Analyzing German Noun Compounds using a Web-Scale Dataset

UIMA Software Project WS 2010/2011

Jens Haase

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Abstract

Noun-compounding in the German language is a tool to combine two or more single words to new words. In information retrieval and search engines this can be a problem because mostly documents with the combined and the splitted words are relevant for a search query. To get better search results it is often required to split words into single parts.

Contents

1 Introduction

1.1 Motivation

Noun-compounding in the German language is a powerful process to build new words with the combination of two or more existing words. *Blumensträuße* (flower bouquet) for example contains the two words *Blumen* (flower) and *Sträuße* (bouquet).

In Unstructured Information Management this can be a handicap for tools like Machine Translation, Speech Recognition or Information Retrieval. Imagine you search for the word *Lackschicht* (paint layer), the expected result will be all documents containing the word *Lackschicht* but also the words *Schicht aus Lack* (layer of paint). To find a expected result the search engine must split all compounds nouns in single words (*Lackschicht -> Lack schicht*).

1.2 Problem definition

In the German language a compounds can be formed by combining nouns, verbs and adjectives. Also compound words can combined to a new compound word. The word *Donaudampfschifffahrtskapitänsmütze* (a cap for captains of steamboats driven on the river Donau), contains the words *Donaudampfschiff* (steamboat on the river Donau) and *Kapitänsmütze* (the cap of the captain). Each word can be split again. The final decompounds words are *Donau*, *Dampf*, *Schiff*, *Fahrt*, *Kapitän* and *Mütze*.

Another problem in noun decompounds are linking morphemes. These morphemes are allowed between two words. The linking morpheme in *Kaptiänsmütze* is the the *s* between *Kapitän* and *Mütze* (*Kapitän(s)+mütze*). Also the word *Orangensaft* (orange juice) has a *n* as linking morpheme (*Orange(n)+saft)*. But in this case the word *Orangen* is a valid word in the German language. It is the plural of the word *Orange*. But the word *Kapitäns* in previous example is not a valid word.

At least not all possible splits have the same meaning. The word *Tagesration* (recommended daily amount), contains the word *Tag* (day), *Rat* (advice) and *Ion* (ion) but also the word *Ration* (ration). Splitting the the word in three parts is wrong, because the chemical ion has nothing to do with the word *Tagesration*. The correct decompounding here will be *Tag(es)+ration*.

1.3 Task description

Decompounding of a word is mostly done in three steps [ea08]:

1. Calculate every possible way of splitting a word in one or more parts
2. Score those parts according to some weighting function
3. Take the highest-scoring decomposition. If it contains one part, it means that the word in not a compound.

At first we have to split a word in all possible parts. For that we need a dictionary with all existing words. It is recommend to use a special dictionary for the analyzing corpus. This can simple be done in extracting all tokens of a document collection. A list of linking morphemes is also needed.

In the next step we need a weighting function for each decompound possibility. The can be for example a simple count frequency of every word. The words with more frequency get a higher weight than other. At the end we can rank the weights. The decomposition with the highest weight wins.

At the end we want to have UIMA component that can be used in other projects. It should able to work with special dictionaries and linking morphemes. Optional different weighting function can be chosen.

2 How to split a word

2.1 Helpful Tools

2.1.1 IGerman98 Dictionary

As mentioned early it is recommend to build a dictionary for a special corpus, but in this case we do not have a special corpus. Instead we want to use a *every day* dictionary, which are used for spell correction. ASpell and ispell are open source spell checker with dictionaries for nearly every language. Also hunspell is a modern spell checker, used in OpenOffice, Mozilla Firefox and Google Chrome.

The dictionary used in these tools is mostly the *IGerman98* dictionary from Björn Jacke. Different versions can be found on his website `http://www.j3e.de/ispell/igerman98/`.

The dictionary contains of two different files. A word list and a rule set. With the rule set the affixes (suffix and/or prefix) of a word can be changed.

2.1.2 Google Web1t and Lucene

Google Web1T corpus contains over 130 billion tokens of the German language, distributed over n-grams ($n \epsilon [1,5]$). The corpus is used for different ranking function.

The corpus itself provides no access or search tools. For that a index must be created. The most common tool for full text search today is Lucene. This will be used to create a index of the corpus and provided a search function.

Another tool to access the the data is the *jWeb1T* (`http://hlt.fbk.eu/en/technology/jWeb1t`) tool. This was not used because it can only check if a given n-gram is contained in the corpus.

2.1.3 Evaluation corpus

For the evaluate the result of the work the corpus of Marek [Mar06] was used. The corpus was semi-automatically created and contains over 160,000 examples.

The corpus can be found in the code resources or on Marek' website `http://diotavelli.net/files/ccorpus.txt`

2.1.4 CouchDB and Javascript

To visualize the splitting algorithm CouchDB and Javascript are used. CouchDB (`http://couchdb.apache.org/`) is an document data store, which stores JSON data. The complete database is accessible via a HTTP interface. Next to the JSON data a document can also have attachments. These can be used to serve a little application to the browser.

With some Javascript it is possible to write a complete application without a middleware between the client and the database. Together with Javascript InfoVis Toolkit `http://thejit.org` it is very easy to visualize the splitting algorithm in a browser.

2.1.5 UimaFit

At the end of the project a complete UIMA `http://uima.apache.org/` component should be implemented to use the work in further projects. UIMA is an framework for unstructured information

management systems. With the framework components can be implements and reused in different applications.

UimaFit `http://code.google.com/p/uimafit/` is build on top of UIMA to make development in UIMA easier, faster and more flexible. The library provides factories, injection and testing utilities.

2.2 Implementation

2.2.1 Prerequirements

Dictionary access

The simplest form of a dictionary is a list of words. This is what the `SimpleDictionary` class can do. It gets a filename as parameter and uses each line in this file as a word of the dictionary. This class can be easily used for your own special dictionary. For example if you export a dictionary from a large dataset.

For the rest of this work we used the popular IGerman98 dictionary from Björn Jacke. This dictionary consist of two parts. The first part is a list of annotated words. These are the base words of the dictionary. The second part is rule set that changes the base word in it's prefixes and suffixes. For example the following dictionary file contains three words; two of them are annotated (word and annotation are separated by a slash).

```
hello
try/B
work/AB
```

The rule set describes the annotation. `PFX` stands for Prefix and `SFX` for suffix. In the following files you see a prefix rule for `A` and a suffix rule for `B`:

```
PFX A Y 1
PFX A 0 re .

SFX B Y 2
SFX B 0 ed [\^y]
SFX B y ied y
```

Using this rules on the base words leads to new words. In this example we end up with six different words:

```
hello
try
tried
work
worked
rework
```

The rule set also allows the combination of suffix rules and affix rules. But in the current implementation of the `IGerman98Dictionary` this is not handled. If you want to create your own dictionary implementation just implement the `IDictionary` interface in your code.

As mentioned above for all further work the *IGerman98* dictionary is used. It currently contains 326,207 words. Without using the rule set the dictionary has only 81,612 words.

The ranking of the different split should be done with the Google Web1T dataset. The German dataset has a compressed file size from around 3.0 GB. The extracted files result in many text file, each containing a sorted list of n-grams. The following listing shows the file format of one row.

```
token-1 <space> ... <space> token-n <tab> frequency
```

Example:

```
relax                   150
relax on the couch      100
working is hard         200
```

While the compressed size of the German corpus is 3 GB, the extracted files have a size of 11,3 GB and contain following data:

Number of tokens:	131,435,672,897
Number of sentences:	15,715,470,319
Number of unigrams:	15,133,396
Number of bigrams:	88,668,144
Number of trigrams:	154,226,178
Number of fourgrams:	140,670,210
Number of fivegrams:	100,542,274
Number of n-grams:	499,240,202

The first plan to access the data in Java was to use the jWeb1T tool (`http://hlt.fbk.eu/en/node/81`). This tool can find n-grams in a logarithmic time. But it can only find the frequency for a given n-gram. For example if the n-gram 'hello world' is listed in the corpus with the frequency 50, the jWeb1T tool will find the n-gram by searching for 'hello world', but not by searching for 'hello' or 'world'.

The ranking function should search for 'hello' and get all n-grams containing the word 'hello'. When it searches for 'hello world' it want to find the all n-grams with the words 'hello' and 'world'.

A second tool is the Java API for Web-1T Corpus of Digital Pebble `http://www.digitalpebble.com/resources.html`. The tool is currently not free available, but it is possible to contact Digital Pebble. According to Julien Nioche of Digital Pebble the tool is comparable to jWeb1T, but they are currently working on a new version that will be freely available soon. Since there was no time to wait for this tool we have to create our own indexer.

To index all n-grams Lucene was used. Each Lucene document contains a n-gram and the frequency. That means we do not need the extracted Web1T file for the next steps anymore. Creating the index is done with `LuceneIndexer` class. but for your own generation you can use the command line tool under `bin/web1TLuceneIndexer`. It comes with the following options:

```
usage: LuceneIndexer
    --web1t <arg>          Folder with the web1t extracted documents
    --outputPath <arg>     File, where the index should be created
    --igerman98            (optional) If this argument is set, only words of
                           the german dictionary will be added to the index
    --index <arg>          (optional) Number of how many indexes should be
                           created. Default: 1
```

First you have to set the folder were all extracted text files of the web1t corpus are listed (`-web1t`). With the `outputPath` parameter you can specify were the index should be created on your disk. When you set the `igerman98` parameter only n-gram with words that are in the dictionary will be added to the index. For example if the dictionary contains the words *Blumen* and *Orange*, n-grams like *Blumen sind*

schön and *Orangen sind orange* will be added but *Hallo Welt* will be ignored. With this parameter you can reduce the size off the index on your disk. At least, with the *index* parameter you can split the index in several independent indexes. Lucene can only store `Integer.MAX_VALUE` number of documents per index. If you have a larger index you can use this option to split the index. Splitting the index to improve the speed will only work if the indexes run on different machines.

To access and search on the generated index the `Finder` class can be used. This class also detects splitted indexes in you put them in the same folder. To search for n-grams use the `find` function in the `Finder` class. This function returns a list of n-gram. Each n-grams contains the text and the frequency.

The generation of the index takes, depending on your hardware, a lot of time. On my machine the generation of the index takes about five hours. Writing to Solid State Disk or RAM can decrease the generation time. At the end, the index has a size of 23.4 GB without the `igerman` parameter. With the `igerman98` parameter the index is only 12.8 GB big.

With a normal hard disk and such a great index the queries are very slow. Most of the time they take about one seconds. This was also a problem for the evaluation. The evaluation of the complete corpus takes several days. For that only on a small amount of 10,000 examples the ranking algorithm were evaluated. To improve the query speed better hardware is need. It is highly recommended to put the index in a RAM. Also an other index mechanism can help to improve the speed.

The corpus used to evaluate the ranking function was generated with following command:

```
./bin/web1TLuceneIndexer \
  --web1t /path/to/web1t/folder \
  --outputPath /path/to/generated/index \
  --igerman98
```

Corpus reader

For the evaluation of the algorithm the corpus of Marek [Mar06] is used. The corpus was generated semi-automatic and contains 158,653 examples. Each line in the file contains a word and the correct split for the word. Splits are marked with + between individual parts. Each part also contains a annotation like `PREP, A, N, ....` Morphemes are separated with a `|` or brackets. The following listing shows some examples.

```
zugangsliste zu{PREP}+gang|s{A,N}+liste{N}
fachkameras fach{V,N}+kamera(s){N}
doppelprozessormaschine doppel{N,V}+prozessor{N}+maschine{N}
filmtauscher film{N,V}+tauscher{N}
minimalanforderungen minimal{A}+anforderung(en){N}
berufungsinstanz berufung|s{N}+instanz{N}
bleistiftstriche blei{N}+stift{N,V}+strich(e){N}
fernmeldeunion fern{A}+meld|e{V}+union{N}
ortsnetzmonopol ortsnetzmonopol
```

The `CcorpusReader` class provides access to data. It extends the `BufferedReader` class and adds the method `readSplit`. When calling this method the next line in the file is read and a `Split` is returned to the caller.

2.2.2 Splitting algorithms

The first step to split a word correctly is to generate all possible splits. For that we need a dictionary that contains correct words. The total algorithm could only as good as the splits that are provided by this part. Here I want to show two different algorithms.

Left to right splitting

The simplest way to split a word in different parts is to iterate from left to right over the word. Every iteration steps checks if the left part of the word is in the dictionary. The result is a list of pairs with a left part and a right part. Each of these pairs can be splitted again until no more split is possible. The following listing shows the splitting algorithm in pseudocode without checking for morphemes:

```
function split(word)
  result = List()
  for (i = 0..word.len)
    left = word[0..i+1]
    right = word[i+1..word.len]

    if (Dictionary.contains(left) and (right.len > 2 or right.len == 0))
      result += (left, right)

  return result
```

The split function can be called recursively on the left and right parts of the result. In the current implementation we store the splits in a tree. This tree can also be visualized, but more on that later. Notice, that this algorithm never checks if the right word is a valid one. If we take the word *Aktionsplan* a split *Akt+ionsplan* will occur. The word *ionsplan* is not correct German word. But it turns out that not checking the right word leads to better result in the evaluation of the algorithm.

The above pseudocode algorithm do not check for any morphemes in the word. In the real implementation this is added. But to give a idea of how the algorithm works this was removed. In the evaluation of the algorithm we want to check if the correct split is part of the list of splits. As mentioned above the return value of the complete split algorithm is a tree. In the evaluation we flat this tree to a list and the check if the correct split is in the list. That means an example is correct when the correct split is in the list and not correct when the split is not in the list. In the evaluation we also differ between correct with morphemes and correct without morphemes. Correct *without* morphemes means that we are only interested if the splits are at the write position. Correct *with* morphemes also check if the morphemes are set correct. The correct splitting of *Aktionsplan* is *Aktion(s)+plan*. This is correct with morpheme and correct without morphemes, but *Aktions+plan* is not correct with morphemes but correct without morphemes. *Akt+ion(s)+plan* is either not correct with morphemes and not correct without morphemes. The following table shows the results of the evaluation:

Algorithm	**Correct with morphemes**	**Correct without morphemes**
Left to right	0.81	0.89

Over 80% of the 158,653 examples are marked as correct with morphemes and nearly up to 90 % are marked as correct without morphemes. This is a good base for the upcoming ranking algorithm. The ranking algorithm can only return good results if the correct split is also in the ranking list.

The splitting of words can also be visualized. Since the output of the algorithm is a tree, it is easy to show see what has happened. All trees are converted to a `JSON` structure and stored in a CouchDB Database. With the help of JQuery (`http://jquery.com/`) and Javascript InfoVis Toolkit (`http://thejit.org/`) I created a little web application. It can be see on `http://jenshaase.couchone.com/noun_decompounding/_design/tree/index.html`. The code of the web application can be found in `src/main/couchapp`. To install the application you need to install CouchDB and `node.couchapp.js` from `https://github.com/mikeal/node.couchapp.js`. See also figure 2.1 for an example tree.

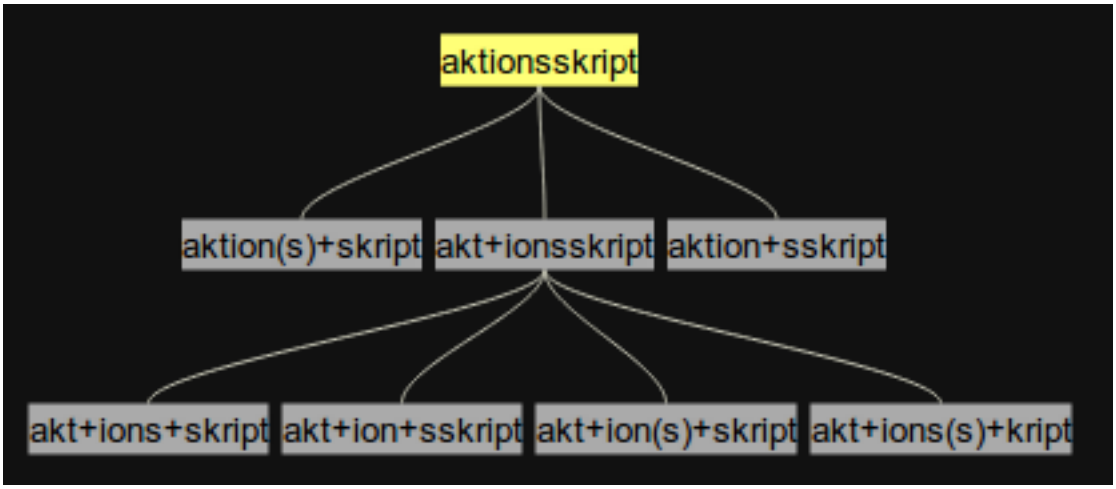

Figure 2.1.: Visualized split of the word *aktionsskript*

Data driven algorithm

As alternative to the left to right splitting algorithm a complete other algorithm was implemented. This algorithm is more data focused. It tries to split the word based on the amount of words that begin and end with the same letters as the word that should be splitted. This idea was first tried by Larson [ea00].

To split a word the IGerman98 dictionary was packed in a Trie (`http://en.wikipedia.org/wiki/Trie`). The trie can now return a number words that begin with the same letters. For example if the trie contains *abc*, *abcde* and *abde* the value of *a* is three, because three words start with *a*, the value for *abd* is *1* because one word starts with this letters. An additional trie is computed with all words in reserve (*abc* -> *cba*). With both tries generated the main algorithm can now start. The next three tables 2.1 , 2.2 and 2.2 show an example (extracted from [ea00]).

f	r	i	e	d	e	n	s	p	o	l	i	t	i	k
-	-	39	29	29	25	24	23	3	1	1	1	1	1	1
1	1	1	1	1	2	7	37	88	89	89	92	99	-	-

Table 2.1.: First extract the trie values for each part. We define that a word has at least three letters.

f	r	i	e	d	e	n	s	p	o	l	i	t	i	k
-	-	-	10	0	4	1	1	20	2	0	0	0	0	0
0	0	0	0	0	5	30	51	1	0	3	7	-	-	-

Table 2.2.: Next we calculated the difference of the neighbors.

For this example the split is made between *friedens* and *politik*; which are correct words. The splitting can be done recursively on each part again until nothing can be splitted anymore. The evaluation for this algorithm works exactly like the evaluation of the left to right algorithm and can be seen in table 2.4.

As we can see the result of this algorithm are not very good compared to the left to right splitting algorithm. It is nearly impossible to use this splitting for the ranking algorithm. But that do not mean that this algorithm is always bad. Depending on your dataset or with another dictionary this can be a helpful alternative.

f	r	i	e	d	e	n	s	p	o	l	i	t	i	k
-	-	-	-		*			*						
							*				-	-	-	-

Table 2.3.: Local maxims are possible splitting points (marked with a asterisks)

Algorithm	Correct with morphemes	Correct without morphemes
Data driven	0.16	0.41

Table 2.4.: Evaluation result data driven splitting algorithm

Conclusion

In the above sections we developed two complete different splitting algorithm. While the left to right algorithm splits a word if a subpart of the word is in the dictionary, the data driven algorithm tries to extract some statistics from the dictionary to find a more natural split. For the current dataset the left to right splitting algorithm works a lot better. The further work only based on this algorithm. But for special dataset the data driven algorithm can be a good alternative.

Algorithm	Correct with morphemes	Correct without morphemes
Left to right	0.81	0.89
Data driven	0.16	0.41

Table 2.5.: Results of splitting algorithm

2.2.3 Ranking algorithms

With the splitting algorithm we are able to create a lot of possible splits for a word. The ranking algorithm now tries to order these splits. The splits with the highest value will be the correct split. The list of possible splits contains also the inital word. If the initial word gets the highest rank no splitting is done.

All three ranking algorithm have to methods of ranking. They can rank a list or rank a tree. Ranking on a list gives all possible splits a weight, while ranking on a tree gives only a subset of all possible splits a weight:

```
function listRanking(list)
  result = List()
  for each (item in list)
    result += rank(item)
  result.order()

  return result(0)


function treeRanking(parent)
  if (parent.isLeaf())
    return parent
```

```
list = parent + parent.children()
res = listRanking(list)

if (res == parent)
  return parent
else
 treeRanking(res)
```

The evaluation of each algorithm checks if the highest ranked value is the correct split. As in the splitting algorithm evaluation we also distinguish between correct with morphemes and correct without morphemes. For the list ranking we also check how many correct results are in the first two and three elements of the ranked list. We call this *Correct@1*, *Correct@2* and *Correct@3*. We must also keep in mind that we can have a maximum of 0.81 with morphemes and 0.89 without morphemes due to the splitting algorithm.

The evaluation of the ranking algorithm was done only on a small subset of examples (10,000), because the Lucene index was very slow. The result of the splitting algorithm on these dataset is the following:

Algorithm	Correct with morphemes	Correct without morphemes
Left to right	0.813	0.888

Table 2.6.: Left to right splitting evaluation on the dataset of 10,000 elements

Frequency based ranking

The first attempt to build a ranking function is a frequency based method as described by Alfonseca [ea08]. For each split S with the words s_i we calculate the geometric mean of the frequencies from the `Web1T` corpus:

$$F_s = \prod_{s_i \in S} freq(s_i))^{\frac{1}{|S|}} \tag{2.1}$$

The `freq` value is calculated with the `Finder` class. This class contains the method `find` which returns a list of n-grams that match the search word. For each n-gram we also get the frequency. The result of `freq` is the sum of all frequencies in the list. The split with the highest F_s will be the highest ranked split.

For the evaluation the command line tool in `bin/frequencyBased.sh` can be used. It can have following parameter:

```
usage: FrequencyBasedRanker
    --limit <arg>          (optional) Limit of results to evaluate
    --luceneIndex <arg>    The path to the web1t lucene index
```

The current evaluation is only done on 10,000 example as mentioned above. For that we can set the `limit` parameter to 10,000. The `luceneIndex` parameter is required and has as argument the path to the generated lucene index. The evaluation of the algorithm is show in table 2.7. The values in brackets are the correct results without morphemes.

As we can see, the result without morphemes are a lot better than the result with morphemes. When we are not interested in morphemes the tree gives us with 0.7204 correct answers the best results but the list with 0.7029 is not far behind. We also can see the increase from Correct@1 to Correct@3 is not the same for the value with/without morphemes. The correct answers with morphemes increase from

Algorithm	Correct tree	Correct@1	Correct@2	Correct@3
Frequency based	0.5226 (0.7204)	0.5078 (0.7029)	0.6653 (0.7847)	0.7263 (0.8291)

Table 2.7.: Evaluation results for the frequency based algorithm

0.578 to 0.7263. That is a difference of 0,2185. The differences for the correct without morphemes is only $0.8291 - 0.7029 = 0.1262$. That means that the results at the second or third place often only differ in the position of the morphemes, but not at the split positions.

Probability based ranking

The second method is also described by Alfonseca [ea08]. For each Split S with the words s_i we calculate

$$P_s = \sum_{s_i \in S} -log(\frac{freq(s_i)}{F}) \tag{2.2}$$

where F is the total amount of frequencies. The split with the lowest P_s will be the highest ranked split.

For the calculation of F the command line tool in `bin/countTotalFreq` can be used with the following options:

```
usage: countTotalFreq
    --index     The path to the web1t lucene index
```

The result is the sum of all frequency values in the Lucene index. This value must be set in the property file `src/main/resources/index.properties` with the property name `frequency`. Example:

```
# src/main/resources/index.properties
frequency = 143782944956
```

As you can see the value can be bigger than the Integer range. In the implementation Java's `BigInteger` class is used.

After setting up the F value the evaluation can be started from the command line (`bin/probabilityBased.sh`). It has the same parameter as the command line tool for the Frequency based algorithm. The evaluation on 10,000 examples returns the result shown in table 2.8.

Algorithm	Correct tree	Correct@1	Correct@2	Correct@3
Probability based	0.1815 (0.2518)	0.1843 (0.2558)	0.5095 (0.6579)	0.6281 (0.7425)

Table 2.8.: Evaluation results for the probability based algorithm

The result are not very good compared to the result of the frequency based method. Both tree and list ranking return only a success rate of 0.18 with morphemes and 0.25 without morphemes. But I we look at the *Correct@2* and *Correct@3* values we see a great increase. This function favors split with a small amount of total split. Here, words with no split or one split will be higher ranked than words with three or four splits. That means the result of this algorithm ca be a good feature for a machine learning algorithm, were several algorithm are combined.

Mutual information based ranking

The previous methods only focus on the individual words and not on the co-occurrence of the words. For that we add the next method, also described by Alfonseca [ea08]. For a word pair w_1 and w_2 we can calculate the mutual information:

$$M(w_1, w_2) = log_2(\frac{\frac{freq(w_1,w_2)}{F}}{\frac{freq(w_1)}{F} \times \frac{freq(w_2)}{F}}) = log_2(\frac{F \times freq(w_1, w_2)}{freq(w_1) \times freq(w_2)}) \tag{2.3}$$

For all word pairs in a split S we can calculate the mutual information and average it. The split with the highest averaged mutual information is the highest ranked split.

Before we can start the evaluation we have to set the F value as the the previous section described. To start the evaluation again the command line tool in `bin/mutualInformationBased.sh` can be used. It has the same parameter as the command line tool for the frequency based evaluation. The results are shown in table 2.9

Algorithm	**Correct tree**	**Correct@1**	**Correct@2**	**Correct@3**
Mutual information based	0.2952 (0.4415)	0.3681 (0.5420)	0.5163 (0.6659)	0.5865 (0.7371)

Table 2.9.: Evaluation results for the mutual information based algorithm

The results of the mutual information based algorithm are a little bit better than the result of the probability based method but not as good as the frequency based method. Also the increase of the *Correct@2* and *Correct@3* values is not so high as in the probability based method.

Conclusion

In total the frequency based algorithm was the best ranking algorithm of all three. The success rate of 72% is not far away from the maximum of 88%. Improving the splitting algorithm can also improve the ranking algorithm. The other two ranking algorithm do not return useful results, but they can be used in combination with some other algorithm in a machine learning based ranking algorithm. Due to a time limit this was not implemented. But the work of Alfonseca [ea08] showed good results.

A current problem for all algorithms is that the time to access the index is very long. This caused the problem that not the complete dataset could be evaluated. Improving the index or using faster hardware is a must for all *real world* problems.

Algorithm	**Correct tree**	**Correct@1**	**Correct@2**	**Correct@3**
Frequency based	0.5226 (0.7204)	0.5078 (0.7029)	0.6653 (0.7847)	0.7263 (0.8291)
Probability based	0.1815 (0.2518)	0.1843 (0.2558)	0.5095 (0.6579)	0.6281 (0.7425)
Mutual information based	0.2952 (0.4415)	0.3681 (0.5420)	0.5163 (0.6659)	0.5865 (0.7371)

Table 2.10.: Results of ranking algorithms

2.2.4 UimaFit Component

To bring the splitting and ranking algorithm together a Uima Compoenent was implemented with the help of the UimaFit library. The component has two Annotator, one for the list based

ranking, called `NounDecompoundingListAnnotator` and one for the tree based ranking, called `NounDecompoundingTreeAnnotator`. Both work very similar, with the difference than one called the tree ranking function and the other the list ranking function.

Before using the component in a analysis engine a tokenizer is needed that creates token annotations on the document. After the tokenizer annotator one of the two above mentioned annotators can be used. Both components have three configuration parameter. With the `index` parameter the path to the lucene index must be set. The `tokenClassName` parameter contains the classname of the token annotation created by a tokenizer. The last and optional parameter is the `ranker` parameter. Here you can set with ranking algorithm you want to use. Default is `FREQUECNY`. Both algorithm use the left to right splitting algorithm with the IGerman98 dictionary.

After processing the annotator on a document a new annotation type is added to the document. For each token the `SplittedToken` type is added. It covers the same text as the token and has as additional information a list of splits elements. Each element can have brackets to identify morphemes. Concatenating the split elements will return the original token.

3 Reflection

3.0.5 Lessons Learned

Time management

Weekly time planning help me to think about the goals for the next week. Most of the time I planned the amount of time correctly or I needed less time. I never came to the point that I planned the time totally wrong. Because of that I think I can understand the complexity of the problems that need to be solve and my own skills of implementation.

Appreciating the time work very good, but doing the things on the day I planned worked not so good. Often I reorganized the day-planning in a week. Most of the time this happened because other work was not done or I don't feel like it. But the efficiency is much higher on the days when I feel very well.

Weekly documentation

Next to the time management also the weekly documentation helped me to get an overview of the next steps and the total project. It was also a very good help to write this report. Some ideas that failed during the project get forgotten very fast. Writing them done in the weekly documentation and the final report, prevents other to do the same mistakes.

Having deadlines for the weekly documentation also helps to reach the goals planned in the current week.

3.0.6 Conclusion

Noun compounding in the German language is a powerful tool to build new words with two or more existing words. The meaning of the new word is not total different. The single words can stand in the same context as the compound word. This is a problem in many NLP task. Search engine for example have to search for the compound and the splitted words to find all relevant examples.

In this work we saw some method to split compound words into base words. In general we take a compound word and try to generate lots of possible splits with the help of a dictionary. Afterwards we rank each split. The split with the highest value is our splitted compound.

To generate possible split we saw two different techniques. The left to right algorithm works best and returns with a rate of 81% a list that contains the correct split. This algorithm can also return a tree instead of list, which gives us the possibility to visualizes the algorithm. We can also use this tree in the ranking function to minimize the calculations.

After developing a successful splitting algorithm, we focus on the ranking algorithms. The base idea for all three developed solution was that splits with more relevant words are correct. This knowledge of what are relevant word comes from the Google Web1T corpus. At the end the frequency based algorithm works best. With a success rate of 72% (without morphemes) we get some useful results. The other two algorithm do not return useful results, but they can be used as features for a machine learning algorithm. Combining all three algorithm in a machine learning algorithm can improve the result, but was not done in this work.

A currently unsolved problem is that splitting a word takes very long because the Lucene index is very slow. Faster hardware can help to resolve this problem, but was not tested.

At the end we have a useful result that can be used in a NLP Task, in form of a Uima component.

A Appendix

A.1 Installation of the visualization tool

The left to right algorithm has simple tool to visualize the splits in a tree. This section shows you how to install this app.

At first need to install all required software.

Warning: CouchDB will be merged with Membase in future. This installation will work with CouchDB 1.0.0+

1. Install `couchdb` from `http://www.couchone.com/get` or install the package `couchdb` on Ubuntu
2. Install `node.js` from `http://nodejs.org/` (tested with version 0.2.3).
3. Install `npm` (Node package manager) as descripted on `http://npmjs.org/`
4. Install `node.couchapp.js` from `https://github.com/mikeal/node.couchapp.js`. See the readme file for more information. This project should also work with newer version of `node.js`.

After all required software is install we can upload the application to `couchdb`. For that we use `node.couchapp.js`.

1. Open a browser and open `http://localhost:5984/_utils/`.
2. Create a database with some name (e.g. “noun_decompound”)
3. Open a terminal a change to the directiory `src/main/couchapp`
4. Execute `couchapp push app.js http://localhost:5984/dbname`. Change `dbname` to what you set in step two.
5. Your application should now run under `http://127.0.0.1:5984/dbname/_design/tree/index.html`. But the database contains now data.

The last step is to update some data to the database.

1. Open a terminal a change the directory to the project
2. Execute `./bin/couchDbExport.sh -limit 20000 -dbname=dbname`. This will generate 20,000 splits an saves them to the database.
3. Reload your the application in the browser. On the left side all generated split should appear. Click on them to see the tree. Have fun!

During the project I uploaded the app to `http://jenshaase.couchone.com/noun_decompounding/_design/tree/index.html`. Use the link to see how should look like.

Bibliography

[ea00] Martha Larson et al. Compound splitting and lexical unit recombination for improved performance of a speech recognition system for german parliamentary speeches. In *Proceedings ICSLP 2000: Sixth International Conference on Spoken Language Processing*, 2000.

[ea08] Enrique Alfonseca et al. German decompounding in a difficult corpus. In *Computational Linguistics and Intelligent Text Processing*, 2008.

[Mar06] Torsten Marek. Analysis of german compounds using weighted finite state transducers. Master's thesis, Eberhard-Karls-Universität Tübingen, 2006.