Resolving User Input Names to RDF Entities

Ernest Kirstein

February 5, 2015

Matching natural language names to RDF entities is essential to evaluating natural language questions over RDF databases. This poses several problems: names can be misspelled (e.g. "Swartseneger" for the label "Schwarzenegger"), may be reordered (e.g. "John Smith" for the label "Smith, John"), or the name may be abbreviated (e.g. "R.L. Stine" for the label "Robert Lawrence Stine"). That's not to meantion the small problem posed by people who have changed their name entirely, sometimes multiple times (e.g. "The artist formerly known as 'The artist formerly known as Prince'"). And despite all of these convolutions, a natural language system will need to recognize and match these arbitrary instances with the often-sparse naming information present in RDF data.

My approach to this problem was to find a string distance function which was robust to these changes and use that to simply find the best match name in $O(n)$ time. Since the names are narrowed down by context (using the parser), this was an acceptable solution: we don't need to iterate over every single label in the RDF set, just those which belong to the particular type of object the user is asking about.

There are a surplus of string comparison functions to choose from [3]. For my particular application, I have used a varation on the *Jaccard index*. It is robust to mispellings and reorderings, with the added benefit of being quite efficient. The Jaccard index is used in datamining for efficiently comparining long documents, but it is comparable to other more complex methods of name comparison [3] and anecdotal evidence suggests it will work well here.

Another problem is resolving multiple close names (or even exactly the same name) to a single entity. I took a simplistic approach (just asking the user) but I will also discuss other more sophisticated possibilities for further research.

1 Name Standardization and Enumeration

Before jumping into the specifics of the name comparison algorithm, there are a few trivialities to deal with. Names with abbreviations, punctuation, and names with multiple parts can all trip up comparison algorithms.

One standardization method I used was to remove punctuation and change all letters to upper case. There may be a few edge cases where "John O'neal" isn't the same as "John Oneal", but the mistake is acceptable the majority of the time. And such names are so close that they would trigger the system to prompt the user for confirmation anyways.

Names with multiple parts, like "John Jacob Jingleheimer Schmidt", need to be matched by partial variation like "Schmidt", "Mr. Schmidt", and "John Schmidt". Using a sufficieintly robust string comparison function, these variations will often still match the full, multi-part name better than other multi-part names. But that's a dubious assumption to rely upon - it's best to tokenize the name and include the different partial variations as other names linked with the entity. As part of my own system, I simply included the first and last names as variations on the name, excluding the middle name(s).

2 Jaccard Index

The Jaccard index is a measure of how similar two sets are to each other. This is useful in a whole host of applications [1], as you might imagine. In this application, using the Jaccard index on fragments of strings yields a very robust string comparrison function.

Let A and B be sets or multisets, then the Jaccard index $J(A, B)$ is defined [3, 5]:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Or if both sets are empty, $J(A, B) = 1$. For comparing strings, the sets might be of characters (e.g. 'HELLO' $\mapsto \{E, H, L, O\}$), of tokens (e.g. 'JOHN H SMITH' $\mapsto$ {'JOHN', 'H', 'SMITH'}), or in this case, *n-grams*.

These n-grams (also known as 'k-grams' or 'shingles' [5]) are all unbroken substrings of length n of a given string. So the 3-grams of the string 'anabasis' are {'ana', 'nab', 'aba', 'bas', 'asi', 'sis'}.

In this application, n-grams were constructed to include imaginary pre and post string characters (represented here as '^' and '$' respectively).

So, for instance, the 3-grams of 'cat' are then {'^^c', '^ca', 'cat', 'at$', 't$$'}. This gives significance to the begining and end of a string when using the Jaccard index with the n-grams.

For a distance function, one can use:

$$d(s_1, s_2) = 1 - J(ngrams(s_1), ngrams(s_2))$$

This doesn't produce a true metric space for the strings (since two different strings can have exactly the same n-grams), but it does satisfy the triangle inequality [1] and is always non-negative. As such, one could concievably construct an M-tree [2] to improve lookup speed. That hasn't been necessary in this research, but then again, this domain might be smaller than one might experience in practice.

2.1 Comparison with Levenshtein (Edit) Distance

Levenshtein distance (also known as 'edit distance') is a more common fuzzy string comparison algorithm than the Jaccard index. Its ubiquity might be due to a simple happenstance: the algorithm for calculating edit distance is a favorite example in algorithm design courses for demonstrating dynamic programming. But it's popularity is by no means a guarantee that it is the best choice, as I'll demonstrate.

Levenshtein distance is define as [4] the minimum number of 'edits' (additions, deletions, or swaps) that must occur before one string matches another. But these are only single-character edits, and so the algorithm doesn't handle large displacements of parts of the name with any sort of grace. This is best demonstrated by example; see figure 2.

```
Levenshtein:
14 'global kirstein investing' 'kirstein global investing'
13 'global kirstein investing' 'scherl global investing'
7 'kirstein global investing' 'scherl global investing'

Jaccard w/3-grams:
0.3125 'global kirstein investing' 'kirstein global investing'
0.594594594595 'global kirstein investing' 'scherl global investing'
0.514285714286 'kirstein global investing' 'scherl global investing'
```

Figure 1: Bad Levenstein Name Matching

Using levenshtein distance to match 'closest' strings, word inversions would be considered bulk deletions and insertions. The Jaccard index handels this better because inverting whole words still preserves the ngrams within those words even if it breaks the joining ngrams between the words.

Admitedly, the Jaccard index is less forgiving of small typos. A single character edit breaks n n-grams. So, for short single word names, levenshtein distance is probably the prefered metric.

2.2 Information Content Sensitive Jaccard Index

The Jaccard index by itself is a fairly good way to compare names. But it would be better if the algorithm also noticed things like how 'unusual' certain name patterns were. "Tom Smith" might be closer to "John Smith" than "Tom" based purely on their Jaccard index, but any reasonable person would pick "Tom Smith" and "Tom" to be the closer names because "Smith" is such a common surname.

In more technical terms, the part of the string "Smith" should recieve less 'weight' in the comparison function because it conveys less information.

Consider the more general form of the Jaccard index [1]:

$$J(\vec{x}, \vec{y}) = \frac{\sum_i \min(x_i, y_i)}{\sum_i \max(x_i, y_i)}$$

Where $\vec{x}$ and $\vec{y}$ are large dimensional vectors rather than sets. To help relate it back to the origional form, imagine that $\vec{x}$ and $\vec{y}$ are lists of counts of all the possible things that could be in the two sets (or multisets) A and B.

$$\begin{aligned} x_i &= count(e_i; A) \\ e_i &\in \{A \cup B\} \end{aligned}$$

Now if we want to consider the amount of information each little portion of the string contains, we can calculate it using the equation layed down by Shannon [6]. Namely, that the self information of an event is the log of the inverse of the probability that event occuring. In this case, the 'event' is the occurence of a certain n-gram in the string.

$$\begin{aligned} I(e_i) &= \log\left(\frac{1}{P(e_i)}\right) \\ &= -\log(P(e_i)) \end{aligned}$$

To determine this probability, we can look at the frequency of the n-gram in a large sample set of names. We need to index those names anyways, as part of the named entity recognition proces. With a large set, we can get pretty close to a true approximation of the probability of an n-gram occuring in a general population of those names:

$$P(e_i) \approx \frac{count(e_i; N) + 1}{|N| + 2}$$

Where N is the multiset containing the union of all the n-grams of all the names. Then, we can weight the vectors x and y such that we have an information sensitive comparision function: $\hat{J}(x, y)$:

$$\begin{aligned}\hat{J}(\vec{x}, \vec{y}) &= \frac{\sum_i \min(x_i I(e_i), y_i I(e_i))}{\sum_i \max(x_i I(e_i), y_i I(e_i))} \\ &= J\left(\vec{x} \cdot \vec{I}, \vec{y} \cdot \vec{I}\right) \\ \vec{I} &= \langle I(e_1), I(e_2), ...I(e_n) \rangle\end{aligned}$$

```
Without Info:
0.647058823529 'tom smith' 'john smith'
0.769230769231 'tom smith' 'tom'
1.0 'john smith' 'tom'

Sample Set:
        adam smith
        bob smith
        carl smith
        dale jones
        ernest kirstein

With Info:
0.729636835639 'tom smith' 'john smith'
0.723483108499 'tom smith' 'tom'
1.0 'john smith' 'tom'
```

Figure 2: Information Sensitive Name Matching

The above example shows this comparison in action. In the top comparision, without considering information content, the closest pair of names is "tom smith" and "john smith". But after indexing the sample names, we can calculate comparisons that do consider the information content. Then we can see that the closer names are "tom smith" and "tom".

3 Ambiguity

What should the system do when the user inputs a name which is close to the names of several entities by the chosen name comparision function?

The easiest solution would be to simply ask the user which of the possible entities they meant. But this isn't always a great solution; what if you're asking about a cornicopia of names? Sure, it might be out of scope for this particular system to resolve questions such as "Which of the actors in 'my_data_file.txt' have been in movies together?" But that's certainly within the realm of possibilities for some future work.

One approach to this problem would be to create a separate system for distinguishing the "right" name from a small(er) selection of possible candidates. The aformentioned Jaccard index (or similar distance metric) might be used to narrow down the problem space to something managable, then a much more sophisticated (yet slower) system could choose the right name from the smaller set.

Since there are a number of string comparison algorithms, one could (if time permited) implement several of them and use a combigned metric to evaluate the names for fitness. For example, a feed-forward neural net could be trained to recognize the 'right' name based on a number of factors:

- String comparison functions (Jaccard, Levenshtein, etc.)
- The self-information [6] of the input and potential match names
- The closeness of other potential match names
- The prevelance of the potential match entity in the RDF data
- The frequency of queries involving the potential match entity
- Etc.

NOTE: I implemented just such a neural net at Discovery Data but I'm not sure how much of that system I can mention in this paper. Pybrain is an awesome library; it takes about 10 lines of code to build a working example (after coding all the factors in that list).

References

[1] Flavio Chierichetti, Ravi Kumar, Sandeep Pandey, and Sergei Vassilvitskii. Finding the jacard median. `http://theory.stanford.edu/~sergei/papers/soda10-jaccard.pdf`.

[2] P. Ciaccia, M. Patella, F. Rabitti, and P. Zezula. Indexing metric spaces with m-tree. `http://www-db.deis.unibo.it/research/papers/SEBD97.pdf`.

[3] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. https://www.cs.cmu.edu/~pradeepr/papers/ijcai03.pdf, 2003.

[4] Gonzalo Navarro. A guided tour to approximate string matching. http://www.captura.uchile.cl/bitstream/handle/2250/132588/Navarro_Gonzalo_Guided_tour.pdf.

[5] Jeff M. Phillips. Data mining - jaccard similarity and shingling. http://www.cs.utah.edu/~jeffp/teaching/cs5955/L4-Jaccard+Shingle.pdf, 2013.

[6] C. E. Shannon. A mathematical theory of communication. http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf, 1948.