LexFindR: A fast, simple, and extensible R package for finding similar words in a lexicon

ZhaoBin Li[1], Anne Marie Crinnion[2,3], & James S. Magnuson[2,3,4,5]

[1] Department of Mathematics and Statistics, Carleton College, Northfield, MN, USA
[2] Institute for the Brain and Cognitive Sciences, University of Connecticut, Storrs, CT, USA
[3] Department of Psychological Sciences, University of Connecticut, Storrs, CT, USA
[4] BCBL. Basque Center on Cognition Brain and Language, Donostia-San Sebastián, Spain
[5] Ikerbasque. Basque Foundation for Science, Bilbao, Spain

Abstract

Language scientists often need to generate lists of related words, such as potential competitors. They may do this for purposes of experimental control (e.g., selecting items matched on lexical neighborhood but varying in word frequency), or to test theoretical predictions (e.g., hypothesizing that a novel type of competitor may impact word recognition). Several online tools are available, but most are constrained to a fixed lexicon and fixed sets of competitor definitions, and may not give the user full access to or control of source data. We present *LexFindR*, an open source R package that can be easily modified to include additional, novel competitor types. LexFindR is easy to use. Because it can leverage multiple CPU cores and uses vectorized code when possible, it is also extremely fast. In this article, we present an overview of LexFindR usage, illustrated with examples. We also explain the details of how we implemented several standard lexical competitor types used in spoken word recognition research (e.g., cohorts, neighbors, embeddings, rhymes), and show how “lexical dimensions” (e.g., word frequency, word length, uniqueness point) can be integrated into LexFindR workflows (for example, to calculate “frequency weighted competitor probabilities”), for both spoken and visual word recognition research.

Keywords: psycholinguistics; lexicon; word recognition

Corresponding author: James Magnuson (james.magnuson@uconn.edu).

Introduction

Language scientists often need to generate sets of related words or words with specific properties. This might be in service of experimental control (e.g., words matched on length and frequency of occurrences, but differing in lexical *neighborhood*; Luce & Pisoni, 1998). Or the need might arise based on a theoretically-motivated or model-driven hypothesis; perhaps your theory proposes – or your model simulations predict – that shorter words embedded within a word should make that word more difficult to process, so you want to find words with many or few words embedded within them. Sets of related items and their characteristics can also be useful for clinical purposes. For example, frequency-weighted lexical neighborhoods have proven useful for clinical assessments and interventions (e.g., Kirk, Pisoni, & Osberger, 1995; Morrisette & Gierut, 2002; Sommers & Danielson, 1999; Storkel, Bontempo, Aschenbrenner, Maekawa, & Lee, 2013; Storkel, Maekawa, & Hoover, 2010). So how do we generate these lists?

Various excellent tools already exist. For example, three web-based tools are Michael Vitevtich's phonotactic probability (Vitevitch & Luce, 1998, 1999) and neighborhood density calculators (http://www.people.ku.edu/~mvitevit/PhonoProbHome.html), the English Lexicon Project (https://elexicon.wustl.edu/; Balota et al., 2007), and the recent Auditory English Lexicon Project (https://inetapps.nus.edu.sg/aelp; Goh, Yap, & Chee, 2020). Other tools exist for semantic variables or languages other than English, such as *Lexique*, which includes English and French (http://www.lexique.org/; New, Pallier, Brysbaert, & Ferrand, 2004) the multilingual *CLEARPOND* (https://clearpond.northwestern.edu/; Marian, Bartolotti, Chabal, & Shook, 2012), and *EsPal* (https://www.bcbl.eu/databases/espal/; Duchon, Perea, Sebastián-Gallés, Martí, & Carreiras, 2013) for Spanish, but it takes considerable independent work for a researcher to combine these resources with things like neighborhood statistics from the other tools.

Furthermore, while these tools are incredibly useful, they have limitations. Many require using web interfaces, so a researcher's workflow must include interacting with the websites and documenting the steps taken, and importing lists of items into the researcher's local workflow (e.g., into R; R Core Team, 2019). One might argue that this is not a major inconvenience, but other limitations are more severe. For example, so far as we are aware, the computer code used to search lexicons on the sites listed above are not readily available, so a researcher can neither easily confirm the code's validity or extend it (for example, to include a new type of potential competitor). Another limitation is that some tools have a predefined lexicon, and a researcher cannot substitute another in its place. Substituting your own lexicon might be useful if you simply prefer a different lexicon, or if you were using an artificial lexicon, either with human subjects or with a computational model, or if you wanted to examine an understudied language or dialect. Finally, we assume that many labs and researchers have developed and will continue to develop their own code for lexical searches. This duplication of effort is unfortunate. An open-source, extensible tool shared via a version-control repository would allow researchers to collaborate and share their extensions, reducing duplication of effort.

We have developed a lightweight R package, *LexFindR* (Li, Crinnion, & Magnuson, 2020), that addresses these limitations. LexFindR comes with a suite of lexical relation finders for common competitor types used in studies of spoken and/or visual word recognition (e.g., neighbors, cohort [onset] competitors, and rhymes), but is also easily extended to incorporate new definitions. LexFindR is also fast, as it uses R's parallelization capabilities to leverage multiple CPU cores (typically found even on contemporary laptops) and efficient core capabilities of R (e.g., R's *apply* family of functions). Appendix 1 provides an example of how to put together aspects of the examples throughout the paper in order to efficiently gather information about multiple lexical dimensions in one script. In the following sections, we review how to install and use LexFindR. Details about how to share extensions with the community via LexFindR's github repository are provided in Appendix 2.

Using LexFindR

Installing and loading LexFindR

The package is implemented in R and can be utilized like any R package. The package is under review for distribution on the R package repository, CRAN. Once the stable version of the package has been released on CRAN, users will be able to install it using the *Tools::Install Packages* menu in R Studio, or via the following command:

```
install.packages("LexFindR")
```

The current developmental version can be installed from github with the following commands:

```
# uncomment the line below to install devtools if needed
# install.packages("devtools")
# the line below only needs to be run once
devtools::install_github("maglab-uconn/LexFindR")
```

Once installed, the package can be loaded with the following command.

```
library(LexFindR)
```

Getting started

The package comes with two lexicons: the 212-word *slex* lexicon (with only 14 phonemes) from the TRACE model of spoken word recognition (McClelland & Elman, 1986) as a small data set for the user to experiment with, and a larger lexicon (*lemmalex*) that we compiled from various open-access, non-copyrighted materials. The primary source is the SUBTLEX subtitle corpus (Brysbaert & New, 2009), which we cross-referenced with the copyrighted Francis and Kučera (1982) database to reduce the word list to "lemma" (base- or uninflected) forms. Pronunciations were drawn from the larger *CMU Pronouncing Dictionary* (CMU Computer Science, 2020) without lexical stress for both lexicons (with those for slex transcribed by Nenadić and Tucker (2020)). The second lexicon is large enough to demonstrate the full capabilities of the package. The two data sets are automatically loaded when we load LexFindR. We can use the *tidyverse* (Wickham et al., 2019) *glimpse* function to view essential details about the lexicons, and view their first few lines.

```
library(LexFindR)
library(tidyverse) # to use glimpse for previewing R objects

glimpse(slex)
```

```
## Rows: 212
## Columns: 3
## $ Item          <chr> "ad", "ar", "ark", "art", "art^st", "bab", "babi", "b...
## $ Pronunciation <chr> "AA D", "AA R", "AA R K", "AA R T", "AA R T AH S T", ...
## $ Frequency     <int> 53, 4406, 50, 274, 112, 45, 23, 341, 87, 125, 125, 95...
```

```r
glimpse(lemmalex)
```

```
## Rows: 17,750
## Columns: 3
## $ Item          <chr> "a", "abandon", "abandonment", "abate", "abbey", "abb...
## $ Frequency     <dbl> 20415.27, 8.10, 0.96, 0.10, 3.18, 0.84, 0.02, 0.24, 3...
## $ Pronunciation <chr> "AH", "AH B AE N D IH N", "AH B AE N D AH N M AH N T"...
```

Both lexicons are loaded as R dataframes with three fields. “Item” is a label (orthography in the case of *lemmalex*, and transcriptions in the original phonemic conventions used for the TRACE model in the case of *slex*). “Pronunciation” is a space-delimited phonemic transcription using the *ARPAbet* conventions of the CMU Pronouncing Dictionary (ARPAbet transcriptions for TRACE items are from Nenadić & Tucker, 2020). We will discuss shortly how to specify alternative delimters, including a “null” delimiter for working with orthographic forms or pronunciation forms that use one character per phoneme without spaces. “Frequency” is occurrences-per-million words; frequencies are based on Kučera and Francis (1967) for *slex* and on Brysbaert and New (2009) for *lemmalex*.

More information about the lexicons can by queried with the ‘?’ command (we do not present the output here as it is rather extensive):

```r
?slex
?lemmalex
```

Note that you can use *any* lexicon you can load into an R dataframe. You may find it convenient to use the same field names as in *slex* and *lemmalex*, but it is not necessary. For work on phonological word forms, you typically will have both “Item” (usually orthography) and “Pronunciation”, but as we will see later, you can do useful things with LexFindR with any list of forms, including orthographic forms. To use this package with orthographic forms, refer to the section below on *Working with orthography or other “undelimited” forms, or other delimiters.*

LexFindR commands

Table 1 provides a list of LexFindR commands along with brief descriptions. To use any of the LexFindR functions, we provide a target pattern and a word list to compare it to. LexFindR will compare the target pattern to the patterns in the word list to find items that have particular relations to the target. The functions can return indices of items that meet the criteria of the function, but we can also tell LexFindR to return instead the list of matching forms, the list of accompanying labels for matching forms (e.g., spellings), or the frequencies of matching forms. As we progress through examples, we will see when these different options are useful.

Cohorts. Let’s begin with *cohorts*. Cohorts are words that overlap at word onset, and are called “cohorts” because they comprise the set of words predicted to be strongly activated as a spoken word is heard (and thus to form the *recognition cohort*) by the Cohort Model (Marslen-Wilson & Welsh, 1978). While definitions vary, LexFindR is equipped to handle overlap in any number of phonemes. By default, it uses a very common cohort definition: overlap in the first two phonemes. However, it contains a parameter – *overlap* – to allow the researcher to adjust how many initial phonemes must match for two words to be cohorts. We can get the set of cohort indices for a pattern with a command like this for the pronunciation of CAR:

Table 1
LexFindR functions briefly described.

Function	Description
get_cohorts	Returns items that overlap at onset
get_cohortsP	Returns cohorts that are not also neighbors
get_embeds_in_target	Returns items that embed in the target
get_embeds_in_targetP	Returns items that embed in the target that are not also cohorts or neighbors
get_fw	Returns the sum of the log frequencies in a list
get_fwcp	Returns the ratio of the target word's log frequency to the summed log frequencies of all words meeting the competitor definition
get_homoforms	Returns items with the same form as the target
get_neighbors	Returns items that differ by no more than a single deletion, addition, or substitution (can be limited to any combination of deletion, addition, and substitution with the *overlap* parameter)
get_neighborsP	Returns neighbors that are not also cohorts or rhymes
get_nohorts	Returns items that meet the definitions for both cohorts and neighbors
get_rhymes	Returns items that mismatch at word onset by no more than a specified number of elements
get_target_embeds_in	Returns items that the target embeds within
get_target_embeds_inP	Returns items that the target embeds within that are not also cohorts or neighbors
get_uniqpt	Returns the position at which the target becomes a unique completion in the lexicon (or word length + 1 if the word is not unique at offset)

```
get_cohorts("K AA R", slex$Pronunciation)
```

```
## [1] 66 67 68 69 70 71
```

This tells us that *slex* entries 66-71 are cohorts of CAR (overlapping in at least the initial two positions, since 2 is the default overlap). To get the competitors themselves rather than the indices, we could specify that we want *forms*:

```
get_cohorts("K AA R", slex$Pronunciation, form = TRUE)
```

```
## [1] "K AA L IY G"   "K AA P"        "K AA P IY"     "K AA R"
## [5] "K AA R D"      "K AA R P AH T"
```

To see the labels of those items (in TRACE's phonemic transcriptions), we can use standard R conventions (and should see the phonemic transcriptions for COLLEAGUE, COP, COPY, CAR, CARD, and CARPET):

```
slex[get_cohorts("K AA R", slex$Pronunciation), ]$Item
```

```
## [1] "kalig"  "kap"    "kapi"   "kar"    "kard"   "karp^t"
```

Alternatively, we could request the *count* of cohorts (going back to the default overlap = 2):

```r
get_cohorts("K AA R", slex$Pronunciation, count = TRUE)
```

```
## [1] 6
```

That is not a large number of cohorts. Let's compare it to the count we get from *lemmalex*:

```r
get_cohorts("K AA R", lemmalex$Pronunciation, count = TRUE)
```

```
## [1] 272
```

As expected, we get many more from a more realistically-sized lexicon. Note that most LexFindR functions have exactly the same structure, returning indices by default, but with options to return forms or counts.

Finally, let's see how we can change the cohort definition in terms of how many phonemes must match. Let's say we want to try a definition of cohorts with overlap in the first three phonemes for the cohort of CARD:

```r
get_cohorts("K AA R D", slex$Pronunciation, form = TRUE, overlap = 3)
```

```
## [1] "K AA R"         "K AA R D"      "K AA R P AH T"
```

We could repeat any of the preceding example commands with 3-phoneme overlap by simply adding "overlap = 3" to each command.

Neighborhood. *Neighbors* are another possible competitor often considered in word recognition research. The standard *neighbor* definition for spoken words comes from the Neighborhood Activation Model (NAM; Luce & Pisoni, 1998). While NAM includes a graded similarity rule, most often, researchers use the simpler *DAS rule*: two words are considered neighbors (and are expected to be strongly activated if either one is heard) if they differ by no more than a single phonemic *d*eletion, *a*ddition, or *s*ubstitution. For example, CAR (/kar/) has many neighbors, including the deletion neighbor ARE (note that neighbors are based on pronunciation here, not spelling), addition neighbors SCAR and CARD, and substitution neighbors at every position, such as BAR, CORE, and COP (though as we will see, CAR has no medial [vowel] substitution neighbors in *slex*). Let's look at CAR's neighbors in *slex*, using analogous commands to those we used for cohorts.

```r
# get indices
get_neighbors("K AA R", slex$Pronunciation)
```

```
## [1]   2  10  67  69  70 104 152 184
```

```r
# get forms
get_neighbors("K AA R", slex$Pronunciation, form = TRUE)
```

```
## [1] "AA R"      "B AA R"    "K AA P"    "K AA R"    "K AA R D" "P AA R"    "S K AA R"
## [8] "T AA R"
```

```r
# get labels
slex[get_neighbors("K AA R", slex$Pronunciation), ]$Item
```

```
## [1] "ar"   "bar"  "kap"  "kar"  "kard" "par"  "skar" "tar"
```

```r
# get count
get_neighbors("K AA R", slex$Pronunciation, count = TRUE)
```

```
## [1] 8
```

Note that in visual word recognition, it is much more common to consider only substitution neighbors (often called "Coltheart's N"; Coltheart, Davelaar, Jonasson, & Besner, 1977). So if you are working with orthography, you may only want substitution neighbors. Or perhaps you would like to explore the relative impact of deletion, addition, and substitution neighbors. LexFindR's *get_neighbors* function anticipates the potential need for such flexibility. By default, it assumes you want all three, but you can specify any single type or any combination with the *neighbors* argument and specifying deletion neighbors with "d", addition neighbors with "a", and/or substitution neighbors with "s". Here are some examples:

```r
# get forms of deletion neighbors (just ARE)
get_neighbors("K AA R", slex$Pronunciation, form = TRUE, neighbors = "d")
```

```
## [1] "AA R"
```

```r
# get forms of addition neighbors (CARD, SCAR)
get_neighbors("K AA R", slex$Pronunciation, form = TRUE, neighbors = "a")
```

```
## [1] "K AA R D" "S K AA R"
```

```r
# get forms of substitution neighbors (BAR, COP, CAR, PAR, TAR)
get_neighbors("K AA R", slex$Pronunciation, form = TRUE, neighbors = "s")
```

```
## [1] "B AA R" "K AA P" "K AA R" "P AA R" "T AA R"
```

```r
# get forms of deletion (ARE) and addition (CARD, SCAR) neighbors
get_neighbors("K AA R", slex$Pronunciation, form = TRUE, neighbors = "ad")
```

```
## [1] "AA R"     "K AA R D" "S K AA R"
```

Of course, we can easily do other things using basic R commands, such as determine what proportion of CAR's neighbors are substitution neighbors:

```r
# what proportion of CAR's neighbors are substitution neighbors?
get_neighbors("K AA R", slex$Pronunciation, neighbors = "s", count = TRUE) /
  get_neighbors("K AA R", slex$Pronunciation, count = TRUE)
```

```
## [1] 0.625
```

Other competitor types. In addition to cohorts and neighbors, LexFindR comes with analogous functions for several other similarity types.

- *get_rhymes*: returns items that mismatch at word onset by no more than a specified number of phonemes, using a *mismatch* argument which the user can supply. The default *mismatch* argument is 1 phoneme, meaning the function will by default return items that mismatch at word onset by a maximum of 1 phoneme (so not a standard definition of poetic rhyme or phonological rime). With this default argument, rhymes will include items that are addition or deletion neighbors at first position (e.g., CAR's rhymes will include ARE and SCAR) as well as substitution neighbors at position 1 (e.g., BAR, TAR). If mismatch were set to 2, for example, CAR would additionally match any 3-phoneme word ending in /r/ and any 4-phoneme word ending in /ar/.

- *get_embeds_in_target*: returns items that are embedded within a target word. For SCAR, this would include ARE and CAR.
- *get_target_embeds_in*: returns items that the target embeds within. For CAR, this would include SCAR and CARD.
- *get_homoforms*: returns items with the same form as the target. We use "homo*form*" because these would be homophones for phonological forms but homonyms for orthographic forms.

LexFindR also anticipates the possibility that a researcher may want to find competitor types that do not overlap. For example, CARD is both a cohort and a neighbor of CAR, so which set should it appear in? We propose a novel category called *nohorts* – neighbors that are also cohorts – and provide "P" (pure) versions of several competitor-type functions that return non-overlapping sets.

- *get_nohorts*: Cohorts and neighbors are overlapping sets, although not all cohorts are neighbors (e.g., CAR and CARPET are cohorts but not neighbors) and not all neighbors are cohorts. *Nohorts* are the intersection of cohorts and neighbors. Note that the target word will be part of the *nohort* set, and not part of *cohortsP* or *neighborsP*, which we define next.
- *get_cohortsP*: the set of "pure" cohorts that are not also neighbors.
- *get_neighborsP*: the set of "pure" neighbors that are not also cohorts *or* rhymes.
- *get_embeds_in_targetP*: set of items that embed in the target that are not also cohorts or neighbors.
- *get_target_embeds_inP*: set of items that the target embeds in that are not also cohorts or neighbors.

The *nohort* and "P" functions use the base-R *intersect* and *setdiff* functions to find set intersections and differences. To see the code for any function in R, you can simply enter the function name with no arguments and no following parentheses. Let's look at the code for *get_nohorts*. Many of the details provided may not be useful for a typical user, but the *intersect* command is the interesting part of this example.

```
get_nohorts
```

```
## function(target, lexicon, neighbors = "das", sep = " ", form = FALSE, count = FALSE) {
##   idx <- intersect(
##     get_cohorts(target, lexicon, sep, form = FALSE, count = FALSE),
##     get_neighbors(target, lexicon, neighbors, sep, form = FALSE, count = FALSE)
##   )
##
##   get_return(idx, lexicon, form, count)
## }
## <bytecode: 0x7ff36ae4dfa0>
## <environment: namespace:LexFindR>
```

Now let's examine the *get_neighborsP* function to see how the *setdiff* is used to find "pure" sets.

```
get_neighborsP
```

```
## function(target, lexicon, neighbors = "das", sep = " ", form = FALSE, count = FALSE) {
##   idx <- setdiff(
##     setdiff(
##       get_neighbors(target, lexicon, neighbors),
##       get_cohorts(target, lexicon, sep, form = FALSE, count = FALSE)
##     ),
##     get_rhymes(target, lexicon, sep, form = FALSE, count = FALSE)
##   )
##
##   get_return(idx, lexicon, form, count)
## }
## <bytecode: 0x7ff38de457a0>
## <environment: namespace:LexFindR>
```

This function uses nested *setdiff* calls to first find neighbors excluding cohorts and then to exclude rhymes from that set. A user could use these functions as examples to create their own specific subsets of items.

Form length. You may wish to calculate form length. This is easy to do with base R. If you use CMU pronunciations, as in *lemmalex*, we can use a technique for counting words separated by whitespace with the *lengths* command in R.

```r
# get lengths by splitting on spaces
lemmalex$Length <- lengths(strsplit(lemmalex$Pronunciation, " "))

glimpse(lemmalex)
```

```
## Rows: 17,750
## Columns: 4
## $ Item          <chr> "a", "abandon", "abandonment", "abate", "abbey", "abb...
## $ Frequency     <dbl> 20415.27, 8.10, 0.96, 0.10, 3.18, 0.84, 0.02, 0.24, 3...
## $ Pronunciation <chr> "AH", "AH B AE N D IH N", "AH B AE N D AH N M AH N T"...
## $ Length        <int> 1, 7, 11, 4, 3, 4, 8, 10, 7, 9, 8, 7, 8, 4, 6, 5, 8, ...
```

If you have a null-delimited form, where each character is a single letter or phoneme, we can use the *nchar* function.

```r
# get lengths by counting characters for orthography or 1-char per phoneme forms
slex$Length <- nchar(slex$Item)

glimpse(slex)
```

```
## Rows: 212
## Columns: 4
## $ Item          <chr> "ad", "ar", "ark", "art", "art^st", "bab", "babi", "b...
## $ Pronunciation <chr> "AA D", "AA R", "AA R K", "AA R T", "AA R T AH S T", ...
```

```
## $ Frequency     <int> 53, 4406, 50, 274, 112, 45, 23, 341, 87, 125, 125, 95...
## $ Length        <int> 2, 2, 3, 3, 6, 3, 4, 4, 4, 3, 4, 5, 2, 4, 3, 4, 3, 4,...
```

Uniqueness point. We have added one other common lexical dimension to the LexFindR functions (*get_uniqpt*), which is the uniqueness point (UP) of a form. This is the position at which an item becomes the only completion in the lexicon. For example, in *slex*, /kard/ (CARD) becomes unique at position 4, as does /karp^t/ (CARPET). SCAR becomes unique at position 3. CAR (/kar/) is not unique at its final position, so its uniqueness point is set to its length *plus one*.

```r
get_uniqpt("K AA R", slex$Pronunciation)
```

```
## [1] 4
```

```r
get_uniqpt("S K AA R", slex$Pronunciation)
```

```
## [1] 3
```

Again, CAR is not unique by word offset, so its UP is its length plus one. SCAR becomes unique at position 3, one before its offset. Let's consider some additional useful steps. We could normalize UPs by dividing them by word length *plus one*, the maximal possible score. So CARD would have a normalized UP of 0.8 (4/5), while CARPET's would be 0.57 (4/7), and CAR's would be 1.0 (4/4). Here are some examples.

```r
# Get UPs for all items in slex
slex$UP <- unlist(lapply(slex$Pronunciation,
  FUN = get_uniqpt, lexicon = slex$Pronunciation
))

# Now let's normalize UP by word length + 1
slex$UP.norm <- slex$UP / (slex$Length + 1)

# Check examples
subset(slex, Item == "kar" | Item == "skar" | Item == "kard" | Item == "karp^t")
```

```
##        Item Pronunciation Frequency Length UP   UP.norm
## 69      kar        K AA R       386      3  4 1.0000000
## 70     kard      K AA R D        62      4  4 0.8000000
## 71   karp^t K AA R P AH T        22      6  4 0.5714286
## 152    skar      S K AA R        22      4  3 0.6000000
```

Helper functions

LexFindR includes two helper functions that can be applied to the output of other functions: *get_fw* and *get_fwcp*.

Log frequency weights: *get_fw*. Intuitively, the number (count) of potential competitors may be important. But some competitors might have more influence than others; in particular, words with higher frequency-of-occurrence may compete more strongly. So we may wish to consider the frequencies of competitors. We can use the indices returned by functions like *get_cohorts* or *get_neighbors* to get the frequencies of the items. Let's do this for the word CAR in *slex* and *lemmalex* and get some summary statistics.

```r
# get CAR's slex cohorts' frequencies
slex_cohort_frequencies <- slex$Frequency[
  get_neighbors("K AA R", slex$Pronunciation)
]
summary(slex_cohort_frequencies)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    10.0    21.5    47.0   632.9   190.2  4406.0
```

```r
# get CAR's lemmalex cohorts' frequencies
llex_cohort_frequencies <- lemmalex$Frequency[
  get_neighbors("K AA R", lemmalex$Pronunciation)
]
summary(llex_cohort_frequencies)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.220   1.353   6.635  58.336  30.830 485.250
```

Typically, frequencies are log scaled, as this provides a better fit when they are used to predict human behavior (e.g., word recognition time). It would be useful, therefore, to weight the count of competitors by log frequencies. The LexFindR helper function *get_fw* does this. You supply it with a list of frequencies, and it takes their logs and returns the sum. This is simple enough that you could do it with basic R functions yourself. However, *get_fw* provides some useful error checking. Specifically, it checks whether the minimum frequency in your set of frequencies is less than one, since taking the log would return a negative value. If so, it also suggests a minimum constant to specify for *pad* to add to each frequency before taking the log. Let's consider how we might use this. First, let's try using *get_fw* to give us summed log frequencies for the frequencies we collected above for CAR's *slex* cohorts.

```r
get_fw(slex_cohort_frequencies)
```

```
## [1] 35.1571
```

This gives us the sum without any problem, as the minimum frequency in *slex_cohort_frequencies* is greater than 1. Now let's try with *llex_cohort_frequencies*.

```r
get_fw(llex_cohort_frequencies)
```

```
## Warning: `min(competitors_freq) + pad` is 0.22 which is < 1;
## * Consider adding pad >= 0.78
```

```
## [1] 55.64038
```

Now we get a value (55.64038) but also a warning because the minimum value is less than 1. So let's add the *pad* option. Using 1 will bring our minimum to a value greater than 1, avoiding results with non-positive values.

```r
get_fw(llex_cohort_frequencies, pad = 1)
```

```
## [1] 65.67193
```

Log *Frequency Weighted Competitor Probabilities*: *get_fwcp*. We could go a step beyond frequency weights and calculate the *Frequency Weighted Competitor Probability (FWCP)* of a word, inspired by the Neighborhood Activation Model's *Frequency-Weighted Neighborhood Probability* (FWNP; Luce & Pisoni, 1998). This is calculated as the ratio of the target word's log frequency to the sum of all words meeting the competitor definition, as in the following equation.

$$FWCP = \frac{log(Frequency_{target})}{\sum_{c \in competitors} log(Frequency_c)}$$

Notably, on most competitor definitions, this includes the target word itself, so we can think of the ratio as expressing what proportion of the "frequency weight" of the target's competitors is contributed by the target itself. For spoken words, the larger the ratio, the more easily the target word tends to be recognized. To calculate this with LexFindR, we supply a set of competitor frequencies and the target word's frequency to the *get_fwcp* function. Note that we can include a *pad* option as for *get_fw*, and it will be applied to both the target word's frequency and the list of competitor frequencies; again, this should be done if the minimum frequency value is less than 1. Let's verify that the minimum frequency in *slex* is greater than 1.

```
# check the minimum frequency
min(slex$Frequency)
```

```
## [1] 10
```

The next two code blocks demonstrate how to get the FWCP for neighbors (i.e., the FWNP) and then for cohorts.

```
# because get_neighbors returns indices by default, we can use its output as
# the keys to get corresponding frequencies from another column in the
# dataframe
competitors_freq <- slex$Frequency[get_neighbors("K AA R", slex$Pronunciation)]
target_freq <- slex$Frequency[which(slex$Pronunciation == "K AA R")]

# now we can the FWCP based on neighbors;
# minimum frequency is > 1 so we won't specify a pad
get_fwcp(target_freq, competitors_freq)
```

```
## [1] 0.1694064
```

```
# Now let's get the FWCP for cohorts
competitors_freq <- slex$Frequency[get_cohorts("K AA R", slex$Pronunciation)]
target_freq <- slex$Frequency[which(slex$Pronunciation == "K AA R")]

get_fwcp(target_freq, competitors_freq)
```

```
## [1] 0.2459427
```

Note that *get_fwcp* is not simply computing the ratio of target-to-competitor frequencies; it is first converting the frequencies to log frequencies. If your lexicon file has frequencies already in log form, you should not use the *get_fwcp* function, but instead you should calculate the ratios directly. Also note that it is fairly standard to express frequencies as occurrences-per-million. If your basis is different (e.g., occurrences-per-six million), you may want to transform your frequencies to the more standard per-million basis. Finally, we recommend that you examine distributions before using the results of *get_fwcp*, as these often exhibit difficult-to-mitigate deviations from normality. One may be better served by examining target frequencies and competitor frequency weights (obtained with *get_fw*) separately.

Working with orthography or other "undelimited" forms, or other delimiters

By default, LexFindR functions expect the forms you supply to be space-delimited, which is the typical convention for CMU pronunciations. Using a delimiter allows you to have form codes (typically phoneme codes) made up of more than one character. But what if you want to work with orthography, or a phoneme code that uses one character per phoneme without delimiters? You can simply specify *sep = ""* to indicate that your forms have a "null" delimiter. We can illustrate this with the orthography in the "Item" field in *lemmalex*.

```
# Let's list orthographic substitution neighbors for CAR in lemmalex
get_neighbors("car", lemmalex$Item, form = TRUE, neighbor = "s", sep = "")
```

```
##  [1] "bar" "cab" "cam" "can" "cap" "car" "cat" "caw" "cur" "ear" "far" "jar"
## [13] "mar" "par" "tar" "war"
```

Now let's try it with TRACE's original phoneme encodings, which use one character per phoneme. Those original forms are in the "Item" field of *slex*:

```
# Let's list orthographic substitution neighbors for CAR in slex
get_neighbors("kar", slex$Item, form = TRUE, neighbor = "s", sep = "")
```

```
## [1] "bar" "kap" "kar" "par" "tar"
```

Batch processing with target list and lexicon

Often, we may need to get the competitors for each word in the lexicon, with respect to the entire lexicon. This would be a prerequisite for selecting words with relatively many vs. few neighbors, for example. One way to do this would be to use the base *R* function *lapply*. Here is how we could do this for cohorts. The final *glimpse* command will show us the first few instances of each field.

```
# reset R
rm(list = ls())
library(LexFindR)

# define the lexicon with the list of target words to compute
# cohorts for; we will use *target_df* instead of modifying
# slex or lemmalex directly
target_df <- slex

# specify the reference lexicon; here it is actually the list
# of pronunciations from slex, as we want to find all cohorts
# for all words in our lexicon. It is not necessary to create
# a new dataframe, but because we find it useful for more
# complex tasks, we use this approach here
lexicon_df <- target_df

# this instruction will create a new column in our target_df
# dataframe, "cohort_idx", which will be the list of lexicon_df
# indices corresponding to each word's cohort set
target_df$cohort_idx <-
```

```
  lapply(
    # in each lapply instance, select the target pronunciation
    target_df$Pronunciation,
    # in each lapply instance, apply the get_cohorts function
    FUN = get_cohorts,
    # in each lapply instance, compare the current target
    # Pronunciation to each lexicon Pronunciation
    lexicon = lexicon_df$Pronunciation
  )

# let's look at the first few instances in each field...
glimpse(target_df)
```

```
## Rows: 212
## Columns: 4
## $ Item          <chr> "ad", "ar", "ark", "art", "art^st", "bab", "babi", "b...
## $ Pronunciation <chr> "AA D", "AA R", "AA R K", "AA R T", "AA R T AH S T", ...
## $ Frequency     <int> 53, 4406, 50, 274, 112, 45, 23, 341, 87, 125, 125, 95...
## $ cohort_idx    <list> [1, <2, 3, 4, 5>, <2, 3, 4, 5>, <2, 3, 4, 5>, <2, 3,...
```

Consider the *cohort_idx* field. We can see that /ad/ (ODD) has only one cohort (itself), while /ar/ (ARE) has four (items 2, 3, 4, 5, or /ar/, /ark/, /art/, and /art^st/, i.e., ARE, ARK, ART, ARTIST).

What if we also want the lists of cohort forms or labels and frequencies? Rather than calling the function three times, we could speed up the process (speed will be very important when we work with large lexicons!) by calling *get_cohorts* only once, and then using the indices to get the other items we want. In the next example, we keep working with *target_df* and its new field *cohort_idx* (which has the list of indices [row counts] of records that meet the cohort definition for each target).

```
# continuing the code block above,
# this instruction creates a new field, cohort_str, which will
# be the list of forms corresponding to the list of indices
# in cohort_idx
target_df$cohort_str <-
  lapply(
    # on each instance of lapply (each target word), we apply
    # this simple function of returning the Item (label) for
    # each cohort index (idx)
    target_df$cohort_idx, function(idx) {
      lexicon_df$Item[idx]
    }
  )

# to create a list of frequencies for each cohort of a
# target item, we do the same thing, but now we get the
# Frequency rather than the Item
target_df$cohort_freq <-
  lapply(
    target_df$cohort_idx, function(idx) {
      lexicon_df$Frequency[idx]
```

```
    }
  )

# to get the count of cohorts for each item, we *could* run
# get_cohorts again with "count = TRUE", but we can use the
# "lengths" command to get the count of items in cohort_str
# (or cohort_idx) instead. We'll put the result in a new
# field in the dataframe called "cohort_count"
target_df$cohort_count <- lengths(target_df$cohort_str)

# finally, we can get the cohort frequency weight for each
# word (the summed log frequencies of all its cohorts)
target_df$cohort_fw <- lapply(target_df$cohort_freq, get_fw)
```

Let's look at the results:

```
glimpse(target_df)
```

```
## Rows: 212
## Columns: 8
## $ Item          <chr> "ad", "ar", "ark", "art", "art^st", "bab", "babi", "b...
## $ Pronunciation <chr> "AA D", "AA R", "AA R K", "AA R T", "AA R T AH S T", ...
## $ Frequency     <int> 53, 4406, 50, 274, 112, 45, 23, 341, 87, 125, 125, 95...
## $ cohort_idx    <list> [1, <2, 3, 4, 5>, <2, 3, 4, 5>, <2, 3, 4, 5>, <2, 3,...
## $ cohort_str    <list> ["ad", <"ar", "ark", "art", "art^st">, <"ar", "ark",...
## $ cohort_freq   <list> [53, <4406, 50, 274, 112>, <4406, 50, 274, 112>, <44...
## $ cohort_count  <int> 1, 4, 4, 4, 4, 7, 7, 7, 7, 7, 7, 7, 3, 3, 3, 3, 3, 3,...
## $ cohort_fw     <list> [3.970292, 22.63437, 22.63437, 22.63437, 22.63437, 3...
```

We can see that *cohort_idx*, *cohort_str*, and *cohort_freq* all contain lists, and we can verify that for a given word, the lists are the same length (e.g., one frequency form for each cohort). There should only be one value per target word in *cohort_count* and *cohort_fw*, which we can see is the case as well.

Working with different target and lexicon lists. In some cases, you may only want to get details for a subset of items in the lexicon – or even for a list of forms that are *not* in the lexicon. In these cases, you can simply specify a shorter target list rather than making the target list and lexicon the same. Note that of course, if you do not have frequencies for your items, you will not be able to use the *get_fwcp* command. As an example, we might want to examine what the neighborhoods of the words in the TRACE lexicon would be in the context of a realistically-sized lexicon. We can do this by using *slex* as our target list and *lemmalex* as our lexicon.

```
# Again, it is not necessary to copy slex and lemmalex to target_df and
# lexicon_df, but doing so can promote clarity in more complex workflows
target_df <- slex
lexicon_df <- lemmalex

# first, *lapply* get_cohorts
target_df$cohort_idx <-
  lapply(
```

```r
    target_df$Pronunciation,
    FUN = get_cohorts,
    lexicon = lexicon_df$Pronunciation
  )

# let's also get cohort counts
target_df$cohort_count <- lengths(target_df$cohort_idx)

glimpse(target_df)
```

```
## Rows: 212
## Columns: 5
## $ Item          <chr> "ad", "ar", "ark", "art", "art^st", "bab", "babi", "b...
## $ Pronunciation <chr> "AA D", "AA R", "AA R K", "AA R T", "AA R T AH S T", ...
## $ Frequency     <int> 53, 4406, 50, 274, 112, 45, 23, 341, 87, 125, 125, 95...
## $ cohort_idx    <list> [<10577, 10578, 10579, 10582>, <762, 763, 764, 765, ...
## $ cohort_count  <int> 4, 69, 69, 69, 69, 64, 64, 64, 64, 64, 64, 64, 32, 32...
```

Comparing this to our earlier results, we see that ODD would have 4 cohorts in *lemmalex* instead of 1 within *slex*.

Parallelizing for speed

If we are getting competitors for every word in a lexicon, speed becomes a concern, especially if we want to do this for many competitor types. To quantify this, let's time how long it takes to calculate cohorts for all words in *lemmalex*. We will use the R *tictoc* package (Izrailev, 2014) to time the process. For this demonstration, we are using a MacBook Pro with an Intel Core i9 CPU and 32gb of RAM.

```r
# load functions for timing
library(tictoc)

# set targets and lexicon to be the large lemmalex lexicon
target_df <- lemmalex
lexicon_df <- target_df

# start the timer
tic("get_cohorts without parallelization")

# lapply the get_cohorts function -- fast, vectorized, but not parallel
# warning: this could take a long time, depending on your hardware
target_df$cohort_idx <-
  lapply(
    target_df$Pronunciation,
    FUN = get_cohorts,
    lexicon = lexicon_df$Pronunciation
  )
toc()
```

```
## get_cohorts without parallelization: 110.576 sec elapsed
```

```r
tic("get additional fields")
# get cohort strings
target_df$cohort_str <- lapply(
  target_df$cohort_idx, function(idx) {
    lexicon_df$Item[idx]
  }
)

# get cohort counts
target_df$cohort_count <- lengths(target_df$cohort_str)

toc()
```

```
## get additional fields: 0.05 sec elapsed
```

```r
glimpse(target_df)
```

```
## Rows: 17,750
## Columns: 6
## $ Item          <chr> "a", "abandon", "abandonment", "abate", "abbey", "abb...
## $ Frequency     <dbl> 20415.27, 8.10, 0.96, 0.10, 3.18, 0.84, 0.02, 0.24, 3...
## $ Pronunciation <chr> "AH", "AH B AE N D IH N", "AH B AE N D AH N M AH N T"...
## $ cohort_idx    <list> [<>, <2, 3, 4, 7, 8, 14, 15, 16, 18, 19, 29, 30, 31,...
## $ cohort_str    <list> [<>, <"abandon", "abandonment", "abate", "abbreviate...
## $ cohort_count  <int> 0, 61, 61, 61, 39, 39, 61, 61, 39, 39, 39, 39, 39, 61...
```

On our demonstration laptop, *get_cohorts* with *lapply* took ~111 seconds (on an older workstation we tested, it took several minutes). If you only have to do this once, that may be tolerable. But we can do better! We could easily parallelize using the R *future* package, and its commands like *future.apply* (Bengtsson, 2013). There are various ways to engage multiple cores with this package, as detailed in its documentation. The *plan(multisession, workers = num_cores)* is quite convenient, and works on Windows, Macintosh, and Linux with Rstudio and base R. In the following code block, we show how to load *future.apply* and set things up to use multiple cores.

```r
# uncomment the line below to install, but you only need
# to do this once.
# install.packages("future.apply")
library(future.apply)

# how many cores do we have?
num_cores <- availableCores()
print(paste0("Using num_cores: ", num_cores))
```

```
## [1] "Using num_cores: 12"
```

```r
# now let future.apply figure out how to optimize parallel
# division of labor over cores
plan(multisession, workers = num_cores)
```

With this setup, the only thing left to do is to replace our *apply* functions with their *future.apply* equivalents. In the example below, we just replace *lapply* with *future_lapply* to parallelize the function that gets competitors (there's no real need to do this with the other *apply* call as it is not the bottleneck; in fact, it is so poorly suited for parallelization that it is slowed by a factor of ~10 if we do use *future_apply*).

```r
# load functions for timing
library(tictoc)

# set targets and lexicon to be the large lemmalex lexicon
target_df <- lemmalex
lexicon_df <- target_df

# start the timer
tic("get_cohorts WITH parallelization")

# lapply the get_cohorts function -- fast, vectorized, but not parallel
# warning: this could take a long time, depending on your hardware
target_df$cohort_idx <-
  future_lapply(
    target_df$Pronunciation,
    FUN = get_cohorts,
    lexicon = lexicon_df$Pronunciation
  )
toc()
```

```
## get_cohorts WITH parallelization: 34.531 sec elapsed
```

```r
# get cohort strings
target_df$cohort_str <- lapply(
  target_df$cohort_idx, function(idx) {
    lexicon_df$Item[idx]
  }
)

target_df$cohort_count <- lengths(target_df$cohort_str)

toc()

glimpse(target_df)
```

```
## Rows: 17,750
## Columns: 6
## $ Item          <chr> "a", "abandon", "abandonment", "abate", "abbey", "abb...
## $ Frequency     <dbl> 20415.27, 8.10, 0.96, 0.10, 3.18, 0.84, 0.02, 0.24, 3...
## $ Pronunciation <chr> "AH", "AH B AE N D IH N", "AH B AE N D AH N M AH N T"...
## $ cohort_idx    <list> [<>, <2, 3, 4, 7, 8, 14, 15, 16, 18, 19, 29, 30, 31,...
## $ cohort_str    <list> [<>, <"abandon", "abandonment", "abate", "abbreviate...
## $ cohort_count  <int> 0, 61, 61, 61, 39, 39, 61, 61, 39, 39, 39, 39, 39, 61...
```

We see an improvement from 111 seconds to approximately 35; it took a bit more than 3 times longer without parallelization. On the older workstation, the improvement was more dramatic, from several minutes to around 35 seconds (around 10 times faster with parallelization). Again, such differences may not seem important if you are running a search once, but if you want to do many different kinds of searches, or explore novel similarity definitions, speed will become important. In Appendix 1, we present an example of parallelized code for conducting several LexFindR competitor searches in series.

Conclusions

LexFindR fills important gaps in the language scientist's toolkit. It provides a free, fast, extensible, tested, and readily shared tool that can be integrated into typical analysis workflow within R. Researchers inclined to contribute extensions to LexFindR should refer to Appendix 2 for basic guidance on how to do so. We hope our fellow researchers will find LexFindR useful.

Author contributions

ZL and JM conceptualized the project; ZL wrote most code and drafted most of this manuscript; AMC contributed significant documentation to the LexFindR package and contributed to the writing and editing of the full manuscript; JM advised on and contributed to code and writing, and contributed to and edited the full manuscript.

Acknowledgments

This work was supported in part by U.S. National Science Foundation grants PAC 1754284 (JM, PI) and IGE NRT 1747486 (JM, PI). The authors are solely responsible for the content of this article. This work was also supported in part by the Basque Government through the BERC 2018-2021 program, and by the Agencia Estatal de Investigación through BCBL Severo Ochoa excellence accreditation SEV-2015-0490.

Open Practices Statement

All materials, including computer code, related to this manuscript are available publicly at the associated github repository (https://github.com/maglab-uconn/LexFindR). The package itself is released as open-source software.

Appendix 1: Extended example – Getting several competitor types

This example shows how you can go through several competitor types for a lexicon, adding columns for the indices, labels, frequencies, counts, frequency weights, and FWCP for each competitor type. For an example implemented in *tidyverse* (Wickham et al., 2019) piping style, see the package vignettes for LexFindR.

```
library(LexFindR)
library(tidyverse) # for glimpse
library(future.apply) # parallelization
library(tictoc) # timing utilities

# In this example, we define a dataframe source for target words
# (target_df) and another for the lexicon to compare the target
# words to (lexicon_df). Often, these will be the same, but we keep
# them separate here to make it easier for others to generalize from
# this example code.

# Code assumes you have at least 3 columns in target_df & lexicon_df:
# 1. Item -- a label of some sort, can be identical to Pronunciation
# 2. Pronunciation -- typically a phonological form
# 3. Frequency -- should be in occurrences per million, or some other
#                 raw form, as the functions below take the log of
#                 the frequency form. See advice about padding in
#                 the main article text.
#
# Of course, you can name your fields as you like, and edit the
# field names below appropriately.
target_df <- slex
lexicon_df <- target_df

# Prepare for parallelizing
# 1. how many cores do we have?
num_cores <- availableCores()
print(paste0("Using num_cores: ", num_cores))
```

```
## [1] "Using num_cores: 12"
```

```
# 2. now let future.apply figure out how to optimize parallel
#    division of labor over cores
plan(multisession, workers = num_cores)

# the functions in this list all return lists of word indices; the
# uniqueness point function is not included because it returns a
# single value per word.
fun_list <- c(
  "cohorts", "neighbors",
  "rhymes", "homoforms",
  "target_embeds_in", "embeds_in_target",
  "nohorts", "cohortsP", "neighborsP",
  "target_embeds_inP", "embeds_in_targetP"
)
```

```
# we need to keep track of the P variants, as we need to tell get_fwcp
# to add in the target frequency for these, as they exclude the target
Ps <- c(
  "cohortsP", "neighborsP", "target_embeds_inP",
  "embeds_in_targetP"
)

# determine how much to pad based on minimum frequency
if (min(target_df$Frequency) == 0) {
  pad <- 2
} else if (min(target_df$Frequency) < 1) {
  pad <- 1
} else {
  pad <- 0
}

# now let's loop through the functions
for (fun_name in fun_list) {
  # start timer for this function
  tic(fun_name)

  # the P functions do not include the target in the denominator for
  # get_fwcp; if we want this to be a consistent ratio, we need to
  # add target frequency to the denominator
  add_target <- FALSE
  if (fun_name %in% Ps) {
    add_target <- TRUE
  }

  # inform the user that we are starting the next function, make sure
  # we are correctly adding target or not
  cat("Starting", fun_name, " -- add_target = ", add_target)
  func <- paste0("get_", fun_name)

  # use *future_lapply* to do the competitor search, creating
  # a new column in *target_df* that will be this function's
  # name + _idx (e.g., cohort_idx)
  target_df[[paste0(fun_name, "_idx")]] <-
    future_lapply(target_df$Pronunciation,
      FUN = get(func),
      lexicon = lexicon_df$Pronunciation
    )

  # list the competitor form labels in functionname_str
  target_df[[paste0(fun_name, "_str")]] <- lapply(
    target_df[[paste0(fun_name, "_idx")]],
    function(idx) {
      lexicon_df$Item[idx]
    }
  )
```

```r
  # list the competitor frequencies in functionname_freq
  target_df[[paste0(fun_name, "_freq")]] <- lapply(
    target_df[[paste0(fun_name, "_idx")]],
    function(idx) {
      lexicon_df$Frequency[idx]
    }
  )

  # put the count of competitors in functionname_num
  target_df[[paste0(fun_name, "_num")]] <-
    lengths(target_df[[paste0(fun_name, "_idx")]])

  # put the FW in functionname_fwt using the "mapply" function
  # to input multiple arguments to the get_fw function.
  # using "lapply" would require a helper function
  target_df[[paste0(fun_name, "_fwt")]] <-
    mapply(get_fw,
      competitors_freq = target_df[[paste0(fun_name, "_freq")]],
      pad = pad
    )

  # put the FWCP in functionname_fwcp
  target_df[[paste0(fun_name, "_fwcp")]] <-
    mapply(get_fwcp,
      target_freq = target_df$Frequency,
      competitors_freq = target_df[[paste0(fun_name, "_freq")]],
      pad = pad, add_target = add_target
    )

  toc()
}
```

```
## Starting cohorts  -- add_target =  FALSEcohorts: 0.212 sec elapsed
## Starting neighbors  -- add_target =  FALSEneighbors: 0.15 sec elapsed
## Starting rhymes  -- add_target =  FALSErhymes: 0.141 sec elapsed
## Starting homoforms  -- add_target =  FALSEhomoforms: 0.135 sec elapsed
## Starting target_embeds_in  -- add_target =  FALSEtarget_embeds_in: 0.133 sec elapsed
## Starting embeds_in_target  -- add_target =  FALSEembeds_in_target: 0.137 sec elapsed
## Starting nohorts  -- add_target =  FALSEnohorts: 0.129 sec elapsed
## Starting cohortsP  -- add_target =  TRUEcohortsP: 0.132 sec elapsed
## Starting neighborsP  -- add_target =  TRUEneighborsP: 0.129 sec elapsed
## Starting target_embeds_inP  -- add_target =  TRUEtarget_embeds_inP: 0.134 sec elapsed
## Starting embeds_in_targetP  -- add_target =  TRUEembeds_in_targetP: 0.129 sec elapsed
```

```r
# Now let's streamline the dataframe; we'll select the num, fwt, and fwcp
# columns and put them in that order, while not keeping some of the other
# 'helper' columns we created

export_df <- target_df %>%
  select(Item | Pronunciation | Frequency
  | ends_with("_num") | ends_with("_fwt") | ends_with("_fwcp"))
```

```r
# save the results
write_csv(
  export_df, "slex_lexdims.csv"
)

glimpse(export_df)
```

```
## Rows: 212
## Columns: 36
## $ Item                  <chr> "ad", "ar", "ark", "art", "art^st", "bab", "...
## $ Pronunciation         <chr> "AA D", "AA R", "AA R K", "AA R T", "AA R T ...
## $ Frequency             <int> 53, 4406, 50, 274, 112, 45, 23, 341, 87, 125...
## $ cohorts_num           <int> 1, 4, 4, 4, 4, 7, 7, 7, 7, 7, 7, 7, 3, 3, 3,...
## $ neighbors_num         <int> 4, 8, 6, 5, 1, 4, 4, 2, 1, 7, 5, 1, 7, 5, 8,...
## $ rhymes_num            <int> 3, 5, 4, 3, 1, 2, 2, 1, 1, 5, 4, 1, 6, 3, 4,...
## $ homoforms_num         <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
## $ target_embeds_in_num  <int> 6, 29, 5, 9, 1, 2, 1, 1, 1, 2, 1, 1, 5, 1, 1...
## $ embeds_in_target_num  <int> 1, 1, 2, 2, 5, 1, 3, 2, 1, 2, 4, 2, 1, 3, 3,...
## $ nohorts_num           <int> 1, 3, 3, 3, 1, 3, 3, 2, 1, 3, 2, 1, 2, 2, 3,...
## $ cohortsP_num          <int> 0, 1, 1, 1, 3, 4, 4, 5, 6, 4, 5, 6, 1, 1, 0,...
## $ neighborsP_num        <int> 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2,...
## $ target_embeds_inP_num <int> 3, 21, 1, 5, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0...
## $ embeds_in_targetP_num <int> 0, 0, 0, 0, 2, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0,...
## $ cohorts_fwt           <dbl> 3.970292, 22.634373, 22.634373, 22.634373, 2...
## $ neighbors_fwt         <dbl> 21.533445, 37.968634, 33.688446, 27.349358, ...
## $ rhymes_fwt            <dbl> 13.142723, 24.473191, 19.684596, 15.046612, ...
## $ homoforms_fwt         <dbl> 3.970292, 8.390723, 3.912023, 5.613128, 4.71...
## $ target_embeds_in_fwt  <dbl> 29.792782, 127.685319, 22.680328, 42.517044,...
## $ embeds_in_target_fwt  <dbl> 3.970292, 8.390723, 12.302746, 14.003851, 35...
## $ nohorts_fwt           <dbl> 3.970292, 17.915874, 17.915874, 17.915874, 4...
## $ cohortsP_fwt          <dbl> 0.000000, 4.718499, 4.718499, 4.718499, 17.9...
## $ neighborsP_fwt        <dbl> 8.390723, 3.970292, 0.000000, 0.000000, 0.00...
## $ target_embeds_inP_fwt <dbl> 16.650059, 88.968478, 2.995732, 22.751933, 0...
## $ embeds_in_targetP_fwt <dbl> 0.000000, 0.000000, 0.000000, 0.000000, 16.5...
## $ cohorts_fwcp          <dbl> 1.00000000, 0.37070710, 0.17283550, 0.247991...
## $ neighbors_fwcp        <dbl> 0.1843779, 0.2209909, 0.1161236, 0.2052380, ...
## $ rhymes_fwcp           <dbl> 0.3020905, 0.3428536, 0.1987352, 0.3730493, ...
## $ homoforms_fwcp        <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
## $ target_embeds_in_fwcp <dbl> 0.13326355, 0.06571407, 0.17248529, 0.132020...
## $ embeds_in_target_fwcp <dbl> 1.0000000, 1.0000000, 0.3179797, 0.4008275, ...
## $ nohorts_fwcp          <dbl> 1.0000000, 0.4683401, 0.2183551, 0.3133047, ...
## $ cohortsP_fwcp         <dbl> 1.0000000, 0.6400626, 0.4532777, 0.5432957, ...
## $ neighborsP_fwcp       <dbl> 0.3211947, 0.6788053, 1.0000000, 1.0000000, ...
## $ target_embeds_inP_fwcp <dbl> 0.19254240, 0.08618315, 0.56632333, 0.197888...
## $ embeds_in_targetP_fwcp <dbl> 1.0000000, 1.0000000, 1.0000000, 1.0000000, ...
```

Appendix 2: Bug reports and user contributions

How to report bugs. Report any bugs at https://github.com/maglab-uconn/LexFindR/issues by clicking on "New Issue".

How to create an extension. To contribute new functions, first please read the R files that are part of the LexFindR package. New functions can be added to *extensions.R* on your local installation. New functions should be carefully tested and the code should be clearly commented. Once you are confident your code is ready to be shared, move on to the next step of submitting your code via github.

How to contribute extensions via github. Extensions are welcomed through a github "pull request". Once the user has created a local clone of the forked repository, the user can edit the *competitors.R* or *extensions.R* file and push their edits to their forked path. Once these edits have been made, users can open a pull request. Before every pull request, run R CMD check to ensure that the code is clean. Please also style your code using the tidyverse style guide at https://style.tidyverse.org/ (Wickham, n.d.) and document your code using *roxygen2* (Wickham, Danenberg, Csárdi, & Eugster, 2020). We will monitor pull requests and merge appropriate changes.

References

Balota, D., Yap, M., Cortese, M., Hutchison, K., Kessler, B., Loftis, B., … Treiman, R. (2007). The english lexicon project. *Behavior Research Methods*, *39*, 445–459. https://doi.org/10.3758/BF03193014

Bengtsson, H. (2013). *future: Unified Parallel and Distributed Processing in R for Everyone*.

Brysbaert, M., & New, B. (2009). Moving beyond Kučera and Francis: A critical evaluation of current word frequency norms and the introduction of a new and improved word frequency measure for American English. *Behavior Research Methods*, *41*, 977–990.

CMU Computer Science. (2020). *CMU Pronouncing Dictionary*. Pittsburgh, PA, USA: Carnegie Mellon University. Retrieved from http://www.speech.cs.cmu.edu/cgi-bin/cmudict, August 25, 2020.

Coltheart, M., Davelaar, E., Jonasson, J. T., & Besner, D. (1977). Access to the internal lexicon. In S. Dornic (Ed.) (pp. 535–555). Hillsdale, NJ, USA: Erlbaum.

Duchon, A., Perea, M., Sebastián-Gallés, N., Martí, A., & Carreiras, M. (2013). EsPal: One-stop shopping for spanish word properties. *Behavior Research Methods*, *45*(4), 1246–1258.

Francis, W., & Kučera, H. (1982). *Frequency Analysis of English usage: Lexicon and Grammar*. Boston: Houghton Mifflin.

Goh, W., Yap, M., & Chee, Q. (2020). The Auditory English Lexicon Project: A multi-talker, multi-region psycholinguistic database of 10,170 spoken words and nonwords. *Behavior Research Methods*. https://doi.org/10.3758/s13428-020-01352-0

Izrailev, S. (2014). *tictoc: Functions for timing R scripts, as well as implementations of Stack and List structures*.

Kirk, K., Pisoni, D., & Osberger, M. (1995). Lexical effects on spoken word recognition by pediatric cochlear implant users. *Ear and Hearing*, *16*, 470–481. https://doi.org/10.1097/00003446-199510000-00004

Kučera, H., & Francis, W. (1967). *Computational Analysis of Present-Day American English*. Providence, RI, USA: Brown University Press.

Li, Z., Crinnion, A. M., & Magnuson, J. S. (2020). *LexFindR: Find related items and lexical dimensions in a lexicon*. Retrieved from https://github.com/maglab-uconn/LexFindR

Luce, P., & Pisoni, D. (1998). Recognizing spoken words: The neighborhood activation model. *Ear and Hearing*, *19*, 1–36. https://doi.org/10.1097/00003446-199802000-00001

Marian, V., Bartolotti, J., Chabal, S., & Shook, A. (2012). CLEARPOND: Cross-linguistic easy-access resource for phonological and orthographic neighborhood densities. *PLoS ONE*, *7*. https://doi.org/10.1371/journal.pone.0043230

Marslen-Wilson, W. D., & Welsh, A. (1978). Processing interactions and lexical access during word recognition in continuous speech. *Cognitive Psychology*, *10*, 29–63. https://doi.org/https://doi.org/10.1016/0010-0285(78)90018-X

McClelland, J., & Elman, J. (1986). The TRACE model of speech perception. *Cognitive Psychology*, *18*, 1–86.

Morrisette, M., & Gierut, J. (2002). Lexical organization and phonological change in treatment. *Journal of Speech, Language, and Hearing Research*, *45*, 143–159. https://doi.org/10.1044/1092-4388(2002/011)

Nenadić, F., & Tucker, B. V. (2020). Computational modelling of an auditory lexical decision experiment using jTRACE and tisk. *Language, Cognition and Neuroscience*, 1–29.

Nenadić, F., & Tucker, B. V. (2020). Computational modelling of an auditory lexical decision experiment using jTRACE and TISK. *Language, Cognition and Neuroscience*, *0*, 1–29. https://doi.org/10.1080/23273798.2020.1764600

New, B., Pallier, C., Brysbaert, M., & Ferrand, L. (2004). Lexique 2: A new french lexical database. *Behavior Research Methods, Instruments, and Computers*, *36*, 516–524. https://doi.org/10.3758/BF03195598

R Core Team. (2019). *R: A language and environment for statistical computing*. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/

Sommers, M., & Danielson, S. (1999). Inhibitory processes and spoken word recognition in young and older adults: The interaction of lexical competition and semantic context. *Psychology and Aging*, *14*, 458–472. https://doi.org/10.1037/0882-7974.14.3.458

Storkel, H., Bontempo, D., Aschenbrenner, A., Maekawa, J., & Lee, S.-Y. (2013). The effect of incremental changes in phonotactic probability and neighborhood density on word learning by preschool children. *Journal of Speech, Language, and Hearing Research*, *56*, 1689–1700. https://doi.org/10.1044/1092-4388(2013/12-0245)

Storkel, H., Maekawa, J., & Hoover, J. (2010). Differentiating the effects of phonotactic probability and neighborhood density on vocabulary comprehension and production: A comparison of preschool children with versus without phonological delays. *Journal of Speech, Language, and Hearing Research*, *53*, 933–949. https://doi.org/10.1044/1092-4388(2009/09-0075)

Vitevitch, M., & Luce, P. (1998). When words compete: Levels of processing in perception of spoken words. *Psychological Science*, *9*, 325–329. https://doi.org/10.1111/1467-9280.00064

Vitevitch, M., & Luce, P. (1999). Probabilistic phonotactics and neighborhood activation in spoken word recognition. *Journal of Memory and Language*, *40*, 374–408. https://doi.org/10.1006/jmla.1998.2618

Wickham, H. (n.d.). *The tidyverse style guide*. Retrieved from https://style.tidyverse.org/

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., … Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, *4*(43), 1686. https://doi.org/10.21105/joss.01686

Wickham, H., Danenberg, P., Csárdi, G., & Eugster, M. (2020). *Roxygen2: In-line documentation for r*. Retrieved from https://CRAN.R-project.org/package=roxygen2