Karlsruhe Institute of Technology

Approaches to Compound Splitting in German Spoken Term Detection

Bachelor Thesis of

Ge Wu

At the Department of Informatics
Institute for Anthropomatics (IFA)

Reviewer:	Prof. Dr. A. Waibel
Second reviewer:	Dr. S. Stüker
Advisor:	Asst. Prof. Dr. F. Metze (CMU)
Second advisor:	M.Sc. Y. Zhang

Duration: July 1, 2012 – October 31, 2012

This work has been supported by the scholarship of Baden-Württemberg Stiftung. It has largely been produced at the Language Technologies Institute of Carnegie Mellon University within the interACT exchange program.

Abstract

In this work, a Spoken Term Detection system for German speech data has been designed and evaluated. The design and evaluation of the system comply with the standards specified by the NIST 2006 Spoken Term Detection evaluation. Emphasis is placed on the ability to detect out-of-vocabulary words, especially compound words in German language.

The system uses confusion network as input for the detection, which is generated by a speech-to-text system. The confusion network is stored and indexed in a database for on-demand querying. A compound splitting component of the system splits the compound words into basic words for detection. A query expansion component is used to generate more detection candidates for the out-of-vocabulary words. After the detection of the search terms, a decision component decides if each detection should be output in the final result.

Different approaches to splitting compound words and expanding search terms are applied in the system and they are also analyzed and compared with each other in the experiments. The system is tested on detecting spoken terms on German lecture speech data. By using random selected spoken terms from the speech transcripts for evaluation, an Actual Term Weighted Value of 0.7365 is achieved.

Contents

1. Introduction

In our highly developed modern societies, there is an increasing demand of *information*. People need information to make decisions, arrange their own activities or simply know what's going on and get connected to the world. On the other hand, people are also generating, processing and spreading information all the time. In other words, everything is driven by information.

Under this background, the information retrieval (IR) technique has achieved remarkable progress, at least for text-based information. Speech-based information, like audio or other media is different than text-based information in nature. It's may be not as clear as the text, because it can contains noise too. Besides it is also not "random accessible", because it's also very hard to locate the useful information that we are interested in a long speech. So there still remains a difficult problem about how to retrieve useful information in speech.

The speech-based information retrieval covers a range of different topics including key word spotting, topic discovery, spoken document retrieval (SDR), spoken data mining (SDM), etc. In this work, we focus on a fundamental problem under the branch information retrieval: Spoken Term Detection (STD).

Spoken Term Detection aims to retrieve spoken terms from a large volume of speech archives reliably and efficiently [Wan10]. It has strong focus on distinct detections of words including the exact position in the audio. However, it is not necessary to find the most relevant or prominent occurrences [Kol11].

In order to detect the spoken terms in the speech, Automatic Speech Recognition (ASR) in the form of speech-to-text (STT) systems is used to convert the speech into text form. But since the speech may contains noises and the ASR system is also error-prone, so the correctness of the result generated by ASR system must be taken into consideration. Some methods or algorithms may be adopted by the *Term Detector* to reconstruct the spoken terms, which are missed or incorrectly recognized before by the ASR system. Finally the a *Decision Maker* examines each detection and decides if it should be output in the final result depending on its detection confidence. Figure 1.1 shows the general structure of a STD system and a prerequisite ASR system.

1.1. Challenges in German Spoken Term Detection

The so-called out-of-vocabulary (OOV) problem is particularly significant in languages with a rich morphology and active compound such as German [MS09]. Unlike in English,

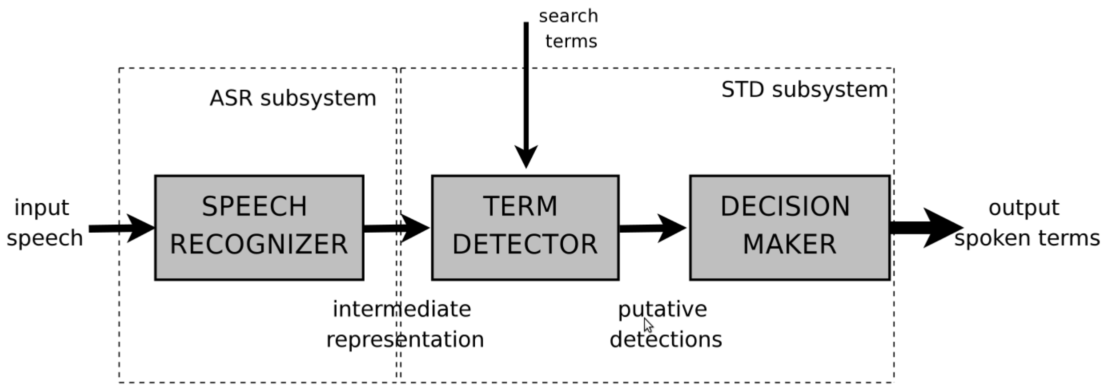


Figure 1.1.: The standard STD architecture [WKFB10]

people use a lot of compound words in German. A Compound word is a word formed by combining several basic words together. In German language, compound words can be combined by basic words in a very arbitrary way and therefore they are generally not included in the vocabulary, because a vocabulary that can hold all the compound words will be extremely large.

Since these words are not in the vocabulary, the ASR system can not recognize them and put them in the output, so the STD detection can not detect them too. Under this background the commonly used compound words in German raise the challenge: how to make the STD system more suitable for German language?

1.2. Objectives of This Work

The main objective of this work is to design and evaluate a STD system that aims at German language. Due to the importance and frequency of compound words in German language, different approaches are designed to handle them. Besides the treatment of compound words, other OOVs are also taken into consideration to improve the detection result. At the end different approaches, which are used to improve the STD system, are evaluated and compared with each other.

1.3. Scope of This Work

This work focus on the STD task on German language. Different confusion networks may be used as input to evaluate the detection systems, but in all cases, an ASR system is always presumed as a prerequisite component to generate the confusion networks. The ASR system is not part of the STD system and is therefore not discussed or analyzed in this work.

1.4. Structure of This Thesis

In chapter 2, the NIST 2006 Spoken Term Detection Task is described and the STD system for English language is introduced.

In chapter 3, the basic architecture of the German STD system and the general consideration of experimental design are introduced.

In chapter 4, the different approaches to splitting compound words are discussed in detail.

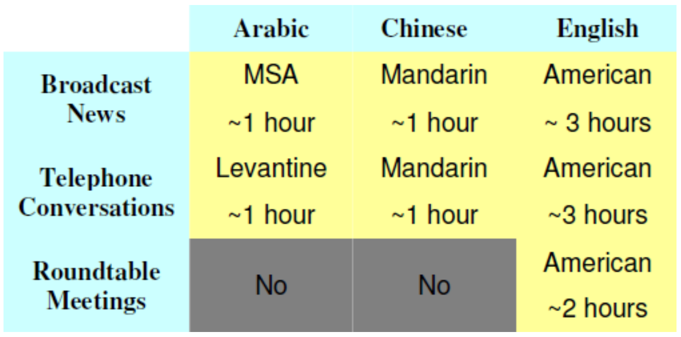
In chapter 5, the experimental results and evaluation are presented. The different approaches to improving the detection result are compared, especially the approaches to splitting compound words.

2. Background and Related Work

2.1. The NIST 2006 Spoken Term Detection Task

In 2006, the National Institute for Standards and Technology (NIST) established the task of Spoken Term Detection. According to the evaluation plan of the STD task, it can be simply summarized as “find all of the occurrences of a specified ‘term’ in a given corpus of speech data” [Nat06].

The speech data provided by NIST cover three different languages and three different source types. The three languages are English, Arabic and Mandarin Chinese, the three source types are Broadcast News (BN), Conversational Telephone Speeches (CTS) and Conference Meetings (CONFMTG) respectively. The speech durations of different language and source type varies. Table 2.1 lists the languages, source types and durations of the audio speech data.



	Arabic	Chinese	English
Broadcast News	MSA ~1 hour	Mandarin ~1 hour	American ~ 3 hours
Telephone Conversations	Levantine ~1 hour	Mandarin ~1 hour	American ~3 hours
Roundtable Meetings	No	No	American ~2 hours

Table 2.1.: Langauge/Source Type pairs to be tested and the durations of audio speech data [Nat06]

There are two sets of search terms, both contain approximately 1000 widely selected terms per language. The search terms include both single-word and multi-word terms and are presented in the language’s native orthography. They are also encoded differently according to the languages: ASCII for English, UTF-8 for Arabic, and GBK for Mandarin Chinese. For an occurrence of a multi-word term, the time gap between each two successive words should be less than a specific value. In the 2006 evaluation this value was set as 0.5 second.

NIST defined the Actual Term Weighted Value (ATWV) as a major metric for the system evaluation, which is shown in formula 2.1.

$$ATWV(\mathcal{T}) = 1 - \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} (P_{Miss}(t) + \beta \cdot P_{FalseAlert}(t)) \tag{2.1}$$

where P_{Miss} is the miss probability of the detection and and $P_{FalseAlert}$ is the false alert probability of the detection. The weight β is a term-independent coefficient used to cooperate the two probabilities, since P_{Miss} is generally much more bigger than $P_{FalseAlert}$. In the 2006 NIST evaluation, it was chosen as 999.9 [Kol11, Nat06].

NIST also defined the Maximum Term Weighted Value (MTWV) as another metric used to assist the evaluation, which is shown in formula 2.2.

$$MTWV(\mathcal{T}) = \max_{\theta \in [0,1]} (P_{Miss}(t, \theta) + \beta \cdot P_{FalseAlert}(t, \theta)) \tag{2.2}$$

where a global threshold θ is used to measure the miss and false alert probability [Kol11, Nat06].

Generally, higher ATWV or MTWV corresponds to better system output: less misses and less false alerts.

NIST also defined Detection-Error-Tradeoff (DET) curves to characterize the general detection performance. The curves are computed as “a function of language and source type as well as for various selections of data and terms” [Nat06].

NIST suggested the system structure presented in figure 2.1 for the STD task.

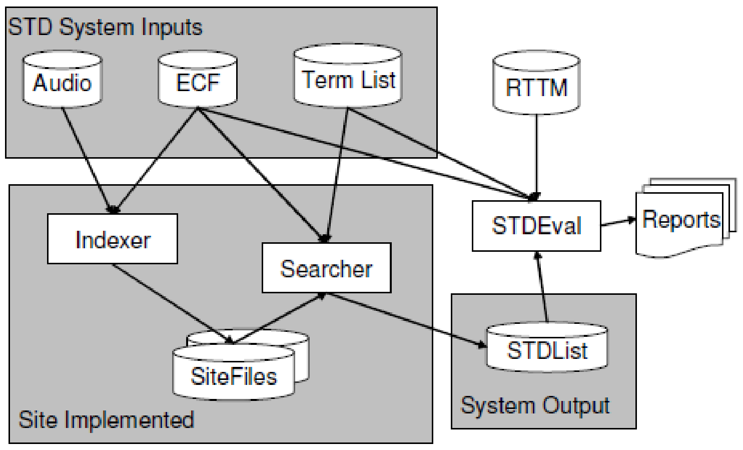


Figure 2.1.: System and evaluation inputs and ouputs [Nat06]

The system suggested by NIST has two major components: the *Indexer* and the *Searcher*. The *Indexer* reads the audio data and the Experiment Control File (ECF) to pre-process the data and store them in Site Files (e.g. databases). The *Searcher* reads the ECF, the pre-processed data as well as the search terms from term list and then generate the STD List, where the detection result is stored. *STDEval* is the evaluation tool provided by NIST, which is not part of STD System from the participants. It compares the STD result list of the participants with the Rich Transcription Timer Mark (RTTM) file, evaluates the system and generates the final reports.

The system has a strict two-phase structure, which means, the *Indexer* has no knowledge about what terms will be detected later and the *Searcher* has no access to the audio data.

All the data that the *Searcher* obtains are the pre-processed data in *Site Files* and the experiment control information in the *ECF* files.

The *RTTM* file is basically the transcripts in a specific file format, in which the beginning time and duration of each word is given. The *ECF* files define "the excerpts within audio files to be used for specific experiments and the language/source type of each file" [Nat06]. The *Term List* and *STD List* are both presented in the Extensible Markup Language(XML) format.

A lot of research institutes participated the NIST 2006 STD evaluation. Part of evaluation result is presented in table 2.2.

	Broadcast News		Telephone		Meeting	
Participant	ATWV	WER	ATWV	WER	ATWV	WER
BBN	–		**0.8335**	14.9%	–	
IBM	**0.8485**	12.7%	0.7392	19.6%	0.2365	47.4%
SRI	0.8238	23.2%[a]	0.6652	17.4%	**0.2553**	44.2%
TUB	0.3890	30.3%	0.1598		0.0500	60.3%

Table 2.2.: Results of the NIST 2006 English STD evaluation and corresponding word error rates on the development data [Kol11]

IBM, BBN and SRI achieved the best result in Broadcast News, Conversational Telephone Speeches and Conference Meetings respectively. One thing noticeable is, high Word Error Rate (WER) doesn't necessarily leads to low ATWV. For example, the system from IBM has higher WER for the CTS data than SRI, but has also achieved higher ATWV result.

2.2. System from English STD Evaluation

Henrich Korlkhorst designed a STD system and tested it with the NIST 2006 STD evaluation task. His system is originally intended to be used for English STD task. Its basic structure is depicted in figure 2.2.

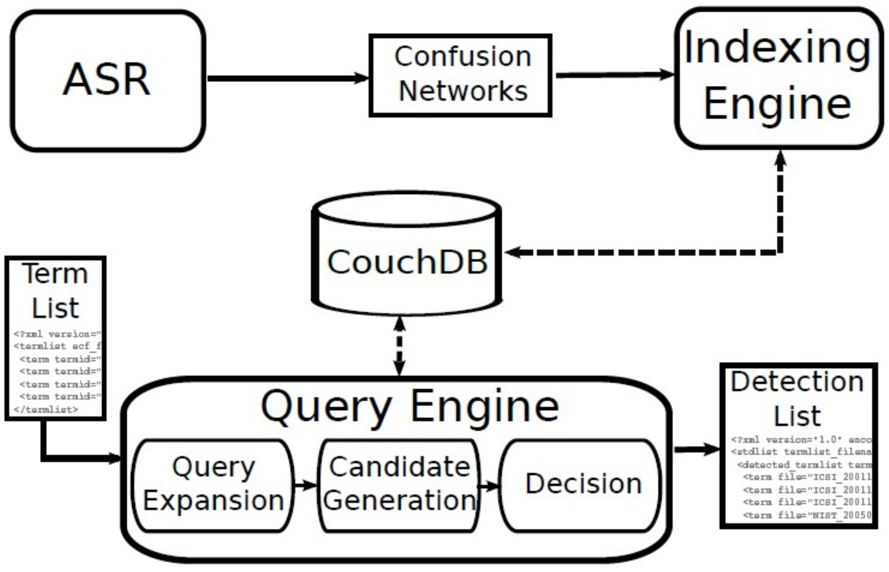


Figure 2.2.: Basic setup of STD system with major components [Kol11]

The system uses *Confusion Networks* as system input, which are created through ASR system. The confusion network has a lattice-based structure. Each lattice presents a time interval, in which several hypotheses with their corresponding probabilities are given. A confusion network also contains other information like the beginning and ending time of each lattice and the best hypothesis in each lattice. A detailed description of confusion networks can be found in [MBS00]. Figure 2.3 shows a brief sketch of confusion networks.

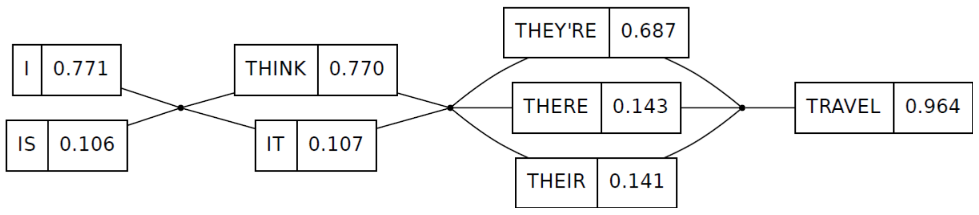


Figure 2.3.: Exemplary Confusion Network [Kol11]

The *Indexing Engine* of the system reads data from the confusion networks, and stores them in the CouchDB database for searching later.

The search terms are read from the *Term List*, The searching processes for in-vocabulary (IV) words and out-of-vocabulary words are different. IV queries are directed searched by looking up the database. OOV queries will first be expanded by the *Query Expansion Component*, which means using similar words from the vocabulary to replace the queries for searching. The definition of similar words varies according to different criteria, such as orthographic similarity or pronunciation based similarity. And then a list of similar words are generated as query candidates and each candidate will be searched in the database. Finally the *Decision Component* makes the decision about which detections are treated as real occurrences of the OOVs. All the results about detections and decisions are stored in the *Detection List* in the XML file format.

The system of Henrich uses several different methods to expand the queries by using different criteria about similarity. And these methods achieved different ATWV results in his experiments. The first type of similarity is based on orthography, i.e. the spelling of words. One option is the Levenshtein similarity, which is defined by formula 2.3. In the formula, $Len(w_i)$ means the length of word w_i and $Dist_{Lev}(w_i, w_j)$ means the Levenshtein distance between word w_i and word w_j.

$$Sim_{Lev}(w_1, w_2) = \frac{Len(w_1) + Len(w_2) - 2 \times Dist_{Lev}(w_1, w_2)}{Len(w_1) + Len(w_2)} \tag{2.3}$$

Another option is the Dice coefficient, which is defined in formula 2.4. In the formula, B_i means the set of bigrams for w_i.

$$Sim_{Dice}(w_1, w_2) = \frac{2 \times |B_1 \bigcap B_2|}{|B_1| + |B_2|} \tag{2.4}$$

The system also uses pronunciation-based similarity for the query expansion. The pronunciations of the OOVs are predicted by using speech synthesis tool such as Festival, and a dynamic programming based algorithm is used to compare the predicted pronunciation with the pronunciations of words in the vocabulary. The calculation of the similarities involves using the confusion matrices, which specifies the substitution probabilities of phones.

The last step of the spoken term detection system is the binary decision. In this step all the detection candidates are decided to be a final detection or not depending on its detection score, which is calculated by multiplying the ASR confidence $P(c)$ and query expansion similarity $Sim(w, c)$. The calculation of the score is shown in formula 2.5.

$$Score(c) = P(c) \cdot Sim(w, c) \tag{2.5}$$

The system uses two types of thresholds to make the decisions. First one is the global threshold: all the detection candidates with score better than a certain global threshold will be accepted. The other one is the so called term-specific threshold. It's designed to improve the ATWV and for each search term. The calculation of the adapted threshold is shown in formula 2.6.

$$\theta(t) = \frac{\beta \cdot N_{true}(t)}{T_{speech} + (\beta - 1) \cdot N_{true}(t)} \tag{2.6}$$

In the formula, T_{speech} is the total length of the speech in seconds, β is the weight used to balance the miss detection probability and false alert probability, which was also introduced in formula 2.1, and $N_{true}(t)$ is the number of real occurrences of term t, which can be estimated by "the sum of the candidates' scores" [Kol11][MKK$^+$07].

3. System Setup and Experimental Design

Although the NIST 2006 Evaluation and the system from Henrich are not designed for German language, it's not hard to adapt them to the German STD task. Compound is very common phenomenon in German language and therefore also a core problem we need to solve in order to improve the STD systems for German. The difficulties in detecting compound words is that, they can be composed by putting random basic words successively together in an arbitrary way and the compound word itself can normally not be found in the vocabulary. In order to detect these compound words, they need to be split into basic words at first.

The words contained in the search terms can basically be classified as in-vocabulary words or out-of-vocabulary words. Compound words in the search terms are generally also OOVs but since this work puts a particular emphasis on compound words, the OOVs are further divided into compound OOVs and other OOVs. Therefore the words in search terms can be divided into categories presented in figure 3.1.

$$\begin{cases} \text{IVs} \\ \text{OOVs} \begin{cases} \text{compound OOVs} \\ \text{other OOVs} \begin{cases} \text{company and proper names} \\ \text{words with text normalization(acronyms, initialisms)} \\ \text{foreign or misspelt words} \\ \text{words using wildcards or other unsupporte query syntax} \\ \text{other words(rare words, rude words)} \end{cases} \end{cases} \end{cases}$$

Figure 3.1.: IVs, compound OOVs and other OOVs [LMT^{+}96]

3.1. System Structure

Our detection system for German language is very similar to the system used for the NIST task. Confusion networks are also used as a prerequisite component for our system and they are generated by Janus ASR system[1]. The major difference is in the Query Engine, here a *Compound Splitting Component* is used to handle the compound words in term list. The generated search candidates include not only the query expansions from *Query Expansion Component* but also splitting results from *Compound Splitting Component.* Figure 3.2 shows the structure of the STD system.

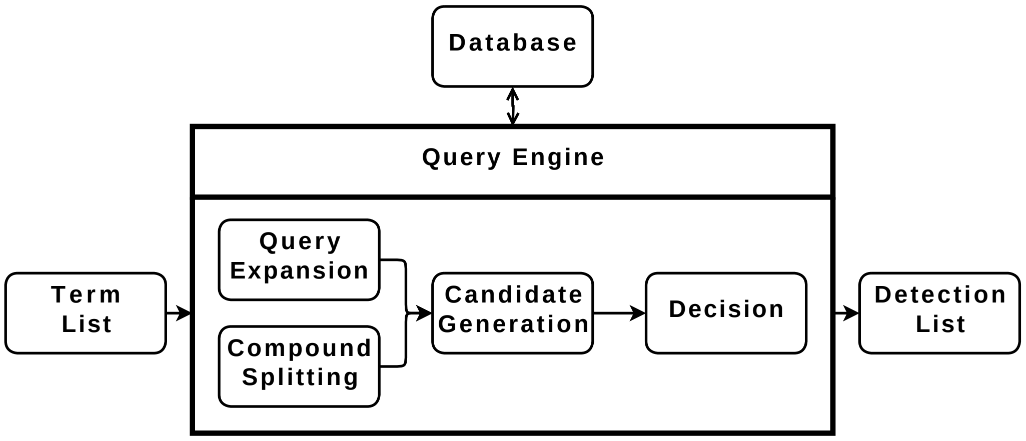


Figure 3.2.: Structure of the STD System

3.2. Compound Splitting Component

The basic idea to split component words is trying out all the sensible splitting possibilities and check if every word segment corresponds a word in the vocabulary. The reason for this "correspondence" is that not every compound word only uses the original form of a word as a part of it. It's quite often that a word is slightly modified before it is put in a compound word. An example is the word "Esszimmer". It can be split into two parts: "ess" and "zimmer", but apparently the former part is not a proper word and may not be found in the vocabulary. Figure 3.3 demonstrates how the word "Startmenge" is splitted.

Plural and genitive case are the most used word forms to create compound words in German language, but other forms are also possible. And the interfix *s* is also frequently used between words. In order to restore the splits of compound words into words which can be found in the vocabulary, there are two different strategies. First one is by using the word formation rules: in most cases the interfixes *s*, *es*, *n*, *en*, *er* and *e* are inserted between basic words. The other strategy is to check if there's any words in the vocabulary which are close to the word segments. We can defined some thresholds about how close the word should be. The threshold can be the maximal number or percentage of letters that differ between a word segment from compound word and a word found in vocabulary.

Normally the algorithms generate a lot of splitting results. In order to find the compound word in the confusion network without brutally examining all the results, the criterion in formula 3.1 is designed to select good splitting results for searching.

$$Quality(S) = (\prod_{p_i \in S} count(p_i))^{\frac{1}{n}} \tag{3.1}$$

In other words, to calculate the quality of a splitting result, we multiple the word count of each word segment and then calculate its nth root. The word count is calculated from the confusion network. The insight for that is, only if a word occurs in confusion networks, it makes sense to search for it, and the more frequently the word segments occur, the bigger the possibility that the splitting is correct [KK03].

[1]http://isl.ira.uka.de/english/1406.php

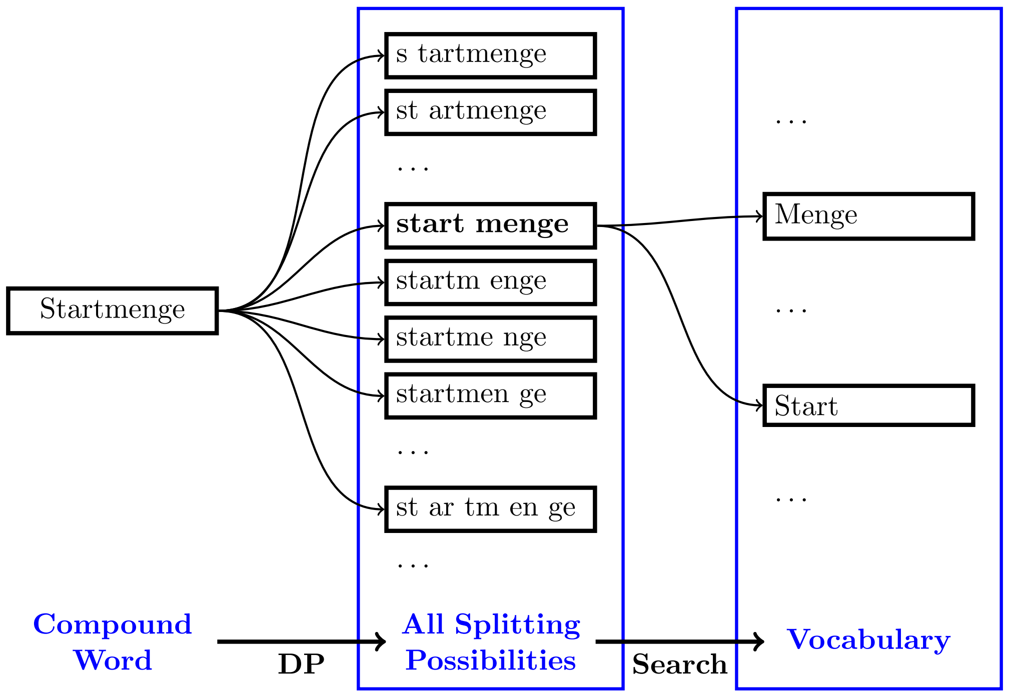


Figure 3.3.: Demonstration of splitting word “Startmenge”

3.3. Query Expansion Component

As the compound splitting component handles the compound words, the query expansion component handles other OOVs. In the experiments we also used the methods which were used by Henrich to see how they improve the detection results. The query expansion methods involved are based on Levenshtein distance, Dice coefficient and pronunciation respectively. The pronunciations of the OOVs are generated by the German version of speech synthesis tool Festival.

3.4. Decision Component

The decision component is used to decide if a detection candidate is treated as a final detection. The threshold used to make the decisions can be configured: global threshold or term-specific threshold. We also try to use different types of thresholds in our experiments to test how it influences the detection results.

3.5. Adding OOVs to ASR System

Another strategy to improve the detection result is to add OOVs to the ASR system and run the recognizer again. In this way the original OOVs are presented as normal words in the new confusion networks and can be found easily by the detection system and the ATWV will also improved.

3.6. Summary

In this chapter, the general structure of spoken detection system and the basic consideration of experimental design were introduced. The detailed compound splitting algorithms will be further talked in the next chapter. And the experimental results will be presented in chapter 5.

4. Algorithms for Compound Splitting

This chapter describes different compound splitting algorithms in detail. Depending on the way how we check if a certain word segment corresponds a word in the vocabulary, two principally different approaches are introduced. One approach is based on word formation rules and introduced in section 4.2, the other approach is based on word similarity and introduced in section 4.3. Furthermore, the data structure BK-Tree is also introduced in section 4.3, which can be used to accelerate searching Levenshtein-similar words in the vocabulary. By using BK-Tree we get a more efficient implementation of the approach based on word similarity.

Section 4.1 describes the basic process to split a compound word.

In section 4.4, some implementation issues are discussed and the time and space complexity of the algorithms are analyzed in theory.

In order to keep the description in this chapter clear, we define some concepts explicitly. A **string** mentioned here is a sequence of letters, it doesn't have to be an existing word in the vocabulary. And a **basic word** always indicates a word that exists in the vocabulary. A **compound word** is the word to be split into basic words, it's generally an OOV. A **sub-word** or **word segment** means a continuous part of a compound word.

4.1. Basic Processes

The basic idea of the compound splitting algorithms is to split the long compound words into sequences of sub-words, which are included in the vocabulary as basic words.

The algorithms use basic dynamic programming technique to check if a substrings of compound words can be split into word sequences. Meanwhile the splitting schemes are saved for later backtracking. The backtracking procedure generate a list of splitting candidates. And the best splitting candidates are chosen according to the selection criterion described by equation 3.1.

The general procedure for calculating the compound splitting is presented in figure 4.1.

```
Input word

splitable[0] := true
splitable[1..word.length] := false
split_schemes[1..word.length] := empty_list

For i:=1 To word.length Do
    For j:=1 To i-1 Do
        If splitable[j-1] Then
        Begin
            splitable[i] := true
            list := vocab.lookup(word[j..i])
            If Not list.empty Then
                split_schemes[i].save(j, list)
        End
```

Figure 4.1.: Pseudocode for splitting compound words

The input is the word to split. The variable *splitable* is boolean array indicates if the substring from the beginning to a specific position can be split into any word sequences. The global variable *split_schemes* is an array of lists, in which the splits are stored. Each element of the list has two fields: *j* and *list*. The field *list* stores a list of words from vocabulary, which are similar to the split of the word started at position *j* and ended at position *i*. The global variable *split_schemes* is used for backtracking the splitting candidates.

The source code can be easily adapted to some specific requirements such as to limit the minimal or maximal length for each sub-word, to control the way to look up a word in the vocabulary or to add a filter to remove some unnecessary sub-words from the list.

The general procedure for backtracking the splitting candidates is presented in figure 4.2.

```
Procedure Backtrack(pos:Integer; splits:Array)
Begin
    If pos=0 Then
        candidates.add(splits)
    Else Begin
        split_scheme[pos].Foreach (j, list) Do
            list.Foreach (sub_word) Do
                Backtrack(j-1, splits + sub_word)
    End
End
```

Figure 4.2.: Pseudocode for selecting splitting candidates

The parameter *pos* is an integer which indicates that, the substring of the word from the beginning to the position *pos* should be backtracked for splitting candidates in the procedure. And the parameter *splits* is an array of strings, which stores the split segments of the substring after position *pos*.

If the parameter *pos* equals zero, it means, the word is completely split and all the segments are store in the parameter *splits*. Then it is added to the global variable *candidates*, which stores all the splitting candidates. If the parameter *pos* doesn't equals zero, all the segments from all the lists which are stored in *split_scheme[pos]* together with the beginning position *j* will be tested for the splitting through the recursive call of the procedure *Backtrack*.

Afterwards, the most similar splitting results in the global variable *candidates* will be chosen. The criterion for similarity can vary and affects the final results. And this is exactly the problem how we look up a word in the vocabulary. In the experiments, different methods are adopted and evaluated. The methods are described in the next sections.

4.2. Algorithm Based on Word Formation Rules

When the basic words are used to compose compound words, sometimes the basic words are directly used as part of the compound word, for example:

Hochhaus : hoch + Haus	(**skyscraper** : high + house)
himmelblau : Himmel + blau	(**azure** : sky + blue)
Kohlsuppe : Kohl + Suppe	(**cabbage soup** : cabbage + soup)

Each sub-word above is also a word by itself that can be directly found in the vocabulary. Corresponding English translation is on the right side.

But sometimes words are slightly modified before being used to compose other words, for example:

Esszimmer : essen + Zimmer	(**dining room** : eat + room)
Gänsehaut : Gans + Haut	(**goose bumps** : goose + skin)
keinesfalls : kein + Fall	(**in no case** : no + case)
Sitzungssaal : Sitzung + Saal	(**conference room** : conference + room)
Schweigeminute : Schweigen + Minute	(**moment of silence** : silence + minute)

Some common situations are: the previous word appears as word stem, like the first example above; the previous word appears as plural form, like the second example above; the previous word appears as the genitive form, like the third example above; interfix *s* is put after the previous word, like the fourth example above; the word stem of verbs and together with interfix *e* is used as previous word, lie the fifth example above. Some other special cases are also described in [Lan98].

In **Algorithm I**, we use six most commonly used word formation rules to split compound words, i.e. to put the interfix *s*, *es*, *n*, *en*, *er* or *e* between two successive words. If a word segment can be formed as a word from the vocabulary followed by one of the interfixes mentioned above, then it will be treated as a valid word segment. And for the implementation, a hash table is used for word lookup to achieve efficient performance.

4.3. Algorithm Based on Word Similarity

Splitting compound words based on word formation rules has some inherent defects. The rules are very hard to conclude completely and new rules need to be added with emergence of new created words.

The other idea is to look for similar words of the word segment in the vocabulary. By similar words, the Levenshtein distance is adopted in the experiment. If the Levenshtein distance between a word from the vocabulary and a sub-word of the compound word is small enough, the two sequences are then treated as similar. This method can cover a lot of situations, how a compound word is formed, without knowing the word formation rules. But this method may also be slower because finding similar words based on Levenshtein distance can not be simply implemented by efficient data structure such as hash table.

In the following parts, algorithms about how to look for Levenshtein-similar words in the vocabulary are described.

4.3.1. Primitive Idea

The most primitive idea is to enumerate every words in the vocabulary and calculate the distance between this word and the given string. If the distance is smaller than some threshold, then the word will be chosen. By implementing this idea we get **Algorithm II**. The speed of this algorithm depends on how fast the distance can be calculated and how large the vocabulary is. Because the vocabulary may contain up to several million words, this algorithm will be very inefficient.

4.3.2. Using Efficient Data Structure: BK-Tree

BK-Tree is data structure suggested by Walter Austin Burkhard and Robert M. Keller for searching file names according to some discrete metric spaces [BK73]. It can be used to accelerate the lookup of Leveshtein-similar words enormously when the required distance is not too large.

BK-Tree is a tree-like data structure. Figure 4.3 shows a BK-Tree with seven nodes.

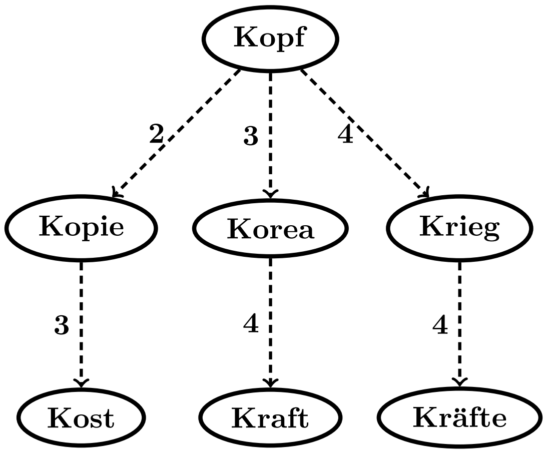


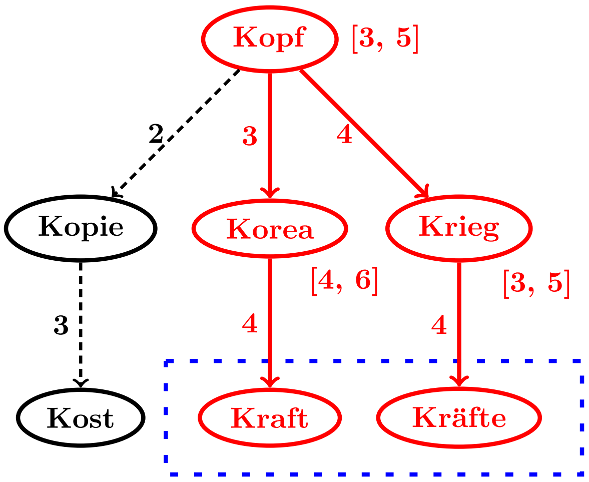
Figure 4.3.: Demonstration of BK-Tree

BK-Tree has one root node and each node may or may not have sub-nodes. The number of sub-nodes for each node can vary and all sub-nodes for each node are numbered. Each node present a word in the vocabulary. The word presented by the node in the ith sub-node of a specific node has a Levenshtein distance i with the word presented by this specific node.

For example, in the BK-Tree depicted above:

$$\begin{aligned}
Dist_{Lev}(Kopf, Kopie) &= 2\\
Dist_{Lev}(Kopf, Kost) &= 2\\
Dist_{Lev}(Kopf, Korea) &= 3\\
Dist_{Lev}(Kopf, Kraft) &= 3\\
Dist_{Lev}(Kopf, Krieg) &= 4\\
Dist_{Lev}(Kopf, Kräfte) &= 4\\
Dist_{Lev}(Kopie, Kost) &= 3\\
Dist_{Lev}(Korea, Kraft) &= 4\\
Dist_{Lev}(Krieg, Kräfte) &= 4
\end{aligned}$$

The figure 4.4 demonstrates the searching process for the string “Krafte” in BK-Tree. The intervals beside the nodes indicate the branches that are searched.



$$\begin{aligned}
Dist_{Lev}(Krafte, Kopf) &= 4\\
Dist_{Lev}(Krafte, Korea) &= 5\\
Dist_{Lev}(Krafte, Kraft) &= 1\\
Dist_{Lev}(Krafte, Krieg) &= 4\\
Dist_{Lev}(Krafte, Kräfte) &= 1
\end{aligned}$$

Figure 4.4.: Demonstration of searching process in BK-Tree

The searching process begins at the root of the tree. We calculate the Levenshtein distance between the search string “Krafte” and the word in the root “Kopf” and get the result 4. And in the next step, we keep searching the sub-trees of the root in the interval $[4-1, 4+1] = [3, 5]$ recursively.

A BK-Tree corresponding to a given vocabulary is build in the preprocessing phase. To look for similar words to a specific word in the vocabulary, the tree will be scanned, and certain information about the Levenshtein distance stored in the tree can be used to limit the number of nodes to be checked. In this way, a lot of time for checking the words is saved and the algorithm can achieve better efficiency.

The Levenshtein distance is a non-negative integer function defined on two strings. It forms a *metric space*, because it satisfies the following three axioms:

$$Dist_{Lev}(s_1, s_2) = 0 \quad \Leftrightarrow \quad s_1 = s_2 \tag{4.1}$$
$$Dist_{Lev}(s_1, s_2) = Dist_{Lev}(s_2, s_1) \tag{4.2}$$
$$Dist_{Lev}(s_1, s_3) \leq Dist_{Lev}(s_1, s_2) + Dist_{Lev}(s_2, s_3) \tag{4.3}$$

where s_1, s_2 and s_3 are any possible strings. The third axiom is often referred as *triangular inequality.*

We specify a variable K, which means we want to find all the words that have a maximal Levenshtein distance K to a specific string s. The variable K is given as a threshold. We start the searching process at the root node. If the word stored at the root node is w_r and it have a Levenshtein distance $Dist_{Lev}(s, w_r)$, then we only need to check all the branches numbered with i that satisfies

$$Dist_{Lev}(s, w_r) - K \leq i \leq Dist_{Lev}(s, w_r) + K$$

Because all the words w' stored in the branches numbered from 0 to $Dist_{Lev}(s, w_r) - K - 1$ has a Levenshtein distance

$$\begin{aligned} Dist_{Lev}(s, w') &\geq Dist_{Lev}(s, w_r) - Dist_{Lev}(w', w_r) \qquad (*) \\ &\geq Dist_{Lev}(s, w_r) - (Dist_{Lev}(s, w_r) - K - 1) \\ &\geq K + 1 \end{aligned}$$

to the query string s. And all the words w' stored in the branches numbered with $Dist_{Lev}(s, w_r) + K + 1$ and above have a Levenshtein distance

$$\begin{aligned} Dist_{Lev}(s, w') &\geq Dist_{Lev}(w', w_r) - Dist_{Lev}(s, w_r) \qquad (*) \\ &\geq (Dist_{Lev}(s, w_r) + K + 1) - Dist_{Lev}(s, w_r) \\ &\geq K + 1 \end{aligned}$$

to the query string s. The deductions marked with $*$ are followed from the formula 4.3.

For the rest of the searching process, we use the same method described above: every time when we reach a sub-tree, we compare the query string with the string stored in the root, so and forth, until there is no branches to search any more. In this way, the number of branches of the BK-Tree to be searched is limited and all the possible similar words are checked. Time will be reached without sacrificing the correctness.

We build the BK-Tree for a given vocabulary in the pre-processing period. We first choose a random word from the vocabulary and make it the root of the BK-Tree. And then we choose another random word, calculate its Levenshtein distance to the word in the root node and insert it into the right branch. Every time we choose a random word from the vocabulary, which hasn't be added to the BK-Tree, calculate its Levenshtein distance to the related words along the path until we find the right position for this word in the BK-Tree.

By using BK-Tree for similar word lookup, we get a more efficient implementation of Algorithm II. Here we number it **Algorithm III** and all the three algorithms will be evaluated in next chapter 5.

4.4. Implementation and Computational Complexity

4.4.1. Similarity Threshold

For the purpose of algorithm analysis, we use two kinds of thresholds to restrict the allowed Levenshtein distance between each sub-word of the compound word and its replacement from the vocabulary.

We use the variables K and R to indicates the two thresholds. The variable K means the allowed number of letter changes, which include insert, remove and substitution of one letter. When we use a word from the vocabulary to replace part of the compound word, the Levenshtein distance between the two strings should not exceeds K. And the variable R means the allowed ratio of letter changes. When we use a word from the vocabulary to replace part of the compound word, the Levenshtein distance between the two strings divide the length of the replaced part should not exceeds R. We use the two thresholds to ensure the replacement doesn't deviate too far away from the replaced string.

For example, regarding the following splitting:

$$\begin{aligned} \text{Blätterwälder} &\Rightarrow \text{Blätter} + \text{Wälder} && \text{(split)} \\ &\Rightarrow \text{Blatt} + \text{Wald} && \text{(replace)} \end{aligned}$$

with the presumption that the vocabulary doesn't contain the plural form of word *Blatt* and *Wald*. The word *Blätterwälder* is split into two parts: *Blätter* and *wälder*. The correct replacements for each part are the word *Blatt* and *Wald* from the vocabulary. For the sub-word *Blätter*, three modifications are made to change it into *Blatt*, including one substitution(letter *ä* to letter *a*) and two deletions(letter *e* and letter *r* respectively). And the ratio of letters changed for the part *Blätter* is then $3/7 \approx 0.429$. For the part *Wälder* the number and ratio of letter changed are respectively 3 and $3/6 = 0.5$. So if the thresholds K and R are set at least 3 and 0.5 respectively, then the word can be correctly split.

4.4.2. Time and Space Complexity

Algorithm I based on word formation rules is basically a searching process in the vocabulary. We use the variable N to indicate the number of words in the vocabulary and the variable L to indicate average word length. We presume the number of word formation rules we use is basically a constant. If we use a balance search tree to implement the word lookup, then the time complex for one lookup is $O(\log(N) \cdot L)$, where L comes from the string comparison. If we use a hash table to implement the word lookup, then the time complexity is $O(L)$. The whole dynamic programming based algorithm for compound splitting contains two loops of length L, therefore the total time complexity for splitting one compound word can reach $O(L^3)$, if hash table is used. And the space complexity is $O(N \cdot L)$, which is the space needed for storing the vocabulary.

Algorithm II is based on examining the Levenshtein distance of all the words to the string to be looked up. The time complexity is $O(N \cdot L^4)$, where N comes from enumerating all the words and L^4 comes from the two loops in the dynamic programming and calculating the Levenshtein distance between two strings, which is actually also based on dynamic programming. The Space complexity is also $O(N \cdot L)$, since the vocabulary is basically the only thing that consumes space.

We use a trick to make Algorithm II more faster: we store the words in different buckets according the their length. If we want to query all the words within distance K to a string s, then we only need to examine all the words in the buckets with length from

$s.length - K$ to $s.length + K$. In this way the number of words to check is restricted. But the time complexity is hard to estimate, since the words are not distributed homogeneously according to their length. The space complexity stays the same.

Algorithm III uses the BK-Tree to store the words to accelerate the searching process. The time complexity is a function of the required distance K. If the K equals 1, the time complexity is about $O(N^{0.639} \cdot L^4)$. If the K equals 2, the time complexity is about $O(N^{0.822} \cdot L^4)$. And if the K equals 3, the time complexity is about $O(N^{\alpha} \cdot L^4)$, where α equals 1.0 approximately. The L^4 of the time complexity comes from the two loops in the dynamic programming and calculating the Levenshtein distance between two strings.

In Algorithm III we can also use the trick introduced before to accelerate the algorithm, i.e. store words according to length and build different BK-Tree for words with length within certain range. But the space complexity will increase, since each word is stored several times in different BK-Tree. And the time complexity is also hard to estimate for the same reason here.

A comprehensive list of time and space complexity for different algorithms is presented in table 4.1.

			Time Complexity	Space Complexity
Algo I			$O(L^3)$	$O(N \cdot L)$
Algo II	no buckets		$O(N \cdot L^4)$	$O(N \cdot L)$
	use buckets		$O(N \cdot L^4)$	
Algo III	no buckets	$K = 1$	$O(N^{0.639} \cdot L^4)$	$O(N \cdot L)$
		$K = 2$	$O(N^{0.822} \cdot L^4)$	
		$K = 3$	$O(N^{\approx 1} \cdot L^4)$	
	use buckets	$K = 1$	$O(N^{0.639} \cdot L^4)$	$O(K \cdot N \cdot L)$
		$K = 2$	$O(N^{0.822} \cdot L^4)$	
		$K = 3$	$O(N^{\approx 1} \cdot L^4)$	

Table 4.1.: Time and space complexity of different algorithms

where "use buckets" means we store the words respectively according to their length and we only check words with certain lengths while searching. "no buckets" means we store all the words together, a search process examines all the words. Part of the result, i.e. the number of comparisons in BK-Tree comes from [ByN98].

4.5. Summary

In this chapter, three algorithms were designed and described in detail: Algorithm I uses the word formation rules to split the compound words, Algorithm II searches for similar words in the vocabulary and Algorithm III uses BK-Tree to accelerate the word search. All the three algorithms are evaluated in chapter 5.

5. Experimental Results and Evaluation

In this chapter, the experimental results and evaluation are presented. Two separate groups of experiments were conducted. One group is designed for evaluating the performance of STD system on German data, which is described in section 5.1. The other group is designed for evaluating the performance of different compound splitting algorithms, which is described in section 5.2. In each section, the basic experimental setup is introduced first and then followed by the experimental results and evaluation.

5.1. Spoken Term Detection Task

Aim of the experiments presented in the section is to evaluate the performance of different STD systems on German data. Different results are achieved by using different methods including: compound splitting, query expansion, changing threshold and recognizing the German audio data with OOVs added into the STT system.

5.1.1. Experimental Setup

5.1.1.1. Experimental Data

The audio data consist of 22 German lecture recordings. The lectures cover various topics such as history, language, copyright, computer science, etc. The total length of all the recordings amount to approximately 55167 seconds.

In order to evaluate the system performance with OOVs added to the STT system, two different confusion networks were used. One is generated by the STT system with normal dictionary and vocabulary and the WER is 27.55%. The other is generated by the STT system with modified dictionary and vocabulary, in which all the OOVs in the search terms are added, and the WER is 27.60%. Taking the randomness of the recognition results into consideration, the difference between the two WERs is not significant.

The vocabulary and dictionary used in the spoken term detection task contain approximately 300k entries.

The term list for the STD systems contains 250 search terms. A search term may be a single word query or a multi-word query composed of up to 4 words. Search terms are primarily short queries and they are randomly selected from the audio transcripts.

The words of search terms can be classified into three different categories: in-vocabulary words (IVs), compound out-of-vocabulary words (compound OOVs) and other out-of-vocabulary words (other OOVs). The category of other OOV contains foreign loanwords, proper names, acronyms, etc.

Some statistics about the search terms are listed in table 5.1.

	Number	Ratio
Number of 1-word search terms	135	54%
Number of 2-word search terms	90	36%
Number of 3-word search terms	20	8%
Number of 4-word search terms	5	2%
Number of search terms containing no OOVs	177	70.8%
Number of search terms containing compound OOVs	63	25.2%
Number of search terms containing other OOVs	10	4.0%
Average word numbers per search terms	1.58	
Total number of search terms	250	

Table 5.1.: Details of search terms

5.1.1.2. Spoken Term Detection System

In order to test the effect of different compound splitting methods, several spoken term detection systems with different compound splitting methods were used during the experiments. In all systems no query expansion features were used and the best 25 splitting results were used for searching the compound words. The systems are listed below:

Baseline No compound splitting features are adopted. This is the baseline of the experiments.

Primitive It uses a primitive method to split the compound words. The program scans a word from the end to the beginning and splits the part it has scanned every time when this part forms a word that can be found in the vocabulary.

Kevin It is the origin method used by the STT system in the institute and implemented by Kevin Kilgour. In the experiments it was used for comparison with other systems.

Rules The system uses the word formation rules to split compound words, as described in section 4.2.

Threshold 1 The system uses compound splitting algorithm based on word similarity with thresholds $K = 1$ and $R = 0.1$, as described in section 4.3.

Threshold 2 The system uses compound splitting algorithm based on word similarity with thresholds $K = 2$ and $R = 0.2$, as described in section 4.3.

Threshold 3 The system uses compound splitting algorithm based on word similarity with thresholds $K = 3$ and $R = 0.3$, as described in section 4.3.

5.1.2. Results and Evaluation

This section presents the effect of different methods introduced before. The ATWV is used as primary evaluation criterion. The DET curves are also presented in the last part of this section.

5.1.2.1. Effect of Compound Splitting Methods

The results are presented in table 5.2 and also in figure 5.1 in form of column chart. The systems was not only tested on all search terms, but also on two other datasets. Compound is the list of search terms containing compound words and OOV is the list of search terms containing other OOVs. Besides ATWV and MTWV, the number of correct detections (Corr), false alerts (FA) and missed detections (Miss) is also listed.

System	Dataset	**ATWV**	MTWV	Corr	FA	Miss
Baseline	All	**0.5366**	0.5269	1425	230	528
Primitive	All	**0.6608**	0.6511	1521	235	432
Kevin	All	**0.6608**	0.6511	1521	235	432
Rules	All	**0.6739**	0.6642	1535	235	418
Threshold 1	All	**0.6655**	0.6559	1506	238	447
Threshold 2	All	**0.6914**	0.6816	1546	280	407
Threshold 3	All	**0.6401**	0.6313	1526	302	427
Baseline	Compound	**0.0000**	0.0000	0	0	139
Primitive	Compound	**0.4513**	0.4510	76	5	63
Kevin	Compound	**0.4513**	0.4510	76	5	63
Rules	Compound	**0.5004**	0.5001	90	5	49
Threshold 1	Compound	**0.4689**	0.4689	61	8	78
Threshold 2	Compound	**0.5489**	0.5473	101	50	38
Threshold 3	Compound	**0.3478**	0.3497	79	72	60
Baseline	OOV	**0.0000**	0.0000	0	0	41
Primitive	OOV	**0.1000**	0.1050	20	0	21
Kevin	OOV	**0.1000**	0.1050	20	0	21
Rules	OOV	**0.1000**	0.1050	20	0	21
Threshold 1	OOV	**0.1000**	0.1050	20	0	21
Threshold 2	OOV	**0.2200**	0.2300	20	0	21
Threshold 3	OOV	**0.2825**	0.2925	22	0	19

Table 5.2.: Performance of different compound splitting Methods

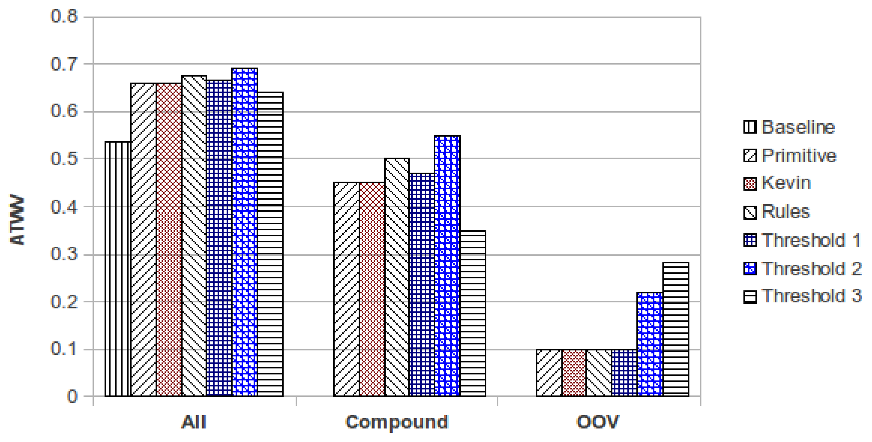


Figure 5.1.: Result overview

Overall, the highest ATWV achieved was 0.6914 by using compound splitting algorithm based on word similarity with thresholds $K = 2$ and $R = 0.2$.

In the baseline system, no compound splitting methods were used and thus no compound words or OOVs were detected. Comparing with the baseline, the adoption of different compound splitting methods improved the ATWV by 0.12 to 0.16 on all the terms.

The compound splitting methods had no obvious effect on OOVs other than compound words. System Threshold 2 and Threshold 3 performed better than other systems, which performed exactly the same on dataset OOV. And system Threshold 3 surpassed system Threshold 2 in particular.

Through all the system using thresholds, system Threshold 2 performed the best on compound words. Due to the loose thresholds, this system could generate more splitting possibilities for searching, which lowered the number of missed detections and improved the ATWV.

Compared with system Threshold 2, Threshold 3 uses even looser thresholds, but this may lead to bad splitting results, which deviate from the original word too much and kick out some better splitting results and consequently the number of missed detection will be higher. This is exactly the reason why system Threshold 3 performed worse than system Threshold 2 on compound words. On the other hand, effect of compound splitting on non-compound words is very similar to query expansion. The system splits or changes the words in an arbitrary way and provides more detection possibilities. In the experiment, system Threshold 3 had less missed detections, which surpassed the effect of more false alerts, and achieved higher ATWV than system Threshold 2 on non-compound OOVs.

The system Primitive and Kevin performed exactly the same on all the dataset. This indicates that, the compound splitting algorithm originally used in the TTS system is basically the same as the algorithm used in system Primitive.

5.1.2.2. Effect of Word Expansion Methods

In order to test the effect of different query expansion methods, each query expansion method was added to the Threshold 2 system to test how much the ATWV was improved. The systems used the same compound splitting methods as in system Threshold 2 and also different query expansion methods. Query expansion methods based on Levenshtein distance, Dice coefficient and pronunciation with different expansion counts varying from 1 to 50 were tested. And all the systems used term-specific threshold during the experiments. Figure 5.2 shows the results.

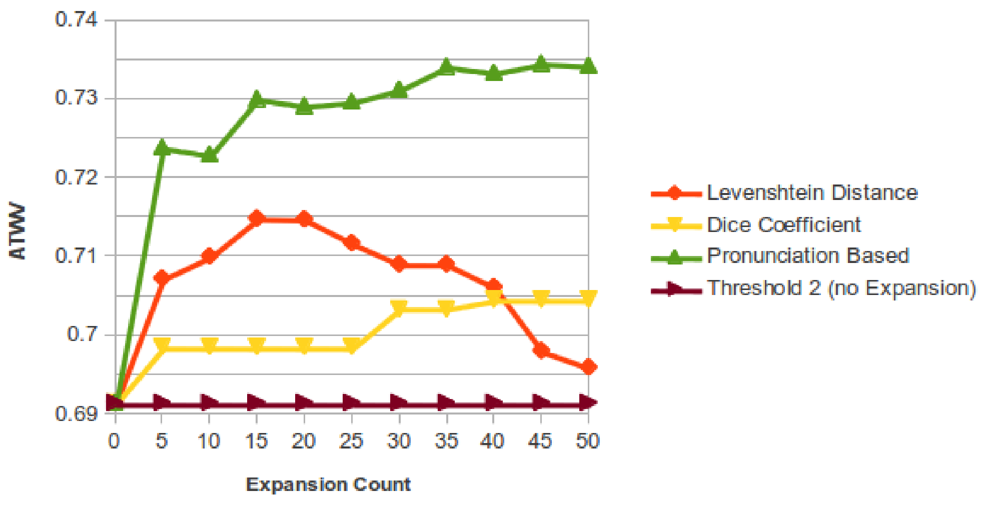


Figure 5.2.: Effect of different query expansion methods

The query expansion method based on pronunciation achieved apparently the best result. Except for the query expansion method based on Levenshtein distance, higher expansion count generally led to higher ATWV. For the query expansion methods based on Levenshtein distance, the ATWV was better than the result achieved by Dice coefficient based method at the beginning. But the ATWV it achieved began to decrease after about expansion count 20. After about expansion count 40 its ATWV was worse than the result achieved by Dice coefficient. Overall, the highest ATWV achieved here is 0.7342 by using pronunciation based method with expansion count 50.

5.1.2.3. Effect of Thresholds

In order to test the effect of different thresholds for binary decisions, the detection system using pronunciation based query expansion method was used here. The system was modified by changing the type and value of the threshold it used. The resulted ATWVs by using global threshold and term-specific threshold with different values are presented in figure 5.3 and 5.4 respectively.

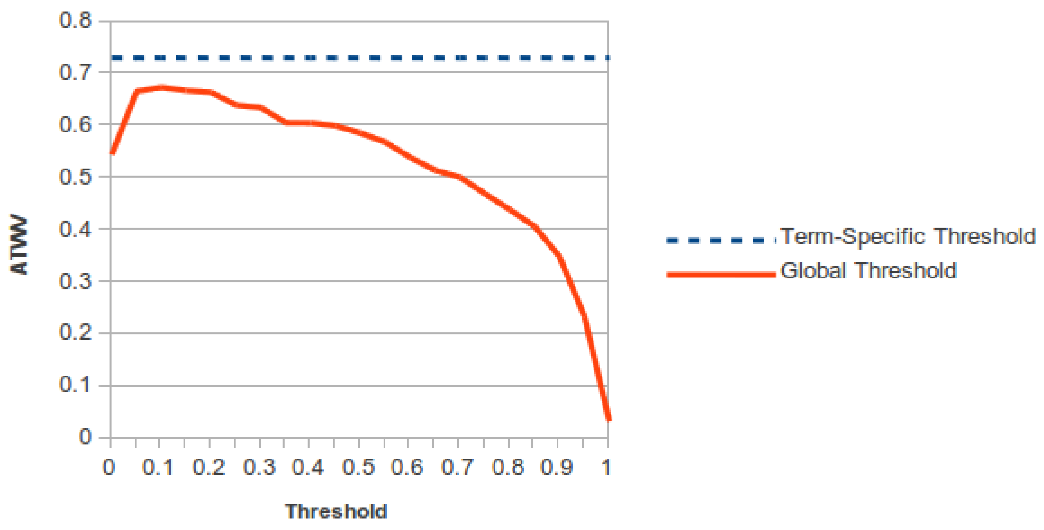


Figure 5.3.: Effect of different thresholds on all search terms

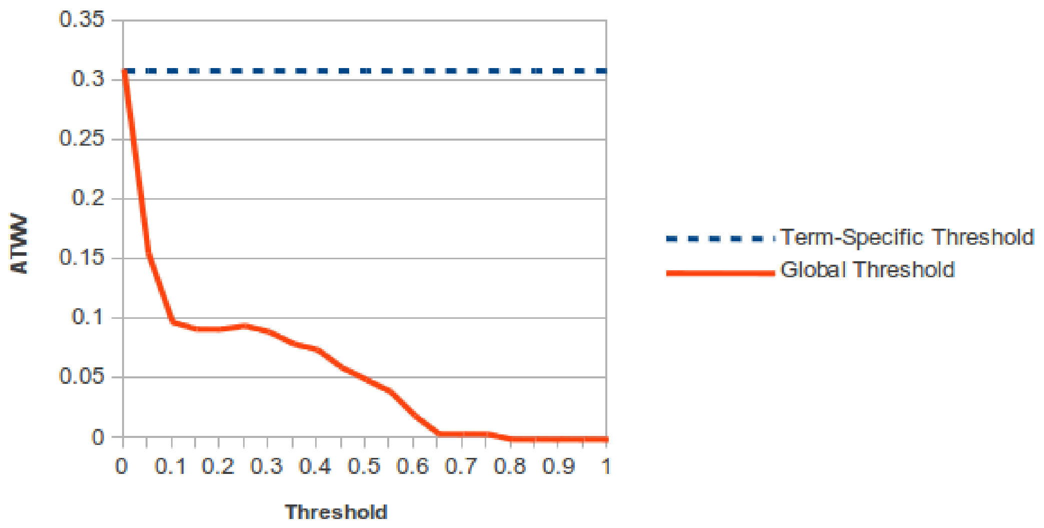


Figure 5.4.: Effect of different thresholds on OOVs

The adoption of term-specific threshold achieved generally better result than the adoption of global threshold.

For the experiment on the whole term list, the increase of global threshold led to higher ATWV at the beginning due to the reduction of false alerts. But soon the ATWV began to decrease for higher global threshold, because high global threshold reduced the missed detections effectively and didn't impact the false alert so much.

5.1.2.4. Effect of Adding OOVs to STT System

The OOVs were added to the STT system to generate a new confusion network for the STD task. In this way we it becomes possible to test how this improved the ATWV. In the table 5.3, "Ref_CN" means the original confusion network generated by the STT system without OOVs and "OOV_CN" means the confusion network generated by the STT system with OOVs. By using compound splitting and query expansion features in the spoken term detection systems respectively or together, different results were achieved. The compound splitting method used was based on word similarity with threshold $K = 2$ and $R = 0.2$ and the query expansion method used was based on pronunciation with expansion count 50. The results are presented in table 5.3 and also in figure 5.5 in form of column chart.

CN	System	**ATWV**	MTWV	Corr	FA	Miss
Ref_CN	Neither	**0.5366**	0.5269	1425	230	528
OOV_CN	Neither	**0.6293**	0.6207	1483	243	470
Ref_CN	Splitt	**0.6914**	0.6816	1546	280	407
OOV_CN	Splitt	**0.7056**	0.6966	1549	291	404
Ref_CN	Expansion	**0.6739**	0.5736	1487	639	466
OOV_CN	Expansion	**0.7125**	0.6431	1513	512	440
Ref_CN	Splitt + Expansion	**0.7342**	0.6824	1561	449	392
OOV_CN	Splitt + Expansion	**0.7365**	0.7013	1560	442	393

Table 5.3.: Effect of adding OOVs to STT system

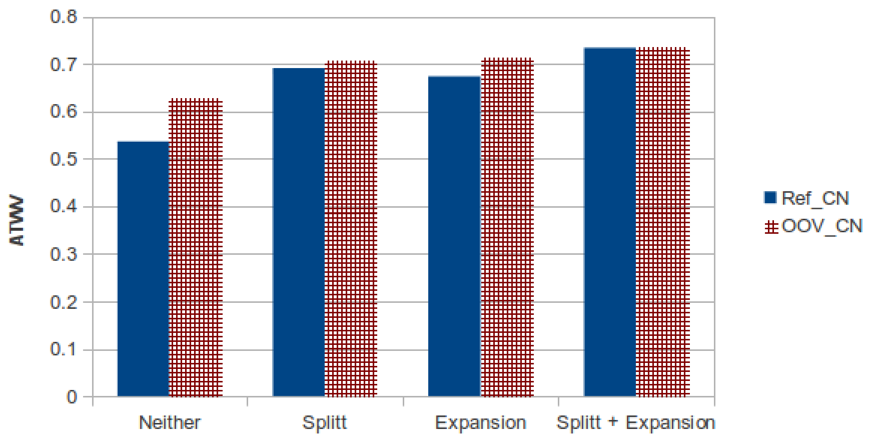


Figure 5.5.: Result overview

In all cases, the systems achieved higher ATWV on confusion network OOV_CN than Ref_CN due to the fact that the STT system used vocabulary and dictionary including OOVs from search terms to generate the confusion network OOV_CN. The difference of ATWVs was about 0.0927 without using compound splitting and query expansion features. But with the adoption of these features the difference began to shrink. With both features used, the difference was only as small as 0.0013. Reason for that is, at the beginning, the detection systems were not able to detect those OOVs from the confusion network Ref_CN, but the adoption of compound splitting and query expansion methods remedied this shortage and made the detection systems possible to find OOVs by searching divided word sequences or similar words in the confusion network.

The ATWV 0.7365 in the last row of the table is the best result achieved so far on this term list.

5.1.2.5. Detection-Error-Tradeoff Curves

The DET curve for the best result achieved so far is shown in figure 5.6.

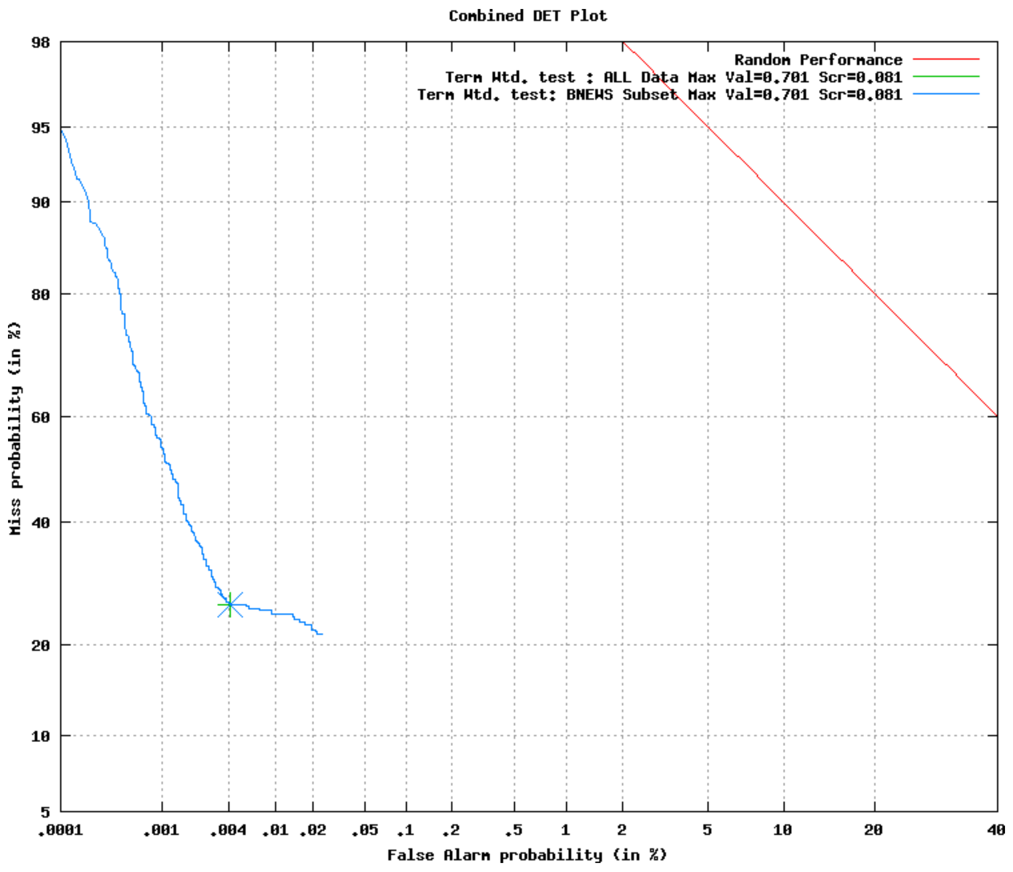


Figure 5.6.: DET curve for confusion network with OOVs

The compound splitting method based on word similarity with threshold $K = 2$ and $R = 0.2$ and the query expansion method based on pronunciation with expansion count 50 were used here.

Overall, the false alert probability didn't change too much, especially at the beginning (from 0.0001% to approximately 0.02%). With the increase of false alert probability, the miss probability decrease however significantly. The best result was achieved at about false alert probability 0.004% and miss probability 25.7% with ATWV 0.7365 and MTWV 0.7013.

5.2. Compound Splitting

This chapter describes the experiments on the different compound splitting algorithms which are introduced in chapter 4. The processing time and correctly split word count are the primary criterion used to evaluate the performance of different algorithms.

5.2.1. Experimental Setup

A list of compounds word was used to test the algorithm performance. The programs first loaded the dictionary into the memory, and then performed the splitting process word by word. In the case of Algorithm III, there was an additional BK-Tree building process after loading the dictionary and before the splitting. When the program split a compound word, a list of splitting candidates was generated, but only the most Levenshtein-similar splitting results were selected as the output. Furthermost, the output list was checked if it contained the correct splitting results which were pre-calculated as a standard results, and the number of correctly split words were also calculated in this way. The programs for this evaluation were written in C++. Generally the program output could also be used as a list of alternative search terms for the STD task.

The word list for the compound splitting task contained 65 compound words. These words were primarily randomly selected from the audio transcripts, a small part was some typical compound word from other sources.

The vocabulary used in the compound splitting task contained relative few entries, approximately $25k$. Some conjugation or declension forms of the words were not included in the vocabulary, so the possible splitting results were restricted in this way and the ability of different compound splitting algorithms was also evaluated better.

5.2.2. Results and Evaluation

For both Algorithm II and Algorithm III six different groups of thresholds were used. The value of these thresholds, the processing time and the correctly split word counts are listed in table 5.4.

Memory	4GB
Processor	2.66GHz
System	Ubuntu 12.04

	K	R	T_{Load}	T_{Build}	T_{Splitt}	t_{Splitt}	**split**
Algo I	NA	NA	0.74s	NA	0.49s	7.54ms	**46**
Algo II	1	0.1		NA	124.19s	1.91s	**35**
	1	0.2			171.45s	2.64s	**49**
	2	0.2			231.97s	3.57s	**53**
	2	0.4			276.28s	4.25s	**61**
	3	0.3			313.91s	4.83s	**58**
	3	0.6			374.07s	5.75s	**62**
Algo III	1	0.1		2.01s	7.88s	0.12s	**35**
	1	0.2			16.84s	0.26s	**49**
	2	0.2		4.77s	60.02s	0.92s	**53**
	2	0.4			104.65s	1.61s	**61**
	3	0.3		7.99s	141.91s	2.18s	**58**
	3	0.6			220.86s	3.40s	**62**
							Total 65

K	Threshold for the number of letter changes
R	Threshold for the ratio of letter changes
$\mathbf{T_{Load}}$	Time for loading vocabulary
$\mathbf{T_{Build}}$	Time for building BK-Tree
$\mathbf{T_{Split}}$	Time for splitting all words
$\mathbf{t_{Split}}$	Average time for splitting one word
split	Number of correctly split words
NA	Not applicable

Table 5.4.: Processing time and splitting results of compound splitting algorithms

The complete list of compound words and splitting results of different algorithms can be found in appendix B.

5.2.2.1. Splitting Ability

Algorithm I performed generally not as good as Algorithm II or Algorithm III. It only split 46 out of 65 compound words correctly.

The numbers of split words from Algorithm II and Algorithm III corresponded, since they were basically the same idea, only Algorithm III adopted the BK-Tree for more efficient word lookup. For these two algorithms, it's obvious that, the higher the thresholds were set, the better the result was. Besides, using BK-Tree for the word lookup saved some amount of time. The speedup was even more obvious when the thresholds were set low, like $K = 1$ and $R = 0.1$.

When thresholds were only set as low as $K = 1$ and $R = 0.1$, the result of Algorithm II or Algorithm III was even worse than Algorithm I.

When thresholds were set as high as $K = 3$ and $R = 0.6$, only 3 out of 65 words were not correctly split. The three words are "Adjazenzliste", "Kantendisjunkt" and "Raytracingverfahren", which all contains English words as sub-word, that can not be found in German vocabulary. But these thresholds are already very high for the compound splitting task, since they indicate that, each part of the compound word can have at most 3 and 60% letter changes. Words that can still not be split by using these thresholds are really rare.

When thresholds $K = 2$ and $R = 0.4$ were used, only one less word was not successfully split. This word is "Blätterwälder", which is composed of the basic words 'Blatt' and "Wald". In both parts, there are three letter changes, which are higher than the threshold $K = 2$. Notice that the plural form of the two words doesn't exist in the vocabulary used in the experiment.

5.2.2.2. Algorithm Efficiency

The tricks for accelerating the algorithms described in section 4.4.2 were used in the experiments: for Algorithm II the words were respectively stored according to their lengths, for Algorithm III different BK-Tree with words within different length ranges were built. It can be told from the statistics that more time were invested for building BK-Trees when K was bigger, since the size of the BK-Trees were larger and also more trees were built. But basically loading vocabulary and building BK-Trees consumed very little time compared with the time used for splitting words.

Algorithm I ran much faster than other two algorithms, because it only did several word lookups in general, which was efficiently implemented by using hashing table. The speed of Algorithm II and algorithms III depended on the thresholds used. When thresholds were set higher, the algorithms consumed more time. And Algorithm III was generally faster than Algorithm II, because BK-Trees accelerated the word lookup, when the overall thresholds were set less than or equal to 3.

5.2.2.3. Overall Evaluation

According to the statistics, it can be generally summarized that, the longer time the algorithm takes, the more splitting results it provides, the stronger is its ability to split a compound word.

As a good compromise between algorithm speed and result quality, thresholds $K = 2$ and $R = 0.4$ are a good choice. It split only one less word than the highest threshold setup $K = 3$ and $R = 0.6$ in the experiments, but consumed less time ($1.61s$ per word in compare with $3.40s$ per word). The splitting effect is even better, when larger vocabulary that contains many conjugation for declension forms is used.

When more speedup is needed, we can use smaller thresholds such as $K = 2$ and $R = 0.2$, which has worse performance but is still better than the idea of splitting words by using word formation rules.

Noteworthy is that, bigger thresholds indicate stronger capability to split compound words, but not necessarily higher performance in the STD task. Because the algorithm provides more splitting results with bigger thresholds, which could also increase the false alert possibility.

Another discovery in the experiments is that, BK-Tree actually accelerate the process of searching similar words based on Levenshtein distance, when the maximal required Levenshtein distance is less than or equals 3. The Bk-tree can also be adopted in other tasks which requires fast similar word lookup. And the criterion of similarity is also adjustable, it is not necessarily Levenshtein distance, it can also be other metric space. In generally, BK-Tree is a widely usable data structure for this kind of function.

5.3. Summary

In this chapter different spoken term detection systems and compound splitting algorithms are evaluated and compared.

In the first section different spoken term detection systems with different configurations were evaluated and compared. The highest ATWV achieved in our experiments on German lecture data is 0.7365 by using the compound splitting method based on word similarity with thresholds $K = 2$ and $R = 0.2$ and the query expansion method based on pronunciation with expansion count 50.

In the second section we compared the speed and splitting capability of different compound splitting methods which were adopted in the spoken term detection systems, and got an overall impression about how these algorithms perform.

6. Conclusion and Future Work

In this work, a system for the German Spoken Term Detection task was designed and evaluated with the design norms and evaluation metrics from the NIST 2006 Spoken Term Detection evaluation. Especially the different approaches to splitting compound words were analyzed and compared with each other.

Different compound splitting methods based on word formation rules or word similarity were tested for improving the detection result of compound words. Generally, if the detection system try more possibilities to split a compound word at different positions, it takes more time to split it. The missed detection rate will decrease because more splitting results will be used for detection, but the false alert rate will also increase because of the same reason. The best approach found to detecting compound words is the method based on word similarity with thresholds $K = 2$ and $R = 0.2$.

Different query expansion methods based on Levenshtein distance, Dice coefficient or pronunciation were also tested for detecting OOVs. The best approach found to achieve this is the query expansion method based on pronunciation.

We also tried the method of adding the OOVs to the ASR system and found the detection results were also improve. When all these methods were adopted, we got a fairly good detection result, the ATWV reached 0.7365 on German lecture data.

The ASR system is presumed as a prerequisite component for the STD system and thus doesn't fall within the scope of this work. However, the STD result depends highly on the output of the ASR system, therefore improving the ASR system is generally a good method to improve the STD result [Kol11]. Besides it would also be helpful to find more capable or efficient algorithms to split compound words or expand the OOVs.

Compounding of words is not only a common linguistic phenomenon in German language, it also exists in some other languages like Dutch, Finish, etc [PSN06]. So the result in this work may also be used to, or at least give some insight into, developing STD system for these languages.

The design and evaluation of the detection system comply with the standard specified by the NIST 2006 STD evaluation. The system is dedicated on detecting the exact matchings of the search terms, but other occurrences of the search terms in different word forms will be ignored. Since German is a typical fusional language, there is a lot of conjugation or declension usage of words in German, it would be much better if there was a system which could also detect the different forms of search terms or even related information to search terms. This is exactly the motivation of Information Retrieval to a large extend.

Bibliography

[BK73] W. A. Burkhard and R. M. Keller, "Some approaches to best-match file searching," *Commun. ACM*, vol. 16, no. 4, pp. 230–236, Apr. 1973. [Online]. Available: http://doi.acm.org/10.1145/362003.362025

[ByN98] R. Baeza-yates and G. Navarro, "Fast approximate string matching in a dictionary," in *In Proc. SPIRE'98.* IEEE Computer Press, 1998, pp. 14–22.

[KK03] P. Koehn and K. Knight, "Empirical methods for compound splitting," in *In Proceedings of EACL*, 2003, pp. 187–193.

[Kol11] H. Kolhorst, "Strategies for out-of-vocabulary words in spoken term detection," 2011.

[Lan98] S. Langer, "Zur morphologie und semantik von nominalkomposita," 1998.

[LMT+96] B. Logan, P. Moreno, J.-M. V. Thong, E. Whittaker, J. manuel Van, and T. Whittaker, "An experimental study of an audio indexing system for the web," in *in Proc. ICSLP*, 1996, pp. 676–679.

[MBS00] L. Mangu, E. Brill, and A. Stolcke, "Finding consensus in speech recognition: Word error minimization and other applications of confusion networks," 2000.

[MKK+07] D. R. H. Miller, M. Kleber, C.-L. Kao, O. Kimball, T. Colthurst, S. A. Lowe, R. M. Schwartz, and H. Gish, "Rapid and accurate spoken term detection." in *INTERSPEECH.* ISCA, 2007.

[MS09] T. Mertens and D. Schneider, "Efficient subword lattice retrieval for german spoken term detection," in *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, april 2009, pp. 4885 –4888.

[Nat06] National Institute of Standards and Technology (NIST), "The 2006 spoken term detection evaluation plan," september 2006.

[PSN06] M. Popović, D. Stein, and H. Ney, "2006. statistical machine translation of german compound words," in *FinTAL - 5th International Conference on Natural Language Processing, Springer Verlag, LNCS*, 2006, pp. 616–624.

[Wan10] D. Wang, *Out-of-Vocabulary Spoken Term Detection.* The University of Edinburgh, 2010.

[WKFB10] D. Wang, S. King, J. Frankel, and P. Bell, "Stochastic pronunciation modelling and soft match for out-of-vocabulary spoken term detection." in *ICASSP.* IEEE, 2010, pp. 5294–5297. [Online]. Available: http://dblp.uni-trier.de/db/conf/icassp/icassp2010.html#WangKFB10

Nomenclature

ASR	Automatic Speech Recognition
ATWV	Actual Term Weighted Valuetem
BN	Broadcast News
CONFMTG	Conference Meetings
CTS	Conversational Telephone Speech
DET	Detetion-Error-Tradeoff
ECF	Experiment Control File
IR	Information Retrieval
IV	In-Vocabulary [Word]
MTWV	Maximum Term Weighted Value
NIST	National Institute for Standards and Technology
OOV	Out-of-Vocabulary [Word]
RTTM	Rich Transcription Timer Mark
SDM	Spoken Data Mining
SDR	Spoken Document Retrieval
STD	Spoken Term Detection
STT	Speech-to-Text [System]
WER	Word Error Rate
XML	Extensible Markup Language

Appendices

Appendix A contains source code of the BK-tree implementation. It demonstrates how this data structure works. The most important parts are the two interfaces *insert* and *query*, which shows how to insert a word in the data structure and how to look up similar words. The query process uses depth-first search to check nodes in the tree recursively. The code presented is written in C++ programming language.

Appendix B contains the list of compound words, which are used to evaluate the performance of compound splitting algorithms. The detailed splitting result is also presented in tabular form.

A. Source Code of BK-Tree Implementation

```
#include <string>
#include <vector>
using namespace std;

const static int MAXL = 100;   //presumptive maximal word length
const static int MAXD = 50;    //presumptive maximal Lev. distance
int f[MAXL][MAXL];             //array for calculating Lev. distance

int min(int x, int y) {
    return x < y ? x : y;
}

int min(int x, int y, int z) {
    return min(x, min(y, z));
}

int levenshtein_dist(wstring w1, wstring w2) {
    f[0][0] = 0;
    for (unsigned i = 1; i <= w1.size(); ++i)
        f[i][0] = i;
    for (unsigned i = 1; i <= w2.size(); ++i)
        f[0][i] = i;
    for (unsigned i = 1; i <= w1.size(); ++i)
        for (unsigned j = 1; j <= w2.size(); ++j)
            if (w1[i-1] == w2[j-1])
                f[i][j] = min(f[i-1][j-1], f[i-1][j]+1, f[i][j-1]+1);
            else
                f[i][j] = min(f[i-1][j-1]+1, f[i-1][j]+1, f[i][j-1]+1);
    return f[w1.size()][w2.size()];
}

class BK_tree {
    struct BK_node {           //tree node
        wstring word;          //word in the node
        int child[MAXD];       //children of the node
    };

    vector<BK_node> nlist;     //node list of BK-tree
    vector<wstring> result;    //temporarily store result
    wstring qword;             //temporarily store query word
    int qdist;                 //temporarily store distance threshold

    void DFS(int root) {       //search similar words recursively
        int dist = levenshtein_dist(qword, nlist[root].word);
        if (dist <= qdist)
            result.push_back(nlist[root].word);
        for (int p = dist-qdist; p <= dist+qdist; ++p)
            if (p>=0 && nlist[root].child[p] != 0)
                    DFS(nlist[root].child[p]);
    }
public:
    BK_tree() {
        nlist.clear();
    }

    void insert(const wstring &word) {
        BK_node x;
        x.word = word;
        for (unsigned i = 0; i < MAXD; ++i)
            x.child[i] = 0;
        nlist.push_back(x);
        if (nlist.size() > 1) {
            int root = 0;
            int dist = levenshtein_dist(word, nlist[root].word);
            while (nlist[root].child[dist] != 0 ) {
                root = nlist[root].child[dist];
                dist = levenshtein_dist(word, nlist[root].word);
            }
            nlist[root].child[dist] = nlist.size()-1;
        }
    }

    vector<wstring> query(const wstring &word, int threshold) {
        result.clear();
        qword = word;
        qdist = threshold;
        DFS(0);
        return result;
    }
};
```

B. Word List of Compound Splitting Task

Compound Word	**Correctly Splitted?**						
Algorithm	**I**	**II/III**					
Threshold K		1	1	2	2	3	3
Threshold R		0.1	0.2	0.2	0.4	0.3	0.6
Antragsteller	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Aufeinanderprallen	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Berichterstattungsschranke	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Bibliothekssystem	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Bombenangriff	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Drehimpulserhaltungssatz	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Einkommensgap	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Eisenkern	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Frauenerwerbstätigkeit	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Freiraumrepräsentation	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Frontkonsole	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Gabelform	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Hintereinanderschaltung	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Hitzewelle	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Hobbyclub	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Koartikulationsphänomen	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Kohlsuppe	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Kulturlandschaftsbegriff	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Linksrechtsmodelle	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Magenschleimhautentzündung	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Militärpsychologie	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Ndimensional	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Perplexitätsmaß	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Polarkoordinatenraum	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Retraktionsverfahren	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Schaltsekunde	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Schifffahrt	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Soziologenkongress	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Startmenge	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Südspanien	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Treibhauseffektgase	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Zweitvoraussetzung	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Abfahrtszeit	Yes	No	Yes	Yes	Yes	Yes	Yes
Dualitätsresultat	Yes	No	Yes	Yes	Yes	Yes	Yes
Geisterstunde	Yes	No	Yes	Yes	Yes	Yes	Yes
Großforschungsprojekte	Yes	No	Yes	Yes	Yes	Yes	Yes
Hundehalter	Yes	No	Yes	Yes	Yes	Yes	Yes
Kindersterblichkeitsraten	Yes	No	Yes	Yes	Yes	Yes	Yes
Sitzungssaal	Yes	No	Yes	Yes	Yes	Yes	Yes
Staatsfeind	Yes	No	Yes	Yes	Yes	Yes	Yes
Stereosichtsysteme	Yes	No	Yes	Yes	Yes	Yes	Yes
Windsbraut	Yes	No	Yes	Yes	Yes	Yes	Yes
Mehrpersonenhaushalte	Yes	No	No	Yes	Yes	Yes	Yes
Nichtlinearitäten	Yes	No	No	Yes	Yes	Yes	Yes

Compound Word	**Correctly Splitted?**						
Algorithm	**I**	**II/III**					
Threshold K		1	1	2	2	3	3
Threshold R		0.1	0.2	0.2	0.4	0.3	0.6
Oberflächeneigenschaften	Yes	No	No	Yes	Yes	Yes	Yes
Straußenei	Yes	No	No	No	Yes	Yes	Yes
Experimentalwissenschaftler	No	Yes	Yes	Yes	Yes	Yes	Yes
Prediktionswahrscheinlichkeit	No	Yes	Yes	Yes	Yes	Yes	Yes
Technikregulierend	No	Yes	Yes	Yes	Yes	Yes	Yes
Aphorismenschatz	No	No	Yes	Yes	Yes	Yes	Yes
Erneuerungspotenzial	No	No	Yes	Yes	Yes	Yes	Yes
Schweigeminute	No	No	Yes	Yes	Yes	Yes	Yes
Stadienverbot	No	No	Yes	Yes	Yes	Yes	Yes
Wöchnerinnenheim	No	No	No	Yes	Yes	Yes	Yes
Gravitationskrafteinfluss	No	No	No	No	Yes	Yes	Yes
Gänseklein	No	No	No	No	Yes	Yes	Yes
Pharmakaanalyse	No	No	No	No	Yes	Yes	Yes
Prinzipienreiter	No	No	No	No	Yes	Yes	Yes
Knackmechanismen	No	No	No	No	Yes	No	Yes
Museenverwaltung	No	No	No	No	Yes	No	Yes
Perzeptronlernalgorithmus	No	No	No	No	Yes	No	Yes
Blätterwälder	No	No	No	No	No	No	Yes
Adjazenzliste	No	No	No	No	No	No	No
Kantendisjunkt	No	No	No	No	No	No	No
Raytracingverfahren	No	No	No	No	No	No	No
Splitted (from 65 in Total)	**46**	**35**	**49**	**53**	**61**	**58**	**62**