Paper 2080-2016

Creating a Q-gram algorithm to determine the similarity of two character strings

Joe DeShon, Boehringer-Ingelheim Vetmedica

ABSTRACT

This paper shows how to program a powerful Q-gram algorithm for measuring the similarity of two character strings. Along with built-in SAS functions -- such as SOUNDEX, SPEDIS, COMPGED, and COMPLEV -- Q-gram can be a valuable tool in your arsenal of string comparators. Q-gram is a method to evaluate letter patterns in words; pairs of words with a high Q-gram score have a large number of similar letter patterns, which is a good measure of their similarity. Q-gram is especially useful when measuring the similarity of strings that are intuitively identical, but which have a different word order, such as “John Smith” and “Smith, John”.

INTRODUCTION

A wide variety of applications require that two strings be compared and a value be assigned that indicates the similarity of the strings. This is especially true as data becomes more unstructured and as computers become more powerful. People expect that the computer should be able to solve a problem such as judging the similarity of two strings as easily and as accurately as an intuitive human being can.

Of course, the expectation is that the computer should be able to solve the problem a million times a minute as databases get larger, more complicated, and less structured.

This paper will teach you how to write a powerful Q-gram algorithm that can be used to perform such string comparison tasks.

COMMON USES FOR STRING SIMILARITY

Common applications that require string similarity measurement include:

- Real-time name/address lookup
  A user types in a piece of a name or address. Even if the name is misspelled, the expectation is that the computer can find the correct record almost instantaneously.
- Database de-duping
  A database of names and address can easily become polluted with duplicate entries, the result of multiple people entering data at different times and under different circumstances. An algorithm is needed to bring together likely duplicate records so the two can be merged or one can be deleted.
- Fuzzy name-and-address matching
  A list of prospective customers with names and addresses may be matched against an existing database of customers to determine which records already exist as customers and which records should be treated as true prospects.

In each case, finding the correct solution is a combination of art and science. No two circumstances are the same. Each company and each industry has their own unique set of challenges which need to be addressed.

EXISTING SOLUTIONS IN BASE SAS®

Fortunately, Base SAS has several built-in functions that can be used to develop applications to do this type of processing. They include:

- SOUNDEX FUNCTION
  Originally developed in the early 20th century, a primary application is in genealogy and census applications. It reduces English words to a basic phonetic characterization. It can be assumed that two strings that reduce to the same (or very similar) SOUNDEX strings are likely variant spellings of

the same word -- or at least homonyms of each other. SOUNDEX doesn't actually measure similarity; it only provides a method for standardizing strings based on their likely pronunciation. For that reason it won't be further mentioned in this paper.

- SPEDIS FUNCTION
  Compares a pair of words and assigns a score based on the character transformations would be required to change one word to the other. Generally, the lower the score, the greater the similarity between the two words.
- COMPLEV FUNCTION
  Uses the Levenshtein algorithm to determine the edit distance between a pair of words. It operates much faster than SPEDIS, and is especially helpful for very long strings.
- COMPGED FUNCTION
  Also calculates the generalized edit distance between two words but is much more complicated and slower and is most useful when accuracy is most important, when there is plenty of processing power available, and when processing time is not as important.

DIFFICULTY WITH WORD ORDER

In all the above cases, word order is significant. If the words in the string are not in the same order, the algorithms may calculate an extremely high edit distance -- or they may fail to see any significant similarities at all.

This can be especially frustrating to a human who can intuitively determine that the following pairs of names and addresses are equivalent:

```
      John Smith                     Smith, John
      4th floor, Empire Building     Empire Building, 4th Floor
```

The Q-gram algorithm addresses that problem by examining the number of pairs of letters each string has in common, regardless of the order. It is surprisingly accurate in assigning similarity scores. When used in conjunction with other tools and with a good approach to data preparation, Q-gram will often find similar strings that other algorithms miss.

Q-GRAM BASICS

Q-gram (sometimes referred to as N-gram) searches for patterns in one string that also exist in a second string. These patterns may be combinations of letters or of words. Just about any number of letters or words can be used -- thus the "Q" is often replaced by an indicator of the number such as "bi-grams" or "tri-grams".

This paper uses the word "Q-gram", recognizing that there are several variations in terminology and specific approaches.

For our purpose, we will consider adjacent pairs of letters. We will exclude all spaces from the strings, and will reduce the strings to just 26 upper case alphabetic characters plus the digits "0" through "9".

Consider the following two strings:

```
  JOHN SMITH          JOHANN SMYTH
```

Our data preparation process will first remove all spaces and duplicate letters. Then the Q-gram algorithm will reduce the strings to pairs of adjacent letters, thus:

```
 JO OH HN NS SM MI IT TH
           and
 JO OH HA AN NS SM MY YT TH
```

Then it will count the number of pairs of letters in the first string that also appear in the second string.

```
  JO = Yes
  OH = Yes
  HN = No
```

```
NS = Yes
SM = Yes
MI = No
IT = No
TH = Yes
```

Of the eight pairs of letters that are in the first string, five of them can also be found in the second string.

The process will also do it the other way around, looking for strings in the second string that appear in the first string. From the two results, it will calculate a score, indicating the similarity of the two strings. The higher the score, the greater the similarity.

DATA PREPARATION

A key to the success of any string matching process includes thorough data preparation. The details of such data preparation are beyond the scope of this paper, but some general principles can be noted.

To find pairs of strings to compare, it is usually best to determine the likelihood that a match is even reasonable. For example, in an application to find duplicates within a name/address database, there is no need to compare every record to every record (a full Cartesian join). Instead, it may be reasonable to assume that the ZIP codes must at least match. Therefore, it is only necessary to match every record to every record that contains the same ZIP code. This can be accomplished through an SQL statement that joins a table to itself, similar to the following:

```
proc sql;
  create table foo_out as
  select
    a.name     as name1,
    a.address  as address1,
    a.zip      as zip1,
    b.name     as name2,
    b.address  as address2,
    b.zip      as zip2
  from
    foo_in a,
    foo_in b
  where
    a.zip eq b.zip;
```

From this query, name1/address1 and name2/address2 can be the starting point to build the strings to be compared.

Of course, the exact query will depend greatly on the size and complexity of the database and the computing power available.

Once the strings have been selected for comparison, further processing is usually appropriate. This might include the removal of punctuation and the removal of all spaces and repeated characters.

It may also be appropriate to replace misspellings or standardize some common variant spellings. It might be appropriate to replace entire words with equivalents, such as replacing all occurrences of “Jim” with “James”.

It might also be appropriate to replace some common terminology with smaller tokens. For example, if the database contains an abundance of veterinary clinics, it would be a good idea to replace every occurrence of the word “Veterinary” (along with some errant misspellings) with the word “Vet”.

All these character string replacements can be managed with functions such as the TRANWRD function in Base SAS.

SOME SIMPLE CLEANUP

For the purpose of this exercise, we will begin with two strings, which are typically the concatenation of a name, an address, and a city/state.

Use the COMPRESS function to remove all characters from the strings except alpha characters and numeric characters. This can be done with the ‘kan’ option. The ‘k’ is an indicator to keep the following characters (as opposed to suppress them); the ‘a’ means to keep all alphabetics, and the ‘n’ means to keep all numerics. This will remove all spaces and special characters like dashes, slashes, commas, etc.

Combine that function with the UPCASE function to convert all characters to upper case. This gives the following code:

```
string1 = upcase(compress(string1,,'kan'));
string2 = upcase(compress(string2,,'kan'));
```

BUILDING A Q-GRAM ALGORITHM IN BASE SAS FROM SCRATCH

Base SAS does not contain a built-in function to calculate Q-gram scores. But it is easy to construct one using other character manipulation functions in a data step.

First, initialize the score to 0:

```
score = 0;
```

The following code stub creates a loop to create substrings from the primary string:

```
do index = 1 to (length(q2string1)-1);
  /** Calculate the score here...  **/
  end;
```

Notice that the limit of the loop is one less than the length of the string. This is because there is no reason to perform the loop on the very last character; there will be no adjoining character to the right of it to pair it with.

Use the SUBSTR function to find all the two-byte strings to look for:

```
substr(string1,index,2)
```

Use the FIND function to see if that string exists in the other string:

```
find(string2,substr(string1,index,2))
```

If that string is found, add one to the score:

```
if find(string2,substr(string1,index,2)) then score + 1;
```

Put that into the loop and then repeat the process, going in the other direction:

```
score = 0;
drop index;
do index = 1 to (length(string1)-1);
  if find(string2,substr(string1,index,2)) then score + 1;
  end;
do index = 1 to (length(string2)-1);
  if find(string1,substr(string2,index,2)) then score + 1;
  end;
```

The score is then divided by two, giving an average of the two:

```
score = score / 2;
```

The above code gives a similarity score, but unless it is placed in its proper context, it’s not very helpful. Longer pairs of strings will give larger scores even if their similarity is less than that of smaller string.

The score needs to be weighted, based on the length of the strings. There are several ways of doing this, and there is no one “correct” way -- as long as the method is reasonable and consistently applied.

For the purpose of this paper, the length of the longest string minus one will be used. By doing this, we will be able to measure the number of matches compared to the total of possible matches. This gives a score between zero and one, which can be represented as a percent.

```
score_pct = score / (max(length(string1)-1,length(string2)-1));
```

Combining all the code gives us the following:

```
string1 = upcase(compress((string1),,'kan'));
string2 = upcase(compress((string2),,'kan'));
score = 0;
drop index;
do index = 1 to (length(string1)-1);
  if find(string2,substr(string1,index,2)) then score + 1;
  end;
do index = 1 to (length(string2)-1);
  if find(string1,substr(string2,index,2)) then score + 1;
  end;
score = score / 2;
score_pct = score / (max(length(string1)-1,length(string2)-1));
```

INTERPRETING THE SCORE

A score of zero means the strings have no pairs of characters in common. A score of 1 (or 100%) means that every pair of characters in one string is found in the other string.

Tests have shown that Q-gram scores give a very accurate indication of the similarity between two strings with very few false positives.

Q-gram is especially valuable when comparing strings that intuitively equal, but that have a different word order, (such as “John Smith” and “Smith, John”).

The score is intuitive because higher scores indicate the greater likelihood of similarity (as opposed to SPEDIS, COMPLEV, and COMPGED, which give “penalties” for character transformation, meaning that higher scores indicate less similarity).

The score is weighted, based on a percentage of 0% to 100% (or zero to one), which means it is equally applicable for short strings and for long strings (as opposed to SPEDIS, COMPLEV, and COMPGED, which tend to give higher scores for longer strings, regardless of similarity).

Results will vary depending on the data and the tolerance for error, but our tests have indicated that a Q-gram score greater than 0.8 (or 80%) is sufficient to indicate that a match is highly likely and needs to be investigated further.

COMPARING DIFFERENT STRING SIMILARITY ALGORITHMS

Consider two pairs of possible matches. In the first pair, the names are intuitively identical, but the word order is different.

```
JOHN SMITH
SMITH, JOHN
```

The second pair of matches are identical, except that one letter has been changed:

```
JOHN SMITH
JOHN SMYTH
```

We would expect each of these pairs to score relatively high, indicating a high likelihood of similarity regardless of the algorithm used.

If we pass both of these pairs through each algorithm, we can see significant differences in how they measure string similarities.

Algorithm	**John Smith/ Smith, John**	**John Smith/ John Smyth**	**Note**
SPEDIS	88	11	8 times difference
COMPLEV	8	1	8 times difference
COMPGED	1000	100	10 times difference
Q-gram	0.875	0.75	17% difference

Notice that each algorithm gives scores on very different scales.

The Q-gram score is intuitive -- a higher score indicates a higher degree of similarity. But with the other algorithms, a penalty is given for each character substitution so a higher score indicates a lower degree of similarity.

The reason for the difference is that SPEDIS, COMPLEV, and COMPGED are measuring “edit distance”, i.e., how dissimilar the two strings are. Q-gram, on the other hand, measures “similarity”, i.e., how many pairs of characters do the two string have in common with each other.

Since the words in the first pair are in a different order, the edit distance is great, so the scores are very high for SPEDIS, COMPLEV, and COMPGED -- indicating dissimilarity. But since the incidence of identical pairs of characters is very high for the first pair of strings, the Q-gram score is very high, indicating high similarity. In fact, Q-gram considers the first pair to be more similar than the second pair.

Of special note is the relative difference of the scores for the same algorithm for the two pairs for strings. Even though the strings are intuitive very similar to each other, reversing the order of the words gives an 8-times difference in the score for SPEDIS and COMPLEV and a 10 times difference in the scores for COMPGED.

But the difference in the scores for Q-gram is only 17%. This means that Q-gram gives a much more stable result, consistently and accurately identifying similarity between two strings.

ALGORITHM SPEED

A test was run to determine the relative speed of processing of each algorithm. Processing power today is so great, speed of an individual function is largely irrelevant when determining the appropriateness of using that function as a tool. Nevertheless, it was important to understand if the Q-gram algorithm would process at approximately the same speed as other character string similarity measuring algorithms.

A test file of 10 million rows was generated, with two 80-bytes columns of randomly-generated alpha-numeric characters. This served as a worse-case proxy for names and addresses in a database that would be measured.

String1 (80 bytes)	**String2 (80 bytes)**
LE7RG9XT9ZD0SS94RQ0PE0RC6KS5LN02YS0DH7PN3A X5KH25LM6UD2SP5NY5RY93TK2BQ1HE1YZ0ZK10	RJ9MX3JK3JV5GC54UF0MD1SD6AE7AQ64MO6OA1HL9I H5OX06QU4XK3GH0TT3YC62MG9VH1ZN9YA3PA91
GQ1RR3NP9HC5WX84EH6LL0NY6LF2KA10JD8VK9WI1N A0PD63JC8HH4BF7RY8DU26UH6PJ6BH2NQ8LF3	DA6DM8YN1HI5NO43XH9IK9ZY9ED5AT35PQ9GN1JH3H A7JI80QQ4PD5QN5CH8ET01TT9EU8ID8DV5DW6
... continue for 10 million rows ...	

Each algorithm was passed through the 10 million records, for ten passes. The only processing that took place was reading the test file and performing the function -- no output file was generated except the SAS log. The run times for each function was averaged over the ten runs.

The results:

Algorithm	**Avg run time:** ***(10 iterations of 10 million rows)***
SPEDIS	5 min 36 sec
COMLEV	3 min 53 sec
COMPGED	11 min 54 sec
Q-gram	3 min 11 sec

The results were impressive. Q-gram consistently ran faster than all the other functions.

ONLY ONE OF MANY TOOLS

Our tests have shown that Q-gram is a powerful and valuable tool for determining similarity in strings, but it is not the only tool we use. Data cleansing, standardization, and tokenization of names and addresses before passing the data through a similarity algorithm is equally important. And a score that blends Q-gram with algorithms such as SPEDIS, COMPLEV, and COMPGED is probably a good approach.

CONCLUSION

There are many different ways to implement a Q-gram algorithm. This paper presented only one. Other options include using triplets rather than pairs of characters, eliminating consecutive duplicate characters, including punctuation rather than suppressing it, and treating special conditions such as the first and last character of each string differently.

Whatever approach is taken, Q-gram is a powerful and effective tool that can be used for measuring string similarity. The lack of a built-in function for Q-gram in Base SAS should not deter a seasoned programmer from writing one from scratch. The task of writing such an algorithm can be both profitable and rewarding.

REFERENCES

"N-gram or Q-gram-based Algorithms", retrieved February 2, 2016 from https://www.melissadata.com/deduplication/n-gram-or-q-gram-based-algorithms.htm

"q-gram approximate matching optimisations", retrieved February 2, 2016 from http://stackoverflow.com/questions/1938678/q-gram-approximate-matching-optimisations

"Comparison of String Distance Algorithms" retrieved February 2, 2016 from http://www.joyofdata.de/blog/comparison-of-string-distance-algorithms/

CONTACT INFORMATION

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

Joe DeShon
joedeshon@yahoo.com