The stringdist Package for Approximate String Matching

by Mark P.J. van der Loo

Abstract Comparing text strings in terms of distance functions is a common and fundamental task in many statistical text-processing applications. Thus far, string distance functionality has been somewhat scattered around R and its extension packages, leaving users with inconsistent interfaces and encoding handling. The **stringdist** package was designed to offer a low-level interface to several popular string distance algorithms which have been re-implemented in C for this purpose. The package offers distances based on counting q-grams, edit-based distances, and some lesser known heuristic distance functions. Based on this functionality, the package also offers inexact matching equivalents of R's native exact matching functions `match` and `%in%`.

Introduction

Approximate string matching is an important subtask of many data processing applications including statistical matching, text search, text classification, spell checking, and genomics. At the heart of approximate string matching lies the ability to quantify the similarity between two strings in terms of string metrics. String matching applications are commonly subdivided into online and offline applications where the latter refers to the situation where the string that is being searched for a match can be preprocessed to build a search index. In online string matching no such preprocessing takes place. A second subdivision can be made into approximate text search, where one is interested in the location of the match of one string in another string, and dictionary lookup, where one looks for the location of the closest match of a string in a lookup table. Here, we focus on dictionary lookup applications using online string matching algorithms.

String metrics may broadly be divided in edit-based distances, q-gram based distances (sometimes called n-grams) and heuristic distances. To determine edit-based distances one counts, possibly weighted, the number of fundamental operations necessary to turn one string into another. Operations may include substitution, deletion, or insertion of a character or transposition of characters. Distances based on q-grams are obtained by comparing the occurrence of q-character sequences between strings. Heuristic measures have no strong mathematical underpinning but have been developed as a practical tool with a particular application in mind. Finally, it is worth mentioning that these families of string metrics have recently been extended with metrics based on string kernels (Lodhi et al., 2002), which have been implemented in the **kernlab** package of Karatzoglou et al. (2004).

The base R installation offers functions for both string metric calculation and online text search. The `adist` function computes the generalized Levenshtein (1966) distance between strings while `agrep`, based on a library of Laurikari (2001), allows for online approximate text search based on the same distance metric. In fact, `agrep` is more advanced than simple approximate search since it allows for approximate matching against regular expressions. Furthermore, the functions `match` and `%in%` can be used for dictionary lookup based on exact matching while `pmatch` and `charmatch` can be used for lookup based on partial matches. A lookup function based on approximate string matching is not available in R's base installation.

The native R implementations are complemented by several several extension packages that offer offline string matching algorithms. For example, **RecordLinkage** (Borg and Sariyar, 2012), **MiscPsycho** (Doran, 2010), **cba** (Buchta and Hahsler, 2013), and **Mkmisc** (Kohl, 2013) offer interfaces to the Levenshtein distance, while **deducorrect** (van der Loo et al., 2011) implements the restricted Damerau-Levenshtein distance[1]. Although the distances implemented in these packages are all based on (an extension of) the Levenshtein distance, interfaces may vary: **RecordLinkage** takes `character` data as input and computes byte-wise distances ignoring the possibility of multibyte character encoding. Packages **MkMisc** and **deducorrect** do exactly the opposite and always compare strings character-wise. The **cba** package can compute bytewise distances between character data as well as between lists of integers, leaving a translation from (possibly multibyte character) data to integers to the user. Finally, the **textcat** package of Hornik et al. (2013), which implements q-gram based text classification methods of Cavnar and Trenkle (1994), offers a range of q-gram based distances which can be computed on a character-wise or byte-wise basis.

There are several packages that offer string-like distance functions applied to specialized objects. The **TraMineR** package (Studer et al., 2011) for example implements various distance measures on

[1]The **vwr** package (Keuleers, 2013) used to offer its own implementation of string distances but currently depends on **stringdist**.

"state sequence objects" including the Hamming (1950) distance, the Longest Common Substring distance and more. Although it may be possible for users to translate character strings representing text to a state sequence object, this obviously means using **TraMineR** for purposes for which it was not intended, which is undesirable. Similarly, the Hamming distance is implemented in at least five more packages for object types ranging from presentations of graphs (Butts, 2013), to rank vectors (Grimonprez, 2013) to phylogenic data (Schliep and Paradis, 2013).

The fact that several authors have taken the trouble to implement similar functionality shows that the ability to compute offline string distance measures is a fairly basic need across fields that deserves specialized attention. Moreover, the variety of input types and interpretations thereof make reuse and comparison of these implementations less than transparent.

The **stringdist** package presented in this paper aims to help users by offering a uniform interface to a number of well-known string distance measures where special values and (non)interpretation of character encoding are handled transparently and consistently across metrics. The package re-implements some previously available distances and adds a number of distance measures which were according to this author's best knowledge previously unavailable in R. Distances implemented by the package include edit-based distances (Hamming, generalized Levenshtein, Longest Common Substring, optimal string alignment, and generalized Damerau-Levenshtein), q-gram based distances (q-gram, Jaccard, and cosine) and the heuristic Jaro and Jaro-Winkler distances. Of these distances, at least the generalized Damerau-Levensthein distance and the Jaccard distance appear to be new in the context of character strings. Core distance functions have been implemented as a C library for speed and exposed to R. Based on this functionality, the package implements approximate matching equivalents of base R's table lookup functions `match` and `%in%`. A a convenience function that lists q-gram occurrences is included as well.

The rest of this paper is organized as follows. In the next section we give a quick overview of the package's main functionality and discuss how special values and character encodings are handled. The section after that is devoted to a concise description of the distance functions offered by **stringdist**. We will work out simple examples, and point out some properties of the various metrics. The section is aimed to serve as a reference to users of the package. We end with some conclusions and an outlook.

The stringdist package

The package offers two basic kinds of functionality: computing string comparison metrics and table lookup by approximate string matching. String distances can be computed with the function `stringdist` or `stringdistmatrix`. Both take at least two character vectors as arguments, but the former computes element-wise string distances where the shorter argument is recycled, and the latter returns the distance matrix.

```
> stringdist('foo', c('fu','bar',NA))
[1]  2  3 NA

> stringdistmatrix(c('foo','bar'), c('fu','bar',NA))
     [,1] [,2] [,3]
[1,]    2    3   NA
[2,]    3    0   NA
```

Here, the default distance is the optimal string alignment distance which will be discussed in more detail below. To specify another algorithm one simply sets the `method` option. For example, to compute the Levenshtein distance one uses the following.

```
> stringdist('foo', 'bar', method='lv')
```

String distance functions have two possible special output values. `NA` is returned whenever at least one of the input strings to compare is `NA` and `Inf` is returned when the distance between two strings is undefined according to the selected algorithm. For example, the Hamming distance is undefined when comparing strings of different length.

```
> stringdist('fu', 'foo', method='hamming')
[1] Inf
```

The functions `stringdist` and `stringdistmatrix` have very similar interfaces, allowing users to select from nine different string distance algorithms, define weights (depending on the chosen distance) and more. The `stringdistmatrix` function has extra options that allow users to parallelize calculation of the distance matrix over columns. Users may specify the number of cores to use or pass a `cluster` object as generated by `makeCluster` of the **parallel** package. For example, the command

```
> stringdistmatrix(c('foo','bar'), c('fu','bar',NA), ncores=3)
```

distributes calculation of the columns of the distance matrix over three local cores.

Approximate matching can be done with the functions `amatch` and `ain`. Function `amatch(x,table)` finds the closest match of elements of `x` in `table`. When multiple equivalent matches are found, the first match is returned. A call to `ain(x,table)` returns a `logical` vector indicating which elements of `x` were (approximately) matched in `table`. Both `amatch` and `ain` have been designed to approach the behaviour of R's native `match` and `%in%` functionality as much as possible. By default `amatch` and `ain` locate exact matches, just like `match`. This may be changed by increasing the maximum string distance between the search pattern and elements of the lookup table.

```
> amatch('fu', c('foo','bar'), maxDist=2)
[1] 1
> ain('fu', c('foo','bar'), maxDist=2)
[1] TRUE
```

The default distance function is again the optimal string alignment distance, but this can be controlled by altering the `method` option. Here, the string '`fu`' matches approximately with '`foo`' since in the default metric the difference is two operations (replace one '`u`' and add an '`o`'). By default, `NA` is matched with `NA`, as in R's native `match` function.

```
> amatch(NA, c('foo',NA))
[1] 2
```

Unlike in `match`, this may be switched off by setting `matchNA=FALSE`.

```
> amatch(NA, c('foo',NA), matchNA=FALSE)
[1] NA
```

Like in `match`, the `nomatch` option controls the output when no match is found.

```
> amatch(NA, c('foo',NA), matchNA=FALSE, nomatch=0)
[1] 0
```

Character encoding

A character encoding system defines how each character in an alphabet is represented as a byte or sequence of bytes in computer memory. Over the past decades many encoding systems have been developed, and currently several encoding standards are widely in use. It is a rather unfortunate state of affairs that encoding standards have evolved to a point where it is impossible to determine the used encoding from a text file with certainty from its contents, although command-line utilities like '`file`' (under Unix-alikes) often do a good job at guessing it.

By default, the character encoding used internally by R depends on the native encoding of the system on which it runs. This means that when one needs to read a text file that is stored in a different encoding (for example because it was produced on a different operating system), the file's encoding has to be explicitly specified. For example, in `read.table` the input encoding can be specified using the `fileEncoding` argument. Upon reading the file, R will attempt to translate input from the specified encoding to the internal encoding native to the operating system it is running on.

As a consequence, reading the same file into R will not always yield the same sequence of bytes internally on each system. Only when one is certain that the input consists solely of characters from the ASCII character set will the internal representation be guaranteed to be the same across systems. Most of R's native functions hide the character representation effectively from the user. For example, `nchar` counts the number of actual characters, not the number of bytes used to represent it (although it has an option to count the number of bytes as well).

Like R's native functions, all functions of the **stringdist** package are designed to yield the same result regardless of the internal encoding used. Like in the `adist` function, this is done by converting strings first to `utf8`, and then to an `integer` representation. The integer representation of strings are passed to the underlying C-routines. At the moment, this double conversion is the only guaranteed way to handle character strings independent of internal encoding. The main reason is that R depends on external libraries for character re-encoding, and those libraries are different across operating systems on which R is supported.

The re-encoding causes an overhead of up to a factor of three or four, when computing distances between character strings consisting of 5-25 characters. The overhead is almost completely due to the conversion to integer and its relative importance decreases with string length. If one is certain that the input strings are restricted to the ASCII character set or when accuracy or cross-platform

independence are of less importance, one can pass `useBytes=TRUE` to avoid re-encoding. In that case the distance between the underlying byte sequences are returned. This option mimics the option that is available in some of R's native functions, including `adist`. Below is an example demonstrating the difference.

```
> stringdist('Motorhead', 'Motörhead')
[1] 1
> stringdist('Motorhead', enc2utf8('Motörhead'), useBytes=TRUE)
[1] 2
```

Here, the default string distance algorithm is the optimal string alignment distance, which counts the number of insertions, deletions, substitutions and transpositions necessary to turn 'Motörhead' into 'Motorhead'. In the first case the letter 'ö' is recognized as a single character, so the distance corresponds to a single substitution. In the second case a byte-wise comparison is used while making sure that 'Motöread' is stored in `utf8` encoding. In this case, the `utf8` 'o' is stored as a single byte and 'ö' as two bytes, and here the distance is determined by deleting one byte and substituting another.

It should be mentioned that there are a number of characters that have multiple `utf8` representations (this is called 'unicode equivalence'). For example, the 'ö' can be represented by a two-byte unicode character as in the above example or by a single-byte 'o', followed by a two-byte (otherwise invisible) modifying character that specifies the umlaut. If one compares the two versions of 'ö' with either `stringdist` or R's native `adist` , the result will be nonzero regardless whether one compares byte-wise or not. The solution is to normalize unicode-encoded strings to either representation before any comparison is made. At the moment, no tools seem to be available from within R but open source commandline tools like `uconv` (Utterström and Arrouye) can be used to normalize `utf8`-encoded text prior to reading into R.

String distance functions

A *string* is a finite concatenation of symbols (characters) from a finite alphabet. We will follow the usual notation and denote a finite alphabet with Σ where the number of elements in Σ is denoted $|\Sigma|$. The q-fold Cartesian product $\Sigma \times \cdots \times \Sigma$ is denoted Σ^q, and the set of all finite strings that can be composed from characters in Σ is denoted Σ^*. The empty string, denoted ε, is also a member of this set. We will denote a general string with s, t, or u and the length of the string, measured as the number of characters, with $|s|$. For example: take for Σ the 26-member lower-case Latin alphabet, and $s =$ 'foo'. We have $|s| = 3$, $s \in \Sigma^3$ and $s \in \Sigma^*$. Individual characters of a string are indicated with a subscript so in the previous example we have $s_1 =$ 'f' and $s_2 = s_3 =$ 'o'. A subsequence is indicated with subscript $m : n$, so in the example $s_{1:2} =$ 'fo'. We also use the convention that $s_{m:n} = \varepsilon$ whenever $n < m$.

Formally, for a function d to be a *distance function* on Σ^* it has to obey the following properties.

$$\begin{aligned}
&\text{nonnegativity} && d(s,t) \geq 0 && \text{(1a)}\\
&\text{identity} && d(s,t) = 0 \text{ only when } s = t && \text{(1b)}\\
&\text{symmetry} && d(s,t) = d(t,s) && \text{(1c)}\\
&\text{triangle inequality} && d(s,u) \leq d(s,t) + d(t,u), && \text{(1d)}
\end{aligned}$$

with s, t, and u strings. However, as will be pointed out below, many string metrics do not have all these properties. For example, distances based on q-grams do not necessarily obey the identity property and weighted versions of the Levenshtein distance are not necessarily symmetric.

Distances based on edit operations

Distances based on edit operations count the number of basic operations necessary to turn one string into another. Edit-like distances can be categorized based on what operations are allowed. The distances discussed below allow one or more of the following operations.

- Substitution of a character, as in 'foo' → 'boo'.
- Deletion of a character, as in 'foo' → 'oo'.
- Insertion of a character, as in 'foo' → 'floo'.
- Transposition of two adjacent characters, as in 'foo' → 'ofo'.

For distances allowing more than a single operation it may be meaningful to assign weights to the different operations, for example to make a transposition contribute less to the distance than character

substitution. Edit-based distances for which such weights can be defined are usually refered to as *generalized* distances (Boytsov, 2011).

The simplest edit distance is the *Hamming distance* (Hamming, 1950), which allows only character substitutions and is therefore only defined for strings of equal length. It is however common to define the Hamming distance to be infinite for strings of different length (Navarro, 2001), so formally we have

$$d_{\text{hamming}}(s,t) = \sum_{i=1}^{|s|} [1 - \delta(s_i, t_i)] \text{ if } |s| = |t| \text{ and } \infty \text{ otherwise.} \tag{2}$$

Here, $\delta(s_i, t_j) = 1$ if $s_i = t_i$ and 0 otherwise. The Hamming distance obeys all the properties stated in Eq. (1). When s and t are of equal length, the maximum value is $|s|$. With the **stringdist** package, the Hamming distance can be computed as follows.

```
> stringdist(c('foo','fu'), 'bar', method='hamming')
[1]   3 Inf
```

The *Longest Common Substring* distance d_{lcs} (Needleman and Wunsch, 1970) counts the number of deletions and insertions necessary to transform one string into another. It can be recursively defined as

$$d_{\text{lcs}}(s,t) = \begin{cases} 0 \text{ if } s = t = \varepsilon, \\ d_{\text{lcs}}(s_{1:|s|-1}, t_{1:|t|-1}) \text{ if } s_{|s|} = t_{|t|}, \\ 1 + \min\{d_{\text{lcs}}(s_{1:|s|-1}, t), d_{\text{lcs}}(s, t_{1:|t|-1})\} \text{ otherwise.} \end{cases} \tag{3}$$

The longest common substring distance obeys all properties of Eq. (1). It varies between 0 and $|s| + |t|$ where the maximum is achieved when s and t have no characters in common. With the **stringdist** package it can be computed by setting `method='lcs'`.

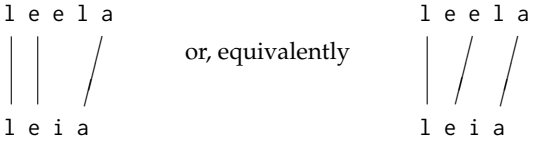
```
> stringdist('leia', 'leela', method='lcs')
[1] 3
```

Here, the distance equals 3 since it takes two deletions and one insertion to turn '`leela`' into '`leia`':

$$\text{leela} \xrightarrow{\text{del. e}} \text{lela} \xrightarrow{\text{del. l}} \text{lea} \xrightarrow{\text{ins. i}} \text{leia}.$$

The above example also shows that in general there is no uniqe shortest path between two strings: one could for example reverse the order of the first two deletions to obtain a second path with total weight 3.

As suggested by its name, the lcs-distance has a second interpretation. With the *longest common substring* we mean the longest sequence formed by pairing characters from s and t while keeping their order intact. The lcs-distance is then the number of unpaired characters over both strings. In the above example this can be visualized as follows.



In both cases, the characters '`e`','`l`' and '`i`' remain unpaired, independent of whether we start pairing from the beginning (left case) or the end of the string (right case), yielding a distance value of three.

The *generalized Levenshtein distance* d_{lv} is computed by counting the weighted number of insertions, deletions and substitutions necessary to turn one string into another. Like the lcs distance it permits a recursive definition.

$$d_{\text{lv}}(s,t) = \begin{cases} 0 \text{ if } s = t = \varepsilon \\ \quad \min\{ \\ \quad\quad d_{\text{lv}}(s, t_{1:|t|-1}) + w_1, \\ \quad\quad d_{\text{lv}}(s_{1:|s|-1}, t) + w_2, \\ \quad\quad d_{\text{lv}}(s_{1:|s|-1}, t_{1:|t|-1}) + [1 - \delta(s_{|s|}, t_{|t|})] w_3 \\ \quad \} \text{ otherwise.} \end{cases} \tag{4}$$

Here, w_1, w_2 and w_3 are the nonnegative penalties for deletion, insertion, and substitution when turning t into s. The generalized Levenshtein distance is also implemented by R's native `adist` function, and with our package it can be computed as follows.

```
> stringdist('leela', 'leia', method='lv')
[1] 2
```

The extra flexibility with respect to the lcs (and hamming) distance yields a smaller distance value since we need but a single deletion and a single substitution:

$$\text{leela} \xrightarrow[+1]{\text{del. e}} \text{lela} \xrightarrow[+1]{\text{sub. l}\to\text{i}} \text{leia},$$

where we denote the penalty for each operation below the arrow.

The generalized Levenshtein distance obeys the properties of Eq. (1) except when $w_1 \neq w_2$, in which case the symmetry property is lost. However, it remains symmetric under simultaneous reversal of s and t and w_1 and w_2 since the number of deletions necessary going from t to s is equal to the number of insertions necessary going from s to t. This is illustrated by the following example.

```
> stringdist('leia', 'leela', method='lv', weight=c(1,0.1,1))
[1] 2
> stringdist('leia', 'leela', method='lv', weight=c(0.1,1,1))
[1] 1.1
> stringdist('leela', 'leia', method='lv', weight=c(1,0.1,1))
[1] 1.1
```

In the first line we get a distance of two since no insertions (of weight 0.1) are involved when going from 'leela' to 'leia', as shown in our previous example. The second and third example may schematically be represented as follows.

$$\text{leela} \xrightarrow[+0.1]{\text{del. e}} \text{lela} \xrightarrow[+1]{\text{sub. l}} \text{leia}$$
$$\text{leia} \xrightarrow[+1]{\text{sub. i}} \text{lela} \xrightarrow[+0.1]{\text{ins. e}} \text{leela}.$$

In other words, reversing w_1 and w_2 is the same as reversing arguments s and t.

The *optimal string alignment* distance d_{osa} is a straightforward extension of the Levenshtein distance that allows for transpositions of adjacent characters:

$$d_{\text{osa}}(s,t) = \begin{cases} 0 \text{ if } s = t = \varepsilon \\ \quad \min\{ \\ \quad\quad d_{\text{osa}}(s, t_{1:|t|-1}) + w_1, \\ \quad\quad d_{\text{osa}}(s_{1:|s|-1}, t) + w_2, \\ \quad\quad d_{\text{osa}}(s_{1:|s|-1}, t_{1:|t|-1}) + [1 - \delta(s_{|s|}, t_{|t|})]w_3, \\ \quad\quad d_{\text{osa}}(s_{1:|s|-2}, t_{1:|t|-2}) + w_4 \text{ if } s_{|s|} = t_{|t|-1}, s_{|s|-1} = t_{|t|} \\ \quad \} \text{ otherwise}, \end{cases} \tag{5}$$

where w_4 is the penalty for a transposition and w_1, w_2 and w_3 were defined under Eq. (4). The optimal string alignment distance is the default distance function for `stringdist`, `stringdistmatrix`, `amatch` and `ain`. Unlike the Hamming, lcs, and Levenshtein distances, the optimal string alignment distance does not obey the triangle inequality. This is demonstrated by the following example, taken from Boytsov (2011).

```
> stringdist('ba','ab') + stringdist('ab','acb')
[1] 2
> stringdist('ba','acb')
[1] 3
```

The reason is that the optimal string alignment distance edits each substring maximally once while recursing through the strings. The above distances should be interpreted as

$$\text{ba} \xrightarrow[+1]{\text{swap b,a}} \text{ab} + \text{ab} \xrightarrow[+1]{\text{ins. c}} \text{acb}$$
$$\text{ba} \xrightarrow[+1]{\text{del. b}} \text{a} \xrightarrow[+1]{\text{ins. c}} \text{ac} \xrightarrow[+1]{\text{ins. b}} \text{acb}.$$

In the last case, the shortcut that can be taken by swapping 'b' and 'a' and then inserting 'c' would force the same substring to be edited twice. Because of this restriction, the optimal string alignment distance is also referred to as the *restricted Damerau-Levenshtein distance* although on the web, it seems fairly often to be mistaken for the actual *Damerau-Levenshtein distance*. The latter distance metric does allow for multiple edits on the same substring and is a real metric in the sense of Eq. (1).

A recursive definition for the full Damerau-Levenshtein distance was first given by Lowrance and Wagner (1975). Their definition replaces the simple swap in the last line of Eq. (5) with a minimization over possible transpositions between the current character and all untreated characters, where the cost

of a transposition increases with the distance between transposed characters. In the **stringdist** package a weighted version of the full Damerau-Levenshtein distance is implemented which is defined as follows.

$$d_{\mathsf{dl}}(s,t) = \begin{cases} 0 \text{ if } s = t = \varepsilon \\ \min\{ \\ \quad d_{\mathsf{dl}}(s, t_{1:|t|-1}) + w_1, \\ \quad d_{\mathsf{dl}}(s_{1:|s|-1}, t) + w_2, \\ \quad d_{\mathsf{dl}}(s_{1:|s|-1}, t_{1:|t|-1}) + [1 - \delta(s_{|s|}, t_{|t|})] w_3, \\ \quad \min_{(i,j)\in\Lambda} d_{\mathsf{dl}}(s_{1:i-1}, t_{1:j-1}) + [(|s| - i) + (|t| - j) - 1] w_4 \\ \} \text{ otherwise,} \end{cases} \tag{6}$$

where the minimization in the last line is over

$$\Lambda = \left\{ (i,j) \in \{1,\ldots,|s|\} \times \{1,\ldots,|t|\} \,:\, s_{|s|} = t_j,\, s_i = t_{|t|} \right\}.$$

If $w_1 = w_2 = w_3 = w_4 = 1$ the original Damerau-Levenshtein distance is obtained. Although it is a trivial generalization of the original distance described by Lowrance and Wagner (1975), this implementation of a generalized Damerau-Levenshtein distance appears to be new in literature.

For the **stringdist** package, the C code for this particular distance is based on a routine developed by Logan (2013). The original code has been adapted to reduce the number of memory allocation calls and to include the weights. The Damerau-Levenshtein distance can be computed with the `method='dl'` directive.

```
> stringdist('ba', 'acb', method='dl')
[1] 2
```

Here, the distance equals two, indicating that the path

$$\text{ba} \xrightarrow{\text{swap b,a}} \text{ab} \xrightarrow{\text{ins. c}} \text{acb},$$

is indeed included in the minimization defining the distance.

The maximum distance between two strings s and t, as measured by either the Levenshtein, optimal string alignment, or Damerau-Levenshtein distance is $\max\{|s|,|t|\}$. However, as the number of possible edit operations increases, the possible number of paths between two strings increases, allowing for possibly smaller distances between strings. Therefore, relations between the distance measures described above can be summarized as follows.

$$\left.\begin{array}{r} \infty \geq |s| \geq d_{\mathsf{hamming}}(s,t) \\ |s| + |t| \geq d_{\mathsf{lcs}}(s,t) \\ \max\{|s|,|t|\} \end{array}\right\} \geq d_{\mathsf{lv}}(s,t) \geq d_{\mathsf{osa}}(s,t) \geq d_{\mathsf{dl}}(s,t) \geq 0. \tag{7}$$

Since the Hamming and lcs distance have no basic edits in common, there is no order relation between their values. The upper limit $|s|$ on the Hamming distance only holds when $|s| = |t|$.

All edit-based distances except the Hamming distance have been implemented using the well-known dynamic programming technique that runs in $\mathcal{O}(|s||t|)$ time. For the Hamming distance, both the time and memory consumption are $\mathcal{O}(|s|)$. For the other edit-based distances the memory consumption is $\mathcal{O}(|s||t|)$, where for the Damerau-Levenshtein some extra memory is used (growing with the number of unique characters in s and t). Finally, we refer the reader to the papers of Boytsov (2011) and Navarro (2001) for a thorough review of edit-based distances in text search or dictionary lookup settings respectively.

Distances based on q-grams

A q-gram is a string consisting of q consecutive characters. The q-grams associated with a string s are obtained by sliding a window of q characters wide over s and registering the occurring q-grams. For example, the digrams associated with '`foo`' are '`fo`' and '`oo`'. Obviously, this procedure fails when $q > |s|$ or $q = 0$. For this reason we define the following edge cases for all distances $d(s,t;q)$ of the **stringdist** package that are based on comparing q-gram occurrence:

$$d(s,t;q) = \infty, \text{ when } q > \min\{|s|,|t|\} \tag{8a}$$
$$d(s,t;0) = \infty, \text{ when } |s| + |t| > 0 \tag{8b}$$
$$d(\varepsilon,\varepsilon;0) = 0. \tag{8c}$$

A simple distance metric between two strings is obtained by listing unique q-grams in two strings and compare which ones they have in common. Indeed, if we write $\mathcal{Q}(s;q)$ to indicate the unique set of q-grams occurring in s, the *Jaccard distance* is written as

$$d_{\text{jaccard}}(s,t;q) = 1 - \frac{|\mathcal{Q}(s;q) \cap \mathcal{Q}(t;q)|}{|\mathcal{Q}(s;q) \cup \mathcal{Q}(t;q)|}, \tag{9}$$

where the vertical bars ($|\cdot|$) indicate set cardinality. The Jaccard distance varies from 0 to 1 where 0 corresponds to full overlap, *i.e.* $\mathcal{Q}(s;q) = \mathcal{Q}(t;q)$, and 1 to no overlap, *i.e.* $\mathcal{Q}(s,q) \cap \mathcal{Q}(t,q) = \varnothing$. As an example, consider result of the following Jaccard distance calculation with the **stringdist** package.

```
> stringdist('leia', 'leela', method='jaccard', q=2)
[1] 0.8333333
```

It is easily checked that $\mathcal{Q}(\text{'leia'};2) = \{\text{'le'},\text{'ei'},\text{'ia'}\}$ and $\mathcal{Q}(\text{'leela'};2) = \{\text{'le'},\text{'ee'},\text{'el'},\text{'la'}\}$, so the distance is computed as $1 - \frac{1}{6} \approx 0.83$.

The *q-gram distance* is obtained by tabulating the q-grams occurring in the two strings and counting the number of q-grams that are not shared between the two. This may formally be denoted as follows.

$$d_{\text{qgram}}(s,t;q) = \|\boldsymbol{v}(s;q) - \boldsymbol{v}(t;q)\|_1 = \sum_{i=1}^{|\Sigma|^q} |v_i(s;q) - v_i(t;q)|. \tag{10}$$

Here, $\boldsymbol{v}(s;q)$ is a nonnegative integer vector of dimension $|\Sigma|^q$ whose coefficients represent the number of occurrences of every possible q-gram in s. Eq. (10) defines the q-gram distance between two strings s and t as the L_1 distance between $\boldsymbol{v}(s;q)$ and $\boldsymbol{v}(t;q)$. Observe that one only needs to store and count the actually occurring q-grams to evaluate the above sum.

With the **stringdist** package, q-gram distances are computed as follows.

```
> stringdist('leia', 'leela', method='qgram', q=1)
[1] 3
> stringdist('leia', 'leela', method='qgram', q=2)
[1] 5
> stringdist('leia', 'leela', method='qgram', q=5)
[1] Inf
```

The 1-gram distance between 'leia' and 'leela' equals 3: counting the 1-grams (individual characters) of the two strings shows that the 'i', of 'leia' and the second 'e' and 'l' of 'leela' are unmatched. The reader may verify that the 2-gram distance between 'leia' and 'leela' equals 5. In the third example, `stringdist` returns `Inf` since the since one of the compared strings has less than 5 characters.

The maximum number of different q-grams in a string s is $|s| - q - 1$, therefore $|s| + |t| - 2q - 2$ is an upper bound on the q-gram distance, occurring when s and t have no q-grams in common. See Boytsov (2011) and references therein for bounds on d_{qgram} in terms of edit-based distances.

Now that we have defined the q-gram distance in terms of vectors, any distance function on (integer) vector spaces can in principle be applied. The **stringdist** package also includes the *cosine distance*, which is defined as

$$d_{\cos}(s,t;q) = 1 - \frac{\boldsymbol{v}(s;q) \cdot \boldsymbol{v}(t;q)}{\|\boldsymbol{v}(s;q)\|_2 \|\boldsymbol{v}(t;q)\|_2}, \tag{11}$$

where $\|\cdot\|_2$ indicates the standard Euclidean norm. The cosine distance equals 0 when $s = t$ and 1 when s and t have no q-grams in common. It should be interpreted as a measure of the angle between $\boldsymbol{v}(s;q)$ and $\boldsymbol{v}(t;q)$ since the second term in Eq. (11) represents the cosine of the angle between the two vectors. It can be computed as follows.

```
> stringdist('leia', 'leela', method='cosine', q=1)
[1] 0.1666667
```

Indeed, defining $\Sigma = \{\text{'a'},\text{'e'},\text{'i'},\text{'l'}\}$, we have $\boldsymbol{v}(\text{'leia'};1) = (1,1,1,1)$ and $\boldsymbol{v}(\text{'leela'};1) = (1,2,0,2)$ giving a distance of $1 - \frac{5}{2\cdot 3} \approx 0.17$.

The three q-gram based distances mentioned above are nonnegative and symmetric. The Jaccard and q-gram distance can be written as a distance on a vector space and obey the triangle inequality as well. The cosine distance does not satisfy the triangle inequality. None of the q-gram based distances satisfy the identity condition because both $Q(s;q)$ and $\boldsymbol{v}(s;q)$ are many-to-one functions. As an example observe that $Q(\text{'ab'};1) = Q(\text{'ba'};1)$ and $\boldsymbol{v}(\text{'ab'};1) = \boldsymbol{v}(\text{'ba'};1)$ so $d_{\text{jaccard}}(\text{'ab'},\text{'ba'};1) = d_{\text{qgram}}(\text{'ab'},\text{'ba'};1) = d_{\cos}(\text{'ab'},\text{'ba'};1) = 0$. In other words, a q-gram based distance of zero does not guarantee that $s = t$. For a more general account of invariance properties of the $\boldsymbol{v}(s;q)$ the reader is referred to Ukkonen (1992).

To allow the user to define their own q-gram based metrics, the package includes the function `qgrams`. This function takes an arbitrary number of (named) `character` vectors as input, and returns an array of labeled q-gram counts. Here's an example with three character vectors.

```
> qgrams(
+   x = c('foo','bar','bar'),
+   y = c('fu','bar'),
+   z = c('foobar'),
+   q = 2 )
  fo oo fu ob ba ar
x  1  1  0  0  2  2
y  0  0  1  0  1  1
z  1  1  0  1  1  1
```

At the moment, q-gram counting is implemented by storing only the encountered q-grams, rather than representing $v(s;q)$ completely. This avoids the need for $\mathcal{O}(|\Sigma|^q)$ storage. Encountered q-grams are stored in a binary tree structure yielding $\mathcal{O}(|Q(s;q)|)$ memory and $\mathcal{O}[(|s|-q-1)\log|Q(s;q)|]$ time consumption.

Heuristic distance measures

The *Jaro distance* was originally developed at the U.S. Bureau of the Census for the purpose of linking records based on inaccurate text fields. Its first public description appeared in a user manual (Jaro, 1978) which might explain why it is not very wide-spread in computer science literature. It has however been successfully applied to statistical matching problems concerning fairly short strings; typically name and address data [see *e.g.* Jaro (1989)].

The reasoning behind the Jaro distance is that character mismatches and transpositions are caused by typing-errors but matches between remote characters are unlikely to be caused by a typing error. It therefore measures the number of matching characters between two strings that are not too many positions apart and adds a penalty for matching characters that are transposed. It is given by

$$d_{\text{jaro}}(s,t) = \begin{cases} 0 \text{ when } s = t = \varepsilon \\ 1 \text{ when } m = 0 \text{ and } |s| + |t| > 0 \\ 1 - \frac{1}{3}\left(w_1 \frac{m}{|s|} + w_2 \frac{m}{|t|} + w_3 \frac{m-T}{m}\right) \text{ otherwise.} \end{cases} \tag{12}$$

Here, the w_i are adjustable weights but in most publications they are chosen equal to 1. Furthermore, m is the number of characters that can be matched between s and t. Supposing that $s_i = t_j$ they are considered a match only when

$$|i-j| < \left\lfloor \frac{\max\{|s|,|t|\}}{2} \right\rfloor,$$

and every character in s can be matched only once with a character in t. Finally, if s' and t' are substrings of s and t obtained by removing the nonmatching characters, then T is the number of transpositions necessary to turn s' into t'. Here, nonadjacent transpositions are allowed.

With `stringdist`, the Jaro-distance can be computed as follows.

```
> stringdist('leia', 'leela', method='jw')
[1] 0.2166667
```

Here, the number of matching characters equals three, and no transpositions are necessary yielding a distance of $1 - \frac{1}{3}(\frac{3}{4} + \frac{3}{5} + 1) = \frac{13}{60} \approx 0.22$. When $w_1 = w_2 = w_3 = 1$, the Jaro distance ranges between 0 and 1, where 0 corresponds to $s = t$ and 1 indicates a complete dissimilarity with $m = T = 0$.

Winkler (1990) extended the Jaro distance by incorporating an extra penalty for character mismatches in the first four characters. The *Jaro-Winkler distance* is given by

$$d_{\text{jw}}(s,t) = d_{\text{jaro}}(s,t)[1 - p\ell(s,t)], \tag{13}$$

where $\ell(s,t)$ is the length of the longest common prefix, up to a maximum of four characters and p is a user-defined weight. We demand that $p \in [0, \frac{1}{4}]$ to make sure that $0 \leq d_{\text{jw}}(s,t) \leq 1$. The factor p determines how strongly differences between the first four characters of both strings determine the total distance. If $p = 0$, the Jaro-Winkler distance reduces to the Jaro distance and all characters contribute equally to the distance function. If $p = \frac{1}{4}$, the Jaro-Winkler distance is equal to zero, even if only the first four characters differ. The reasoning is that apparently, people are less apt to make mistakes in the first four characters or perhaps they are more easily noted, so differences in the first four characters point to a larger probability of two strings being actually different. Winkler (1990) and Cohen et al. (2003) use a value of $p = 0.1$ and report better results in a statistical matching benchmark

than with $p = 0$. The default value of p for the `stringdist` function with `method='jw'` is 0 so by altering it, the Jaro-Winkler distance is obtained.

```
> stringdist('leia', 'leela', method='jw', p=0.1)
[1] 0.1733333
```

Here, we have $\ell(\text{'leia'}, \text{'leela'}) = 2$ so the Jaro-Winkler distance is computed as $\frac{13}{60}(1 - \frac{2}{10}) \approx 0.17$.

It is easy to see from Eqs. (12) and (13) that conditional on $w_1 = w_2 = w_3 = 1$, the Jaro and Jaro-Winkler distance are nonnegative, symmetric and obey the identity property. However, the triangle inequality is not satisfied by these distances. As an example consider $s =$ 'ab', $t =$ 'cb' and $u =$ 'cd'. Since s and u have no characters in common we have $d_{\text{jaro}}(s, u) = 1$, while $d_{\text{jaro}}(s, t) = d_{\text{jaro}}(t, u) = \frac{1}{3}$ so in this case $d_{\text{jaro}}(s, u)$ is larger than $d_{\text{jaro}}(s, t) + d_{\text{jaro}}(t, u)$. It is not difficult to verify that the Jaro-Winkler distance fails the triangle inequality for the same example, for any $p \in [0, \frac{1}{4}]$.

The C-implementation of the Jaro and Jaro-Winkler distance take $\mathcal{O}(|s||t|)$ time and $\mathcal{O}(\max\{|s|, |t|\})$ memory.

What metric to use?

In the end the choice depends on the application, but there are some general considerations. The choice between an edit-based or heuristic metric on one hand or a q-gram based distance on the other, is to an extend prescribed by string length. Contrary to edit-based or heuristic metrics, q-gram based metrics can easily be computed beween very long text strings since the number of q-grams encountered in natural language (for say, $q \geq 3$) is usually much less than the q-grams allowed by the alphabet. The choice of edit-based distance depends mostly on the needed accurracy. For example, in a dictionary lookup where differences between matched and dictionary items are small, an edit distance that allows for more types of edit operations (like the optimal string alignment or Damerau-Levenshtein distance) may give better results. The heuristic Jaro- and Jaro-Winkler distances were designed with human-typed, relatively short strings in mind, so their area of application is clear.

Summary and conclusions

The **stringdist** package offers, for the first time in R, a number of popular string distance functions through a consistent interface while transparently handling or ignoring the underlying character encoding scheme. The package offers interfaces to C-based string distance algorithms that either recycle elements or return the full distance matrix. The same algorithms are used in approximate string matching versions of R's native `match` and `%in%` functions: `amatch` and `ain` respectively.

In this paper we have given a concise description of the available distance functions in terms of their mathematical definitions and showed how to compute them. The algorithmic complexity of the current implementation in terms of computational time and memory consumption was mentioned as well.

In the future, we expect to make the C-library available for export to other languages and to reduce the memory consumption for some of the algorithms.

Acknowledgements

The author is grateful to Dr. Rob van Harrevelt for carefully reading the manuscript.

Bibliography

A. Borg and M. Sariyar. *RecordLinkage: Record Linkage in R*, 2012. URL http://CRAN.R-project.org/package=RecordLinkage. R package version 0.4-1. [p111]

L. Boytsov. Indexing methods for approximate dictionary searching: comparative analyses. *ACM Journal of Experimental Algorithmics*, 16:1–86, 2011. [p115, 116, 117, 118]

C. Buchta and M. Hahsler. *cba: Clustering for Business Analytics*, 2013. URL http://CRAN.R-project.org/package=cba. R package version 0.2.12. [p111]

C. T. Butts. *sna: Tools for Social Network Analysis*, 2013. URL http://CRAN.R-project.org/package=sna. R package version 2.3-1. [p112]

W. Cavnar and J. Trenkle. N-gram-based text categorization. In *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, 1994. [p111]

W. Cohen, P. Ravikumar, and F. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of the IJCAJ-03 workshop on Information Integration on the Web*, pages 73–78, 2003. URL http://www.isi.edu/integration/workshops/ijcai03/proceedings.htm. [p119]

H. C. Doran. *MiscPsycho: Miscalleaneous Psychometric Analyses*, 2010. URL http://CRAN.R-project.org/package=MiscPsycho. R package version 1.6. [p111]

Q. Grimonprez. *Rankcluster: Model-based clustering for multivariate partial ranking data*, 2013. URL http://CRAN.R-project.org/package=Rankcluster. R package version 0.90.3. [p112]

R. Hamming. Error detecting and error correcting codes. *The Bell system technical journal*, 29:147–160, 1950. [p112, 115]

K. Hornik, P. Mair, J. Rauch, W. Geiger, C. Buchta, and I. Feinerer. The textcat package for n-gram based text categorization in R. *Journal of Statistical Software*, 52(6):1–17, 2 2013. ISSN 1548-7660. URL http://www.jstatsoft.org/v52/i06. [p111]

M. Jaro. *UNIMATCH: A record linkage system: User manual*. United States bureau of the census, 1978. pp. 103-108. [p119]

M. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida. *Journal of the American Statistical Association*, 84:414–420, 1989. [p119]

A. Karatzoglou, A. Smola, K. Hornik, and A. Zeileis. kernlab – an S4 package for kernel methods in R. *Journal of Statistical Software*, 11(9):1–20, 2004. URL http://www.jstatsoft.org/v11/i09/. [p111]

E. Keuleers. *vwr: Useful functions for visual word recognition research*, 2013. URL http://CRAN.R-project.org/package=vwr. R package version 0.3.0. [p111]

M. Kohl. *MKmisc: Miscellaneous functions from M. Kohl*, 2013. URL http://CRAN.R-project.org/package=MKmisc. R package version 0.94. [p111]

V. Laurikari. Efficient submatch addressing for regular expressions. Master's thesis, Helsinki University of Technology, 2001. URL https://github.com/laurikari/tre. [p111]

V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966. [p111]

H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. *Journal of Machine Learning Research*, 2:419–444, 2002. [p111]

N. Logan. C code for Damerau-Levenshtein distance, 2013. URL https://github.com/ugexe/Text--Levenshtein--Damerau--XS/blob/master/damerau-int.c. Last accessed 2014-03-04. [p117]

R. Lowrance and R. Wagner. An extension of the string-to-string correction problem. *Journal of the Association of Computing Machinery*, 22:177–183, 1975. [p116, 117]

G. Navarro. A guided tour to approximate string matching. *ACM Computing surveys*, 33:31–88, 2001. [p115, 117]

S. Needleman and C. D. Wunsch. A general method applicable to the search of similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970. [p115]

K. Schliep and E. Paradis. *Phangorn: phylogenetic analyses in R*, 2013. URL http://CRAN.R-project.org/package=Phangorn. R package version 1.99-1. [p112]

M. Studer, G. Ritschard, A. Gabadinho, and N. Müller. Discrepancy analysis of state sequences. *Sociological Methods and Research*, 40:471–510, 2011. [p111]

E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, 92:191–211, 1992. [p118]

J. Utterström and Y. Arrouye. *uconv - convert data from one encoding to another (Linux man page)*. [p114]

M. van der Loo, E. de Jonge, and S. Scholtus. *deducorrect: Deductive correction, deductive imputation, and deterministic correction.*, 2011. URL https://github.com/markvanderloo/deducorrect. R package version 1.3-5. [p111]

W. Winkler. String comparator metrics and enhanced decision rules in the Fellegi-Sunter model of record linkage. *Proceedings of the Section on Survey Research Methods (American Statistical Association)*, page 354–359, 1990. [p119]

Mark P.J. van der Loo
http://www.markvanderloo.eu
mark.vanderloo@gmail.com