JudiLing: An implementation for Discriminative Learning in Julia

Xuefeng Luo
Advisor: Prof. Dr. Harald Baayen & Dr. Yu-Ying Chuang*

February 2021

Abstract

Baayen et al. (2019) introduce the Linear Discriminative Learning (LDL) and give an implementation of LDL in R, the `WpmWithLdl` (Baayen et al., 2018a). It works well for modest-size datasets, for example, a Latin dataset with 672 inflectional verbs. However since `WpmWithLdl` has not been optimized, it takes days to process large datasets and sometime even crashes because it runs out of memory. Julia natively supports Linear Algebra and computes faster for various matrix operations compared with what `WpmWithLdl` implemented, so we re-implement Linear Discriminative Learning in Julia and give three worked examples on Latin, French and Estonian using `Judiling`. In JudiLing, the model performs much faster and no longer runs out of memory for larger datasets. We also present present two new path finding algorithms where one (learn_paths) can provide positional learnablities and the other (build_paths) restricts candidate cues to decrease the number of candidate paths. While maintaining the same accuracy as in `WpmWithLdl`, JudiLing speeds up the process time and makes it possible to study on much larger datasets.

1 Introduction

Many studies introduce the morpheme as the smallest linguistic unit under the assumption that words are built from morphemes. According to Plag (Plag, 2018), for example, the word ***unhappy*** is build from three morphemes: ***hap*** is a root which cannot be analyzed further into morphemes; $-\boldsymbol{y}$ is affix where it can attach to a base (root) ***hap*** to form a derivative ***happy***; Then, $\boldsymbol{un}-$ is a suffix which can attach in front of another base, in this case, ***happy***. Other complex word ***colonialization*** are combined with more morphemes: ***colony***, $-\boldsymbol{al}$, $-\boldsymbol{ize}$ and $-\boldsymbol{ation}$.

*Special thanks to Prof. Dr. Douglas M. Bates helping solving matrix inversion using Cholesky Decomposition and Julia coding styles.

However, other studies argue that morpheme is also problematic. Also in Plag's book, he points out that one major problem with the morpheme is that not all morphemes instantiate a one-to-one mapping from meaning to form. For example, the verb "water" derived from the noun "water" without adding anything so here a morphemic approach can be rescued by positing zero morphemes. For another example, a so-called causative morpheme "make X", such as "humid" to "humidify" (make humid) and "black" to "blacken" (make black), doesn't fit well with the pattern of "fall" and "fell" (make fall).

In this case, Baayen et al. (2019) introduces a novel model, Linear Discriminative Learning (LDL), and then provides a step-by-step introduction to its `R` implementation (Baayen et al., 2018a). This study presents a loss-free model for 672 inflected Latin verbs without morphemes play any role in the model.

Since the implementation of LDL in `R` has not been optimized, it quickly runs into its limits. When studying this model with large datasets, the implementation spends days to compute and sometimes crashes because it runs out of memory. Therefore, we have re-implemented this model in Julia. Compared with R, Julia has many advantages, but one major advantage is that Julia natively supports Linear Algebra computations like matrix multiplication and matrix inversion, operations which LDL makes heavy use of.

In `WpmWithLdl`, Baayen et al. (2018a) use a simple path algorithm (Csardi et al., 2006) to find all candidates paths for producing a word given its n-phones. This algorithm works initially well for datasets of modest size. However, there are two major issues found in later studies: 1) it doesn't handle paths with cycles well since paths with cycles will lead to infinite number of paths; 2) it doesn't scale up to larger datasets because as the number of candidate cues increasing, the number of candidate paths increases exponentially. Therefore we first introduce the concept of timesteps to implement paths (including paths with cycles) and second we develop two new paths finding algorithms, `learn_paths` and `build_paths`. The first algorithm provides positional learnablities for each timestep while the other algorithm further restricts the number of candidates cues by only considering cues for the k nearest form neighbors.

In what follows, section 2 introduces LDL model in detail with a toy example corpus, section 3 describes the bottlenecks of the R implementation and then gives solutions for those bottlenecks, section 4 describes other useful features implemented in `JudiLing`, section 5 further provides three worked examples using `JudiLing` and finally section 6 presents the take home messages for this implementation.

2 Linear Discriminative Learning (LDL)

Linear Discriminative Learning (LDL, Baayen et al., 2018a, 2019) is a computational model that makes use of a wide two-layer network, without any hidden layers. This model aims to construct an interpretable two-directional language model for both comprehension and production. On the comprehension side, this model learns in one step the mapping from word forms to word meanings,

Table 1: A toy example of a small lexicon with four words. Their forms and meanings (including base and inflectional meanings) are provided here.

Words	Features
walk	WALK, PRESENT
walks	WALK, PRESENT, SINGULAR
walked	WALK, PAST
walked	WALK, PAST, PARTICIPLE

while on the production side, the model learns to map word meanings onto word forms. To understand how LDL works, we use a toy example (Table 1) with four words: *walk*, *walks*, *walked* (past) and *walked* (participle). In what follows, we introduce the representations of words' forms and meanings, the model's underlying mathematical algorithms, and the evaluation of model performance.

2.1 Form and meaning representations

In LDL, both word forms and word meanings are represented by numeric vectors. For word forms, we extract sublexical cues from all words and represent the presence and absence of these cues in a given word with one-hot encoding. For our toy example, here we use letter trigrams as sublexical cues. A word form matrix, denoted by $\boldsymbol{C}$, can now be constructed:

$$\boldsymbol{C} = \begin{array}{c} \\ \text{walk} \\ \text{walks} \\ \text{walked} \\ \text{walked} \end{array} \begin{array}{c} \begin{array}{ccccccccc} \#\text{wa} & \text{wal} & \text{alk} & \text{lk}\# & \text{lks} & \text{ks}\# & \text{lke} & \text{ked} & \text{ed}\# \end{array} \\ \left[\begin{array}{ccccccccc} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \end{array}\right] \end{array}. \qquad (1)$$

The columns in $\boldsymbol{C}$ list all the letter trigrams in the dataset, and the rows represent words' form vectors. The word *walks*, for example, contains the first five letter trigrams listed in the column names of $\boldsymbol{C}$, which are coded with ones. All other trigrams are coded with zeroes.

With regard to meaning representations, word embeddings such as produced by word2vec (Mikolov et al., 2013), glove (Pennington et al., 2014), fasttext (Bojanowski et al., 2016), etc. can be used. However, in order to better understand morphological processing, in our model special consideration is given to the construction of semantic vectors for morphologically complex words. As illustrated in Table 1, the meaning of *walks* is constructed from the base meaning WALK, and two inflectional meanings, PRESENT and SINGULAR. In the framework of discriminitive learning (Baayen et al., 2011, 2016), these basic meanings (represented by small caps) are referred to as lexomes. As the three lexomes all contribute the meaning of *walks*, we therefore construct the semantic vector of *walks* (henceforth $\overrightarrow{walks}$) by summing up the semantic vectors of the three lexomes:

$$\overrightarrow{walks} = \overrightarrow{\text{WALK}} + \overrightarrow{\text{PRESENT}} + \overrightarrow{\text{SINGULAR}}.$$

For our toy example, we simulate numeric vectors (here of length 6) from a normal distribution for all the lexomes, which we bring together in a lexome matrix $\boldsymbol{L}$.

$$L = \begin{array}{c} \\ \text{WALK} \\ \text{PRESENT} \\ \text{PAST} \\ \text{SINGULAR} \\ \text{PARTICIPLE} \end{array} \begin{array}{c} \begin{array}{rrrrrr} \text{S1} & \text{S2} & \text{S3} & \text{S4} & \text{S5} & \text{S6} \end{array} \\ \left[\begin{array}{rrrrrr} -1.52 & -0.69 & 0.05 & -0.31 & 1.60 & 0.23 \\ -0.92 & -0.86 & 1.30 & 0.01 & 0.22 & -0.58 \\ -0.69 & 0.98 & 2.94 & -0.06 & 1.10 & 2.10 \\ -0.01 & 0.69 & -0.26 & -0.56 & -0.89 & -1.09 \\ -1.37 & -0.98 & 0.81 & 0.01 & 0.56 & 0.31 \end{array}\right] \end{array}$$

Given $\boldsymbol{L}$, we can then construct semantic vectors for all words, which are given by the rows of the semantic matrix $\boldsymbol{S}$:

$$\boldsymbol{S} = \begin{array}{c} \\ \text{walk} \\ \text{walks} \\ \text{walked}_{past} \\ \text{walked}_{part} \end{array} \begin{array}{c} \begin{array}{rrrrrr} \text{S1} & \text{S2} & \text{S3} & \text{S4} & \text{S5} & \text{S6} \end{array} \\ \left[\begin{array}{rrrrrr} -2.44 & -1.55 & 1.35 & -0.30 & 1.82 & -0.35 \\ -2.45 & -0.86 & 1.09 & -0.86 & 0.93 & -1.44 \\ -2.21 & 0.29 & 2.99 & -0.37 & 2.70 & 2.33 \\ -3.58 & -0.69 & 3.80 & -0.36 & 3.26 & 2.64 \end{array}\right] \end{array}.$$

2.2 Comprehension and production networks

Given word form representations $\boldsymbol{C}$ and meaning representations $\boldsymbol{S}$, the model learns to map from one representation to the other. For comprehension, word forms are mapped onto meanings, and for production, word meanings are mapped onto word forms. This is formally represented by (2) and (3), where $\boldsymbol{F}$ and $\boldsymbol{G}$ denote the comprehension and production networks respectively:

$$\boldsymbol{C}\boldsymbol{F} = \boldsymbol{S}, \tag{2}$$
$$\boldsymbol{S}\boldsymbol{G} = \boldsymbol{C}. \tag{3}$$

The mapping $\boldsymbol{F}$ is obtained by solving (2), which can be accomplished with the help of the generalised inverse of $\boldsymbol{C}$, henceforth $\boldsymbol{C}^{-1}$:

$$\begin{aligned} \boldsymbol{C}^{-1}\boldsymbol{C}\boldsymbol{F} &= \boldsymbol{C}^{-1}\boldsymbol{S}, \\ \boldsymbol{F} &= \boldsymbol{C}^{-1}\boldsymbol{S}. \end{aligned} \tag{4}$$

Below, we will discuss a computationally more sophisticated way of solving (2). For our toy example, we obtain the $\boldsymbol{F}$ matrix, as presented in (5). $\boldsymbol{F}$ is equivalent to a two-layer network such that every cell ($\boldsymbol{f}_{ij}$) of F specifies the connection weight between a given cue i and a given semantic dimension j.

$$\boldsymbol{F} = \begin{array}{c} \\ \text{\#wa} \\ \text{wal} \\ \text{alk} \\ \text{lk\#} \\ \text{lks} \\ \text{ks\#} \\ \text{lke} \\ \text{ked} \\ \text{ed\#} \end{array} \begin{array}{c} \begin{array}{cccccc} \text{S1} & \text{S2} & \text{S3} & \text{S4} & \text{S5} & \text{S6} \end{array} \\ \left[\begin{array}{rrrrrr} -0.71 & -0.31 & 0.47 & -0.13 & 0.50 & -0.04 \\ -0.71 & -0.31 & 0.47 & -0.13 & 0.50 & -0.04 \\ -0.71 & -0.31 & 0.47 & -0.13 & 0.50 & -0.04 \\ -0.30 & -0.61 & -0.05 & 0.09 & 0.31 & -0.24 \\ -0.16 & 0.04 & -0.15 & -0.23 & -0.29 & -0.66 \\ -0.16 & 0.04 & -0.15 & -0.23 & -0.29 & -0.66 \\ -0.25 & 0.25 & 0.67 & 0.01 & 0.49 & 0.87 \\ -0.25 & 0.25 & 0.67 & 0.01 & 0.49 & 0.87 \\ -0.25 & 0.25 & 0.67 & 0.01 & 0.49 & 0.87 \end{array}\right] \end{array} \tag{5}$$

The production network $\boldsymbol{G}$ is obtained in the same way:

$$\begin{aligned} \boldsymbol{S}^{-1}\boldsymbol{S}\boldsymbol{G} &= \boldsymbol{S}^{-1}\boldsymbol{C}, \\ \boldsymbol{G} &= \boldsymbol{S}^{-1}\boldsymbol{C}. \end{aligned} \tag{6}$$

A cell $\boldsymbol{g}_{ij}$ in $\boldsymbol{G}$ gives us the weight for cue j given semantic dimension i.

$$\boldsymbol{G} = \begin{array}{c} \\ \text{S1} \\ \text{S2} \\ \text{S3} \\ \text{S4} \\ \text{S5} \\ \text{S6} \end{array} \begin{array}{c} \begin{array}{ccccccccc} \text{\#wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \end{array} \\ \left[\begin{array}{rrrrrrrrr} 0.07 & 0.07 & 0.07 & 0.45 & -0.35 & -0.35 & -0.03 & -0.03 & -0.03 \\ 0.19 & 0.19 & 0.19 & -0.40 & 0.36 & 0.36 & 0.23 & 0.23 & 0.23 \\ 0.01 & 0.01 & 0.01 & -0.37 & 0.23 & 0.23 & 0.15 & 0.15 & 0.15 \\ -0.16 & -0.16 & -0.16 & 0.25 & -0.33 & -0.33 & -0.08 & -0.08 & -0.08 \\ 0.70 & 0.70 & 0.70 & 1.08 & -0.43 & -0.43 & 0.05 & 0.05 & 0.05 \\ -0.38 & -0.38 & -0.38 & -0.26 & -0.22 & -0.22 & 0.11 & 0.11 & 0.11 \end{array}\right] \end{array}$$

The underlying mathematics is the same as Multivariate Multiple Regression where we can obtain the predicted matrix ($\hat{\boldsymbol{S}}$ and $\hat{\boldsymbol{C}}$) by multiplying input matrix ($\boldsymbol{C}$ and $\boldsymbol{S}$) with weights ($\boldsymbol{F}$ and $\boldsymbol{G}$). When observing ($\hat{\boldsymbol{S}}$ and $\hat{\boldsymbol{C}}$), they are not exactly the same as ($\boldsymbol{C}$ and $\boldsymbol{S}$) just as in any regression model, we seek to obtain the best possible estimate, but have to accept that there is by-observation noise that we cannot explain. We thus need further analysis of the predictions as discussed in subsection 2.3 and 2.4.

2.3 Evaluating comprehension

To evaluate the performance of the comprehension model, we first multiply $\boldsymbol{C}$ by $\boldsymbol{F}$ to obtain the predicted semantic matrix $\hat{\boldsymbol{S}}$,

$$\hat{\boldsymbol{S}} = \boldsymbol{C}\boldsymbol{F}, \tag{7}$$

resulting in the following matrix for our example:

$$\hat{\boldsymbol{S}} = \begin{array}{r} \\ \text{walk} \\ \text{walks} \\ \text{walked}_{past} \\ \text{walked}_{part} \end{array} \begin{array}{c} \begin{array}{rrrrrr} \text{S1} & \text{S2} & \text{S3} & \text{S4} & \text{S5} & \text{S6} \end{array} \\ \left[\begin{array}{rrrrrr} -2.44 & -1.55 & 1.35 & -0.30 & 1.82 & -0.35 \\ -2.45 & -0.86 & 1.09 & -0.86 & 0.93 & -1.44 \\ -2.90 & -0.20 & 3.40 & -0.37 & 2.98 & 2.49 \\ -2.90 & -0.20 & 3.40 & -0.37 & 2.98 & 2.49 \end{array}\right] \end{array}.$$

Comparing $\hat{\boldsymbol{S}}$ and $\boldsymbol{S}$, we can see that the predicted semantic vectors are very similar to the gold standard ones. This is due to the small size of the current toy example. We shall see below that as datasets grow bigger, $\hat{\boldsymbol{S}}$ and $\boldsymbol{S}$ will be less similar.

To quantify prediction accuracy, we calculate the correlations of a given predicted semantic vector with all the gold standard semantic vectors in the dataset. These correlations are presented in Table 2. For every predicted semantic vector, we take the gold standard vector with which it is correlated the most as the meaning that the model recognizes. For example, as $\hat{\boldsymbol{s}}_{walk}$ and $\boldsymbol{s}_{walk}$ have the highest correlation ($r = 1$), $\boldsymbol{s}_{walk}$ is selected as the predicted meaning.

Table 2: Pair-wise correlations between the predicted semantic vectors and the gold standard semantic vectors.

	$\boldsymbol{s}_{walk}$	$\boldsymbol{s}_{walks}$	$\boldsymbol{s}_{walked_{past}}$	$\boldsymbol{s}_{walked_{part}}$
$\hat{\boldsymbol{s}}_{walk}$	1.000	0.922	0.866	0.908
$\hat{\boldsymbol{s}}_{walks}$	0.922	1.000	0.798	0.815
$\hat{\boldsymbol{s}}_{walked_{past}}$	0.893	0.810	0.997	0.998
$\hat{\boldsymbol{s}}_{walked_{part}}$	0.893	0.810	0.997	0.998

Homophones such as *walked*$_{past}$ and *walked*$_{part}$ share the same form vector $\boldsymbol{c}$ but map onto different semantic vectors $\boldsymbol{s}_{past}$ and $\boldsymbol{s}_{part}$. Precisely because they have the same form, homophones' predicted semantic vectors $\hat{\boldsymbol{s}}_{past}$ and $\hat{\boldsymbol{s}}_{part}$ are identical, and hence their correlations with the corresponding gold standard vectors $\boldsymbol{s}_{past}$ and $\boldsymbol{s}_{part}$ are identical as well. However, in general the correlation of $\boldsymbol{s}$ and $\hat{\boldsymbol{s}}$ will be larger for one of the homophones, in the present example, this holds for $\boldsymbol{s}_{walkedpart}$. This would suggest successful recognition for *walked*$_{part}$ but unsuccessful recognition for *walked*$_{pest}$. However, in single word recognition tasks, it is impossible to tell apart the meanings of homophones. Arguably, model performance can be considered to be correct if one of the homophone meanings is correctly retrieved. Therefore, the function in the `JudiLing` package that evaluates comprehension accuracy provides an option for lenient evaluation of homophones (see Section 3.2).

2.4 Evaluating production

Similar to the evaluation of the comprehension model, for production we first obtain the predicted form matrix $\hat{\boldsymbol{C}}$ with equation 7. Unlike $\boldsymbol{C}$ with binary values of 0's and 1's (equation 1), $\hat{\boldsymbol{C}}$ consists of real-valued vectors, as presented in equation 9.

$$\hat{\boldsymbol{C}} = \boldsymbol{SG} \tag{8}$$

$$\hat{\boldsymbol{C}} = \begin{array}{c} \\ \text{walk} \\ \text{walks} \\ \text{walked}_{past} \\ \text{walked}_{part} \end{array} \left[\begin{array}{ccccccccc} \#\text{wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \\ 1.0 & 1.0 & 1.0 & 1.0 & -0.0 & -0.0 & -0.0 & -0.0 & -0.0 \\ 1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 1.0 & -0.0 & -0.0 & -0.0 \\ 1.0 & 1.0 & 1.0 & 0.0 & -0.0 & -0.0 & 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & -0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 \end{array} \right] \tag{9}$$

Although it is possible to evaluate $\hat{\boldsymbol{C}}$ in the same way as $\hat{\boldsymbol{S}}$, i.e., locating the gold standard vector with the highest correlation, this method is however not precise enough. This is because these form vectors are unordered, and we do not know whether the predicted form vectors can actually generate the intended forms. For the small corpus above, the model is good enough to generate perfect $\hat{\boldsymbol{C}}$ in order to form one and exact one path for each word. However, considering a second language learner who has strong background in verb declensions in numbers. Therefore, he wants to know about the correct form of WALK for PAST tense and SINGULAR who already knows that English words have to be inflected for number, but who has never encountered the form "walked" before. Can the model correctly predict this form?

We first construct the pertinent semantic vector $\boldsymbol{s}$ by summing up its lexome vectors (10).

$$\begin{aligned} \boldsymbol{s} = & \ \overrightarrow{\text{WALK}} + \overrightarrow{\text{PAST}} + \overrightarrow{\text{SINGULAR}} \\ = & \ \left[\begin{array}{cccccc} \text{S1} & \text{S2} & \text{S3} & \text{S4} & \text{S5} & \text{S6} \\ -2.90 & -0.20 & 3.40 & -0.37 & 2.98 & 2.49 \end{array} \right] \end{aligned} \tag{10}$$

We as English speakers may know that the form is exactly the same as the PAST tense form *walked*, but the model doesn't know that because it has never been trained on this word. However, we can use equation (8) to obtain a predicted form vector:

$$
\begin{aligned}
\hat{c} = & \quad \boldsymbol{SG} \\
= & \begin{array}{c} \\ \left[\right. \end{array}\begin{array}{ccccccccc} \text{\#wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \\ 1.0 & 1.0 & 1.0 & -1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \end{array}\begin{array}{c} \\ \left. \right]. \end{array}
\end{aligned}
\tag{11}
$$

We then find that the predicted form vector $\hat{c}$ could potentially form two possible paths, *walks* and *walked*. To deal with the ordering issue, Baayen et al. (2019) adopts a path-finding algorithm in the graph theory. As a first step, the cues with activation above a given threshold are collected. Next, all possible paths are constructed for these cues based on an adjacency matrix. By way of example, for our toy example, the adjacency matrix $\boldsymbol{A}$, shown in (12), is a 9×9 matrix, with all the trigrams in the dataset listed in both the rows and columns. The binary value in $\boldsymbol{A}$, $\boldsymbol{a}_{ij}$, indicates whether cue i can be attached to cue j. The trigram *lk#*, for example, can follow *alk* ($\boldsymbol{a}_{93} = 1$) but not *lks* ($\boldsymbol{a}_{94} = 0$).

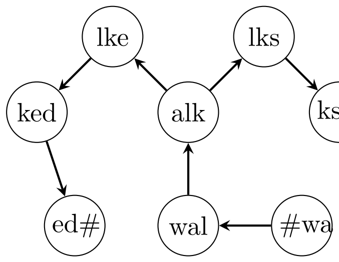
$$
\boldsymbol{A} = \begin{array}{c|ccccccccc|}
 & \text{\#wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \\
\text{\#wa} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{wal} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{alk} & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{lk\#} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{lks} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{ks\#} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\text{lke} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{ked} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\text{ed\#} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}
\tag{12}
$$

Now to construct all possible paths, we only consider trigrams that receive high activation. Trimming as such is not strictly necessary, but for reasons that will become clearer later, the thresholding parameter was introduced in the `R` implementation of LDL (Baayen et al., 2018b) to reduce computational costs. Take the predicted form vector $\hat{\boldsymbol{c}}$ (11) for example. With a threshold of 0.1, eight trigrams are left, including *#wa*, *wal*, *alk*, *lks*, *ks#*, *lke*, *ked* and *ed#*. Based on $\boldsymbol{A}$, a trigram graph is constructed, as shown in Figure 1. The trigrams are placed on the vertices, and the directed edges indicate valid continuations from one trigram to another.

According to Figure 1, two candidate forms can be found. One is *walked*, with a trigram path starting from *#wa*, followed by *wal*, *alk*, *lke*, *ked* and finally to *ed#*. The other predicted form is *walks*, which has the same first three trigrams as the former path, but continues with *lks*, and ends with *ks#*.

The last step for production evaluation is to determine which candidate form (when there is more than one) should be selected as the model's predicted form. The procedure in (Baayen et al., 2018a) is to select the candidate form that resonates most with the intended meaning. Specifically, we create another predicted form matrix for the candidate forms ($\hat{\boldsymbol{C}}$, 13), and then multiply it by the comprehension network $\boldsymbol{F}$ to generated the predicted semantic matrix for

Figure 1: Possible paths for $\hat{\boldsymbol{c}}$ in a directed triphone graph.



these candidate forms ($\hat{\boldsymbol{S}}$, 14). Following the steps for evaluating comprehension (cf. Section 2.3), we calculate the correlations between the predicted semantic vectors of the candidate forms with the intended semantic vector $\boldsymbol{s}$ (10). As presented in Table 7, $\hat{\boldsymbol{s}}_{\boldsymbol{walked}}$ has a higher correlation score than $\hat{\boldsymbol{s}}_{\boldsymbol{walks}}$. The model then picks *walked* as the predicted form, which is the correct form. This procedure is referred to as 'synthesis-by-analysis' in (Baayen et al., 2018a).

$$\hat{\boldsymbol{C}} = \begin{array}{c} \\ \text{walked} \\ \text{walks} \end{array} \begin{array}{c} \begin{array}{ccccccccc} \#\text{wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \end{array} \\ \left[\begin{array}{ccccccccc} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \end{array}\right] \end{array} \tag{13}$$

$$\hat{\boldsymbol{S}}_{walk} = \begin{array}{c} \\ \text{walked} \\ \text{walks} \end{array} \begin{array}{c} \begin{array}{cccccc} \text{S1} & \text{S2} & \text{S3} & \text{S4} & \text{S5} & \text{S6} \end{array} \\ \left[\begin{array}{cccccc} -2.90 & -0.20 & 3.40 & -0.37 & 2.98 & 2.49 \\ -2.45 & -0.86 & 1.09 & -0.86 & 0.93 & -1.44 \end{array}\right] \end{array} \tag{14}$$

Table 3: Pair-wise correlations between predicted path semantic vectors and original semantic vector.

	$\boldsymbol{s}_{\boldsymbol{walked}}$	selected	correct
$\hat{\boldsymbol{s}}_{\boldsymbol{walked}}$	0.93	✓	✓
$\hat{\boldsymbol{s}}_{\boldsymbol{walks}}$	0.83	✗	✗

3 Bottlenecks and new implementation

LDL was first implemented in `R` (R Core Team, 2020). With the package `WpmWithLdl` (Baayen et al., 2018b), a number of studies have so far been done to model various comprehension and production tasks in different languages (e.g., Chuang et al., 2020a,b,c; Heitmeier and Baayen, 2020; Baayen and Smolka, 2020). While the package generally works well, problems however arise when one intends to work with bigger datasets. In particular, modeling big datasets with `WpmWithLdl` becomes prohibitively expensive in terms of both computing time and resources. In what follows, we will discuss three major bottlenecks of `WpmWithLdl`, and how these difficulties are now dealt with by the new implementation of LDL in `Julia` (Bezanson et al., 2017).

3.1 Matrix inversion and sparsity

As mentioned previously, to obtain $\boldsymbol{F}$ and $\boldsymbol{G}$, we need the inverse of $\boldsymbol{C}$ and $\boldsymbol{S}$, i.e., $\boldsymbol{C}^{-1}$ and $\boldsymbol{S}^{-1}$ (cf. equation 4 and 6). As inverting a large matrix is computationally expensive, Baayen et al. (2018a) first make $\boldsymbol{C}$ and $\boldsymbol{S}$ into square matrices by multiplying with their respective transposed matrix. The inversion can then be done on smaller matrices of $\boldsymbol{C}^t\boldsymbol{C}$ and $\boldsymbol{S}^t\boldsymbol{S}$. Following this method, we can likewise obtain $\boldsymbol{F}$ and $\boldsymbol{G}$. The formulae are provided in equations (15) and (16).

$$\begin{aligned}
(\boldsymbol{C}^t\boldsymbol{C})^{-1}\boldsymbol{C}^t\boldsymbol{C}\boldsymbol{F} &= (\boldsymbol{C}^t\boldsymbol{C})^{-1}\boldsymbol{C}^t\boldsymbol{S} \\
\boldsymbol{F} &= (\boldsymbol{C}^t\boldsymbol{C})^{-1}\boldsymbol{C}^t\boldsymbol{S} \qquad (15) \\
(\boldsymbol{S}^t\boldsymbol{S})^{-1}\boldsymbol{S}^t\boldsymbol{S}\boldsymbol{G} &= (\boldsymbol{S}^t\boldsymbol{S})^{-1}\boldsymbol{S}^t\boldsymbol{C} \\
\boldsymbol{G} &= (\boldsymbol{S}^t\boldsymbol{S})^{-1}\boldsymbol{S}^t\boldsymbol{C} \qquad (16)
\end{aligned}$$

In `WpmWithLdl`, the Moore-Penrose pseudoinverse is calculated with the `ginv` function from the `MASS` (Venables and Ripley, 2002) package. An equivalent function in `Julia` is the `pinv` function from the `Linear Algebra` standard library in `Julia`. Figure 2 shows that calculating pseduoinverses is already generally faster in `Julia` (the orange line) than in `R` (the blue line). To speed up the calculation further, in `JudiLing`, we incorporate Cholesky decomposition (factorization), also provided by the `Linear Algebra` package. This reduces computing time substantially, as shown by the green line in Figure 2.

Another novelty of `JudiLing` is to take the sparsity of matrices into account. In LDL, not all matrices are of equal sparsity/density. The semantic matrix $\boldsymbol{S}$, for example, is usually dense, whereas the form matrix $\boldsymbol{C}$ is often very sparse. This is because among all the sublexical cues in a language, words usually only contain a small number of these cues. This results in a lot of 0's and few 1's in $\boldsymbol{C}$. In `Julia`, matrices can be represented/coded with either a regular dense array format or a sparse format, the latter of which is supported by the `Sparse Arrays` in `Julia` standard library. Importantly, the proper matrix format affords efficient matrix operations including inversion. Figure 3 presents the amount of time saved when inverting matrices of different densities by using the sparse format in Julia. When a matrix is small, the inverting time is practically the same no matter which format is used. Nevertheless, when dealing with big matrices, the advantage of the sparse format gradually emerges, and the amount of time saved, especially for really sparse matrices (the blue and orange lines), grows exponentially. In the light of this, in `JudiLing`, we always use the sparse format for $\boldsymbol{C}$ and the dense format for $\boldsymbol{S}$ to optimize matrix inversion. As for other matrices such as $\boldsymbol{F}$ and $\boldsymbol{G}$, the density of which is less predictable, a decision function is designed to automatically determine the appropriate format.

Figure 2: Processing time for inverting matrices of varying sizes in `R` and in `Julia`. Note that the y-axis presents time in a logarithmic scale.

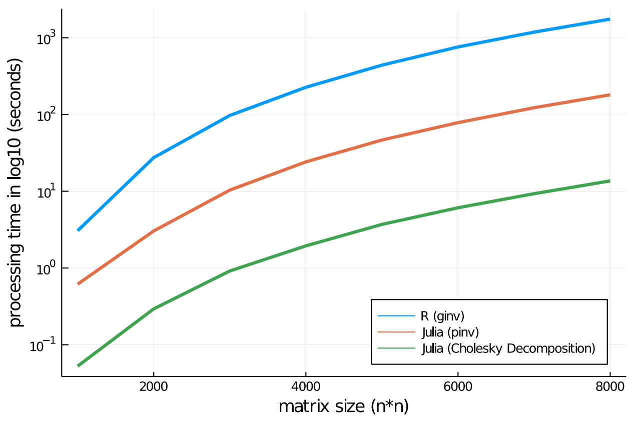
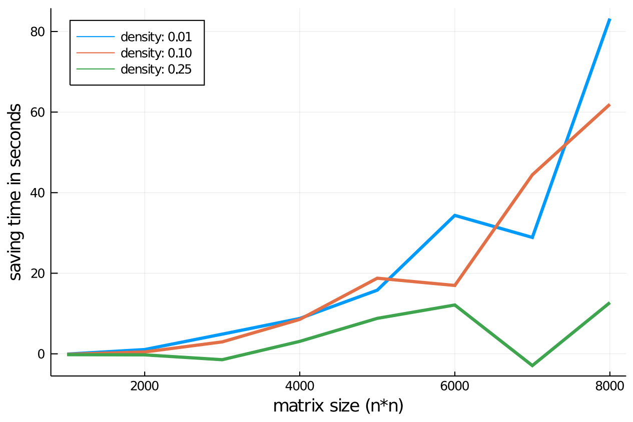


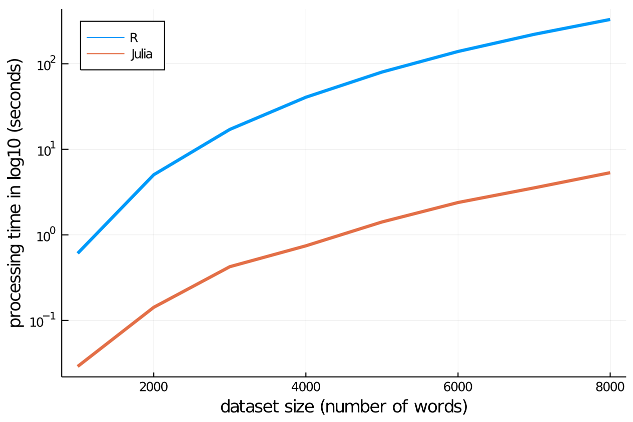
Figure 3: Time saved for matrix inversion in Julia when matrices of different densities and sizes are coded in the sparse format.



3.2 Pair-wise correlation

The second bottleneck pertains to evaluation. As mentioned previously in Section 2.3 and 2.4, the evaluation of both comprehension and production models involve calculating correlations. In `WpmWithLdl`, this is done with the `cor` function from the `stats` standard library in `R`. Similar to the problem of matrix inversion, computing time increases exponentially when big datasets are dealt with. For `JudiLing` we use the `cor` function from the `Statistics` in `Julia` standard library . A comparison of the `R` and the `Julia` function is presented in Figure 4. Across datasets of all sizes, calculating correlations is faster in `Julia`. Note that the y-axis is presented in a logarithmic scale. Thus, for a dataset contains 8000 words, for example, while calculation takes more than 100 seconds in `R`, it takes only less than 10 seconds in `Julia`. Apparently the `Julia` code has been optimized to calculate correlations, which helps `JudiLing` to conduct more efficient evaluation to a large extent.

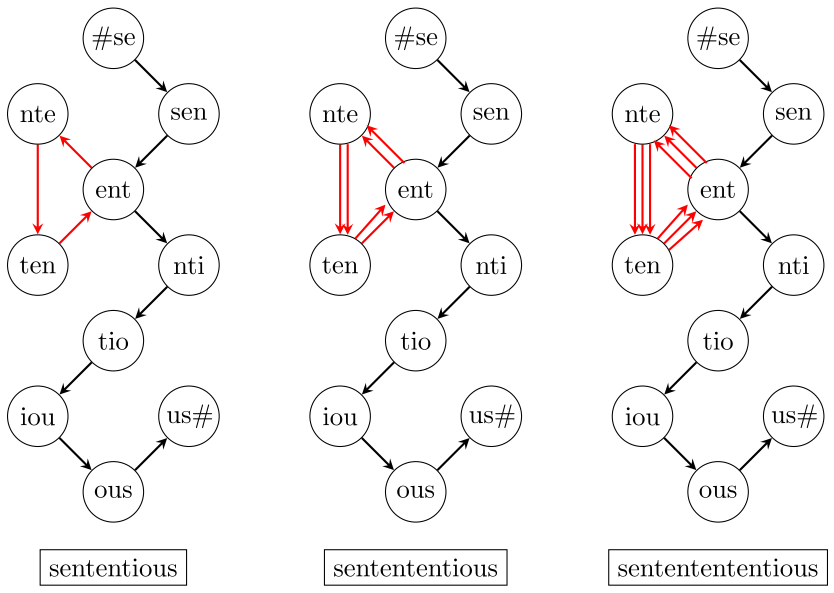
Figure 4: Time taken to calculate pair-wise correlations for datasets of varying sizes.



3.3 Path Finding Algorithm

We mentioned in Section 2.4 that part of the evaluation of production involves finding the correct orders to sequence the sublexical cues (i.e., letter trigrams). In `WpmWithLdl`, the order of trigrams is searched for by applying the path-finding algorithm in the graph theory (cf. Figure 1) from `igraph` package (Csardi et al., 2006). The major problem with this algorithm is that the number of possible paths for a word can easily reach hundreds or even thousands. This is because most trigrams, except for the word-initial and word-final ones, are not bound by positions. Thus, the non-boundary trigrams can occur in any position of a word, given the right neighboring trigrams. This problem is even aggravated when there cycles in a given word. By way of example, Figure 5 presents the trigram path of the word *sententious*. The graph in the left panel shows the correct order, where a cycle begins and ends with the trigram *ent*. Crucially, the cycle can repeat multiple times. With one extra repetition (middle panel), the form is *sentententious*; with two extra repetitions (right panel), we obtain the form *sententententious*. In other words, when cycles are allowed for path construction, the graph algorithm will find an infinite number of paths if no restrictions are imposed. `WpmWithLdl` therefore first obtains all possible shortest paths, and then implements various heuristics to include paths with short cycles. For larger datasets, unfortunately, this algorithm becomes prohibitively expensive.

Figure 5: The trigram path of the word *sententious* and two pesudo-words *sentententious* and *sententententious*, with an internal cycle.



To address this issue, in `JudiLing` we design two new path-finding algorithms, `learn_paths` and `build_paths`. The two algorighms will be introduced in Section 3.3.1 and 3.3.2 respectively.

3.3.1 The `learn_paths` algorithm

The major difference between `learn_paths` and the graph-based algorithm is that this new algorithm takes positional information into consideration. Given that the model learns to predict trigrams at different positions, there is an infinite number of paths to be traced and maybe potentially infinite.

To do so, we first copy the form matrix $\boldsymbol{C}$ for t times, where t stands for the number of trigrams of the longest word in the dataset. In other words, t denotes the longest path possible in the dataset. In our toy example, the longest word is *walked*, thus $t = 6$. The copied matrices, referred to as $\boldsymbol{Y_t}$, code the presence of the trigrams of all the words at position t. For example, the first trigram of all the words in the toy dataset is *#wa*. As a consequence, in $\boldsymbol{Y_1}$ (Equation 17), the first column (*#wa*) receives the value 1 in all the rows. At position 4 ($\boldsymbol{Y_4}$, Equation 20), *walk* reaches its last trigram *lk#*. The fourth trigram of *walks* is *lks*, and that of *walked* and *walked*$_{3rd}$ is *lke*. At the final position ($\boldsymbol{Y_6}$, Equation 22), since *walk* and *walks* have no trigrams at this position, their vectors contain all 0's. For *walked* and *walked*$_{3rd}$, their last trigram *ed#* is coded as 1.

$$\boldsymbol{Y}_1 = \begin{array}{r} \\ \text{walk} \\ \text{walks} \\ \text{walked}_{past} \\ \text{walked}_{part} \end{array} \begin{array}{c} \begin{array}{ccccccccc} \#\text{wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \end{array} \\ \left[\begin{array}{ccccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array} \qquad (17)$$

$$\boldsymbol{Y}_2 = \begin{array}{r} \\ \text{walk} \\ \text{walks} \\ \text{walked}_{past} \\ \text{walked}_{part} \end{array} \begin{array}{c} \begin{array}{ccccccccc} \#\text{wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array} \qquad (18)$$

$$\boldsymbol{Y}_3 = \begin{array}{r} \\ \text{walk} \\ \text{walks} \\ \text{walked}_{past} \\ \text{walked}_{part} \end{array} \begin{array}{c} \begin{array}{ccccccccc} \#\text{wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array} \qquad (19)$$

$$\boldsymbol{Y}_4 = \begin{array}{r} \\ \text{walk} \\ \text{walks} \\ \text{walked}_{past} \\ \text{walked}_{part} \end{array} \begin{array}{c} \begin{array}{ccccccccc} \#\text{wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}\right] \end{array} \qquad (20)$$

$$\boldsymbol{Y}_5 = \begin{array}{r} \\ \text{walk} \\ \text{walks} \\ \text{walked}_{past} \\ \text{walked}_{part} \end{array} \begin{array}{c} \begin{array}{ccccccccc} \#\text{wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right] \end{array} \qquad (21)$$

$$\boldsymbol{Y}_6 = \begin{array}{c} \\ \text{walk} \\ \text{walks} \\ \text{walked}_{past} \\ \text{walked}_{part} \end{array} \begin{array}{c} \begin{array}{ccccccccc} \text{\#wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \tag{22}$$

As a next step, the model learns to predict the positional form matrices $\boldsymbol{Y_t}$ from the non-positional form matrix $\boldsymbol{C}$. The equations are presented in (23).

$$\begin{aligned} \boldsymbol{C M_t} &= \boldsymbol{Y_t} \\ \boldsymbol{M_t} &= (\boldsymbol{C}^t \boldsymbol{C})^{-1} \boldsymbol{C}^t \boldsymbol{Y}_t \end{aligned} \tag{23}$$

And the predicted positional form matrices $\boldsymbol{\hat{Y}_t}$ are then obtained by multiplying the predicted non-positional form matrix, $\boldsymbol{\hat{C}}$, by $\boldsymbol{M_t}$.

$$\boldsymbol{\hat{Y}_t} = \boldsymbol{\hat{C} M_t}$$

Take the PAST tense and SINGULAR form (WALKED) metioned in section 2.4 for example. To obtain the predicted trigram vector at position 1, we multiply $\boldsymbol{\hat{c}}$ by $\boldsymbol{M}_1$, which gives $\boldsymbol{\hat{y}}_1$, shown in 24.

$$\boldsymbol{\hat{y}}_1 = \begin{array}{c} \begin{array}{ccccccccc} \text{\#wa} & \text{wal} & \text{alk} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} & \text{lk\#} \end{array} \\ \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \end{array} \tag{24}$$

At position 1, only one trigram receives high activation, i.e., *#wa*, assuming the threshold is set to 0.1. We therefore only keep this trgiram as a candidate trigram at position 1. We repeat the same procedure to obtain $\boldsymbol{\hat{y}}_t$ at positions 2 to 6. The resulting vectors are shown in equations 25 to .

$$\boldsymbol{\hat{y}}_2 = \begin{array}{c} \begin{array}{ccccccccc} \text{\#wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \end{array} \\ \begin{bmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \end{array} \tag{25}$$

$$\boldsymbol{\hat{y}}_3 = \begin{array}{c} \begin{array}{ccccccccc} \text{\#wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \end{array} \\ \begin{bmatrix} 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \end{array} \tag{26}$$

$$\boldsymbol{\hat{y}}_4 = \begin{array}{c} \begin{array}{ccccccccc} \text{\#wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \end{array} \\ \begin{bmatrix} 0.0 & 0.0 & 0.0 & -1.0 & 1.0 & 0.0 & 1.0 & 0.0 & 0.0 \end{bmatrix} \end{array} \tag{27}$$

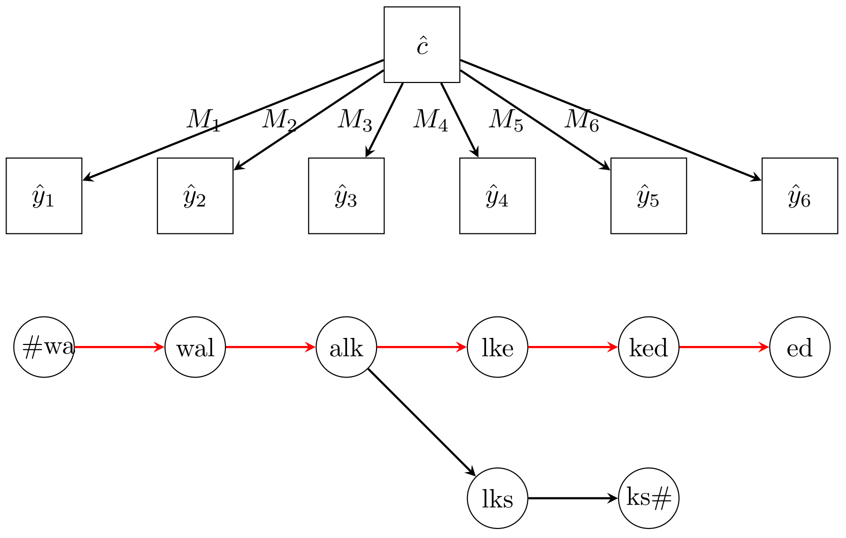
$$\boldsymbol{\hat{y}}_5 = \begin{array}{c} \begin{array}{ccccccccc} \text{\#wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \end{array} \\ \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 & 0.0 \end{bmatrix} \end{array} \tag{28}$$

$$\boldsymbol{\hat{y}}_6 = \begin{array}{c} \begin{array}{ccccccccc} \text{\#wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} \end{array} \\ \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \end{array} \tag{29}$$

Note that at position 4 and 5 (27, 28), there are two candidate trigrams so that paths can fork, potentially leading to multiple word forms.

Finally, after gathering all the candidate trigrams at different positions, we then construct the paths according to the adjacency matrix (cf. 12). In total, two paths can be found, giving us the candidate forms *walks* and *walked*. We then follow the same evaluation procedure described in Section 2.4, (i.e., synthesis-by-analysis) to select the best form. The algorithm of `learn_paths` is summarized in Figure 6, and the pseudo-code is provided in 1.

Figure 6: Graphical illustration of the procedure conducted in `learn_paths` for the word *walk*. The lower part of the figure presents all possible paths, and the one selected by the model is marked in red.



3.3.2 The `build_paths` algorithm

Like `learn_paths`, `build_paths` finds paths based on positional trigrams. However, the positional information is taken into account to a lesser extent in `build_paths`. This is because instead of learning to predict positional trigrams (as done via the $\boldsymbol{M}_t$ matrices in `learn_paths`), the candidate trigrams considered in `build_paths` are gathered from a shortlist of k nearest form neighbors. Positional information is therefore implied in the candidate trigrams, and path-finding can quickly tune into the existing forms of the neighbors. Although hardly any novel paths will be found in this way, it however provides a method to evaluate model performances more efficiently.

To illustrate, again we take the PAST tense and SINGULAR form (WALKED). We first select the form neighbors of this word by correlating its predicted form

Algorithm 1: Pesudo-code for `learn_paths` algorithms.

```
input  : $dataset$, $C$, $\hat{C}$, $A$ and $threshold$
output: A list of all candidate paths
paths_working, paths_working_tmp, paths_completed;
/* in order to save memery, $Y_t$, $M_t$ and $\hat{Y}_t$ are initialized
   and dumped at each timestep                              */
/* initialize paths in timestep 1                           */
make $Y_1$ from $dataset$ ;
learn $M_1$ from $C$ and $Y_1$;
predict $\hat{Y}_1$ form $\hat{C}$ and $M_1$ ;
for $i \leftarrow 1$ to $max\_data$ do
    collect cues for all cues $support > threshold$;
    for $c \in$ cues do
        if $c$ has start boundary then
            if $c$ also has end boundary then add $c \rightarrow$ paths_completed[$i$];
            else add $c \rightarrow$ paths_working[$i$];
        end
    end
end
/* construct paths from timestep 2 to the end               */
for $t \leftarrow 2$ to $max\_timestep$ do
    make $Y_t$ from $dataset$ ;
    learn $M_t$ from $C$ and $Y_t$;
    predict $\hat{Y}_t$ form $\hat{C}$ and $M_1$ ;
    for $i \leftarrow 1$ to $max\_data$ do
        collect cues for all cues $support > threshold$;
        while paths_working is not empty do
            $path \leftarrow$ dequeue(paths_working) ;
            for $c \in$ cues do
                if $c$ is attachable to $path$ then // look up in $A$
                    attach $c \rightarrow path$ ;
                    if $c$ has end boundary then add
                      $path \rightarrow$ paths_completed[$i$];
                    else add $path \rightarrow$ paths_working_tmp[$i$];
                end
            end
        end
        move paths_working_tmp $\rightarrow$ paths_working ;
    end
end
```

vector $\hat{\boldsymbol{c}}$ with all the gold standard form vectors in the dataset ($\boldsymbol{C}$). As shown in Table 4, when the number of nearest neighbors is set to 3 ($k = 3$), we identify three form neighbors (two forms are identical) for $\hat{\boldsymbol{c}}$, i.e., *walks* and *walked*.

Table 4: The correlations of $\hat{\boldsymbol{c}}$ with the form vectors of all the other words in the dataset. The selected form neighbors are marked in red.

	$\boldsymbol{c}_{walk}$	$\boldsymbol{c}_{walks}$	$\boldsymbol{c}_{walked_{past}}$	$\boldsymbol{c}_{walked_{part}}$
$\hat{\boldsymbol{c}}$	-0.40	0.40	0.50	0.50

There are in total eight unique trigrams in the two form neighbors, which are *#wa*, *wal*, *alk*, *lks*, *ks#*, *lke*, *ked*, and *ed#*. All eight trigrams are then considered at each position, and the algorithm again looks for all possible paths. Similar to the `learn_paths` algorithm presented in the previous section, here `build_paths` also finds two paths, giving us the candidate forms *walks* and *walked*. Following the same selection procedure (i.e., synthesis-by-analysis), the model selects *walked* as the predicted form. Figure 7 summarizes how `build_paths` finds the paths, and the pseudo-code is provided in 2.

For comparison, we run the three path-find algorithms on a French dataset (up to 12000 words) on a personal computer and the results are presented in Table 9. As can be seen, `learn_paths` and `build_paths` by far outperform the graph-based algorithm in terms of computation time, with the same accuracy level retained.

Table 5: Processing time of the three path-finding algorithms applied to a French dataset with up to 12000 words. The tests were run on a mid-2017 MacBook Pro with 2.9 GHz Quad-Core Intel Core i7 processor. Time and accuracy are measured with default threshold (0.1). The model does better with a lower threshold but takes more time to process.

Algorithm	Time (minutes)	Accuracy
graph-based	255.64	0.992
`learn_paths`	7.01	0.977
`build_paths`	4.90	0.999

4 Other functionalities in JudiLing

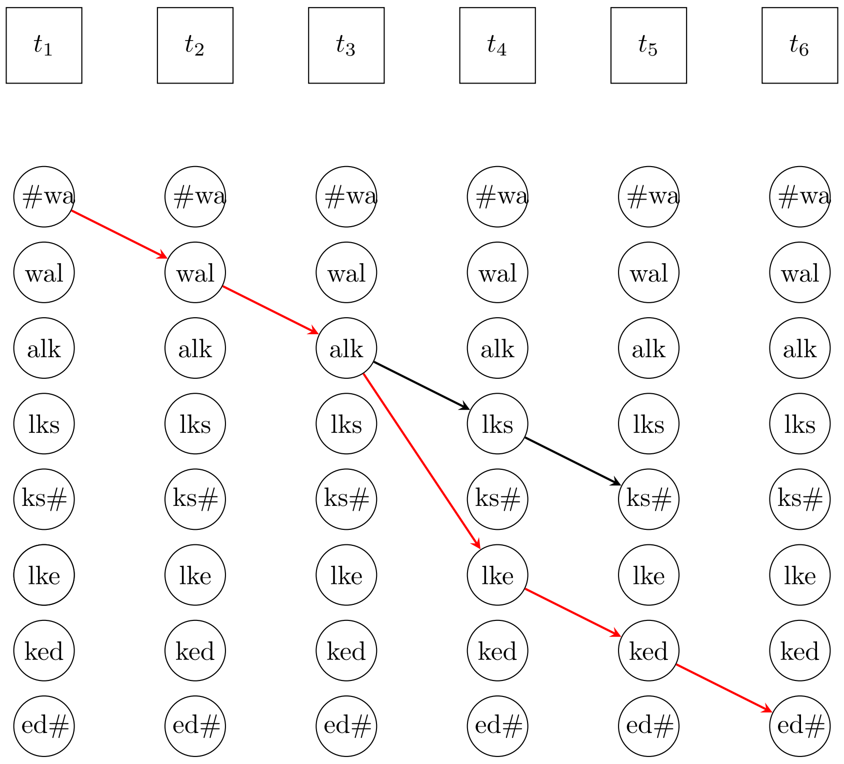
4.1 cross validation

To test our model's productivity, we also designed functions for conducting cross-validation. In this section we introduce two cases of cross-validation, again with our toy example. Depending on the nature of the validation dataset, different functions and parameter settings are required. In one situation, if the training and validation datasets are carefully separated so that no novel

Algorithm 2: Pesudo-code for `build_paths` algorithms.

```
input  : $dataset$, $C$, $\hat{C}$, $A$ and $k$
output: A list of all candidate paths
paths_working, paths_working_tmp, paths_completed;
collect cues within $k$ nearest form neighbors ;
/* initialize paths in timestep 1                          */
for $i \leftarrow 1$ to $max\_data$ do
    for $c \in$ cues[$i$] do
        if $c$ has start boundary then
            if $c$ also has end boundary then add $c \rightarrow$ paths_completed[$i$];
            else add $c \rightarrow$ paths_working[$i$];
        end
    end
end
/* construct paths from timestep 2 to the end              */
for $t \leftarrow 2$ to $max\_timestep$ do
    for $i \leftarrow 1$ to $max\_data$ do
        while paths_working is not empty do
            $path \leftarrow$ dequeue(paths_working) ;
            for $c \in$ cues[$i$] do
                if $c$ is attachable to path then // look up in $A$
                    attach $c \rightarrow path$ ;
                    if $c$ has end boundary then add
                     $path \rightarrow$ paths_completed[$i$];
                    else add $path \rightarrow$ paths_working_tmp[$i$];
                end
            end
        end
        move paths_working_tmp $\rightarrow$ paths_working ;
    end
end
```

Figure 7: Graphical illustration of the procedure conducted in `build_paths` for the word *walk*. The lower part of the figure presents all possible paths, and the one selected by the model is marked in red.



cues or novel semantic features are present in the validation dataset, cross-validation can be straightforwardly conducted with some modification of the basic code. On the other hand, if the validation dataset contains novel cues or semantic features, some further measures will be required. In what follows, we will explain the methodological procedures, and detailed code will be given in Section 5 (worked examples 5.2 and 5.3).

Previously in section 2.1, our toy lexicon included four words, *walk*, *walks*, *walked* (past) and *walked* (participle). Now we add four new words, which are *talk*, *talks*, *talked* (past) and *talked* (participle). We also add one lexome for TALK (see (30)). To do cross-validation, we will first need to divide the words into a training and a validation dataset. For the convenience of illustration we include five words in the training data: *walks*, *walked* (past), *walked* (participle), *talked* (past) and *talked* (participle) (Table 6). In the first situation, the validation data contains the word *talks*. As shown in the matrices of C_{train1} and C_{val1}, all letter trigrams of *talks* are already present in the letter trigram set of the training data.

$$L = \begin{array}{c} \\ \text{WALK} \\ \text{PRESENT} \\ \text{PAST} \\ \text{SINGULAR} \\ \text{PARTICIPLE} \\ \text{TALK} \end{array} \begin{array}{c} \begin{array}{rrrrrr} \text{S1} & \text{S2} & \text{S3} & \text{S4} & \text{S5} & \text{S6} \end{array} \\ \left[\begin{array}{rrrrrr} -1.52 & -0.69 & 0.05 & -0.31 & 1.60 & 0.23 \\ -0.92 & -0.86 & 1.30 & 0.01 & 0.22 & -0.58 \\ -0.69 & 0.98 & 2.94 & -0.06 & 1.10 & 2.10 \\ -0.01 & 0.69 & -0.26 & -0.56 & -0.89 & -1.09 \\ -1.37 & -0.98 & 0.81 & 0.01 & 0.56 & 0.31 \\ 2.15 & -0.77 & 0.91 & 0.59 & 0.81 & 1.53 \end{array}\right] \end{array} \tag{30}$$

Table 6: Training and validation data.

Training data	**Validation data 1** **no unseen cues**	**Validation data 2** **unseen cues**
walks, walked$_{past}$, walked$_{part}$ talked$_{past}$, talked$_{part}$	talks	talk

$$\boldsymbol{C}_{train1} = \begin{array}{c|cccccccccc} & \#wa & wal & alk & lks & ks\# & lke & ked & ed\# & \#ta & tal \\ \hline \text{walks} & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ \text{walked}_{past} & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ \text{walked}_{part} & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ \text{talked}_{past} & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ \text{talked}_{part} & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{array}$$

$$\boldsymbol{C}_{val1} = \begin{array}{c|cccccccccc} & \#wa & wal & alk & lks & ks\# & lke & ked & ed\# & \#ta & tal \\ \hline \text{talks} & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{array}$$

In the second case, the word *talk* is held out for the validatation data. Different from *talks*, *talk* has one novel letter trigram that is absent in the training data, which is *lk#*. Given that it is impossible for the model to predict the form of this word correctly without this trigram, when constructing the $\boldsymbol{C}$ matrix, we reserve a place-holder for this novel cue. In $\boldsymbol{C_{train2}}$, the novel cue *lk#* receives all zeros, as no words in the training data have this cue. Now with the addition of this novel cue, the form of *talk* can then be properly represented ($\boldsymbol{C_{val2}}$). As will become clearer later, this step is crucial for the model to construct the form.

$$\boldsymbol{C}_{train2} = \begin{array}{c|ccccccccccc} & \text{\#wa} & \text{wal} & \text{alk} & \text{lk\#} & \text{lks} & \text{ks\#} & \text{lke} & \text{ked} & \text{ed\#} & \text{\#ta} & \text{tal} \\ \hline \text{walks} & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ \text{walked}_{past} & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ \text{walked}_{part} & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ \text{talked}_{past} & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ \text{talked}_{part} & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{array}$$

$$\boldsymbol{C}_{val2} = \begin{array}{c} \\ \text{talk} \end{array} \left[\begin{array}{ccccccccccc} \#\text{wa} & \text{wal} & \text{alk} & \text{lk}\# & \text{lks} & \text{ks}\# & \text{lke} & \text{ked} & \text{ed}\# & \#\text{ta} & \text{tal} \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array} \right]$$

The procedure of cross-validation goes as follows. For comprehension, we first obtain the network $\boldsymbol{F}$ with the cue and semantic matrices of the training data, following the formulae given by equations (31) and (32). Next by multiplying the cue matrix of the validation data with $\boldsymbol{F}$, we can obtain the predicted semantic matrix for the validation words.

$$\boldsymbol{C_{train}F} = \boldsymbol{S_{train}}, \tag{31}$$
$$\boldsymbol{C_{val}F} = \boldsymbol{\hat{S}_{val}}. \tag{32}$$

Since there is only one word for each evaluation, the comprehension evaluation is not necessary here. However, we can still take the correlations between two validation words *talks* and *talk* which are 1 and 0.91.

For production, we obtain the predicted form matrix for the validation words with equations (33) and (34).

$$\boldsymbol{S_{train}G} = \boldsymbol{C_{train}}, \tag{33}$$
$$\boldsymbol{S_{val}G} = \boldsymbol{\hat{C}_{val}}. \tag{34}$$

For the first validation data, only (and all) the trigams of *talks* receive high support from its semantics.

$$\boldsymbol{\hat{c}}_{talks} = -0.0 \left[\begin{array}{cccccccccc} \#wa & wal & alk & lks & ks\# & lke & ked & ed\# & \#ta & tal \\ -0.0 & 1.0 & 0.0 & 1.0 & 1.0 & -0.0 & -0.0 & -0.0 & 1.0 & 1.0 \end{array} \right]$$

It is therefore is easy for the path-finding algorithm to derive the correct path, hence to predict the form correctly. For the second validation with the novel cue, however, the crucial target trigam *lk#* of *talk* does not receive any semantic support due to the lack of training, as can be seen from its predicted vector shown below:

$$\boldsymbol{\hat{c}}_{talk} = \left[\begin{array}{ccccccccccc} \#\text{wa} & \text{wal} & \text{alk} & \text{lk}\# & \text{lks} & \text{ks}\# & \text{lke} & \text{ked} & \text{ed}\# & \#\text{ta} & \text{tal} \\ -0.43 & -0.43 & 0.61 & 0.0 & 0.53 & 0.53 & 0.08 & 0.08 & 0.08 & 1.04 & 1.04 \end{array} \right].$$

If we go on to run the path-finding algorithm, the correct path can never be formed because the trigram *lk#* will be excluded by the thresholding mechanism (cf. Section 3.3.1). We therefore introduce a second thresholding mechanism, which is referred to as the “tolerance” mode. The “tolerance” mode allows a pre-defined number of cues with low supports into the path-finding process. In this way, these cues will still be available for path construction. By setting the tolerance threshold to a very small value (e.g., -1), the algorithm now needs to consider more trigram cues. That is, in addition to the cues that receive supports higher than the general threshold, cues whose supports are higher than the tolerance threshold will be taken into account as well. In our example, these "wildcard" trigram cues are *lk#*, *lke*, *ked* and *ed#*. With these cues together, the algorithm can now construct to candidate paths: *talk* and *talks*. As a final step, the "synthesis-by-analysis" (cf. Section 2.4) ultimately picks *talk* to be the model's prediction (Table 7).

Table 7: "Synthesis-by-analysis" for word *talk*.

	$\boldsymbol{s_{talk}}$	selected	correct
$\boldsymbol{\hat{s}_{talk}}$	0.91	✓	✓
$\boldsymbol{\hat{s}_{talks}}$	0.86	✗	✗

4.2 incremental learning

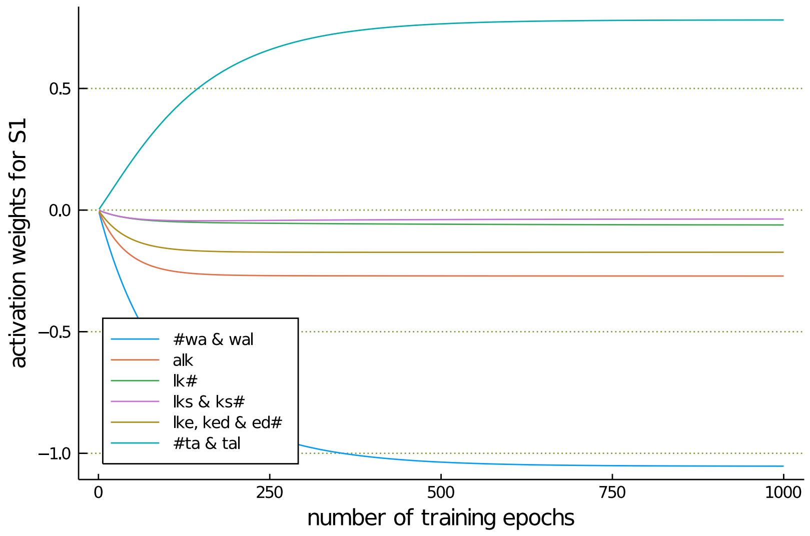
The matrix solution as described thus far provides an efficient way to estimate the networks. It, however, has some limitations. The first limitation is that even with Cholesky Decomposition, it is still computationally costly to calculate the transformation matrix when the dataset is really big. For example, it took about 13 minutes to compute the comprehension mapping $\boldsymbol{F}$ for a dataset with 104 thousands Finnish nouns forms. This creates difficulties for data exploration and analyses. Furthermore, as the matrix estimates the endstate of learning, a theoretical state where any further learning only results in negligible weight changes, we miss out on the detailed information for the learning trajectory. An analogy with human language learning is that we can only look at adults' linguistic system, being unable to know how children acquire languages. To overcome these limitations, in the `JudiLing` package we also implemented incremental learning by using the Widrow-Hoff learning rule, the pseudo-code of which is provided below.

With incremental learning, now we can trace weight changes in the network across the time course. By way of example, using the toy example described in Section 4.1, we presented 1000 learning events (with each word form appearing roughly the same number of times) to the model, with learning rate set to 0.001. During training, we traced the changes of weights between the cues (trigams) and the outcomes (semantic dimensions). The development of weights between all trigrams to the first semantic dimension (S1) is shown in Figure 8. As can be seen, for some cues (e.g., *#wa* and *wal*), the weights connected to S1 gradually increase. For other cues (e.g., *#ta* and *tal*), by contrast, the weights decrease. Notably, after the 500th training epoch, the trajectory curves of all weights asymptote, and all weight changes are minimum. In fact, if we train the model enough epochs (over 200,000), the weights are almost identical to the values obtained by using the matrix solution.

Algorithm 3: Pseudo-code for `Widrow-Hoff` learning rules implemented in JudiLing.

```
input  : $X$, $Y$, $init\_weights$ and $eta$
output: The transformation weight.
$weights \leftarrow init\_weights$;
for $x, y \leftarrow zip(X, Y)$ do
    $preds \leftarrow x * init\_weights$ ;
    $obsv \leftarrow y - preds$ ;
    $updates \leftarrow eta * (x^t * obsv)$ ;
    $weights \leftarrow weights + updates$ ;
end
```

Figure 8: The development of weights between all trigram cues and the first semantic dimension (S1) over time.



5 Worked examples

In this section, we present three worked examples: Latin, Estonian and French. The Latin and Estonian datasets have been previously studied, and the modeling results using `WpmWithLdl` are presented in Baayen et al. (2018a) and Chuang et al. (2020b). The French dataset was constructed by Modeling French Verbs with Discriminative Learning Bachelor Thesis submitted by Marei Beukman, Tuebingen, 2020, where preliminary modeling results with `JudiLing` were reported as well.

In what follows, we are going to demonstrate how to use the `JudiLing` package with these three datasets. All modeling was run on a server with Intel(R) Xeon(R) CPU E5-4620 v4 @ 2.10GHz of 20 cores and 1,000GB memroy. Detailed code for each dataset is given in the following sections. As an overview, Table 8 and 9 present a comparison of comprehension and production accuracies obtained with `WpmWithLdl` and `JudiLing`. As can be seen, the accuracies are comparable between the two packages. With respect to computing time, the computation time is substantially reduced with `JudiLing` (Table 9). The computation time of comprehension model is also reduced significantly, but since comprehension processing is already fast using `WpmWithLdl`, speed gain from comprehension side is less noticeable. Moreover, specifically for the validation data (Estonian), the `learn_paths` result is even better than the result obtained with the igraph algorithm provided in `WpmWithLdl`. It is noteworthy that `buile_path` does not work well for the validation dataset. This is because `build_path` depends heavily on the form neighbors of the target words. As the predicted form vector for the target word in the validation data is usually not close to the targeted form vector, the form neighbors obtained based on the suboptimal predicted form vector are therefore also not accurate. This then leads to failure of finding the correct path for the targeted form.

In the following examples, the modeling steps are similar: 1) make cue matrix or matrices; 2) make semantic matrix or matirces; 3) solve transformation matrix F; 4) evaluate comprehension model; 5) solve transformation matrix G; 6) run through path finding algorithms; 7) evaluate production model. All three examples build semantic matrices with simulated distributional semantic vectors as we implemented in `WpmWithLdl)`, but it is also acceptable by `JudiLing` if semantic matrix is already made by other sources.

With the Estonian and French datasets, we illustrate how to do cross-validation (thus in both cases the data is split into training and validation data). For the former, the splitting is carefully done, so that no novel trigram cues or semantic features are present in the validation data. For the latter, word forms are randomly assigned to the training and validation data. We will show how to conduct cross-validation with these two different types of validation data.

Table 8: Comprehension accuracy for Latin, Estonian and French datasets.

Datasets	**WpmWithLdl**	**JudiLing**
Latin	1.000	1.000
Estonian$_{train}$	0.992	0.992
Estonian$_{val}$	0.975	0.983
French$_{train}$	N/A	0.997
French$_{val}$	N/A	0.950

Table 9: Processing time and production accuracy for Latin, Estonian and French datasets.

	WpmWithLdl		`learn_paths`		`build_paths`	
Datasets	**Time**	**Acc**	**Time**	**Acc**	**Time**	**Acc**
Latin	14.8 s	0.997	0.7 s	0.998	0.2 s	0.997
Estonian$_{train}$	4.6 m	0.916	43.1 s	0.988	5.6 s	0.975
Estonian$_{val}$	$>$ 1 d	0.695	48.2 s	0.899	8.7 s	0.460
French$_{train}$	N/A	N/A	82.4 m	0.982	25.3 m	0.998
French$_{val}$	N/A	N/A	20.6 h	0.669	27.5 m	0.082

5.1 Latin

The Latin dataset contains 8 lexemes with 5 different inflectional features: *Person*, *Number*, *Tense*, *Voice* and *Mood*, resulting in total of 672 data records. Table 10 shows different forms of lexeme "vocare" in different Person and Number features.

Table 10: 6 forms of Lexeme "vocare" in different Person and Number

Word	Lexeme	Person	Number	Tense	Voice	Mood
vocoo	vocare	p1	sg	present	active	ind
vocaas	vocare	p2	sg	present	active	ind
vocat	vocare	p3	sg	present	active	ind
vocaamus	vocare	p1	pl	present	active	ind
vocaatis	vocare	p2	pl	present	active	ind
vocant	vocare	p3	pl	present	active	ind

We assume that the dataset is already in a subfolder `data` under the project directory. The first step is to load CSV file into a dataframe.

```
using JudiLing # our package
using CSV # read csv files into dataframes

# load latin file
latin = CSV.DataFrame!(CSV.File(joinpath(
  @__DIR__, "data", "latin.csv")))
```

Then we are going to make cue matrix and semantic matrix for the dataset. In the example, we choose to use letter trigrams as cues, and their semantic vectors are simulated vectors composed by 6 semantic features: *Lexeme*, *Person*, *Number*, *Tense*, *Voice* and *Mood*. Also, we set the dimension of semantic vectors equal to the total number of cues.

```
# create C matrixes for Latin
cue_obj = JudiLing.make_cue_matrix(
  latin,
  grams=3,
  target_col=:Word,
  tokenized=false,
  keep_sep=false
  )

# retrieve dim of C
# we set the S matrixes as the same dimensions
n_features = size(cue_obj.C, 2)

# create S matrix for Latin
S = JudiLing.make_S_matrix(
  latin,
  ["Lexeme"],
  ["Person","Number","Tense","Voice","Mood"],
  ncol=n_features,
  add_noise=true)
```

The next step is to learn the transform mapping ***F*** and ***G***. Currently, we use Cholesky Decomposition to invert the matrix and calculate ***F*** and ***G***, as discussed in section 2.2.

```
# we use cholesky function to calculate mapping G from S to C
G = JudiLing.make_transform_matrix(S, cue_obj.C)
# we calculate F as we did for G
F = JudiLing.make_transform_matrix(cue_obj.C, S)
```

After obtaining ***F***, the comprehension model is complete. We can evaluate the accuracy with `eval_SC` function. Note that as the Latin dataset contains homophones, here we have options of whether recognizing the meaning of one's homophonic counterpart should be considered correct or incorrect.

```
Shat = cue_obj.C * F
@show JudiLing.eval_SC(Shat, S) # 0.991
@show JudiLing.eval_SC(Shat, S, latin, :Word) # 1.0
```

If we count homophones as correct predictions, then the accuracy of the comprehension model is 100%.

Let's now turn to the production model. In order to glue cues together, we need an adjacency matrix which specifies all the possible continuations of a given cue. There are several approaches making adjacency matrix. Here we use the default adjacency matrix created along with the cue matrix by the `make_cue_matrix` function, in which only attested transitions are allowed. Another way of constructing adjacency matrices is introduced in the Section 5.2. For the path finding part, we use both `learn_path` and `build_path`. Here we have the option to request the function to return information about the gold path, i.e., the correct path of the targeted form. The code is presented below.

```
Chat = S * G

# here we only use a adjacency matrix as we got it
# from the training dataset
A = cue_obj.A

# we calculate how many timestep we need for
# learn_paths and build_paths function
max_t = JudiLing.cal_max_timestep(latin, :Word)

# we apply the learn_paths and build_paths functions
res_learn, gpi_learn = JudiLing.learn_paths(
  latin,
  latin,
  cue_obj.C,
  S,
  F,
  Chat,
  A,
  cue_obj.i2f,
  cue_obj.f2i,
  check_gold_path=true,
  gold_ind=cue_obj.gold_ind,
  Shat=Shat,
  max_t=max_t,
  max_can=10,
  grams=3,
  threshold=0.1,
  tokenized=false,
  keep_sep=false,
  target_col=:Word,
  verbose=true)

res_build = JudiLing.build_paths(
    latin,
    cue_obj.C,
    S,
    F,
    Chat,
    A,
    cue_obj.i2f,
    cue_obj.gold_ind,
    max_t=max_t,
    n_neighbors=3,
    verbose=true
    )
```

```
acc_learn = JudiLing.eval_acc(
  res_learn,
  cue_obj.gold_ind,
  verbose=false
)

acc_build = JudiLing.eval_acc(
  res_build,
  cue_obj.gold_ind,
  verbose=false
)

println("Acc for learn: $acc_learn") # 0.998
println("Acc for build: $acc_build") # 0.997
```

The accuracies for `learn_path` and `build_path` are high. The `learn_path` result contains one error, and that of `build_path` contains two errors. Further analyses on the errors show that the correct paths are found by both `learn_paths` and `build_paths`, but the errors turn out to have slightly higher correlations with the gold standard semantic vector than the correct paths, according to "synthesis-by-analysis" (Table 12 and 13). Since both path-finding algorithms use the same comprehension network $\boldsymbol{F}$, the correlation values of both algorithms are therefore identical.

Table 11: Errors of two JudiLing path finding algorithms and WpmWithLdl.

	Prediction	**Gold Label**
WpmWithLdl	curriaaris	curraaris
	curreereris	currereeris
learn_path	sapieebar	sapiar
build_path	sapieebar	sapiar
	sapieebam	sapiam

Table 12: Candidate paths found by `learn_path` and their "synthesis-by-analysis" support.

Golden Label	**Prediction**	**Support**
sapiar	sapieebar	0.853
	sapiar	0.850
	sapieemur	0.826
	sapieeris	0.809
	sapientur	0.701
	sapiirer	0.611

Table 13: Candidate paths found by `build_path` and their "synthesis-by-analysis" support.

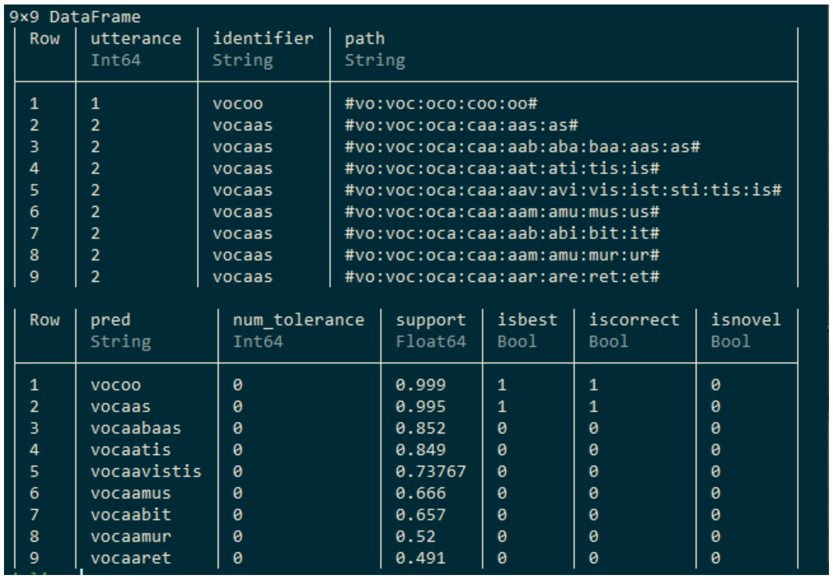
Golden Label	Prediction	Support
sapiar	sapieebar	0.853
	sapiar	0.850
sapiam	sapieebam	0.867
	sapiam	0.866

Finally we can save both the results into CSV files for further analysis. This can be done for both the general results and the gold path results. The former contains detailed information about the candidate paths of a given word form, including "synthesis-by-analysis" support, accuracy, and whether prediction is a novel form. A screenshot of this output is presented in Figure 9. On the other hand, the latter tells us not only the "synthesis-by-analysis" support of the gold path, but also the semantic support that each trigram obtains at time *t* (Figure 10).

```
JudiLing.write2csv(
  res_learn,
  latin,
  cue_obj,
  cue_obj,
  "latin_learn_res.csv",
  grams=3,
  tokenized=false,
  sep_token=nothing,
  start_end_token="#",
  output_sep_token="",
  path_sep_token=":",
  target_col=:Word,
  root_dir=@__DIR__,
  output_dir="latin_out"
  )

JudiLing.write2csv(
  gpi_learn,
  "latin_learn_gpi.csv",
  root_dir=@__DIR__,
  output_dir="latin_out"
  )
```

Figure 9: A screenshot of results produced by JudiLing for Latin example.



```
9×9 DataFrame
 Row │ utterance  identifier  path
     │ Int64      String      String
─────┼──────────────────────────────────────────────────────────────
   1 │ 1          vocoo       #vo:voc:oco:coo:oo#
   2 │ 2          vocaas      #vo:voc:oca:caa:aas:as#
   3 │ 2          vocaas      #vo:voc:oca:caa:aab:aba:baa:aas:as#
   4 │ 2          vocaas      #vo:voc:oca:caa:aat:ati:tis:is#
   5 │ 2          vocaas      #vo:voc:oca:caa:aav:avi:vis:ist:sti:tis:is#
   6 │ 2          vocaas      #vo:voc:oca:caa:aam:amu:mus:us#
   7 │ 2          vocaas      #vo:voc:oca:caa:aab:abi:bit:it#
   8 │ 2          vocaas      #vo:voc:oca:caa:aam:amu:mur:ur#
   9 │ 2          vocaas      #vo:voc:oca:caa:aar:are:ret:et#

 Row │ pred          num_tolerance  support  isbest  iscorrect  isnovel
     │ String        Int64          Float64  Bool    Bool       Bool
─────┼──────────────────────────────────────────────────────────────────
   1 │ vocoo         0              0.999    1       1          0
   2 │ vocaas        0              0.995    1       1          0
   3 │ vocaabaas     0              0.852    0       0          0
   4 │ vocaatis      0              0.849    0       0          0
   5 │ vocaavistis   0              0.73767  0       0          0
   6 │ vocaamus      0              0.666    0       0          0
   7 │ vocaabit      0              0.657    0       0          0
   8 │ vocaamur      0              0.52     0       0          0
   9 │ vocaaret      0              0.491    0       0          0
```

5.2 Estonian

The Estonian dataset contains over 5,000 words nouns with different cases and numbers, as can be seen in Table 14. Unlike the Latin example, where we trained and evaluated the entire dataset, for this dataset we are going to train the model with a portion of data (5154), and evaluate it with the rest of data (1288). The data was carefully split into the training and validation datasets, so that no novel cues and semantic features are present in the validation dataset.

Table 14: 4 Estonian word forms with their cases and numbers specified.

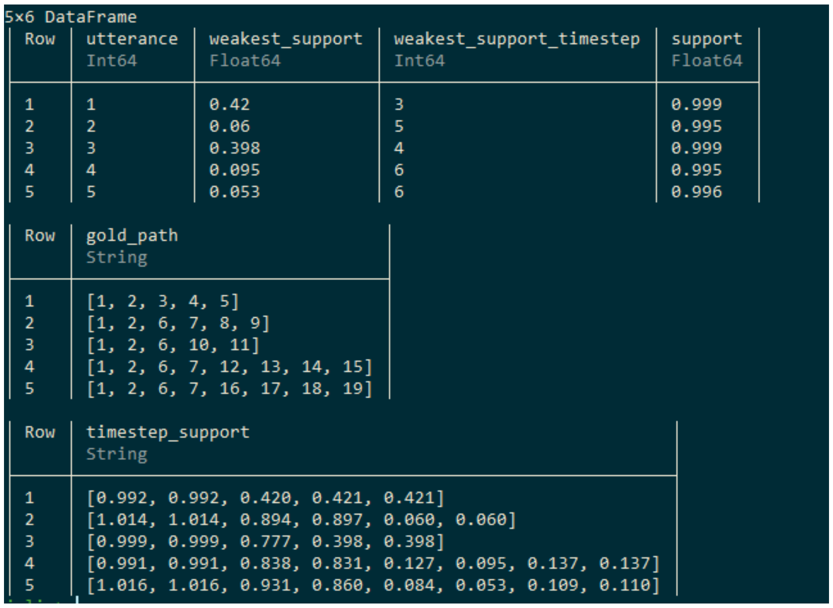
Word	**Lexeme**	**Case**	**Number**
ümbrikul	ümbrik	ad	sg
rüüdele	rüü	all	pl
jõesse	jõgi	ill	sg
koer	koer	nom	sg

The first step is to load the datasets, but this time we need to load both the training and validation datasets.

```
using JudiLing # our package
using CSV # read csv files into dataframes

# load estonian files
```

Figure 10: A screenshot of gold path information produced by JudiLing for Latin example.



```
5×6 DataFrame
 Row │ utterance  weakest_support  weakest_support_timestep  support
     │ Int64      Float64          Int64                     Float64
─────┼───────────────────────────────────────────────────────────────
   1 │ 1          0.42             3                         0.999
   2 │ 2          0.06             5                         0.995
   3 │ 3          0.398            4                         0.999
   4 │ 4          0.095            6                         0.995
   5 │ 5          0.053            6                         0.996

 Row │ gold_path
     │ String
─────┼─────────────────────────────
   1 │ [1, 2, 3, 4, 5]
   2 │ [1, 2, 6, 7, 8, 9]
   3 │ [1, 2, 6, 10, 11]
   4 │ [1, 2, 6, 7, 12, 13, 14, 15]
   5 │ [1, 2, 6, 7, 16, 17, 18, 19]

 Row │ timestep_support
     │ String
─────┼─────────────────────────────────────────────────────────────
   1 │ [0.992, 0.992, 0.420, 0.421, 0.421]
   2 │ [1.014, 1.014, 0.894, 0.897, 0.060, 0.060]
   3 │ [0.999, 0.999, 0.777, 0.398, 0.398]
   4 │ [0.991, 0.991, 0.838, 0.831, 0.127, 0.095, 0.137, 0.137]
   5 │ [1.016, 1.016, 0.931, 0.860, 0.084, 0.053, 0.109, 0.110]
```

```
estonian_train = CSV.DataFrame!(CSV.File(
    joinpath(@__DIR__, "data", "estonian_train.csv")))
estonian_val = CSV.DataFrame!(CSV.File(
    joinpath(@__DIR__, "data", "estonian_val.csv")))
```

Next we construct the cue matrices and semantic matrices, for both training and validation datasets.

```
cue_obj_train, cue_obj_val = JudiLing.make_cue_matrix(
  estonian_train,
  estonian_val,
  grams=3,
  target_col=:Word,
  tokenized=false,
  keep_sep=false
  )

# retrieve dim of C
# we set the S matrixes as the same dimensions
n_features = size(cue_obj_train.C, 2)
```

```
S_train, S_val = JudiLing.make_S_matrix(
  estonian_train,
  estonian_val,
  ["Lexeme"],
  ["Case","Number"],
  ncol=n_features,
  add_noise=true)
```

The transformation matrix for comprehension is derived based on the cue and semantic matrices of the training datasets. We then obtain the predicted semantic matrices for both the training and validation data, and evaluate them accordingly.

```
F_train = JudiLing.make_transform_matrix(cue_obj_train.C, S_train)

Shat_train = cue_obj_train.C * F_train
Shat_val = cue_obj_val.C * F_train

# count homophones as incorrect
@show JudiLing.eval_SC(Shat_train, S_train) # 0.975
@show JudiLing.eval_SC(Shat_val, S_val) # 0.976

# count homophones as correct
@show JudiLing.eval_SC(
    Shat_train, S_train, estonian_train, :Word) # 0.992
@show JudiLing.eval_SC(
    Shat_val, S_val, estonian_val, :Word) # 983
```

As we can see, the accuracies of both training and validation data are very high especially when homophones are treated as correct predictions.

With respect to production, again we first obtain the predicted cue matrices for both the training and validation datatsets. We then run the path-finding algorithms, `learn_paths` and `build_paths`.

```
G_train = JudiLing.make_transform_matrix(S_train, cue_obj_train.C)
Chat_train = S_train * G_train
Chat_val = S_val * G_train

# here we only use a adjacency matrix as we got it
# from the training dataset
A_train = cue_obj_train.A

# we calculate how many timestep we need for
# both training and validation data
max_t = JudiLing.cal_max_timestep(
    estonian_train, estonian_val, :Word)
```

```
# we apply the learn_paths and build_paths functions
res_learn_train, gpi_learn_train = JudiLing.learn_paths(
  estonian_train,
  estonian_train,
  cue_obj_train.C,
  S_train,
  F_train,
  Chat_train,
  A_train,
  cue_obj_train.i2f,
  cue_obj_train.f2i,
  check_gold_path=true,
  gold_ind=cue_obj_train.gold_ind,
  Shat_val=Shat_train,
  max_t=max_t,
  max_can=10,
  grams=3,
  threshold=0.05,
  tokenized=false,
  keep_sep=false,
  target_col=:Word,
  verbose=true)

acc_learn_train = JudiLing.eval_acc(
  res_learn_train,
  cue_obj_train.gold_ind,
  verbose=false
)

res_learn_val, gpi_learn_val = JudiLing.learn_paths(
  estonian_train,
  estonian_val,
  cue_obj_train.C,
  S_val,
  F_train,
  Chat_val,
  A_train,
  cue_obj_train.i2f,
  cue_obj_train.f2i,
  check_gold_path=true,
  gold_ind=cue_obj_val.gold_ind,
  Shat_val=Shat_val,
  max_t=max_t,
  max_can=10,
  grams=3,
  threshold=0.05,
  is_tolerant=true,
  tolerance=-0.1,
  max_tolerance=3,
  tokenized=false,
  keep_sep=false,
  target_col=:Word,
  verbose=true)

acc_learn_val = JudiLing.eval_acc(
  res_learn_val,
  cue_obj_val.gold_ind,
  verbose=false
)

res_build_train = JudiLing.build_paths(
  estonian_train,
  cue_obj_train.C,
  S_train,
  F_train,
  Chat_train,
  A_train,
  cue_obj_train.i2f,
  cue_obj_train.gold_ind,
  max_t=max_t,
  n_neighbors=3,
  verbose=true
  )

acc_build_train = JudiLing.eval_acc(
  res_build_train,
  cue_obj_train.gold_ind,
  verbose=false
)

res_build_val = JudiLing.build_paths(
    estonian_val,
    cue_obj_train.C,
    S_val,
    F_train,
    Chat_val,
    A_train,
    cue_obj_train.i2f,
    cue_obj_train.gold_ind,
    max_t=max_t,
    n_neighbors=20,
    verbose=true
    )

acc_build_val = JudiLing.eval_acc(
  res_build_val,
  cue_obj_val.gold_ind,
  verbose=false
)

println("Acc for learn train: $acc_learn_train") # 0.988
println("Acc for learn val: $acc_learn_val")     # 0.801
println("Acc for build train: $acc_build_train") # 0.975
println("Acc for build val: $acc_build_val")     # 0.402
```

As expected, the accuracies of the training data are higher than those of the validation data. The cross-validation accuracy is increased from 69.5% to 80.1% compared with `WpmWithLdl` (Chuang et al., 2020b). In fact, the accuraies of the validation data also hinges on how the adjacency matrix is constructed. Here we used the default setting, in which only the attested connections are allowed. To increase the cross-validation accuracy further, we can also build the adjacency matrix in such a way that we want the model to be prepared for all possible connections. To do so, we re-construct the "full" adjacency matrix with the following code:

```
A = JudiLing.make_combined_adjacency_matrix(
  estonian_train,
  estonian_val,
  grams=3,
  target_col=:Word,
  tokenized=false,
  keep_sep=false
  )
```

With the full adjacency matrix, the accuracy of `learn_paths` increases by almost 10% (Table 15), although that of `build_paths` hardly changes. A further examination of the errors shows that for `learn_paths`, among all the 256 errors, 83 (32.4%) of them have the targeted forms in the candidate lists. As to `build_paths`, the correct paths are in the candidate lists for only 2.3% of the errors. As explained previously, the success of `build_paths` depends on the "seen" form neighbors (i.e., word forms in the training data). That is, if a given trigram cue is not found in any of the form neighbors, there is no way that the algorithm can find the correct path, even with the full adjacency matrix. The productivity of `build_paths` is therefore much more restricted.

5.3 French

The French dataset contains 21,306 short phrases, such as *j'abandonne*. Each phrase is specified with the following semantic features: *Lexeme*, *Tense*, *Aspect*,

Table 15: Results of `learn_paths` (lp), `build_paths` (bp), `learn_paths` with full adjacency matrix (lp_full) and `build_paths` with full adjacency matrix (bp_full), in Estonian unseen data and trained data.

	lp	**bp**	**lp_full**	**bp_full**
trained data	**0.988**	0.975	**0.988**	0.975
unseen data	0.801	0.402	**0.899**	0.406

Person, *Number*, *Gender*, *Class* and *Mood*. For each form, we also have its phone representation, with syllable boundary indicated by "-". An example of the coding of one short phrase is presented in Table 16. The dataset also contains more complex phrases with auxiliary verbs such as "qu'elles eussent abandonné" and "que tu eusses abandonné".

Table 16: Example of "j'abandonne" in French dataset

	Orthography	j'abandonne
Cues	**Syllables**	Za-b@-dOn
Semantics	**Lexeme**	abandonner
	Tense	present
	Aspect	n/a
	Person	p1
	Number	singular
	Gender	common
	Class	action
	Mood	indicative

For modeling set-up, we first import all necessary packages and load the dataset into a dataframe.

```julia
using JudiLing # our package
using CSV # read csv files into dataframes
using Random # shuffle the dataset

# load french file
french = CSV.DataFrame!(CSV.File(joinpath(
  @__DIR__, "data", "french.csv")))
```

Unlike the Estonian example, where we have the training and validation data separated, this time we randomly split French dataset into 20,306 training data and 1,000 validation data.

```julia
# randomly reorder the data
rng = MersenneTwister(314)
french = french[shuffle(rng, 1:size(french, 1)),:]
```

```
# split the dataset
tv = 1000
french_train = french[1:end-tv,:]
french_val = french[end-tv+1:end,:]
```

As the dataset is randomly split into the training and validation datasets, it is expected that there are novel cues (here we use syllables bigrams) and semantic features in the validation dataset. To address this issue, we reserve placeholders for these unseen cues and semantic features in the cue and semantic matrices of the training data. This is done with the functions `make_combined_cue_matrix` and `make_combined_S_matrix`.

```
# create cue matrices for
# both training and validation datasets
# while holding unseen cues as 0
cue_obj_train, cue_obj_val = JudiLing.make_combined_cue_matrix(
  french_train,
  french_val,
  grams=2,
  target_col=:Syllables,
  tokenized=true,
  sep_token="-",
  keep_sep=true
  )

# retrieve dim of C
# we set the S matrixes as the same dimensions
n_features = size(cue_obj_train.C, 2)

# create semantice matrices for
# both training and validation datasets
# also make lexome matrix for unseen semantice features
S_train, S_val = JudiLing.make_combined_S_matrix(
  french_train,
  french_val,
  ["Lexeme"],
  ["Person", "Number", "Gender", "Tense", "Aspect", "Class", "Mood"],
  ncol=n_features,
  add_noise=true)
```

The code for training and evaluating the comprehension network is presented below.

```
# we use cholesky function to calculate mapping F from C to S
F_train = JudiLing.make_transform_matrix(cue_obj_train.C, S_train)

# the model produces predictions for
# both training and validation datasets
Shat_train = cue_obj_train.C * F_train
Shat_val = cue_obj_val.C * F_train

# evaluation taking homophones as incorrect
@show JudiLing.eval_SC(Shat_train, S_train) # 0.941
@show JudiLing.eval_SC(Shat_val, S_val) # 0.913

# evaluation taking homophones as correct
@show JudiLing.eval_SC(Shat_train, S_train, french_train,
  :Syllables) # 0.997
@show JudiLing.eval_SC(Shat_val, S_val, french_val,
  :Syllables) # 0.929
```

For production, again we start with creating the predicted form vectors.

```
# we use cholesky function to calculate mapping G from S to C
G_train = JudiLing.make_transform_matrix(S_train, cue_obj_train.C)

# we calculate Chat matrixes by multiplying S and G
Chat_train = S_train * G_train
Chat_val = S_val * G_train
```

For the adjacency matrix, here we use the one with attested connections. This is because for such a big dataset, finding paths with all possible continuations require considerable computing resources.

```
# here we have the adjacency matrix for
A_train = cue_obj_train.A

# we calculate how many timestep we need
max_t = JudiLing.cal_max_timestep(
  french_train, french_val, :Syllables)
```

For `learn_paths`, we set the `threshold` to 0.1 for the training data. For the validation data, we maintain the first `threshold` to 0.1, but turn on the tolerance mode with second `threshold` of -0.1 and to allow maximum 3 weak connections.

```
# learn_path for training data
res_learn_train = JudiLing.learn_paths(
  french_train,
  french_train,
  cue_obj_train.C,
  S_train,
  F_train,
  Chat_train,
  A_train,
```

```julia
  cue_obj_train.i2f,
  cue_obj_train.f2i,
  max_t=max_t,
  max_can=10,
  grams=3,
  threshold=0.1,
  tokenized=true,
  sep_token="-",
  keep_sep=true,
  target_col=:Syllables,
  verbose=true)

# learn_path for validation data
res_learn_val = JudiLing.learn_paths(
  french_train,
  french_val,
  cue_obj_train.C,
  S_val,
  F_train,
  Chat_val,
  A_train,
  cue_obj_train.i2f,
  cue_obj_train.f2i,
  max_t=max_t,
  max_can=10,
  grams=2,
  threshold=0.1,
  is_tolerant=true,
  tolerance=-0.1,
  max_tolerance=3,
  tokenized=true,
  sep_token="-",
  keep_sep=true,
  target_col=:Syllables,
  verbose=true)

# build_path for training data
res_build_train = JudiLing.build_paths(
  french_train,
  cue_obj_train.C,
  S_train,
  F_train,
  Chat_train,
  A_train,
  cue_obj_train.i2f,
  cue_obj_train.gold_ind,
  max_t=max_t,
  n_neighbors=3,
  verbose=true
  )

# build_path for validation data
res_build_val = JudiLing.build_paths(
  french_val,
  cue_obj_train.C,
  S_val,
  F_train,
  Chat_val,
  A_train,
  cue_obj_train.i2f,
  cue_obj_train.gold_ind,
  max_t=max_t,
  n_neighbors=20,
  verbose=true
  )

# learn_paths for training data
@show acc_learn_train = JudiLing.eval_acc(
  res_learn_train,
  cue_obj_train.gold_ind,
  verbose=false
) # 0.982

# learn_paths for validation data
@show acc_learn_val = JudiLing.eval_acc(
  res_learn_val,
  cue_obj_val.gold_ind,
  verbose=false
) # 0.669

# build_paths for training data
@show acc_build_train = JudiLing.eval_acc(
  res_build_train,
  cue_obj_train.gold_ind,
  verbose=false
) # 0.998

# buil_paths for validation data
@show acc_build_val = JudiLing.eval_acc(
  res_build_val,
  cue_obj_val.gold_ind,
  verbose=false
```

Table 17: Accuracies of different categories in validation data performed by `learn_paths`.

		Total	**Correct**	**Accuracy**
randomly split	with unseen cues	108	74	0.685
	without unseen cues	892	595	0.667
	total	1000	669	0.669
carefully split	total	1000	670	0.670

```
) # 0.082
```

We only evaluate the model with 1000 randomly selected data because conducting cross-validation with reasonable accuracy is still time-consuming. The production accuracies of both algorithms for the training data are high, but as expected for this complex dataset, `build_paths` is useless. By contrast, the overall accuracy of `learn_paths` is at 66.9%. If we further divide the validation data into word forms with and without unseen cues, as presented in the upper part of Table 17, we can see that the accuracies are roughly the same. This suggests that with proper model settings, the path-finding algorithms are not really disturbed by unseen cues. This is further confirmed by a further test, where we carefully selected the validation data so that it does not contain any novel cues. Nevertheless, we obtained a similary accuracy, 67% (lower part of Table 17).

Finally we output the results by saving it as CSV files.

```
JudiLing.write2csv(
  res_learn_train,
  french_train,
  cue_obj_train,
  cue_obj_train,
  "french_learn_res_train.csv",
  grams=2,
  tokenized=false,
  sep_token=nothing,
  start_end_token="#",
  output_sep_token="",
  path_sep_token=":",
  target_col=:Syllables,
  root_dir=@__DIR__,
  output_dir="out"
  )

JudiLing.write2csv(
  res_learn_val,
  french_val,
  cue_obj_train,
  cue_obj_val,
  "french_learn_res_val.csv",
  grams=2,
  tokenized=false,
  sep_token=nothing,
  start_end_token="#",
  output_sep_token="",
  path_sep_token=":",
  target_col=:Syllables,
  root_dir=@__DIR__,
  output_dir="out"
  )

JudiLing.write2csv(
  res_build_train,
  french_train,
  cue_obj_train,
  cue_obj_train,
  "french_build_res_train.csv",
  grams=3,
  tokenized=true,
  sep_token="-",
  start_end_token="#",
  output_sep_token="",
  path_sep_token=":",
  target_col=:Syllables,
  root_dir=@__DIR__,
  output_dir="out"
  )

JudiLing.write2csv(
  res_build_val,
  french_val,
  cue_obj_train,
  cue_obj_val,
  "french_build_res_val.csv",
  grams=3,
  tokenized=true,
  sep_token="-",
  start_end_token="#",
  output_sep_token="",
  path_sep_token=":",
  target_col=:Syllables,
  root_dir=@__DIR__,
  output_dir="out"
  )
```

6 Conclusion

LDL is a simple two-layer bidirectional neural network model. It designed to model both comprehension and production. Given the cue matrix $\boldsymbol{C}$, it transforms $\boldsymbol{C}$ into $\hat{\boldsymbol{S}}$ by multiplying with transformation weights $\boldsymbol{F}$, while given the semantic matrix $\boldsymbol{S}$, it produces the $\hat{\boldsymbol{C}}$ matrix by multiplying $\boldsymbol{S}$ with $\boldsymbol{G}$. The second step, namely the path-finding algorithm, is required on the production side to assemble n-gram cues in the proper order.

Baayen et al. (2018a) first implemented LDL in R (`WpmWithLdl`). The modeling results of LDL thus far demonstrate that this simple network works well for a number of morphologically complex languages. However, the R implementation has its limits in terms of speed and memory usage. It is difficult to model large datasets with `WpmWithLdl`. We tried to model the French dataset with `WpmWithLdl` after 8 hours of training, the program crashed because it ran out of memory. Compared with R, Julia supports Linear Algebra computation natively and it provides fast and reliable computation without having to import external libraries. There are also two new algorithms that `Judiling` makes available. The two path-finding algorithms both reduce computation time and memory required. They also do not suffer from the problem that in large graphs, there are far too many paths, and it is also no longer necessary to implement workarounds for dealing with words with cycles. The two path- finding algorithms provide different functionalities. `Learn_path` provides positional learnablities at each timestep while `build_path` further restricts the number of candidate cues by only accepting cues from k nearest form neighbors.

Along with the substantially reduced computation time, several new features are designed and included in the package to facilitate cross-validation. For example, for novel connections and cues that are not attested in the training data, one can opt to use the full adjacency matrix, or one can turn on the tolerance mode to allow for weaker links during path construction. Finally, the Widrow-Hoff learning rule is implemented. This incremental learning rule gives the model the flexibility to trace the learning process, and the possibility to evaluate huge datasets that cannot be handled well by matrix inversion.

In conclusion, because `Julia` optimizes numeric computation within Linear Algebra and uses memory wisely, `JudiLing` reduces computation time and memory substantially specially for production model. Along with the two new path-finding algorithms and other useful features, JudiLing brings LDL to the next level by making it feasible to model substantially larger datasets within reasonable time.

References

Baayen, R. H., Chuang, Y.-Y., and Blevins, J. P. (2018a). Inflectional morphology with linear mappings. *The Mental Lexicon*, 13(2):230–268.

Baayen, R. H., Chuang, Y.-Y., and Heitmeier, M. (2018b). *WpmWithLdl: Im-*

plementation of Word and Paradigm Morphology with Linear Discriminative Learning. R package version 1.0.

Baayen, R. H., Chuang, Y.-Y., Shafaei-Bajestan, E., and Blevins, J. P. (2019). The discriminative lexicon: A unified computational model for the lexicon and lexical processing in comprehension and production grounded not in (de) composition but in linear discriminative learning. *Complexity*, 2019.

Baayen, R. H., Milin, P., urević, D. F., Hendrix, P., and Marelli, M. (2011). An amorphous model for morphological processing in visual comprehension based on naive discriminative learning. *Psychological Review*, 118(3):438–482.

Baayen, R. H., Shaoul, C., Willits, J., and Ramscar, M. (2016). Comprehension without segmentation: A proof of concept with naive discriminative learning. *Language, Cognition, and Neuroscience*, 31(1):106–128.

Baayen, R. H. and Smolka, E. (2020). Modeling morphological priming in german with naive discriminative learning. *Frontiers in Communication*, 5.

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98.

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2016). Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*.

Chuang, Y.-Y., Bell, M. J., Banke, I., and Baayen, R. H. (2020a). Bilingual and multilingual mental lexicon: A modeling study with linear discriminative learning. *Language Learning*, pages 1–52.

Chuang, Y.-Y., Lõo, K., Blevins, J. P., and Baayen, R. H. (2020b). *Estonian Case Inflection Made Simple: A Case Study in Word and Paradigm Morphology with Linear Discriminative Learning*, page 119–141. Cambridge University Press.

Chuang, Y.-Y., Vollmer, M.-L., Shafaei-Bajestan, E., Gahl, S., Hendrix, P., and Baayen, R. H. (2020c). The processing of nonword form and meaning in production and comprehension: A computational modeling approach using linear discriminative learning. *Behavior Research Methods*, pages 1–32.

Csardi, G., Nepusz, T., et al. (2006). The igraph software package for complex network research. *InterJournal, complex systems*, 1695(5):1–9.

Heitmeier, M. and Baayen, R. H. (2020). Simulating phonological and semantic impairment of english tense inflection with linear discriminative learning.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26:3111–3119.

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Plag, I. (2018). *Word-formation in English*. Cambridge University Press.

R Core Team (2020). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Venables, W. N. and Ripley, B. D. (2002). *Modern Applied Statistics with S*. Springer, New York, fourth edition. ISBN 0-387-95457-0.