Project practical on scalable string similarity metrics

Algorithm: Jaro-Winkler

Kevin Dreßler

University Leipzig `mam10bpb@studserv.uni-leipzig.de`

1 Abstract

This work is a construction and evaluation of scalable filters for record linkage in Semantic Web contexts using the Jaro-Winkler distance[4], intended for usage within the LIMES[1] framework.

2 Subject

In the cutting-edge research field of semantic web one has to handle very large (more than 10 million entries) knowledge bases. It is a common practice to link entities of two or more knowledge bases to each other to discover hidden knowledge or to simply provide richer content for potential (automated) readers. Absolutely essential to real world applications like LIMES is the ability to filter data pairs before applying any fuzzy string matching scoring algorithm, because naively matching every pair of two large knowledge bases may result in trillions of string comparisons and the fuzzy matching algorithms involved in scoring pair similarity come with a considerable amount of complexity of their own.

This project practical demanded the construction of such scalable filters for the Jaro-Winkler distance with the option of the results being integrated within LIMES.

3 The Jaro-Winkler distance

The Jaro[3] distance is a string matching algorithm originally developed for name comparison in the U.S. Census. It takes into account the number of character matches m and the ratio of their transpositions t:

$$d_j = \begin{cases} 0 & \text{if } m = 0 \\ \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) & \text{otherwise} \end{cases} \tag{1}$$

Jaro-Winkler is an enhancement to Jaro to put emphasis on matching prefixes if the Jaro distance exceeds a certain "boost threshold" b_t, originally set to $0.7.$:

$$d_w = \begin{cases} d_j & \text{if } d_j < b_t \\ d_j + (\ell p(1 - d_j)) & \text{otherwise} \end{cases} \tag{2}$$

This is due to Winklers observation that typing errors most of the time occur in the middle or at the end of a word, but very rarely in the beginning.

3.1 Example

For the readers understanding, we exercise an example of Jaro-Winkler with strings $s_1 = "DEMOCRACY", s_2 = "DEMOGARPHY"$, with s_2 being intentionally misspelled.

- $|s_1| = 9, |s_2| = 10$
- $w_m = 4$
- $m = 7$
- $t = \frac{2}{2} = 1$
- $d_j = \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) = \frac{1}{3}\left(\frac{7}{9} + \frac{7}{10} + \frac{6}{7}\right) = 0,778$
- $d_w = d_j + \ell p\,(1 - d_j) = 0,867$

4 Implemented filters

The main principle behind the filters that are to be presented is that they all return a upper bound estimation $\theta_e(s_1, s_2) \geq d_w(s_1, s_2)$ for some easily retrievable properties of the input strings.
For a given threshold θ, if $\theta_e(s_1, s_2) \leq \theta$, then we can safely ignore the input (s_1, s_2).

4.1 Filtering by length

The first idea that comes to mind naturally is a length filter, as a strings length is probably its most used and best known property in all programming paradigms, and, dependent on language implementation, has a low complexity of either $O(1)$ or $O(n)$.
For a given pair of strings s_1, s_2 and their lengths $|s_1|, |s_2|$ we have to ask how we can adopt the Jaro-Winkler formula to give us an upper bound approximation of similarity.
Let s_1, s_2 be the strings we are interested in ($|s_1| \leq |s_2|$) and m be the number of matches. Because ($m \leq |s_1|$) will always be true, we can substitute m with $|s_2|$ and gain the following upper bound estimation for $d_j(s_1, s_2)$:

$$d_j = \frac{1}{3}(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}) \leq \frac{1}{3}(1 + \frac{m}{|s_2|} + 1) \leq \frac{2}{3} + \frac{m}{3|s2|} \leq \frac{2}{3} + \frac{|s1|}{3|s2|} \quad (3)$$

The application of this approximation on Winklers extension is trivial:

$$d_w = d_j + \ell \cdot p \cdot (1 - d_j) \leq \frac{2}{3} + \frac{|s1|}{3|s2|} + \ell \cdot p \cdot (\frac{1}{3} - \frac{|s1|}{3|s2|}) = \theta_e \quad (4)$$

By using this approach we can decide in $O(1)$ (at least in Java, where `String↩ .length()` is $O(1)$) if a given pairs score is greater than a given threshold, which saves us the much more expensive score computation for a big number of pairs, provided that the input strings sufficiently vary in length.

4.2 Filtering ranges by length

The above approach may be reversed to limit the number of pairs that we are going to iterate over. We can QuickSort the input lists of strings (e.g. $list1, list2$) and then construct two integer lists (e.g. $minTargetIndex, maxTargetIndex$) in such a manner that $list1.get(x)$ will only attempt to match against $list2.get(y)$ where $minTargetIndex.get(x) \leq y \leq maxTargetIndex.get(x)$ So for a given string $s_1 \in list1$, respectively its length $|s_1|$, and a threshold θ, what is the minimum length difference $\delta = |s_1| - |s_2|$ so that $\theta \geq \theta_e(s_1, s_2)$ is satisfied?
We transpose equation 4 to the following for our lower bound (s_2):

$$|s_2| = \frac{0.6 \cdot |s_1|}{3 \cdot \theta - 2 - \ell \cdot p} \quad (5)$$

And analog for our upper bound (s_1):

$$|s_1| = \frac{|s_2|(3 \cdot \theta - 2 - \ell \cdot p)}{0.6} \tag{6}$$

4.3 Filtering by character frequency

The last approach to be reflected in this report is a filter by absolute character frequency.
Let $e(s,c)$ be the character frequency function of a string s and character c. It returns the number of occurences of c in s. The number of maximum possible matches m_{max} can be expressed as

$$m_{max} = \sum\nolimits_{c \in s_1} min(e(s_1,c), e(s_2,c)) \geq m \tag{7}$$

In the Jaro distance calculation we now substitute m for m_{max}:

$$d_j = \frac{1}{3}(\frac{m_{max}}{|s_1|} + \frac{m_{max}}{|s_2|} + \frac{m_{max} - t}{m_{max}}) \leq \frac{1}{3}(\frac{m_{max}}{|s_1|} + \frac{m_{max}}{|s_2|} + 1) \tag{8}$$

5 Evaluation

5.1 Benchmarks

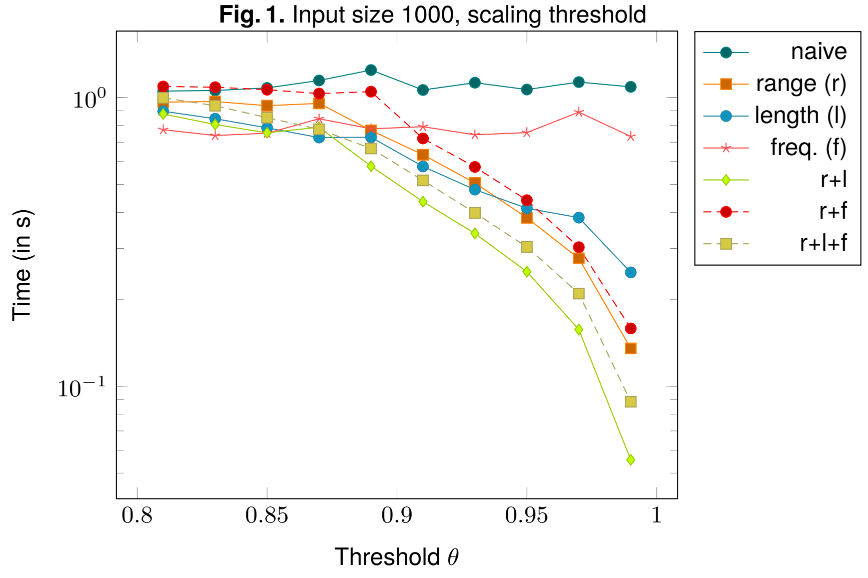
The Jaro-Winkler distance, above mentioned filters and test classes all were implemented in Java. Two kinds of charts were created, one with fixed threshold and scaling input size and one with fixed input size and scaling threshold.

As shown in Figure 1 the best performing setup is filtering both ranges and individual pairs by length. We therefore just plot this setup for higher input sizes to save benchmarking time. The remaining charts can be found at the end of this report.

5.2 Interpretation

The individual length filter is very scalable, because of its $O(1)$ performance. Every time it gets triggered it saves us $O(|s_1| \cdot |s_2|)$ time. Even better is the ranges by length filter, with a small overhead of $O(n \log n + m \log m)$, n and m being the input lists sizes, we can "chop off" a large portion of the problematic $O(n \cdot m)$. Combined they performed the best in every benchmark test that has been done. Not so well performing is the character frequency filter. Even when precomputing hashmaps to speed it up its overhead is unbearable and in benchmarks not shown in this report it took even longer than the native approach.
It has to be mentioned that the great speedup gained from combined range and length filtering only applies to thresholds over 0.8, at least in normal configurations with $p = 0.1$ and $\ell \leq 4$. However this is not a big issue, since Jaro-Winkler distance is very optimistic and distances under 0.8 usally do not make for good alignments.

Fig. 1. Input size 1000, scaling threshold



6 Summary

The speedup on the test machine ranges between 10 and 20. Implementations in more low-level oriented languages like C++ are estimated to perform even better.
The source code repository used for benchmarking purposes in Java can be found at GitHub[2].

References

1. AKSW. Limes. http://aksw.org/Projects/LIMES.html.
2. Kevin Dreßler. Semanticweb-quickjarowinkler. https://github.com/kvndrsslr/SemanticWeb-QuickJaroWinkler.
3. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. Journal of the American Statistical Association 84(406):414–420, 1989.
4. Winkler. Overview of record linkage and current research directions. http://www.census.gov/srd/papers/pdf/rrs2006-02.pdf, 2006.

Fig. 2. Input size 10000, scaling threshold

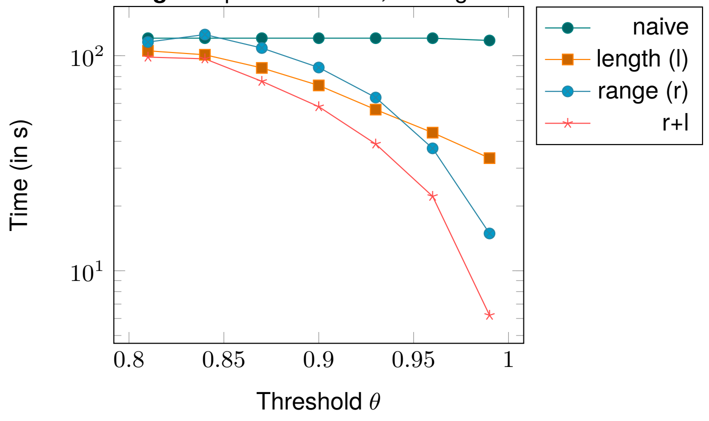


Fig. 3. Input size 100000, scaling threshold

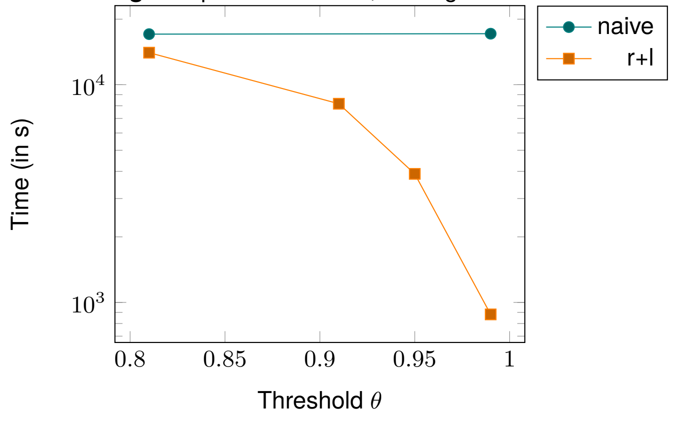
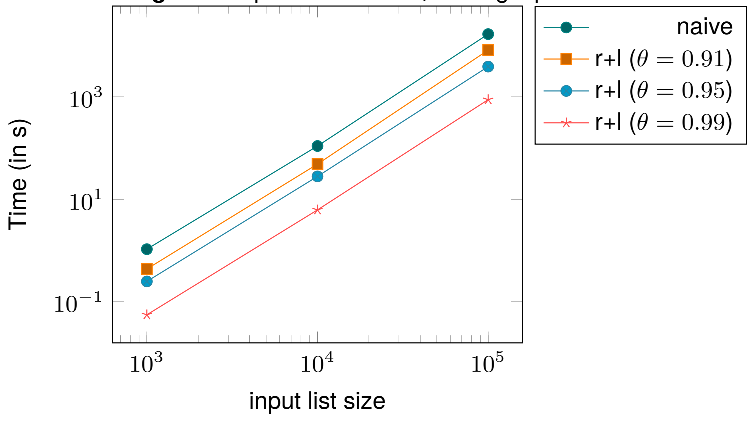


Fig. 4. Multiple thresholds θ, scaling input size


naive
r+l ($\theta = 0.91$)
r+l ($\theta = 0.95$)
r+l ($\theta = 0.99$)
Time (in s)
10^3
10^1
10^{-1}
10^3
10^4
10^5
input list size