

BACHELOR THESIS
ARTIFICIAL INTELLIGENCE

Radboud University



**Orthographic Sorting for Second
Language Acquisition Using
MindSort**

Author:
Aline Boels
s1002984

First supervisor:
dr. F.T.M. Leoné
Donders Institute
f.leone@donders.ru.nl

Second supervisor:
prof. dr. A.F.J. Dijkstra
Donders Institute
t.dijkstra@donders.ru.nl



August 24, 2022

Abstract

For foreign vocabulary acquisition, similar words are sometimes found to be confusing, even for more advanced learning. MindSort, a brain-inspired application for learning foreign vocabulary, focuses on this challenge of similarity. Hereto, MindSort maps more similar words closer together on the screen. To investigate the best way to map the words on the screen, this study investigates the question *How do participants judge orthographic similarity between unknown words in a 2D spatial sorting task?* To this end, an experiment was conducted where 20 participants were asked to spatially sort 15 Swahili words on a screen for two vocabulary lists, based on how (dis)similar they judged words to be. The resulting data from this experiment consisted of lists with word pair distances, which were compared to five different orthographic models of word pair distances: Levenshtein distance, Damerau-Levenshtein distance, normalized Levenshtein distance, weighted open bigram, and extended spatial coding. To get the word pair distance per model, the following four steps were taken: (1) implement the algorithm, (2) calculate the distances between words in a distance matrix, (3) map the distances to a 2D map using Multidimensional Scaling (MDS), and (4) calculate the Euclidean distances between word pairs. We calculated the correlation between the experimental data of the word pair distances in the sorting of the participants and the data from the models using Spearman's rank correlation coefficient. Comparing these correlation for the different models using Friedman's test and Wilcoxon signed rank test showed that the weighted open bigram model scored significantly better than Damerau-Levenshtein and normalized Levenshtein. Contributing to further developments in the field of psycholinguistics we notice that participants mentioned sound as most important word feature for sorting besides the first letter of the word. Because of the limited scope of this study, we suggest to do further research to see whether our results also hold with more data. Especially the selection of the words for the word list needs further attention for which our research provides several starting points.

Contents

1	Introduction	3
1.1	Vocabulary learning	4
1.2	Measuring orthographic similarity	6
1.3	Overview of the present article	7
2	Models of orthographic similarity	9
2.1	Levenshtein	9
2.1.1	Damerau-Levenshtein	10
2.1.2	Normalized Levenshtein	10
2.2	Weighted Open Bigrams	11
2.3	Extended Spatial Coding	11
3	Method	13
3.1	Participants	13
3.2	Materials	13
3.2.1	Word lists	13
3.2.2	Used software and hardware	14
3.2.3	Questionnaire	15
3.3	Experimental procedure	16
3.4	Data analysis	17
3.4.1	Processing the experimental data	17
3.4.2	Processing the models	18
3.4.3	Spearman's rank correlation coefficient	19
3.4.4	Statistical analysis	20
4	Results	21
4.1	Statistical analysis	23
4.2	Qualitative analysis	24
5	Discussion	27
5.1	Limitations	28
5.2	Contributions	29
5.3	Future research	30
5.4	Conclusions	31

References	32
A Explanation of the experiment	35
B Experimental outcomes	38
C Code	43

Chapter 1

Introduction

In the Netherlands, every child in primary and secondary school learns at least two foreign languages (Rijksoverheid, n.d.), which can be challenging. Vocabulary learning, including learning specific words, is a crucial part of the learning process (Celce-Murcia & McIntosh, 1991), but especially similar words are sometimes found to be confusing, even for more advanced learners (Laufer, 1988; Llach, 2015). For example, the two words bear and beer look quite similar but have a slightly different spelling and a completely different meaning. Similarity of such words is called orthographic similarity. An open question is, what is an effective and natural method to learn such orthographically similar words. MindSort is an application that is used for researching such questions. It is a brain-inspired application that focuses on this challenge of word similarity, by mapping similar words closer together on the screen and less similar words further apart (Leoné, in prep.; *MindSort*, 2022). As explained in more detail in Section 1.1, contrasting such similar words can have a positive effect on learning.

To map foreign vocabulary based on word similarity, it is necessary to have a method for measuring similarity between words. Measuring word similarity of a word pair can be based on similarity in semantics (meaning), phonology (sound) or orthography (spelling). Based on studies of human similarity judgement, constraints for orthographic similarity have been identified (Forster & Davis, 1984; Hannagan, Dupoux, & Christophe, 2011; Peressotti & Grainger, 1999; Fischer-Baum, 2017; Davis, 2010a), which can be used to test how well models of human orthographic similarity judgement perform (Dijkstra, 2017; Davis, 2010a). An example of a constraint is “transposition neighbours are perceived more similar than substitution neighbours”: *slat-salt* > *stop-shop* (Perea & Lupker, 2003; Davis, 2010a; Hannagan et al., 2011). However, comparing the performance of those models based on constraints does not necessarily mean that the best performing model is actually most similar to how humans judge word similarity. Therefore, this study compares human behaviour in a foreign vocabulary similarity

judgement task to that of the different models. This is done in a one-shot measurement, which differs from other studies that measured the results in several rounds (Ansteeg, Leoné, & Dijkstra, 2022). A one-shot measurement means that participants sort all words at once, instead in several rounds. Sorting all words at once is closer to the procedure used in MindSort. Finding the model which is most similar to human judgement can improve the performance of MindSort—the application that is used for the experiment—as this might improve learning performance, see also Section 1.1. The results of our study can also help improve computational models and can provide insights into language acquisition relevant for the field of psycholinguistics in general.

Before we further introduce the experiment, the next sections explain the brain-inspired assumptions on vocabulary learning that MindSort is based on, as well as why this experiment focuses on orthographic similarity and how to measure orthographic similarity.

1.1 Vocabulary learning

The goal of MindSort is to learn foreign vocabulary, for which three brain-inspired assumptions are used in the application: (1) vocabulary is stored in spatial representations in the brain (Darling, Havelka, Allen, Bunyan, & Flornes, 2020), (2) vocabulary is stored in the mental lexicon based on semantics, phonology and orthography, which dates back to findings of Baron (1977), and (3) orthographic similarity can benefit learning (Baxter et al., 2021, 2022). All of these are discussed in the following subsections.

Spatial representations

Information is thought to be stored in spatial representations in the brain (Bellmund, Gärdenfors, Moser, & Doeller, 2018; Bottini & Doeller, 2020), which is why MindSort presents the vocabulary cards visuospatially. This feature is also inspired by findings that visuospatial information in the form of unfamiliar displays can have a positive influence on learning performance (Darling et al., 2020). The visuospatial representations used in MindSort might also improve learning performance, especially when these representations are similar to the spatial representations of the participants.

Triangle framework

Vocabulary is thought to be stored in the mental lexicon based on semantics (meaning), phonology (sound) and orthography (spelling), which are three broad domains of knowledge for lexical processing which interact, see Figure 1.1 (Baron, 1977; Coltheart, 2005). For the lexical quality of words, grammar and constituent binding are also important (C. A. Perfetti & Hart, 2002;

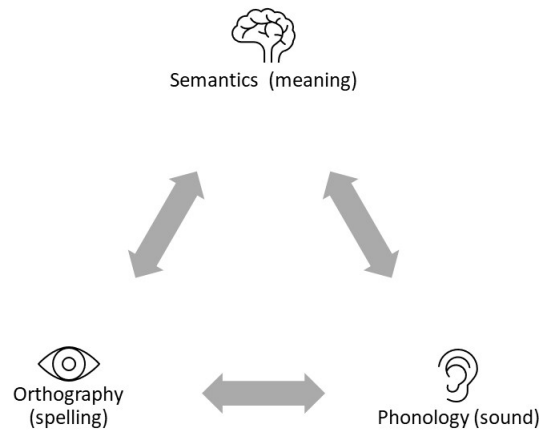


Figure 1.1: Triangle framework of lexical processing.

C. Perfetti, 2007). Constituent binding is the binding between semantics, phonology, orthography and grammar. All five together can result in a high lexical quality representation, meaning that the representation of a word in your brain is of high quality; you know exactly what it means (semantics), how it sounds (phonology), how to conjugate the word (grammar) and how to spell the word (orthography). This study only focuses on the orthography of vocabulary, specifically with regards to orthographic similarity.

Orthographic similarity

Orthographic similarity is the similarity between words based on their spelling. In our experiment we use two Swahili word lists. An example from the word lists used in our experiment is *chupa-chura*, which differ by one letter and are thus orthographic similar, especially when compared to *shimo-degaga*, which do not have any letters in common. Orthographic similarity has been thought to be confusing (Laufer, 1988; Llach, 2015). However, recent findings indicate it can also facilitate learning when contrasting is used (Baxter et al., 2021, 2022). With contrasting, similar words are learned together, to sharpen the representations of those words. This effect was found to be present more for orthography than it was for semantics in a multiple-choice learning task.

1.2 Measuring orthographic similarity

To study orthographic similarity, we first need to know how to measure orthographic similarity and dissimilarity between words. That is why this section explains how human judgement of orthographic similarity has been modelled in constraints, after which we look into models to measure orthographic similarity.

Constraints based on human judgement

Several authors have identified constraints for orthographic similarity based on studies using the masked priming paradigm (Forster & Davis, 1984; Hannagan et al., 2011; Peressotti & Grainger, 1999; Fischer-Baum, 2017; Dijkstra, 2017; Davis, 2010a). These constraints are rules that indicate which word is perceived to be more equal than the other. Constraints tell us more about how people judge orthographic similarity for word pairs, which can be used as inspiration for models on how to measure orthographic similarity.

Hannagan et al. (2011) combined the studies into four categories of constraints: stability, edge effect, transposition effect and relative priming effect. The stability category for example tells us that a word is always most similar to itself, meaning that deletions, additions and substitutions always result in a lower similarity value. Dijkstra (2017) added constraints from Peressotti and Grainger (1999) and Fischer-Baum (2017) to those categories, resulting in a total of 25 constraints that depict how people judge orthographic similarity of word pairs. These constraints can be used to compare the performance of models that measure orthographic similarity.

Models

Models of orthographic similarity are often based on human orthographic similarity judgement. The most well-known model for orthographic similarity is the Levenshtein distance, also sometimes referred to as the edit distance. This model calculates the orthographic distance between two words based on the amount of additions, deletions and substitutions of letters to go from the prime to the target word. Examples of other models are holographic slot coding (Hannagan et al., 2011), weighted open bigrams (Whitney, 2008; Whitney, Bertrand, & Grainger, 2012), spatial coding (Davis, 2010b) and extended spatial coding (Fischer-Baum, 2017; Dijkstra, 2017). Weighted open bigrams and extended spatial coding are used in this experiment and discussed in more detail in Chapter 2.

Comparing human judgement to models

We want to compare the orthographic similarity as judged by people to several orthographic models that measure orthographic similarity. Dijkstra

(2017) has compared several models that measure orthographic similarity on 25 constraints, where weighted open bigrams (Whitney, 2008; Whitney et al., 2012) and extended spatial coding (Davis, 2010b; Fischer-Baum, 2017; Dijkstra, 2017) turned out to be the best options.

When models are compared to constraints, the distances between word pairs are compared. However, in a spatial sorting task, this is more challenging. That is why a recent study by Ansteeg et al. (2022) used inverse Multidimensional Scaling (inverse MDS) in a spatial sorting task (Kriegeskorte & Mur, 2012). Multidimensional scaling is a technique which can map a distance matrix with similarity distances between word pairs to a 2D map (Borg & Groenen, 2005). Inverse MDS does the opposite, from a 2D map, it can transfer the data to a distance matrix with similarity distances between word pairs. Inverse MDS uses an adaptive design where participants in multiple rounds judge the (dis)similarity of words. The words chosen for each round consists of word pairs for which the algorithm is not confident about the similarity value, because there is not enough evidence yet. This iterative approach ensures that in the end, the algorithm is able to make a distance matrix with similarity values for all word pairs.

1.3 Overview of the present article

In this study, we want to investigate which models best explain human similarity judgement in a visuospatial sorting task, to see which model can be best used in MindSort. Dijkstra (2017) has found that based on 25 similarity constraints, weighted open bigrams and extended spatial coding score best. However, those constraints might not reflect the human behaviour. Ansteeg et al. (2022) did compare models to human behaviour using inverse MDS. A downside of using inverse MDS, is that it you have to measure it in several rounds, instead of in one shot. In the present study we use one shot as we want to provide new insights into human similarity judgement by answering the question *How do participants judge orthographic similarity between unknown words in a 2D spatial sorting task?*

To answer this question, we compared the human similarity judgement with the similarity judgement of models that measure orthographic distance. We did this by conducting an experiment where participants spatially sort foreign vocabulary cards, which was then compared to the spatial sorting (mapping) of five different models. We also asked participants what their sorting strategies were to gain insights into the reasoning behind their similarity judgements, which can inspire future research.

The structure of the rest of the article is as follows. First, in the next chapter, background knowledge is provided about the models we use in our study. Next, those models are compared to experimental outcomes, for which the procedure and analysis are further explained in Chapter 3. The

results of the experiment are presented in Chapter 4, after which we end with our discussion and conclusion in Chapter 5.

Chapter 2

Models of orthographic similarity

In this chapter, some theoretical background is given about the models that were used in this study: Levenshtein distance, Damerau-Levenshtein distance, normalized Levenshtein distance, weighted open bigram, and extended spatial coding.

2.1 Levenshtein

The Levenshtein distance, often also referred to as the edit distance, is the orthographic distance between two words based on three basic operations on the letters: deletion, insertion and substitution (Levenshtein et al., 1966). The formula for the Levenshtein distance is given by equation (2.1) (Wikipedia, n.d.), where *tail* is a string without its first character.

$$lev(a, b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ lev(tail(a), tail(b)) & \text{if } a[0] = b[0] \\ 1 + \min \begin{cases} lev(tail(a), b) \\ lev(a, tail(b)) \\ lev(tail(a), tail(b)) \end{cases} & \text{otherwise,} \end{cases} \quad (2.1)$$

Wagner and Fischer (1974) made an algorithm to iteratively make a matrix of all edits needed to go from the prime to the target, where the resulting number in the right bottom row is the minimal edit distance, see Figure 2.1 for an example for the words 'kitten' and 'sitting'. In this example, the yellow 1 in the row after the letters 's' stands for substituting the letter 's'

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

Figure 2.1: Example of Levenshtein distance using the Wagner-Fischer algorithm. The resulting minimal edit distance of *sitting-kitten* is 3.

by 'k', while the '2' thereafter stands for two actions: substituting the letter 's' by 'k' and then adding the 'i'.

This algorithm was not compared by Dijkstra (2017), but is included in this study because it is widely used in linguistic research and it is a relatively easy algorithm, whereas some of the other orthographic similarity metrics are more sophisticated. Since the Levenshtein distance is widely used, there are also variants of it which we have included in this study.

2.1.1 Damerau-Levenshtein

The Damerau-Levenshtein distance is in the basis the same as the Levenshtein distance, but it has an additional operation: transposition (Damerau, 1964). With transposition, two letters can be swapped, for example **slat-salt** are transposition neighbours and **stop-spot** are nonadjacent transposition neighbours (Davis, 2010a). This variant of Levenshtein is relevant because transposition neighbours have been found to be perceived more similar than substitution neighbours, where one letter is replaced by another (Grainger, Whitney, et al., 2004; Hannagan et al., 2011; Davis, 2010a).

2.1.2 Normalized Levenshtein

For the normalized Levenshtein distance, the outcomes of the Levenshtein distance are normalized. The normalized Levenshtein distance takes the length of words into account, whereas Levenshtein does not. The reason why this algorithm is included here, is because big differences in word length can be misleading when using Levenshtein. Assume two words with equal

length of four letters, that differ three letters and compare this with two words with equal length of eight letters that differ three letters, for example, *bake-hide* and *consults-consumed*. The two words with length of eight letters will be perceived more similar than the words with four letters, because those words have five letters in common whilst the first two have only one letter in common. Levenshtein will give the same distance for both examples, whilst normalized Levenshtein will give a bigger distance for the four letter words than for the eight letter words. This is accomplished by dividing through the maximum possible Levenshtein distance: $\max(\text{length}(\text{word1}), \text{length}(\text{word2}))$.

2.2 Weighted Open Bigrams

A bigram is an ordered string consisting of two consecutive letters from a word (Whitney, 2008; Whitney et al., 2012). A word thus consists of several bigrams, of which examples are shown in Figure 2.2. Closed bigrams consists of letters directly next to each other, open bigrams are letter pairs separated from each other and a special case is the edge bigram, the combination of a first or last letter with its edge. Different bigrams have a different weighting, which depends on the separation of the letters and the length of the prime word. In this study, the implementation and weights of Dijkstra (2017) are used, which are based on the open bigram model of Whitney (2008). The weighted open bigrams satisfied 23 out of 25 constraints from Dijkstra (2017), resulting in a total score of 92%. It scored best after extended spatial coding.

Closed bigrams of "Nyasi"					
NY	YA	AS	SI		
Open bigrams of "Nyasi"					
NA	NS	NI	YS	YI	AI
Edge bigrams of "Nyasi"					
#N	I#				

Figure 2.2: Example of bigrams of word "Nyasi". Open bigrams consist of both the closed bigrams, and the new open bigrams underneath.

2.3 Extended Spatial Coding

The extended spatial coding model (Dijkstra, 2017), is based on the spatial coding model (Davis, 2010b). The spatial coding model returns a spatial

code of letter representations, which are based on relative positions of letters. The similarity of a target and prime word is calculated in the first six steps of the spatial coding model: (1) encoding the letter positions in the prime, (2) determining their relative positions compared to the target letters, (3) determining the matches for relative positions, (4) calculating the receiver output, (5) inspecting matching begin- and end-letters, and (6) calculating the similarity value from the receiver output and end letter match. The full model is described by Davis (2010b). A simplified explanation of the model can be found in Dijkstra (2017); Oude Kempers (2020).

Dijkstra (2017) modified the spatial coding model to the extended spatial coding model, inspired by an approach suggested by Fischer-Baum (2017). The extended spatial coding model adds an extra step called dynamic double letter marking, which is explained in Dijkstra (2017). With this extra step, it satisfies all 25 constraints it was tested on by Dijkstra (2017), except for one constraint which had the least evidence.

Chapter 3

Method

3.1 Participants

The experiment was completed by a convenience sample of 20 participants. All participants were native Dutch speakers, students of the Radboud University, aged 19-25 (mean age 21.5), of which 13 male, 6 female and 1 unknown, with all no prior knowledge of Swahili. The participants participated voluntarily and did not receive any form of compensation for this experiment. Informed consent was given by all participants in accordance with the guidelines of the Radboud University Social Sciences Ethics Committee.

3.2 Materials

3.2.1 Word lists

For this experiment, two word lists were compiled in Swahili consisting of 15 words each, taken from a pre-selected set used for ongoing research (Goertz, Leoné, De Groot, & Bekkering, in prep.). For these two lists, several factors were considered: word length, frequency en concreteness of Dutch words, similarity between Dutch and Swahili words and the Levenshtein similarity mapping. The lists can be found in Table 3.1.

Swahili was chosen as the language to learn in this study for two reasons. First, like Dutch, it uses the Latin alphabet, which ensures that participants can understand and recognize the letters. Secondly, it is a Bantu language and not a Germanic like Dutch, German or English, nor a Romance language like French, which are all languages being taught in primary and secondary school in the Netherlands (Rijksoverheid, n.d.). Using Swahili reduces the possibility of prior (semantic) knowledge influencing this study (Baxter et al., 2022).

We used two word list for several reasons. First, spatially sorting 15 words on a screen is easier than sorting 30 words. With 30 words, words

could end up close to each other due to space limitations instead of similarity. Secondly, using two word lists generates more data compared to one. Thirdly, using two lists makes it possible to examine the possible influence of the word lists on the perceived versus the calculated similarity. In addition, both lists were also used in a parallel research by another bachelor student on students’ learning (de Groot, 2022). The combination of that and our research can inform future development of MindSort.

Table 3.1: Dictionary

List 1		List 2	
Swahili	Dutch	Swahili	Dutch
bahari	zee	bamba	bord
bahasha	envelop	chanjo	schaar
barua	brief	chikiru	vogel
chupa	fles	dhahabu	goud
chura	kikker	dhoruba	storm
duara	wiel	farasi	paard
degaga	bril	fasihi	boek
kanisa	kerk	funguo	sleutel
kupanda	fabriek	kamba	touw
lango	deur	kinywa	mond
mchanga	zand	mchuma	geweer
mizizi	wortels	miguu	voeten
mkono	hand	mlima	berg
nyasi	gras	nyoka	slang
shimo	gat	ramani	kaart

3.2.2 Used software and hardware

All participants conducted the experiment on a laptop with a screen size of 14 inch. The experiment was done using MindSort (*MindSort*, 2022; Leoné, in prep.). The built in touchpad of the laptop was used to spatially sort the vocabulary cards with one word per card.

MindSort

MindSort is a brain-inspired language learning application to learn foreign vocabulary using vocabulary cards (*MindSort*, 2022; Leoné, in prep.). During a learning session, the vocabulary cards are already spatially sorted

based on a similarity model like the Levenshtein distance. However, in our experiment we used a functionality in the admin panel to customize the sorting. For each participant, a new list was added in the admin panel, which they could edit and spatially sort themselves. The screen they saw at the start of the experiment is shown in Figure 3.1. MindSort does not have a functionality to randomize the order of the cards, thus every participant saw the vocabulary cards in the same order. The vocabulary cards could be arranged by drag-and-drop operations with a touch pad. Participants could finalize the arrangement by clicking on the "save" button (not shown in the figure).

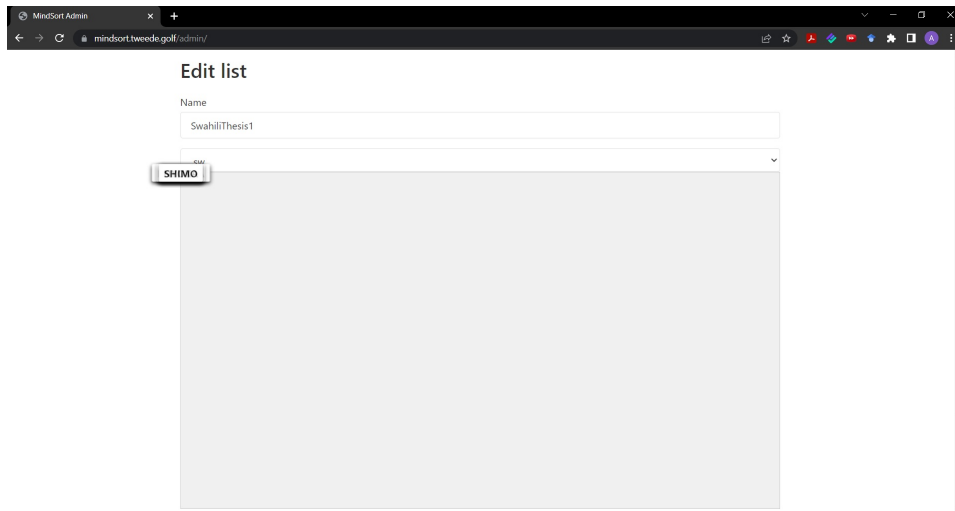


Figure 3.1: The screen participants saw at the start of their experiment, with in the left top corner the vocabulary cards stacked in a pile. When they scrolled down, they could see a green button to save the sorting.

3.2.3 Questionnaire

For this experiment, a questionnaire was made in Qualtrics. The questionnaire was in Dutch and consisted of the consent letter, questions about the participants their age, gender and knowledge of Swahili, explanation about the experiment and open ended questions about the participants' strategy. The explanation of the experiment that participants saw in the questionnaire can be found in appendix A.

For the reported strategy, the researcher asked the participants open ended questions. The questions were in Dutch, but would be translated to:

- How would you describe your strategy?
- Was your strategy the same for both lists?

- To which word features did you pay attention to?
- To which relative distances did you pay the most attention? Mostly to the relative distances between words that are close to each other, those that are far from each other, those between clusters or some combination of these?

These questions help to not only give insights on which model resembles the participants' sorting best, but to also know why that might be the case.

3.3 Experimental procedure

Participants were recruited at Radboud University, with the requirement that they had to be native Dutch, aged 18-28. The experiment was conducted on the laptop of the researcher with the researcher sitting next to the participants. The procedure is visualised in Figure 3.2.

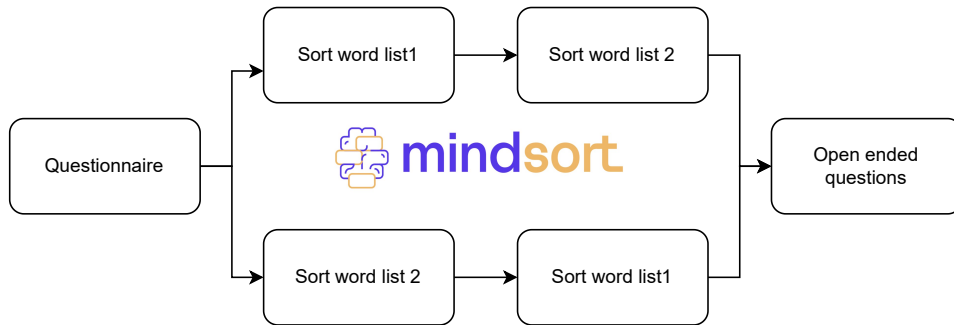


Figure 3.2: Experimental procedure. Participants first got questions in a questionnaire, then randomly got assigned either word list 1 or word list 2 first, after which they had to answer open ended questions asked by the researcher.

All participants started with the questionnaire, where they had to sign the letter of consent, answer the general questions and got the explanation about the experiment. Then, the questionnaire randomly assigned either one of the lists first to the participant, after which they also had to sort the other list. The researcher redirected the participants to MindSort and also gave a verbal explanation of the experiment. For both word lists, all words were stacked on a pile in the left top corner of the screen at the start of the experiment, as shown in 3.1. The participants could take as much time as they wanted for the sorting process. An example of how word list 1 looked for one of the participants is shown in 3.3. When participants were done sorting both lists, the researcher asked the open ended questions from Section 3.2.3, after which the questionnaire was closed and saved.

To make sure participants focused only on the orthography (spelling) and not also on the semantics (meaning), participants only saw the Swahili words during the experiment and not the Dutch translation.

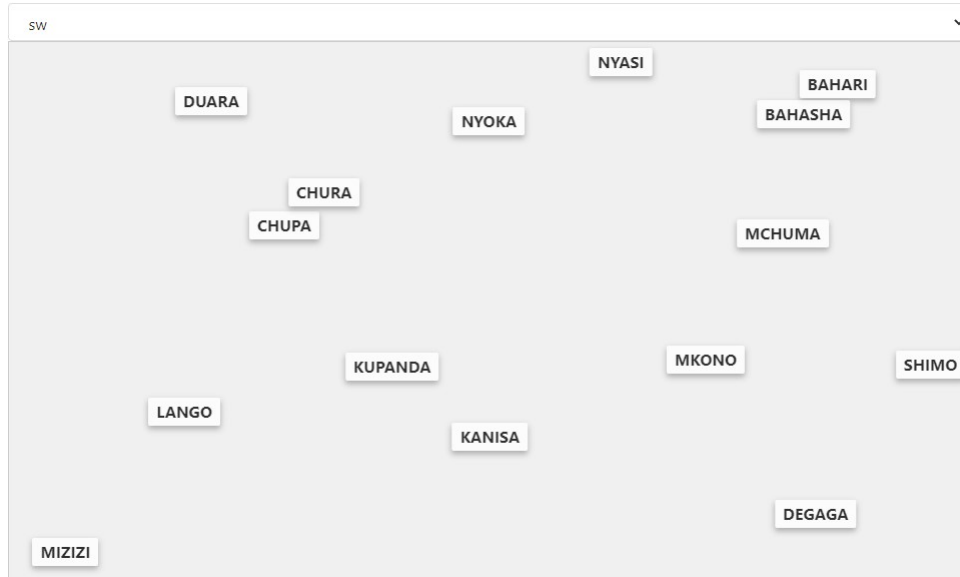


Figure 3.3: Example of how the screen in MindSort looked after a participant sorted word list 1.

3.4 Data analysis

In the data analysis, the experimental outcomes were compared with the outcomes of the five models described in Chapter 2. This process is visualized in Figure 3.4 and is further explained in this section.

3.4.1 Processing the experimental data

Each participant sorted both word lists in MindSort, which resulted in two lists per participant with x- and y-coordinates for each word. The x- and y-coordinates were exported from MindSort. These coordinates were used to calculate the Euclidean distances between word pairs, so that for each word list we had 20 lists with word pair distances (one list for each participant that finished the experiment). These lists could then, after processing the models, be used to compare the participants' outcomes with those of the models.

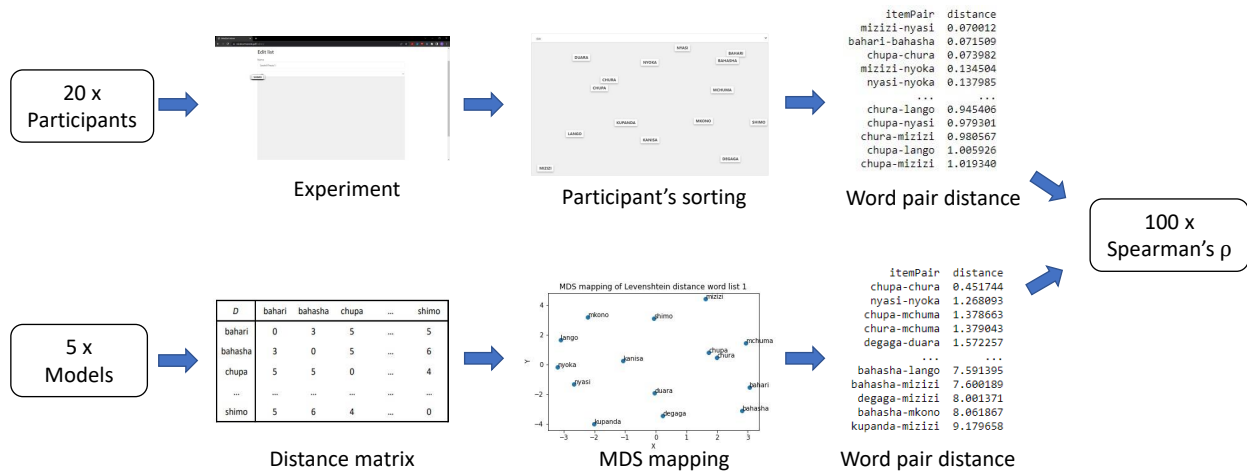


Figure 3.4: Visualisation of the data analysis which was done once for every word list. The distances between word pairs for the experimental outcomes and for the five models were compared using Spearman’s rank correlation coefficient ρ . This resulted in $20 \times 5 = 100$ ρ values per word list, and $2 \times 100 = 200$ ρ values in total.

3.4.2 Processing the models

To be able to compare the experimental data with the five models, we need to have the same data format for the models as we have for the experimental data, namely lists of word pair distances. To this end, four steps were implemented in Python: (1) implement the model, (2) calculate the distances between words in a distance matrix, (3) map the distances to a 2D map using Multidimensional Scaling (MDS), and (4) calculate the Euclidean distances between word pairs.

Implementation of the models

Five models were used: Levenshtein distance, Damerau-Levenshtein distance, normalized Levenshtein distance, weighted open bigrams and extended spatial coding. The code of the implementation of all models can be found in appendix C.

The Levenshtein distance was implemented using the Wagner-Fischer model with the weights for addition, deletion and substitution all set to 1. For the normalized Levenshtein distance, the results of the Levenshtein model were used and divided by the maximum possible Levenshtein distance between each word pair, to account for the word length. The Damerau-Levenshtein distances was implemented using the code of Jensen (2013).

Weighted open bigrams and extended spatial coding were implemented in python using the code of Dijkstra (2017). Both those models however calculated a similarity value instead of a distance, with one being similar and zero being dissimilar. Because of that, we calculated the distance using $1 - \textit{similarity}$.

Distance matrices

For all models, the similarity distances between word pairs were calculated and put in a distance matrix. For both weighted open bigrams and extended spatial coding, this resulted in an asymmetric distance matrix because the distance from prime to target gave a different value than the distance from target to prime. This difference was however very small, with no differences bigger than two decimals behind the comma, which is why it was decided to take the average of the two.

The results of the different models are in different ranges. For example, the Levenshtein distance outputs a similarity value between zero and $\max(\textit{length}(\textit{word1}), \textit{length}(\textit{word2}))$, but the normalized Levenshtein distance outputs a similarity value between one and zero.

MDS mapping

These distances matrices were mapped to a 2D map using Multidimensional Scaling (MDS) (Borg & Groenen, 2005). It was decided to first map the distances to a 2D map using MDS instead of using the similarity distances calculated in the step before, because that resembles the experimental setting of the participants more.

The different ranges in the results of the models in Section 3.4.2 also influence the mappings returned by MDS, since this means that the ranges on the x- and y-axes differ per model.

Word pair distances

For both the MDS mapping of the models and the experimental data, the euclidean distances for each word pair were calculated using Pythagorean theorem implemented in Python. This resulted in a list of word pairs with their distances, which was then sorted on similarity, with the smallest euclidean distance first. Here again, the ranges of the results differed between the models. This however is not a problem, because the lists of word pair distances are compared using Spearman's rank correlation coefficient.

3.4.3 Spearman's rank correlation coefficient

To see which models resembled the experimental data most, Spearman's rank correlation coefficient was used to compare each model with each par-

ticipant, resulting in a correlation coefficient (ρ) for each participant per model (Kokoska & Zwillinger, 2000). We use Spearman to compare the results of each participant to the results of each model. This is done by ranking all word pairs based on their similarity distance and then comparing the ranking of the participant to that of a model, for which Spearman returns a rank correlation coefficient ρ . Per word list, 20 participants were compared with five algorithms, which resulted in $20 \times 5 = 100$ ρ values per word list. In total, for both word lists, we had $2 \times 100 = 200$ ρ values.

Spearman’s rank correlation was used for several reasons. First, a Gaussian distribution cannot be assumed for this data, which means that a non-parametric rank correlation method is good practice (Dancey & Reidy, 2007). Next to that, with a rank correlation the distances do not need to be normalized, which would have been needed for other metrics, since the results of the models were in different ranges. Lastly, for this experiment we want to know which word pairs are perceived most similar and which most dissimilar and whether that is the same for participants’ sorting compared to the models’ sorting. For that, the distances are a medium to make that visible, but the factual distances do not matter that much. Spearman’s rank correlation coefficient gives a rho (ρ) value, where $\rho = 1$ implies a perfect positive correlation, and $\rho = -1$ implies a perfect negative correlation (Dancey & Reidy, 2007). A value of $\rho < 0.4$ indicates a weak correlation, $0.4 < \rho < 0.7$ indicates a moderate correlation and $0.7 < \rho < 1$ indicates a strong correlation.

3.4.4 Statistical analysis

With Spearman’s ρ , we can see whether there is a positive or negative correlation between a model and the experimental outcome of a participant. For this study however, we would like to know how participants judge orthographic similarity in the experiment, and thus more specifically whether there is one model that resembles the participants’ judgement most. To do this, two statistical tests were performed on the data. First, the Friedman test was used to identify whether there was any model that significantly differed from the other model. This test was used because the data does not have a normal distribution, we want to compare more than two models, and the data is paired, since we compare models which use the same word list and within participants (Wiki statistiek Amsterdam UMC, n.d.). Next, Wilcoxon signed rank test was used to compare all models pairwise, to see which models significantly differed from each other. Wilcoxon is suitable to compare two models instead of two or more, but also for paired data which is not normally distributed (Wiki statistiek Amsterdam UMC, n.d.). To correct for multiple testing, a significance level of $p < 0.005$ was used, using the Bonferroni correction. The tests were not only applied to the two separate word lists, but also to the two word lists concatenated.

Chapter 4

Results

The experimental data of all participants is visualized in appendix B. The comparison of the experimental data and data from the models resulted in 100 Spearman’s rank correlation coefficient values (ρ) per word list, which are visualized in the heatmaps in Figure 4.1.

From visually inspecting the heatmaps, you can see that there are differences between participants and between models, but there is no one model that scores best for all participants. Also notable is that, for example, participant 11 and 12 seem to have a weak to moderate positive correlation for all models of word list 1, but no correlation to a weak negative correlation for all the models of word list 2. In the open-ended questions, participant 11 indeed indicated to have used a different strategy per word list, but participant 12 indicated to have used the same strategy for both, which does not explain the differences in outcome. In addition, what is clearly notable from the heatmaps, is that the scores for Levenshtein and Damerau-Levenshtein are the same for word list 1—as the colors in both columns are the same.

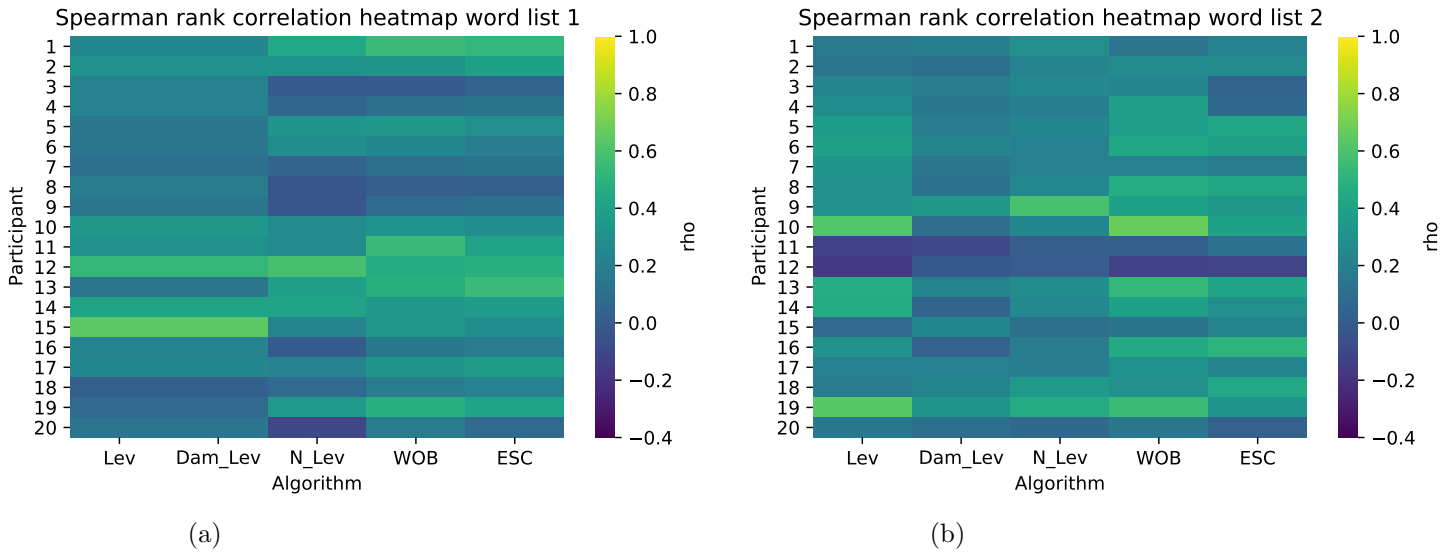


Figure 4.1: Heatmap of Spearman’s rank correlation coefficient. No clear pattern emerges from the heatmap.

Lev = Levenshtein, Dam-Lev = Damerau-Levenshtein, N Lev = Normalized Levenshtein, WOB = Weighted Open Bigrams, ESC = Extended Spatial Coding

The Spearman’s rank correlation coefficients are summarized in the box-plots in Figure 4.2. Some differences can be seen between models, which are tested on significance in the next section.

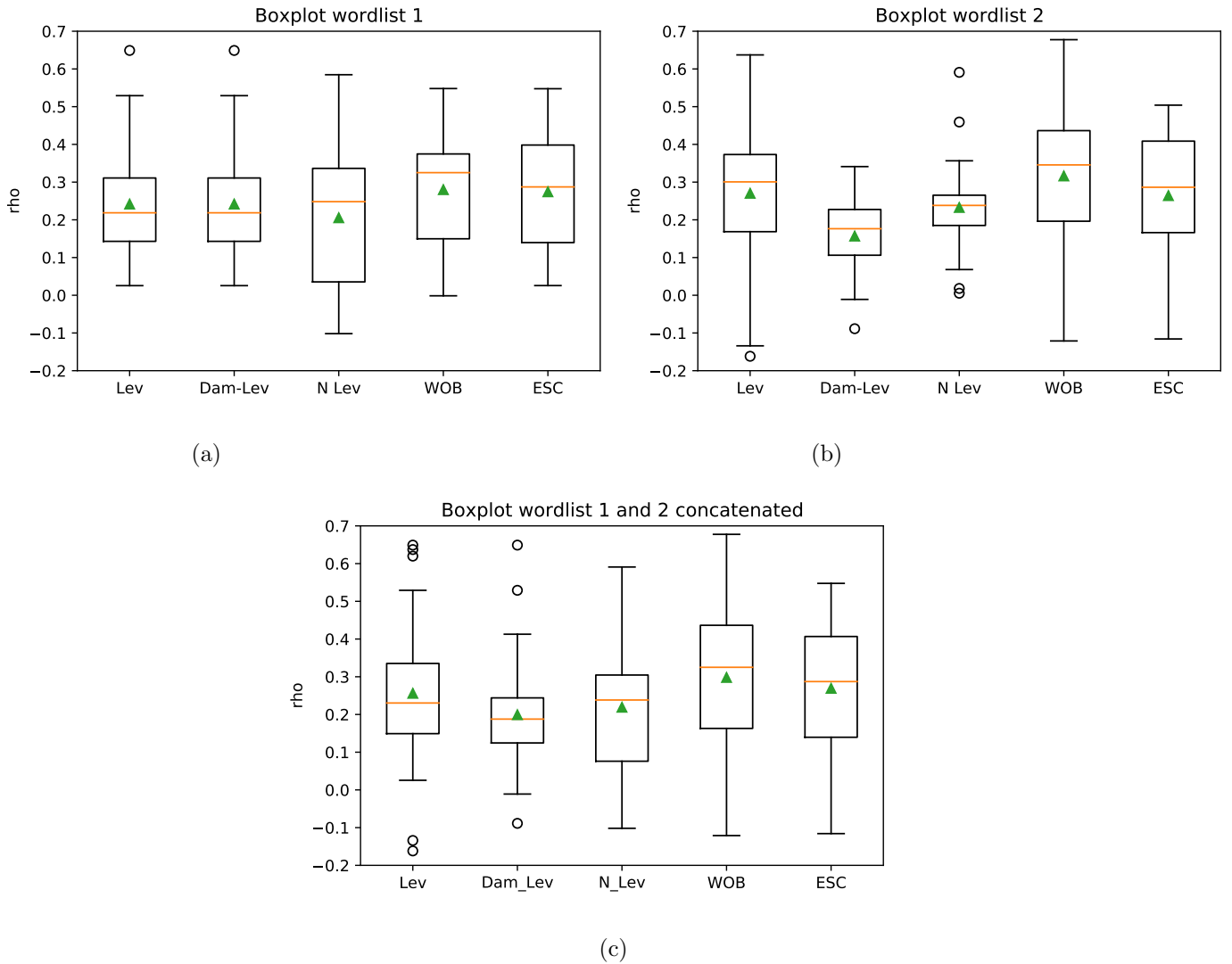


Figure 4.2: Boxplots of the Spearman's rank correlation coefficient of (a) word list 1, $n = 20$, (b) word list 2, $n = 20$, and (c) the data of both word lists concatenated, $n = 40$.

Lev = Levenshtein, Dam-Lev = Damerau-Levenshtein, N Lev = Normalized Levenshtein, WOB = Weighted Open Bigrams, ESC = Extended Spatial Coding

— = median and ▲ = mean

4.1 Statistical analysis

The Friedman test is used to test whether there are significant differences between any of the models. For word list 1, the difference is not significant $X^2(4) = 6.65, p = 0.155$. However, this result might be misleading, because the results of Levenshtein and Damerau-Levenshtein are exactly the same for

word list 1, as this word list does not contain any transposition neighbours. When leaving Damerau-Levenshtein out and performing the Friedman test on only four of the models for word list 1, we find that there is a significance difference $X^2(3) = 8.58, p = 0.035$. The result using all models for word list 2 is $X^2(4) = 16.72, p = 0.002$, the result for the concatenation of both word lists is $X^2(4) = 15.08, p = 0.005$, and both are significant.

Post hoc analysis with Wilcoxon signed-rank tests was conducted with the significance level set at $p < 0.005$, corrected with the Bonferroni correction. All results of the post hoc analysis can be found in Table 4.1, 4.2 and 4.3. Significant results are highlighted. We see that for word list 1, there is a significance difference between weighted open bigrams and normalized Levenshtein, and a significance difference between extended spatial coding and normalized Levenshtein. When comparing those in the boxplot in Figure 4.2c, we see that both weighted open bigrams and extended spatial coding score significantly better than normalized Levenshtein. For word lists 2, when we compare the highlighted results in Table 4.2 with the boxplot in Figure 4.2b, we see that both normalized Levenshtein and weighted open bigrams score significantly better than Damerau-Levenshtein. Lastly, in the overall analyses, we see that weighted open bigrams scores significantly better than Damerau-Levenshtein and normalized Levenshtein.

Table 4.1: Outcomes Wilcoxon statistics word list 1

Word list 1	Levenshtein	Damerau-Levenshtein	Normalized Levenshtein	Weighted Open Bigrams
Damerau-Levenshtein	-			
Normalized Levenshtein	$Z = 84.0,$ $p = 0.4524$	$Z = 84.0,$ $p = 0.4524$		
Weighted Open Bigrams	$Z = 86.0,$ $p = 0.4980$	$Z = 86.0,$ $p = 0.4980$	$Z = 29.0,$ $p = 0.0032$	
Extended Spatial Coding	$Z = 90.0,$ $p = 0.5958$	$Z = 90.0,$ $p = 0.5958$	$Z = 30.0,$ $p = 0.0037$	$Z = 99.0,$ $p = 0.8408$

4.2 Qualitative analysis

For the qualitative analysis, participants had to answer open ended questions asked by the researcher, of which the researcher filled in the answers in the questionnaire. All answers have been categorized by the researcher and are summarized in Figure 4.3.

In Figure 4.3a, you can see that most participants used the same strategy for both word lists, but four indicated to use a different one. The strategies

Table 4.2: Outcomes Wilcoxon statistics word list 2

Word list 2	Levenshtein	Damerau-Levenshtein	Normalized Levenshtein	Weighted Open Bigrams
Damerau Levenshtein	$Z = 40.0,$ $p = 0.0136$			
Normalized Levenshtein	$Z = 78.0,$ $p = 0.3300$	$Z = 26.0,$ $p = 0.0020$		
Weighted Open Bigrams	$Z = 42.0,$ $p = 0.0172$	$Z = 21.0,$ $p = 0.0009$	$Z = 46.0,$ $p = 0.0266$	
Extended Spatial Coding	$Z = 102.0,$ $p = 0.9273$	$Z = 38.0,$ $p = 0.0107$	$Z = 78.0,$ $p = 0.3300$	$Z = 66.0,$ $p = 0.1536$

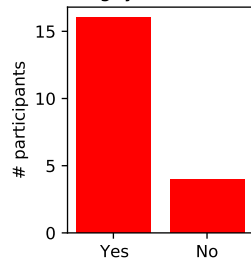
Table 4.3: Outcomes Wilcoxon statistics word lists concatenated

Word lists concatenated	Levenshtein	Damerau-Levenshtein	Normalized Levenshtein	Weighted Open Bigrams
Damerau-Levenshtein	$Z = 40.0,$ $p = 0.0152$			
Normalized Levenshtein	$Z = 315.0,$ $p = 0.2016$	$Z = 324.0,$ $p = 0.2477$		
Weighted Open Bigrams	$Z = 272.0,$ $p = 0.0636$	$Z = 201.0,$ $p = 0.0050$	$Z = 156.0,$ $p = 0.0006$	
Extended Spatial Coding	$Z = 387.0,$ $p = 0.7572$	$Z = 255.0,$ $p = 0.0372$	$Z = 220.0,$ $p = 0.0107$	$Z = 310.0,$ $p = 0.1789$

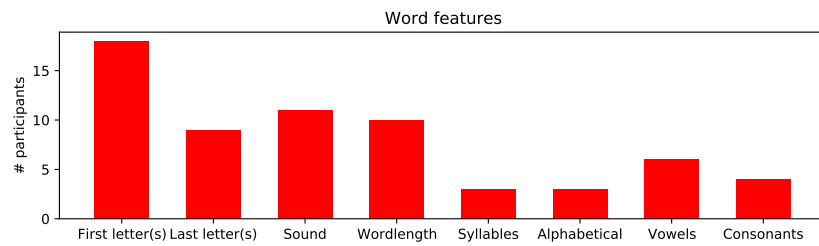
reported were categorized in making clusters (10) and starting with one word, and looking at similarities from there on (7), as shown in Figure 4.3c. The participants in the category other (3) did not report a clear strategy. One participant that worked starting with one word, indicated to work in the shape of a snake, whereas another indicated to work in a X shape.

All participants indicated to pay attention to two to five features, which are summarized in Figure 4.3b. Notable are the number of participants that mention first letter(s) (18), last letter(s) (9), sound (11), and word length (10). The relative distance question showed that participants payed attention to different kind of distances, where some looked more at relative distances for words close to each other and some more for words far apart, see Figure 4.3d.

Same strategy for both word lists

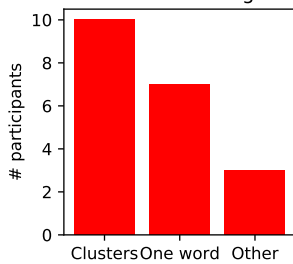


(a)



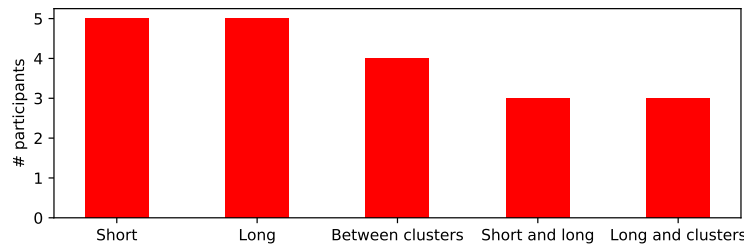
(b)

Described strategies



(c)

Distance focus



(d)

Figure 4.3: Outcomes of qualitative analyses after the participants sorted both word lists. The question depicted in 4.3a was a closed question, all other questions were open questions which were later categorized. (a) $n = 20$, (b) $n = 64$, (c) $n = 20$, and (d) $n = 20$

Chapter 5

Discussion

In this exploratory study, we investigated how participants judge orthographic similarity between unknown words in a 2D spatial sorting task. This is relevant for the brain-inspired application MindSort because earlier work found that contrasting orthographically similar words seems to facilitate learning (Baxter et al., 2021, 2022), but that study did not investigate how participants judge orthographic similarity. Other studies have investigated orthographic similarity judgement by comparing models based on constraints (Dijkstra, 2017) or based on a full measurement with inverse-MDS (Ansteeg et al., 2022). New in our study is that the models are compared based on human behaviour in a one-shot measurement, instead of multiple rounds of sorting one word list. In this study we answer the research question: *How do participants judge orthographic similarity between unknown words in a 2D spatial sorting task?*

Taken the wordlists together, our results show that weighted open bigrams performed significantly better than Damerau-Levenshtein and normalized Levenshtein in the 2D spatial sorting task. This indicates that in this exploratory study, the Weighted Open Bigram model seems to be more similar to human orthographic similarity judgement than Damerau-Levenshtein and normalized Levenshtein. It is not surprising that weighted open bigrams did better, since it satisfied 23 out of 25 constraints in Dijkstra (2017)’s study. Damerau-Levenshtein and normalized Levenshtein were not tested on those constraints, but would satisfy less constraints.

However, what is surprising, is that weighted open bigrams did not perform better than the less advanced original model of Levenshtein. The reason why weighted open bigrams did perform better than Damerau-Levenshtein and normalized Levenshtein, but not better than Levenshtein, might be because the word lists were not balanced enough, which is discussed further in Section 5.1.

After participants completed the experiment, we asked them for their strategies. We were not able to link the qualitative outcomes of these ques-

tions to our quantitative data in a meaningful way due to the limited number of participants and variety in their answers. Some suggestions for how to handle this in future research are given in Section 5.3. We did look at one example where the differences between the two word lists were notable for participant 11 and 12. This example could not be explained by our qualitative data. This might be due to some limitations of our quantitative data, which are discussed in the next section, together with suggestions on how to improve this in future research in Section 5.3.

5.1 Limitations

There are several limitations in this study, of which the first one is the scope. The scope was limited in the number of participants, the number of word lists, words per word lists, the number of words with transposition neighbours, the chosen word lengths and the number of models. Because of the limited study, our findings only give some indications for future research, but no conclusions can be drawn from them. The limitation of having only two word lists is problematic because of the differences you see in results between the two word lists. This can be solved by having bigger word lists with a multi-arrangement task using inverse MDS (Ansteeg et al., 2022; Kriegeskorte & Mur, 2012), which does have the limitation that it does not mimic the future implementation of MindSort, which is a one-shot situation. An option that could solve that is to have more word lists that are used in a one-shot measurement. When making word lists, it is good to take into account how balanced the word lists are. The models used to compare are often based on constraints and if one word list takes more constraints into account than the other, it can have a big influence on the results. In our study, the lists contained almost no transposition neighbours (word list 1 had none, word list 2 had one word), possibly negatively influencing the results for the Damerau-Levenshtein model. Next to that, the words had a length between five and seven letters, which was deliberately chosen because it could influence the parallel study (de Groot, 2022). This however imposes a limitation for the present study, since it takes away the benefit of normalized Levenshtein, which takes the difference in word length into account, as explained in Section 2.1.2.

Secondly, it might be the case that individual differences play a role (Dewaele, 2009; Zafar & Meenakshi, 2012), which is also suggested by some of our qualitative data. In our study we looked at which model scored best taken all participants together, therefore not taking possible relevant individual differences into account.

Thirdly, we only looked at orthographic similarity as that performed better than semantic similarity with contrasting (Baxter et al., 2021, 2022). It could be worthwhile to also consider semantics, and phonology com-

bined with orthography. However, a recent study found that in a multi-arrangement task using inverse MDS, semantic similarity is captured better (Ansteeg et al., 2022) than orthographic similarity. The same study found that there was almost no difference between phonology and orthography. The word features that participants in our study mentioned for sorting seem to support these findings, since 11 out of 20 participants indicated to take the feature "sound" (how they thought a word would sound) into account in the sorting process, as shown in Figure 4.3b.

Lastly, our experimental setup has some limitations. Participants indicated that the shape of the screen influenced their choices in the sorting process, since the rectangle steered them into placing groups of words in the four corners. In addition, all words were stacked in the left top of the screen at the start of the experiment, in the same order for each participant. It would have been better if the words would have been spread out from the start on in a random order, so that the participants can see all words at once and the order cannot influence their choices. However, this random spread out could influence the sorting process as well. Another option would be, to randomize the pile of words. Both options most likely require more participants. Furthermore, a limitation of our experimental setup is the open ended questions after the experiment. The classification of the answers was done by one researcher and was not verified by another researcher. Now that several word features used by participants are known mentioned, in a follow up study a multiple answers choice list including the option 'other' could be compiled. In addition, most participants thought the question about distances to be too vague. A lot of participants also started mentioning word features already when asked to describe their strategy.

5.2 Contributions

Contributing to the broader field of psycholinguistics we first notice that phonology of words is taken into account when sorting words, which is in line with findings from (Ansteeg et al., 2022). Possibly this also influences the learning of words. In addition, participants mentioned several other word features besides sound that they used for sorting words such as the first or last letter and word length. Knowing such features is not only important for the field of psycholinguistics but can also support developers of MindSort and researchers when composing word lists.

Secondly, we notice that edge effects seem important for sorting words as participants mentioned the first letter of a word as the most important word feature for sorting, and the last letter of a word was also mentioned frequently. Models that also take these edge effects into account, are therefore more likely to reflect natural research strategies.

Contributing to further development of MindSort, we notice that the

Levenshtein distance algorithm currently used in MindSort, possibly does not best reflect how people naturally sort words. Other algorithms that take the mentioned word features more into account, seem better candidates, especially weighted open bigrams.

Contributing to the body of literature on orthographic similarity we used a research method in which participants had most likely no semantic knowledge of the words that were sorted, which makes it plausible that participants concentrated more on orthographic similarity in contrast to the study of Ansteeg et al. (2022), where participants did have semantic word knowledge but were asked to ignore it and concentrate on orthographic similarity.

5.3 Future research

Besides the options already mentioned in the limitations section as to overcome these limitations, several options can be explored in future research. We start by focusing on future research directions to understand orthographic similarity judgement better. The first would be to have a broader scope. This can be broader in terms of number of participants, number of word lists, number of words per word lists and which models to compare it with. Our advice would be to start with more word lists, as the present study noticed a difference in outcomes of which models performed best between both word lists and transposition neighbours in words were hardly present.

Secondly, future research could examine participants' strategies in a more systematic way now that our exploratory study revealed some of these. Examining such strategies can provide better insight in the sorting process. Instead of asking participants for their strategy afterwards, think aloud protocols can be considered (Fonteyn, Kuipers, & Grobe, 1993). Another way to gain insight for participants' strategies, is to use eye tracking, which has recently been done to understand participants' strategies for histograms (Boels, Lyford, Bakker, & Drijvers, 2022).

Thirdly, qualitative data can be linked to quantitative data. This can for example be done by using a machine learning algorithm like random forest (Boels et al., 2022), or by clustering using a clustering method for mixed-type data (Foss, Markatou, & Ray, 2019). Linking qualitative data to quantitative data might also help identifying whether individual differences play a role in the strategies, which could be the case, given that individual differences play a role in second language acquisition (Dewaele, 2009; Zafar & Meenakshi, 2012).

Fourth, besides future research on orthographic similarity judgement, it would be good to assess the influence of (orthographic) similarity on learning.

Lastly, our qualitative data can be used to inspire new models of orthographic similarity. One aspect that we think is relevant to be investigated, is that participants mentioned to take phonology into account.

5.4 Conclusions

This exploratory study compared orthographic similarity judgement in a visuospatial sorting task to the orthographic similarity scores of five different models. We found that the weighted open bigrams models scored significantly better than the Damerau-Levenshtein and normalized Levenshtein models. However, our study has several limitations, of which one is the limited scope, thus future research is still needed. For future research, we suggest to have a broader scope, investigate participants' strategies with think aloud protocols or eye tracking and link qualitative data to quantitative data more systematically. Specifically, the influence of phonology of words needs further investigation.

References

- Ansteeg, L., Leoné, F., & Dijkstra, T. (2022). Characterizing the semantic and form-based similarity spaces of the mental lexicon by means of the multi-arrangement method. *Frontiers in Psychology*, 4975.
- Baron, J. (1977). Mechanisms for pronouncing printed words: Use and acquisition. *Basic processes in reading: Perception and comprehension*, 175–216.
- Baxter, P., Bekkering, H., Dijkstra, T., Droop, M., van den Hurk, M., & Leoné, F. (2022). Contrasting orthographically similar words facilitates adult second language vocabulary learning. *Learning and Instruction*, 101582.
- Baxter, P., Droop, M., Van Den Hurk, M., Bekkering, H., Dijkstra, T., & Leoné, F. (2021). Contrasting similar words facilitates second language vocabulary learning in children by sharpening lexical representations. *Frontiers in Psychology*, 12, 2648.
- Bellmund, J. L., Gärdenfors, P., Moser, E. I., & Doeller, C. F. (2018). Navigating cognition: Spatial codes for human thinking. *Science*, 362(6415), eaat6766.
- Boels, L., Lyford, A., Bakker, A., & Drijvers, P. (2022). *Assessing students' learning when interpreting histograms: A gaze-based machine learning analysis*. (Manuscript submitted for publication)
- Borg, I., & Groenen, P. J. (2005). *Modern multidimensional scaling: Theory and applications*. Springer Science & Business Media.
- Bottini, R., & Doeller, C. F. (2020). Knowledge across reference frames: Cognitive maps and image spaces. *Trends in Cognitive Sciences*, 24(8), 606–619.
- Celce-Murcia, M., & McIntosh, L. (1991). Teaching english as a second or foreign language.
- Coltheart, M. (2005). Modeling reading: The dual-route approach. *The science of reading: A handbook*, 6, 23.
- Damerau, F. J. (1964). A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3), 171–176.
- Dancey, C. P., & Reidy, J. (2007). *Statistics without maths for psychology*. Pearson education.

- Darling, S., Havelka, J., Allen, R. J., Bunyan, E., & Flornes, L. (2020). Visuospatial bootstrapping: spatialized displays enhance digit and non-word sequence learning. *Annals of the New York Academy of Sciences*, *1477*(1), 100–112.
- Davis, C. J. (2010a). Orthographic input coding: A review of behavioural evidence and current models. *From Inkmarks to Ideas*, 210–236.
- Davis, C. J. (2010b). The spatial coding model of visual word identification. *Psychological review*, *117*(3), 713.
- de Groot, L. (2022). *The effect of cluster-based learning in a visuospatial similarity-based mapping of foreign vocabulary*. (unpublished)
- Dewaele, J.-M. (2009). Individual differences in second language acquisition. *The new handbook of second language acquisition*, *2*, 623–646.
- Dijkstra, R. (2017). *Modelling orthographic similarity between words*. (unpublished)
- Fischer-Baum, S. (2017). The independence of letter identity and letter doubling in reading. *Psychonomic Bulletin & Review*, *24*(3), 873–878.
- Fonteyn, M. E., Kuipers, B., & Grobe, S. J. (1993). A description of think aloud method and protocol analysis. *Qualitative health research*, *3*(4), 430–441.
- Forster, K. I., & Davis, C. (1984). Repetition priming and frequency attenuation in lexical access. *Journal of experimental psychology: Learning, Memory, and Cognition*, *10*(4), 680.
- Foss, A. H., Markatou, M., & Ray, B. (2019). Distance metrics and clustering methods for mixed-type data. *International Statistical Review*, *87*(1), 80–109.
- Goertz, R., Leoné, F., De Groot, R., & Bekkering, H. (in prep.). *The effect of a visual-spatial similarity structure on foreign vocabulary learning*.
- Grainger, J., Whitney, C., et al. (2004). Does the huamn mnid raed wrods as a wlohe? *Trends in cognitive sciences*, *8*(2), 58–59.
- Hannagan, T., Dupoux, E., & Christophe, A. (2011). Holographic string encoding. *Cognitive science*, *35*(1), 79–118.
- Jensen, J. M. (2013). *Damerau-levenshtein edit distance calculator in python*. Retrieved from <https://gist.github.com/badocelot/5327337> (accessed: 21.06.2022)
- Kokoska, S., & Zwillinger, D. (2000). *Crc standard probability and statistics tables and formulae*. Crc Press.
- Kriegeskorte, N., & Mur, M. (2012). Inverse mds: Inferring dissimilarity structure from multiple item arrangements. *Frontiers in psychology*, *3*, 245.
- Laufer, B. (1988). The concept of ‘synforms’ (similar lexical forms) in vocabulary acquisition. *Language and Education*, *2*, 113–132.
- Leoné, F. (in prep.). *Computational modeling as a prescriptive bridge between neuroscience and education*.

- Levenshtein, V. I., et al. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady* (Vol. 10, pp. 707–710).
- Llach, M. d. P. A. (2015). Lexical errors in writing at the end of primary and secondary education: Description and pedagogical implications. *Porta Linguarum: revista internacional de didáctica de las lenguas extranjeras*(23), 109–124.
- Mindsort. (2022). Retrieved from <https://mindsort.tweede.golf/> (accessed: 17.05.2022)
- Oude Kempers, E. (2020). *Effects of orthographic similarity mapping on second language acquisition*. (unpublished)
- Perea, M., & Lupker, S. J. (2003). Does judge activate court? transposed-letter similarity effects in masked associative priming. *Memory & Cognition*, 31(6), 829–841.
- Peressotti, F., & Grainger, J. (1999). The role of letter identity and letter position in orthographic priming. *Perception & Psychophysics*, 61(4), 691–706.
- Perfetti, C. (2007). Reading ability: Lexical quality to comprehension. *Scientific studies of reading*, 11(4), 357–383.
- Perfetti, C. A., & Hart, L. (2002). The lexical quality hypothesis. *Precursors of functional literacy*, 11, 67–86.
- Rijksoverheid. (n.d.). *Welke vreemde talen krijg ik in de onderbouw van het voortgezet onderwijs?* Retrieved from <https://www.rijksoverheid.nl/onderwerpen/voortgezet-onderwijs/vraag-en-antwoord/vreemde-talen-onderbouw-voortgezet-onderwijs> (accessed: 23.08.2022)
- Wagner, R. A., & Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1), 168–173.
- Whitney, C. (2008). Supporting the serial in the serial model. *Language and Cognitive Processes*, 23(6), 824–865.
- Whitney, C., Bertrand, D., & Grainger, J. (2012). On coding the position of letters in words: A test of two models. *Experimental psychology*, 59(2), 109.
- Wiki statistiek Amsterdam UMC. (n.d.). *Keuze toets*. Retrieved from https://wikistatistiek.amc.nl/KEUZE_TOETS (accessed: 17.08.2022)
- Wikipedia. (n.d.). *Levenshtein distance*. Retrieved from https://en.wikipedia.org/wiki/Levenshtein_distance (accessed: 21.06.2022)
- Zafar, S., & Meenakshi, K. (2012). Individual learner differences and second language acquisition: a review. *Journal of Language Teaching & Research*, 3(4).

Appendix A

Explanation of the experiment



Uitleg experiment

In dit experiment zal je twee woordenlijsten zien met elk 15 woorden in Swahili. Deze ga je ruimtelijk sorteren in de applicatie MindSort, een applicatie om vreemde talen te leren.

Volgende

Figure A.1: General explanation of the experiment in Dutch. Can be translated to: "Explanation of experiment. In this experiment you will see two word lists, each containing 15 words in Swahili. You will sort these spatially in the MindSort application, a tool for learning foreign languages."



Sorteren woordenlijst 1

Bij dit onderdeel zal je woordenlijst 1 ruimtelijk sorteren gebaseerd op gelijkenis. Voor het sorteren vraag ik je de woorden te slepen naar posities zodat woorden die je op elkaar vindt lijken dichter bij elkaar staan en woorden die je minder op elkaar vind lijken verder van elkaar af staan. Zet het neer op een manier die voor jou fijn zou zijn om de woorden te leren. Let daarbij op alle relatieve afstanden, dus zowel op korte, als op lange afstand. Zorg dat de woorden over het gehele scherm verdeeld zijn.

Ga naar de experimentleider en geef daar aan dat je woordenlijst 1 ruimtelijk moet sorteren gebaseerd op je eigen sortering. Jouw onderzoeksnummer is [REDACTED]

Keer daarna terug naar deze vragenlijst en druk op volgende.

Figure A.2: Explanation of the task for word list 1. Can be translated to: "Sorting word list 1. In this section, you will sort word list 1 spatially based on similarity. For sorting, I ask you to drag the words to positions so that words you find similar are closer together and words you find less similar are further apart. Put it in a way that would be nice for you to learn the words. Pay attention to all relative distances, both short and long. Make sure the words are spread out over the whole screen. Go to the experiment leader and tell them you need to sort word list 1 spatially based on your own sorting. Your research number is: ... Then return to this questionnaire and press next."



Sorteren woordenlijst 2

Bij dit onderdeel zal je woordenlijst 2 ruimtelijk sorteren gebaseerd op gelijkenis. Voor het sorteren vraag ik je de woorden te slepen naar posities zodat woorden die je op elkaar vindt lijken dicht bij elkaar staan en woorden die je minder op elkaar vindt lijken verder van elkaar af staan. Zet het neer op een manier die voor jou fijn zou zijn om de woorden te leren. Let daarbij op alle relatieve afstanden, dus zowel op korte, als op lange afstand. Zorg dat de woorden over het gehele scherm verdeeld zijn.

Ga naar de experimentleider en geef daar aan dat je woordenlijst 2 ruimtelijk moet sorteren gebaseerd op je eigen sortering. Jouw onderzoeksnummer is: XXXXXXXXXX

Keer daarna terug naar deze vragenlijst en druk op volgende.

Figure A.3: Explanation of the task for word list 2. Can be translated to: "Sorting word list 2. In this section, you will sort word list 2 spatially based on similarity. For sorting, I ask you to drag the words to positions so that words you find similar are closer together and words you find less similar are further apart. Put it in a way that would be nice for you to learn the words. Pay attention to all relative distances, both short and long. Make sure the words are spread out over the whole screen. Go to the experiment leader and tell them you need to sort word list 2 spatially based on your own sorting. Your research number is: ... Then return to this questionnaire and press next."

Appendix B

Experimental outcomes

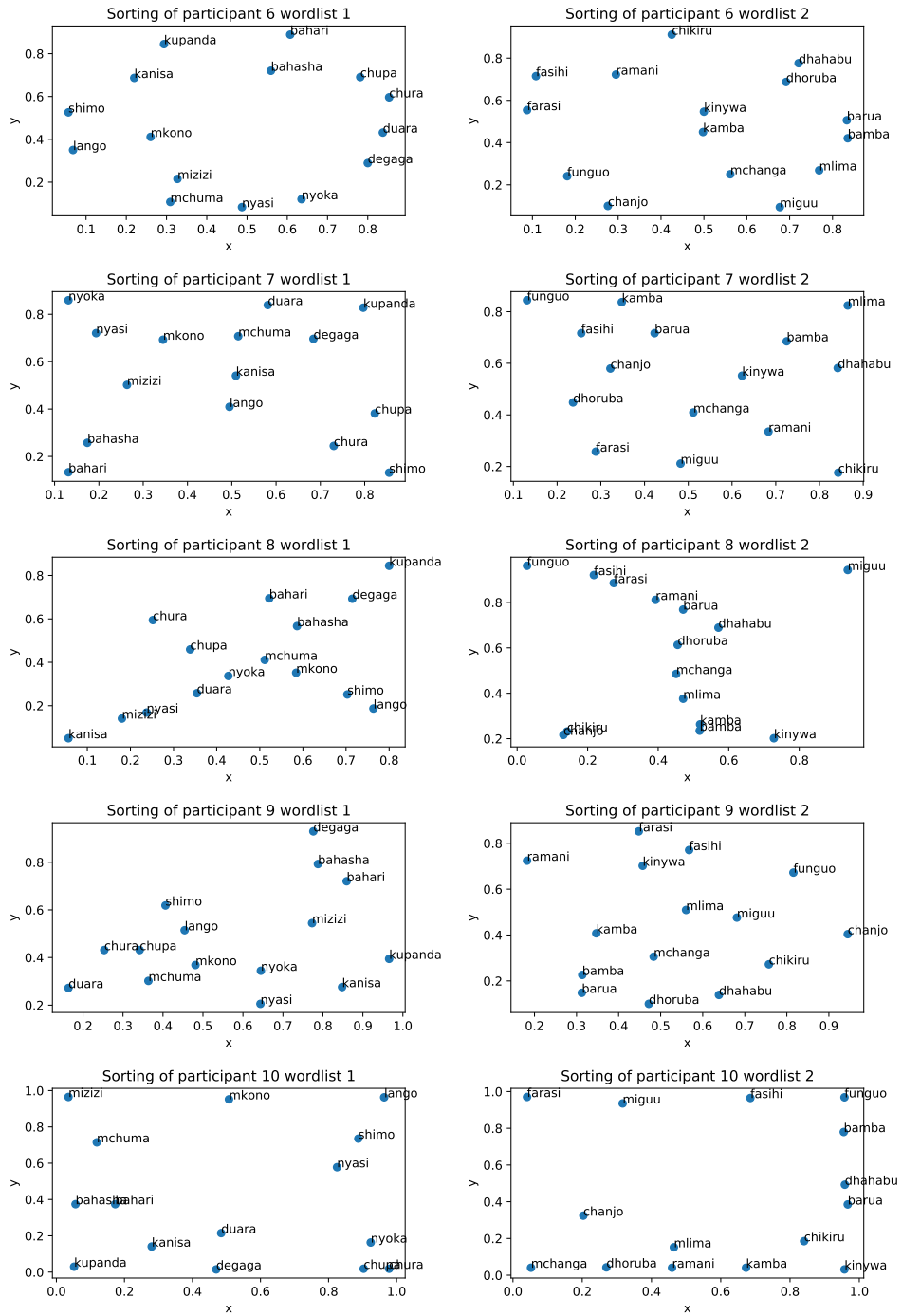


Figure B.2: Experimental data of participant 6-10 visualized using Python.

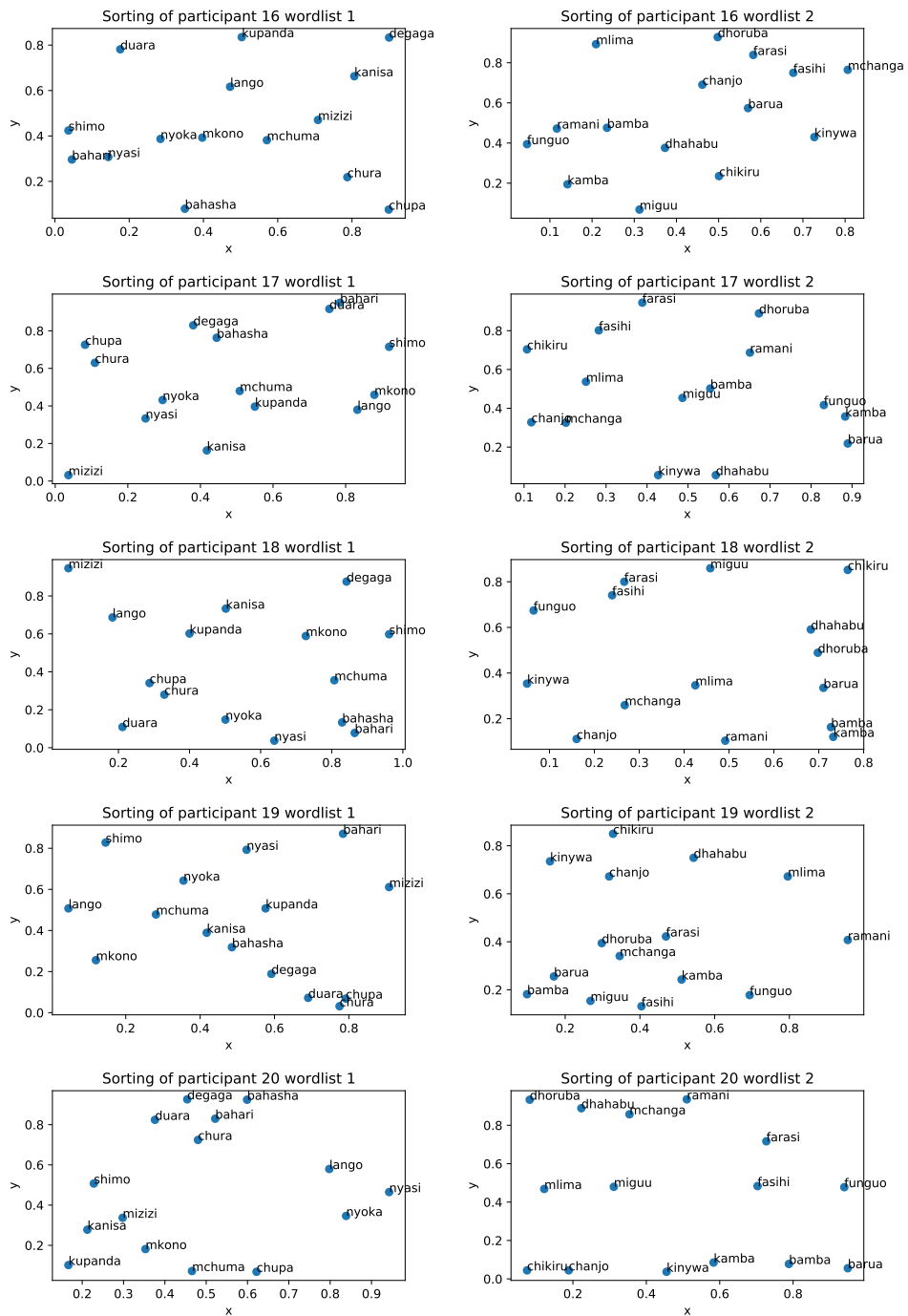


Figure B.4: Experimental data of participant 16-20 visualized using Python.

Appendix C

Code

```
# **Import experiment data**

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Get list with Response IDs
resp = pd.read_csv(r"C:\Users\aline\OneDrive\Documenten\Radboud
    \2021-2022\Thesis\Experiment\Data_analyse\Resultaten_
    experiment_Qualtrics\Qualtrics_data_20220621.csv")
resp_id = resp['ResponseId'].to_numpy()
resp_id = resp_id[2:]

# Make dataframe dictionary with all responses of wordlist 1
sw1_exp_coord = {}
for i in range(len(resp_id)):
    sw1 = pd.read_csv(r"C:\Users\aline\OneDrive\Documenten\
        Radboud\2021-2022\Thesis\Experiment\Data_analyse\
        Resultaten_experiment_MindSort\Exp_AB_Sw1-{}.csv".format(
            resp_id[i]))
    sw1 = sw1[sw1.TAG == 'sw']
    sw1_exp_coord[i] = sw1.loc[:,['ITEM', 'X', 'Y']]
    sw1_exp_coord[i].index = np.arange(1,16)

# Make dataframe dictionary with all responses of wordlist 2
sw2_exp_coord = {}
for i in range(len(resp_id)):
    sw2 = pd.read_csv(r"C:\Users\aline\OneDrive\Documenten\
        Radboud\2021-2022\Thesis\Experiment\Data_analyse\
        Resultaten_experiment_MindSort\Exp_AB_Sw2-{}.csv".format(
            resp_id[i]))
    sw2 = sw2[sw2.TAG == 'sw']
    sw2_exp_coord[i] = sw2.loc[:,['ITEM', 'X', 'Y']]
    sw2_exp_coord[i].index = np.arange(1,16)

# **Import wordlists**
```

```

# In[2]:

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Import word lists to numpy arrays
w11_words = pd.read_csv(r"C:\Users\aline\OneDrive\Documenten\
    Radboud\2021-2022\Thesis\Experiment\Woordenlijst\
    Woordenlijst1_def.20220616.csv")
w11_words = w11_words[w11_words.TAG == 'sw']
w11_words = w11_words['ITEM'].to_numpy()
w12_words = pd.read_csv(r"C:\Users\aline\OneDrive\Documenten\
    Radboud\2021-2022\Thesis\Experiment\Woordenlijst\
    Woordenlijst2_def.20220616.csv")
w12_words = w12_words[w12_words.TAG == 'sw']
w12_words = w12_words['ITEM'].to_numpy()

# **Plot experiment sortings**

# In[3]:

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Visualize experimental data
fig, axs = plt.subplots(20, 2, figsize=(10, 80))
for i in range(20):
    for j in range(2):
        exp_coord = vars()["sw{}_exp_coord".format(j+1)]
        axs[i, j].scatter(exp_coord[i]['X'].to_numpy(),
            exp_coord[i]['Y'].to_numpy())
        words = vars()["wl{}_words".format(j+1)]
        for z, label in enumerate(words):
            axs[i, j].annotate(label, (exp_coord[i]['X'].to_numpy()
                [z], exp_coord[i]['Y'].to_numpy()[z]))
            axs[i, j].set_title("Sorting of participant {} wordlist {}"
                .format(i+1, j+1))

for ax in axs.flat:
    ax.set(xlabel='x', ylabel='y')

plt.subplots_adjust(wspace=0.2, hspace=0.3)
plt.savefig('Part_sort.pdf')
plt.show()

# **Calculate Levenshtein and normalized Levenshtein distances**

```

```

# In[4]:

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Calculate Levenshtein distances
# Based on psuedocode of Wagner Fischer algorithm on Wikipedia
# : https://en.wikipedia.org/wiki/Wagner%E2%80%93Fischer\_algorithm (accessed: 30.5.2022)
def lev_distance(a, b):

    # make empty distances matrix
    n = len(a)+1
    m = len(b)+1
    dist_mat = np.zeros((n), (m))

    # fill first row and column
    for i in range(n):
        dist_mat[i, 0] = i
    for j in range(m):
        dist_mat[0, j] = j

    # apply levenshtein
    for j in range(1, m):
        for i in range(1, n):
            if a[i-1] == b[j-1]:
                subCost = 0
            else:
                subCost = 1

            dist_mat[i, j] = min(dist_mat[i-1, j] + 1, # deletion
                                dist_mat[i, j-1] + 1, #
                                insertion
                                dist_mat[i-1, j-1] + subCost) #
                                substitution

    return dist_mat[n-1, m-1]

# make empty distances matrix for word pairs
w11_lev_dist = np.zeros((len(w11_words), len(w11_words)))
w11_levnorm_dist = np.zeros((len(w11_words), len(w11_words)))
w11_lev_dist_list = pd.DataFrame({'itemPair': [], 'distance': []})

# calculate distances
for i in range(len(w11_words)):
    for j in range(len(w11_words)):
        w11_lev_dist[i, j] = lev_distance(w11_words[i], w11_words
                                          [j])

    # normalize Levenshtein distances by dividing through
    # the maximum possible distance between the two words
    w11_levnorm_dist[i, j] = lev_distance(w11_words[i],
                                          w11_words[j]) / max(len(w11_words[i]), len(w11_words[j]

```

```

    ))

    # calculate distance list
    pair_dist = pd.DataFrame({'itemPair': [w11_words[i]+'-'+w11_words[j]], 'distance': [w11_lev_dist[i,j] ]})
    w11_lev_dist_list = pd.concat([w11_lev_dist_list,
        pair_dist], ignore_index=True, axis=0)
# print(w11_lev_dist_list)

# make empty distances matrix for word pairs
w12_lev_dist = np.zeros((len(w12_words), len(w12_words)))
w12_levnorm_dist = np.zeros((len(w12_words), len(w12_words)))

# calculates distances
for i in range(len(w12_words)):
    for j in range(len(w12_words)):
        w12_lev_dist[i,j] = lev_distance(w12_words[i], w12_words[j])

        # normalize Levenshtein distances by dividing through
        # the maximum possible distance between the two words
        w12_levnorm_dist[i,j] = lev_distance(w12_words[i],
            w12_words[j])/max(len(w12_words[i]), len(w12_words[j]))

# **Damerau-Levenshtein**

# In [5]:

# Damerau-Levenshtein edit distance implementation by James M.
# Jensen II: https://gist.github.com/badocelot/5327337 (
# accessed 30.5.2022)
# Based on pseudocode from Wikipedia: https://en.wikipedia.org/
# wiki/Damerau-Levenshtein\_distance

def damerau_levenshtein_distance(a, b):
    # "Infinity" — greater than maximum possible edit distance
    # Used to prevent transpositions for first characters
    INF = len(a) + len(b)

    # Matrix: (M + 2) x (N + 2)
    matrix = [[INF for n in range(len(b) + 2)]]
    matrix += [[INF] + list(range(len(b) + 1))]
    matrix += [[INF, m] + [0] * len(b) for m in range(1, len(a)
        + 1)]

    # Holds last row each element was encountered: 'DA' in the
    # Wikipedia pseudocode
    last_row = {}

    # Fill in costs
    for row in range(1, len(a) + 1):

```

```

# Current character in 'a'
ch_a = a[row-1]

# Column of last match on this row: 'DB' in pseudocode
last_match_col = 0

for col in range(1, len(b) + 1):
    # Current character in 'b'
    ch_b = b[col-1]

    # Last row with matching character; 'i1' in
    # pseudocode
    last_matching_row = last_row.get(ch_b, 0)

    # Cost of substitution
    cost = 0 if ch_a == ch_b else 1

    # Compute substring distance
    matrix[row+1][col+1] = min(
        matrix[row][col] + cost, # Substitution
        matrix[row+1][col] + 1, # Addition
        matrix[row][col+1] + 1, # Deletion

        # Transposition
        matrix[last_matching_row][last_match_col]
            + (row - last_matching_row - 1) + 1
            + (col - last_match_col - 1))

    # If there was a match, update last_match_col
    # Doing this here lets me be rid of the 'j1'
    # variable from the original pseudocode
    if cost == 0:
        last_match_col = col

# Update last row for current character
last_row[ch_a] = row

# Return last element
return matrix[-1][-1]

# make empty distances matrix for word pairs wordlist 1
wl1_DL_dist = np.zeros((len(wl1_words), len(wl1_words)))

# calculates distances
for i in range(len(wl1_words)):
    for j in range(len(wl1_words)):
        wl1_DL_dist[i, j] = damerau_levenshtein_distance(
            wl1_words[i], wl1_words[j])

# make empty distances matrix for word pairs wordlist 2
wl2_DL_dist = np.zeros((len(wl2_words), len(wl2_words)))

# calculates distances
for i in range(len(wl2_words)):

```

```

    for j in range(len(wl2_words)):
        wl2_DL_dist[i, j] = damerau_levenshtein_distance(
            wl2_words[i], wl2_words[j])

# **Weighted open bigram model**

# In [6]:

# Implementation of the weighted open bigram model taken from
# Dijkstra, R. (2017). Modelling orthographic similarity
# between words. (unpublished)

class BigramModel:
    """Implementing the weighted open bigram model.

    Implementation of weighted open bigrams following the paper
    from Whitney
    (2008).

    Args:
    target: A string representing the target.
    prime: A string representing the prime.

    Attributes:
    target: A string representing the target.
    prime: A string representing the prime.
    weight_bigrams: a list of floats representing the
    weights for
    different levels of separation.
    """

    def __init__(self, target, prime):
        self.target = target
        self.prime = prime
        self.weight_bigrams = [1.0, 0.8, 0.4] #weights from
            whitney (2008)
        self.activated_bigrams = []
        self.target_bigrams = self.make_target_bigrams()
        self.similarity_score = self.calculate_similarity_score()

    def make_target_bigrams(self):
        """Make the target bigrams.

        Make a list for each level of separation between the
        bigrams, each
        list containing a list of bigrams at that level.

        Returns:
        list of list of strings: The return value.
        representing the bigrams
        at different levels of separation.
    """

```

```

"""
target_bigrams = []
for separation_level in range(3):
    bigrams = self.make_bigrams(separation_level, self.
        target)
    target_bigrams.append(bigrams)
unique_target_bigrams = self.delete_double_bigrams(
    target_bigrams)
return unique_target_bigrams

def delete_double_bigrams(self, bigrams):
    """Delete the double bigrams

    Delete the bigrams that are double and do not have the
    heighest
    weights from the bigrams in the target.

    Args:
        bigrams: A list of lists of bigrams representing the
            bigrams at
            different levels of separation.
    Returns:
        list of lists of bigrams. The return value
        representing unique
        bigrams at different levels of separation.
    """
    not_doubled = []
    for level in bigrams:
        for bigram in level:
            if bigram in not_doubled:
                level.remove(bigram)
            else:
                not_doubled.append(bigram)
    return bigrams

def make_bigrams(self, separation, string):
    """Determine bigrams.

    Determine the bigrams at a certain separation level from
    a certain
    string.

    Args:
        separation: An integer coding the level of
            separation for the
            letters in the bigram in their string.
        string: A string representing the string from which
            the bigrams will
            be determined.
    Returns:
        list of strings: The return value. the bigrams from
        a certain string
        at a certain level of separation.
    """

```

```

bigrams = []
max_first_letter_position = len(string)-separation-1
for first_letter_position in range(
    max_first_letter_position):
    last_letter_position = first_letter_position + 1 +
        separation
    first_letter = string[first_letter_position]
    last_letter = string[last_letter_position]
    bigram = first_letter+last_letter
    bigrams.append(bigram)
return bigrams

def calculate_similarity_score(self):
    """Calculate the similarity score.

    Calculate the similarity score by dividing the raw score
    for comparing
    the target with the prime by the raw score of comparing
    the target with
    itself.

    Returns:
        float: The return value. Representing the similarity
            score.
    """
    max_raw_score = self.calculate_raw_match(self.target ,
        self.target)
    raw_score = self.calculate_raw_match(self.target , self.
        prime)
    similarity_score = raw_score/max_raw_score
    return similarity_score

def calculate_raw_match(self , target , compare):
    """Calculate the raw score for the match between two
    strings.
    Calculate the match by adding the product of bigram
    weights from bigrams
    matching between the two strings bigrams. This is done
    for all three
    different levels of separation.
    Then the edge bigram score is added and finally the
    score is normalized.

    Args:
        target: A string representing the target.
        compare: A string representing the string comparing
            to the target.

    Returns:
        float: The return value. Representing the raw match
            score.
    """
    match = 0.0
    self.activated_bigrams = []
    for separation in range(3):

```

```

        w = self.weight_bigrams[separation]
        compare_bigrams = self.make_bigrams(separation ,
            compare)
        separation_match = w * self.sum_matching(
            compare_bigrams)
        match = match + separation_match
    match = match + self.add_edge_score(target , compare)
    normalisation_factor = 20.0/(20.0+len(compare))
    normalized_match = normalisation_factor*match
    return normalized_match

def sum_matching(self , bigrams):
    """ sum the matching bigrams.

    Args:
        bigrams: A list of strings representing the bigrams
            to be matched.

    returns:
        float: The return value. Representing the weighted
            sum of the
            matching bigrams.
    """
    score = 0.0
    for bigram in bigrams:
        for separation in range(3):
            weight = self.weight_bigrams[separation]
            if bigram in self.target_bigrams[separation]:
                # Only the highest (and first) bigram can
                # contribute to the
                # final score. So if a bigram has already
                # been used, the
                # newer and lower (or equal) weighted bigram
                # will not
                # contribute.
                if bigram not in self.activated_bigrams:
                    score = score + weight
                    self.activated_bigrams.append(bigram)
    return score

def add_edge_score(self , target , compare):
    """ Calculate the edge score for two strings.

    Args:
        target: A string representing the target.
        compare: A string representing the comparing word
            which edges have
            to match with the target.

    Returns:
        float: The return value. representing the sum of the
            weight of the
            matching edge bigrams.
    """

```

```

        first_edge = 2.0 * (target[0] == compare[0])
        last_edge = 2.0 * (target[-1] == compare[-1])
        score = first_edge + last_edge
        return score

# Calculating weighted open bigram scores for word pairs of both
# word lists

# make empty distances matrix for word pairs wordlist 1
w11_BM_dist = np.zeros((len(w11_words), len(w11_words)))

# calculates distances
for i in range(len(w11_words)):
    for j in range(len(w11_words)):
        w11_BM_dist[i, j] = 1-BigramModel(w11_words[i], w11_words
            [j]).similarity_score
# make the matrix symmetric by taking the average value
for i in range(len(w11_words)):
    for j in range(i+1, len(w11_words)):
        dist = (w11_BM_dist[i, j] + w11_BM_dist[j, i])/2
        w11_BM_dist[i, j] = dist
        w11_BM_dist[j, i] = dist

# make empty distances matrix for word pairs wordlist 2
w12_BM_dist = np.zeros((len(w12_words), len(w12_words)))

# calculates distances
for i in range(len(w12_words)):
    for j in range(len(w12_words)):
        w12_BM_dist[i, j] = 1-BigramModel(w12_words[i], w12_words
            [j]).similarity_score

# make the matrix symmetric by taking the average value
for i in range(len(w12_words)):
    for j in range(i+1, len(w12_words)):
        dist = (w12_BM_dist[i, j] + w12_BM_dist[j, i])/2
        w12_BM_dist[i, j] = dist
        w12_BM_dist[j, i] = dist

# **Extended Spatial coding**

# In [7]:

# Implementation of the weighted open bigram model taken from
# Dijkstra, R. (2017). Modelling orthographic similarity
# between words. (unpublished)

import numpy as np
import math

class SpatialModelExtended:
    """Implementation of the extended version of spatial coding.

```

Like the implementation of spatial coding, the implementation is based on the first 16 equations in Daviss paper of spatial coding (2010). This model is extended with a letter doubling function, based on the paper of Fischer–Baum.

Args:

*target: A string containing the target.
prime: A string containing the prime.
ELM: A boolean indicating whether ELM will be used.*

Attributes:

*target: A string containing the target.
prime: A string containing the prime.
ELM: A boolean indicating if ELM will be used.
sigma_0: A float value set for sigma_0
k_0: A float value set for k_0
sigma: A float representing a level of uncertainty.
Based on sigma_0,
k_0 and the length of the target.
banks_of_receivers: A list of the banks of receivers
belonging to the
target.
weight: A float value for the size of the weights for
the ELM and the
raw match score.
double_weight: A float value used for combining the
match score with the
double letter marker.
similarity_score: the score obtained comparing a target
with a prime*

"""

```
def __init__(self, target, prime, ELM=True):
    self.target = target
    self.prime = prime
    self.ELM = ELM
    self.banks_of_receivers = []

    # Determine sigma.
    self.sigma_0 = 0.25 # As stated in table 3 in spatial
        coding.
    self.k_0 = 0.19 # As stated in table 3 in spatial coding
    self.sigma = self.calculate_sigma(len(prime))

    # Determine weights
    self.weight = 1.0/(len(target))
    if ELM:
        self.weight = 1.0/(len(target)+2)
    self.double_weight = 1.0 / self.sigma
```

```

# Set up the receivers
self.initialise_receivers(target, prime, self.sigma)

# Calculate the similarity.
self.similarity_score = self.match()

def calculate_sigma(self, length):
    """Calculate the value for sigma.

    Equation 3 in spatial coding. The assumption that longer
    strings result
    in more uncertainty and thus bigger sigmas is
    implemented here.

    Args:
        length: An integer counting the number of letters in
        a word.

    Returns:
        float: The return value. The value for sigma.
    """
    sigma = self.sigma_0 + self.k_0*length
    return sigma

def initialise_receivers(self, target, prime, sigma):
    """Determines and correctly activates the receivers.
    Determines what banks of receivers will belong to the
    target. Then
    activates the clones in the banks based on the letters
    of the prime.
    After that, determines the resonating phase and finally
    inhibits the
    clones that fit the resonating phase less.

    Args:
        target: A string containing the target.
        prime: A string containing the prime.
        sigma: A float representing the value of sigma.
    """

    # Determine the Banks of Receivers.
    for position, identity in enumerate(target):
        double = self.letter_is_double(position, identity)
        new_bank = ESCBank(identity, len(prime), position,
            sigma, double)
        self.banks_of_receivers.append(new_bank)

    # Activate the receivers based on the letters of the
    prime.
    for position, identity in enumerate(prime):
        for bank_pos, bank in enumerate(self.
            banks_of_receivers):
            bank.activate_receivers(identity, position)

    # Determine the resonating Phase

```

```

max_range = min(len(target), len(prim)) + 1
res_phase = self.find_resonating_phase(max_range)

# inhibit losing receivers within a bank
for bank in self.banks_of_receivers:
    bank.update_receivers(res_phase)

# inhibit losing receivers between banks
for bank in self.banks_of_receivers:
    if bank.win_rec_pos is not None:
        bank.cross_bank_winner(self.banks_of_receivers,
                                res_phase)

def letter_is_double(self, position, letter):
    """Checks whether a letter is double.

    Check if previous or next letter of the targetword is
    the same as the
    current letter.

    Args:
        position: integer count for the position of the
            letter that is
            checked.
        letter: String with the identity of the letter that
            is checked.

    Returns:
        bool: The return value. True if the previous or next
            letter has the
            same identity. False otherwise.
    """
    double = False
    if position > 0: # Else there is no previous letter
        if self.target[position-1] == letter:
            double = True
    if position < len(self.target) - 1: # Else there is no
        next letter
        if self.target[position+1] == letter:
            double = True
    return double

def find_resonating_phase(self, max_dist):
    """Determining the resonating phase.

    Implementing the determination of the resonating phase:
    According to
    Davis the resonating phase corresponds to the peak of
    the superposition.

    Args:
        max_dist: An integer giving the maximum distance

    Returns:

```

```

        integer: The return value. Represents the position
            with the highest
            activation.
    """
    # initialise
    min_dist = -1*max_dist
    best_pos = min_dist
    best_score = self.super_position(min_dist)

    # Find the position with the highest peak (= summed
    activation)
    for position in range(min_dist, max_dist):
        score = self.super_position(position)
        if score > best_score:
            best_pos = position
            best_score = score
    return best_pos

def super_position(self, position):
    """Calculating the superposition.

    Equation 15 in spatial coding. Following from equation
    10: The
    superposition function is found by summing across the
    receiver functions
    for each of the target AZs receivers .
    In this extended version of spatial coding, the
    superposition function
    is extended with a double letter marker.
    This DLM will increase the activation of a bank if the
    bank represents a
    doubled letter and the prime contains doubled letters.

    Args:
        position: An integer representing the relative
            position where the
            superposition function will add.

    Returns:
        float: The return value. The score obtained by
            summing the
            activation of the receiver clone nodes at a certain
            position.
    """
    super_position_score = 0
    for bank in self.banks_of_receivers:
        weight = self.weight
        dw = self.double_weight

        # Calculating the activation of the receiver in this
        bank.
        if self.prime_has_double() and bank.double: #
            increase activation.
            new_weight = weight * (1.0 - dw)
            extra_act = new_weight * dw

```

```

        activated_receiver_score = new_weight*bank.
            receiver(position)
        activated_receiver_score =
            activated_receiver_score + extra_act
    else: # No increasing activation.
        activated_receiver_score = weight*bank.receiver(
            position)

    # Adding the activation of the receiver in this bank
    super_position_score = super_position_score +
        activated_receiver_score
    return super_position_score

def prime_has_double(self):
    """Check if there is a double letter in the input

    Returns:
        bool: The return value. True if the prime has
            doubled letters. False
            otherwise.
    """
    old_letter = ''
    for letter in self.prime:
        if letter == old_letter:
            return True
        old_letter = letter
    return False

def match(self):
    """Calculating the similarity value between a target and
        a prime.

    Equation 14 in spatial coding. Combining the
        superposition at the
        resonating phase with the score of the end letter
        marking.

    Returns:
        float: The return value. Representing the similarity
            value between a
            target and a prime.
    """
    max_dist = min(len(self.target), len(self.prime)) + 1
    res_phase = self.find_resonating_phase(max_dist)
    match_score = self.super_position(res_phase)+self.
        ELM_score()
    return match_score

def ELM_score(self):
    """Calculating the external letter match.

    Calculating the match between the external letters of
        the prime and the

```

external letters of the target with the product of the amount of matches and the weight for the external letters.

Returns:

float: The return value. represents the activation given by ELM.

"""

if self.ELM **is** False:

return 0

score = 0

if self.target[0] == self.prime[0]: # Match between first letters

score = self.weight

if self.target[-1] == self.prime[-1]: # Match between last letters.

score = score + self.weight

return score

class ESCBank:

"""The class representing a bank of receivers.

A bank of receiver contains different clone nodes that code for different relative distances of letters. It can determine the winning receiver clone node in the bank itself and the winning receiver node compared to the other banks of receivers.

Args:

identity: A string representing the identity of the letter the bank represents.

n_receivers: An integer coding the number of receiver clones in this

bank.

position: An integer coding the position of the letter

the bank represents.

sigma: A float representing the value for sigma.

double: A bool. True if the letter in this bank is

doubled with another

letter.

Attributes:

identity: A string representing the identity of the letter the bank

represents.

position: An integer coding the position of the letter

the bank

represents.

sigma: A float representing the value for sigma.

double: A bool. True if the letter in this bank is

doubled with another

```

    letter.
    receivers: A list containing the receiver clone nodes.
    win_rec_pos: an integer pointing to the position of the
        winning
        receiver clone node.
    """
def __init__(self, identity, n_receivers, position, sigma,
double):
    self.identity = identity
    self.position = position
    self.sigma = sigma
    self.double = double
    self.receivers = []
    for pos_receiver in range(n_receivers):
        self.receivers.append(ESCReceiver(pos_receiver,
            False, sigma))
    self.win_rec_pos = 0
    self.first_active = None

def activate_receivers(self, prime_letter, prime_pos):
    """Activates the receivers.

    This function will activate those receiver clone nodes
    that node are
    connected with the prime_letter through an active
    position-specific
    channel. If the identities of the letters is the same,
    the corresponding
    clone node is Activated.

    Args:
        prime_letter: A string representing a letter of the
            prime.
        prime_pos: An integer coding the position in the
            position-specific
            channel.
    """
    if self.identity == prime_letter:
        self.receivers[prime_pos].activate(self.position)
        if self.first_active == None:
            self.first_active = prime_pos
            self.win_rec_pos = self.first_active
def update_receivers(self, res_phase):
    """Determines which receivers will still be activated.

    Evaluates the receivers with each others and only leaves
    the highest
    activated.

    Args:
        res_phase: The resonating phase determining the
            position the
            receivers have to be closest to.
    """

```

```

if self.first_active == None: # With no activated
    receivers, none have
    return # to be inhibited.
win_id = self.first_active
for id_r, r in enumerate(self.receivers):
    if self.closer(self.receivers[win_id].pos, r.pos,
        res_phase):
        self.receivers[win_id].lost()
        self.win_rec_pos = id_r
        self.activated_clone_activation =
            self.receiver(self.
                position)
        win_id = id_r
def cross_bank_winner(self, bank, res_phase):
    """Inhibits all but the highest activated clone across
        the banks.

    Checks for its active clone if there are contestants
        across the other
    banks. And if so, will inhibit the least activated clone
        .

    Args:
        bank: A list containing all receivers in the same
            bank.
        res_phase: An integer coding the resonating phase.
    """
    if self.first_active == None: # With no activated
        receivers, none have
        return # to be inhibited.
    for other_rec in bank:
        if other_rec.first_active == None:
            break # If there is no competition, this bank
                wins.
        if self.identity == other_rec.identity:
            if self.position == other_rec.position:
                break # A node does not inhibit itself.
            # Inhibit the receiver with the least activation
                .
            rel_self = self.win_rec_pos - self.position
            if other_rec.win_rec_pos is not None: # This was
                added by Aline Boels to prevent a TypeError
            rel_other = other_rec.win_rec_pos - other_rec
                .position
            if self.closer(rel_self, rel_other,
                res_phase):
                self.inhibit_receiver(self.win_rec_pos)
            return
        else:
            other_rec.inhibit_receiver(self.
                win_rec_pos)

def inhibit_receiver(self, position):
    """Inhibits a receivers

```

Inhibit a receiver by calling its function that notifies it to be inhibited and by resetting the info about it in the bank

Args:

position: An integer coding which receiver is to be inhibited.

"""

```
self.receivers[position].lost_across()
self.win_rec_pos = None
self.activated_receiver_activation = 0
```

```
def receiver(self, position):
```

"""*Calculate the activation of the receivers.*

Calculates the activation of the receivers in this bank at a certain position. The activation is equal to the activation of the highest activated receiver node.

Args:

position: An integer coding the position where the letter represented by this bank is supposed to be.

Returns:

float: The return value. The activation value for the winning receiver clone node at the position coded by position.

"""

```
highest_score = 0
```

```
for receiver in self.receivers:
```

```
    highest_score = max(highest_score, receiver.receiver(position))
```

```
return highest_score
```

```
def closer(self, current, contender, res_phase):
```

"""*Determine which position is closer.*

Determines which position is closer to the right position.

Args:

current: Integer coding the position of the current receiver.

contender: Integer coding the position of the contending receiver

res_phase: Integer coding the position to which the current and

contending receiver are compared.

```

Returns:
    bool: the return value. True if the contender
          receiver is closer to
          the resonating phase than the current receiver.
          False
          otherwise.
"""
if contender is None: # No active receiver -> no closer
    receiver
    return False
if current is None: # No active receiver -> The other
    receiver is closer
    return True
current_distance_relative = abs(current - res_phase)
contender_distance_relative = abs(contender - res_phase)
if current_distance_relative >
    contender_distance_relative:
    return True
return False

class ESCReceiver:
    """The Receiver clone node class for extended spatial coding

    Each receiver clone node is connected to a certain letter
    node through a
    position-specific channel, to other nodes in the same bank
    and to nodes
    with the same position-specific channel in other banks.
    They can be activated and inhibited and they can return
    activation based on
    the relative distance from the position they code for.

    Args:
        position: An integer coding for the position-specific
                  channel the
                  receiver is connected to.
        activated: A bool indicating whether this receiver is
                  activated.
        sigma: A float indicating the value for sigma in the
               gaussian function
               that determines the activation of the receiver.

    Attributes:
        position: An integer coding for the position-specific
                  channel the
                  receiver is connected to.
        activated: A bool indicating whether this receiver is
                  activated.
        sigma: A float indicating the value for sigma in the
               gaussian function
               that determines the activation of the receiver.
        pos: An integer coding for the relative distance between
             the position in
             the position-specific channel and the position of the

```

```

        receiver bank.
"""

def __init__(self, position, activated, sigma):
    self.activated = activated
    self.position = position
    self.sigma = sigma
    self.pos = None

def lost(self):
    """Inhibits the node. """
    self.activated = False
    self.pos = None

def lost_across(self):
    """Inhibits the node. """
    self.activated = False
    self.pos = None

def activate(self, difference):
    """Activates the node.

    Activates the node and calculates the relative distance
    between the
    position in the position-specific channel and the
    position of the
    receiver bank.

    Args:
    difference: An integer representing the position of
    the bank.
    """
    self.pos = self.position - difference
    self.activated = True

def receiver(self, position):
    """Calculate the activation of the receiver clone node.

    Calculate the activation of the receiver clone node at a
    given position
    according to equation 9 in spatial coding. With the
    difference of no
    delay Calculated. Delay is not implemented because we do
    not need the
    time factor.

    Args:
    position: An integer coding for the position where
    the activation is
    measured.

    Returns:
    float: The return value. Indicating the activation
    of the receiver
    clone node at a given position.
    """

```

```

"""
return self.signal(position)
def signal(self, position):
    """Calculate the activation of the receiver clone node.

    Calculate the activation of the receiver clone node at a
        given position
    according to equation 4 in spatial coding.

    Args:
        position: An integer coding for the position where
            the activation is
            measured.

    Returns:
        float: The return value. Indicating the activation
            of the receiver
            clone node at a given position.
    """
    return self.activation()*self.spatial(position)

def activation(self):
    """Calculate if a clone is activated.

    Return:
        integer: The return value. 1 if the clone is
            activated. 0 otherwise.
    """
    if self.activated is True:
        return 1
    return 0

def spatial(self, letter_pos):
    """Calculate the raw activation of the receiver clone
        node.
    Calculate the raw activation of the receiver clone node
        at a given
    position according to equation 4 in spatial coding.

    Args:
        position: An integer coding for the position where
            the raw
            activation is measured.

    Returns:
        float: The return value. Indicating the raw
            activation of the
            receiver clone node at a given position.
    """
    # Test if the clone is activated. Otherwise the other
        variables could be
    # unspecified. Because the activation method would also
        return 0, this
    # value does not matter.

```

```

        if self.activated is False:
            return 0
        # Calculate the signal.
        power = (letter_pos - self.pos) / self.sigma
        return math.exp(-1 * power ** 2)

# Calculating extended spatial coding scores for word pairs of
  both word lists

# make empty distances matrix for word pairs wordlist 1
w11_SME_dist = np.zeros((len(w11_words), len(w11_words)))

# calculates distances
for i in range(len(w11_words)):
    for j in range(len(w11_words)):
        w11_SME_dist[i, j] = 1 - SpatialModelExtended(w11_words[i],
            w11_words[j]).similarity_score

# make the matrix symmetric by taking the average value
for i in range(len(w11_words)):
    for j in range(i + 1, len(w11_words)):
        dist = (w11_SME_dist[i, j] + w11_SME_dist[j, i]) / 2
        w11_SME_dist[i, j] = dist
        w11_SME_dist[j, i] = dist

# make empty distances matrix for word pairs wordlist 2
w12_SME_dist = np.zeros((len(w12_words), len(w12_words)))

# calculates distances
for i in range(len(w12_words)):
    for j in range(len(w12_words)):
        w12_SME_dist[i, j] = 1 - SpatialModelExtended(w12_words[i],
            w12_words[j]).similarity_score

# make the matrix symmetric by taking the average value
for i in range(len(w12_words)):
    for j in range(i + 1, len(w12_words)):
        dist = (w12_SME_dist[i, j] + w12_SME_dist[j, i]) / 2
        w12_SME_dist[i, j] = dist
        w12_SME_dist[j, i] = dist

# **Calculate Multidimensional Scaling (MDS)**

# In[8]:

# Implementation and parameters of MDS taken from MindSort
  sourcecode (accessed: 30.5.2022)

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import MDS

```

```

# Calculate MDS
def MDS_calculator(distances):
    data_arr = np.array(distances)
    mds_model = MDS(
        n_components=2,
        metric=True,
        n_init=4,
        max_iter=300,
        verbose=0,
        eps=0.001,
        n_jobs=None,
        random_state=0,
        dissimilarity='precomputed'
    )
    transl = np.array(mds_model.fit_transform(data_arr))
    X = transl[:,0]
    Y = transl[:,1]
    return X, Y

# Apply MDS to the distance scores of all word pairs of the five
# algorithms to receive coordinates for each word
wl1_all_dist = ["wl1_lev", "wl1_DL", "wl1_levnorm", "wl1_BM", "
wl1_SME"]
wl2_all_dist = ["wl2_lev", "wl2_DL", "wl2_levnorm", "wl2_BM", "
wl2_SME"]

for i in range(len(wl1_all_dist)):
    dist = vars()["{}_dist".format(wl1_all_dist[i])]
    X, Y = MDS_calculator(dist)
    coord = "{}_coord".format(wl1_all_dist[i])
    vars()[coord] = pd.DataFrame({'ITEM':wl1_words, 'X':X, 'Y':Y
}, index=np.arange(1,16), columns=['ITEM', 'X', 'Y'])

for i in range(len(wl2_all_dist)):
    dist = vars()["{}_dist".format(wl2_all_dist[i])]
    X, Y = MDS_calculator(dist)
    coord = "{}_coord".format(wl2_all_dist[i])
    vars()[coord] = pd.DataFrame({'ITEM':wl2_words, 'X':X, 'Y':Y
}, index=np.arange(1,16), columns=['ITEM', 'X', 'Y'])

# In [9]:

import matplotlib.pyplot as plt

# Visualize algorithm data
algorithm_names = ["Levenshtein", "Damerau-Levenshtein", "
Normalized_Levenshtein", "Weighted_open_bigrams", "Extended_
spatial_coding"]
fig, axs = plt.subplots(len(wl1_all_dist), 2, figsize=(15,30))
for i in range(len(wl1_all_dist)):
    for j in range(2):

```

```

        all_dist = vars()["wl{}_all_dist".format(j+1)]
        coord = vars()["{}_coord".format(all_dist[i])]
        axs[i, j].scatter(coord['X'], coord['Y'])
        for z, label in enumerate(wl1_words):
            axs[i, j].annotate(label, (coord['X'].to_numpy()[z],
                                       coord['Y'].to_numpy()[z]))
        axs[i, j].set_title("MDS_mapping_of_{}_dist._wl_{}".
                             format(algorithm_names[i], j+1))

for ax in axs.flat:
    ax.set(xlabel='x', ylabel='y')

plt.subplots_adjust(wspace=0.2, hspace=0.3)
plt.savefig('MDS_mappings.pdf')
plt.show()

# **Calculate Euclidean distances list**

# In[10]:

import pandas as pd
import numpy as np
from scipy import stats

# Get euclidean distances between wordpairs
def eucl_dist(wl):
    wl = wl.sort_values(by=['ITEM'])
    wl_X = wl['X'].to_numpy()
    wl_Y = wl['Y'].to_numpy()
    dist = pd.DataFrame({'itemPair':[], 'distance':[]})
    for i in range(len(wl_X)):
        for j in range(i+1, len(wl_Y)):
            eucl = np.sqrt(np.square(wl_X[i]-wl_X[j])+np.square(
                wl_Y[i]-wl_Y[j]))
            pair_dist = pd.DataFrame({'itemPair':[wl['ITEM'].
                to_numpy()[i]+'-'+wl['ITEM'].to_numpy()[j]],
                'distance':[eucl]})
            dist = pd.concat([dist, pair_dist], ignore_index=True
                , axis=0)
    return dist

# Calculate euclidean distances experimental data wordlist 1
sw1_exp_eucl = {}
for i in range(len(sw1_exp_coord)):
    sw1_exp_eucl[i] = eucl_dist(sw1_exp_coord[i])

# Calculate euclidean distances experimental data wordlist 2
sw2_exp_eucl = {}
for i in range(len(sw2_exp_coord)):
    sw2_exp_eucl[i] = eucl_dist(sw2_exp_coord[i])

# Calculate euclidean distances per algorithm word list 1

```

```
for i in range(len(wl1_all_dist)):
    dist = vars()["{}_coord".format(wl1_all_dist[i])]
    myStr = "{}_eucl".format(wl1_all_dist[i])
    vars()[myStr] = eucl_dist(dist)

# Calculate euclidean distances per algorithm word list 2
for i in range(len(wl2_all_dist)):
    dist = vars()["{}_coord".format(wl2_all_dist[i])]
    myStr = "{}_eucl".format(wl2_all_dist[i])
    vars()[myStr] = eucl_dist(dist)
```