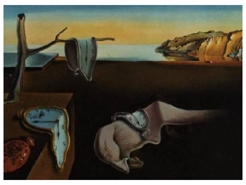

Journal of Statistical Software

October 2020, Volume 95, Issue 8. doi: 10.18637/jss.v095.i08

Phonetic Spelling Algorithm Implementations for R

James P. Howard, II
Johns Hopkins Applied Physics Laboratory

Abstract

The **phonics** package provides several functions for indexing words by their English language pronunciation. Over nearly one hundred years, many different algorithms have been developed to support word and name indexing. From Soundex, developed in the early 20th century and predating the digital computer, through to modern digital phonetic algorithms like Phonex, the **phonics** package provides support for more than a dozen methods. Together, these provide phonetic algorithms appropriate for use in name indexing and name matching across a variety of English language use cases.

Keywords: text processing, phonetics, linguistics, record linkage, demography, R.

1. Introduction

The **phonics** package (Howard, II 2018, 2020) for R (R Core Team 2020) is designed to provide a variety of phonetic indexing algorithms in common and not-so-common use today. The algorithms generally reduce a string to a symbolic representation approximating the sound made by pronouncing the string (Zobel and Dart 1996). They can be used to match names, words, and as a proxy for assorted string distance algorithms.

The general form of a phonetic spelling algorithm is to remove all of the nonletter characters, so that numbers, spaces, hyphens, and other punctuation characters are removed from the subject string. Then, the string is transformed into a single case, typically upper case. These basic operations are so common that several of the implementations within this package share the same opening lines to preprocess a string.

After these initial steps, a variety of processes are used to phonetically reduce the string based on the algorithm. For instance, some remove vowels, while others reduce all vowels to a single symbolic representation. Some, like the Soundex algorithm, use numbers to represent a core set of distinctive sound types, while others like Caverphone, use a more diverse alphabetical symbol array to represent more complex sounds.

```
R> library("phonics")
R> turtles <- c("Leonardo", "Donatello", "Michelangelo", "Raphael")
R> turtles.soundex <- soundex(turtles)
R> turtles.cvphone <- caverphone(turtles)
```

A string array like `Leonardo, Donatello, Michelangelo, Raphael` in combination with the Soundex algorithm becomes `L563, D534, M245, R140` whereas using Caverphone, the array becomes `LNT111, TNTL11, MKLNKL, RF1111`. Together, closely related names can be matched to account for spelling errors, spelling changes associated with language evolution, and related names such as "Steven" and "Stephen."

Some of the algorithms, especially Soundex, are relatively common and implemented in other packages (for instance, **RecordLinkage** by Borg and Sariyar 2020). Others, such as RogerRoot, are not commonly available or are only rarely implemented. The overriding goal off this package is to provide a single interface allowing for each algorithm to be used as a drop-in replacement for the others. Except for the match rating approach, detailed later, this goal is met.

This paper will explain the **phonics** package and some of the implementation details. First, we will explain how to install the package, give a brief overview of the history of the algorithms, along with sources. Each algorithm discussion closes with examples and output samples. Next, we will discuss high-level design choices as well as the unit testing regime. We will then provide notes on the relative performance of the given algorithms. Finally, we will conclude this paper.

2. Description, usage, and examples

The R package **phonics** is available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=phonics` and can be installed using:

```
R> install.packages("phonics")
```

The **phonics** package has several dependencies, but the most notable is the **Rcpp** package (Eddelbuettel and François 2011). Therefore, a C++ compiler is necessary to install the package from source. After installation, the **phonics** package is loaded in the usual way with `library()`. There are no global options that are required or optional. Once loaded, the package's functions are directly usable.

```
R> library("phonics")
```

2.1. Soundex

Soundex is probably the oldest and most well-known system for creating phonetic indices (Wright 1960; Newcombe and Kennedy 1962). Soundex was originally patented in 1918 and 1922, well before the use of digital computers (Knuth 1998, pp. 394–395), and is sometimes known as Russell Soundex. The purpose of Soundex was to provide an index that worked from the sound of a name. Soundex is used by the United States National Archives and Records Administration (NARA) to provide indexing over United States Census Records. In addition, Soundex is available in many database management systems, such as Oracle, MySQL, and others. Due to its precomputer invention, the algorithm itself is quite simple and easy to implement. However, variants on the implementation are also common. This package implements the Knuth description of Soundex.

The final form of a Soundex-encoded index is a letter followed by three digits. Except for the first letter, vowels and vowel sounds are not encoded. If a code were to be longer than four characters, the first letter plus three digits, the code is normally truncated. However, this is the normal operation of the algorithm and not required.

In addition to the traditional Soundex method, this package implements the Refined Soundex algorithm. The Refined Soundex algorithm changes the letter bins from Soundex, allowing for closer sounding groupings, and also removes the truncation step, allowing for full-length encodings. The Refined Soundex algorithm seems to have originated with the implementation of phonetic algorithms included with the Apache Commons library (Fossati and Eugenio 2008), though the underlying ideas of rebinning and lengthening the Soundex encodings goes back to at least Zobel and Dart (1995).

The Soundex algorithm is implemented as the `soundex` function and the Refined Soundex method is given in the `refinedSoundex` function, and we can observe them in the following examples.

```
R> soundex("Catherine")

[1] "C365"

R> soundex("Kathryn")

[1] "K365"

R> soundex(c("Catherine", "Kathryn", "Katrina", "William"))

[1] "C365" "K365" "K365" "W450"

R> refinedSoundex("Catherine")

[1] "C30609080"

R> refinedSoundex("Kathryn")

[1] "K3060908"
```

Both functions accept a `maxCodeLen` that limits the length of the returned code. Except where noted, all the algorithms support the `maxCodeLen` option to change the maximum or expected code length returned, as appropriate.

```
R> refinedSoundex(c("Catherine", "Kathryn", "Katrina", "William"),
+    maxCodeLen = 5)

[1] "C3060" "K3060" "K3069" "W0708"
```

2.2. Metaphone

Metaphone is a family of loosely related phonetic spelling algorithms created by Lawrence Philips (Philips 1990, 2000, 2007). The original algorithm, usually just called Metaphone, is implemented in this package. Metaphone captures 16 core consonant sounds in multiple languages and represents them in the final phonetic spelling. In addition to source language flexibility, Metaphone is also adept at encoding ordinary words, not just names like the Soundex family.

The second and third algorithms, Double Metaphone and Metaphone 3 are not implemented in this package. Double Metaphone was created by Philips to address perceived limitations of Metaphone. In particular, Double Metaphone can provide two results, allowing for more potential avenues to index. Metaphone 3 is patent-encumbered and is unlikely to be provided in short term.

The original Metaphone algorithm can be accessed via the `metaphone` function.

```
R> metaphone(c("Catherine", "Kathryn", "Katrina", "William"))

[1] "KORN" "KORN" "KTRN" "WLM"
```

2.3. New York State identification and intelligence system

The New York State identification and intelligence system (NYSIIS) method, named for the New York State agency that developed it (Silbert 1970), has become a commonly used name indexing algorithm. This is largely due to its relative simplicity, good documentation from multiple sources, and ease of use. The NYSIIS, in contrast to Soundex, tries to capture fine differentiations in pronunciation in different names (Taft 1970). Therefore, “knight” and “night” both observe the “n”-sound at the start, and while the NYSIIS method captures this, it does draw distinctions between roughly similar sounds. Therefore, NYSIIS does not merge, for instance, the b-sound and p-sound into one. In addition to this algorithm, a modified version was documented by Lynch and Arends (1977).

The algorithm is available via the `nysiis` function and the modified version of NYSIIS is accessed via the option `modified`.

```
R> nysiis(c("Catherine", "Kathryn", "Katrina", "William"))

[1] "CATARA" "CATRYN" "CATRAN" "WALAN"

R> nysiis(c("Catherine", "Kathryn", "Katrina", "William"), modified = TRUE)

[1] "CATARA" "CATRAN" "CATRAN" "WALAN"
```

2.4. Oxford name compression algorithm

Despite its name, the Oxford name compression algorithm (ONCA) is not a compression algorithm in the traditional sense (Sayood 2012) that the original text can be restored from the output of the algorithm. Instead, ONCA is a phonetic spelling algorithm. ONCA was developed for use in linking medical records across the British National Health Service (Gill 1997).

The ONCA method is unique among those phonetic spelling algorithms presented here in that it is a two-step method where each step is another method. First, a string is phonetically reduced using the NYSIIS algorithm. Second, the result of the NYSIIS process is then run through Soundex. The result of the second step is the final result of the ONCA method.

Gill (1997) notably claims the first step of NYSIIS reduces occurrences of some edge cases better than Soundex alone, while still retaining the characteristic four-character index that Soundex produces. He also notes that the algorithm has been successfully used to index and link ten million records in the Oxford Record Linkage Study.

The ONCA algorithm is provided in the `onca` function.

```
R> onca(c("Catherine", "Kathryn", "Katrina", "William"))

[1] "C365" "C365" "C365" "W450"
```

It is important to note that the output of the ONCA function is, at first glance, indistinguishable from the output of Russell Soundex over a similar input string. As a result, visual inspection cannot be used to discern what algorithm was used on such an output.

2.5. Caverphone

Caverphone is a family of two phonetic algorithms first documented by Hood (2002). Caverphone was created as part of the Caversham project at the University of Otago, documenting social mobility in late 19th and early 20th century in New Zealand. Accordingly, Caverphone was developed to provide efficient matching of names on electoral rolls. In practice, the Caversham project applied Caverphone after exact matches were identified. This reduced the likelihood of false positives by removing potential targets from the index pool.

Caverphone was not meant for general use. While openly documented, the algorithm was developed specifically for the data at hand in the Caversham project. The Caverphone 2 algorithm was updated by Hood (2004) to provide a more generalized approach to a phonetic algorithm based on the experiences of Caversham and the original Caverphone. While generalized, both are based on local pronunciation in and around Dunedin, Otago, New Zealand.

The Caverphone family works by addressing more complex sounds. For instance, the sound "tch", such as at the start of "Tchaikovsky," is encoded with a sound of "ch" like it is pronounced. In contrast, an algorithm like Soundex would encode Tchaikovsky as starting with a T, pulling it out of the class of names starting with the "ch" sound. Other complex sounds such as "ough," "gn," and "dg" are included and coded like their English-language pronunciation.

The original Caverphone function returns a six-character code for the name. If a name encoding is under six characters, it is padded with `1`s to reach six characters. Caverphone 2 works similarly, but with a 10-character output code, padded with `1`s.

The Caverphone algorithm is provided by the `caverphone` functions. The Caverphone 2 algorithm is provided by setting the `modified` option to `TRUE`.

```
R> caverphone(c("Catherine", "Kathryn", "Katrina", "William"))

[1] "KTRN11" "KTRN11" "KTRN11" "WLM111"

R> caverphone(c("Catherine", "Kathryn", "Katrina", "William"),
+    modified = TRUE)

[1] "KTRN111111" "KTRN111111" "KTRNA11111" "WLM1111111"
```

2.6. Cologne

The Cologne phonetic algorithm was developed by Postel (1969) at IBM to provide Soundex-like functionality for German language names. German names often include additional diacritics (as demonstrated by the German name for Cologne, “Köln”) and more variant spellings for common names, due to later orthographic standardization. Accordingly, Soundex was not suitable for German names and Cologne was meant to fill that void.

Due to orthographic conventions of German, additional characters, such as vowels with umlauts (“ä,” “ö,” and “ü,”), and the eszett (“ß”) are included. The examples below include four samples including these characters. However, even if these characters are included, other unknown characters will return a warning if encountered. As shown below, names with hyphens or spaces will produce a warning unless the `clean` parameter is set to `FALSE`.

Cologne, unlike many of the other algorithms in this package, does not produce a fixed-length or maximum-length output, under normal operations. The algorithm will continue encoding a string until there are no characters left to encode. The code itself will include digits from 1 to 9.

```
R> cologne(c("Catherine", "Kathryn", "Katrina", "William"))

[1] "4276" "4276" "4276" "356"

R> cologne(c("Müller", "Schluß"))

[1] "657" "858"
```

2.7. Lein

The origins of the Lein name coding algorithm are unclear, though it seems probable Lein is an acronym standing for “law enforcement information network,” a name used by several states for centralized law enforcement databases. We do know that Lein was documented by Lynch and Arends (1977) as early as 1977 as part of a United States Department of Agriculture (USDA) project to analyze the use of several name coding algorithms.

The algorithm itself retains the first character, eliminates vowels and duplicate sounds, then codes the remaining letters as numbers. Each letter is coded to a number from 1 to 5. The output codes are limited to `maxCodeLen` characters, a letter followed by digits, where `maxCodeLen` defaults to four. If the resultant code is fewer than `maxCodeLen` characters, the output is padded with 0 characters to reach the correct length.

The Lein algorithm is provided in the `lein` function.

```
R> lein(c("Catherine", "Kathryn", "Katrina", "William"))

[1] "C132" "K132" "K132" "W320"
```

2.8. Census-modified Statistics Canada

Like Lein, the census-modified Statistics Canada method is documented by Lynch and Arends (1977). Given the name, the provenience is almost certainly a modification of an approach developed by Statistics Canada. Fair (2004) has more recent information on methods for record-linkage used by Statistics Canada.

The algorithm is also simple like Lein. Vowel sounds are eliminated and duplicate sounds are reduced. Individual letters are not recoded making this an extremely fast algorithm.

The census-modified Statistics Canada algorithm is provided in the `statcan` function.

```
R> statcan(c("Catherine", "Kathryn", "Katrina", "William"))

[1] "CTHR" "KTHR" "KTRN" "WLM"
```

2.9. RogerRoot

Like Lein, the RogerRoot method is documented by Lynch and Arends (1977). There seems to be no other documentation that provides any details about the origin or purpose of the algorithm beyond the obvious application to phonetic spelling and indexing. The algorithm is also simple like Lein. Vowel sounds are eliminated and duplicate sounds are reduced, leading to a five-digit numerical code. Because of the narrow field of options (the digits 0–9), the RogerRoot method is more likely to lead to a larger number of collisions for different names than algorithms with a larger output space, like Metaphone.

The RogerRoot algorithm is provided in the `rogerroot` function.

```
R> rogerroot(c("Catherine", "Kathryn", "Katrina", "William"))

[1] "07142" "07142" "07142" "45300"
```

2.10. Phonex

Phonex was created by Lait and Randell (1996) after a detailed analysis of several other algorithms with three specific goals in mind: improved accuracy, faster runtime, and overall simplicity. The creators note that by starting with Soundex as a baseline and making improvements from there, Phonex was likely to be well-suited to English language names and not well suited to other languages or general-purpose word matching.

The specific problems Lait and Randell (1996) observed with some algorithms was that identically-sounding names such as "Filip" and "Philip" would map to different output codes, names starting with an H followed by a vowel are often phonetically the same as the same name without an H, and several sound-equivalent single-character pairs are ignored. By creating an output algorithm and changing it slightly over a defined test input, the authors were able to search for a viable replacement.

The Phonex algorithm is provided via the `phonex` function. Like Soundex, Phonex encodings are a single letter followed by three digits encoding the first four sounds of the name.

```
R> phonex(c("Catherine", "Kathryn", "Katrina", "William"))

[1] "C365" "C365" "C365" "W450"
```

2.11. Match rating approach

The match rating approach (MRA) was developed by Western Airlines to match names within their reservation system (Moore, Kuhns, Trefftzs, and Montgomery 1977). Unlike other algorithms described here, MRA is a two-stage algorithm with separate encoding and comparison routines. For instance, the results of Soundex on two different strings can be directly compared to test for equality:

```
R> soundex("Catherine") == soundex("Kathryn")

[1] FALSE

R> soundex("Kathryn") == soundex("Katrina")

[1] TRUE
```

The MRA encoding algorithm may return different encodings for similar strings that should match. So a second stage is used to compare to MRA-encoded strings. There, an algorithm that measures the amount of similarity between two encoded strings, similar to a string distance algorithm (Van der Loo 2014), is used. The encoding algorithm is provided by `mra_encode` and the comparison algorithm is provided by `mra_compare`.

```
R> (Katherine <- mra_encode("Katherine"))

[1] "KTHRN"

R> (Katarina <- mra_encode("Katarina"))

[1] "KTRN"

R> mra_compare(Katherine, Katarina)

[1] TRUE
```

The threshold necessary to establish similarity *gets smaller* as the encoded strings get larger. This leads to some interesting results. For instance, Catherine and William match as names.

```
R> mra_compare(mra_encode("Catherine"), mra_encode("William"))
```

```
[1] TRUE
```

On the other hand, Kate and Will do not match, though Will and Bill do successfully match.

```
R> mra_compare(mra_encode("Kate"), mra_encode("Will"))
```

```
[1] FALSE
```

```
R> mra_compare(mra_encode("Bill"), mra_encode("Will"))
```

```
[1] TRUE
```

Fully understanding the implications of the MRA comparison algorithm is advised before adopting MRA for production use.

2.12. The `phonics` command

The `phonics` function provides a unified interface to all of the included algorithms, except MRA. The `phonics` function requires two parameters. The first is `word`, which is a character string or vector of character strings to be processed.

The `method` is a vector containing one or more of:

- `"caverphone"`,
- `"caverphone.modified"`,
- `"cologne"`,
- `"lein"`,
- `"metaphone"`,
- `"nysiis"`,
- `"nysiis.modified"`,
- `"onca"`,
- `"onca.modified"`,
- `"onca.refined"`,
- `"onca.modified.refined"`,
- `"phonex"`,
- `"rogerroot"`,
- `"soundex"`,
- `"soundex.refined"`, or
- `"statcan"`.

The method names including “modified” use the underlying method with the `modified` parameter set to `TRUE`. The method names including “refined” replace the Soundex method with the Refined Soundex method.

The `phonics` function returns a data frame with a column names `"word"` containing the `word` vector. The remaining columns are the results for each phonetic algorithm included in the `method` option.

3. Implementation

The implementations of the functions in the **phonics** package follow one of two outlines, depending on the nature of the function. The majority of these functions are implemented as sequential processes of search and replace runs over the subject string. Others are implemented by processing the string character by character into a final form; these are implemented in **Rcpp** to provide greater speed. The implementation chosen for each function flows naturally from the string processing rules given by the algorithm.

3.1. Rcpp-based implementations

Three of the algorithms contained in the **phonics** package are implemented in C++ and bridged into R using **Rcpp**. These are Soundex (`soundex`), Refined Soundex (`refinedSoundex`), and Metaphone (`metaphone`).

3.2. Regular expression implementations

All other algorithms included in this package are implemented in pure R, usually through the extensive use of regular expressions for string replacement (Teetor 2011; Thompson 1968). As many phonetic algorithms rely on common themes, such as replacing leading "KN" (as in "knight") with an "N" to reflect pronunciation, regular expressions provide a convenient way to implement these translations.

The simplest case is most likely the census-modified Statistics Canada algorithm (`statcan`). It consists of the following steps, other than boilerplate nonalphabetical character removal and truncation for length:

1. Capture the first letter and remove it.
2. Remove all vowels and the letter "Y."
3. Add the first letter back to the start of the encoded string.
4. Remove all duplicated letters.

Steps 1 and 3 are accomplished using R's internal string splitting and reassembly routines. Steps 2 and 4 are easily implemented via regular expressions. Other regular expression based algorithms may include more steps, but are not substantively more complex and follow the same basic pattern.

3.3. Oxford name compression algorithm

Due to the unique method used by ONCA, the implementation is also slightly different. The `onca` function accepts a vector of strings and will first process the vector with the `nysiis` function and then reprocess the vector with `soundex`. We do not separately implement ONCA.

3.4. Unknown character handling

Each of these algorithms operates in a similar way and because of that, they have some common features. These features are expressed in a standard opening at the start of each. This example is taken from the `nysiis` function, but similar code exists in each function.

These algorithms are not case sensitive and the first step each of these take is to transform all of the strings into a single case. Generally, this is upper case, but some algorithms use lower case, depending on implementation details.

```
word <- toupper(word)
```

Further, we want to ensure special cases are handled appropriately. The input parameter `word` might be a single character string or vector of character strings. The next two lines first test for `NULL`, which can only be passed as a single `NULL`. We treat that as `NA` and convert it into an `NA`. Then we record a list of all entries in the vector which contain `NA`. This will be used later.

```
word[is.null(word)] <- NA
listNAs <- is.na(word)
```

The next three lines handle unknown characters. For instance, we test for any characters not in the range of A–Z (a–z if the algorithm used `tolower()` earlier), and record their position. If there are any and the parameter `clean` is `TRUE`, the default, then we post a warning and continue processing. Finally, we remove all characters that are not in the range of A–Z (again, a–z if the strings were converted to lower case).

```
if (any(nonalpha <- grepl("[^A-Z]", word, perl = TRUE)) && clean) {
  warning("unknown characters found, results may not be consistent")
}
word <- gsub("[^A-Z]*", "", word, perl = TRUE)
```

If the parameter `clean` is `FALSE`, no warning is posted. Either way, the algorithm continues processing the same way. This works for any phonetic algorithm operating over the standard English alphabet. For those functions operating over the German or French languages, special characters like `"a` or `"o` are converted to `a` or `o`, or other characters as the algorithm requires, before the standard opening.

Similarly, these implementations share a common closing. First, any entry in the vector which contained `NA` initially is set to `NA`. Depending on the internal operation of the algorithm, any `NA` may not be preserved through the entire operation. Therefore, this ensures a return of `NA` for any input `NA` or `NULL`.

```
word[listNAs] <- NA
if (clean) {
  word[nonalpha] <- NA
}
```

Any element of the original vector which was marked as including nonalphabetical characters is marked as `NA` if `clean` is `TRUE`. Through this, nonconforming inputs are marked as `NA`. Regardless of the value of `clean`, the algorithm tried to process the element. Accordingly, if `clean` is `FALSE`, the function returns whatever was processed. This may or may not conform with any expected outputs and is not consistent across different platforms or even versions of R.

3.5. Maximum return length

The final common option is the argument `maxCodeLen`. Some of the algorithms have a fixed output length; for instance, Soundex values are one letter followed by three numbers. Each of the algorithms included allows with `maxCodeLen` for truncation at a different length. If the algorithm has a fixed output length, there is a defined padding value; for instance, if a Soundex value would only be two characters long, two zeros are appended.

If `maxCodeLen` is greater than the default, the padding is extended to meet the length of `maxCodeLen`. If an algorithm does not use padding normally, none is used, and the returned value may be shorter than `maxCodeLen`. Function `phonics()` and the MRA functions do not include a `maxCodeLen` parameter.

3.6. Unit testing

The **phonics** package includes a complete test suite for regression using the **testthat** package (Wickham 2011). For each phonetic algorithm, and major variant, there is a comma-separated values (CSV) file containing sample strings and correct sample outputs for the algorithm (Shafranovich 2005, pp. 2–4; Raymond 2003, pp. 112–122). The CSV file is processed twice for each algorithm. First, each element of the vector is tested individually to ensure the algorithm is implemented correctly. Second, the vector is tested again in one step, to ensure the vectorized implementation is correct.

Each set of samples has been separately generated, therefore, we are regressing against an independent data source. These data sources include documentation, examples from independent literature, independent implementations of the algorithm, or independently-generated test suites. Accordingly, we have some level of assurance that the results generated are correct and not just correct compared to our own self-generated outputs.

4. Performance metrics

In order to better understand the algorithms and when to use which one, we provide a set of key performance metrics for evaluating the algorithms. We provide a measure of the amount of time necessary to use each algorithm. This will obviously vary from system to system, however, the relative performance timings would remain approximately consistent across computer systems and architectures.

We also provide metrics for the collision space of each algorithm. Reducing the chance of collision in the outputs of phonetic spelling algorithms for nonsimilar inputs is a critical performance measurement. Like Lynch and Arends (1977), we provide three important measurements of the collision space that can be used to evaluate suitability of each algorithm for a given task.

4.1. Relative timings

Using the Comet supercomputer system (Moore *et al.* 2014; Strande *et al.* 2017) at the San Diego Supercomputer Center (SDSC), each algorithm was benchmarked with the **microbenchmark** package (Mersmann 2019).

We use a list of all surnames appearing 100 or more times in the 2010 United States Census

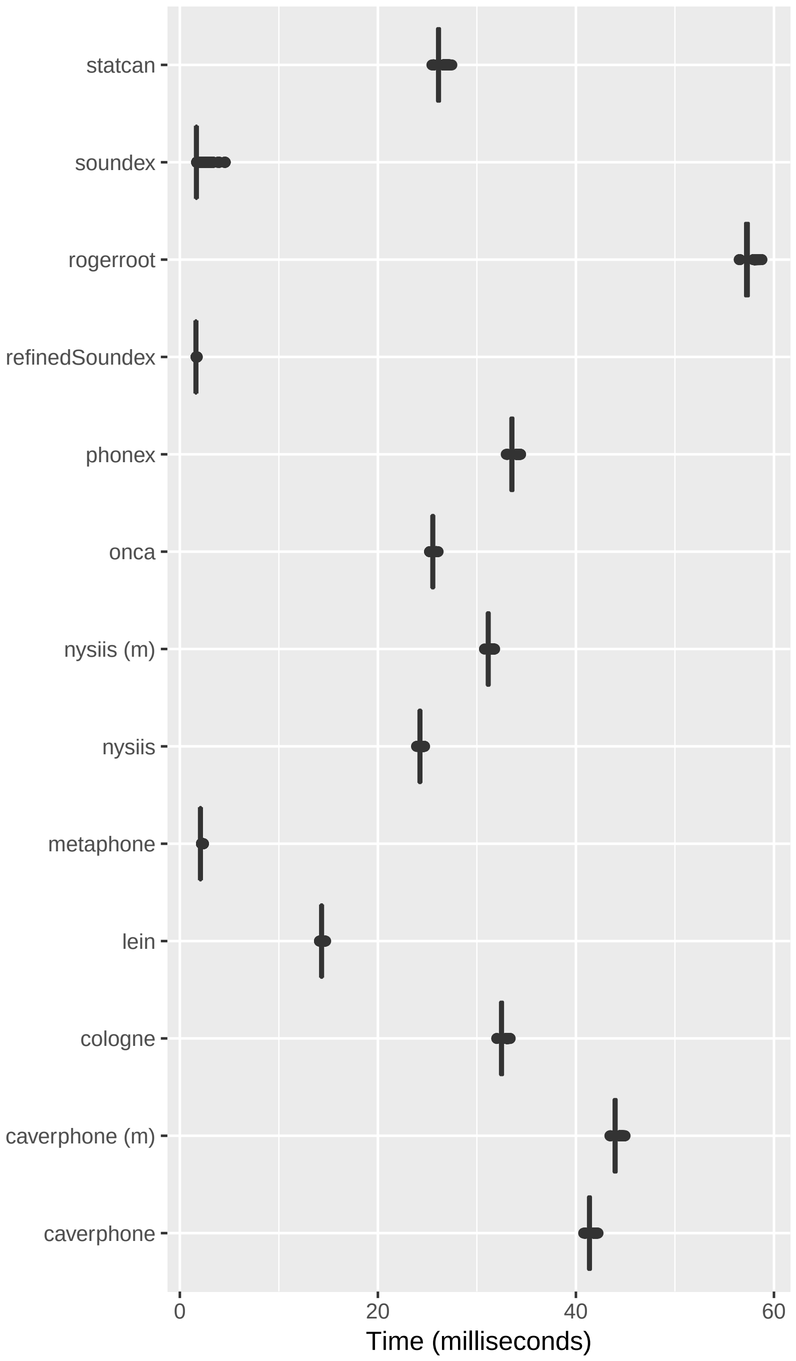

Figure 1: Absolute timings of different phonetic algorithms.

as provided by the United States Census Bureau (USCB; United States Census Bureau 2016). This dataset consists of 162,253 different surnames in the United States. This dataset also includes relative rank and frequency of appearance. We remove all entries with rank zero, to eliminate an "other" category, representing all names appearing fewer than 100 times. The names were resampled into a random dataset of 2000 names with replacement and weighted by the frequency of appearance.

Each phonetic spelling algorithm is run over the random datasets 100 times and timed using the `microbenchmark` function. Using the `mclapply` function from the **parallel** package, the resampling process and benchmarking were executed 10,000 times. There were a total of 1,000,000 timed executions made for each of 13 phonetic spelling algorithms included in this package. The MRA algorithm is excluded from this benchmark due to its operational requirements. Since the MRA algorithm functions by calculating a distance between two encoded character strings, this would not be comparable to the straightforward encoding used by all of the other included algorithms.

The mean of each `microbenchmark` execution was collected and the boxplot of those 10,000 means for each algorithm is provided in Figure 1.

As we would expect, the three phonetic spelling functions implemented using **Rcpp** (`soundex`, `refinedSoundex`, and `metaphone`) outperform those implemented in pure R. In addition, the **Rcpp** implementations provide more consistent results. We also note that `lein` provides the best performance of those implementations in pure R. This is because `lein` consists of only seven regular expression replacements and does not require complex bookkeeping to manage out-of-band replacements.

4.2. Collision space analysis

In addition to timing metrics, we also include an analysis of the collision space for each algorithm. Again, using the Comet supercomputer system, the collection of phonetic spelling algorithms was evaluated against the USCB name list. We remove all entries with a rank of zero, to eliminate an "other" category. The names were resampled into a random dataset of 100,000 names with replacement and weighted by the frequency of appearance. Sampling with replacement was chosen to model actual populations that the algorithms may be applied to.

The suite of 13 algorithms, not including MRA, were applied to each name and the resultant collisions measured. The process was repeated as a job of 10,000 repetitions, and the job was repeated 100 times on the Comet system. The job was completed using 6457 compute hours. Using 2400 cores, this simulation required less than three wall-clock hours to complete.

The collision space analysis is predicated on the idea that a good name coding operation has two properties. First, variations on a name should produce the same code. Second, the size of the codes should be minimized. Over a random sample of 100,000 names, with replacement, there are likely to be some names that repeat across the list. There will also be some names that are variants of each other appearing on the list, and these can possibly produce the same phonetic spelling. Finally, there will be some names that are unrelated to each other on the list, that produce the same encoding.

The first metric we consider is how many unique codes are produced by the chosen algorithm on a simulated dataset. This metric feeds additional analysis about the collision space. The five-number summary, mean, and standard deviation for each algorithm are given in Table 1.

For instance, we are interested in understanding what the potential output space of the encoding algorithms are. Soundex has a potential output space of 6734 codes and of those, the most any simulation required was 3679. Lein is simpler, with a potential output space of only 4056 codes. Of those, at most 2642 are required in any simulation. Phonex, like Soundex, has a potential outputspace of 6734 codes but only required at most 1905 codes in the simulation. At the other end of the spectrum, the Refined Soundex algorithm can produce more than 10^9 potential codes, though the most required was only 13,996 for any simulation,

Algorithm	Min.	1st Qtr	Median	3rd Qrt	Max.	Mean	SD
`soundex`	3475	3564	3578	3592	3679	3577.87	20.52
`refinedSoundex`	13304	13590	13635	13681	13996	13635.38	67.89
`nysiis`	11992	12254	12295	12337	12581	12295.27	61.16
`nysiis` (m)	10931	11154	11192	11231	11486	11192.33	57.24
`lein`	2482	2551	2561	2572	2642	2561.45	15.48
`caverphone`	6370	6555	6585	6615	6790	6585.00	45.10
`caverphone` (m)	7153	7362	7395	7427	7665	7394.65	47.93
`cologne`	6770	6951	6981	7012	7238	6981.51	45.24
`metaphone`	11348	11603	11646	11688	11992	11645.81	62.88
`onca`	3297	3383	3396	3410	3494	3396.34	20.30
`phonex`	1789	1838	1846	1854	1905	1845.85	11.85
`rogerroot`	5015	5163	5186	5209	5372	5185.72	33.87
`statcan`	10309	10508	10544	10581	10795	10544.32	53.73

Table 1: Summary statistics for the number of surname codes by algorithm.

Algorithm	Min.	1st Qtr	Median	3rd Qrt	Max.	Mean	SD
`soundex`	7.92	8.12	8.16	8.19	8.40	8.16	0.05
`refinedSoundex`	2.10	2.13	2.14	2.15	2.18	2.14	0.01
`nysiis`	2.33	2.37	2.37	2.38	2.43	2.37	0.01
`nysiis` (m)	2.56	2.60	2.61	2.62	2.67	2.61	0.01
`lein`	11.04	11.34	11.39	11.44	11.75	11.39	0.08
`caverphone`	4.29	4.41	4.43	4.45	4.57	4.43	0.03
`caverphone` (m)	3.84	3.93	3.95	3.96	4.06	3.95	0.02
`cologne`	4.06	4.16	4.18	4.20	4.30	4.18	0.03
`metaphone`	2.45	2.50	2.51	2.51	2.56	2.51	0.01
`onca`	8.34	8.56	8.59	8.63	8.87	8.59	0.06
`phonex`	15.27	15.74	15.81	15.89	16.35	15.81	0.11
`rogerroot`	5.44	5.60	5.63	5.65	5.80	5.63	0.04
`statcan`	2.70	2.76	2.77	2.78	2.83	2.77	0.01

Table 2: Summary statistics for the number of unique surnames per code by algorithm.

the largest of any phonetic algorithm we included here.

The second metric included is the number of unique surnames per code generated. As we have already noted, a single name may appear multiple times in each simulated dataset due to sampling with replacement. In particular, high frequency names are more likely to appear multiple times in each simulation. Accordingly, we want to know how many unique surnames, on average, map to each code. This performance metric captures that answer. The five-number summary, mean, and standard deviation for each algorithm are given in Table 2.

The number of unique surnames per code tells us how the algorithm will behave with respect to different names. For instance, while we may prefer that similar names should map to the same phonetic code, we cannot guarantee that with any of these algorithms. So the fewer number of unique names per code suggests that names that are less different from each other are more likely to map to different codes.

In the case of the Phonex algorithm, we can see that more than 15 unique names are assigned for each unique code. This suggests a high rate of collisions for only moderately close names. This is the highest collision rate of any of the provided algorithms. The next highest rate is the Lein algorithm with more than 11 unique names per code; both Soundex and ONCA transform more than 8 unique names per unique code. The higher this metric is, the more unique names are assigned to each code and this might imply that some clearly differentiated names are matched. There are extensive examples for Soundex, in particular, where this applies. For instance, the name Washington has several matches:

```
R> soundex(c("Washington", "Wiggins", "Wozniak"))

[1] "W252" "W252" "W252"
```

Neither Wiggins nor Wozniak are similar enough to Washington to justify a match. These are clearly not alternative forms of the same name and are not misspellings. Further, there are a total of 78 names in the USCB name list with a Soundex value of W252. These sorts of accidental matches are more common when there are more unique names per encoding.

At the other end of the spectrum, several algorithms produce fewer than three unique names per code. Those algorithms, i.e., the Refined Soundex, NYSIIS, the modified NYSIIS, Metaphone, and the census-modified Statistics Canada algorithms, have very broad output spaces reducing the number of collisions that occur. We can see the advantages with the previous example:

```
R> nysiis(c("Washington", "Wiggins", "Wozniak"))

[1] "WASANG" "WAGAN"  "WASNAC"

R> statcan(c("Washington", "Wiggins", "Wozniak"))

[1] "WSHN" "WGNS" "WZNK"
```

For comparison, there are only 11 names in the USCB name list with an NYSIIS value of WASANG. Of these, all but Washington are variant forms of the name Weissinger. Similarly, the census-modified Statistics Canada algorithm provides only 9 entries in the name list with a value of WSHN, the census-modified Statistics Canada value of Washington.

The final metric we consider is how many surnames from the simulated dataset map to each code. This is slightly different from the last metric because it allows for a surname to appear multiple times in the simulated dataset. For instance, consider the last name Smith, the most common last name in the United States. We would expect any reasonable sample to include more than one Smith (compare to McKinney 1966). This metric, therefore, provides a measurement of how many times a code will appear in a real world dataset, and an associated picture of how well a phonetic spelling algorithm would fit as an indexing algorithm. The five-number summary, mean, and standard deviation for each algorithm are given in Table 3.

The results of this metric are not surprising given the results of the second metric. The Lein and Phonex algorithms match a larger number of names to a smaller number of output codes. As we already know, Lein has a very small output space, and this is completely expected.

Algorithm	Min.	1st Qtr	Median	3rd Qrt	Max.	Mean	SD
`soundex`	27.18	27.84	27.95	28.06	28.78	27.95	0.16
`refinedSoundex`	7.14	7.31	7.33	7.36	7.52	7.33	0.04
`nysiis`	7.95	8.11	8.13	8.16	8.34	8.13	0.04
`nysiis` (m)	8.71	8.90	8.93	8.97	9.15	8.93	0.05
`lein`	37.85	38.88	39.05	39.20	40.29	39.04	0.24
`caverphone`	14.73	15.12	15.19	15.26	15.70	15.19	0.10
`caverphone` (m)	13.05	13.46	13.52	13.58	13.98	13.52	0.09
`cologne`	13.82	14.26	14.32	14.39	14.77	14.32	0.09
`metaphone`	8.34	8.56	8.59	8.62	8.81	8.59	0.05
`onca`	28.62	29.33	29.45	29.56	30.33	29.44	0.18
`phonex`	52.49	53.94	54.17	54.41	55.90	54.18	0.35
`rogerroot`	18.62	19.20	19.28	19.37	19.94	19.28	0.13
`statcan`	9.26	9.45	9.48	9.52	9.70	9.48	0.05

Table 3: Summary statistics for the number of non-unique surnames per code by algorithm.

At the other end, we can see the effects of the output space width on the Refined Soundex algorithm, as it produced the lowest mean number of surnames to output codes. Similarly, the NYSIIS, the modified NYSIIS, Metaphone, and the census-modified Statistics Canada algorithms all produce roughly comparable collision frequencies over the sample datasets.

Unfortunately, there is no good decision rule to use when selecting a phonetic spelling algorithm for a project. Excepting some obviously divergent solutions, such as the Lein algorithm, the different phonetic spelling algorithms behave roughly equivalently on the sample datasets. There are, however, rules of thumb available to support the selection process. One should clearly select an algorithm that supports the language they are coding, quite possibly with accent restrictions. The Caverphone algorithms were designed with a New Zealand English-speaker in mind, for instance. Or the Cologne algorithm was designed with the German language in mind.

Beyond the rules of thumb, it is also important to understand the application the algorithm will be used for. If the purpose is to create an index of the name, the Lein algorithm may be a good answer, since the narrow output space can be advantageous. If the purpose is to help catch spelling errors in names from a data entry source, then something at the lower end of the middle ground is likely best. The application will match some errors, but not more extreme errors. Some testing is probably necessary to find the best fit for a given application.

Much of this analysis, however, may not be helpful. Often when working with an algorithm like this, the issue of compatibility with some other system may be necessary. In that case, the selection is likely already made when the other system dictates the algorithm to use. If some other system is using Soundex to index a dataset, then using Soundex is a requirement.

Finally, when selecting a phonetic spelling algorithm without external constraints, it is reasonable to consider the actual application, including example cases. In the unlikely event your dataset is composed of every person in the United States, this analysis presented is likely sufficient. However, like some of these algorithms' creators, your application may have a highly constrained or particular dataset. For instance, it might only be dominated by names originating from a specific language. In this case, it is reasonable to sample the list of names and perform an *ad hoc* collision space analysis to better understand what algorithm performs best on the data. In that case, regardless of the analysis presented here, there is strong justification for adopting that algorithm for that application.

5. Summary

In this paper, we have outlined the **phonics** package for R, and some key performance characteristics of the algorithms provided. Included in this package are several English-, German-, and French-language suitable algorithms for phonetically reducing names and strings. These can be used for comparison and indexing, as well as later record-linkage. In addition to providing suitable implementations of many different phonetic algorithms, this package provides a suite of test values for each algorithm allowing us to test the veracity of output in unit testing, or by other implementations in other languages. Finally, this document has provided a sketch outline of the history of all included algorithms. This includes, where possible, authoritative sources and earliest available sources for all of the algorithms.

Additional work is required to better understand the performance of these algorithms. Modern data scientists have adopted performance metrics based on the percentage of hits that are true-positives and true-negatives. We usually measure this as the area under the curve (AUC) under the receiver operating characteristic (ROC) curve (Hanley and McNeil 1982; Bradley 1997). However, the field producing phonetic algorithms does not have a standard dataset of "correctly" and "incorrectly" spelled names to provide a common baseline of analysis. Further work to create that list would be beneficial. With such a baseline, these algorithms could undergo a series of refinements to optimize their performance.

Finally, we are forced to accept that different algorithms are written to support different languages and pronunciation rules. Rapidly diversifying populations globally (e.g., Kabisch and Haase 2011), will place some pressure on the use of these algorithms. Some names are not transliterated well and others adhere to pronouncing rules and sounds not available in the native host language. A new set of algorithms will be necessary as time goes on that can allow for broad source language diversity and complex evolving rules in the host language.

Acknowledgments

The author thanks Oliver Keyes, Amanda Hood, and Kyle Haynes, for their contributions and improvements to this package. The author also thanks Kristine Harshaw for reviewing early drafts of this paper. Finally, the author thanks two anonymous reviewers who significantly improved this package.

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation (NSF) grant number ACI-1548562 (Towns *et al.* 2014). This work used the XSEDE Comet supercomputer at the SDSC through allocations TG-DBS170012 and TG-ASC150024 (Moore *et al.* 2014; Strande *et al.* 2017).

References

Borg A, Sariyar M (2020). ***RecordLinkage**: Record Linkage in R.* R package version 0.4-12, URL `https://CRAN.R-project.org/package=RecordLinkage`.

Bradley AP (1997). "The Use of the Area under the ROC Curve in The Evaluation of Machine Learning Algorithms." *Pattern Recognition*, **30**(7), 1145–1159. doi:10.1016/s0031-3203(96)00142-2.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08.

Fair M (2004). "Generalized Record Linkage System – Statistics Canada's Record Linkage Software." *Austrian Journal of Statistics*, **33**(1&2), 37–53.

Fossati D, Eugenio BD (2008). "I Saw TREE Trees in the Park: How to Correct Real-Word Spelling Mistakes." In N Calzolari, K Choukri, B Maegaard, J Mariani, J Odijk, S Piperidis, D Tapias (eds.), *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*. European Language Resources Association (ELRA), Marrakech.

Gill LE (1997). "OX-LINK: The Oxford Medical Record Linkage System." In *Proceeding of the International Record Linkage Workshop and Exposition*, chapter 2, pp. 15–33. National Academy Press.

Hanley JA, McNeil BJ (1982). "The Meaning and Use of the Area Under a Receiver Operating Characteristic (ROC) Curve." *Radiology*, **143**(1), 29–36. doi:10.1148/radiology.143.1.7063747.

Hood D (2002). "Caverphone: Phonetic Matching Algorithm." *Technical Report Caversham Project Occasional Technical Paper CTP060902*, University of Otago, Dunedin, Otago, New Zealand.

Hood D (2004). "Caverphone Revisited." *Technical Report Caversham Project Occasional Technical Paper CTP150804*, University of Otago, Dunedin, Otago, New Zealand.

Howard, II JP (2018). "Phonetic Algorithms in R." *Journal of Open Source Software*, **3**(22), 480. doi:10.21105/joss.00480.

Howard, II JP (2020). ***phonics**: Phonetic Spelling Algorithms.* R package version 1.3.8, URL https://CRAN.R-project.org/package=phonics.

Kabisch N, Haase D (2011). "Diversifying European Agglomerations: Evidence Of Urban Population Trends for the 21st Century." *Population, Space and Place*, **17**(3), 236–253. doi:10.1002/psp.600.

Knuth DE (1998). *The Art of Computer Programming*, volume 3. 2nd edition. Addison-Wesley, Upper Saddle River, New Jersey.

Lait AJ, Randell B (1996). "An Assessment of Name Matching Algorithms." *Technical report*, University of Newcastle upon Tyne, Newcastle upon Tyne, England.

Lynch BT, Arends WL (1977). "Selection of a Surname Coding Procedure for the SRS Record Linkage System." *Technical report*, United States Deptartment of Agriculture, Statistical Reporting Service, Washington. URL http://handle.nal.usda.gov/10113/27833.

McKinney EH (1966). "Generalized Birthday Problem." *The American Mathematical Monthly*, **73**(4), 385–387. doi:10.2307/2315408.

Mersmann O (2019). ***microbenchmark**: Accurate Timing Functions.* R package version 1.4-7, URL https://CRAN.R-project.org/package=microbenchmark.

Moore GB, Kuhns JL, Trefftzs JL, Montgomery CA (1977). *Accessing Individual Records from Personal Data Files Using Non-Unique Identifiers*, volume 500-2. United States Government Printing Office, Washington.

Moore RL, Baru C, Baxter D, Fox GC, Majumdar A, Papadopoulos P, Pfeiffer W, Sinkovits RS, Strande S, Tatineni M, Wagner RP, Wilkins-Diehr N, Norman ML (2014). "Gateways to Discovery: Cyberinfrastructure for the Long Tail of Science." In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, XSEDE'14, pp. 39:1–39:8. ACM, New York. doi:10.1145/2616498.2616540.

Newcombe HB, Kennedy JM (1962). "Record Linkage: Making Maximum Use of the Discriminating Power of Identifying Information." *Communications of the ACM*, **5**(11), 563–566. doi:10.1145/368996.369026.

Philips L (1990). "Hanging on the Metaphone." *Computer Language*, **7**(12).

Philips L (2000). "The Double Metaphone Search Algorithm." *C/C++ Users Journal*, **18**(6), 38–43.

Philips LBF (2007). "System and Method for Phonetic Representation." Google Patents. US Patent App. 11/890,334.

Postel HJ (1969). "Die Kölner Phonetik. Ein Verfahren zur Identifizierung von Personennamen auf der Grundlage der Gestaltanalyse." *IBM-Nachrichten*, **19**, 925–931.

Raymond ES (2003). *The Art of Unix Programming*. Addison-Wesley Professional, Boston.

R Core Team (2020). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Sayood K (2012). *Introduction to Data Compression*. The Morgan Kaufmann Series in Multimedia Information and Systems, 4th edition. Morgan Kaufmann.

Shafranovich Y (2005). "Common Format and MIME Type for Comma-Separated Values (CSV) Files." RFC 4180 (Informational). Updated by RFC 7111.

Silbert JM (1970). "The World's First Computerized Criminal-Justice Information Sharing System the New York State Identification and Intelligence System (NYSIIS)." *Criminology*, **8**(2), 107–128. doi:10.1111/j.1745-9125.1970.tb00734.x.

Strande SM, Cai H, Cooper T, Flammer K, Irving C, von Laszewski G, Majumdar A, Mishin D, Papadopoulos P, Pfeiffer W, Sinkovits RS, Tatineni M, Wagner R, Wang F, Wilkins-Diehr N, Wolter N, Norman ML (2017). "Comet: Tales from the Long Tail: Two Years In and 10,000 Users Later." In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, pp. 38:1–38:7. ACM, New York. doi:10.1145/3093338.3093383.

Taft RL (1970). "Name Search Techniques." *New York State Identification and Intelligence System*, Bureau of Systems Development, Albany.

Teetor P (2011). *R Cookbook*. O'Reilly Media, Inc., Sebastopol, California.

Thompson K (1968). "Programming Techniques: Regular Expression Search Algorithm." *Communications of the ACM*, **11**(6), 419–422. doi:10.1145/363347.363387.

Towns J, Cockerill T, Dahan M, Foster I, Gaither K, Grimshaw A, Hazlewood V, Lathrop S, Lifka D, Peterson GD, *et al.* (2014). "XSEDE: Accelerating Scientific Discovery." *Computing in Science & Engineering*, **16**(5), 62–74. doi:10.1109/mcse.2014.80.

United States Census Bureau (2016). "Frequently Occurring Surnames from the 2010 Census." URL https://www.census.gov/topics/population/genealogy/data/2010_surnames.html.

Van der Loo MPJ (2014). "The **stringdist** Package for Approximate String Matching." *The R Journal*, **6**(1), 111–122. doi:10.32614/rj-2014-011.

Wickham H (2011). "**testthat**: Get Started with Testing." *The R Journal*, **3**(1), 5–10. doi:10.32614/rj-2011-002.

Wright MA (1960). "Mechanizing a Large Index." *The Computer Journal*, **3**(2), 76.

Zobel J, Dart P (1995). "Finding Approximate Matches in Large Lexicons." *Software: Practice and Experience*, **25**(3), 331–345. doi:10.1002/spe.4380250307.

Zobel J, Dart P (1996). "Phonetic String Matching: Lessons From Information Retrieval." In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 166–172. ACM.

Affiliation:

James P. Howard, II
Johns Hopkins Applied Physics Laboratory
11100 Johns Hopkins Road Laurel
Maryland 20723, United States of America
E-mail: jh@jameshoward.us
URL: https://jameshoward.us/

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
October 2020, Volume 95, Issue 8 *Submitted:* 2017-07-26
doi:10.18637/jss.v095.i08 *Accepted:* 2019-06-17