `\w+` or whatever is enclosed in parentheses `\(.+?\)` using the non-greedy quantifier `+?`. By specifying this, the function `%PRXSEARCHALL` accurately counts the number of terms to be tested. We also see the necessity of the delimiter `|` due to the existence of spaces within parentheses. Then, we need to remove all the parentheses in `testVars` and call `%loop` to implement the inner loop.

```
%let testVars=%PRXREPLACE(string=&testVars, pattern=%str(s/(\(|\))/ /i));
%put &testVars;  /* The following appears in the log */
Z1 Z2 |Z3| Z4 Z5 Z6
%loop  /* The following appears in the log */
1: Y=  X1 X2 X3 X4  Z1 Z2 / dist=poisson link=log
2: Y=  X1 X2 X3 X4  Z3 / dist=poisson link=log
3: Y=  X1 X2 X3 X4  Z4 Z5 Z6 / dist=poisson link=log
```

We see that each group of variables is successfully added to the model statement. The update of `model`, `nTest` and `testVars` can be performed in a similar fashion as before, and is not further pursued here.

Now, consider a second scenario where the candidate list can include regression splines. One may want to specify the spline of variable X with N degrees of freedom as `SPLINE(X,df=N)`, as is consistent with that in the GAM procedure. Suppose such a specification has resulting bases named `X_basis1`, `X_basis2`, ..., `X_basisN`, and we have the following list of candidate variables/terms:

```
%let test= Z1 SPLINE(Z2, df=4) SPLINE(Z3, df=3);
```

We can pre-process the variable `test` by replacing the `SPLINE(...)` term with the indicated basis functions as follows:

```
%let test= %PRXREPLACE(string=&test,
  pattern=%str(s/\bspline\s*\((\w+).+?(\d+)\)/$1_basis1-$1_basis$2/i));
%put &test;  /* The following appears in the log */
Z1 Z2_basis1-Z2_basis4 Z3_basis1-Z3_basis3
```

We see that the two capture buffers extract the name of spline variable and the corresponding degree of freedom, respectively, and then they are used in the substitution value to create the right basis names.

Then we can proceed, similarly as the above, to count the number of terms and execute the inner loop as follows:

```
%PRXSEARCHALL(string=&test,pattern=%str(/((\w|-)+)/i), buffer=1,
                count=nTest, collapse=testVars)
%loop   /* The following appears in the log */
1: Y=  X1 X2 X3 X4  Z1 / dist=poisson link=log
2: Y=  X1 X2 X3 X4  Z2_basis1-Z2_basis4 / dist=poisson link=log
3: Y=  X1 X2 X3 X4  Z3_basis1-Z3_basis3 / dist=poisson link=log
```

3.3 Argument validation

The power of Perl regular expression can also assist in the tedious but important work of macro inputs validation. The following example validates the spline term in `test` to see if the degree of freedom is specified, and if not, set it to be three, along with a warning message.

```
%let test= Z1 SPLINE(Z2) ;
%macro validate ;
%let spline=%PRXSEARCH(string=&test, pattern= %str(/spline\s*\(.+?\)/i), buffer=0);
%if %PRXSEARCH(string=&spline, pattern=%str(/df\s*=/i))= %then %do ;
        %put %str(!WARNING*** NO DEGREE OF FREEDOM SPECIFIED. SET IT TO BE 3!);
        %let test= %PRXREPLACE(string=&test,
                pattern=%str(s/spline\s*\(\s*(\w+).*?\)/SPLINE($1,df=3)/i));
%end;
%put &test;
%mend ;
%validate;   /* The following appears in the log */
!WARNING*** NO DEGREE OF FREEDOM SPECIFIED. SET IT TO BE 3!
Z1 SPLINE(Z2,df=3)
```

4 Conclusion

In this paper, we extend the Perl regular expression facility to the macro environment via three new macro functions. They are demonstrated to be effective in accomplishing and enhancing the work of macro programming. We hope that the power of Perl regular expression in the macro environment will be increasingly appreciated and explored in the future.

5 References

1. Cody R. (2004). *An Introduction to Perl Regular Expressions in SAS 9*, SUGI 29.
2. Hastie T., Tibshirani R., Friedman J. (2008). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer.
3. Schwartz R., Phoenix T., Foy B. (2005). *Learning Perl*, O'REILY.