INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Masterarbeit

ETeC – Ensemble Text Classification without Learning

Jakob Schneider

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Masterarbeit

ETeC – Ensemble Text Classification without Learning

Jakob Schneider

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Markus Wiedemann
Abgabetermin:	18. Januar 2019

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 18. Januar 2019

. . . . . .
(Unterschrift des Kandidaten)

Abstract

Während die Entwicklung von technischen Systemen und Computern in den letzten 50 Jahren massiv vorrangeschritten ist, hat sich die Interaktion zwischen Mensch und Maschine allerdings kaum weiterentwickelt. Obwohl Sprachsteuerungen sich auf Smartphones und vor allem im Smart Home Bereich immer mehr verbreiten, ist es schwierig, ein flexibles System zu finden, das sich für spezielle Anwendungsfälle anpassen lässt.

Diese Arbeit stellt ein solches System vor, um aus kurzen Anfragen, meist Sätzen, die Intention der Anfrage extrahieren zu können. Dafür werden bekannte Algorithmen verwendet, die Ähnlichkeiten zwischen Zeichenketten messen, wie beispielweise die Levenshtein Distanz. Zusätzlich werden Sätze als Zeichenketten interpretiert, bei denen die Zeichen aus Worten bestehen. Ähnlichkeitssuchen werden also auf zwei Ebenen durchgeführt, auf Wort- und auf Satzebene. Um die Genauigkeit der Ähnlichkeitssuchen zu verbessern, werden mehrere der vorher erwähnten Algorithmen zusammengeschlossen. Durch diesen Zusammenschluss von mehreren Algorithmen bleibt es dem Anwender überlassen, die ideale Balance zwischen Genauigkeit und Geschwindigkeit für seinen Anwendungsfall zu finden, indem er beispielsweise die Anzahl der Klassifikatoren in einem Ensemble kontrollieren kann.

Ein weiterer Vorteil des vorgestellten Systems ist die Flexibilität bei der Handhabung der bekannten Wörter und Sätzen. Da es komplett ohne Lernphase auskommt, kann zu jedem Zeitpunkt das Wörterbuch oder die Sammlung von Anfragen erweitert oder verkleinert werden.

Der zentrale Punkt dieser Arbeit ist die Evaluation der verschiedenen Ensemble-Methoden und die Gegenüberstellung mit den einzelnen Algorithmen in den Bereichen Genauigkeit und Geschwindigkeit.

While the evolution of technical systems and computers advanced massively in the last 50 years, the interaction between human and computer stagnates. Although speech control systems are widespread on smartphones and smart homes, finding a flexible systems to fit the assumed use case can be difficult.

This thesis presents such a system, that is designed to extract the intent of short queries. To this end, common algorithms to measure similarities between strings are used, such as the Levenshtein distance. Additionally, queries, or sentences, are interpreted as strings as well, where the individual segments are words instead of characters. Therefore, similarity searches are performed on two levels, on the word- and the sentence level. To augment the accuracy of the classifiers, they are incorporated into ensembles. This combination of algorithms allows the user to find an optimal balance between performance and accuracy for his use case, for example by reducing or increasing the number of classifiers in an ensemble.

An additional advantage is the flexibilty while handling the dictionary size. As there is no learning involved, sentences and words can be added or removed at any time.

The main point of this thesis is the evaluation of the different ensemble methods and the comparison with the individual algorithms in terms of accuracy and performance.

Contents

1 Introduction

Technical systems have evolved massively since the invention of the turing machine, but the interface between human and machine barely made any progress. For example the keyboard layout used today still goes back to Christopher Latham Sholes' proposition in 1878 [22]. Even on the most modern devices like smartphones, the same layout is used to input data. The other input device used on computers, the mouse, hasn't undergone much evolution either since its invention in 1963 by Douglas Engelbart[1].

The challenge to navigate complex and convoluted menus is common in many programs. A more natural way of interacting could be by using natural language. One possibility would be to use chatbots instead of menus to navigate the functionalities of a service. According to Business Insider[2], automated technologies are very attractive to businesses, as they can improve the user experience and cut down the cost for customer service and sales representatives.

The user benefits as well, because automated systems can be scaled to deal with many customers at once, eliminating the need to wait for the next available staff member. But even in more personal spaces, such as smart homes, controlling devices by language is much more comfortable than using remotes or switches.

The difference of entering text via speech or keyboard ignored, all these systems have a step in common, and that is the extraction of the user's intent. The text that is given to the system needs to be classified and information needs to be extracted. This is what ETeC, the **E**nsemble **Te**xt **C**lassifier, is designed to do in a very adjustable manner.

1.1 Motivation

The motivation to build a highly flexible text classifation system originated in the idea of bringing speech control to an application for mobile phones. This would allow for much more functionality that can be easily reached, without overloading the UI.

There are a few tools that already offer speech control for mobile apps, such as the Google Assistant[3], Siri[4] or Alexa[5], but all these cannot offer the flexibility and control to the developer that ETeC was designed to provide. Furthermore, they all have reduced or very limited offline functionalities, a downside that does not impact ETeC at all. Howeer, text-to-speech functionality is not part of ETeC's design, but there are solutions like Google Voice Typing[6].

Challenges need to be met

1. The input texts must be classified correctly, meaning that from a sentence, the intent and possible variables must be identified correctly. For example the classification of

[1] http://www.dougengelbart.org/content/view/162/94/, visited on 06.01.2019
[2] https://www.businessinsider.de/80-of-businesses-want-chatbots-by-2020-2016-12, visited on 28.12.2018
[3] https://assistant.google.com/, visited on 06.01.2019
[4] https://www.apple.com/siri/, visited on 06.01.2019
[5] https://www.amazon.com/b?node=17934671011, visited on 06.01.2019
[6] https://support.google.com/docs/answer/4492226, visited on 06.01.2019

`"Where can I get a hot-dog?"` should have something like this as result: `[type: directions, variable: food/hot-dog]`

2. The system must be flexible, for example, if a request is wrongly classified, this fault must be fixable without the need to restart the system, let alone train a model.

3. The text classification must be fast, as the idea is to alleviate the interaction a user has with a system. If the system is slow, there is no benefit over navigating complex menus.

1.2 Outline

This thesis presents the ETeC system, a flexible text classifier without training or learning phases. It uses collections of common fuzzy string matching algorithms to improve accuracy. Additional to typical string matching algorithms, such as the Levenshtein Distance, it uses various representations of words and sentences in order to utilize algorithms from other areas, such as statistics or biology.

The remainder of the thesis is structured as follows:

- The second chapter gives background information on word representations, string distance metrics and classification methods. Additionally, different apporaches of combining the results from multiple classifiers are presented.

- Chapter three gives an overview of the design of ETeC and it's subsystems, such as internal representations, preparations and the classification process.

- Chapter four presents the testing methods and results. These results contain accuracy and performance statistics for ensembles and individual classifiers. Different approaches to building ensembles are also presented and evaluated.

- The final chapter provides an evaluation and conclusion of the presented ETeC system. It highlights the areas, where future work can improve the functionalities.

2 Fundamentals and Related Work

This chapter describes the fundamentals required to understand and develop the ETeC, an ensemble text classificator for short texts.

First it illuminates word representations, in particular stemming, the soundex system, and cologne phonetics. These systems simplify the process of reducing a word to its original form and are a first step towards finding the most similar word.

The core of this similarity search, or fuzzy string matching, are the string distance metrics, that express the similarity of two words in terms of a number between 0 and 1. As the term distance metric implies, a lower value means a higher similarity.

To assign a class to a text, the most likely class must be determined. To this end, the k-nearest neighbor algorithm and the nearest centroid classifier are explained.

ETeC is an ensemble classificator, which means that there are multiple weak classificators, that form a strong one. Several methods of combining these weak classificators are illuminated, such as averaging, voting or mixture of experts.

2.1 Word Representations

To classify a text, it is important to be able to identify different forms of a word as the same word. These forms are mostly different postfixes (rarely prefixes), but the common stem is usually not the morphological root of the word (for example the forms *argue*, *argues* and *arguing* only share the stem *argu*).

Additional to finding a common sequence of characters, words can be grouped by their sound. Soundex and cologne phonetics transform words into sequences of numbers, which represent different sounds. These sequences might not be equal for different forms of the same word, but they are shorter and therefore faster to compare.

2.1.1 Lemmatisation and Stemming

Lemmatisation is the process of reducing a word to its basic form (for example *arguing* to *argue*). This process relies on dictionary lookups, as there are words that cannot be reduced to their original form in another way (for example *am* $\rightarrow$ *be*). As the goal of ETeC is a system that is as lightweight as possible, lemmatisation is not an option.

Stemming on the other hand is very suitable for ETeC. In contrast to lemmatisation, the goal of stemming is not the morphological root of a word, but the longest common substring of all forms of the same word. This substring might not be a valid word.

Altough there are stemming algorithms that rely on lookup tables, the majority are suffix-stripping algorithms. They consist of a set of rules that determines, whether a suffix, or word ending, belongs to the word form and thus can be stripped of or substituted [15]. Such rules can be: *if the word ends in 'ed', 'ing' or 'ly', remove these suffixes.*

As the proof of concept for ETeC is the mobile app of the Europa-Park Rust in Germany, the stemmer used for this thesis is CISTEM [26]. It is a state-of-the-art german stemmer that

shines in performance as well as in precision. Leonie Weißweiler and Alexander Fraser "performed a comparative analysis of six publicly available German stemmers, where CISTEM achieved the best results for f-measure and state-of-the-art results for runtime"[1].

2.1.2 Soundex

The soundex algorithm was developed by Robert C. Russell and Margaret King Odell in the early 1900s and was used to analyse US censuses [24]. It is used to give every word a code that corresponds to the sound of the word. This code consists of the first letter of the word, followed by 3 digits that are obtained by substituting the following consonants with predefined numbers. Similar sounding consonants are assigned the same number. Fig. 2.1 shows the corresponding numbers and letters.

number	**letter**
1	B, F, P, V
2	C, G, J, K, Q, S, X, Z
3	D, T
4	L
5	M, N
6	R

Figure 2.1: The substitution table according to R. Russel and M. King[24]

The algorithm consists of the following steps:

1. Keep the first letter
2. Replace all following letters by their corresponding numbers, ignoring a, e, i, o, u, y, h, w
3. If a number appears multiple times in a row, without being separated by a vowel, remove the duplicates
4. Remove all remaining characters except the first one
5. If the resulting code has more or less than four characters, remove all surplus characters or add zeros until the code has exactly four characters

Following these rules, the strings `Smith` and `Smythe` are converted to the same code `S530`. `Lee` is reduced to `L000`.

2.1.3 Cologne Phonetics

The cologne phonetics algorithm is very similar to the soundex algorithm, but has its focus on the german language [21]. Additional to the direct conversion from letters to numbers, the context of a letter is taken into account, too. Fig. 2.2 illustrates these contexts and presents the numbers.

The following steps must be taken to convert a string to the corresponding cologne phonetics code:

[1] https://github.com/LeonieWeissweiler/CISTEM, visited on 06.01.2019

letter	context	number
A, E, I, J, O, U, Y		0
H		-
B		1
P	not before H	1
D, T	not before C, S, Z	2
F, V, W		3
P	before H	3
G, K, Q		4
C	in the initial sound before A, H, K, L, O, Q, R, U, X	4
C	before A, H, K, O, Q, U, X except after S, Z	4
X	not after C, K, Q	48
L		5
M, N		6
R		7
S, Z		8
C	after S, Z	8
C	in initial position except before A, H, K, L, O, Q, R, U, X	8
C	not before A, H, K, O, Q, U, X	8
D, T	before C, S, Z	8
X	after C, K, Q	8

Figure 2.2: The substitution table for the cologne phonetics algorithm

1. Replace each letter by the number specified in the lookup table
2. Remove consecutive duplicates
3. Remove all zeros except if it is at the beginning

The cologne phonetics code for the german surnames `Schmitt` and `Schmid` is 862. Note that the codes produced by this algorithm are not truncated, so the name `Müller-Lüdenscheidt` generates the code `65752682`.

2.2 String Distance Metrics

To analyze words and sentences, several string metrics are employed. These allow the comparison of strings beyond equality and give a measure, how similar two strings are.

Some of these metrics, however, are not true metrics in the mathematical sense, as the triangle inequality is not always fulfilled, but for the needed cases of finding the most resembling words or sentences it has no impact. For example, when searching the most similar word to w_a, the only distances that matter are between w_a and all known words w_i. The distance between w_i and w_{i+1} is of no importance.

Normalizing: All these metrics are normalized to the range of [0,1], with 0 meaning equality or high similarity and 1 meaning low similarity. To scale a value from its current range to

any desired range, the following formula is used:

$$value_{scaled} = \frac{(max_{dr} - min_{dr}) \cdot (value - min_{cr})}{max_{cr} - min_{cr}} + min_{dr} \tag{2.1}$$

where

- min_{dr} and max_{dr} are the lower and upper bounds of the desired range
- min_{cr} and max_{cr} are the lower and upper bounds of the current range

As the desired range is [0,1], the formula collapses to:

$$value_{normalized} = \frac{1 \cdot (value - min_{cr})}{max_{cr} - min_{cr}} + 0 = \frac{value - min_{cr}}{max_{cr} - min_{cr}} \tag{2.2}$$

n-Grams: The following section depicts the used string distance algorithms and their functionality is illustrated by an example, using the two strings $\texttt{s}_1$ `= saturday` and $\texttt{s}_2$ `= sunday`. Their lengths are $\texttt{|s}_1\texttt{|}$ `= 8` and $\texttt{|s}_2\texttt{|}$ `= 6`. For all results, see Sec. 2.2.4.

Some of these algorithms use n-grams for the string comparison. These n-grams are the collections of all substrings with the length of n. For example, the n-grams for `saturday` and $n = 2$ would be {`sa, at, tu, ur, rd, da, ay`}. These are also called bigrams.

2.2.1 String-based Algorithms

The first set of algorithms uses the strings directly as input, so order and count of individual characters matters.

Levenshtein Distance

The most common metric to measure string similarity is the Levenshtein Distance [[14]]. It is also refered to as *Edit Distance*. The similarity between two strings is described by the number of insertions, deletions and substitutions of characters, that is necessary to transform one string into the other.

For example the Levenshtein Distance between the strings `saturday` and `sunday` is three, as it takes three operations to transform one into the other:

1. *delete 'a':* `sturday`
2. *delete 't':* `surday`
3. *replace 'r' with 'n':* `sunday`

The minimum distance between two strings is 0 (when both strings are equal), and the maximum distance is the length of the longer word (when every letter of the shorter word must be substituted and the other characters inserted). These two boundaries are used to normalize the Levenshtein Distance to the required range of [0,1].

As the maximum distance in this case is $max(|s1|, |s2|) = |s1| = 8$, the normalized Levenshtein Distance is $\frac{3-0}{8-0} = \frac{3}{8} = \mathbf{0.375}$.

Longest Common Substring

The longest common substring of two strings is the longest sequence of characters that is present in both sources. In the context of ETeC, only the length of this substring is relevant. A longer common substring means higher similarity.

The bounds for the length of the longest common substring are 0 (no matching character) and the length of the shorter word (the shorter word is a substring of the longer one or both are equal).

A greater substring length means a higher similarity, which is opposed to the initial mentioned range of [0,1], so the normalized result must be subtracted from 1.

The longest common substring of the prior example `saturday` and `sunday` is `day` with the length 3. For the example the result would be $1 - \frac{3-0}{6-0} = 1 - \frac{3}{6} = 1 - 0.5 = \mathbf{0.5}$.

The longest common substring is also known as longest common continuous subsequence.

Longest Common Subsequence

Similar to the longest common substring, the longest common subsequence describes a sequence of characters present in both strings, but this sequence can be interrupted by other characters. Again, the longer the sequence, the higher the similarity.

Its bounds are 0 (no matching character) and the length of the shorter string (the longer string holds the complete sequence of the shorter one).

For the two example strings `saturday` and `sunday` the longest common subsequence is `suday`, so the normalized result is $1 - \frac{5-0}{6-0} = 1 - \frac{5}{6} = 1 - 0.833 = \mathbf{0.166}$.

Jaro Similarity

The Jaro Similarity[9] is a string edit distance that was developed by Matthew Jaro for a survey of hard-to-count population groups. As part of that survey, a record-linkage software was needed, which was able to match records with a high degree of accuracy. The presented algorithm uses the linear sum assignment model to match characters in strings.

As there are different components of this algorithm, that can not be depicted with the example strings `saturday` and `sunday`, another example will be used for these components.

The formula for the Jaro Similarity SIM_j is:

$$SIM_j(s_1, s_2) = \begin{cases} 0 & \text{if m} = 0 \\ \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) & \text{otherwise} \end{cases} \tag{2.3}$$

where

m is the number of matching characters
t is $\lfloor$half the number of transpositions$\rfloor$

Matching characters are characters that appear in both strings, but can only be matched once. Additionally, matching characters' indices must be within a certain range to each other. The maximum allowed distance between characters is:

$$d_{max} = \left\lfloor \frac{max(|s_1|, |s_2|)}{2} \right\rfloor - 1 \tag{2.4}$$

The Jaro distance is the sum of three quotients, namely the number of matching characters divided by the length of the first string, the number of matching characters divided by the length of the second string, and the rate of transpositions.

For the strings `greenhouse` and `seabreeze`, the maximum allowed distance between matching characters is $\lfloor \frac{max(10,9)}{2} \rfloor - 1 = 5 - 1 = 4$. The following table illustrates the matching characters of these two strings:

position	1	2	3	4	5	6	7	8	9	10
s_1	g	r	e	e	n	h	o	u	s	e
index of match in s_2	-	5	2	6	-	-	-	-	-	7
s_2	s	e	a	b	r	e	e	z	e	

The last `e` of `seabreeze` has no match in `greenhouse`, as all `e`s there are already matched. There are four matching characters.

The two extracted matching sequences are `reee` for $\mathtt{s_1}$ and `eree` for $\mathtt{s_2}$. The number of transpositions is the number of positions in these two sequences, where the characters do not match, in this case position 1 and 2. Now we have all values for all unknowns:

$$m = 4$$
$$|s_1| = 10$$
$$|s_2| = 9$$
$$t = \lfloor \tfrac{2}{2} \rfloor = 1$$

Inserting these values into the formula results in $\frac{1}{3}(\frac{4}{10} + \frac{4}{9} + \frac{4-1}{4}) = \frac{1}{3}(0.4 + 0.444 + 0.75) = \frac{1}{3}1.594 = 0.531$. Again the outcome must be subtracted from 1 to get the distance between $\mathtt{s_1}$ and $\mathtt{s_2}$, $1 - 0.531 = 0.469$.

Using the same algorithm for the example strings `saturday` and `sunday` yields in the following values:

$$m = 5$$
$$|s_1| = 8$$
$$|s_2| = 6$$
$$t = \lfloor \tfrac{3}{2} \rfloor = 1$$

The Jaro Similarity for `saturday` and `sunday` is $(\frac{1}{3}(\frac{5}{8} + \frac{5}{6} + \frac{5-1}{5})) = (\frac{1}{3}(0.625 + 0.833 + 0.8)) = (\frac{1}{3}2.258) = 0.753$ and the Jaro Distance is $1 - 0.753 = \mathbf{0.247}$.

Jaro-Winkler Distance

William Winkler[27] extended the Jaro Similarity by adding an additional rating boost if the prefix of both strings is equal. The size of the boost depends on the length of the common prefix. Like the Jaro Similarity, the Jaro-Winkler distance originated from the

$$SIM_w(s_1, s_2) = SIM_j(s_1, s_2) + lp(1 - SIM_j(s_1, s_2)) \tag{2.5}$$

need of automatically merging survey lists, ignoring typographical errors in first names or surnames.

The Jaro-Winkler distance is defined as:
where

l is the common prefix length, up to four characters
p is a constant scaling factor

The scaling factor p should not exceed 0.25, as the total similarity could then be greater than 1. Winkler used the value $p = 0.1$.

The Jaro-Winkler Distance then is: $d_w(s_1, s_2) = 1 - SIM_w(s_1, s_2)$

For the example strings `saturday` and `sunday`, the Jaro-Winkler distance is $0.753 + 1 * 0.1 * (1 - 0.753) = 0.753 + 0.1(0.247) = 0.778$ and the Jaro-Winkler Distance is $1 - 0.778 =$ **0.222**

Hamming Distance

Richard Hamming developed the Hamming Distance for error detecting and correction [6]. It is defined as the number of substitutions required to transform one string into another. This definition implies that the two strings must have equal length, but can be extended to any two strings, if the shorter string is filled up with empty characters. In this augmented form, the Hamming Distance between two strings is the number of substitutions plus the difference in their lengths.

The lower and upper bounds are 0 (both strings are equal) and the length of the longer string (all characters of the shorter string must be substituted plus the length difference). The Hamming Distance between `saturday` and `sunday` is 1, as the following table illustrates:

s_1	s	a	t	u	r	d	a	y
s_2	s	u	n	d	a	y	/	/
matches	✓	✗	✗	✗	✗	✗	✗	✗

This distance is again normalized, using its lower and upper bounds, and subtracted from 1: $1 - \frac{1-0}{8-0} = 1 - \frac{1}{8} = 1 - 0.125 =$ **0.875**.

Note that in cases where the prefixes of the strings have different lengths the Hamming Distance is high, as a shifts of parts of the word are not considered, in this example the substring `day`.

2.2.2 Set-based Algorithms

The set-based algorithms construct sets from the characters of the strings. The sets can contain the characters, which means that neither the order, nor the count of the individual characters matter. To reduce this loss of features, the aforementioned n-grams are used to build the sets.

Jaccard Index

The Jaccard Index was originally created to measure the distribution of the flora, but in general it is applicable to any two sets [8]. It is also known as *Intersection over Union*, which describes the operations of this algorithm:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.6}$$

The *Jaccard Distance* is the complement to the Jaccard Index, and it ranges from 0 to 1, so no normalization is needed:

$$d_J(A, B) = 1 - J(A, B) \tag{2.7}$$

In the context of strings, the two sets can be collections the characters of the strings. This implies the following sets of characters:

$\texttt{charset}_{\texttt{saturday}}$:	{a,d,r,s,t,u,y}	size: 7
$\texttt{charset}_{\texttt{sunday}}$:	{a,d,n,s,u,y}	size: 6
intersection:	{a,d,s,u,y}	size: 5
union:	{a,d,n,r,s,t,u,y}	size: 8

The Jaccard Similarity therefore is $1 - \frac{5}{8} = 1 - 0.625 = \mathbf{0.375}$.

For a more strict similarity calculation, bigrams (n-grams where $n = 2$) are used. The two example strings would produce the following sets of bigrams:

$\texttt{bigrams}_{\texttt{saturday}}$:	{at, ay, da, rd, sa, tu, ur}	size: 7
$\texttt{bigrams}_{\texttt{sunday}}$:	{ay, da, nd, su, un}	size: 5
intersection:	{ay, da}	size: 2
union:	{at, ay, da, nd, rd, sa, su, tu, un, ur}	size: 10

Accordingly, the Jaccard Distance between `saturday` and `sunday` using bigrams is $1 - \frac{2}{10} = 1 - 0.2 = \mathbf{0.8}$.

Dice's Coefficient

Similar to the Jaccard Index, Dice's Coefficient was created to measure the association between species of plants[3]. It is a measurement for the number of common elements, divided by the sum of the cardinalities of both sets. The original formula is the following:

$$DC(A, B) = \frac{2|A \cap B|}{|A| + |B|} \tag{2.8}$$

Dice's Coefficient ranges between 0 and 1, but semantically inverse to the needed range, so it will be subtracted from 1.

Identical to the Jaccard Index, the characters and the bigrams of the two strings are used as sets.

The character sets entail the following result: $1 - \frac{2*5}{6+7} = 1 - \frac{10}{13} = 1 - 0.769 = \mathbf{0.231}$.

Using the same bigram sets and intersection from the Jaccard Index, Dice's Coefficient for `saturday` and `sunday` is: $1 - \frac{2*2}{5+7} = 1 - \frac{4}{12} = 1 - 0.333 = \mathbf{0.666}$

Overlap Coefficient

The overlap coefficient, also known as Szymkiewivz-Simpson coefficient, is also related to the Jaccard Index. The intersection is divided by the size of the smaller set:

$$OC(A, B) = \frac{|A \cap B|}{min(|A|, |B|)} \tag{2.9}$$

To get a distance measure, the coefficient must be substracted from 1. All values plugged into the formula, this results in $1 - \frac{5}{6} = 1 - 0.833 = \mathbf{0.167}$ for the character sets and $1 - \frac{2}{5} = 1 - 0.4 = \mathbf{0.6}$ for the bigrams.

2.2.3 Vector-based Algorithms

As the name implies, vector-based distance algorithms need some form of vector representations of the input strings. While there are established systems, such as word2vec[2], that convert words into vectors that hold semantic meaning, ETeC uses a different approach. These systems need large datasets and training to build a model, and should be run on appropriate hardware. ETeC is focused on devices with low end hardware, such as mobile phones. Therefore the cost of the vector generation should be low.

ETeC uses two methods of generating vectors, that are described in the next two sections. After that, the vector-based distance algorithms are presented.

Character Vectors

Character Vectors are a very simple vector representation of a string. The dimensions of the vector are the characters and the values are the quantity of the respective characters. For the sample string `saturday`, the corresponding vector would look like this:

dimension	a	d	r	s	t	u	y
value	2	1	1	1	1	1	1

When comparing two strings, there might be characters in one string, that don't occur in the other one, so the vectors of these strings must be extended by these missing characters. These vectors can be built by generating the union of the sets of characters of each string.

The set of characters for `saturday` is, as in the example above, $char_{s1} = \{a, d, r, s, t, u, y\}$, the set for `sunday` is $char_{s2} = \{a, d, n, s, u, y\}$. The union is $char_{s1} \cup char_{s2} = \{a, d, n, r, s, t, u, y\}$, so the vectors to compare are the following:

[2] https://code.google.com/archive/p/word2vec/

dimension	a	d	n	r	s	t	u	y
$v_{saturday}$	2	1	0	1	1	1	1	1
v_{sunday}	1	1	1	0	1	0	1	1

TF-IDF

TF-IDF, short for *term frequency - inverse document frequency*, tries to provide a measure for how meaningful a word is to a document in a collection of documents, or corpus. Term frequency describes how often a term is used in a document [**?**] and inverse document frequency describes how much meaning a single term conveys [25][11]. This is achieved by examining how often this term is used in other documents of this corpus. For example, the term *the* will appear in many documents of different topics, so the meaning of this word is rather minor.

The term frequency $tf(t, d)$ is calculated by simply counting the occurances of the term t in the document d. To obtain the inverse document frequency, one must first divide the number of documents in the corpus D_{total} by the number of documents containing the term D_t. This quotient is logarithmically scaled. Equation 2.10 illustrates this calculation.

$$idf(t, D) = log\frac{D_{total}}{D_t} \tag{2.10}$$

The tf-idf value for a document d out of the corpus D can be calculated as seen in equation 2.11.

$$tfidf(t, d, D) = tf(t, d) * idf(t, D) \tag{2.11}$$

Euclidian Distance

The Euclidian Distance is a simple distance measurement, specifically the length of a straight line that connects two points in euclidian space. The character vectors of the two strings to compare can be interpreted as two points in an n-dimensional space, where n is the size of the union of the charactersets.

The formula to calculate the euclidian distance is

$$\begin{aligned} d_{euclidian}(v1, v2) &= \sqrt{\sum_{i=1}^{n}(v1_i - v2_i)^2} \\ &= \sqrt{(v1_1 - v2_1)^2 + (v1_2 - v2_2)^2 + ... + (v1_n - v2_n)^2} \end{aligned} \tag{2.12}$$

To normalize the euclidian distance, the lower and upper bounds are needed. The lower bound is 0, when both strings are equal, and as an upper bound, the maximum distance between two strings of the same lengths is used. The maximum distance is reached, are composed of only one character each.

So for the example strings `saturday` and `sunday`, the upper bound is the distance between `aaaaaaaa` and `bbbbbb`. The character vectors of these strings are:

The upper bound is then $d(v_a, v_b) = \sqrt{(8-0)^2 + (0-6)^2} = \sqrt{64+36} = \sqrt{100} = 10$.

Therefore the euclidian distance for `saturday` and `sunday` (using the character vectors from 2.2.3) is:

dimension	a	b
v_a	8	0
v_b	0	6

$d = \sqrt{(2-1)^2 + (1-1)^2 + (0-1)^2 + (1-0)^2 + (1-1)^2 + (1-0)^2 + (1-1)^2 + (1-1)^2} = \sqrt{1+0+1+1+0+1+0+0} = \sqrt{4} = 2$. Normalizing using the calculated lower and upper bounds results in $\frac{2-0}{10-0} = \frac{2}{10} = \mathbf{0.2}$.

Taxicab Metric

The Taxicab Metric, also known as *Manhattan Distance* or *L_1 Distance*, has its name from the grid layout of streets (most prominent in Manhattan). Altough the roots of this metric date back to Hermann Minkowski in the 19th century, the term taxicab geometry was coined by Eugene Krause [12].

The taxicab distance between two points is not the shortest straight line, but the sum of line segments aligned to a grid, that need to be traversed to get from the first to the second point. This grid is defined by the axes of the coordinate system. So the taxicab distance can be expressed as the sum of distances for each dimension:

$$\begin{aligned} d_{taxicab}(v1, v2) &= \sum_{i=1}^{n} |v1_i - v2_i| \\ &= |v1_1 - v2_1| + |v1_2 - v2_2| + ... + |v1_n - v2_n| \end{aligned} \tag{2.13}$$

Again, two strings are most dissimilar when both are composed of only one character. So to normalize the taxicab distance between `saturday` and `sunday`, `aaaaaaaa` and `bbbbbb` with the aforementioned character vectors are used to calculate the upper bound: $d_{max} = |8-0| + |0-6| = 14$.

The not normalized taxicab distance between `saturday` and `sunday` is $d_{taxicab}(v1, v2) = |2-1|+|1-1|+|0-1|+|1-0|+|1-1|+|1-0|+|1-1|+|1-1| = 1+0+1+1+0+1+0+0 = 4$, which is normalized to $\frac{4-0}{14-0} = \frac{4}{14} = \mathbf{0.286}$.

Simple Matching Coefficient

The simple matching coefficient is originally designed for objects that have boolean values in their attributes. It can be adapted for string matching purposes by neglecting the count of characters in a word and simply storing whether this character is present, or by comparing the count for each character. The second option results in a stricter measurement, so within the context of this work this algorithm will compare strings by the frequency of its characters.

The simple matching coefficient is defined as the number of matching attributes divided by the number of total attributes:

$$smc(v1, v2) = \frac{\text{number of matching attributes}}{\text{number of total attributes}} \tag{2.14}$$

The number of total attributes is the number of dimensions of the character vectors.

For the example strings `saturday` and `sunday`, the character vectors from section 2.2.3 can be used. The matching attributes are `d`, `s`, `u` and `v`, so the number of matching attributes is 4.

The number of total attributes is 8, so the simple matching coefficient is $smc(v1, v2) = \frac{4}{8} = \mathbf{0.5}$.

Cosine Similarity

The cosine similarity between two vectors is defined by the direction of the vectors that are compared. More specific, the cosine similarity is the cosine of the angle between the vectors, so if the angle is 0°, the similarity is 1. For vectors that are perpendicular to each other (90° angle), the cosine is 0 and for vectors that point in exactly different directions (180° angle), the cosine similarity is -1.

As the dimensions of the vectors describe the frequentness of each character, the values can only be greater or equal to 0, therefore there are no two vectors that point in different directions and the cosine similarity is limited to values between 0 and 1, but it has to be subtracted from 1.

The euclidian dot product formula ($v1 \cdot v2 = ||v1||\ ||v2||\ cos\theta$) can be reordered to get the cosine of the angle between two vectors. The resulting formula is the dot product of the vectors divided by the product of the vectors' lengths.

$$cos\theta = \frac{v1 \cdot v2}{||v1||\ ||v2||} \tag{2.15}$$

$||v1||$ denotes the length of $v1$. The dot product can be calculated by adding the products of the vectors' components:

$$v1 \cdot v2 = \sum_{i=1}^{n} v1_i v2_i \tag{2.16}$$

The cosine similarity for the two example strings therefore is

$$\begin{aligned} cs(v1, v2) &= 1 - \frac{2*1+1*1+0*1+1*0+1*1+1*0+1*1+1*1}{\sqrt{2^2+1^2+0^2+1^2+1^2+1^2+1^2+1^2}\sqrt{1^2+1^2+1^2+0^2+1^2+0^2+1^2+1^2}} \\ &= 1 - \frac{2+1+0+0+1+0+1+1}{\sqrt{4+1+0+1+1+1+1+1}\sqrt{1+1+1+0+1+0+1+1}} \\ &= 1 - \frac{6}{\sqrt{10}\sqrt{6}} = 1 - \frac{6}{3.162 \cdot 2.449} = 1 - \frac{6}{7.746} = 1 - 0.775 \\ &= \mathbf{0.225} \end{aligned} \tag{2.17}$$

2.2.4 Example Result Overview

The following table contains all values for the distance between `saturday` and `sunday`, measured by the above presented distance metrics. The input for the string-based algorithms are the words themselves (`saturday` and `sunday`), their stems (`saturdai` and `sundai`), their soundex codes (`S363` and `S530`) and their cologne phontics codes (`8272` and `862`). For the set-based and vector-based algorithms, $1 \leq n \leq 3$ is used for the n-grams. Values are rounded to three decimals.

Distance Metric				
String-based Algorithms	**Word**	**Stem**	**Soundex**	**Cologne**
Levenshtein Distance	0.375	0.375	0.75	0.5
Longest Common Substring	0.5	0.5	0.75	0.667
Longest Common Subsequence	0.167	0.167	0.5	0.333
Jaro Similarity	0.247	0.247	0.333	0.278
Jaro-Winkler Distance	0.223	0.223	0.3	0.25
Hamming Distance	0.875	0.875	0.75	0.75
Set-based Algorithms	**n=1**	**n=2**	**n=3**	
Jaccard Index	0.375	0.8	0.889	
Dice's Coefficient	0.231	0.667	0.8	
Overlap Coefficient	0.167	0.6	0.75	
Euclidian Distance	0.2	0.329	0.392	
Vector-based Algorithms	**n=1**	**n=2**	**n=3**	
Taxicab Metric	0.286	0.667	0.8	
Simple Matching Coefficient	0.5	0.8	0.889	
Cosine Similarity	0.225	0.662	0.796	

2.3 Classification

Classification is the process of categorizing new observations into predefined classes. In the case of this thesis, the observations are queries and the classes are predefined categories of queries.

It is the main feature of ETeC, as all input queries must be assigned one of the known classes of queries, or it must be determined that the query does not fit any of the known classes.

Two classification algorithms are employed in ETeC, the *weighted k-nearest neighbor* and the *nearest centroid classifier*. These are outlined in this section.

2.3.1 Weighted k-Nearest Neighbor Classifier

The k-nearest neighbor algorithm calculates the distance to all other (classified) observations and considers only the closest k neighbors. The observation is assigned to the class that the majority of these k neighbors belongs to [2].

The selection of the parameter k plays an important part in the accuracy of the classification. A bigger k can reduce the effect of noise in the data, but may include observations with large distances to the query. Using a small k can lead to results distorted by noise. For k=1 the result is a voronoi diagram.

Fig. 2.3 shows the effect of selecting larger or smaller values for k. The green dot is the new observation, the continuous line shows the relevant neighbors for $k = 3$, two red triangles and one blue box. In this case, the green dot would be assigned to the class of the red triangles. The relevant neighbors for $k = 5$ are shown by the dotted line, three blue boxes and still only two red triangles, which means that the green dot would be assigned to the blue box class in this case.

Additionally, the neighbors can be weighted so that closer neighbors play a larger role in the decision than neighbors that are further away [7]. This can significantly improve

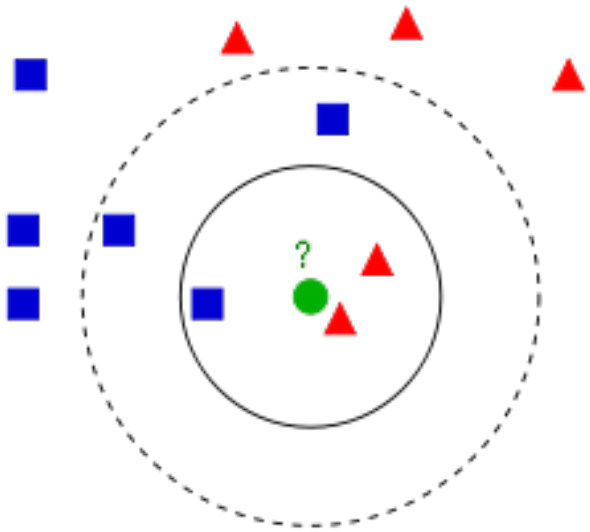


Image Source: https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

Figure 2.3: Illustration of the k-nearest neighbor Algorithm with k=3 (continuous line) and k=5 (dotted line)

precision, especially when k is large, as it reduces the effect of observations that are far away. A common and simple weighting function in dependence of the distance d is $w = \frac{1}{d}$.

2.3.2 Nearest Centroid Classifier

To classify an observation using the nearest centroid classifier, the mean distance from the observation to each class must be calculated. The class with the nearest mean is the class that is assigned to the observation.

Using this algorithm with the TF-IDF vectors is also known as Roccio Classifier[10], due to its similarity with the Roccio Algorithm[23].

2.3.3 Nearest Sample Classifier

The nearest sample classifier is a special case of both the nearest centroid and the nearest neighbor classifier, if every class only consists of one sample. In the case of classes with several samples, the class with the smallest distance from any of its samples to the new observation is chosen.

2.4 Ensemble Methods

Ensemble methods enjoy great popularity in machine learning contexts [20][4], as they can overcome several problematic fields, such as learners getting stuck in local optima or wrong hypothesis caused by the choice of the training data set. For pure classification without learning there is less information, but the principles should apply as well, as the presented string distance metrics could as well be trained classifiers.

Condorcet's jury theorem serves as basis for ensemble methods [1]. Based in political science, the Marquis de Condorcet argued, that, given a choice between two options, a

jury of independent voters is more likely to pick the correct choice, the larger the jury is. Furthermore, each voter must have a probability higher than 50% of choosing the right decision[13].

Within the scope of this thesis, the voters are the distance metrics. There are several methods of combining the individual votes, that are presented in the following sections.

2.4.1 Averaging

Averaging is the most common method for ensembles that handle numeric values. As the name implies, the ensemble's result is the average of the individual classifiers. There are two different kinds of averaging, simple and weighted averaging.

The formula for simple averaging is shown in equation 2.18. The individual distances d_i of each string distance metric are added up and divided by the number of metrics n.

$$D(v1, v2) = \frac{1}{n} \sum_{i=1}^{n} d_n(v1, v2) \tag{2.18}$$

Weighted averaging introduces the idea of different importances of the individual distance metrics. These weights can be predefined or dependent on the input and its attributes, within the scope of this thesis these attributes are for example number of words of the query or lengths of the words. The formula for weighted averaging is shown in equation 2.19. Each distance d_i is multiplied by a weighting factor w_i specific for this metric.

$$D_w(v1, v2) = \frac{1}{n} \sum_{i=1}^{n} w_i d_n(v1, v2) \tag{2.19}$$

Simple averaging can be seen as a special form of weighted averaging with all weights $w_i = 1$, but weighted averaging is not superior in most cases [28]. A lot of weights have to be learned which can quickly lead to overfitting, and is contrary to the no-learning design of ETeC.

2.4.2 Voting

Analogous to averaging, voting is the go to method for nominal outputs. Each classifier determines the class it should belong to, and then a vote is cast to determine the final result. Several methods are common for voting procedures:

- **Majority Voting:** If any class gets more than 50% of the votes, this class is assigned to the new observation, otherwise the observation gets rejected.
- **Plurality Voting:** The class with the most votes is assigned to the observation. Rejection is not possible, as there is always at least one class with the largest number of votes. Ties are possible, and must be broken arbitrarily.
- **Weighted Voting:** Similar to plurality voting, but each classifier gets a weight according to its classification precision.

3 Conceptual Design

This section provides insight into the conceptual design of ETeC. The goal is to build an ensemble classifier that is able to identify short text requests. The first use case is planned to be a speech control tool for smartphones. But in contrast to its existing competitors like Alexa[1], Siri[2] or the Google Assistant[3], ETeC is designed to work offline and without the need for machine learning.

This chapter provides information about the requirements for ETeC, the basic structure and the scientific issues that are adressed in this thesis.

3.1 Requirements

ETeC must meet a set of requirements that set it apart from other text classifiers. These requirements are as follows:

- **Complete offline functionality:** All calculations must take place on the device where the query is created, no communication to other devices needed.
- **Easily updateable:** Adding, removing or changing stored queries must be possible without much effort
- **Low demands to CPU and Memory:** As the majority of devices ETeC runs on are smartphones, memory and cpu requirements must be low.
- **Fast:** Queries should be evaluated quickly.

3.2 Internal Representations

There are four different types that are used in ETeC:

- **Words** are the basic building blocks.
- **Queries** are short texts that are either in the list of known observations or are to be classified. Essentially, queries are sentences with a specific intent, for example, "Where can I get a Hamburger" would be a query with the intent of directions to a hamburger restaurant.
- **Variables** are optional parts of queries. They can be exchanged without altering the class of a query. An example would be the query "Where can I get a *Hamburger*", where *Hamburger* is the variable (cf. "Where can I get a *Hot Dog*"). Variables have

[1] https://www.amazon.com/alexa/
[2] https://www.apple.com/ios/siri/
[3] https://assistant.google.com/

names and a list of manifestations. In this case the variable's name could be `Fast Food` while the manifestations could be *[Hamburger, Cheeseburger, Hot Dog, ...]*.

- **Query Classes** are collections of queries that have the same intent. For example, the queries "Where can I get a [Fast Food]?" and "Where is the nearest [Fast Food] joint." both belong to the class "local search". Classes store information on which variables are allowed, therefore every query belonging to this class has the same constraints regarding variables. The class "local search" could allow variables such as [fast food] and [drinks].

To reduce the workload for the classification algorithm, all the different forms a word can be in are created at the startup.

3.2.1 Words

Words are stored as purely lower case strings. Additionally, the soundex and cologne phonetics codes and the stem are saved as strings as well.

The n-grams are stored in two different versions: as sets and as vectors. All n-grams for $1 \leq n \leq 3$ are stored. Finally, the TF-IDF vector is calculated and stored as well.

For example, the complete internal representation of Mississippi is

id	42						
word	mississippi						
soundex	m221						
cologne phonetics	6881						
stem	mississippi						
1-gram set	i	m	p	s			
1-gram vector	4	1	2	4			
2-gram set	ip	is	mi	pi	pp	si	ss
2-gram vector	1	2	1	1	1	2	2
3-gram set	ipp	iss	mis	ppi	sip	sis	ssi
3-gram vector	1	2	1	1	1	1	2
TF-IDF	0.2	0.81	... (random values for demonstration purposes)				

In the table the TF-IDF vector values are random values and are just for demonstration purposes. It is also notable that the *corpus* in the scope of this thesis is the collection of all queries, and a *document* is the collection of all queries of a single class. The collection of all words is referred to as *dictionary*.

3.2.2 Queries and Classes

Queries are saved as text and as a sequence of word IDs. Additionaly, their type is stored as well. Classes are represented by their name and a list of allowed variables.

A sample dictionary, query and query class is demonstrated in the following tables:

Dictionary	
word	**id**
a	1
can	2
get	3
how	4
i	5
we	6
when	7
where	8

Query	
text	”where can i get a [fast food]”
ID sequence	8, 2, 5, 3, 1
type	local search

Class	
name	local search
constraints	[fast food], [drinks], ...

3.3 Preparations vs Runtime

To keep the query-catalogue updateable, ETeC must be able to accept a list of queries in a human readable form. Once loaded and processed, the catalogue can be stored in binary to reduce load times. Unless there are updates to the catalogue, the binary version can be used.

When the human readable version is loaded, some preparations must be done:

- Store the words used in the queries. For each word:
  - transform it to lower case
  - replace special characters (ä to ae, for example)
  - remove all remaining non alphanumeric characters
  - create the soundex code
  - create the cologne phonetics code
  - extract the stem
  - extract n-grams for $1 \leq n \leq 3$ as sets and vectors
  - assign an ID
- Store the queries with word IDs and class
- Calculate TF-IDF vectors for each word
- Store the type of variables for each class
- Store constant variables

These operations are done to keep the computational cost at runtime as low as possible. Storing all the different forms for each word means a higher memory load, but decreases runtime significantly.

Once everything is stored, calculated and extracted, it can be saved in a binary file, which speeds up the loading time considerably. All of these internal representations must be created only, when updates to the query-catalogues are issued.

3.4 Classification Process

The first step of the classification is the word-level analysis. The goal is to identify the words from the input and generate a sequence of their ids, so that the sentence-level analysis can compare these instead of strings.

3.4.1 Word-Level Analysis

The word-level analysis is divided into two phases, the preselection and the actual comparison. Every word of the input text must be matched against all words in the dictionary. This process is even more costly, as the matching process consists of many or all of the string distance calculations presented earlier.

To improve performance, a preselection can be made. Words are ruled out, that don't fit certain criteria before the comparison even begins.

Preselection

The preselection process is a very simple character comparison. If the first n characters of the input word don't match the first n characters of the word in the dictionary, the string distance calculations will not be performed and the next dictionary word is tested.

The subject of this comparison can vary. Either the word, the stem, the soundex or cologne phonetics code can be used.

It is very important to determine the right value for n, as if it is too large, the right word might be pruned, and if it is too low, the search will slow down. n must be dependent on the word size.

Section 4.3.1 will provide a list of parameters, that work best in the scope of the conducted tests, and insight into the testing method.

Additionally to filtering out unfitting words before the comparison, a threshold can be used to stop comparison, once a word is found, that is similar enough to the input word, i.e. the distance between the words is below the given threshold. Section 4.3.2 will provide the testing methods and optimal thresholds.

Comparison

The actual comparison will take place for all words that are permitted by the preselection algorithm. For each chosen distance metric, the distance between every input word and every word from the dictionary is calculated. The decision on the most similar word can be reached either by choosing the word with the lowest average distance or by selecting the word with the most votes (see section 2.4).

To find the correct ensemble size, composition and method, several tests were conducted. Section 4.4.5 shows the correlation tests and provides a table with the results, Section 4.2 features the performance and accuracy of the string distance metrics, and Section 4.4 exhibits the effectiveness of the ensemble methods.

3.4.2 Sentence-Level Analysis

The sentence-level analysis is very similar to the word-level analysis, as both entities are just sequences; words being sequences of characters and sentences being sequences of words. Once

the input queries are broken down into sequences of word-ids, the comparison of sentences mirrors the comparison of words.

The main difference between sentence-level- and word-level analysis is that on the sentence level, multiple sequences with major differences can belong to the same class, for example the sentences *"Where is the nearest supermarket"* and *"How can I get to the nearest supermarket"* basically have the same meaning, but are vastly different. The classes could be introduced to the word-level in the form of groups of synonyms, for example the verbs *start*, *begin* and *commence*, but in the context of this thesis, this additional overhead would lack an associated benefit.

3.4.3 Named Entities

Named entity is a term that originated in the field of message understanding and information extraction [5]. In this context, a named entity is a sequence of words that depicts a real-world entity, like a person, an object or a brand name. In the context of this thesis, named entities are the variables of the queries.

At the time of this thesis, the named entity extraction is motivated by the assumption that the variable manifestations are part of the dictionary with the same id. For example, *burger*, *pizza* and *sandwich* all have the same id, as they are part of the variable group *fast food*.

Many other techniques for named entity recognition exist[19] [17], but are not part of this thesis.

3.5 Research Questions

This thesis aims to provide answers to the following questions:

- How accurate and how fast can a text classifier be without learning?
- What is the ideal ensemble method for this use case?
- What are the ideal weights and thresholds for the string distance metrics?
- What is the optimal method and the corresponding parameters for preselection?

4 Implementation and Evaluation

This section will explain the tests that were conducted. These tests can be split into three different categories:

- **Correlation Test:** This test focuses on the correlation of the distance metrics. As the condorcet-jury-theorem states, the accuracy of the ensemble improves with more independent jury members. This test should provide insight into the process of building ensembles that outperform the individual metrics.

- **Performance-based tests:** As one requirement of ETeC is to run on smaller devices, such as smartphones, performance may be critical. These tests aim to clarify how large the ensembles may be and still provide acceptable performance. Another performance factor may be the size of the dictionary.

- **Accuracy-based tests:** The accuracy of the individual distance metrics, as well as the ensembles, plays the largest role. These tests state how well each ensemble or metric perform.

These tests are performed on two different tiers: The word-level-analysis and the sentence-level-analysis.

The goal of the tests of the word-level is to find the best method to determine for a given word the closest word in a dictionary. The definition of the best method can vary for each application area, for example in a mobile environment performance might be more important than accuracy.

The sentence-level analysis focuses on finding the correct corresponding query class. The similarity between the sentence- and the word-level analysis becomes apparent when taking into account the representation of words and sentences: both are sequences. While words are made up of characters, sentences are composed of words. Thus the algorithms can be applied to both with minor adjustments. These adjustments are explained in section 4.5

With the exception of the correlation test, each test is executed with a number of different testset sizes. These testsets are either random collections of words or classified sentences.

In this section, the distance metrics, word representations and precompare methods will be referred to by abbreviations, depicted in the following table:

Term	Abbreviation
Distance Metrics	
Levenshtein Distance	lev
Longest Common Subsequence	lss
Longest Common Substring	lcs
Jaro Similarity	jar
Jaro-Winkler Distance	jwd
Hamming Distance	ham
Jaccard Index	jac
Dice's Coefficent	dic
Overlap Coefficient	ovr
Euclidian Distance	euc
Taxicab Metric (Manhattan Distance)	man
Simple Matching Coefficient	smd
Cosine Similarity	cos
Word Representations	
Word	wrd
Soundex	snd
Cologne Phonetics	clg
Stem	stm
1-gram	1gr
2-gram	2gr
3-gram	3gr
Precompare Methods	
Word-based	wopc
Soundex-based	sopc
Cologne Phonetics-based	copc
Stem-based	stpc
No Precompare	nopc

4.1 Testset Generation and Sources

There are two types of testsets, words with their different forms and a classified collection of sentences.

The set of words is extracted from Daniel Naber's Language Tool, using the instructions on his website [1]. It contains a list of 362.671 forms of 50.307 different german words. At the start of the tests, all forms are read and stored as strings (see Appendix 3). A set of indices with the size of the word-set is created and shuffled using the Mersenne Twister [16]. The first n indices of the shuffled set are utilized to build the final testset, where n is the number of words defined by the user. The testset contains the words in their different forms (`wrd`, `snd`, `clg` and `stm`).

There are two sets that are being generated, the `words` set and the `stems` set. For the `stems` set, the `wrd` representation is replaced by the `stm` of the word, thus creating a potentially smaller set (The `stems` set is smaller, when two words are present in the `words` set, that share

[1] http://www.danielnaber.de/morphologie/, visited on 08.12.2018

the same stem). The following table shows the average size of the `stem` sets for different `words` set sizes. For each size, there were four random testsets generated.

`Words` set size	Average `stems` set size	Percentage
10	10	100%
50	50	100%
100	99,75	99,75%
250	248,5	99,4%
500	494,25	98,85%
1000	977,25	97,73%
2000	1907	95,35%
4000	3660	91,5%
8000	6675,25	83,44%
16000	11545	72,16%
32000	18164,5	56,76%
100000	31912	31,91%
200000	41726	20,86%
362671	50307	13,87%

For the tests, the `words` set is used as source, and the `stems` set as target, i.e. for every word from the `words` set, the goal is to find the corresponding stem in the `stems` set. For fast validation, the corresponding ids are stored in a map.

The collection of classified sentences was created in a collaboration with the Europa Park Rust[2]. In a survey amongst the visitors of the amusement park, a list of five topics with up to nine different questions per topic was collected. These questions were classified by an employee of the park and the writer of this thesis. Additionally, the FAQ section of the park's hotel page[3] was incorporated to the collection of questions, raising the total number of questions to 35. Along with these questions, 78 variables were introduced, consisting mostly of points of interest (PoIs) in the park. This testset is interesting because almost every question uses variables, and the variables may be easily confused (for example there are the PoIs `eurosat`, `euromir` and `eurotower`), but there are only few questions and classes.

Therefore another testset is used, the SMS Spam Collection Data Set from the UCI Machine Learning Repository[4]. This dataset contains 5574 sms messages that are classified into two categories, spam and ham.

4.2 Distance Metrics

Before ensembles are generated and tested, it is important to see how the individual distance metrics perform. Each distance metric was tested by providing a source testset, from which every word needed to be matched to its corresponding stem in the target testset. This was done by simply iterating over the target testset, calculating the distance to every candidate and picking the one with the lowest distance (see Appendix 4). The overall accuracy of an algorithm is the number of correct classifactions divided by the size of the source testset.

[2]https://www.europapark.de/, visited on 08.12.2018
[3]https://www.europapark.de/de/uebernachten/info-service/faq-europa-park-hotels, visited on 03.01.2019
[4]https://archive.ics.uci.edu/ml/datasets/sms+spam+collection, visited on 03.01.2019

For larger testsets this is a very time-consuming process, therefore the employed testsets contained 10, 50, 100, 250, 500, 1000, 2000, 4000, 8000 and 16000 words. According to Liu Na [18], 95% of common texts are made up by about 3000 words, so the testsets are big enough to reproduce different use cases. For each testset size, four random sets were generated and the results of these four tests were averaged.

In addition to the accuracy, the performance of each algorithm was recorded, i.e. the total time it took to classify all words. This time was also divided by the number of comparisons, which is the number of words in the source testset multiplied by the size of the target testset. Appendix 6 shows the class that was used to measure the time.

The process of finding the most similar word in the dictionary can be seen as the special case of both the nearest neighbour and nearest centroid classifier presented in Section 2.3.3, where the classes only consist of one observation, or phrased the other way around, each word in the dictionary represents its own class. In this case with one-piece classes, both classifiers behave the same.

4.2.1 Accuracy Results

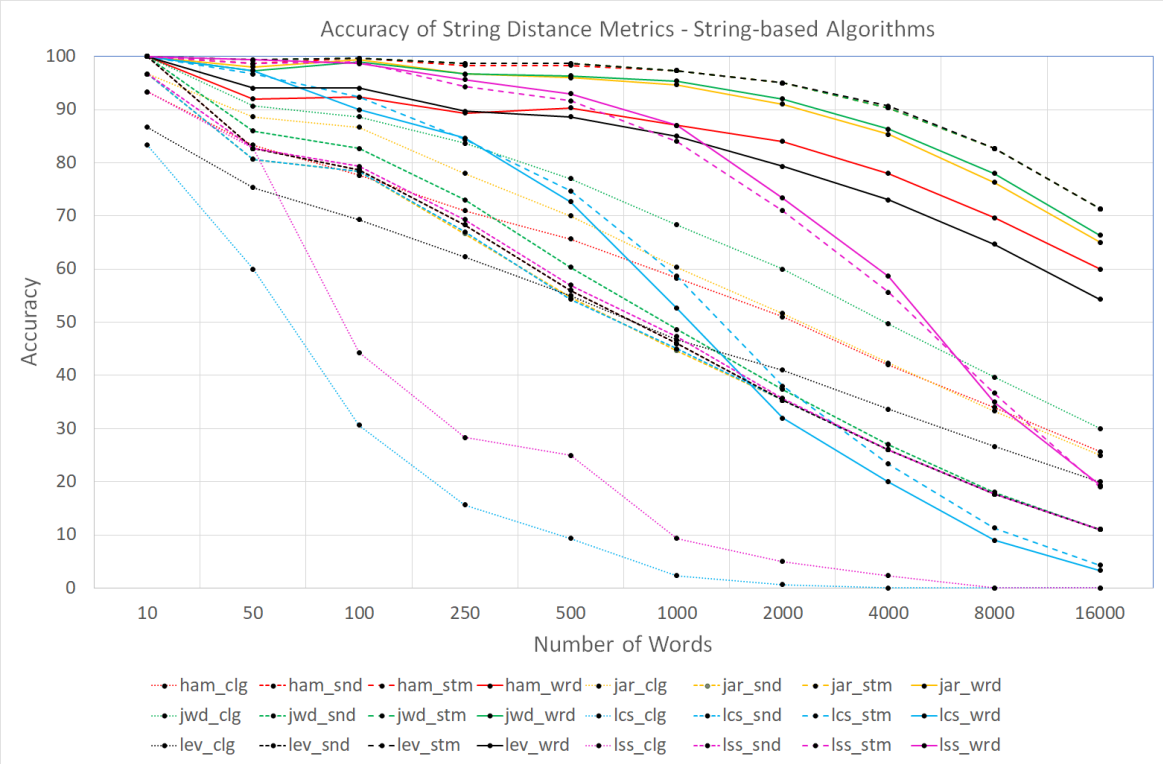


Figure 4.1: Results of the individual accuracy tests for each string-based distance metric. Metrics with the same algorithm but different input types (such as `stm` or `clg`) share the same color.

Fig. 4.1 shows the test results for the string distance metrics, a table with the results can be found in Appendix 9. What can be seen is that up until 1000 words, several metrics still have accuracies higher than 90%, with `ham_stm`, `jwd_stm` and `lev_stm` with the highest

accuracy of 97,3%. At 4000 words, these three still lead, but their accuracy is reduced to 90,333%, dropping to 71.333% at 16000 words. At 500 words, the last metric (`ham_wrd`) has a local maximum, but with increasing testset size the accuracy of every algorithm declines.

The different manifestations of the `lss` algorithm also show that the choice of type plays an important role. While `lss_wrd` and `lss_stm` are among the top classificators, `lss_clg` is the second worst and `lss_snd` can be found in the lower third of all string-based distance metrics. This does not come as a surprise, given that with increased word count, it is more likely to have words that sound similar.

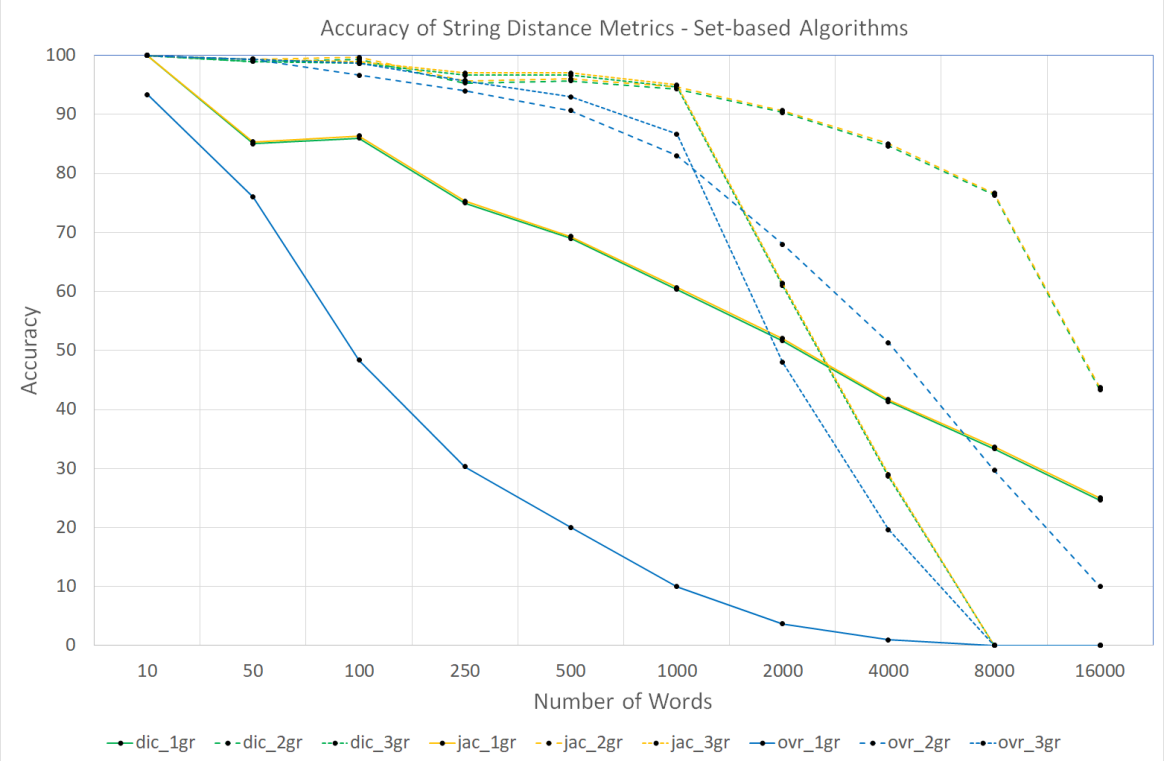


Figure 4.2: Test results of set-based string distance metrics.

The set-based algorithms draw a similar picture (see Fig. 4.2). Interestingly, the `2gr` versions of the algorithms show the best results, while the `1gr` and `3gr` versions compete for the least accurate method. Again, up until 1000 words, results of up to 95% are reached, but at 16000 words the highest accuracy is reached by `jac_2gr` with only 43,67%.

In line with expectations, the results of the vector-based algorithms are very similar to the results of the set-based methods (see Fig. 4.3). Only the absence of an approach with an accuracy as low as `ovr_1gr` sets the two charts apart.

Appendix 7 provides a summary of all individual string distance metrics, colored by type. Here can be seen, that every algorithm suffers the same fate of declining accuracy with an increasing number of words.

4.2.2 Performance Results

The performance of the string distance metrics is highly dependent on word size and for the set and vector based approaches on the number of unique characters in a word. Fur-

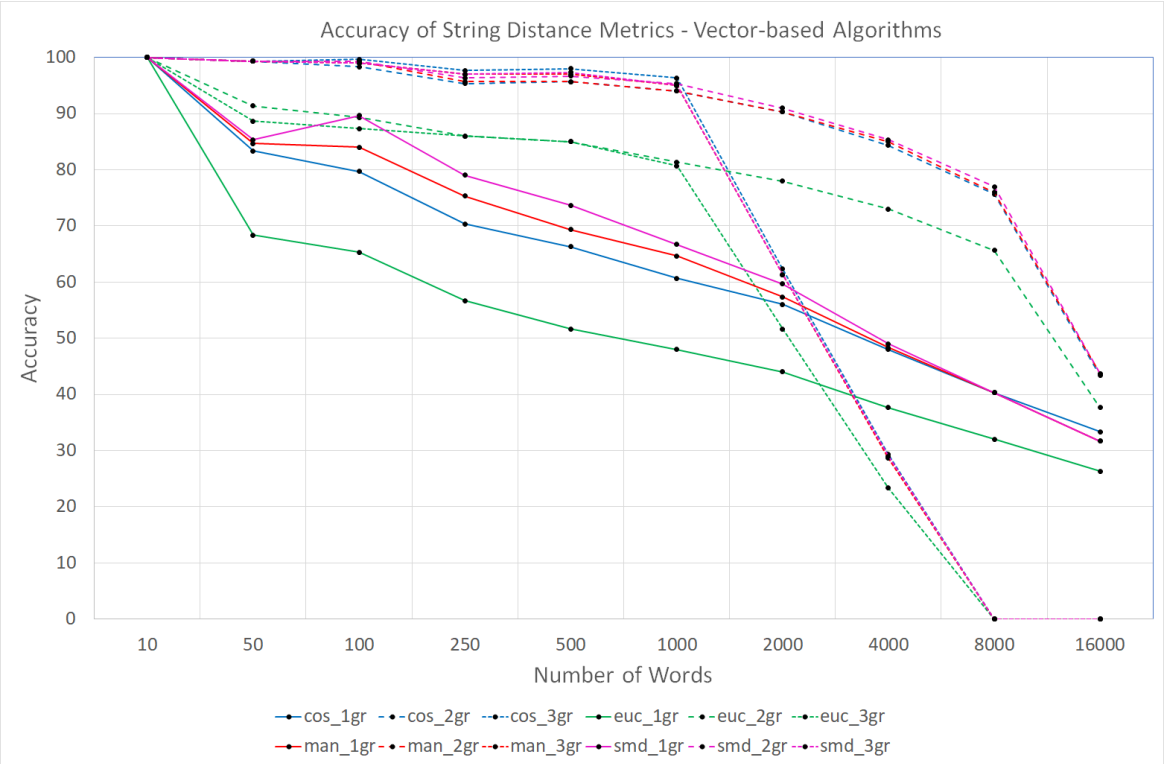


Figure 4.3: Test results of vector-based string distance metrics.

thermore, the implementation of the set- and vector-based algorithms leaves much room for optimization, as vectors must be augmented for every comparison. This decision was made with memory requirements in mind, as otherwise each word would need to reserve space for each character of the alphabet, even when this character is not present in the word (see Section 2.2.3). An implementation of sparse vectors would solve this problem, but the aim of the thesis is not the fastest implementation, but improving the performance by other means.

To smooth out the effect of different word sizes, the presented results are from a testrun with 16000 words. The tests were run without any multithreading optimizations on an Intel Core i7 4790k with 16GB RAM. The testing program was built using Visual Studio 2017 and its default compiler flags for the *Release* configuration on the *x64* Platform:

```
/GS /GL /W3 /Gy /Zc:wchar_t /Zi /Gm- /O2 /Fd"x64\Release\vc141.pdb"
/Zc:inline /fp:precise /D "NDEBUG" /D "_CONSOLE" /D "_UNICODE" /D "UNICODE"
/errorReport:prompt /WX- /Zc:forScope /Gd /Oi /MD /FC /Fa"x64\Release\"
/EHsc /nologo /Fo"x64\Release\" /Fp"x64\Release\ETEC-Tests.pch"
/diagnostics:classic
```

Figure 4.4 shows the average time per comparison for every distance metrics. The jumps from `lcs_wrd` to `ovr_1gr` and from `ovr_2gr` to `man_1gr` can be explained by the aforementioned implementation details.

Interestingly, the set- and vector-based algorithms show clearly, that 1-grams are fastest to process, while 2-grams mostly come in last place. The reason for the faster 3-gram processing seems to lie in quantity of n-grams that needs to be processed. While one could think that a lower n means more n-grams, it is important to notice that multiple occurences of the same

n-gram are only counted once for the set-based algorithms. For the vector-based algorithms it does not make a difference either, as recurring n-grams only increase the associated value in the vector, not the count of n-grams. As clarification, the table from Section 3.2.1 shows that the word `Mississippi` has only four 1-grams and seven 2- and 3-grams.

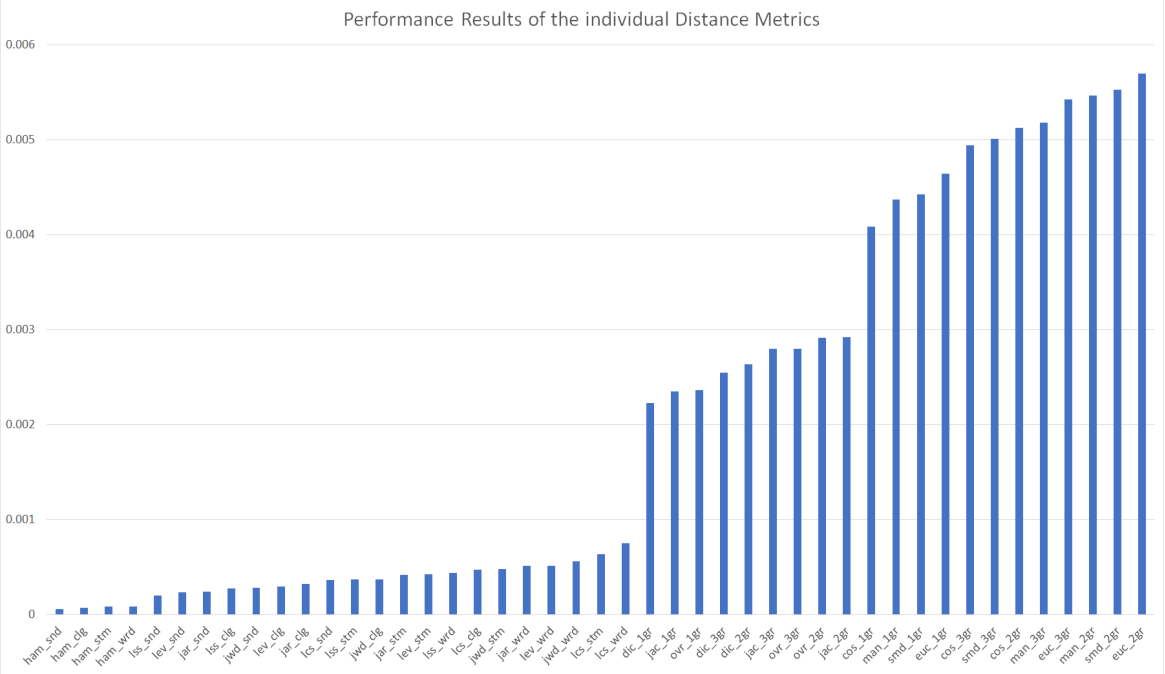


Figure 4.4: Performance comparison of the algorithms. The y-axis depicts the average time per comparison in ms.

As the `ham`-algorithms simply perform comparisons for each position in the strings, it comes as no surprise that the different versions of this algorithm take first place.

4.3 Preselection and Thresholds

The performance results from Section 4.2.2 show that finding the most similiar word in a small to medium dictionary can be done quite fast, but for very large dictionaries long computation times are predetermined. For example, the Second Edition of the *Oxford English Dictionary* contains 218.632 words (of which only 141.476 words are currently in use)[5]. Using this dictionary, it would take 12,68ms to find the most similar word using the fastest algorithm, (`ham_snd`). Using the slowest algorithm, `euc_2gr`, it would take 1.2 seconds. Considering the usage of mutliple algorithms, these times will be even higher.

This section presents two methods to reduce computation time while keeping accuracy the same, sometimes even improving it.

[5]https://en.oxforddictionaries.com/explore/how-many-words-are-there-in-the-english-language/, visited on 26.12.2018

4.3.1 Preselection Algorithms

One possible solution to improve scaling is the application of the preselection algorithm presented in Section 3.4.1. The goal of this algorithm is to reduce the number of costly comparisons, by ruling out words from the comparison based on the first few characters. The idea is related to stemming, because they both follow the same assumption, that words and their different forms mostly have the same characters in the beginning.

For a given input word, the first n characters are matched against the first n characters of every word in the dictionary. If they are the congruent, the actual comparison using the string distance metrics is performed. If not, the word from the dictionary is ignored.

The goal of this algorithm is to reduce the number of costly comparisons and thus increase performance. To this end, a suitable n must be found, as a too small one would lead to too many comparisons and a large n might decrease accuracy, if the correct words are filtered out. Additionally, the size of n might vary with word sizes.

To find a fitting n, all words from Daniel Naber's Language Tool were taken and sorted by size. Then for each size, the percentage of words that start with the same n characters as their stem was calculated for all ns. Beginning with $n = 1$, each n was tested, and the first n with a percentage below 95% was chosen. The same was done using the soundex and cologne phonetics algorithms on the words and their stems. Words with sizes smaller than 4 have been ignored, but this limit was ignored for the cologne phonetics codes. Soundex codes have the length 4 by definition.

A table with optimal values ns for all word sizes can be found in Appendix 10. These values are only optimal in the scope of the presented tests and may need to be recalculated for other use cases.

Each of the 45 string distance metrics was tested with five different preselection algorithms: word-based, stem-based, soundex-based and cologne phonetics-based preselection. The fifth algorithm was a dummy function, that always returned `true`, so it is equivalent to using no preselection. These tests were performed on a testset of size 16000. For better readability, only the results for the fastest (`ham_snd`), the slowest (`euc_2gr`), one of the most accurate (`lev_stm`) and one of the least accurate (`ovr_1gr`) algorithms are presented. These four also represent all three types of string distance metrics.

Figure 4.5 shows the changes in accuracy and computation for the four algorithms. As expected, every preselection method reduces the computation time, in the case of `euc_2gr` with the stem-based preselection from 1082.22 seconds down to 4.05 seconds. Additionally, the accuracy benefits greatly from the preselection. `ovr_1gr` jumps from 0.625% to 72.32%, as the stemming-based preselection filters out many unfit words.

4.3.2 Thresholds

So far, the whole dictionary is iterated over in order to find the most similar word. With the preselection the computation time can be greatly reduced, but still every word needs to be examined. This can be prevented by using thresholds. Once the distance between an input word and any word in the dictionary is low enough, i.e. below a given threshold, the iteration can be stopped. The thresholds are specific to the distance metric.

Once again, it is important to find a fitting threshold, so that the accuracy is not diminished. To find optimal thresholds, every threshold between 0 and 0.5 in steps of 0.01 was tested using multiple testset sizes, until the thresholds converged to the final values,

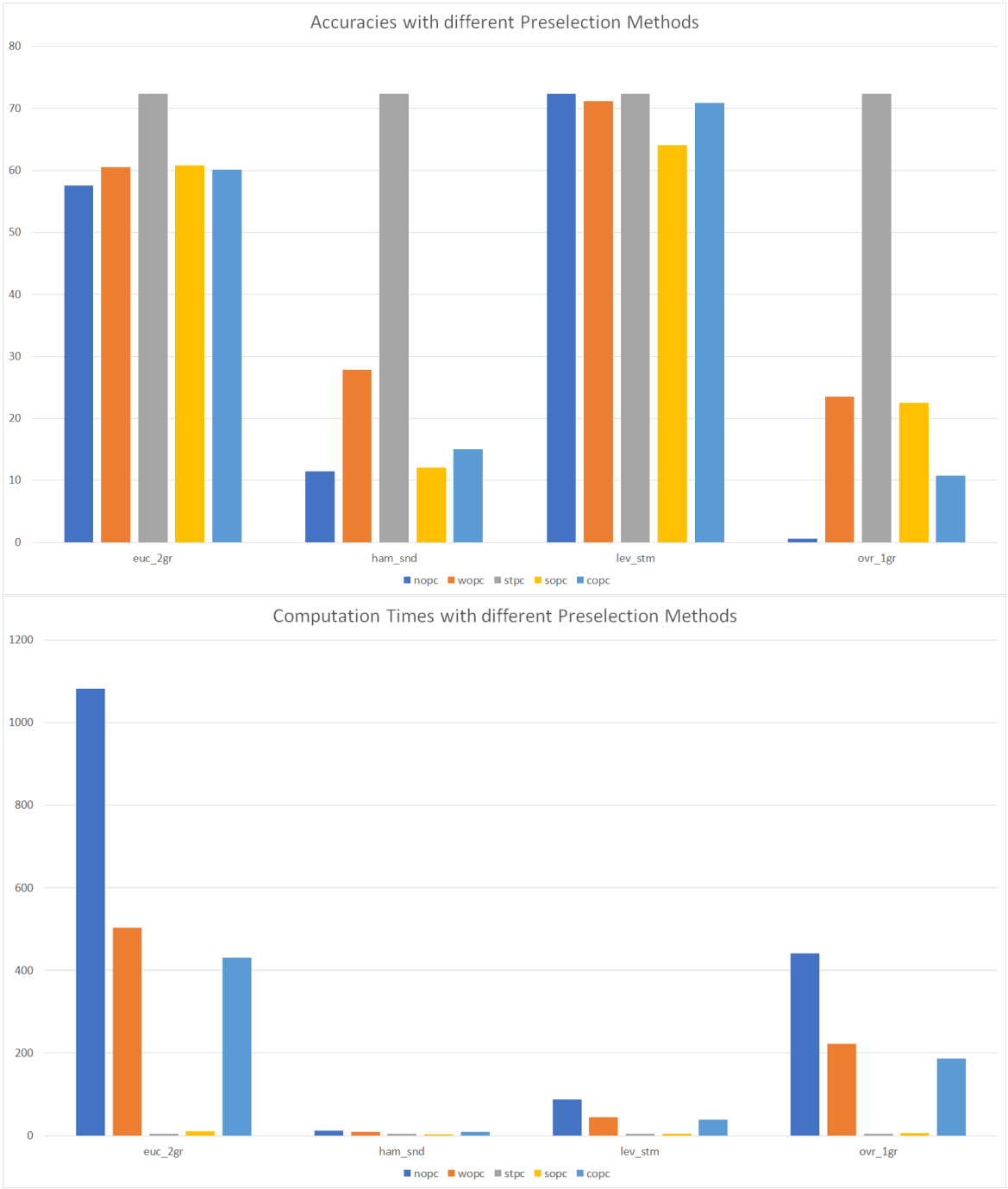


Figure 4.5: Comparison of accuracy and performance of the different preselection algorithms. These results are based on a testset of size 16000. The unit used in the accuracy figure is percent, the computation times are in seconds.

presented in Appendix 12.

Figure 4.6 shows the change in accuracy and performance for the aforementioned algorithms with and without stopping below the given threshold. This test was performed using

a testset size of 4000 words. As one can see, the accuracy drops by two to four percent, but computation time is reduced by a significant amount.

4.3.3 Combination of Thresholds and Preselection

Preselection and thresholds can be used in combination, further reducing the computation time, but also reducing the accuracy, as Figure 4.6 shows. For clarity, only the most accurate of the previous algorithms (`lev_stm`) is shown with the different preselection algorithms and the thresholds.

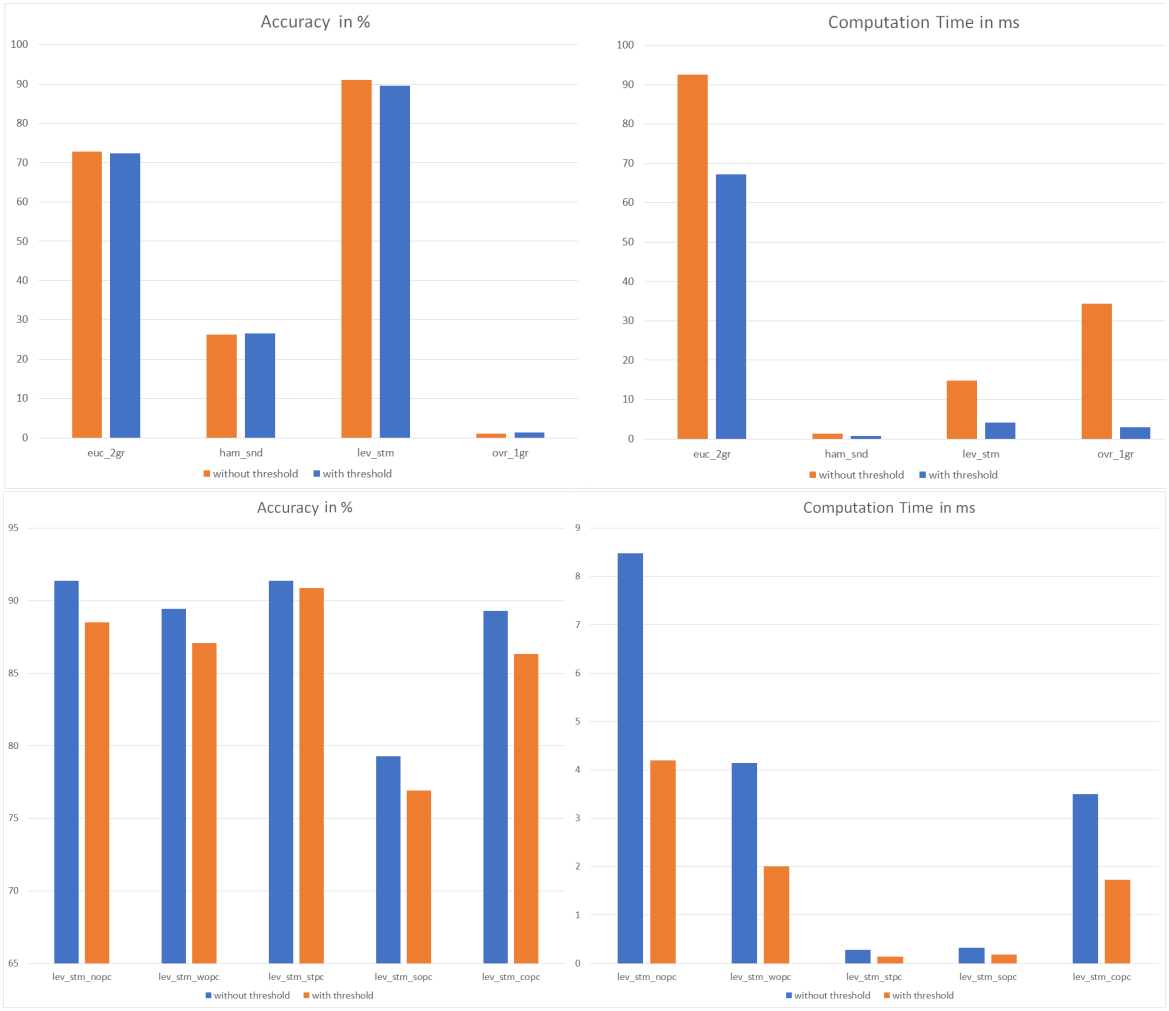


Figure 4.6: Comparison of accuracy and performance of the algorithms with and without thresholds. The testset used for this test contained 4000 words.

4.4 Ensemble Methods

The last section shows the possibilities and limitations of the string distance metrics when used individually. This section illustrates the effects of using an ensemble of multiple al-

gorithms. As presented in Section 2.4, two methods of combining the results are used: averaging and voting.

4.4.1 Assembling Ensembles

Ensembles are subsets of a set containing all 45 algorithms, $S = \{\texttt{lev\_wrd}, \texttt{lev\_stm}, ..., \texttt{cos\_3gr}\}$, so the number of possible combinations is the cardinality of the power set minus one, as the empty set is not an ensemble. The count of all possible ensembles therefore is $|\mathcal{P}(S)| - 1 = 2^{|S|} - 1 = 2^{45} - 1 = 35,184,372,088,832 - 1 = 35,184,372,088,831$. As it would be unreasonable to test all of these ensembles, a selection must be made.

First, all ensembles with an even number of algorithms were pruned, so that voting ensembles could not produce a tie. Then, a selection was made based on results from tests with all ensembles of size 3 and size 5. There are 14,190 ensembles of size 3 and 1,221,759 ensembles of size 5. It was found out, that over 80% of the 1000 most accurate ensembles of size 5 were supersets of the 1000 most accurate ensembles of size 3. This led to the assumption that supersets of good ensembles will still perform good.

Following this hypothesis, the 20 most accurate ensembles (the best ten voting and the best ten averaging ensembles) for a given size i were taken and all supersets of size $i+2$ were generated. These were tested with a testset of 1000 words four times each. Through repetition of this method up to $i = 43$, in total 137,577 ensembles were generated and tested with 1000 words.

4.4.2 Ensemble Accuracy

Additional tests with 2000, 4000, 8000 and 16000 words were made, but in order to keep testing times within reason, the number of ensembles was gradually tuned down. The following table shows the number of ensembles used for each testset size:

Testset Size	Number of Ensembles	Size Explanation
1,000	201	best 10 of each size
2,000	201	best 10 of each size
4,000	41	best 2 of each size
8,000	21	best of each size
16,000	6	best 6 overall

Figure 4.7: Illustration of ensemble count for each testset size between 1,000 and 16,000 words.

After each round of tests, the best n ensembles were kept for each ensemble size. Ensemble sizes are all odd numbers in range from 5 to 45. There is only one ensemble with size 45, namely the ensemble that contains all individual algorithms.

Each testset size was tested four times, and the results were averaged. Figure 4.8 shows the highest accuracy of the individual algorithms compared to the highest accuracy of all ensembles, grouped by testset size. While the accuracy of the best individual distance metric (`lev_stm`) clearly decreases with increasing word count, there is always at least one averaging ensemble that reaches 100% accuracy. The voting ensembles are slightly below 100%, but are still by far more accurate than the individual metrics.

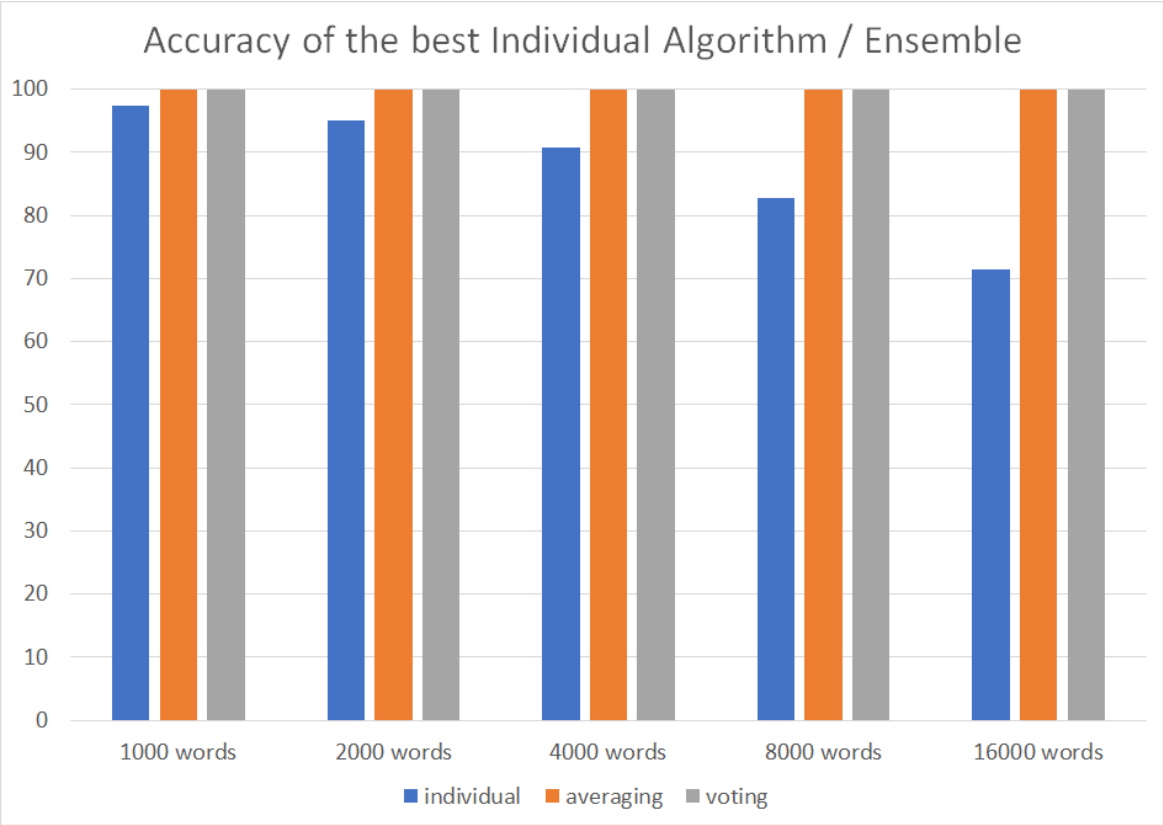


Figure 4.8: Comparison of the most accurate individual metric and ensembles with averaging and voting.

The smallest tested averaging ensemble, that classified all 16000 words correctly, contains `lev_stm`, `lcs_wrd`, `jar_stm`, `jwd_stm` and `man_2gr`. The best voting ensemble had 15952 correct classifications (99.7%) and contains `lev_stm`, `lss_wrd`, `jwd_wrd`, `ham_stm` and `jac_3gr`. The dominance of word-, stem- and cologne phonetics-based algorithms is apparent, but might be caused by the method of generating ensembles. Set- and vector-based algorithms seem to be less succesful in ensembles. The next sections try to enlighten the factors that determine the quality of an ensemble.

The used method for generating good ensembles is very time consuming, as one has to:

1. build ensembles
2. generate their supersets
3. test these supersets
4. prune the bad performing ensembles
5. repeat from step 2 until maximum size is reached

To find a better way to build ensembles, one must first understand, why a specific ensemble performs good or bad. The condorcet-jury-theorem states that the quality of a jury increases with the number of independet jurors. Translated to fit to the proposition of this thesis,

the quality of an ensemble increases with the number of individual algorithms, but they also must be as independent of each other as possible.

The following sections examine different aspects of ensembles, for example the size of an ensemble, and opposes them with the accuracy of the ensemble. To provide a measure of correlation, the Pearson correlation coefficient is calculated using the following formula:

$$r = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}} \tag{4.1}$$

where:

- $\sum$ is short for $\sum_{i=0}^{n}$
- n is the total number of word pairs
- x_i is the parameter, for example the ensemble size for ensemble i
- y_i is the accuracy of ensemble i

The pearson correlation coefficient has values between -1 and +1, where 0 is no correlation and -1 and +1 are total negative and positive correlations.

4.4.3 Average Accuracy

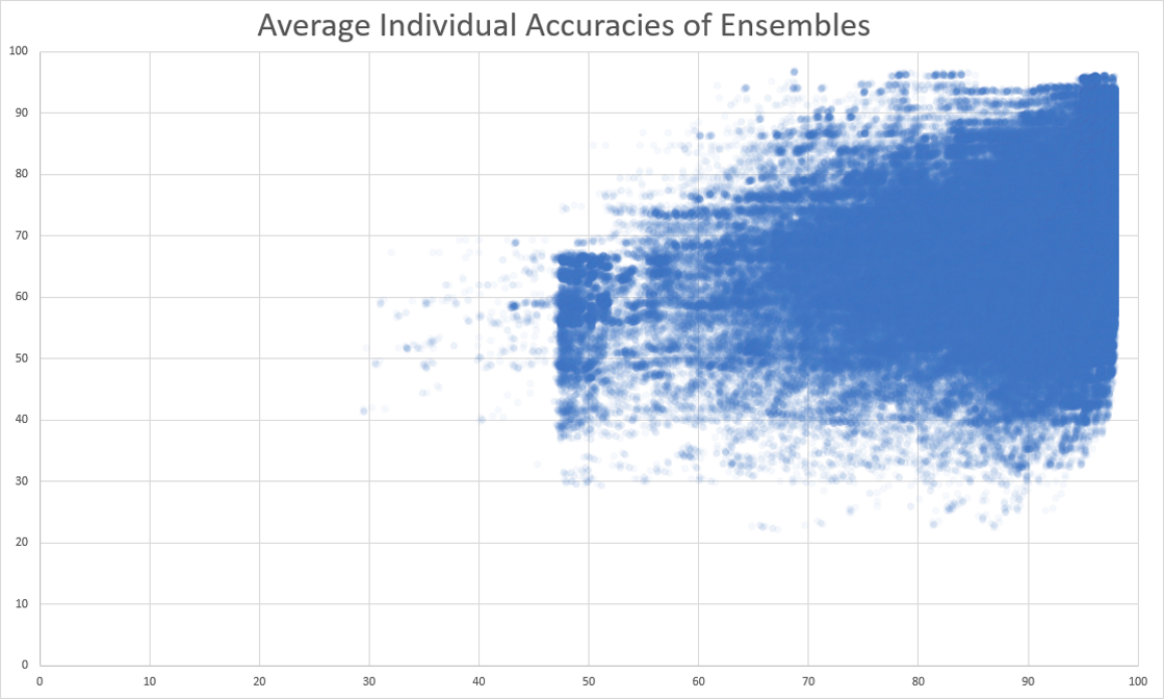


Figure 4.9: The average accuracy of ensemble members for each ensemble of size 5. The x-axis shows the accuracy of the ensemble, the y-axis the average accuracy of the ensemble's members.

To determine the correlation between the accuracies of the members of an ensemble and the ensemble's accuracy, all ensembles with five members were generated and tested with a testset of 1000 words. The results are presented in Figure 4.9.

The Pearson correlation coefficient between the average accuracy of an ensemble's members and the accuracy of the ensemble is 0.34040113, indicating a slight correlation.

4.4.4 Size

The previously mentioned 137,577 generated ensembles were taken as a basis for the size-based accuracy comparison. Each of these ensembles were tested four times with a testset containing 1000 words. Figure 4.10 shows the average accuracies of these ensembles. As one can see, the average accuracy meanders between 97% and 98% with the only outlier, namely size 3 with only 94.2439%. This suggests that up until five ensemble members, size plays an important role, but it is important to remember that the tested ensembles represent only a fraction of the possible ones (137,577 out of 35,184,372,088,831).

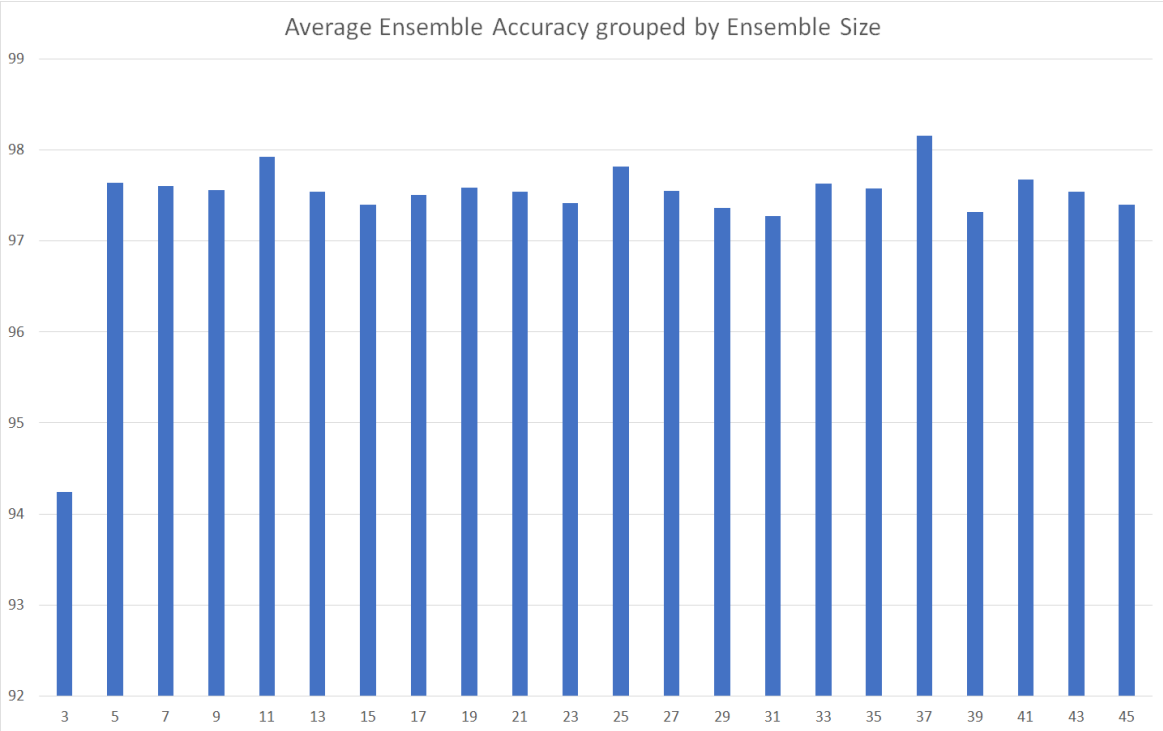


Figure 4.10: Comparison of the accuracy of different sized ensembles.

The correlation coefficient between the size of an ensemble and its accuracy is -0.09057907, supporting the assumption that size does play a minor role.

4.4.5 Correlation of Distance Metrics

With the subordinated role the size of the ensemble seems to play, this section illustrates the effect the average correlation between distance metrics in an ensemble has on the accuracy.

The correlation of distance metrics is also calculated using Pearson's correlation coefficient. The input data sets are the distances for each pair of words w_i, w_j calculated by

two different distance metrics X and Y. So each distance $d_X(w_i, w_j)$ is matched with the distance $d_Y(w_i, w_j)$.

The correlation test was conducted nine times with randomly generated testset containing 1000 words each. For a collection of 1000 words, there are $1000 \cdot 999 = 999000$ pairs of words. Distances are calculated for each of these pairs and for each distance metric, resulting in $999000 \cdot 45 = 44955000$ distance computations. The pearson correlation coefficient is calculated for every pair of distance metrics, i.e. $45 \cdot 44 = 1980$ times.

Appendix 8 shows a table for all 45 distance metrics. The correlation coefficient is lower for blue cells, and red cells indicate a higher correlation. The noticable red box area at the bottom right shows that the set-based and vector-based algorithms are more interrelated than the string-based algorithms. This does not come as a surprise regarding that each set- and vector-based algorithm is present three times, with different n-gram sizes.

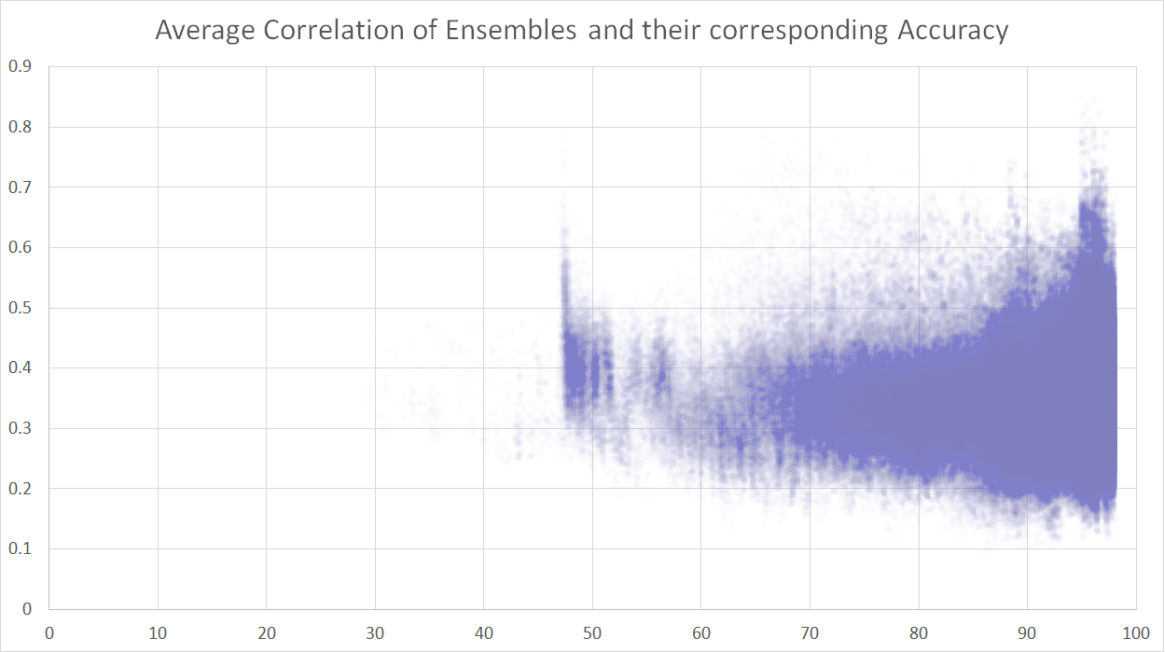


Figure 4.11: Relationship between Accuracy and Average Correlation in Ensembles of Size 5. The opacity illustrates the number of ensembles in this area, with darker color meaning more ensembles.

To calculate the average correlation of an ensemble, all pairs of members are built and their correlations are summed up. This sum is then divided by the number of member pairs.

Figure 4.11 shows the average correlation for all 1,221,759 ensembles of size 5 together with their accuracy. Although it cannot be observed that a lower average correlation entails a higher accuracy, most ensembles' average correlation lies below 0.5. It is interesting to notice that the spread of the average correlation grows with the accuracy of the ensemble.

The average correlation of metrics in an ensemble was contrasted to the accuracy, and the pearson correlation coefficient was calculated for this comparison, yielding a result of 0.04673. Again, it can not be generally said that the correlation between ensemble members plays an important role for the accuracy of the ensemble.

4.4.6 Ensemble Performance

This section compares the performance overhead of the two ensemble methods, averaging and voting. To this end, 45 ensembles of all sizes were built. The following pseudo code snippets show the different algorithms:

```
//averaging
stopwatch.start();
foreach (input_word : input_dictionary) {
  foreach (target_word : target_dictionary) {
    foreach (metric : ensemble) {
      calculate distance between input_word and target_word
    }
    calculate the average distance
    if (average distance < current minimum) {
      store average distance as current minimum
    }
  }
  return target_word with lowest average
}
stopwatch.stop();
```

```
//voting ensemble
stopwatch.start();
foreach (input_word : input_dictionary) {
  foreach (metric : ensemble) {
    foreach (target_word : target_dictionary) {
      calculate distance between input_word and target_word using metric
      if (distance < current minimum) {
        store distance as current minimum
      }
    }
  }
  count the votes by each ensemble
  return the target_word with most votes
}
stopwatch.stop();
```

The tests were conducted with 100 input words and a 10.000 target words. Figure 4.12 shows the time it took each ensemble to perform the classification. The steep rise in time at ensemble size 24 can be explained by looking at Figure 4.4, as the metric with number 24 is the first set-based algorithm, that is substantially slower.

It can be seen that voting introduces more performance overhead than averaging.

4.5 Sentences

Sentences can be classified similarly to the way words are found in the dictionary.

Sentences can be seen as a sequence of words (or their ids). These sequences are very similar to strings, but the number of characters increases from 26 characters (only lower case) to the number of words in the dictionary. As the distance metrics do not compare individual items in a sequence, or characters in a string (for example `a` is closer to `b` than to `c`), but only equality checks (`a` is not equal to `b`), this has no effect on the distance metrics.

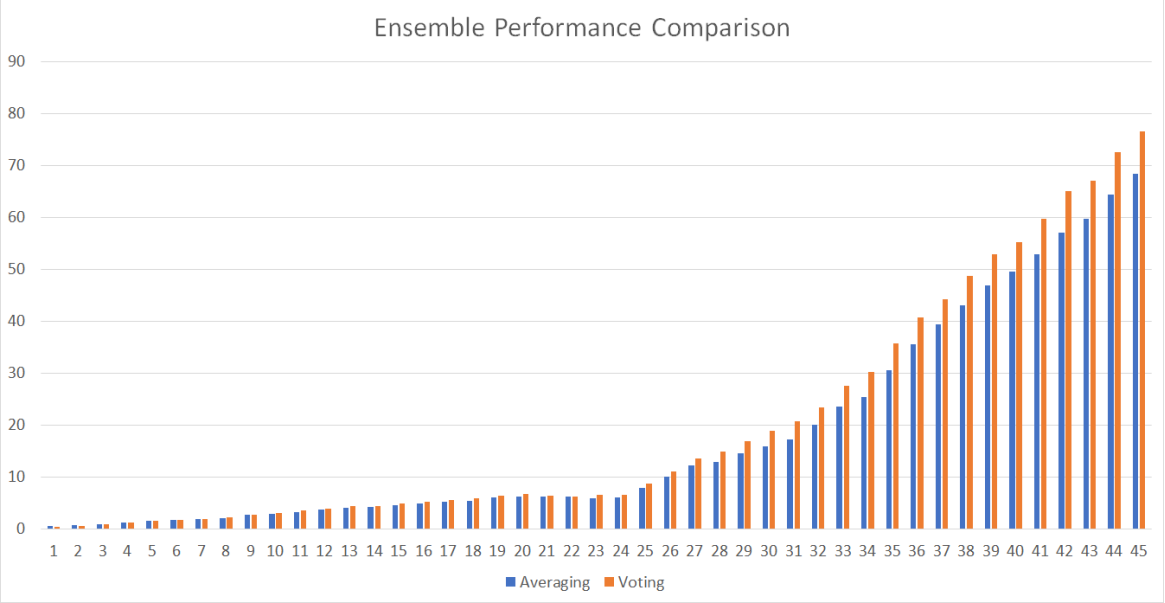


Figure 4.12: Time in seconds it takes to classify a word using different ensemble sizes.

The addition of variables as presented in Section 3.4.3 brings another slight change. To realize this feature, all variables of a single type are mapped to the same id. For example, `hot dog`, `pizza` and `burger` could all share the id 42. This way, the sentences `"Where can I buy a hot dog"`, `"Where can I buy a pizza?"` and `"Where can I buy a burger?"` are transformed into the same sequence of ids.

The last change is the introduction of classes. One could argue that the dictionary is a collection of classes with only one observation each. Or in other words, each word of the dictionary represents its own class. For classes with multiple observations, there are two other classification algorithms that can be used. These methods are the k-nearest-neighbour algorithm and the nearest centroid classifier, presented in Section 2.3.

As the triangle inequality is not fulfilled for all distance metrics, the centroid for every class must be recalculated for every input question.

To test the classification of sentences it makes sense to split the testsets into two groups, a sample set and a validation set. The sample set depicts the texts that are known to the ETeC system, and the validation set are a combination of known and unknown sentences that need to be classified.

4.5.1 Europa Park Testset

The Europa Park testset is used to validate the functionality of variables in sentences. As there are only a few questions it is not well suited for splitting these questions into a sample and validation set, so all input sentences were known to the ETeC system. The used questions and variables can be seen in Appendix 13 and 14.

For the tests, all questions were assigned random variables of the correct type four times, so each sentence was classified four times with different variables. The dictionary contained the 189 words that were present in the sentences, and the most accurate ensemble was used to classify the words of the input questions, namely the averaging ensemble presented in

Section 4.4.2.

As in the context of questions, neither stemming nor soundex or cologne phonetics are useful, the string-based distance metrics were used on the sequences of word ids. Set- and vector-based approaches were given the bigrams of the sequences, as they proved to be the most accurate. Sentence classification was done by an ensemble that contained all of these metrics and used the averaging method.

With these small sentence and word collections, it does not come as a surprise that all sentences and variables were classified correctly. On average the classification of a single sentence took less than 5ms, which allows for the use of the ETeC system on a variety of different devices.

4.5.2 SMS Spam Collection

This testset contrasts with the europa park testset, as it has many questions (5574), but only two classes. This allows for a few interesting observations considering the needed number of sample questions per class. For the tests, a fraction of these questions were chosen randomly to build the classes. These were used to classify all questions. The number of correct classifications divided by the number of questions provides the accuracy of the method. All tests were performed with an ensemble containing all the distance metrics presented in the previous section.

The following table shows the different sizes for the sample set and classification set, as well as the results using the different classification algorithms, namely k-nearest neighbor (knn), nearest centroid classification (ncc) and simply assuming the class of the closest sample (cs). Additionally, the total time it took to run all three methods was recorded. The sets were chosen randomly from the collection. To reduce the factor of randomness, each test was conducted four times and the results were averaged.

Sample Set Size	**Test Set Size**	**cs**	**knn**	**ncc**	$\frac{\textbf{total time}}{\textbf{sample set size}}$
100	100	100%	92%	50%	21.873s
100	200	92%	87.5%	39%	42.4111s
100	500	90.6%	90%	40.4%	107.987s
100	1000	88.4%	88%	43.1%	189.126s
100	2000	88.3%	88.7%	48.2%	365.318s
100	5574	92.88%	91.8%	41.6%	804.433s
50	1000	84.7%	85.6%	44.1%	130.284s
200	1000	89.4%	89.4%	29.5%	231.544s
500	1000	92.6%	90.1%	34.6%	482.468s
1000	5574	93.88%	91.8%	41.6%	3615.08s

The last four tests show that a larger sample set does not mean a significantly higher accuracy, if the augmentation of the sample set is not specifically tailored to the wrongly classified observations.

Generally it can be seen that the nearest centroid classifier seems to be unfitting for this use case. Additionally, the number of sample question must be chosen based on the requirements of the case of application. A high number increases the accuracy but also reduces performance. As ETEC does not need to be trained, it is possible to start with a minimum number of questions and add ones that are not classified correctly, thus increasing the number of questions based on real demand.

5 Conclusion and Outlook

This thesis presents the ETeC, a system for text classification using ensembles of simple string-, set- and vector-based algorithms, with the restriction of not using any kind of learning algorithms. Without learning, the classification can be refined and improved at runtime by adding new sample texts. There is no model that needs to be trained first.

First, common string comparison methods are presented, as well as transformations from strings to sets and vectors, thus enabling the system to use comparison algorithms from these domains. In total, 13 algorithms are presented. Additionally, different representations, such as the soundex algorithm or n-grams are illustrated and used to augment the number of possible ensemble methods to 45. For the six string-based algorithms, the different representations are the plain string, its stem, its soundex code and its cologne phonetics code. The seven set- and vector-based metrics are applied using n-grams with $1 \leq n \leq 3$. Each of these metrics is used as a classifier, assigning the class, or word, with the lowest distance to a new observation.

After describing the functionalities of these comparison algorithms, the used ensemble methods are presented, along with different classification approaches. An ensemble is a collection of individual classifiers that can overcome the shortcomings of individual metrics by combining the results. The presented methods for the combination of results are averaging for numeric values and voting for nominal values.

To determine the correct class of a new observation, there are two techniques that are used in this thesis: the k-nearest-neighbor classifier (kNN) and the nearest centroid classifier (NCC). Using the kNN, an object is classified by a majority vote of its k-nearest neighbors. The NCC calculates the mean distance to each class for a new observation, called the centroid. As its name implies, the new observation is assigned the class with the nearest centroid.

There are two classifications that take place in the ETeC system: first, each word is matched to its most similar word from a dictionary, thus reduced to an id, and then the sequence of these ids are matched against the classes of known sentences, thus determining the correct class of the sentence.

The dictionary is built by providing the ETeC system with a collection of sample sentences. Each word from these sentences is then reduced to its stem and put in the dictionary. This method is chosen to minimize the dictionary size, as different forms of the same word are reduced to the same stem. Finding the most similar word in the dictionary can be seen as the kNN algorithm with $k = 1$, the finding of the single nearest neighbor.

Several tests were conducted to contrast the individual metrics with ensembles regarding performance and accuracy. Obviously in terms of performance, an ensemble is always slower than any of its members, but the accuracy of ensembles surpasses the individual metrics. Using two testsets, the SMS Spam Collection[1] and an testset of questions individually built for this thesis in collaboration with the Europa Park Rust[2], several tests were conducted to

[1] https://archive.ics.uci.edu/ml/datasets/sms+spam+collection, visited on 03.01.2019
[2] https://www.europapark.de/, visited on 06.01.2019

show that sentences can be correctly classified, even when containing variables.

The tests also showed that performance can be greatly improved by using thresholds and preselection. If the distance to an examined word is below a threshold, all other comparisons can be skipped, as it is likely that the current word is the correct one. Preselection prevents costly comparisons with words that can be ruled out based on easily recognizable features. In this thesis, these features were the first few characters of the words or their soundex and cologne phonetics codes.

The thesis tries to provide insight into the building of ensembles and what to look for in order to build an adequate ensemble. To this end, the average accuracy of ensemble members, the correlation among the ensemble members and the size of an ensemble is examined and contrasted with the ensemble's accuracy. Opposing to the condorcet-jury-theorem, which states that the more independent members a jury has, the more likely it is to choose the correct alternative of two options, the only factor that seems to play a major role in ensemble building is the accuracy of the individual members. The importance of the size of an ensemble is greatly diminished, once the ensemble has five or more members, and the correlation between members does not seem to have an impact at all.

The main advantage of ETeC over comparable systems is the flexibility. As there is no learning or training involved at any time, words and sentences can be added or removed at will. For example, if in a specific use case, there is a question that is often assigned the wrong class, this question can simply be added to the collection of known questions without further effort. Even classes can be added and removed at runtime.

The research questions proposed in section 3.5 are all answered within this thesis.

5.1 Future Work

Although the tests provided show the advantages of using ensembles, there is still room for a vast amount of improvements.

One shortcoming is that variables can only consist of one word. Having multi-word variables would probably lead to wrong classifications, especially when the variable part is longer than the static part. For example, asking for showtimes in a cinema could be done using the template `"When is 'movie-variable' on?"`. With long titles this could lead to sentences like `"When is 'Fantastic Beasts and Where to Find Them' on?"`.

This thesis ignores the possible optimizations through parallelism. The ensembles can be highly paralleled as there is no interaction of the metrics, only their results are used to build the average or the vote. Additionally, currently the classification is implemented as a linear search, which can be split up and handled by multiple threads at once.

The concept of not having a corresponding value in the dictionary does not exist in neither the word-level nor the sentence-level analysis, there is always one element with a minimal distance. Adding this concept might affect the accuracy considerarbly and must be examined. A simple solution could be a threshold, over which a word or sentence is marked unknown. For voting, a minimal number of votes could be specified, for example, half of all voters must vote for the same item.

Much insight is to be gained in the area of ensemble building. The suggested reasons for ensembles with high accuracies leave much to be desired, as in the case of the presented 45 individual classifiers, 35,184,372,088,832 ensembles can be built. The trial and error procedure of this thesis is not feasible for higher numbers of classifiers, and even in this

case only covered a small fraction of all ensembles. The set of examined ensembles consists of all ensembles of size three (14,190 ensembles) and five (1,221,759 ensembles) and the generated supersets of the best performing ensembles (137,577 ensembles), which make up only 0.000003903% (1.373.526 of 35.184.372.088.832 ensembles).

Lastly, ETeC groups sentences by intent, i.e. stores sentences in classes, but for words, the possibility of using classes is neglected so far. It is thinkable, that words could also be grouped by meaning, so that synonyms belong to the same class. This could reduce the amount of stored sentences, as for example `Let's go over there` and `Let's walk over there` would provide the same sequence of word-class ids.

Appendix

1 Model

1.1 The Word Class

```
// source: own implementation
class Word {
public:
  Word();
  Word(const std::string & pWord, const int & pId);
  Word(const Word & other);
  ~Word();

  std::string word, soundex, cologne, stem;
  std::map<std::string, int> unigram_vector, bigram_vector, trigram_vector;
  std::set<std::string> unigram_set, bigram_set, trigram_set;
  int id;

  bool operator <(const Word & other) {
    return word < other.word;
  }
  bool operator ==(const Word & other) {
    return word.compare(other.word) == 0;
  }
};
```

1.2 The Question Class

```
// source: own implementation
class Question
{
private:
  std::vector<size_t> ids;
  std::string type;
  std::string text;

  std::string answer_text;

  std::map<std::string, std::vector<size_t>> vars;
public:
  Question(const std::vector<size_t> & pIds, const std::string pType, const
      std::string pText);
  Question(const Question & other);
  ~Question();

  void SetAnswer(const std::string & answer);
  const std::string & GetAnswer();

  const std::string & GetType() const;
  const std::vector<size_t> & GetIDs() const;
  std::string GetText() const;
};
```

2 Distance Metrics

2.1 Levenshtein Distance

```
//Source: https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/
    Levenshtein_distance#C++
//visited on 23.12.2018
float levenshteinDistance(const Word & wrd1, const Word & wrd2, const short &
     type) {
  const std::string & str1 = type == 1 ? wrd1.word :
    type == 2 ? wrd1.soundex :
    type == 3 ? wrd1.cologne :
    type == 4 ? wrd1.stem : std::string();
  const std::string & str2 = type == 1 ? wrd2.word :
    type == 2 ? wrd2.soundex :
    type == 3 ? wrd2.cologne :
    type == 4 ? wrd2.stem : std::string();

  if (str1.empty() || str2.empty()) {
    return -1;
  }

  const std::size_t len1 = str1.size(), len2 = str2.size();
  std::vector<unsigned int> col(len2 + 1), prevCol(len2 + 1);

  for (unsigned int i = 0; i < prevCol.size(); i++) {
    prevCol[i] = i;
  }
  for (unsigned int i = 0; i < len1; i++) {
    col[0] = i + 1;
    for (unsigned int j = 0; j < len2; j++) {
      col[j + 1] = std::min(std::min(prevCol[1 + j] + 1, col[j] + 1), prevCol
          [j] + (str1[i] == str2[j] ? 0 : 1));
    }
    col.swap(prevCol);
  }
  return ScaleToRange(prevCol[len2], { 0,std::max(len1, len2) }, { 0,1 });
}
```

2.2 Longest Common Substring

```
//Source: https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/
    Longest_Common\_Substring#C++_2
//visited on 23.12.2018
float longestCommonSubstring(const Word & wrd1, const Word & wrd2, const
    short & type)
{
  const std::string & str1 = type == 1 ? wrd1.word :
    type == 2 ? wrd1.soundex :
    type == 3 ? wrd1.cologne :
    type == 4 ? wrd1.stem : std::string();
  const std::string & str2 = type == 1 ? wrd2.word :
    type == 2 ? wrd2.soundex :
    type == 3 ? wrd2.cologne :
    type == 4 ? wrd2.stem : std::string();

  if (str1.empty() || str2.empty()) {
    return -1;
  }

  int *curr = new int[str2.size()];
  int *prev = new int[str2.size()];
  int *swap = nullptr;
  int maxSubstr = 0;

  for (int i = 0; i<str1.size(); ++i) {
    for (int j = 0; j<str2.size(); ++j) {
      if (str1[i] != str2[j]) {
        curr[j] = 0;
      } else {
        if (i == 0 || j == 0) {
          curr[j] = 1;
        } else {
          curr[j] = 1 + prev[j - 1];
        }

        if (maxSubstr < curr[j]) {
          maxSubstr = curr[j];
        }
      }
    }
    swap = curr;
    curr = prev;
    prev = swap;
  }
  delete[] curr;
  delete[] prev;

  return 1 - ScaleToRange(maxSubstr, { 0,std::min(str1.size(),str2.size()) },
      { 0,1 });
}
```

2.3 Longest Common Subsequence

```
//Source: https://www.geeksforgeeks.org/space-optimized-solution-lcs/
//visited on 11.01.2019
float longestCommonSubsequence(const Word & wrd1, const Word & wrd2, const
    short & type) {
  const std::string & str1 = type == 1 ? wrd1.word :
    type == 2 ? wrd1.soundex :
    type == 3 ? wrd1.cologne :
    type == 4 ? wrd1.stem : std::string();
  const std::string & str2 = type == 1 ? wrd2.word :
    type == 2 ? wrd2.soundex :
    type == 3 ? wrd2.cologne :
    type == 4 ? wrd2.stem : std::string();

  if (str1.empty() || str2.empty()) {
    return -1;
  }

  std::vector<std::vector<int>> L(2, std::vector<int>(str2.size() + 1));

  // Binary index, used to index current row and previous row.
  bool bi;

  for (int i = 0; i <= str1.size(); ++i) {
    // Compute current binary index
    bi = i & 1;

    for (int j = 0; j <= str2.size(); ++j) {
      if (i == 0 || j == 0) {
        L[bi][j] = 0;
      } else if (str1[i - 1] == str2[j - 1]) {
        L[bi][j] = L[1 - bi][j - 1] + 1;
      } else {
        L[bi][j] = std::max(L[1 - bi][j], L[bi][j - 1]);
      }
    }
  }

  return 1 - ScaleToRange(L[bi][str2.size()],
    { 0,std::min(str1.size(),str2.size()) }, { 0,1 });
}
```

2.4 Jaro Similarity

```
// based on https://rosettacode.org/wiki/Jaro_distance
// visited on 15.01.2019
float jaroSimilarity(const Word & wrd1, const Word & wrd2, const short & type
    ) {
  const std::string & str1 = type == 1 ? wrd1.word :
    type == 2 ? wrd1.soundex :
    type == 3 ? wrd1.cologne :
    type == 4 ? wrd1.stem : std::string();
  const std::string & str2 = type == 1 ? wrd2.word :
    type == 2 ? wrd2.soundex :
    type == 3 ? wrd2.cologne :
    type == 4 ? wrd2.stem : std::string();

  if (str1.empty() || str2.empty()) {
    return -1;
  }

  const unsigned int l1 = str1.size(), l2 = str2.size();
  if (l1 + l2 == 0) {
    return 0;
  }
  if (l1 + l2 == 1) {
    return 1;
  }
  if (l1 == 1 && l2 == 1) {
    return str1[0] == str2[0] ? 0 : 1;
  }

  std::vector<short> s1_matches(l1);
  std::vector<short> s2_matches(l2);

  for (auto & e : s1_matches) {
    e = 0;
  }
  for (auto & e : s2_matches) {
    e = 0;
  }

  //calculate matches
  float m = 0;
  int max_distance = std::floor(std::max(l1, l2) / 2.f) - 1;
  for (int i = 0; i < l1; ++i) {
    for (int j = -max_distance; j <= max_distance; ++j) {
      //bounds checking
      if (i + j >= 0 && i + j < l2) {
        if (str1[i] == str2[i + j] && !s2_matches[i + j]) {
          ++m;
          s1_matches[i] = 1;
          s2_matches[i + j] = 1;
          break;
        }
      }
    }
  }
```

```
  if (m == 0) { return 1; }

  //calculate number of transpositions
  float t = 0.0;
  unsigned int k = 0;
  for (unsigned int i = 0; i < l1; i++) {
    if (s1_matches[i])
    {
      while (!s2_matches[k]) k++;
      if (str1[i] != str2[k]) t += 0.5;
      k++;
    }
  }

  t = floor(t);

  return 1.f - ((m / l1 + m / l2 + (m - t) / m) / 3.0);
}
```

2.5 Jaro Winkler Distance

```
// source: own implementation
float jaroWinklerDistance(const Word & wrd1, const Word & wrd2, const short &
    type, const float & jaroSim)
{
  const std::string & str1 = type == 1 ? wrd1.word :
    type == 2 ? wrd1.soundex :
    type == 3 ? wrd1.cologne :
    type == 4 ? wrd1.stem : std::string();
  const std::string & str2 = type == 1 ? wrd2.word :
    type == 2 ? wrd2.soundex :
    type == 3 ? wrd2.cologne :
    type == 4 ? wrd2.stem : std::string();

  if (str1.empty() || str2.empty()) {
    return -1;
  }

  float distance = 1.f - (jaroSim != -1 ? jaroSim : jaroSimilarity(wrd1, wrd2
      , type));
  unsigned int prefixLength = 0, i = 0;
  while (str1.size() > i && str2.size() > i && i < 4 && str1[i] == str2[i]) {
    ++prefixLength;
    ++i;
  }

  return 1.f - (distance + (prefixLength * 0.1 * (1 - distance)));
}
```

2.6 Hamming Distance

```
// source: own implementation
float hammingDistance(const Word & wrd1, const Word & wrd2, const short &
    type) {
  const std::string & str1 = type == 1 ? wrd1.word :
    type == 2 ? wrd1.soundex :
    type == 3 ? wrd1.cologne :
    type == 4 ? wrd1.stem : std::string();
  const std::string & str2 = type == 1 ? wrd2.word :
    type == 2 ? wrd2.soundex :
    type == 3 ? wrd2.cologne :
    type == 4 ? wrd2.stem : std::string();

  if (str1.empty() || str2.empty()) {
    return -1; // abstention
  }

  float hamming = abs(static_cast<float>(str1.size()) - str2.size());

  for (int i = 0; i < std::min(str1.size(), str2.size()); ++i) {
    if (str1[i] != str2[i]) {
      ++hamming;
    }
  }

  return hamming / std::max(str1.size(), str2.size());
}
```

2.7 Jaccard Index

```
// source: own implementation
float jaccardIndex(const Word & w1, const Word & w2, const short & n)
{
  const std::set<std::string> set1 = n == 1 ? w1.unigram_set :
    n == 2 ? w1.bigram_set :
    n == 3 ? w1.trigram_set : std::set<std::string>();
  const std::set<std::string> set2 = n == 1 ? w2.unigram_set :
    n == 2 ? w2.bigram_set :
    n == 3 ? w2.trigram_set : std::set<std::string>();

  if (set1.empty() || set2.empty()) {
    return -1; // abstention
  }

  float intersections = 0;
  for (const auto & c : set1) {
    if (std::find(set2.begin(), set2.end(), c) != set2.end()) {
      ++intersections;
    }
  }

  return 1 - (intersections / (set1.size() + set2.size() - intersections));
}
```

2.8 Dice's Coefficient

```
// source: own implementation
float dicesCoefficient(const Word & w1, const Word & w2, const short & n)
{
  const std::set<std::string> set1 = n == 1 ? w1.unigram_set :
    n == 2 ? w1.bigram_set :
    n == 3 ? w1.trigram_set : std::set<std::string>();
  const std::set<std::string> set2 = n == 1 ? w2.unigram_set :
    n == 2 ? w2.bigram_set :
    n == 3 ? w2.trigram_set : std::set<std::string>();

  if (set1.empty() || set2.empty()) {
    return -1; // abstention
  }

  float intersections = 0;

  for (const auto & e : set1) {
    if (set2.find(e) != set2.end()) {
      ++intersections;
    }
  }

  return 1 - ((2.f*intersections) / (set1.size() + set2.size()));
}
```

2.9 Overlap Coefficient

```
// source: own implementation
float overlapCoefficient(const Word & w1, const Word & w2, const short & n)
{
  const std::set<std::string> set1 = n == 1 ? w1.unigram_set :
    n == 2 ? w1.bigram_set :
    n == 3 ? w1.trigram_set : std::set<std::string>();
  const std::set<std::string> set2 = n == 1 ? w2.unigram_set :
    n == 2 ? w2.bigram_set :
    n == 3 ? w2.trigram_set : std::set<std::string>();

  if (set1.empty() || set2.empty()) {
    return -1; // abstention
  }

  float intersection = 0;
  for (const auto & c : set1) {
    if (std::find(set2.begin(), set2.end(), c) != set2.end()) {
      ++intersection;
    }
  }

  return 1 - (intersection / std::min(set1.size(), set2.size()));
}
```

2.10 Euclidian Distance

```
// source: own implementation
float euclidianDistance(const Word & w1, const Word & w2, const short & n)
{
  const std::map<std::string, int> & vec1 = n == 1 ? w1.unigram_vector :
    n == 2 ? w1.bigram_vector :
    n == 3 ? w1.trigram_vector : std::map<std::string, int>();
  const std::map<std::string, int> & vec2 = n == 1 ? w2.unigram_vector :
    n == 2 ? w2.bigram_vector :
    n == 3 ? w2.trigram_vector : std::map<std::string, int>();

  if (vec1.empty() || vec2.empty()) {
    return -1; //abstention
  }

  // build vectors
  int sum1 = 0, sum2 = 0;
  std::map<std::string, std::pair<int, int>> vectors;
  for (const auto & c : vec1) {
    vectors[c.first].first = c.second;
    sum1 += c.second;
  }
  for (const auto & c : vec2) {
    vectors[c.first].second = c.second;
    sum2 += c.second;
  }

  //calculate distance
  float result = 0;
  for (const auto & e : vectors) {
    result += pow(e.second.first - e.second.second, 2);
  }

  float max_distance = sqrt(pow(sum1, 2) + pow(sum2, 2));

  return ScaleToRange(sqrt(result), { 0,max_distance }, { 0,1 });
}
```

2.11 Manhattan Distance

```
// source: own implementation
float manhattanDistance(const Word & w1, const Word & w2, const short & n)
{
  const std::map<std::string, int> & vec1 = n == 1 ? w1.unigram_vector :
    n == 2 ? w1.bigram_vector :
    n == 3 ? w1.trigram_vector : std::map<std::string, int>();
  const std::map<std::string, int> & vec2 = n == 1 ? w2.unigram_vector :
    n == 2 ? w2.bigram_vector :
    n == 3 ? w2.trigram_vector : std::map<std::string, int>();

  if (vec1.empty() || vec2.empty()) {
    return -1; //abstention
  }

  // build vectors
  int sum1 = 0, sum2 = 0;
  std::map<std::string, std::pair<int, int>> vectors;
  for (const auto & c : vec1) {
    vectors[c.first].first = c.second;
    sum1 += c.second;
  }
  for (const auto & c : vec2) {
    vectors[c.first].second = c.second;
    sum2 += c.second;
  }

  //calculate distance
  float result = 0;
  for (const auto & e : vectors) {
    result += abs(e.second.first - e.second.second);
  }

  return ScaleToRange(result, { 0,sum1+sum2 }, { 0,1 });
}
```

2.12 Simple Matching Coefficient

```
// source: own implementation
float simpleMatchingCoefficient(const Word & w1, const Word & w2, const short
    & n)
{
  const std::map<std::string, int> & vec1 = n == 1 ? w1.unigram_vector :
    n == 2 ? w1.bigram_vector :
    n == 3 ? w1.trigram_vector : std::map<std::string, int>();
  const std::map<std::string, int> & vec2 = n == 1 ? w2.unigram_vector :
    n == 2 ? w2.bigram_vector :
    n == 3 ? w2.trigram_vector : std::map<std::string, int>();

  if (vec1.empty() || vec2.empty()) {
    return -1; //abstention
  }

  // build vectors
  std::map<std::string, std::pair<float, float>> vectors;
  for (const auto & c : vec1) {
    vectors[c.first].first = c.second;
  }
  for (const auto & c : vec2) {
    vectors[c.first].second = c.second;
  }

  float common_attributes = 0;
  for (const auto & e : vectors) {
    if (e.second.first == e.second.second) {
      ++common_attributes;
    }
  }

  return 1 - (common_attributes / vectors.size());
}
```

2.13 Cosine Similarity

```
// source: own implementation
float cosineSimilarity(const Word & w1, const Word & w2, const short & n)
{
  const std::map<std::string, int> & vec1 = n == 1 ? w1.unigram_vector :
    n == 2 ? w1.bigram_vector :
    n == 3 ? w1.trigram_vector : std::map<std::string, int>();
  const std::map<std::string, int> & vec2 = n == 1 ? w2.unigram_vector :
    n == 2 ? w2.bigram_vector :
    n == 3 ? w2.trigram_vector : std::map<std::string, int>();

  if (vec1.empty() || vec2.empty()) {
    return -1; //abstention
  }

  // build vectors
  std::map<std::string, std::pair<float, float>> vectors;
  for (const auto & c : vec1) {
    vectors[c.first].first = c.second;
  }
  for (const auto & c : vec2) {
    vectors[c.first].second = c.second;
  }

  float sum = 0, magn_s1_sq = 0, magn_s2_sq = 0;
  for (const auto & e : vectors) {
    sum += e.second.first * e.second.second;
    magn_s1_sq += pow(e.second.first, 2);
    magn_s2_sq += pow(e.second.second, 2);
  }


  return 1 - (sum / sqrt(magn_s1_sq * magn_s2_sq));
}
```

3 Testset Generation

```
// source: own implementation
void Dictionary::ReadWords(std::string filename, size_t max_lines) {
  words.clear();
  stems.clear();

  correct_ids.clear();

  //open the file
  std::ifstream is(filename);
  std::string line;

  //store all words
  std::vector<std::string> all_words;
  while (std::getline(is, line)) {
    all_words.emplace_back(line);
  }

  //create a randomizer
  std::random_device rd;
  std::mt19937 g(rd());

  //create a vector of randomized indices
  std::vector<unsigned int> indices(all_words.size());
  std::iota(indices.begin(), indices.end(), 0);
  std::shuffle(indices.begin(), indices.end(), g);

  for (int i = 0; i < max_lines; ++i) {
    Word w = Word(all_words[indices[i]], words.size());
    Word b = Word(w.stem, stems.size());
    words.emplace(w);
    stems.emplace(b);

    correct_ids[w.id] = b.id;
  }
}
```

4 String Distance Accuracy Test

```
// source: own implementation
void Dictionary::IndividualAccuracyTest() {
  float min_dist = std::numeric_limits<float>::max();
  float curr_dist = std::numeric_limits<float>::max();
  size_t correct = 0;
  int result_id = -1;

  J::Duration duration;

  for (const auto & f : functions) {

    correct = 0;
    duration.start();

    for (const auto & w : words) {

      min_dist = std::numeric_limits<float>::max();
      curr_dist = std::numeric_limits<float>::max();
      result_id = -1;

      for (const auto & s : stems) {
        curr_dist = f.function(w, s);
        if (curr_dist < min_dist) {
          min_dist = curr_dist;
          result_id = s.id;
        }
      }

      if (correct_ids[w.id] == result_id) {
        ++correct;
      }
    }

    duration.stop();

    saveResult(f.id, correct / words.size(), duration);
  }

}
```

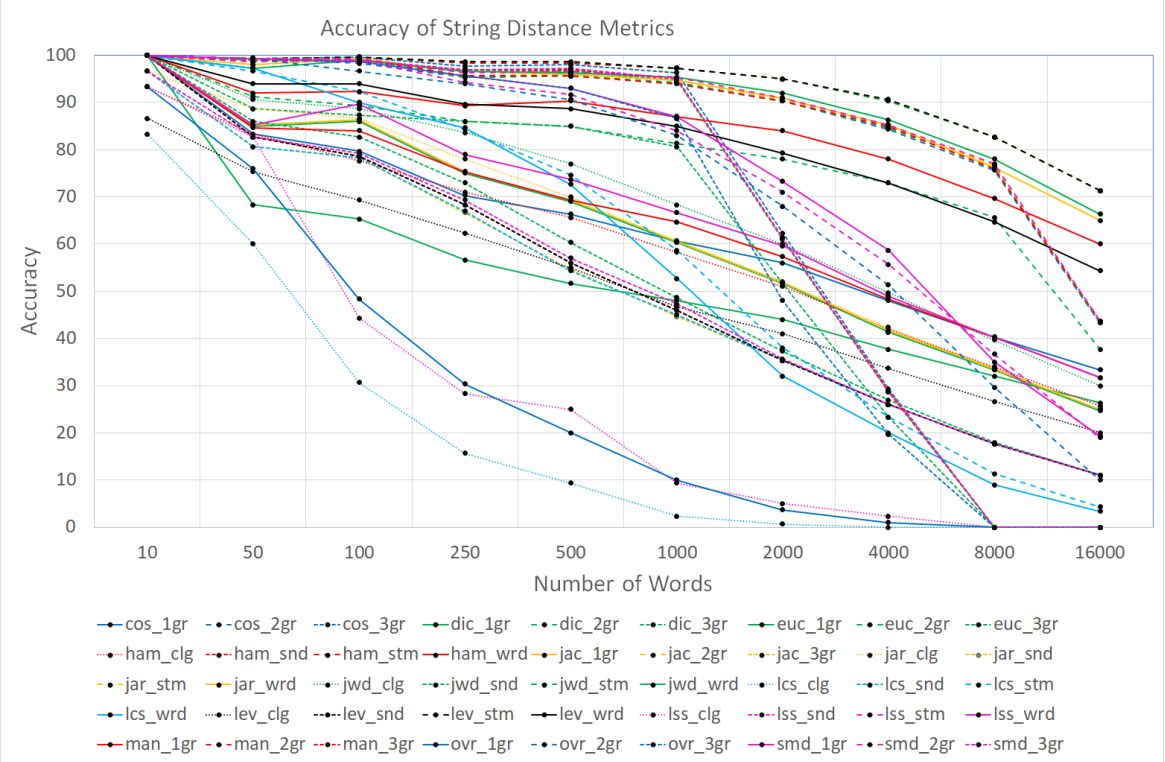
5 Calculation of Preselection Parameters

```
// source: own implementation
void Dictionary::CalculateWordPrecompareParams() {
  //number of words with length
  std::map<int, int> nowwl;

  //      strlen        #chars #correct
  std::map<int, std::map<int, int>> correct;

  for (auto & w : words) {
    ++nowwl[w.word.size()];

    for (int i = 0; i < w.word.size() && i<w.stem.size(); ++i) {
      if (w.word[i] == w.stem[i]) {
        ++correct[w.word.size()][i];
      }
    }
  }


  for (int i = 0; i < nowwl.size(); ++i) {

    for (int j = 0; j < correct[i].size(); ++j) {

      float percentage = 100 * static_cast<float>(correct[i][j]) / nowwl[i];

      if (percentage < 95) {
        word_precompare_params[i] = j;
        break;
      }
    }
  }
}
```

6 Time Measurement

```cpp
// source: own implementation
class Duration {
public:
  void start() {
    t_checkpoints.clear();
    stopped = false;
    t_start = std::chrono::high_resolution_clock::now();
  }

  void checkpoint(std::string comment = "") {
    if (!stopped) {
      t_checkpoints.emplace_back(std::chrono::high_resolution_clock::now());
      comments.emplace_back(comment);
    }
  }

  void stop() {
    if (!stopped) {
      t_stop = std::chrono::high_resolution_clock::now();
    }
    stopped = true;
  }

  double getTotalTime() {
    return std::chrono::duration<double>(t_stop - t_start).count();
  }

private:
  std::chrono::high_resolution_clock::time_point t_start, t_stop;
  std::vector<std::chrono::high_resolution_clock::time_point> t_checkpoints;
  std::vector<std::string> comments;
  bool stopped = true;
};
```

7 Individual Accuracies



Summary of the accuracies of all individual distance metrics. String-based algorithms are depicted in yellow, set-based algorithms in green and vector-based algorithms in blue.

8 Correlation

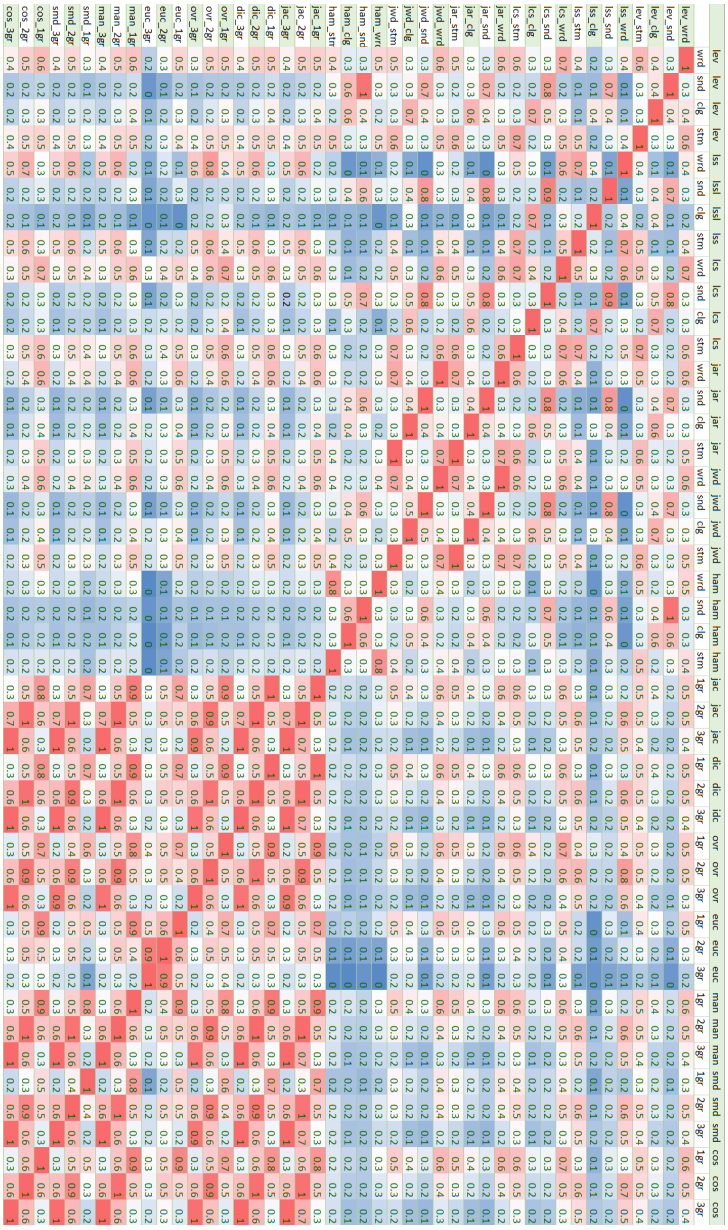


Illustration of Pearsons correlation coefficient for every pair of distance metrics. Low correlation is depicted by the color blue and high correlation with red.

9 Individual Accuracy Test Results

	10	50	100	250	500	1000	2000	4000	8000	16000
lev_clg	86.6667	75.3333	69.3333	62.3333	55	46.6667	41	33.6667	26.6667	20
lev_snd	100	82.6667	78.6667	68.3333	56	46	35.3333	26	17.6667	11
lev_stm	100	99.3333	99.6667	98.6667	98.6667	97.3333	95	90.6667	82.6667	71.3333
lev_wrd	100	94	94	89.6667	88.6667	85	79.3333	73	64.6667	54.3333
lcs_clg	83.3333	60	30.6667	15.6667	9.33333	2.33333	0.666667	0	0	0
lcs_snd	96.6667	80.6667	78.3333	67	54.3333	45	35.3333	26	17.6667	11
lcs_stm	100	96.6667	92.3333	84.3333	74.6667	58.6667	38	23.3333	11.3333	4.3333
lcs_wrd	100	97.3333	90	84.6667	72.6667	52.6667	32	20	9	3.3333
lss_clg	93.3333	82.6667	44.3333	28.3333	25	9.33333	5	2.33333	0	0
lss_snd	96.6667	82.6667	79.3333	69.3333	57	47.3333	35.6667	26	17.6667	11
lss_stm	100	98.6667	99	94.3333	91.6667	84	71	55.6667	36.6667	19
lss_wrd	100	99.3333	98.6667	95.6667	93	87	73.3333	58.6667	35	19.3333
jar_clg	96.6667	88.6667	86.6667	78	70	60.3333	51.6667	42.3333	33.3333	25
jar_snd	96.6667	80.6667	78.3333	66.6667	54.6667	44.6667	35.3333	26	17.6667	11
jar_stm	100	99.3333	99.6667	98.6667	98.6667	97.3333	95	90.3333	82.6667	71.3333
jar_wrd	100	98	99.3333	96.6667	96	94.6667	91	85.3333	76.3333	65
jwd_clg	100	90.6667	88.6667	83.6667	77	68.3333	60	49.6667	39.6667	30
jwd_snd	100	86	82.6667	73	60.3333	48.6667	37.3333	27	18	11
jwd_stm	100	99.3333	99.6667	98.6667	98.6667	97.3333	95	90.3333	82.6667	71.3333
jwd_wrd	100	97.3333	99	96.6667	96.3333	95.3333	92	86.3333	78	66.3333
ham_clg	93.3333	83.3333	77.6667	71	65.6667	58.3333	51	42	34	25.6667
ham_snd	100	82.6667	78.6667	68.3333	56	46	35.3333	26	17.6667	11
ham_stm	100	98.6667	99.6667	98.3333	98.3333	97.3333	95	90.3333	82.6667	71.3333
ham_wrd	100	92	92.3333	89.3333	90.3333	87	84	78	69.6667	60
jac_1gr	100	85.3333	86.3333	75.3333	69.3333	60.6667	52	41.6667	33.6667	25
jac_2gr	100	99.3333	99.6667	95.6667	96	94.6667	90.6667	85	76.6667	43.6667
jac_3gr	100	99.3333	99	97	97	95	61.3333	29	0	0
dic_1gr	100	85	86	75	69	60.333	51.666	41.333	33.333	24.666
dic_2gr	100	99	99.333	95.333	95.666	94.333	90.333	84.666	76.333	43.333
dic_3gr	100	99	98.666	96.666	96.666	94.666	61	28.666	0	0
ovr_1gr	93.3333	76	48.3333	30.3333	20	10	3.66667	1	0	0
ovr_2gr	100	99.3333	96.6667	94	90.6667	83	68	51.3333	29.6667	10
ovr_3gr	100	99.3333	98.6667	95.6667	93	86.6667	48	19.6667	0	0
euc_1gr	100	68.333	65.333	56.666	51.666	48	44	37.666	32	26.333
euc_2gr	100	91.3333	89.3333	86	85	81.3333	78	73	65.6667	37.6667
euc_3gr	100	88.6667	87.3333	86	85	80.6667	51.6667	23.3333	0	0
man_1gr	100	84.6667	84	75.3333	69.3333	64.6667	57.3333	48.3333	40.3333	31.6666
man_2gr	100	99.3333	99.3333	95.6667	95.6667	94	90.3333	85	76	43.6666
man_3gr	100	99.3333	99	97	97	95	61.3333	28.6667	0	0
smd_1gr	100	85.3333	89.6667	79	73.6667	66.6667	59.6667	49	40.3333	31.6667
smd_2gr	100	99.3333	99.3333	96.3333	96.6667	95.3333	91	85.3333	77	43.6667
smd_3gr	100	99.3333	99	97	97.3333	95	61.3333	29	0	0
cos_1gr	100	83.3333	79.6667	70.3333	66.3333	60.6667	56	48	40.3333	33.3333
cos_2gr	100	99.3333	98.3333	95.3333	95.6667	94	90.3333	84.3333	75.6667	43.3333
cos_3gr	100	99.3333	99.6667	97.6667	98	96.3333	62.3333	29.3333	0	0

10 Preselection Parameters

Word size	**word - n**	**cologne phonetics - n**	**soundex - n**
2	-	1	-
3	-	1	-
4	3	0	3
5	3	0	-
6	3	0	-
7	3	2	-
8	0	3	-
9	0	4	-
10	0	4	-
11	0	5	-
12	0	6	-
13	4	8	-
14	6	9	-
15	7	11	-
16	8	13	-
17	9	12	-
18	10	14	-
19	11	14	-
20	12	-	-
21	2	-	-
22	12	-	-
23	0	-	-
24	14	-	-
25	0	-	-
26	20	-	-
27	0	-	-
28	23	-	-
29	24	-	-
30	23	-	-
31	24	-	-

11 Preselection Results

Algorithm and Preselection Method	Accuracy	Time
euc_2gr_nopc	57.587498	1082.216064
euc_2gr_wopc	60.512501	503.865051
euc_2gr_stpc	72.324997	4.054456
euc_2gr_sopc	60.799999	10.76867
euc_2gr_copc	60.087502	430.779846
ham_snd_nopc	11.4625	11.807742
ham_snd_wopc	27.80625	9.341061
ham_snd_stpc	72.324997	4.002839
ham_snd_sopc	12.012501	3.754305
ham_snd_copc	14.974999	8.656034
lev_stm_nopc	72.368752	87.545044
lev_stm_wopc	71.181252	44.329468
lev_stm_stpc	72.324997	3.954727
lev_stm_sopc	64.037498	4.358475
lev_stm_copc	70.924995	38.080471
ovr_1gr_nopc	0.625	441.137115
ovr_1gr_wopc	23.5	221.559845
ovr_1gr_stpc	72.324997	3.942868
ovr_1gr_sopc	22.543751	6.064643
ovr_1gr_copc	10.7375	187.13974

12 Thresholds

Algorithm	Threshold
lev_wrd	0.29
lev_snd	0.24
lev_clg	0.19
lev_stm	0.14
lss_wrd	0.14
lss_snd	0.24
lss_clg	0.16
lss_stm	0.14
lcs_wrd	0.12
lcs_snd	0.24
lcs_clg	0.14
lcs_stm	0.11
jar_wrd	0.12
jar_snd	0.08
jar_clg	0.06
jar_stm	0.06
jwd_wrd	0.07
jwd_snd	0.06
jwd_clg	0.04
jwd_stm	0.03
ham_wrd	0.33
ham_snd	0.24
ham_clg	0.19
ham_stm	0.16
jac_1gr	0.11
jac_2gr	0.41
jac_3gr	0.5
dic_1gr	0.05
dic_2gr	0.26
dic_3gr	0.5
ovr_1gr	0.11
ovr_2gr	0.14
ovr_3gr	0.5
euc_1gr	0.1
euc_2gr	0.15
euc_3gr	0.5
man_1gr	0.13
man_2gr	0.27
man_3gr	0.5
smd_1gr	0.24
smd_2gr	0.43
smd_3gr	0.5
cos_1gr	0.05
cos_2gr	0.23
cos_3gr	0.5

13 Europa Park Testset

Variables in sentences are marked by a leading dash.

Category	**Question**
showtime	wann ist die naechste -show
showtime	wann faengt die -show an
showtime	wann beginnt -show
showtime	wann beginnen die -show
weg	wo ist -poi
weg	wo befindet sich -poi
weg	wie weit ist es zu -poi
weg	wie komm ich zu -poi
weg	wie kommt man -poi
weg	zeig mir den weg zu -poi
weg	wo gehts zu -poi
weg	was ist der schnellste weg zu -poi
weg	zeig mir den schnellsten weg zu -poi
waitingtimes	wie lange ist die wartezeit bei -poi
waitingtimes	wie lang ist die schlange bei -poi
waitingtimes	wie lange steht man bei -poi an
waitingtimes	wieviel ist bei -poi los
shopping	wo kann ich -product kaufen
shopping	wo bekomm ich -product
shopping	ich brauche -product
detail	wie gross muss man fuer -poi sein
detail	wie alt muss man fuer -poi sein
detail	was ist die mindestgroesse fuer -poi
detail	was ist das mindestalter fuer -poi
faq1	sind eintrittskarten im uebernachtungspreis enthalten
faq2	wo erhalte ich die eintrittskarten fuer den europa-park
faq3	ist es guenstiger die eintrittskarten im voraus zu buchen
faq4	was ist der sogenannte vip-eintritt
faq5	darf ich schon vor 15:30 uhr anreisen
faq6	ist es moeglich, das zimmer auch erst am abend nach schliessung des europa-park zu beziehen
faq7	darf ich als hotelgast alle wellness- und poolbereiche der fuenf 4-sterne erlebnishotels nutzen
faq8	ist in den restaurants und bars der europa-park hotels eine tischreservierung erforderlich
faq9	gibt es in den europa-park hotels eine lademoeglichkeit fuer elektrofahrzeuge
faq10	wird es vom hotel kronasar aus eine verbindung zu den anderen erlebnishotels sowie dem europa-park geben
faq11	warum wird es einen expeditionspreis zur sommersaison 2019 geben

14 Europa Park Variables

Type	Values
PoI	haupteingang, silverstar, euromir, poseidon, wildwasserbahn, eisstadion, arena, waffelbaeckerei, taverna mykonos, shoppingpassage, geisterschloss, geisterbahn, schlittenfahrt, dschungelflossfahrt, piraten in batavia, eurosat, abenteuer atlantis, abenteuerspielplatz, african queen, alpenexpress, atlantica supersplash, ballpool, berliner mauer, bluefire, piccolo mondo, crazy taxi, flug des ikarus, elfenfahrt, euro-tower, fjord-rafting, fluch der kassandra, jungfrau-gletscherflieger, koffiekopjes, kolumbusjolle, lada autodrom, london bus, volo da vinci, magic cinema, marionetten-bootsfahrt, matterhorn-blitz, mini-scooter, jim knopf-reise durch lummerland, oldtimer-fahrt, pegasus, fliegender hollaender, raumstation mir, roter baron, russisches dorf, santa marian, schloss balthasar, schweizer bobbahn, silverstonepiste, wiener wellenflieger, the british carousel, vindjammer, schaukelschiff, wasserspielplatz, wichtelhausen, zaubergarten, kaffi hus, colonial food station, crocodilebar, flammkuchenstand, strandgoed, candy-kopje, friethuys, foodloop, casa dei dolci, pizzeria venezia
Show	eisshow, variete show, ritterspiele
Product	pizza, burger, hotdog, getraenke, was zu trinken, was zu essen

List of Figures

Bibliography

[1] Marquis de Condorcet. Essay on the application of analysis to the probability of majority decisions. *Paris: Imprimerie Royale*, 1785.

[2] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[3] Lee R Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[4] Thomas G Dietterich. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15. Springer, 2000.

[5] Ralph Grishman and Beth Sundheim. Message understanding conference-6: A brief history. In *COLING 1996 Volume 1: The 16th International Conference on Computational Linguistics*, volume 1, 1996.

[6] Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[7] Klaus Hechenbichler and Klaus Schliep. Weighted k-nearest-neighbor techniques and ordinal classification. 2004.

[8] Paul Jaccard. The distribution of the flora in the alpine zone. 1. *New phytologist*, 11(2):37–50, 1912.

[9] Matthew A Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406):414–420, 1989.

[10] Thorsten Joachims. A probabilistic analysis of the rocchio algorithm with tfidf for text categorization. Technical report, Carnegie-mellon univ pittsburgh pa dept of computer science, 1996.

[11] Karen Sparck Jones. Index term weighting. *Information storage and retrieval*, 9(11):619–633, 1973.

[12] Eugene F Krause. *Taxicab geometry: An adventure in non-Euclidean geometry*. Courier Corporation, 1986.

[13] Louisa Lam and SY Suen. Application of majority voting to pattern recognition: an analysis of its behavior and performance. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 27(5):553–568, 1997.

[14] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[15] Julie Beth Lovins. Development of a stemming algorithm. *Mech. Translat. & Comp. Linguistics*, 11(1-2):22–31, 1968.

[16] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[17] Behrang Mohit. Named entity recognition. In *Natural language processing of semitic languages*, pages 221–245. Springer, 2014.

[18] Liu Na and ISP Nation. Factors affecting guessing vocabulary in context. *RELC journal*, 16(1):33–42, 1985.

[19] David Nadeau and Satoshi Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30(1):3–26, 2007.

[20] Michael P Perrone and Leon N Cooper. When networks disagree: Ensemble methods for hybrid neural networks. In *How We Learn; How We Remember: Toward an Understanding of Brain and Neural Systems: Selected Papers of Leon N Cooper*, pages 342–358. World Scientific, 1995.

[21] Hans Joachim Postel. Die kölner phonetik. ein verfahren zur identifizierung von personennamen auf der grundlage der gestaltanalyse. *IBM-Nachrichten*, 19:925–931, 1969.

[22] David Rempel. The split keyboard: An ergonomics success story. *Human factors*, 50:385–92, 07 2008.

[23] Joseph John Rocchio. Relevance feedback in information retrieval. *The SMART retrieval system: experiments in automatic document processing*, pages 313–323, 1971.

[24] R Russell and M Odell. Soundex. *US patent*, 1:88, 1918.

[25] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

[26] Leonie Weissweiler and Alexander Fraser. Developing a stemmer for german based on a comparative analysis of publicly available stemmers. In *International Conference of the German Society for Computational Linguistics and Language Technology*, pages 81–94. Springer, 2017.

[27] William E Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. 1990.

[28] Lei Xu, Adam Krzyzak, and Ching Y Suen. Methods of combining multiple classifiers and their applications to handwriting recognition. *IEEE transactions on systems, man, and cybernetics*, 22(3):418–435, 1992.