Paper 091-2013

Information Retrieval in SAS®:
The Power of Combining Perl Regular Expressions and Hash Objects

Lingxiao Qi, Kaiser Permanente Southern California, California, USA;

Fagen Xie, Kaiser Permanente Southern California, California, USA

ABSTRACT

The volume of unstructured data is rapidly growing. Effectively extracting information from these huge unstructured data is a challenging task. With the introduction of Perl regular expressions and hash objects in SAS 9, the combination of these two tools is very powerful in information retrieval. Perl regular expressions is useful in searching and manipulating various complex string patterns. The hash object provides an efficient and convenient way for fast data storage and retrieval. By leveraging both tools on the electronic medical data, we are able to demonstrate how pattern searching on free text can be simplified while reducing coding effort and running time.

INTRODUCTION

The volume of unstructured electronic text is rapidly growing in many areas, including electronic medical records. A simple term such as “disease” embedded in free text can be presented in many different ways. Therefore, finding an approach to accurately and quickly retrieve information becomes an interesting yet challenging topic.

Suppose there is a huge unstructured text document that needs to be scanned through for vital information using hundreds of complex patterns. One approach is to use a bunch of IF THEN and ELSE statements with INDEX, SUBSTR and FIND functions, but that is time-consuming and prone to typing errors. Instead, we could rewrite these complex patterns in concise Perl regular expressions, which can be stored into a hash object, and be searched on using a hash object iterator. This paper will give a few examples of Perl regular expression functions, show how to create hash objects, and finally give a complete program to demonstrate the power of combining these two tools.

PERL REGULAR EXPRESSION (PRX) FUNCTIONS

While Perl regular expressions (combinations of characters and special characters) offer a quick and simple way to define a complex string pattern, its syntax can be a complicated concept to grasp. The focus of this section is to illustrate a few PRX functions rather than drilling into details of Perl syntaxes. Users can find more information from the SAS documentation or the SAS Tip Sheet [1]. This paper will briefly discuss three PRX functions and call routines PRXPARSE, PRXFREE and PRXMATCH, which are used in the final program.

PRXPARSE

This function compiles a string or regular expression and returns a pattern identifier number that can be called later by other PRX functions. The text string or regular expression passed into the parameter needs to be enclosed in either single or double quotes. The following expression finds the string that contains the words “sleepy” and “bunny”. Specifically, it searches for any string that contains “sleepy” followed by any character, followed by a space, followed by “bunny”, followed by any characters or non-character until a word boundary is reached. The compiled expression id is saved in variable **rc**.

```
rc = PRXPARSE('/\b(sleepy\w* bunny)\b/');
```

A function can also be passed into PRXPARSE. The next piece of code shows the SYMGET function that extracts the value of a macro variable named **myvar** and its value is then passed into PRXPARSE.

```
%let myvar = /\b(sleepy\w* bunny)\b/;
rc = PRXPARSE(SYMGET('myvar'));
```

PRXFREE

To release the memory allocated for a regular expression id, use call routine PRXFREE. This takes in a regular expression id, in this case the variable **rc** retruned by PRXPARSE. It is a good coding practice to release memory after a task is completed, especially when memory is limited.

```
Call PRXFREE(rc);
```

PRXMATCH

This function takes in two parameters. The first parameter can either be a string, a Perl regular expression, or a pattern id number returned by a PRXPARSE function. The second parameter is the text string to be searched against. This function returns the first position at which the pattern is found or 0 if not.

Here is an example that searches the word “bunny” in “The bunny on the lawn” and returns position number 5.

```
pos = PRXMATCH('/bunny/', 'The bunny on the lawn');
```

In this example, variable **rc** contains the pattern id from PRXPARSE example, and PRXMATCH returns position 12.

```
pos = PRXMATCH(rc, 'There is a sleepy bunny on the lawn');
```

HASH OBJECTS AND HASH ITERATOR

The hash object is efficient and convenient when it comes to data storage and retrieval. In short, the hash object is essentially a lookup table stored in computer memory with a unique key and data components. The hash object stores and retrieves data based on lookup keys, while the hash iterator object acts as a pointer and iterates the hash object in forward or reverse key order. Although the iterator object is not necessary, it is very useful for direct accessing, adding and removing data entries.

Hash objects are declared in DATA steps. While a hash object can only exist within the duration of its DATA step, multiple hash objects can be created in the same DATA step. This section will briefly introduce the key steps to create a hash object. For more information, check the SAS online documentation or SAS Hash Object Tip Sheet [2].

There are four basic components in a hash object:

- DECLARE statement
- DEFINE method
- DEFINEDATA method
- DEFINEDONE method

DECLARE STATEMENT

The simplest declaration and instantiation of a hash object is made by calling DECLARE, which allocates memory for the object. The following code creates a hash object called **myhash**.

```
DECLARE hash myhash ( );
```

The parentheses following **myhash** can also take in parameters in the form of tags. In the following example, there are two tags DATASET and ORDERED. The DATASET tag is followed by **mydataset**, a lookup table with unique keys. The ORDERED tag followed by ‘YES’ indicates the keys are to be sorted in ascending order when the lookup table is loaded. Other sorting options include a (ascending), d (descending) and no (keep the order as it is). The values of the tags should be included in either single or double quotes.

```
DECLARE hash myhash (dataset: 'mydataset', ordered:'YES');
```

DEFINEKEY METHOD

The DEFINEKEY method specifies the name of the hash key for the lookup table. In object oriented programming, invoking a hash object method is made by a dot notation. In the following code, **myhash** object calls DEFINEKEY method along with the name of the hash key **mykey** in single quotes. This method returns zero for successful executions, and non-zero otherwise.

```
rc = myhash.DEFINEKEY('mykey');
```

DEFINEDATA METHOD

The DEFINEDATA method specifies the data variables (the data part of the hash object) that are associated with the hash key. For any data variables that need to be included in the DATA step, their names need to be listed in quotes, separated by commas. In the following example, **myvar** from **myhash** object will be included as part of the output. Since **mykey** is not listed it will not be included when the DATA step is finished. This method returns zero for successful executions and non-zero otherwise.

```
rc = myhash.DEFINEDATA('myvar');
```

DEFINEDONE METHOD

To finish the declaration, DEFINEDONE method is called with no arguments. The DEFINEDONE method is called by **myhash** object in the following code.

```
rc = myhash.DEFINEDONE();
```

Using hash objects, we can merge two data sets sharing the same linking key in one DATA step, which accomplishes the same task as a MERGE statement. The advantage of using hash objects to merge two data sets is that no sorting is needed beforehand.

The code below defines a large data set called **largedata** that contains linking key **patient_id** and several other variables, and a small patient name lookup table called **mydataset** that contains a unique hash key **patient_id** and a data component **patient_name**. We would like to match every record from **largedata** with **mydataset** to obtain the **patient_name**. The LENGTH statement assigns the attributes of the variables listed in the DEFINEKEY and DEFINEDATA methods. SAS log will print variables uninitialized if variables are not initialized, which can be avoided by using CALL MISSING routine. The FIND method is used in searching the hash table **myhash** for each key that matches **largedata**. When rc=0 (match found) is returned by FIND method, data is outputted. The resulting data set **outdataset** will contain records both exist in **largedata** and **mydataset**.

```
data outdataset;
   length patient_id $10 patient_name $10;
   if _N_ = 1 then do;
   DECLARE hash myhash (dataset: 'mydataset', ordered:'YES');
   rc = myhash.DEFINEKEY('patient_id');
   rc = myhash.DEFINEDATA('patient_id', 'patient_name');
   rc = myhash.DEFINEDONE();
   CALL MISSING (patient_id, patient_name);
   end;
set largedata(keep=patient_id diagnosis visit_type);
   rc = myhash.FIND();
   if (rc = 0) then output;
   drop rc;
run;
```

HASH ITERATOR

The hash iterator, or hiter, is much like a pointer that iterates through the hash object from the beginning to the end and vice versa. To declare a hash iterator object, use DECLARE *hiter* followed by an assigned name, **myiter** in the sample code. The name of the hash object **myhash** is passed in single quotes to establish a connection between the hash object and its iterator.

```
DECLARE hiter myiter('myhash');
```

Four basic iterator methods are available:

- myiter.FIRST() – Retrieves the value of the first entry of the hash object.
- myiter.PREV() – Retrieves the value of previous entry of the hash object.
- myiter.NEXT() – Retrieves the value of the next entry of the hash object.
- myiter.LAST() – Retrieves the value of the last entry of the hash object.

APPLYING PERL REGULAR EXPRESSIONS AND HASH OBJECTS

In this section, we apply Perl regular expressions and hash objects in information retrieval for healthcare research use. In medical fields, doctors often document discharge summaries and clinical notes on surgery progress, radiology exams, patient history and etc. These notes are usually free text documents with substantial amount of important data for medical researches and care management improvements. By leveraging regular expressions and hash objects, we are able to skim through these unstructured texts and only focus on relevant information.

To illustrate, we conducted a research project to identify women with preeclampsia during pregnancy based on their clinical notes. In order to identify this cohort, the first step is to exclude patients that did not develop this complication, which means that we need to gather and collect English phrases that are construed as negation. The list of phrases is gerenated from a published comprehensive negation terms [3] and phrases collected from our research. Examples of negation terms include “no evidence of”, “denied” or “rule out”. These terms are then saved into a text file called “negation.list”.

Here is the complete code that searches clincal notes using different combintations of negation phrases containing preeclampsia condition:

```
data negation_concepts;
length key $4 concepts $200;
infile "negation.list" truncover;
input      @01 key $4.
           @05 negterms $100.;
concepts="/(^|\W)"||strip(negterms)||strip("(PRE(\s|\-)?ECLAMPSIA)(\W|$)/");
run;

data  cond_possible rule_out;
length key  $4  concepts  $200;
set clinical_note(in=a keep=phrase line patient_id);

 if _N_ = 1 then do;
   DECLARE hash myhash(dataset:"work.negation_concepts",ordered:'YES');
   rc = myhash.DEFINEKEY('key');
   rc = myhash.DEFINEDATA('key','concepts');
   rc = myhash.DEFINEDONE();
   CALL MISSING (key, concepts);
   DECLARE hiter myiter('myhash');
end;

rc = myiter.first();
do while (rc = 0);
    rc1=PRXPARSE(strip(concepts));
    if PRXMATCH(rc1, phrase)>0 then do;
       output rule_out;
       call PRXFREE(rc1);
       goto next_obs;
       end;
    call PRXFREE(rc1);
    rc = myiter.next();
end;

output cond_possible;
next_obs:;
run;
```

The first DATA step reads in a list of negation terms from “negation.list” and concatenate each term with the medical condition of interest. Since preeclampsia could be written differently, it is sensible to express preeclampsia using regular expressions. An example of resulting variable **concepts** is the following:

```
"/(^|\W)(NO EVIDENCE OF )(PRE(\s|\-)?ECLAMPSIA)(\W|$)/";
```

This expression can be broken down into four parts:

- The first part, (^|\W) matches either the beginning of the line or any non-word character such as a parenthesis, bracket, space or plus sign.
- The second part is the negation term, in this case NO EVIDENCE OF followed by a space.
- The third part is the medical condition, which is presented by characters PRE followed by an optional space or a dash, followed by ECLAMPSIA.
- The last part (\W|$) matches either the end of the line or a non-word character.

In the second DATA step, the hash object searches for **concepts** matches against the free text variable **phrase** in the **clinical_note** data set.

- After the first iteration (when _N_=1), a hash object **myhash** with a hash key **key** is created along with a hash iterator object **myiter** on the **negation_concepts** lookup table.
- Then **myiter** calls the iterator method FIRST, which copies the contents of the first item from **myhash** into the data variable **concepts**. While there are items to be read (rc=0), each item of the hash table is accessed one by one in the DO loop.
- The regular expression variable **concepts** is passed into PRXPARSE function in the next step.
- Next, the PRXMATCH method searches the **phrase** variable for matching string patterns. If a match is found (PRXMATCH > 0) then all variables from **clinical_note** plus data variables from DEFINEDATA are outputted into **rule_out** dataset. Then a function call to PRXFREE releases the memory space of the regular expression **rc1** and exits the DO loop.
- If no match is found (PRXMATCH = 0), occupied memory is also released. Then **myiter** object calls NEXT method to point to the next entry in **myhash** object in ascending order.
- Records with no string patterns matched are outputted to the **cond_possible** data set.

The accuracy of the data matching depend on the preciseness of the search criteria. The records in **rule_out** data set should be comprised of women without preeclampsia, but it is plausible there are missed subjects in the **cond_possible** data set due to typing mistakes in the text or missed negation terms. Furthermore, this program only takes into consideration of pre-UMLS negation phrases (negation terms that occur before the medical concept). Other types of negation, pseudo neagtion (terms that are not true negation phrases) and post-UMLS negation (terms that follow the medical concept they are negating) [4] have not taken into account. As mentioned previously, the presented code is only the first phase of our data exploration, further text analysis is required to obtain the desired cohort.

As shown in the above program, pattern matching on unstructured text is done in one simple DO loop. If string patterns were to be added or modified, we just simply change the negation list without touching the SAS code. In addition, the code can be easiliy customized. With adjustments such as adding another hash object to accommodate multiple medical conditions of interest, this program is very versatile.

CONCLUSION

Perl regular expressions and hash objects are already powerful on their own, joining them together will greatly improve text analytics. This paper is intended to show how text searching is simplified by utilizing both tools rather than giving an in depth discussion of each tool. In this paper, we demonstrated both perl regular expressions and hash objects for effective information retrieval in clinical notes. This combined method can be easily applied for information retrieval in other areas.

REFERENCES

1. SAS Perl Regular Expressions Tip Sheet
http://support.sas.com/rnd/base/datastep/perl_regexp/regexp-tip-sheet.pdf

2. SAS Hash Object Tip Sheet
http://support.sas.com/rnd/base/datastep/dot/hash-tip-sheet.pdf

3. Chapman, Wendy and others *NegEX algorithm*
http://code.google.com/p/negex

4. Chapman WW, Bridewell W, Hanbury P, Cooper GF, Buchanan BG. *A Simple Algorithm for Identifying Negated Findings and Diseases in Discharge Summaries*. Journal of Biomedical Informatics, 2001. 34: p. 301-310.

RECOMMENDED READING

1. Cody, Ron (2004), *An Introduction to Perl Regular expression in SAS 9*, Proceedings of the 2004 SAS Users Group International Conference.

2. Eberhardt, Peter (2011). The SAS® Hash Object: *It's Time to .find() Your Way Around*, Proceedings of the 2011 SAS Global Forum (SGF) Conference.

3. Dorfman, Paul, and Koen Vyverman (2006). *DATA Step Hash Objects as Programming Tools*, Proceedings of the 2006 SAS Users Group International Conference.

4. Secosky, Jason and Janis Bloom (2007). *Getting Started with the DATA Step Hash Object*. Proceedings of the 2007 SAS Global Forum (SGF) Conference.

CONTACT INFORMATION

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

Lingxiao Qi
Kaiser Permanente Southern California
Department of Research and Evaluation
100 S. Los Robles Ave, 2nd Floor
Pasadena, CA 91101
Work Phone: (626) 564-5738
Email: lingxiao.qi@kp.org

Fagen Xie
Kaiser Permanente Southern California
100 S. Los Robles Ave, 2nd Floor
Pasadena, CA 91101
Work Phone: (626) 564-3294
Email: fagen.xie@kp.org