CHARLES UNIVERSITY / TRENTO UNIVERSITY

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Jonathan Oberländer

Splitting Word Compounds

Supervisor:	RNDr. Pavel Pecina, Ph.D.
Co-Supervisor:	Dr. Roberto Zamparelli
Study programme:	Computer Science / Cognitive Science
Study branch:	Mathematical Linguistics / LMI

2017

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

In ..... date ..... signature of the author

Title: Splitting Word Compounds

Author: Jonathan Oberländer

Supervisors: RNDr. Pavel Pecina, Ph.D., Dr. Roberto Zamparelli

Abstract: Unlike the English language, languages such as German, Dutch, the Scandinavian languages or Greek form compounds not as multi-word expressions, but by combining the parts of the compound into a new word without any orthographical separation. This poses problems for a variety of tasks, such as Statistical Machine Translation or Information Retrieval. Most previous work on the subject of splitting compounds into their parts, or "decompounding" has focused on German. In this work, we create a new, simple, unsupervised system for automatic decompounding for three representative compounding languages: German, Swedish, and Hungarian. A multi-lingual evaluation corpus in the medical domain is created from the EMEA corpus, and annotated with regards to compounding. Finally, several variants of our system are evaluated and compared to previous work.

Keywords: decompounding morphology NLP

I want to thank the following people:
Gergely Morvay and Gero Schmidt-Oberländer (for the annotation of the Hungarian and Swedish corpora), my supervisor Pavel Pecina (for his help and support), co-supervisor Roberto Zamparelli (for solving bureaucratic issues), Bobby Pernice (for everything regarding the LCT program itself), the members of the examination commissions of both universities, my family (for their continuous support), and Laura Bostan.

Contents

1. Introduction

In the English language, when we want to use a specific meaning of a word, we can use prepositions such as "of" or "for": a club *for* sports, a bottle *of* water. Here, the first noun remains the *head* of the entire constituent, and the second one is the *modifier* used for restricting the kind of the first noun (along with other functions, see Chapter 2). In other words, a bottle of water is still a bottle (but not a water).

Alternatively, the words can be combined without prepositions by merely using them sequentially. In this case, the modifier is the first noun, and the second one is the head: a *sports club*, a *water bottle*.

In some languages (called *compounding languages* here) this process of forming *compounds* results in words that (orthographically) look like proper words themselves. There are a few examples of these kinds of words in English, too, but they are lexicalized and largely come from the German influences of English: A *bookshelf* is a shelf for books, but a *bottleshelf* is not a shelf for bottles, it is a non-word; the correct form would at best be *bottle shelf*.

In compounding languages sets of nouns can freely be combined into other nouns. In German, a water bottle is called *Wasserflasche* (water + bottle), a shelf for bottles could be called *Flaschenregal* (bottle + shelf). This process can be repeated recursively, such that a shelf for water bottles could be called a *Wasserflaschenregal*, and so on, leading to famous extreme examples such as *Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz* (roughly *beef labeling supervision task delegation law*, a compound with 7 parts). This results in longer word lengths in compounding languages (see Figure 1.1).

For processing of natural language, this can cause problems: The more complex a compound is, the lower the likelihood of it appearing in corpora, and while some common compounds can be found in dictionaries, a large part of them will not. We cannot dismiss these words as made-up words or names and ignore the problem, because any typical speaker of such a language will effortlessly understand their meaning. They also make up a significant part of language: Schiller [2005] find that in a newspaper corpus, 5.5% of all tokens and 43% of all types were compounds.

Consider, for example, the task of Machine Translation from German to English: Compounds that are encountered need to be translated, and without having ever seen them before, this becomes difficult (see for example Koehn and Knight [2003]). In addition, this is problematic for word alignment, because one word in the source language has to be aligned to two or more words in the target language, or vice versa. This difficulty is illustrated in Figure 1.2.

In Information Retrieval, compounds can also cause problems: A user searching for a rare compound will not get many results, but if the compound can be analyzed and understood, more relevant results (not containing the original compound, but its head and modifier) can be retrieved. This is especially relevant in cross-lingual Information Retrieval.

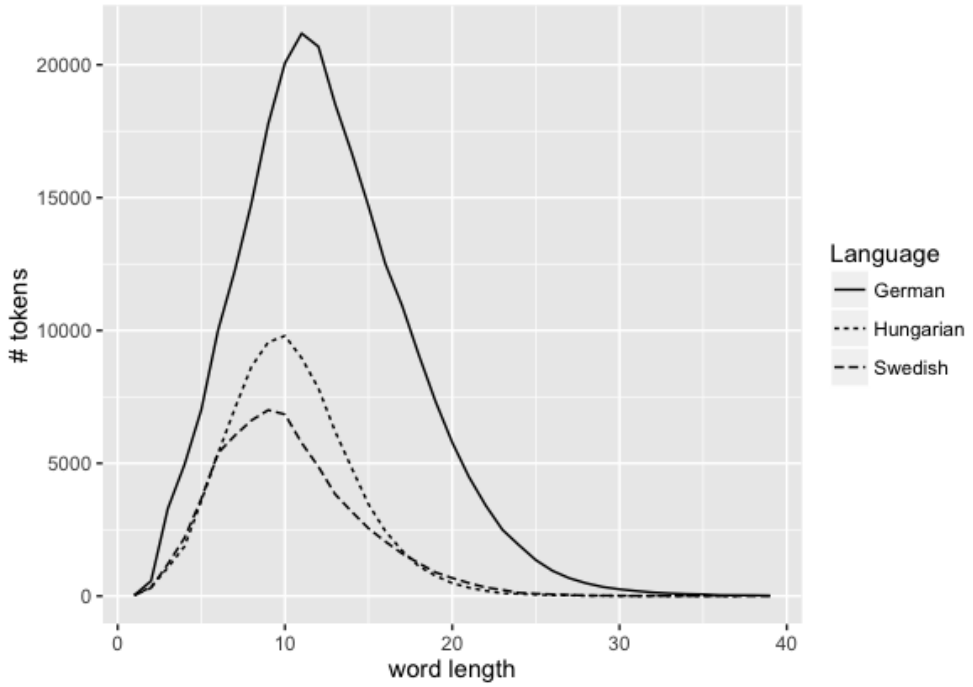

Figure 1.1: Word lengths of tokens in our evaluation corpus (see Chapter *Corpora*. Compared to English, where the most common word length is 7 [Adda-Decker and Adda, 2000], compounding languages have higher modes of 9 (Swedish), 10 (Hungarian) or even 11 (German).

Die$_1$ **Wasserflasche**$_2$ steht$_3$ auf$_4$ dem$_5$ *Holztisch*$_6$.

The$_1$ **Water**$_2$ **bottle**$_3$ is$_4$ on$_5$ the$_6$ *wooden*$_7$ *table*$_8$.

Figure 1.2: Word alignment is tricky between compounding and non-compounding languages: **Wasserflasche**$_2$ has to be aligned with **Water**$_2$ **bottle**$_3$, and *Holztisch*$_6$ with *wooden*$_7$ *table*$_8$.

The solution of these problems of course lies in reversing the process: *decompounding*, splitting a compound into its parts. A system that has not seen *Wasserflasche*, but has seen both *Wasser* and *Flasche*, can, once it knows that *Wasserflasche* indeed consists of those two parts infer that the word must mean "water bottle".

Decompounding is not always as easy as in the previous example. Some compounding languages insert letters between the parts when phonologically required, so called *linking morphemes*: The German word for eye drops is *Augentropfen*, consisting of *Auge* (eye), *tropfen* (drop), and an *n* in the middle.

The process also is not necessarily unambiguous, for (at least) two reasons: morphosyntactic ambiguity and morphological ambiguity.

Compounds that are truely morphologically ambiguous are not that common, but they exist, for example in the German word *Fluchtraum* (shelter; safe room): It can either (correctly) be split into *Flucht* (escape) and *Raum* (room), or (incorrectly) into *Fluch* (curse) and *Traum* (dream). There is no systematic reason

the former is correct, in fact, in the right context, one could even use it with the second way of splitting it meaning something akin to nightmare. The only reason *shelter* is the more "correct" solution is that it is a somewhat commonly used word.

Morphosyntactic ambiguity is more common, but also less of a problem. In any compound of three or more parts, there is a question of hierarchy, with this problem also arising in languages that do not freely compound: Is a *house door key* a *key* for the *house door*, or is it a *door key* of the *house*? The question is as difficult to answer as it is irrelevant for most purposes.

Most previous work has focused on German, since it is the compounding language with the biggest population by far. In this work, we take a look at more compounding languages, and build a system that is largely independent of any specific language. In the process, we also contribute a multi-lingual evaluation corpus for this task.

There are a few goals for our system:

- **Language agnosticism.** While a few languages are built-in, it should be possible to extend it to new languages in a short amount of time.
- **Low resource friendliness.** Not all languages have as much annotated data and NLP tools as German, so the system should work with as little annotated data as possible. Usage of raw text corpora is fine, since they exist wherever such a system may be needed.
- **Usability.** The system should be easy to set up, not require large dependencies, and be quick to run.

The rest of this thesis is organized as follows: In Chapter 2, we look deeper into the process of compounding itself, followed by investigating previous work in the area in Chapter 3. We continue by describing the creation of our corpora in Chapter 4. In Chapter 5 we explain the general algorithm and our improvements, with our specific implementation being described in chapter 6. Its performance is evaluated compared to previous work in Chapter 7, and in Chapter 8 we conclude and discuss possible improvements and related challenges to be addressed in future work.

2. Compounds

Compounding is an extremely frequent process occurring in many languages. It is the process of combining two or more content words to form a novel one. While in many languages (such as English), these are typically expressed as multi-word expressions, in a few languages, such as German, Dutch, Hungarian, Greek, and the Scandinavian languages the resulting words, or *compounds*, are written as a single word without any special characters or whitespace in-between. In this work, these languages are called *compounding languages*, even though compounding in a broader sense also occurs in languages that write compounds as separate words, such as English.

The most frequent use of compounding by far [Baroni et al., 2002] is compounds consisting of two nouns, but adjectives and verbs form compounds as well. Compounds are also formed from any combination of nouns, verbs, and adjectives.

Languages such as English also have a few examples of compounds that are written as a single word, but these cases are lexicalized, i.e. they have become words of their own. New compounds of this kind cannot be arbitrarily created, whereas in compounding languages they can. In compounding languages, there exists some kind of fluid lexicality, as frequently used compounds are fully lexicalized, newly created ones are not lexicalized, while most compounds are somewhere in-between; not lexicalized in the sense that you would find them in a dictionary, but frequent enough, that many speakers will have heard or even used them.

Looking at the German noun compounds as an example, compounding performs different semantic functions on the compound parts, but in all cases the first part is the modifier and the second part is the head: A *Taschenmesser* (pocket knife), consisting of *Tasche* (pocket) and Messer (knife) is a certain kind of knife, not a certain kind of pocket (see Figure 2.1).

In some cases, it is no longer true that the resulting compound is still some kind of its head: A *Geldbeutel* (wallet) consists of *Geld* (money) and *Beutel* (bag), and while one probably would not call a wallet a bag, it is still clear that the bag is the head of the compound and the money the modifier. In many of these cases this has historical reasons: At some point in time, a compound got lexicalized, then it later shifted in meaning. According to Lieber and Stekauer [2009], *endocentric compounds* are compounds where the compound is an instance of the head (e.g. a bookshelf is also a shelf), while *exocentric compounds* are those where they are not (a skinhead is not a head).

The compounding process can be repeated in a recursive way: A *Taschenmesserklinge* (blade of a pocket knife) consists of the compound *Taschenmesser* (pocket knife) and *Klinge* (blade). This way of combining existing compounds could conceivably be repeated, making compounding a recursive process.

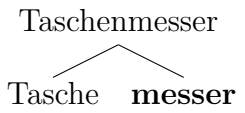

Figure 2.1: Decompounding *Taschenmesser*. The head is marked in bold.

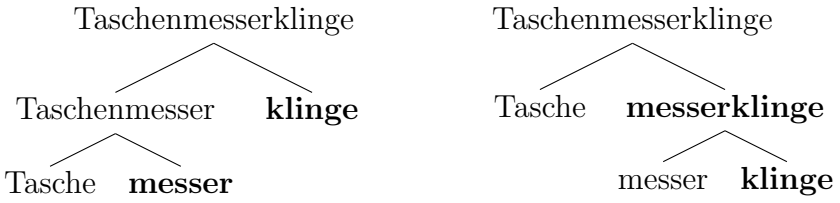

Figure 2.2: Morphosyntactic ambiguity: The correct (left) and incorrect (right) way of splitting *Taschenmesserklinge*.

As soon as three parts are combined to a compound as above, morphosyntactic ambiguity arises: A *Taschenmesserklinge* could also be constructed from *Tasche* (pocket) and *Messerklinge* (knife blade). For an illustration of this effect, see Figure 2.2.

In most cases these ambiguities are either not perceivable by humans, or one reading is strongly preferred, other readings not even considered. This is especially true when a compound part is a lexicalized compound itself, like with *Taschenmesser*.

Morphological ambiguity is more subtle, and occurs when a word has several possibilities to be split into compound parts. The already mentioned *Fluchtraum* is one example, and there are even cases of non-compounds that can be analysed as compounds: *Verbrennen* is a verb meaning to burn something, which can (but should not) be analysed as a compound of *Verb* and *Rennen*, or "verb running".

The compounding process is not always a simple concatenation of the compound parts: In *Taschenmesser*, the *n* is neither part of *Tasche* nor of *messer*, it is a so called *linking morpheme* (German: *Fugenmorphem* or *Fugenelement*)[1]. Sometimes, linking morphemes can be interpreted as endings of an inflected form of the modifying compound part (in this case *Taschen* is the plural of *Tasche*), but oftentimes they cannot.

Any given language can have several linking morphemes, and some of them can even be "negative": A *Schulbuch* (school book) consists of *Schule* (school) and *Buch* (book), but the final *e* of *Schule* is removed when using it as the first part of a compound.

The choice of a linking morpheme depends entirely on the modifier, never on the head [see Baroni et al., 2002].

During composition, more changes can happen to the modifier than just an attached linking morpheme: In some cases, umlauting is happening to a vowel inside the modifier: Combining *Blatt* and *Wald* creates the compound *Blätterwald*. Due to the unpredictability of this process, especially (for the author) in languages other than German, we simply ignore this, as it does not occur very often: In German, less than 0.3% of all compound types experience umlauting, according to Langer [1998].

[1]Langer [1998] criticizes the term, prefering the term "compositional suffix", since it does not have much to do with the head of the compound, and must really be seen as just another suffix, one that is used for composition.

3. Related Work

Wanting to split German compounds, Koehn and Knight [2003] learn splitting rules from both monolingual as well as parallel corpora. They generate all possible splits of a given word, and take the one that maximizes the geometric mean of the word frequencies of its parts, although they find that this process often leads to both oversplitting words into more common parts (e.g. *Freitag* (friday) into *frei* (free) and *tag* (day)), as well as not splitting some words that should be split, because the compound is more frequent than the geometric mean of the frequencies of its parts.

Since their objective is to improve machine translation, they then make use of a parallel corpus: Given a split S of a compound c, if translations of the parts of S occur in the translation of a sentence containing c, the word should probably be broken up: *Freitag* will probably have never beed translated as "free day", so it would not be split. This, of course, can only work for those compounds that actually occur in the parallel corpus, not for completely unseen ones. Furthermore, they constrict parts using a part-of-speech tagger: Words that are not usually parts of compounds (like determiners), are excluded with this method. Their combined method using frequency information, parallel corpus data and POS tag restriction gives them the highest result, with a recall of about 90.1% and a Precision of about 93.8%[1].

Schiller [2005] use a weighted finite-state transducer to split German compounds based on the output of a morphological analyser, which returns some, but not all possible splits. The necessary weights for the finite-state transducer are obtained from the frequencies of compound parts in manually decompounded word lists. This work is also the source for the claim that the split with the lowest number of (unsplittable) parts is the most likely correct one.

Marek [2006] also use weighted finite-state transducers, but the main contribution of this work for us is the creation of an evaluation corpus and an annotation scheme that was created for annotating it. Our own evaluation corpus is annotated based on this annotation scheme.

Alfonseca et al. [2008] approach the task from an Information Retrieval perspective and wanting to handle noisy data in user input, such as misspellings. They also propose that a split is more likely to be the correct split if its compound parts have a positive *Mutual Information*: If they can be composed, that means there must be some semantic relation between them. For instance, in the compound *Blumenstrauß* (flower bouquet), consisting of the parts *Blume* (flower) and *Strauß* (bunch, bouquet, but also: ostrich), the two parts will co-occur with each other more often than would be expected of two random words of their frequency.

Soricut and Och [2015] use vector representations of words to uncover morphological processes in an unsupervised manner. Their method is language-agnostic, and can be applied to rare words or out-of-vocabulary tokens (OOVs).

Morphological transformations (e.g. *rained* = *rain* + *ed*) are learned from the word vectors themselves, by observing that, for instance, $\overrightarrow{rain}$ is to $\overrightarrow{rained}$ as $\overrightarrow{walk}$ is to $\overrightarrow{walked}$. To be able to do that, candidate pairs are first extracted

[1] Precision and recall are defined slightly differently than usual, see Chapter 7 for details.

by looking at whether it is possible to replace, add, or remove a suffix or prefix of one word to get to the other word. Given that candidate list, they then use word embeddings to judge the quality of candidate rules by trying to find several analogy pairs with the same suffix/prefix in similar relations in the vector space. This filters out spurious suffixes like *-ly* in *only*, or prefixes like *s-* in *scream*.

Daiber et al. [2015] apply the approach of Soricut and Och [2015] to decompounding. Using word embeddings of compound and head, they learn prototypical vectors representing the modifier. During the splitting process, the ranking occurs based on comparing analogies of the proposed heads with the entire compound.

They achieve good results on a gold standard, and improved translation quality on a standard Phrase-Based Machine Translation setup.

Bretschneider and Zillner [2015] develop a splitting approach relying on a semantic ontology of the medical domain. They first try to disambiguate candidate splits using semantic relations from the ontology (e.g. *Beckenbodenmuskel* (pelvic floor muscle) is split into *Beckenboden* and *muskel* using the *part-of* relation). If that fails, they back off to using a frequency-based strategy.

Erbs et al. [2015] compare the performance of recent work in decompounding using the evaluation corpus from Marek [2006], looking at the correctness scores of the compared systems. They find that the **ASV toolbox** [Biemann et al., 2008] delivers the best results. They then also investigate what benefit decompounding has on the results of a specific task, *keyphrase extraction*.

4. Corpora

This work made use of several corpora, and produced a new evaluation corpus for evaluation of decompounding systems in German, Swedish, and Hungarian.

4.1 EMEA corpus

The EMEA corpus [Tiedemann, 2009] is a parallel corpus based on documents by the European Medicines Agency[1]. It consists of parallel texts for each possible pairing of the numerous languages it contains, including English and all the languages we are investigating (German, Swedish, and Hungarian). For English and Swedish, the texts are also available as parsed corpus files, but the parsed files were not used by us.

The corpus was chosen as the basis of our evaluation corpus because of our decision to evaluate in the medical domain.

We also use it as the raw text corpus for our word embeddings, since its size is approximately the same for all three languages, meaning we can later exclude different training data size as a cause for differences in the results.

4.2 Wikipedia as a raw source

Our system requires some kind of lexicon to serve as a repository of known words. A good source for arbitrary raw text is Wikipedia, because it is relatively big — big enough to contain all kinds of words, in many kinds of inflections. As an encyclopedia, the language use is not necessarily representative, but it was still chosen because in contrast to some other more balanced corpora, it is available for most languages in common use, such as the ones we are investigating here.

The corpus was created from publicly available Wikipedia dumps from the Wikimedia download page[2]. The specific dump identifiers are listed in Table 4.1.

Language	Precise Wikipedia dump
German	`dewiki-20170201-pages-articles-multistream.xml.bz2`
Swedish	`svwiki-20170201-pages-articles.xml.bz2`
Hungarian	`huwiki-20170201-pages-articles.xml.bz2`

Table 4.1: List of Wikipedia dumps used. All dumps were created (by Wikimedia) on February 1st, 2017.

The downloaded XML files are large in size (about 50 MB for Swedish and Hungarian, about 100 MB for German). They contain — apart from XML tags themselves — not raw text, but text written in a markup language called Wikitext[3] (more precisely: The Wikipedia flavor of Wikitext). Wikitext is a relatively concise language similar in spirit (but not in concrete looks) to Markdown[4].

[1] `http://www.ema.europa.eu/ema/`

[2] `https://dumps.wikimedia.org/backup-index.html`

[3] `https://en.wikipedia.org/wiki/Wiki_markup`

[4] `https://daringfireball.net/projects/markdown/syntax`

```
'''Alan Smithee''' steht als [[Pseudonym]] für einen fiktiven Regisseur,
der Filme verantwortet, bei denen der eigentliche [[Regisseur]] seinen
Namen nicht mit dem Werk in Verbindung gebracht haben möchte. Von 1968
bis 2000 wurde es von der [[Directors Guild of America]] (DGA) für solche
Situationen empfohlen, seither ist es '''Thomas Lee'''.&lt;ref&gt;[[Los
Angeles Times|latimes]].com: [http://articles.latimes.com/2000/jan/15/
entertainment/ca-54271 ''Name of Director Smithee Isn't What It Used to
Be''], zuletzt geprüft am 2. April 2011&lt;/ref&gt; ''Alan Smithee'' ist
jedoch weiterhin in Gebrauch.
Alternative Schreibweisen sind unter anderem die Ursprungsvariante
''Al&lt;u&gt;len&lt;/u&gt; Smithee'' sowie ''Alan Sm&lt;u&gt;y&lt;/u&gt;
thee'' und ''A&lt;u&gt;dam&lt;/u&gt; Smithee''. Auch zwei teilweise
asiatisch anmutende Schreibweisen ''Alan Smi Thee'' und ''Sumishii Aran''
gehören { so die [[Internet Movie Database]] { dazu.&lt;ref name=&quot;
IMDb&quot;&gt;[http://www.imdb.com/name/nm0000647/ Eigener Eintrag für
''Alan Smithee'' in der IMDb]&lt;/ref&gt;

== Geschichte ==
=== Entstehung ===
Das Pseudonym entstand 1968 infolge der Arbeiten am Western-Film ''Death
of a Gunfighter'' (deutscher Titel ''[[Frank Patch - Deine Stunden sind
gezählt]]''). Regisseur [[Robert Totten]] und Hauptdarsteller [[Richard
Widmark]] gerieten in einen Streit, woraufhin [[Don Siegel]] als neuer
Regisseur eingesetzt wurde.
```

Figure 4.1: The first lines of the contents of the <text> tag of the first article of the German wikipedia dump. Note the large amount of noise.

Wikitext documents can contain bold and italic text, images, (internal and external) links, tables, references to templates and categories, and, depending on the setup of the particular wiki, a subset of HTML, which is (because it is embedded in XML) encoded with XML character entity references[5]. For illustration purposes, the first lines of the German wikipedia dump (inside the <text> tag only) can be seen in Figure 4.1.

To process this text, a Python script was written that, given a stream of text of this kind, extracts word counts. The script is included in the attachment as `utils/counts_from_wikipedia.py`, and expects raw text from standard input, meaning it is called with `bzcat` *`wikipedia-dump`*`.bz2 | python3 counts_from_wikipedia.py`.

The method of this script is now described in detail.

Due to the expectable format of the XML file, no XML parser is used: No simple XML parser would have worked, since having the entire file in memory might have posed problems. A streaming XML parser could have worked, but instead, we use a simple line-based state model:

The script starts in the *outside* state, and stays in that state until it sees a line that contains the starting <text> tag. It then switches to the *inside* state and processes each line (including that first one, with the XML tag removed),

[5]For instance, > is encoded as `>`

until it finds a line that contains the closing </text> tag, at which point it will revert to the *outside* state.

The processing of a line starts with using the `unescape` function from Python's `html` package to turn XML character entity references back into the original characters (for example `>` into >). Next, it removes triple and double single quote characters (''' and ''), which are used for bold and italic text in the Wikitext language.

A line is skipped if it starts with any of the following:

- a template reference ({{ or }})
- part of a table (starts with | or !)
- a list (starts with *)
- a heading (starts with ==)
- a comment (<!-- and -->)
- an indentation (starts with :)
- a category reference (starts with e.g. [[Kategorie:)

As a next step, we try to replace links with their labels using a series of regular expressions:

- **Internal links:** Anything matching `\[\[(?:[^\]]*?\|)?([^|\]]*?)\]\]` gets replaced with the contents of the first capture group. This will replace `[[lemma|label]]` with `label` and `[[label and lemma]]` with `label and lemma`.
- **External links:** Anything matching `\[\S+ ([^\]]*?)\]` will be replaced with the first capture group, replacing `[http://example.com label]` with `label`.
- **HTML/XML elements:** Anything matching `<[^>]*?>.*</[^>]*?>` is deleted, removing opening and closing HTML and XML tags (for example and).
- **Template links:** Anything matching {{[^}]+?}} is removed, purging template references like {{Siehe auch|Geschichte Deutschlands}}.
- **HTML/XML standalone elements:** Anything matching `<[^>/]*?/>` is removed, getting rid of standalone XML tags (for example <hr />).

After having done all these replacements on a line, all matches of the regular expression `[A-Za-zäöüÄÖÜßÅåÁáÉéÍíÓóŐőÚúŰű]+`[6] are extracted and added to a word counter, after being transformed to all lowercase letters.

[6]This is admittedly an unelegant expression, but there is currently no better way to do this within the standard library of Python. The `regex` module (`https://pypi.python.org/pypi/regex/`), which is supposed to eventually replace Python's builtin `re` module could use the pattern `\p{L}+` instead to refer to all sequences of all letters in all languages.

	Types	Tokens	$\frac{\text{Types}}{\text{Tokens}}$ (%)
German	7 089 242	748 064 839	0.95
Swedish	3 319 114	397 902 042	0.83
Hungarian	3 008 758	112 148 271	2.68

Table 4.2: Corpus statistics for the Wikipedia corpus

The resulting data structure is printed to a file, one word per line, with a tab character separating the word and its count across the corpus, sorted by the count, such that the most frequent words are in the beginning of the file. This results in the lexicon files, such as `lex/de.lexicon.tsv`.

Corpus statistics about the resulting corpora can be seen in Table 4.2. The German part is obviously the biggest one (the German Wikipedia is the second-biggest Wikipedia, after the original English one), but what is interesting is that Hungarian has by far the highest type to token ratio. This is due to the more complex morphology of Hungarian, compared to the other languages.

4.3 Marek [2006]

The work of Marek [2006] includes an evaluation corpus that was created based on articles from the German computer magazine *c't — Magazin für Computertechnik*. The corpus itself is not used, but our own evaluation corpus annotation format is based on it. We drop the annotation of part-of-speech of the compound parts, and concentrate only on where a compound should be split. The annotation scheme is described in more detail below.

4.4 Stopwords and affixes

In order for our simple heuristics to filter out obviously wrong splits, we also employ the use of two additional corpora per language: A list of stopwords and a list of suffixes.

The stop word lists[7] contain 232 German words, 199 Hungarian words, and 114 Swedish words.

Because our candidate extraction algorithm goes from left to right (see Chapter 5), it might happen that a word or subword is split into root and suffix, if the root is a known word. Since this is just inflection, not compounding, we want to filter out such cases. For this purpose, we extract a list of suffixes for each language from Wiktionary (a sister project of Wikipedia):

We simply take all page titles in the `Category:`*`language`*`_prefixes` and `Category:`*`language`*`_suffixes`[8] and remove the dash at the beginning of each page title. The resulting suffix list contains 115 suffixes in German, 85 suffixes in Swedish, and 545 suffixes in Hungarian. The prefix lists contain 116 prefixes in German, 48 prefixes in Swedish, and 100 prefixes in Hungarian.

[7] `https://github.com/Alir3z4/stop-words`

[8] For example `https://en.wiktionary.org/wiki/Category:German_suffixes`

Original token	Annotated token
anwendung	anwendung
beachten	beachten
weitere	weitere
warnhinweise	warn+hinweis(e)
vorsichtsmaßnahmen	vorsicht|s+maßnahme(n)
cholestagel	cholestagel
angewendet	angewendet
einnahme	einnahme
bondenza	bondenza
anderen	anderen
arzneimitteln	arznei+mittel(n)
informieren	informieren
apotheker	apotheker
arzneimittel	arznei+mittel
einnehmen	einnehmen
anwenden	anwenden
eingenommen	eingenommen
verschreibungspflichtige	verschreibung|s+pflichtig(e)
handelt	handelt
lietuva	lietuva

Table 4.3: The first lines of the German part of the evaluation corpus

4.5 Evaluation Corpus

The creation of the evaluation corpus started with the EMEA corpus, which was first shuffled on the sentence level to ensure that the selection of words was not biased. Care was taken however, to shuffle the corpus in exactly the same way for each language, so that randomness did not introduce a difference between the languages.

Next, the words were filtered based on their length. Since compounds consist of at least two parts, they have a certain minimum length. Any word shorter than seven characters was therefore discarded.

This number is a somewhat arbitrary number, but is based on the observation that shorter words are less likely to be compounds, and if they are, they are more likely to be lexicalized already. Had we chosen a lower number, the workload of the annotators would have increased, too. Increasing the number to a much higher one would make the task easier, since it is self-evident that the longer a word is, the more likely it is to be a compound.

This list of words was then given to annotators, with the German part being annotated by the author himself. The annotators were given the task to split the words into compound parts using a plus character, when necessary. Non-compounds were to be left untouched.

Word endings of compounds were put in parentheses, linking morphemes split off with a pipe character (|). As in the original annotation scheme in Marek [2006], ablaut and umlaut changes are annotated using capital letters.

As an example, the first lines of the annotated German corpus can be seen in

	German	Swedish	Hungarian
Words	958	565	3361
# Compounds	193	162	512
% Compounds	20.0%	28.7%	15.2%
% 2 parts	91%	93%	93%
% 3 parts	9%	7%	6%
% >3 parts	<1%	0%	1%

Table 4.4: Corpus statistics of the evaluation corpora

Table 4.3.

When annotating anything there is a choice to be made between accuracy of the annotation process, and the annotation feeling "natural" for native speakers.

A special case of this is how to handle fully lexicalized compounds: They are not a problem for NLP, since they are frequent enough to be lexicalized. In fact, splitting them could even worsen results in some cases, where the current meaning of a compound is not directly related to the meaning of its parts anymore:

If a Machine Translation system wants to translate *Geldbeutel*, but translates the decompounded parts instead, the translation might be "money bag" instead of the correct "wallet".

In our case, a decision was made to trust the annotators and let them annotate as compounds what feels like a compound to them.

The German and Swedish parts of the corpus were each annotated by a single person, whereas 10% of the Hungarian corpus was additionally annotated by a second annotator to calculate inter-annotator agreement. Cohen's Kappa was determined to be at $\kappa = 0.95$.

The other part of data required for each language was the list of linking morphemes. The German list is based on Alfonseca et al. [2008], Hungarian and Swedish linking morphemes were extracted from the annotated corpora. They are displayed in Table 4.5.

Corpus statistics about the evaluation corpora can be seen in Table 4.4. Most compounds in the evaluation corpora have only two parts, there are very few examples of compounds of three or more parts.

The evaluation corpus in all three languages is included in the Attachment in the folder `gold_corpus/`.

Language	Linking morpheme	Example
German	s	Hemd**s**ärmel
	e	Hund**e**hütte
	en	Strahl**en**therapie
	nen	Lehrerin**nen**ausbildung
	ens	Herz**ens**wunsch
	es	Haar**es**breite
	ns	Wille**ns**bildung
	er	Schild**er**wald
Swedish	s	utgång**s**datum
Hungarian	ó	old**ó**szer
	ő	gyűjt**ő**doboz
	ba	forgalom**ba**hozatali
	ítő	édes**ítő**szerként
	es	kék**es**barna
	s	szürké**s**barna
	i	ízület**i**fájdalom
	a	kor**a**szülött

Table 4.5: Linking morphemes in German, Swedish, and Hungarian

5. Algorithm

This chapter describes the basic algorithm decompounding is based on and our improvements and experiments upon it. As input, the algorithm takes a word as a string of characters of non-zero length. The output is one of the possible *splits* of the word.

A *split* of a word is any unique division of a word into parts. A split of a word of length n has at least 1 and at most n parts. Since every character of a word can either belong to the previous part or form a new part, any word of length n has 2^{n-1} different splits (see Figure 5.1).

foo	→	foo	(1 part)
		fo+o	(2 parts)
		f+oo	(2 parts)
		f+o+o	(3 parts)

Figure 5.1: The word *foo* has three characters, therefore it has $2^{3-1} = 4$ splits. Splits of *foo* have at least one and at most three parts. Parts are separated by plus signs.

The generation of candidate splits (see Section 5.3) is recursive and based on *binary splits*: i.e. a split with up to two parts.We define a word as a sequence of characters, since a part is also a (sub)sequence of characters, we can also apply splitting on individual parts.

While splits of a word have the theoretical bound of 2^{n-1}, in practice, the number of somewhat sensible splits (as licensed by a given lexicon) is much lower. We will use the term *possible split* to refer to splits where all parts (except for the last one) are known words (from a lexicon) or linking morphemes.

5.1 General idea

The algorithm has three main steps:

1. **Candidate generation:** From a given word, a lexicon, and a list of linking morphemes, a list of *possible splits* is generated. This is done in a greedy way that is described in detail in Section 5.3.

2. **Cleaning:** The list of possible splits is changed, either by filtering (i.e. removing) entries, or by modifying it in some way (for example to join parts into one under some conditions). This is described in Section 5.4.

3. **Ranking:** The cleaned list of candidate splits is then ranked using one or more methods. When multiple ranking methods are used, the second one will make a decision only when the first one ranks two splits as equally good, more on that in Section 5.5.

The last step returns one chosen split. If that split has only one part, the "verdict" is that the word does not need to be decompounded.

5.2 Initialization

Based on the language, the list of linking morphemes is set. This list is adapted from Alfonseca et al. [2008] for German, and based on input from the annotators for the other supported languages. The linking morphemes are listed in Table 4.5.

In addition to the linking morphemes, a lexicon of known words is read in. As a source for the lexicon, a simple Wikipedia dumpis used (see Section 4.2). The reason for opting for a raw text corpus instead of a real lexicon is that we will find not only base forms, but also most inflected forms in it, and will not have to worry about morphology too much. Since it is our goal to produce a system that is adaptable to any language where it is needed, we want to use as little training data as possible, in fact, the only piece of knowledge we do not obtain from a raw corpus is the list of linking morphemes. This obviously does not include the evaluation corpus, which is not needed for the system itself.

5.3 Generation of candidate splits

After this initialization, words can be decompounded by the system. First, a sequence of possible binary splits (i.e. splits that have two parts) of a given word is obtained. This includes the "split" where the point of splitting is after the word, meaning nothing is split at all. Starting with the smallest possible split from left to right, we check whether *either*

- the left part of the split is a known word
- the left part of the split is a linking morpheme

If neither of those conditions are met, the process continues. Otherwise, this process is repeated recursively for the remaining part.

This method has several implications:

- **The right-most part of the proposed split need not be a known word.** This helps alleviate the need for true morphological analysis as endings of the head do not matter so much.
- **The process succeeds for any given word.** It may however return a split which contains only a single word, i.e. is not a split at all.
- **The process often returns several possible ways of splitting.** Afterwards, a decision needs to be made which of the splits to keep.

5.4 Cleaning the splits

As a next step, initial cleaning of the candidate splits is being done. These cleaning methods are intended as simple heuristics that disregard obviously wrong candidates, or change candidates when it appears to be necessary. This is to make it easier for the ranking methods later on, as the list of possible splits is reduced, but there is often still a choice to be made. We use five different cleaning methods.

5.4.1 general

- Linking morphemes cannot be at the beginning of a word. If the first part of a split is a linking morpheme, it is merged with the second part.
- Given two consecutive parts, if the right one is not a known word, the left one is a linking morpheme, and the concatenation of the two parts is a known word, they are merged together. This does not need to be checked the other way around, because of the left part *always* being either a known word or a linking morpheme, based on the previous steps.

5.4.2 last_parts

The motivation for this cleaning method comes from the assumption that base forms of words are more likely to be known than inflected ones. As we assume suffix-based morphology, this can lead to suffixes being split off.

Therefore, we merge the last two parts whenever the last part is shorter than 4 characters. This number is somewhat arbitrary, and could be reduced or increased, with reducing it leading to a smaller effect of the cleaning method, and increasing it leading to a higher number of falsely discarded candidate splits.

5.4.3 suffix

As a more informed variant of `last_parts`, we also use our list of suffixes (see Section 4.4) to further filter out incorrectly separated suffixes.

If the last part starts with a known suffix, and is not much larger than it[1], we merge the last two parts. We do not simply check for the last part being identical with a suffix, because we observed (in the German suffix list) that many suffixes are also only listed in their base form (for example, *-iv*, the equivalent suffix to the English *-ive* is found in the German suffix list, but not *-ive*, *-iver*, *-ives*, and so on). The additional check for similar length is a protective one, because some words may also start with the same characters as some suffixes.

From manual inspection of the affix lists, we estimate that this has the potential to remove legitimate splits: For example, the German suffix list contains *-zentrisch* ("centrical"), which could conceivably occur as the head of a compound, but it is still our assumption that this filtering will improve more than it will hurt recall.

5.4.4 prefix

In a similar way, we also use the list of prefixes. Since inflection does not seem to happen in prefixes in the languages we investigate, we are stricter, and require a part to be identical with a prefix. We do not just look at the first part, because the prefix might be attached to the second (or third, ...) part of a compound.

If any part of a proposed split is a prefix, we disregard the split.

[1] Precisely: If the last part p starts with a known suffix s, and $|p| \leq |s| + 2$.

5.4.5 `fragments`

Finally, this cleaning method tries to get rid of other obviously incorrect splits by disregarding any splits that contain parts which:

- Consist of only one or two characters
- And are not in the list of linking morphemes

5.5 Ranking the splits

Next, we use one or more ranking methods to re-rank the list of candidate splits, and return the top-ranked split.

When combining several ranking methods, the second one will decide the cases that the first one could not decide, and so on. Our best methods (see Chapter 7) are such combined methods.

The various ranking methods are described below:

5.5.1 `most_known`

This ranking methods assigns a score to each split based on the fraction of known words in its parts. For example, it would prefer *master+arbeit* (two out of two parts are known words) over *mast+erarbeit* (one out of two parts are known words).

Since a split may contain linking morphemes, we only take parts into consideration that are not found in the list of linking morphemes here.

5.5.2 `shortest` and `longest`

`shortest` assigns a score to splits based on the number of parts, preferring splits with fewer parts, following Rackow et al. [1992].

Just for completeness, we also have a method called `longest` that does the opposite of `shortest`; it prefers long splits. This is a special-case ranking method that does not make sense in a normal setting, but could have a use in recreational linguistics (to find alternative compound interpretations of long words).

5.5.3 `beginning_frequency` and `avg_frequency`

Frequency information has been shown to be of use in previous work [Koehn and Knight, 2003, Baroni et al., 2002]. Since we are working with a raw text corpus as a lexicon, it is available to us, too.

Apart from using frequencies for filtering words (see Section 5.6.2), we also rank using them:

The ranking method `avg_frequency` takes the arithmetic mean of the frequencies of the parts in our lexicon. To not distort our results, we disregard parts that are linking morphemes.

We also implement a variant called `beginning_frequency` that is for the most part identical with `avg_frequency`, but uses only the first six characters of the part for lookup. For this purpose, a separate lexicon of beginnings is constructed while the normal lexicon is loaded.

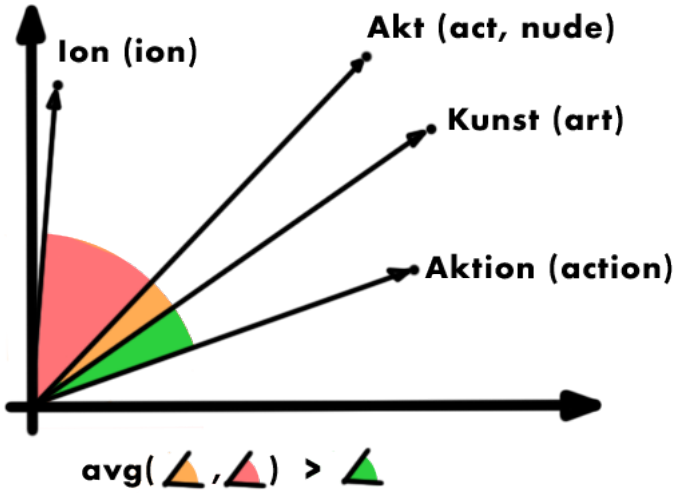

Figure 5.2: A (hypothetical) 2D projection of a vector space. The splits of "Kunstaktion" can be rated according to semantic vectors of their respective parts.

5.5.4 `semantic_similarity`

A relatively recent trend in Natural Language Processing is the use of semantic vectors that represent the meaning of words or other textual entities. Such vectors have been shown to be successful in tasks such as parsing [Socher et al., 2013] or sentiment analysis [Maas et al., 2011].

Semantic vector spaces are constructed in such a way that vectors of semantically similar words are close to each other, and vectors of dissimilar words further apart. One can then query such a space to compute the similarity between two words ("How similar is *book* and *newspaper*?), or to retrieve a list of n closest vectors to a given word.

If such a vector space is available, we could score a given split of a word by looking at the distances of its parts. For example, given the German word "Kunstaktion" (roughly *art performance*) could either be split into the parts "Kunst" (*art*) and "Aktion" (*action*), or, incorrectly, into "Kunst" (*art*), "Akt" (*act*), and "Ion" (*ion*). As shown in Figure 5.2, we assume that the vectors for "Kunst" and "Aktion" would be closer to eachother on average than "Kunst", "Akt", and "Ion".

A fast, memory-efficient way to obtain a semantic vector space is a family of methods called **word2vec** [Mikolov et al., 2013], of which we use the continuous skip-gram variety: Word and context vectors are initialized randomly, then, iterating over all words in a given corpus, the model tries to predict context words from a given word. In the process, word vectors are brought closer to the context vectors of their context words, and, using *negative sampling*, distanced from random "noise" contexts. We use the default size of 100 dimensions and a window size of 5, i.e. the distance between the current and the predicted word can be at most five words.

Distance between two vectors is measured in terms of *cosine similarity*; the cosine of the angle between the vectors. Using cosine similarity is equivalent to using a euclidean distance between unit vectors, and results in values between 0 and 1, inclusive[2]:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

As word embeddings require only raw data, we also investigate its use in our scenario. The idea is to look at the (proposed) parts of a split, and see if adjacent parts are at least a little bit similar to each other, and then returning the split with the highest average similarity. We choose the average of the similarities because a product would massively favour short splits, and a sum would favour longer splits. Comparing only adjacent parts makes sense, because we assume only they share some meaning: In *Taschenmesserklinge* (pocket knife blade), *Tasche* (pocket) and *Messer* (knife) are often seen together, even more so with *Messer* and *Klinge* (blade), but *Tasche* and *Klinge* are likely not to be similar at all.

One of the proposed splits is always the hypothesis that the given word is not a compound, we also need to assign a value to this case. A default value of 1 would make it such that this ranking method never splits, making it equivalent with the baseline described in Chapter *Evaluation*, we therefore assign a value of 0, which happens to force it to always split, if any other candidate has at least one part of pairs for which the vectors return a non-zero similarity.

Typically, lemmatization or stemming is used as a preprocessing step for creating vector spaces. On any given language, we will not necessarily have tools for that available, which is why we use a very primitive stemming method for all languages: From all words, we strip away any characters after the 6th one. We assume that this will be enough to uniquely identify most words, but also to strip away any possible suffixes. Using this process, *goat* remains as is, but *hamster* is transformed to *hamste*.

We use the **gensim** implementation [Řehůřek and Sojka, 2010] of the skip-gram model. As the `gensim` package has quite a few bigger dependencies[3], we save the vector spaces as pickled objects[4], which means that `gensim` will only be loaded when the respective ranking method is used.

We use the raw EMEA corpus for training the embeddings, throwing away any tokens that contain non-alphabetic characters. The corpus contains about 23 million tokens in every language.

[2]Cosine distance results in values between -1 and 1, but if all vector components are always positive, the resulting number will instead be at least 0.

[3]`smart-open`, `six`, `scipy`, and `numpy`

[4]`https://docs.python.org/3/library/pickle.html`

5.6 Improvements

In addition to the above described, we experiment with other parameters of the algorithm.

5.6.1 Stop words

Since compound parts are typically content words, we want to limit the available words the system sees as valid. Since we are building on raw data, part-of-speech tags are not available. We therefore use stop word lists: After the lexicon is read, any words present in the list of stop words is removed from it again.

This prevents splits into pseudo-compounds which supposedly begin with a closed-class word like a determiner. For example, *Dermatologe* (dermatologist) is not a compound, but without excluding stop words from the list of known words, it might be split into *Der* (nominative masculine singular "the") and *matologe* (spurious).

5.6.2 Frequency minimum

Any real-world corpus contains some amount of noise, especially web-based corpora. No matter how well our scraping methods work, it is assumed that some amount of noise will always make it into the data, in our case the lexicon. In order to minimize spurious words, we therefore set a (low) limit of minimum frequency for each word, meaning that we discard any word that occurs less than n times for some n, in our case 2.

Because we impose a limit on the lexicon size, this does not have an effect in our case, but with a smaller lexicon, or a higher word limit this might prove beneficial.

5.6.3 Forcing decomposition

Finally, we experiment with whether it makes sense to insist on splitting words, whenever possible: This setting causes the ranking system to always produce a split with more than one part (i.e. the claim that any word is a compound).

We do not expect this to do better than the other methods, but assume it might be useful when it is already known that a word must be a compound.

6. Implementation

This chapter describes the implementation of the algorithm described in Chapter 5. It consists of a script written in Python 3, and is not backwards compatible with Python 2, because it makes heavy use of features missing in Python 2 (and not importable from the `__future__` module).

The script reads a file containing one word per line, and prints a decompounded (if necessary) version of every word. Alternatively, the `Splitter` class can be imported from another module.

The source code, along with all the lexical resources required to get it to run, is included as an Attachment. Except for the lexica, which are hosted separately, the contents of the attachment are also published on Github[1]. This will also be the place for further development.

6.1 Setup

As our system is written in the programming language Python 3, a recent version of a Python interpreter (3.5 or higher) needs to be installed first. While this was developed and tested with Python's standard implementation `CPython`, it should also work with up-to-date `IPython` [Pérez and Granger, 2007]. Python interpreters that do not support C extensions (e.g. `IronPython`[2]) or are not up-to-date (for instance, at the time of writing, `PyPy3`[3] implements Python 3.3) are not supported.

Apart from Python itself, the system has one hard and one soft dependency:

- `docopt`[4] is a command line argument parser that enables its users to write the standard usage/help message and automatically parse arguments following the schema described. This module is used by our script to simplify the creation of its command line interface. It is required to be installed in order to use the decompounding system.

- The `gensim` package was used for training and handling the vector space. If the word embedding method is being used, `gensim` and its requirements need to be installed too. If this method is not used, it does not need to be installed, the script will not try to import it in that case.

Both of these packages can be installed with

```
$ pip3 install -r requirements.txt
```

using the requirements file included in the distribution. In that case, the exact versions used during development will be installed.

[1] `https://github.com/L3viathan/compound-splitter`
[2] `https://github.com/IronLanguages/ironpython3`
[3] `https://pypy.org`
[4] `https://github.com/docopt/docopt`

6.2 Command-line interface

The system can be used in two different ways: As an imported module, or using its command-line interface. The latter is described in this part.

The script takes one required argument, which is the name of a file which contains the candidates to decompound. This file should contain a single word per line. When the decomposition should happen on demand, the file name can be replaced with `-` to instead read from standard input.

In addition, the script can take a number of options:

- A two-letter language[5] code can be specified using the `--lang` option. This defaults to `--lang=de`, German.
- If the list of stop words should not be used, one can pass `--no-stopwords`.
- If the system should try to split words whenever possible, one can pass `--force-split` (or `-f`).
- To specify a minimum word frequency under which words should be discarded from inclusion in the lexicon, the option `--min-freq=...` can be used. By default, a value of 2 is used, meaning that any word occurring less than twice will be discarded.
- An alternative way to restrict the amount of words the lexicon will consist of can be achieved by passing the option `--limit=<n>`. If this option is provided, only the top n lines of the lexicon file are used, the rest is discarded. By default, only up to 125,000 words are read in.
- For debugging purposes, the output can be made more verbose by giving the option `-v`. This option can be provided several times, each time increases the verbosity level (up to a maximum of 3).
- To customize the cleaning methods that should be used, a comma-seperated list of methods can be specified with the `--cleaning=...` switch. The available values by default are:
  - `general`
  - `last_parts`
  - `suffix`
  - `prefix`
  - `fragments`

  Available methods are all methods of the `Splitter` class that start with `clean_`, more on that in Section 6.4. If this option is not specified, the system will perform cleaning as it would if it had been called with the option `--cleaning=general,last_parts,suffix,prefix`. All cleaning methods are described in Section 5.4.

[5] ISO 639-1; `http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=22109`

- The ranking methods can be specified in a similar manner, using the option `--ranking=...`. The available values by default are:
  - `semantic_similarity`
  - `avg_frequency`
  - `beginning_frequency`
  - `most_known`
  - `longest`
  - `shortest`

  Available methods are all methods of the `Splitter` class that start with `rank_`. When not explicitly specified, this option defaults to the value `avg_frequency,semantic_similarity,shortest`. All ranking methods are described in Section 5.5.

- When given the switch `--evaluate`, instead of decompounding a file consisting of one word per line, the script expects a tab-separated file of one word, and one gold annotation per line. It will then attempt to decompound all words in the file and print evaluation metrics.

Wherever it makes sense, short-form arguments are also supported.

When called, the script first reads in the lexicon and then the list of stop words from two files in the `lex` directory. Next, it will read the suffix and prefix lists. All lexical files are named with the system *language code.resource.filetype*, for example `de.lexicon.tsv`. Languages codes are the two-digit ISO 639-1 codes for each languages (i.e. `de`, `sv`, and `hu`). How resource types relate to file format is shown in Table 6.1.

Resource	**File type**	**File extension**
lexicon	Tab-Seperated Values	`tsv`
prefixes	Raw text	`txt`
suffixes	Raw text	`txt`
stopwords	Raw text	`txt`
vectors	Pickled Python object	`pkl`

Table 6.1: The file types of the different lexical resources.

While the lexicon is expected to contain one word per line, in the format *frequency*\t*word*\n, it will also accept lines that start with some amount of whitespace. This is such that it can handle output from `cat raw_words.txt | sort | uniq -c | sort -nr`. The stop word list simply consists of one stop word per line.

After this initial setup is done, the system will read words from the input file, one at a time. The decompounded result is then printed to the standard output, in the same format the evaluation corpus uses.

Because it does not wait for the input file to be read in completely, it is possible to run the decompounding system as a server and use it on-demand, for

example using a named pipe created using `mkfifo`, or by making it read from the standard input by using `-` as a file name.

6.3 Python class interface

An alternative to using the command-line interface is writing Python code that imports and uses the `Splitter` class. This has several advantages:

- All parameters including the list of cleaning and ranking methods can be manually set, and even changed without having to restart (or re-import) the software.
- Multiple `Splitter` objects with different settings or languages can be instantiated at the same time.
- The program can be extended, for example to add new cleaning or ranking methods. For details see Section 6.4.

When used from another Python script, the returned value can alternatively be in tuple form, then the result will be a tuple of compound parts.

6.4 Extending the splitter

While it is possible to adapt our system to many needs by tweaking parameters, there are a few cases in which it will not be enough:

- When wanting to use the system on a language other than German, Swedish, or Hungarian
- When wanting to use new cleaning or ranking methods that are not included.
- When wanting to change core parts of the program.

In these cases, instead of writing an entirely new system from scratch, one can build on top of our system by sub-classing it. The program contains a single class called `Splitter`, which a new class can inherit from.

What follows is a description of how to extend it in the most common ways. If it is desired to change the program in a different way, one has to read the code anyways, so it does not make sense to describe it here in detail.

6.4.1 Adding languages

Adding support for a new language requires only a few lines of code. All that needs to be done is to implement the method `set_language(self, language)`, which takes a two-letter language code. In it one should:

1. Call `super(type(self), self).set_language(language)` to set the language and handle the existing languages
2. If a `NotImplementedError` occurs, check for `language` being any newly supported language, in which case `self.binding_morphemes` should be set to a list of linking morphemes.
3. If `language` is not one of the newly supported languages, re-raise the error.

An example of how this would look is shown in Figure 6.1.

While this is the only code that would need to be written, lexical resources will still need to be added to the `lex/` folder, at the very minimum the lexicon (*language_code*.lexicon.tsv), and depending on the cleaning methods, ranking methods, and options used the lists of stopwords, prefixes, and suffixes, as well as the vector space obtained from calling `utils/train_embeddings.py`.

```
import splitter

class DanishSplitter(splitter.Splitter):
    def set_language(self, language):
        try:
            super(type(self), self).set_language(language)
        except NotImplementedError as e:
            if language == 'da':
                self.binding_morphemes = ['en', 's']
            else:
                raise e
```

Figure 6.1: Code example of adding a language.

6.4.2 Adding ranking or cleaning methods

Adding new ways to rank or clean the splits is done by adding new methods with specific names: Any method starting with `rank_` will be accessible from the command-line interface as a ranking method, the same is true for `clean_` and cleaning methods.

Cleaning methods are given an iterable of splits (which are tuples of strings) and are also supposed to return an iterable of splits. The built-in cleaning methods are all generators, meaning they use the `yield` keyword to return splits individually and lazily. An example of a new cleaning method that removes all splits that have exactly three parts is shown in Figure 6.2.

```
import splitter

class SplitterWithCleaning(splitter.Splitter):
    def clean_not_three_parts(self, splits):
        for split in splits:
            if len(split) != 3:
                yield split
```

Figure 6.2: Code example of adding a cleaning method.

Ranking methods are given a split, i.e. a tuple of compound parts. They return a number (floating point or integer[6]), where a higher number stands for a better split. What range these numbers come from is irrelevant, as long as they are comparable *within one ranking method*. An example of a new ranking method that ranks splits by their number of breaks between double letters is shown in Figure 6.3.

```
import splitter

class SplitterWithRanking(splitter.Splitter):
    def rank_double_letter_splits(self, split):
        return sum(
            1 if left[-1] == right[0] else 0
            for left, right in zip(split, split[1:])
            )
```

Figure 6.3: Code example of adding a ranking method.

To be able to do anything serious with added cleaning or ranking methods, one should read the entirety of the code as to get an understanding of the existing data structures and the inner workings of the program.

[6]Technically, it does not even need to be a number, just a type that can be compared to other values of its type, but for practical purposes a numerical type is probably always the best choice.

7. Evaluation

Our system is evaluated against other systems (in the case of German) and a baseline (in all cases) here. The competing systems are:

- **Baseline:** This is a hypothetical[1] identity system which when given any word, returns the same word. When seen as a decompounding system, it always postulates that any given word is not a compound. Since most words in the evaluation corpus are not compounds, its Accuracy will be fairly high, but Recall and F1-score are zero (since it never finds a single compound), and its Precision is undefined.
- **ASV Toolbox:** This is a supervised system described in Biemann et al. [2008].
- **jWordSplitter:** Originally created by Sven Abels, and now maintained by Daniel Naber, this is a rule-based Java library for splitting German compounds[2].

Our system is evaluated in various configurations. They are named mostly based on the ranking methods, using an abbreviation system:

- **K** stands for the ranking method `most_known`, the fraction of known (in-vocabulary) parts.
- **F** stands for the ranking method `avg_frequency`; the arithmetic mean of frequencies of the parts in our lexicon.
- **B** stands for a variant of `avg_frequency`: `beginning_frequency`, in which we rank by the arithmetic mean of frequencies of the first (up to) six letters of the part. An additional lexicon for this purpose is automatically created when the normal one is read.
- **S** stands for `shortest`, the ranking method that will prefer splits with few parts.
- **L** is the ranking method `longest`; the opposite of `shortest`.
- **V** stands for the ranking method `semantic_similarity`; the vector-based ranking method.
- **force** specifies that the `--force-split` switch was used, which makes the program always prefer splits with at least two parts, if they exist.

A large number of combinations of ranking methods, cleaning methods, and other options is possible, of which we will only evaluate a reasonable subset:

- **K**: This is the most basic version using the raw words without frequency information. The splits are ranked based on how many parts are known words.

[1] It was not actually implemented as its performance metrics are evident from the data.
[2] `https://github.com/danielnaber/jwordsplitter`

- **K+S**: We now additionally chose the shortest of the splits (the split with the fewest (non-linking-morpheme) parts) in the event of a tie.
- **F+S**: Instead of ranking by the fraction of known parts, we rank by the average (arithmetic mean) of the parts.
- **K+F**: Next, we combine the two methods. Since the frequency-based method is less likely to produce ties, we rank by the fraction of known words first.
- **B+S**: This method uses the frequencies of the beginnings of parts. Since this way of unifying parts is also used in the `semantic_similarity` ranking method, we test whether it also helps with the frequency-based methods.
- **V+S**: Ranks first by the average cosine distance between vectors of parts, and then by length (preferring splits with fewer parts).
- **K+V+S**: We investigate whether we can improve the semantic method with pure word knowledge.
- **V+S+force**: This method enforces splitting when possible. It is not expected to perform well, because of the high number of non-compounds in the evaluation data (and in the real world). However, when this system is used within another system that "knows" that a word is a compound, but does not know how to split it, it could be of use. To enforce splitting, use the command-line switch `--force-split` (see Section 6.2).

The default list of cleaning methods is used here, the effects of cleaning will be evaluated separately.

7.1 Metrics

Since this is not a simple binary classification task, the definitions for Precision and Recall (and therefore, F-Score) differ slightly:

$$\text{Precision} = \frac{\text{correct splits}}{\text{correct splits} + \text{superfluous splits} + \text{wrong splits}}$$

$$\text{Recall} = \frac{\text{correct splits}}{\text{correct splits} + \text{incorrect non-splits}}$$

As accuracy is only dependent on the correctness of the work, it keeps the familiar definition:

$$\text{Accuracy} = \frac{\text{correct splits} + \text{correct non-splits}}{\text{all instances}}$$

The F1-score is defined as a trade-off between Precision and Recall, giving them equal importance:

$$\text{F1-score} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

	Precision	Recall	Accuracy	F1
Baseline	-	.00	.80	.00
ASV Toolbox	.41	.42	.75	.41
jWordSplitter	**.88**	.57	**.91**	.69
K	.49	.41	.82	.45
K+S	.50	.41	.83	.45
F+S	.45	.46	.80	.45
K+F	.53	.53	.84	.53
B+S	.55	.43	.83	.48
V+S	*.78*	***.63***	***.91***	***.70***
K+V+S	.54	.54	.85	.54
V+S+force	.31	.31	.66	.31

Table 7.1: Performance of all systems compared on the German evaluation corpus. The best system for a score is marked in bold, the best of our systems in italic.

Whether the F1-score is a fitting evaluation metric depends on the task at hand, on what is more important; Precision or Recall. Since we report both of them here, too, only the F1-score is included here, but any other kinds of F-scores can easily be computed from Precision and Recall.

7.2 Results

The results for the German evaluation task can be seen in Table 7.1. As with all classification systems, the question is which metrics are more important to the user:

Putting an emphasis on Recall means caring more about splitting all compounds than about potential incorrect decompositions. Higher Precision focuses on being more "careful", and preferring to decompound easier compounds correctly, rather than decompounding too much. Since most words in the evaluation corpora are not compounds, obtaining a high accuracy value is easy: The simple identity baseline obtains 80% accuracy, because 80% of the instances in the German corpus are not compounds. One could argue for such a system to have perfect Precision, but it is marked here as not having a value. However, a system that has a few cases hard-coded could still easily obtain a precision of 100%.

In this task, it is not as easy to obtain a high Recall. Whereas in binary classification tasks selecting *every instance* yields a recall of 100%, due to the more complex job a decompounding system has, the same result cannot be obtained here.

Our system with the setting **V+S** wins in Recall and F1-score against all other systems. While its Precision is the highest among our systems, it loses against the rule-based **jWordSplitter**. This is to be expected: With simple, conservative rules, you can already achieve a high Precision, or even a perfect one when hard-coding at least one case and not splitting anything else. Such a system would have bad Recall and Coverage scores however.

	Precision	Recall	Accuracy	F1
Baseline	-	.00	.72	.00
K	.45	.33	.73	.38
K+S	.45	.33	.73	.38
F+S	.35	.32	.68	.33
K+F	.44	.36	.73	.40
B+S	.45	.32	.73	.38
V+S	**.78**	**.38**	**.81**	**.51**
K+V+S	.46	.37	.74	41
V+S+force	.34	.30	.63	32

Table 7.2: Performance of all systems compared on the Swedish evaluation corpus. The best system for a score is marked in bold.

That is not to say that jWordSplitter does this — it still performs quite well in the other metrics and is almost as good in the F1-score, but its rule-based nature does lead to a higher Precision and lower Recall.

While **V+S** is the clear winner, some observations can be made about the other systems, too: Forcing decomposition (**V+S+force**) — as expected — is a bad idea in general, it has the worst results of all compared systems. Running it on a data set that consists entirely of compounds would be expected to improve upon **V+S** — at the very least it will not make it worse.

Using frequency information from only the beginnings seems to be better than using complete words. The idea of **B** was a trade-off: Losing a little information through only considering the beginnings of words, but gaining information through the much higher frequency of individual counts, and through reducing data sparsity significantly. Depending on the lexicon (especially its size), this might have more or less of an effect. In our case, despite having a very large source for the lexicon, this method pays off, increasing Precision quite a bit, while losing much less in Recall, if anything at all. It is expected that the technique would be more beneficial if a large source text for the creation of the lexicon was not available.

As the situation is similar in the other languages, we set the configuration **V+S** as the default configuration for our system.

In the Swedish data set (see Table 7.2), the situation is similar. While the Precision of **V+S** is as high as on the German data, Recall is much lower. The reason for this is not clear, but what is interesting is that the Accuracy of the baseline is also worse than in German, which means that more of the data is compounds. This does not mean that more Swedish words are compounds compared to German, instead it could point to Swedish words being shorter in general, leading to less non-compounds in the data set (since we removed words shorter than seven characters prior to the annotation). This suspicion is confirmed on our evaluation corpus (see Figure 1.1).

On the Hungarian corpus, the results are worse than on Swedish, and Precision also drops significantly here. This could be due to language-specific details that the author does not know about. Our methods (especially the cleaning methods) are based mostly on our intuition with German (and Swedish, to an extent, which

	Precision	Recall	Accuracy	F1
Baseline	-	.00	**.85**	.00
K	.23	.18	.76	.20
K+S	.23	.18	.76	.20
F+S	.18	.17	.66	.18
K+F	.23	.21	.74	.22
B+S	.36	.25	.82	.30
V+S	**.46**	**.31**	**.85**	**.37**
K+V+S	.24	.21	.75	23
V+S+force	.16	.16	.58	.16

Table 7.3: Performance of all systems compared on the Hungarian evaluation corpus. The best system for a score is marked in bold.

is of course much more similar to German than Hungarian). It is reassuring though that **V+S** wins in all three languages, excluding the possibility of this being the result of randomness.

Annotation quality discrepancies could also have played a role in the differences between the languages, but since the author speaks neither Swedish nor Hungarian, that is merely another hypothesis.

It should be noted, that the numbers given here do not necessarily say much about the usefulness of the decompounding systems within other tasks (this is why it can be more insightful to evaluate using a task-based method; i.e. measuring how much a system's results on a task improve when using decompounding, this is e.g. how Daiber et al. [2015] and Erbs et al. [2015] evaluate):

A given system using a decompounder is typically not completely unknowledgable about what word is a compound: It will often have POS-tagged the source text already, and might only resort to using the decompounding system when a given word cannot be found in a lexicon. In such a case, Recall can be more important than Precision.

We therefore also report coverage percentages, defined as the fraction of compounds that are correctly identified as compounds by the various systems (but not necessarily correctly split):

$$\text{Coverage} = \frac{\text{correct splits} + \text{incorrect splits (of compounds)}}{\text{all compounds}}$$

The Coverage values can be seen in Table 7.4 for all languages. In this case, Coverage cannot be seen as a direct measure of system quality, but more of the eagerness of a system to split. From our systems and across all languages, **V+S+force** is the one with the highest coverage, simply because it will always split when it can, and therefore misses very few compounds, at the price of decompounding many non-compounds.

The next-best systems in terms of coverage are those using the `avg_frequency` ranking method (**F+S** and **K+F**).

	German	Swedish	Hungarian
Baseline	.00	.00	.00
ASV Toolbox	**.95**		
jWordSplitter	.62		
K	.65	.54	.50
K+S	.65	.54	.50
F+S	.86	.64	.65
K+F	.86	.61	.61
B+S	.63	.53	.47
V+S	.73	.43	.49
K+V+S	.85	.60	.59
V+S+force	*.87*	***.66***	***.70***

Table 7.4: Coverage of the various systems on all languages

7.3 Quantitative error analysis

We also do an error analysis of all systems but **V+S+force** (because the mistakes it makes are expectable) in Table 7.5. Most of the time a system makes a mistake, it is in how many splits a word should be split, rarely is there a case where a system splits the compound in the right amount of pieces, but in the wrong position(s). This is to be expected.

From our systems, we can see that once word frequencies are introduced, the amount of under-split compounds is drastically reduced. Introducing word embeddings significantly lowers over-splits, while at the same time not increasing under-splits by a large degree.

What is interesting is that although **B+S** did not turn out to improve our performance metrics, it does reduce over-splits and incorrect splits.

Interestingly, while in German we reach almost the same number of over- and under-splits, in the other languages there are more under-splits by quite a bit. This points to too many words not being known; i.e. a corpus that is too small.

7.4 Qualitative error analysis

To better understand the kinds of errors our system still makes, we looked into some of these errors on the German data. The proportions of the results of this analysis might be different in the other languages, but it would be surprising to see other error types.

In our investigation we found 6 main error types:

1. **over-splits because of spurious words in the lexicon:** Noise in the lexicon causes splits to be generated that make no sense at all. For example, *Senkung* (reduction; lowering) gets split as *Sen+kung*, with both *sen* and *kung* not being German words.

2. **over-splits that could be considered correct, but were annotated differently in the evaluation corpus:** The annotation task was deliberately left ambiguous in some cases, because the task is not always one with

	system	under-split	over-split	wrongly-split
German	ASV Toolbox	12	214	7
	jWordSplitter	76	13	0
	K	73	81	28
	K+S	73	80	27
	F+S	30	140	32
	K+F	29	106	28
	B+S	77	78	15
	V+S	59	27	15
	K+V+S	34	95	28
Swedish	**K**	58	77	26
	K+S	77	58	26
	F+S	61	93	37
	K+F	64	73	28
	B+S	78	60	22
	V+S	94	15	6
	K+V+S	65	63	27
Hungarian	**K**	278	494	91
	K+S	278	494	91
	F+S	197	885	124
	K+F	217	632	110
	B+S	295	297	48
	V+S	281	202	47
	K+V+S	232	579	98

Table 7.5: Error analysis. *under-split* are those instances that are split into less parts than they should have been. *over-split* are those instances that are split into more parts than they should have been. *wrongly-split* are those instances, which have the right amount of splits, but are incorrectly split.

a single right answer. Annotators were told to annotate lexicalized compounds as non-compounds, and non-lexicalized compounds as compounds, but were free to make their own decisions in cases between lexicalized and non-lexicalized compounds. For this reason, some of the errors made are because the system made a different decision than the annotator, that could in some cases also be considered correct. An example of this is *Zeitraum*, which our system split into *Zeit+raum*, but was not split in the annotation.

3. **under-splits that could be considered correct, but were annotated differently in the evaluation corpus:** The opposite of the previous case also happens, and actually seems to be the most frequent kind of error. In many cases, the author has to agree with the system more than with his past self, such as splitting *Zahnfleischbluten* (gum bleeding) as *Zahnfleisch+bluten* (gum + bleeding), and not as annotated in the evaluation corpus (*Zahn+fleisch+bluten*; tooth + meat + bleeding).

4. **over-splits of morphological (non-compound) boundaries:** In some cases, despite our best efforts, the system still splits off sub-word units, like derivational suffixes. For example, *Schlaflosigkeit* (sleeplessness), instead of not being split at all, was split into *Schlaf+losigkeit* (sleep + "lessness").

5. **incorrect splits because of annotation errors:** One case of undoubtedly incorrect annotation was found that the system correctly judged to be a non-compound: *Überdosierungen* (overdosages) was incorrectly annotated as *Über+dosierungen*.

6. **incorrect ranking:** The remaining cases are the truly ambiguous compounds that the system split incorrectly. *Patientinnen* (patients (female)) was split into *Patient+innen* (patient inside).

In many cases, it is hard to unambiguously assign a given misclassification to one of these error types, which is why we don't report their distribution, as we feel that such numbers would be skewed by subjectiveness.

7.5 Parameter analysis

In order to further investigate the results, we look into how our results change if we use different settings (apart from the ranking methods). We first look at the effect of lexicon size, then at our methods of cleaning the candidate splits.

7.5.1 Lexicon size

It is self-evident that the size of our "vocabulary"; our lexicon we use for generating candidate splits and ranking in the case of the ranking methods *avg_frequency*, `beginning_frequency`, and `most_known` has an influence on our scores, but the question is how exactly that influence looks like.

We can certainly assume that the law of diminishing returns will apply, that is: the more frequent a word is, the more it will help to have it in our lexicon, while adding a very infrequent word will not help much, if at all.

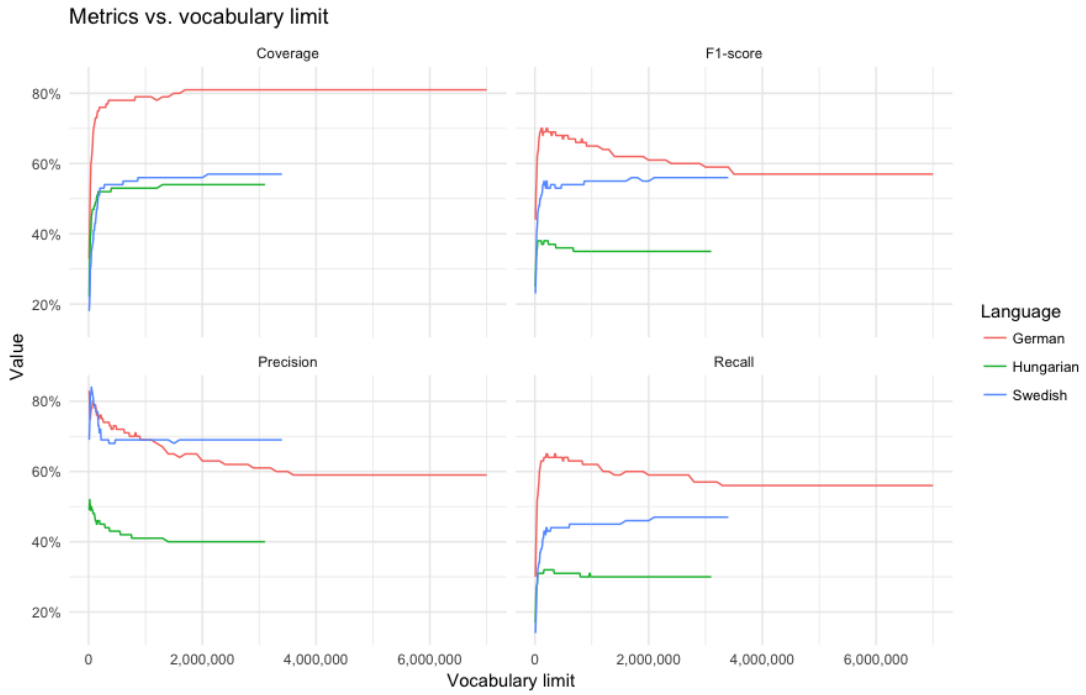

Figure 7.1: Impact of vocabulary size on Precision, Recall, F-Score, and Coverage. The lines for Hungarian and Swedish cut off at their respective full vocabulary size. German is cut off at 7 million, which is very close to its actual limit (see Table 4.2).

We further proposed that at some point, adding more words will actually harm our performance. This claim comes from the fact that our lexicon is based on raw data, which no matter how well it is cleaned, contains some noise. The more (infrequent) words we will take into our lexicon, the more likely is it that we also include noise. Even in the case of a perfect, handcrafted lexicon, a similar effect might be present where infrequent words are correct, but so unlikely that they are even rarer to occur in compounds. The more words exist in our dictionary, the more choice we give to our ranking methods, and the more chances to choose the wrong answer.

We performed an experiment where we ran our best system (**V+S**) with a given vocabulary size limit (`--limit` option in the command-line interface), and plotted the Coverage, F1-score, Precision, and Recall in relation to that limit (see Figure 7.1).

As expected, Coverage increases when the vocabulary size increases. This is due to the fact that the system will know more ways to split a word.

The interesting results lie in how Precision, Recall, and F1-score behave: Initially, adding information drastically increases the performance, but it drops very quickly afterwards (before 500,000). While this effect was expected, it is perhaps a bit surprising just how quickly it starts to have an effect. On the other hand, a vocabulary size of half a million is already quite big in languages without very intricate morphology.

What is the most surprising is that in F1-score and Recall, the described effect does not seem to occur in Swedish at all.

7.5.2 Cleaning methods

V+S	German			Swedish			Hungarian		
without	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
general	+2%	+2%	+2%	±0%	+1%	+1%	±0%	±0%	±0%
last_parts	−9%	−6%	−7%	−14%	−2%	−5%	−9%	−3%	−5%
prefix	−4%	−1%	−3%	±0%	+1%	+1%	±0%	+2%	+2%
fragments	±0%	±0%	±0%	±0%	±0%	±0%	±0%	±0%	±0%
suffix	−12%	−7%	−9%	−7%	−1%	−3%	−4%	−1%	−2%
all	−35%	−22%	−28%	−56%	−19%	−32%	−35%	−19%	−26%

Table 7.6: Change in Precision, Recall, and F1-Score by removing one of the cleaning methods. While most methods help the scores, fragments does not change anything at all, and general actually hurts the scores. This comparison is made on our best system (**V+S**).

We also analyse the effect of the cleaning methods on the results. In Table 7.6, our results of **V+S** with the default cleaning methods on all languages are compared to the same system without one specific cleaning method. Removing either the cleaning methods `last_parts`, or `suffix` makes Precision, Recall, and F1-score worse in all three languages, so these are methods that always seem to help.

`prefix` has a small benefit for German, and slightly hurts the results in Swedish and Hungarian. The `general` cleaning method apparently hurts the results in German and Swedish, and does not change anything in Hungarian. Removing it changes our F1-score in German to 72%, and in Swedish to 52%.

Perhaps surprisingly, the `fragments` cleaning method does not change anything anywhere, across all languages.

The analysis furthermore shows that although our intuition did not lead to the best possible setup, the cleaning methods did significantly improve the results together. Our reported F1-score would be about thirty percent worse across languages without the use of any cleaning methods.

7.5.3 Stopwords

Finally, the effect of our stopword list is evaluated. We re-run the evaluation of our best system (**V+S**), but provide the `--no-stopwords` switch to disable the use of the stopword list. The change in percent compared to the normal system can be seen in Table 7.7.

The effect is not large, and differs by language. In Swedish, the stopword lists seems to help, whereas in German and Hungarian Recall is hurt slightly by using them.

We offer two explanations for these results:

- Mistakes that would be solved by removing stopwords from the lexicon are also resolved to a large degree by a cleaning method, for example `prefix` (many stopwords, in particular prepositions, are also prefixes). In that case, removing stopwords could occasionally prevent legitimate splits, reducing

	Precision	Recall	Accuracy	F1-Score
German	±0%	+1%	±0%	±0%
Swedish	−3%	±0%	−1%	±0%
Hungarian	±0%	+1%	±0%	±0%

Table 7.7: Change in Precision, Recall, and F1-Score when using `--no-stopwords`. The switch seems to have little effect. The comparison is made on our best system

Recall (as it can be observed). This would also explain why removing `prefix` did not have a big effect (see Table 7.6) — because the removal was mitigated by the removal of stopwords from the lexicon.

- The results might be different on other ranking methods. We do not exhaustively evaluate on all possible configurations, partly due to the large parameter space, and partly because it would be more likely that a configuration wins purely by chance. It might be that our semantic method (`semantic_similarity`) is unlikely to rank splits containing stopwords because their vectors are very distant to content words in our vector spaces.

8. Conclusion

This work compared previous work on decompounding applied to the medical domain, and developed a new system that is compatible with German, Swedish, and Hungarian, and can easily be extended to work with other languages as well.

It is clear that as easy as the task might seem at first, it is far from solved. Given a good lexicon, all systems can decompound obvious, clear cases, but more complex compounds, those that are ambiguous, and those being noisy in other ways are still hard to split.

This is especially true in a specific domain such as the medical one, as the language and the used compounds are different from regular language. While some compounds, like *Krankenschwester* (nurse, literally "sick sister", as in "sister for the sick") are in common use in colloquial language, more specific terms, such as *Thrombozytenzahlen* are not. This makes the task more difficult than it would be for non-expert language[1], and explains the comparably low results of all systems when compared to other evaluations (such as in Erbs et al. [2015]).

It is also apparent that the amount of data used influences the accuracy of the various decompounding systems, as the results for German are consistently higher than for the other languages.

Evidently this kind of language-agnostic word splitting system is one with limited use. When building a system for a specific language, one could employ any number of language-specific improvements. While we did end up using word embeddings to improve the ranking procedure, our stemming method was very primitive in order to stay language-independent. It was nevertheless required:

Given a candidate split S consisting of the parts A, B, we need to either lemmatize or stem both A and B to be able to map them to their semantic vectors. One benefit of our basic left-to-right system described in the Chapter *Methods* is that we can largely ignore morphological changes like suffixes, because they typically only happen to the head, apart from the binding morphemes that we do handle. This is why we also chose such a simple stemming method, but better pre-processing tools would probably improve our results.

The downsides of our system are also its benefits: Without any annotated training data, it will likely work for any additional languages with composition systems similar to Germanic, Skandinavian, or Uralic languages. For example, Modern Greek also exhibits compounding, and given a Wikipedia dump or something similar as a source corpus, it is hypothesized that it will also work for Greek.

[1] Usually, when solving a subset of a problem, or when solving a problem on a subset of some amount of data, the problem gets *easier*, not harder. The average performance on a harder-than-average subset can still go lower, if this subset is not a random subset, with respect to difficulty. For example, in computer vision, the task of identifying the presence of an animal in a picture might be easier in general then the sub-task of identifying zebras in tall grass. The performance of identifying zebras in tall grass might be higher by restricting the system to all tall-grass pictures, but the average performance of finding zebras in tall grass can still be lower than the average performance of finding animals in pictures in general.

Like all systems not exclusively concerned with semantics, it can handle both endocentric and exocentric compounds, while some of the systems, like those that are only based on word embeddings cannot decompound exocentric compounds (e.g. Daiber et al. [2015]).

For future work we propose the following:

- **Evalutating on other domains.** Our medical evaluation corpus has proven to be a tough one for all systems. A cleaner, balanced evaluation corpus not restricted to a specific domain would be more informative regarding the performance of decompounding systems in naturally occurring language.

- **Learn from easy compounds.** A system could, instead of splitting compounds one by one, require a list of words to decompound, before it will start the decompounding process. It could then first decompound those instances that are less ambiguous, or for other reasons easier to decompound, and then learn from those presumably correct splits. This way, the system would stay unsupervised, but still acquire examples to learn from during execution. In very recent work, Riedl and Biemann [2016] use a similar method to generate a dictionary of "single atomic word units".

- **Evaluation on non-compounding languages.** As a sort of "sanity check", one could run decompounding systems on non-compounding languages. Ideally, very few words would be tagged as compounds and split, except possibly for lexicalized compounds.

- **Knowledge transfer from high-resource languages.** We purposefully do not use methods requiring large amounts of annotated data, or language-specific tools such as POS-taggers. One way of staying true to our goals, but still using such methods would be to transfer information from high-resource languages to low-resource languages: For example, word vectors could be first trained on German corpora, and then retrained on other (Germanic) languages, or a (more sophisticated) stemmer from German could be blindly applied to other languages.

Looking back at our goals defined in the Chapter *Introduction*, **Language agnosticism** is achieved, because the only expert knowledge required to extend the system to another language is a list of linking morphemes of that language. Even when not supplying this list, the system will work, although to a lesser degree. The additional lexical resources are either automatically obtainable from raw text (such as the word embeddings, and the lexicon), or optional and available freely on the internet (the stopword lists and lists of affixes).

Another big goal was **low resource friendliness**. Corpora of the size we used[2] are available in many languages, not just the few big ones with lots of attention from the NLP community. We also do not require the existence of any other parts of the NLP pipeline (stemmers, lemmatizers, POS-taggers, parsers, ...).

[2]Although the Wikipedia corpora are still relatively big, we have shown diminishing returns from larger sizes.

In terms of **usability**, if the word embedding method is not used[3], the system has only one small dependency. Once the dictionary and the lists of stop words and affixes are read in, decomposition takes less than a millisecond even for long, complex examples.

[3]The `gensim` package required for the use of our embeddings requires bigger dependencies, but those (namely `numpy` and `scipy`) are frequently used packages in the scientific Python community.

Bibliography

Martine Adda-Decker and Gilles Adda. Morphological decomposition for ASR in german. In *Workshop on Phonetics and Phonology in ASR, Saarbrücken, Germany*, pages 129–143, 2000.

Enrique Alfonseca, Slaven Bilac, and Stefan Pharies. German decompounding in a difficult corpus. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 128–139. Springer, 2008.

Marco Baroni, Johannes Matiasek, and Harald Trost. Predicting the components of german nominal compounds. In *ECAI*, pages 470–474, 2002.

Chris Biemann, Uwe Quasthoff, Gerhard Heyer, and Florian Holz. Asv toolbox: a modular collection of language exploration tools. In *LREC*, 2008.

Claudia Bretschneider and Sonja Zillner. Semantic splitting of german medical compounds. In *International Conference on Text, Speech, and Dialogue*, pages 207–215. Springer, 2015.

Joachim Daiber, Lautaro Quiroz, Roger Wechsler, and Stella Frank. Splitting compounds by semantic analogy. *arXiv preprint arXiv:1509.04473*, 2015.

Nicolai Erbs, Pedro Bispo Santos, Torsten Zesch, and Iryna Gurevych. Counting What Counts: Decompounding for Keyphrase Extraction. *Acl-Ijcnlp 2015*, pages 10–17, 2015.

Philipp Koehn and Kevin Knight. Empirical methods for compound splitting. In *Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics-Volume 1*, pages 187–193. Association for Computational Linguistics, 2003.

Stefan Langer. Zur Morphologie und Semantik von Nominalkomposita. In *Tagungsband der 4. Konferenz zur Verarbeitung natürlicher Sprache (KONVENS)*, pages 83–97, 1998.

Rochelle Lieber and Pavol Stekauer. *The Oxford handbook of compounding*. Oxford University Press, 2009.

Andrew L Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 142–150. Association for Computational Linguistics, 2011.

Torsten Marek. Analysis of German Compounds using Weighted Finite State Transducers. *Bachelor thesis, University of Tübingen*, 2006.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. ISSN 1521-9615. doi: 10.1109/MCSE.2007.53. URL `http://ipython.org`.

Ulrike Rackow, Ido Dagan, and Ulrike Schwall. Automatic translation of noun compounds. In *Proceedings of the 14th conference on Computational linguistics-Volume 4*, pages 1249–1253. Association for Computational Linguistics, 1992.

Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

Martin Riedl and Chris Biemann. Unsupervised compound splitting with distributional semantics rivals supervised methods. In *Proceedings of NAACL-HLT*, pages 617–622, 2016.

Anne Schiller. German compound analysis with wfsc. In *International Workshop on Finite-State Methods and Natural Language Processing*, pages 239–246. Springer, 2005.

Richard Socher, John Bauer, Christopher D Manning, and Andrew Y Ng. Parsing with compositional vector grammars. In *ACL (1)*, pages 455–465, 2013.

Radu Soricut and Franz Och. Unsupervised morphology induction using word embeddings. In *Proc. NAACL*, 2015.

Jörg Tiedemann. News from opus-a collection of multilingual parallel corpora with tools and interfaces. In *Recent advances in natural language processing*, volume 5, pages 237–248, 2009.

Attachments

The attachment is provided in shape of a Github repository at `https://github.com/L3viathan/compound-splitter`, containing:

1. **splitter.py:** The main system that this work is concerned with; a Python script.
2. **README.md:** Usage instructions for the splitter in Markdown format.
3. **requirements.txt:** A requirements file for use with `pip` to install the required dependencies.
4. **gold_corpus:** A folder containing the evaluation corpus files in German, Swedish, and Hungarian. Each file contains one word-annotation pair per line, seperated by a tab character.
5. **utils:** A folder containing additional scripts for training of the embeddings and creating the lexica from a raw Wikipedia corpus.
6. **lex:** A folder containing all the lexical resources used by the script (lexica, stopword lists, affix lists, and word embeddings).