12/01/2003

Information Retrieval for Languages that lack a fixed orthography

Jan Strunk
MA student, Linguistics Department
Stanford University, California
jstrunk@stanford.edu

ABSTRACT

In this paper, I describe and compare different solutions to the problem of information retrieval for languages that lack a fixed orthography. For these languages, even simple Boolean queries are a problem, because the person writing the query might use a different orthographic system from the people whose documents have been indexed. Moreover, if the aim is to find all documents containing a certain word, this is very difficult or nearly impossible for such languages because of the sheer variety of different but equivalent orthographic forms. The goal of the project described in this paper was to develop and evaluate different approximate matching algorithms in order to achieve a better retrieval performance for such languages. I compare different variants of edit distance algorithms and briefly discuss other possible solutions.

Keywords

Information Retrieval, Orthographic variation, Edit Distance, Approximate String Matching, Lesser-used languages.

1. INTRODUCTION

Linguists estimate that there are about 6000 languages spoken in the world today. However, of these 6000, only a tiny fraction are official languages in some state, enjoy prestige and political support. Most other languages lack any official status and are often used in small communities. In this age of globalization, many of these minor languages are threatened with extinction. The internet as a new revolutionary publishing medium is an exciting opportunity for some of these languages as it allows cheap and uncensored publishing of texts, while publishing in ordinary media is often not economically feasible or even forbidden in some states. The main problem for information retrieval in these languages is the fact that many non-official languages have never been standardized in the way that most official languages have been, i.e. they often show a remarkable variation in lexicon, grammar, and orthography. I will mainly try to deal with the problem of orthographic variation here.

The language I will be concerned with is Low Saxon (a.k.a. Low German, Plattdeutsch, etc.). It is a West Germanic language spoken in Germany, the Netherlands, and in emigrant communities throughout the world. It can be considered a "major" minor language in that estimates of the number of speakers are sometimes as high as 10,000,000, cf. [1]. This language presents a good test case for the development and evaluation of robust information retrieval algorithms, because it shows an astounding dialectal and orthographic variety. This is due to the fact that it is spoken by people all over northern Germany and the western Netherlands, as well as in communities in Russia and the whole American continent. As it is used in different states, the respective official languages have influenced Low Saxon and largely shaped the way it is written. This means that Low Saxon varieties in the Netherlands are usually written in an orthographic system resembling that of Dutch, whereas the orthography used for dialects in Germany is largely based on Standard German. Mennonites in Canada or the US who speak a dialect called Plautdietsch sometimes even use English orthographic devices for their vernacular.

Example (1) shows different variants of the word *söken* which translates to English "search".

(1) *säkje, säuken, seuken, söken, sööken, zoeken, zuiken*

Although one could simply regard all seven as variants and say no more, I would like to introduce a distinction between graphemic and dialectal variants here. First, we can observe that there are two general orthographic systems represented here: *zoeken* and *zuiken* are based on Dutch spelling, the rest on German. However, these variants can be grouped together into pronunciation (or dialectal variants):

1. *säkje*
2. *säuken, seuken, zuiken*
3. *söken, sööken*
4. *zoeken*

This means that e.g. *säuken* and *zuiken* are just graphemic variants of the same pronunciation (or dialectal) variant, whereas the four groups can be considered different dialectal variants of the same cognate word.

The goal of the project described in this paper was to allow a user to enter any of the variants above as query term and to get all documents containing any of the variants above as result. A system that allows this should first and foremost recognize different graphemic variants and as far as possible also pronunciation variants.

2. RELATED WORK

I am not aware of any studies that have been done on the specific problem of information retrieval for languages that lack a fixed orthography. The next related problem seems to be what is usually called ***approximate name matching***. It is used for task such as phone directory lookup and also genealogical research.

For this study, I drew on the experiments conducted by Zobel and Dart ([3] and [4]) on English, and Erikson [3] on Swedish. Zobel and Dart [3] evaluate different approximate matching algorithms for retrieval effectiveness, memory requirement, and speed. They propose a two-stage search on the lexicon that I will also use in my experiments. First, a "coarse search" over an index into the lexicon is used to retrieve likely candidates relatively quickly in order to avoid searching the whole index with the actual matching algorithm. The second stage consists of a "fine search" on these likely candidates with a more accurate but usually slower algorithm. Erikson also uses this retrieval model. He divides the lexicon into bins according to the first letter of the terms, whereas Zobel and Dart try out different methods. They conclude that traditional phonetic codings such as the well-known Soundex method perform very poorly and that n-gram indexing methods should be preferred. Moreover, they find that either a simple edit distance algorithm (such as the so-called Levenshtein Distance, cf. [5] and section 4.1.1), or an n-gram similarity approach should be used for fine searching. Erikson comes to the same conclusions: if speed does not matter much, a variant of the Levenshtein distance should be preferred, his second most effective algorithm and the fastest one uses bigram similarity.

Erikson shows that a linguistically knowledgeable edit distance algorithm that assigns different costs to edit operations (cf. section 4.1.1 and 4.1.2) according to the "phonetic classes" of the letters involved is quite effective, but a little slow. Zobel and Dart [4] present two similar algorithms which make use of phonetic information. The first one is called ***editex*** and like Erikson's algorithm is a modified edit distance algorithm. The other called ***ipadist*** first tries to transform the strings into phonemic representations using text-to-phoneme algorithms and then compares them using edit distance. In their evaluation, they conclude that editex performs quite well and usually better than the more complex ipadist.

Before I describe the different algorithms I have tested, I want to bring up some differences between the problem of approximate name matching and information retrieval for languages without a fixed orthography. Zobel and Dart [4], define phonetic matching as "the process of finding strings that, prior to possible changes that broadly preserve the sound, may have had the same pronunciation". In their paper, they compare the output of their system with human judgments about whether two strings have a similar pronunciation. The task of finding variants of the same word is slightly different; I only consider spelling and pronunciation variants of the same cognate word (with approximately the same meaning) as true positives, which means that correct pairs of strings can actually have quite a different pronunciation sometimes and also that words which are pronounced the same but are not cognates and have different meanings are not considered as true positives. This difference will show up again in the evaluation in section 5.

3. THE DOCUMENT COLLECTION

I developed and tested the algorithms that I describe in this paper on a document collection that I build manually by harvesting the internet for Low Saxon texts. The Low Saxon community on the net is quite large and luckily well interlinked, so that it was relatively easy to find a large number of websites wholly or partly in Low Saxon. I downloaded about 2700 documents of which about 1700 contain Low Saxon only text while the rest is only partly Low Saxon. Downloading these documents to my local file system and saving them in utf-16 format resulted in about 74 MB of html files of which about 40 MB are Low Saxon only. I collected a large diversity of texts ranging from Wikipedia[1] articles to poetry in a large number of different dialects and orthographic systems (some quite idiosyncratic and experimental). Although this collection is relatively small, I estimate that it contains a sizeable portion of all Low Saxon texts on the internet. I then used the Jakarta Lucene search engine classes[2] to build an index of all Low Saxon only documents. I stripped the html tags from the documents using a simple regular expressions approach, because of various problems with existing html parsers. Except for case folding, every word form was indexed as it occurred in the document collection and no stop word removal was carried out. This resulted in an index comprising 93700 distinct terms which seems to be quite a lot for such a small document collection. All further work I describe in the following sections builds on this Lucene index using newly created Java modules that preprocess queries entered by the user before sending them on to Lucene as simply Boolean queries.

4. THE CURRENT SYSTEM

4.1 Basic algorithms

4.1.1 Levenshtein Distance

In section 2, I have repeatedly mentioned the family of so-called edit distance algorithms. These algorithms calculate the distance between to strings by transforming one of them into the other using certain operation, usually: *ins(ert)*, *del(ete)* and *sub(stitute)*. The pair *söken – säuken* for example has a mininum edit distance of 3, if we assign the costs 1 for *ins* and *del* and 2 for *sub* (substituting a character with itself has zero cost) because we first have to substitute the *ö* with *ä* and then insert a *u* after the first vowel. The minimum edit distance between two strings of length *m* and *n* can be found in O(nm) time using dynamic programming, cf. Jurafsky and Martin [5] pp. 153-156. A very simple minimum edit distance algorithm that assigns a uniform cost to the different operations independent of the string it operates on is often called ***Levenshtein distance***. I will use the Levenshtein distance as one approach in my system, using insertion and deletion costs of 1 and a substitution cost of 2 for different segments.

4.1.2 Low Saxon distance

Alternatively, the same minimum edit distance algorithm can be used with more complex cost functions. Similar to the "phonetic" edit distance algorithms proposed by Erikson, and Zobel and Dart, I will use such a linguistically more informed algorithm as an alternative to the Levenshtein distance. My algorithm – lets call it ***Low Saxon distance*** – uses a more complex cost function that bases the costs it assigns on equivalence classes that contain characters and combinations of characters (called ***graphs***). Based on the distinction between graphemic and pronunciation variants I introduced in section 1, I use two levels of equivalence: Firstly, one graph can be a graphemic variant of another such as *äu* and *eu* in *seuken* vs. *säuken*. These spellings are interchangeable

[1] http://nds.wikipedia.org/wiki.cgi

[2] http://jakarta.apache.org/lucene/

variants without a difference in pronunciation. I therefore consider them to belong to one graphemic class. Secondly, two graphemes (i.e. graphemic classes) can represent cognate sounds, i.e. slightly different pronunciations that are used in different dialects in the same word, e.g. the class {*äu*, *eu*} would be a pronunciation variant of the class {*ö*, *öö*} as in *söken* and *sööken*.

Parallel to Zobel and Dart [4], I allow one graph to be a member of multiple grapheme classes and one grapheme to be a member of multiple pronunciation classes. For example, the graph *öö* sometimes represents a long vowel and sometimes a diphthong (i.e. a combination of two different vowels). I have built a large data structure containing these equivalence relations[3]. As well as a set containing graphs that are often omitted in certain dialects, and which should accordingly cost less to insert or delete during minimum edit distance computation.

The costs I use for the Low Saxon Distance algorithm are as follows:

Ins(g) = 0.5 if g is a graph that is easily omitted, Ins(g) = 1 otherwise.

Del(g) = 0.5 if g is a graph that is easily omitted , Del(g) = 1 otherwise.

Sub(g1, g2) = 0 if g1 and g1 are the same. Sub(g1, g2) = 0.25 if g1 and g2 are members of the same graphemic class. Sub(g1, g2) = 0.5 if g1 and g2 are members of the same pronunciation class, and Sub(g1, g2) = 2 otherwise.

In all, I defined about 180 different vocalic graphs combining into 30 vocalic graphemic classes and 15 sound classes. As for consonants, I assume 110 different graphs that are assigned to 25 different consonantal graphemes and 27 consonantal sound classes.

4.1.3 Graphemic parsing

In section 4.1.2, I stated that graphs which are elements of graphemic classes can not only be single letters but also combinations of letters which form a functional unit. These so-called polygraphs have to be identified in order to correctly split a string into its graphs prior to comparison with another string. For example, *sööken* should be split into *s* | *öö* | *k* | *e* | *n*, as I consider *öö* to be a polygraph belonging to the same graphemic class as e.g. *ö* and *öh*. This kind of graphemic parsing of a string is already exploited indirectly by Zobel and Dart [4] in their ***ipadist*** approach by using a grapheme-to-phoneme converter. Moreover, they also employ a crude heuristics in their ***editex*** algorithm which considers double letters equivalent to their single letter counterpart.

The problem of graphemic parsing is nontrivial, because a given string cannot always be correctly divided into graphemes by a deterministic parser at least not without a lexicon. The string *ph* e.g. can sometimes represent the consonants *p* and *h* following each other (as in *doruphen* "as a consequence of that") or a polygraph representing the sound *f* (as in *elephanten* "elephants"). In this paper, I evaluate three different parsing algorithms: First, as a baseline, I simply consider each individual letter as a separate graph. I call this algorithm *letter parse*. Second, I use a left-to-right greedy algorithm that simply searches for the longest possible grapheme that starts from a given position in the string, which is correct a lot of the time, but would e.g. get *doruphen* wrong. I call this algorithm ***simple parse***. Third, I use a nondeterministic algorithm I call ***multi parse*** which outputs a set of possibly correct parses: such as *d* | *o* | *r* | *u* | *p* | *h* | *e* | *n* and *d* | *o* | *r* | *u* | *ph* | *e* | *n* for *doruphen*. During the actual matching, all alternative parses of the first are compared with all parses of the second string. The lowest distance of any pairing is taken to be the distance of the two strings. This usually ensures that the two strings are aligned correctly at least once. However, as can easily be seen this may result in a dramatic increase in the time needed to compare two strings if they can be split in different ways although the average number of different parses per term in the whole lexicon is about 1.35.

In order to keep the searching time as low as possible, the different parses of the terms in the lexicon are pre-computed and stored during indexing.

4.2 Coarse search

In contrast to Erikson, and Zobel and Dart, I decided not to use n-gram indexing in my system, because the orthographic variation exhibited by the Low Saxon texts seemed to me to be too high to allow effective indexing using letter n-grams. However, due to limited time, I did not actually perform any experiments to underpin this conclusion. This is therefore still an open question. Instead I implemented two basic coarse search approaches: First, I decided to use some simple representation of the syllable structure of the words as an index into the lexicon. For all the graphemic classes contained in my character class data structure, I also provided information about whether they are consonantal or vocalic. I assumed that this would make a relatively good coarse index into the lexicon because the syllable structure (i.e. the sequence of consonants and vowels rarely changes in different graphemic or dialectal variants[4]). Each word form in the lexicon gets assigned a code that is built by substituting every vocalic grapheme with a 1, every consonantal grapheme with a 2, and unknown letters (such as letters not normally used to write Low Saxon, e.g. Arabic, Cyrillic, Chinese, etc.) with a 0. For example the word *zuiken* would get the code 21212. I then built an index from these codes to the terms in the lexicon. In the version of my system, where nondeterministic graphemic parsing is used, one term can be indexed under multiple codes (one for each possible parse).

During searching, the possible parses for the query term are generated and the resulting codes are matched against all codes in the code index using a simple Levenshtein distance. All terms that have codes which match the query term code within a certain maximal distance are returned as the term set for fine searching. However, the system usually does not have to match against all codes in the code index, but can restrict itself to those that have approximately the same length as the query word code.

---

[3] These classes as well as the functions to access them are contained in the Java class CharacterClasses under the package LowSaxonIndexer.string.

[4] This is a different from the situation in spelling error correction where errors exchanging a vowel with a consonant or omitting parts of a word, etc. commonly occur.

As an alternative approach, I followed Erikson [2] by considering all terms that start with the same letter, the same grapheme, or the same sound class as the query term during fine searching. However, due to time constraints I only evaluated this approach using the first grapheme, cf. section 5.1.

4.3 Fine search

During fine search, the set of candidate terms returned by the initial coarse search (which is hopefully much smaller than the whole lexicon) is matched against the query term using either the simple Levenshtein Distance or the linguistically informed Low Saxon Distance. All terms in the lexicon that are less different from the query term than a certain threshold are considered to be variants of the query term and collected into a set of matching terms. From the resulting set of variant forms, a disjunctive Boolean query is constructed that is sent on to Lucene in order to retrieve all documents that contain at least one of the terms in this set of variants. In contrast to both Erikson, and Zobel and Dart, I do not rank the variants according to their similarity to the query term. Rather, I use a fixed threshold as I considered a yes/no decision easier to implement and more appropriate for the current problem where the goal is to find as many true variants as possible without to much user interaction. However, one further area of research would certainly be how to combine the usual document relevance ranking used in information retrieval with the linguistic distance between document term(s) and query term. Ideally, the relevance of a document should not depend on how different the variant contained in it is from the query term as long as it indeed is a variant of the query term.

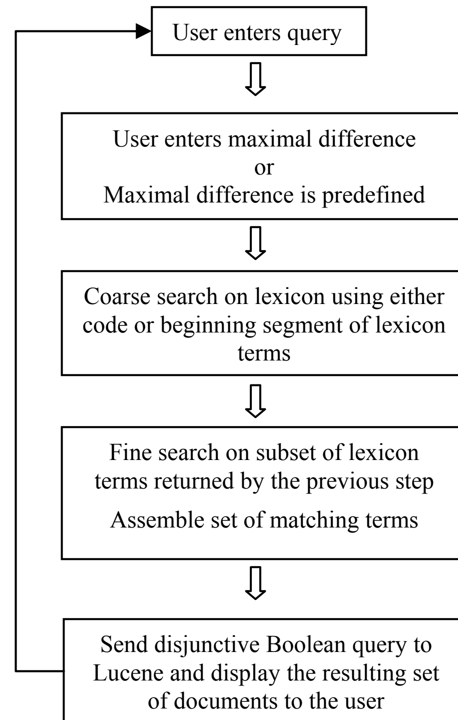

Figure 1: General overview of searching

4.4 Overview of the whole system

4.4.1 General overview

Figure 1 shows the general program flow while searching with the system described in this paper. First the user is prompted to enter a query term[5]. Next, the user can provide a maximal distance threshold. Alternatively, a predefined threshold value can be used. The query term is then parsed to obtain a string of graphemes using either ***letter parse***, ***simple parse***, or ***multi parse*** as described in section 4.1.3. Depending on the coarse search method used, either all lexicon terms starting with the same segment (letter, grapheme or sound) are retrieved or the query term's code is matched against the codes in the code index into the lexicon. The resulting set of candidate terms is matched against the query term using either Levenshtein Distance or Low Saxon Distance. From the resulting set of terms that matched the query term (within a certain difference threshold), a disjunctive Boolean query is constructed that is sent to Lucene in order to retrieve all documents containing one of the matching terms. The set of retrieved documents is the presented to the user.

[5] Currently only single word searches are possible.

4.4.2 Implementation

All of the above mentioned components of the current system have been implemented in Java and assembled into a package called ***LowSaxonIndexer***. This package consists of the components needed for constructing the various indexes and searching them later on. The subpackage ***indexing*** is made up of the tools needed for building the Lucene index and the additional indexes over the lexicon. It currently only allows building up a new index from scratch. But modules to add terms to an existing index could be added if needed. ***Searching*** is a subpackage that implements some tools for searching an index from the command line. Moreover, it contains ***CodeTableReader*** classes which are used to access the indexes over the lexicon from within a Java program. The subpackage ***fuzzymatching*** contains the basic algorithms that implement the Levenshtein Distance and the Low Saxon Distance as described above. Furthermore, it comprises different matchers that given a query term, a maximal difference, and access to a lexicon via a ***CodeTableReader*** class use the different coarse and fine search algorithms to return a set of matching terms from the lexicon. From the resulting set of terms, a Lucene Boolean query can be constructed using the class ***FormulateQuery***.

The steps needed to use my system are thus the following:

1. Assemble a collection of documents.
2. Build an index using indexing.BuildIndexFuzzy.
3. Use searching.SearchIndexFuzzy from the command line to search the index.
4. Alternatively, you can try out the web interface I provide under www2.linguistics.rub.de/LowSaxon/.

4.5 Brief overview of alternative approaches

Originally, I planned to implement two additional approaches to the same problem. First, as an alternative to actually searching the lexicon for approximate matches, I had the idea of simply expanding a query term to a number of linguistically plausible variants. However, I have come to the conclusion that this approach seems to be quite unrealistic. Using the alternatives encoded in the CharacterClasses module to expand a given query term results in a huge number of different possible variants and takes a long time and lot of memory. Consider a query term that contains 8 graphemes. If there are only 2 variants for each grapheme, this already results in $2^8 = 256$ possible combinations. This problem makes this approach infeasible even for smaller and more restricted character classes.

Second, I originally planned to implement a type of encoder that would normalize all terms during indexing to a common form as much as possible in order to save space and speed up searching by reducing lexicon size. However, due to the ambiguity of different graphemes and the sounds they represent in different dialects, this is a very hard task that ideally requires recognizing the dialect and writing system a document is written in during indexing. It seemed unrealistic to try this approach within the time provided for the completion of this project.

5. EVALUATION

Zobel and Dart [4] state that "*The parallels between information retrieval and phonetic matching mean that they can be measured by the same kinds of techniques. For example, it is, arguably, appropriate to compare phonetic matching techniques using recall and precision.*" I agree with them and use recall and precision to evaluate different combinations of the basic algorithms described in section 4. However, as I am using a non-ranked retrieval algorithm (at least for matching terms in the lexicon), I do not use recall-precision as Zobel and Dart, and Erikson do, but rather calculate simple precision, recall, and f-measure[6] for two different levels. First, I calculate these measures on the type level indicating the effectiveness of retrieving the correct variant forms of the query term from the lexicon (this is parallel to the evaluation in [2], [3], and [4]). Second, as the ultimate goal is to provide effective information retrieval for languages such as Low Saxon, I also calculate precision, recall, and f-measure on the token level thus evaluating the quality of the set of documents retrieved and displayed to the user. The difference in the evaluation methodology and distinct tasks make it impossible to directly compare the system described here to those described in Zobel and Dart [3] and [4], and Erikson [2].

On the type level, I count each matched term as a true positive if in some document this term represents the same word[7] as the query term (e.g. only in one dialect, or one orthographic system, etc.). Even more than in languages with a codified standard orthography, the ambiguity problem that is omnipresent in natural language processing makes itself felt. Homonymy (or homography) and polysemy are already problematic for standardized languages, but for a corpus of Low Saxon containing documents in different dialects and writing systems, these problems become worse. Two terms that are clearly distinguishable in one dialect might be homographs in another, or one term that has no homograph within one dialect could be confused with a different word that is incidentally pronounced and written the same as that term in another dialect. For example, the term *lief* used in one dialect means "body" and is clearly distinct from *leef* meaning "dear". However, in other dialects *lief* is the word for "dear" leading to interdialectal homonymy. As long as a matched term as a type is correct in at least one document, I consider it a true positive on the type level.

On the token level, on the other hand, I count every document retrieved that contains at least one correct variant of the query term with the desired meaning as a true positive. For example, all documents containing *lief* in the desired sense "body" would be counted as true positives, all other containing *lief* or a variant thereof in a different sense such as "dear" would be counted as false positives.

For the different experiments described below, I compiled a set of test queries that consists of 24 query terms extracted randomly from documents not indexed because they contained large portions of non Low Saxon text and 10 more words picked from a single issue of Radio Bremen's Plattdüütsche Nohrichten[8] (Low Saxon news). The test query terms are listed in the appendix of this paper. They all occur at least once in one variant in the test collection of the approx. 1700 Low Saxon only documents.

For all test query terms, I tried to find all different variants forms and all documents that contain one of these variants by hand in order to obtain a gold standard for use during the evaluation. I am confident that I have succeeded in accomplishing this with a high accuracy.

5.1 Experiments

I have performed four experiments in all using different combinations of the basic algorithms and the test query set described above.

As a baseline for comparison, I simply used Lucene directly without searching for variants of the test query terms. The resulting average type-level precision was 100 % which is understandable because I only used test query terms that occur in one variant or another in the document set. So, if the baseline found anything, it was very likely that this type was actually used in the desired sense somewhere in the collection. However, the average type-level recall was only 16.76 % (variance 6.05 %), which shows that on average less than one fifth of all correct types are found. This resulted in an average type-level f-measure of 30.69 % (variance 6.05 %). Seven of all 34 test types were not found at all. On the token level, average precision reached 98.15 % (variance 0.25 %) already showing effects of homography, etc., average recall was 35.05 % (variance 11.23 %) and the f-measure averaged 56.17 % (variance 8.44 %).

For the queries which resulted in no retrieved documents, no precision and f-measure could be calculated, so that they also did

[6] Defined here as: 2 • recall • precision / (recall + precision), cf. [6] p. 269.

[7] I count different inflectional forms of the same part of speech as true positives.

[8] http://www.radiobremen.de/bremeneins/platt/news/index.php3

not enter into the average precision and average f-measure calculations.

My first experiment was to compare six different combinations of fine searching algorithms with different graphemic parsers. I used the following configurations:

1. Low Saxon Distance (LS) with letter parse (LP)
2. Low Saxon Distance (LS) with simple parse (SP)
3. Low Saxon Distance (LS) with multi parse (MP)
4. Levenshtein Distance (LD) with letter parse (LP)
5. Levenshtein Distance (LD) with simple parse (SP)
6. Levenshtein Distance (LD) with multi parse (MP)

These threshold values for these combinations have been hand calibrated by testing on 10 distinct test query terms. They did not change with the length of the query term. See table 1 for the threshold values used.

Table 1. Threshold values used in the experiments

	LS LP	LS SP	LS MP	LD LP	LD SP	LD MP
threshold	1.5	1	1	3	2	2

Tables 2 and 3 contain the results obtained for this experiment. For each combination, I give average precision, recall, and f-measure as well as the respective variance, all as percentages. The best values are indicated in boldface.

Table 2. Results of first experiment – type level

	avg. P	var. P	avg. R	var. R	avg. F	var. F
LS / LP	39.64	12.42	53.36	10.22	39.52	8.28
LS / SP	47.89	12.64	61.41	8.94	45.67	7.58
LS / MP	48.25	12.78	**63.98**	9.06	**46.75**	7.84
LD / LP	31.77	12.32	54.09	8.73	33.16	8.20
LD / SP	**55.87**	13.59	41.62	6.55	42.86	6.78
LD / MP	55.38	13.98	42.59	6.68	43.16	6.97

On the type level, it can be seen that algorithms matching by individual letters perform much worse than those approaches that use a graphemic splitting of the strings. The difference between ***simple parse*** and ***multi parse*** in contrast is not that big. Using the more complicated ***multi parse*** still results in a slight increase in average recall. All combinations surpass the baseline by almost 9 percent on the f-measure. LS/LP did not find a result for three queries, LS/SP did not return any document for two queries, as did LS/MP and LD/LP. LD/SP was the only combination that found a document for all queries. LD/MP did not return documents for one query.

The results on the type level mirror those on the token level. The combination of Low Saxon Distance and ***multi parse*** is still the best method according to recall and f-measure. However, the difference between LS/MP and the next best method LD/SP is slighter on the token level than on the type level.

Table 3. Results of first experiment – token level

	avg. P	var. P	avg. R	var. R	avg. F	var. F
LS / LP	45.86	10.91	70.03	11.10	48.42	6.91
LS / SP	56.14	12.46	76.74	9.61	57.35	8.93
LS / MP	55.99	12.57	**77.73**	9.90	**57.64**	9.23
LD / LP	37.36	11.02	71.22	9.76	41.16	7.19
LD / SP	**63.08**	10.18	62.55	9.53	57.10	6.68
LD / MP	61.93	10.83	62.72	9.49	56.54	6.93

The increase in the f-measure compared to the baseline is not that high, more important though recall has increased significantly. On average about 64 % of all variants of the query term are found with the best method LS/MP, which means that about 78 % of all documents containing a variant of the query term are retrieved. However, the loss in precision which is greater for the combinations using the Low Saxon Distance is problematic for user satisfaction. In section 5.3, I present a first practical solution which makes the whole system more useful for actual users.

But first, I wanted to try a lower threshold value for the best approach LS/MP in a second experiment in order to see whether a better precision could be attained without too great losses in recall. I used a threshold value of 0.5. This resulted in an average type-level precision of 71.67 % (variance 10.49 %), recall of 41.61 % (variance 8 %), and an f-measure of 49.97 % (variance 5.72 %). On the token level, precision reached 75.36 % (variance 9.05 %), recall was 58.24 % (variance 10.86 %), and the f-measure averaged 62.25 % (variance 7.35 %). We thus see a high increase in precision and less dramatic losses in recall. For fairness' sake, I should have tried to recalibrate LD/SP, too. However, time did not permit it.

In the third experiment, I wanted to test at least one approach that used a coarse search strategy based on the initial segment of the query term. I decided to use the initial grapheme class as a good middle course, i.e. all terms in the lexicon which had a graphemically equivalent first segment as the query term were returned by the coarse search algorithm. Evaluating the combination LS/MP using this coarse search strategy resulted in an average type-level precision of 50.88 % (variance 12.73 %), a recall of 63.96 % (variance 8.69 %), and an f-measure of 49.07 % (variance 7.99 %). On the token level, average precision reached 59.16 % (variance 12.22 %), recall showed an average of 77.30 % (variance 10.39 %), and the f-measure averaged 60.18 % (9.42 %). It seems that this coarse search method is preferable to the

coding method because of its better precision. However, see section 5.2 for a discussion of this conclusion.

In the final experiment, I tested a semi-automatic method of searching that allows the user to throw certain matched lexicon terms out from the initial results obtained by the method LS/MP as used in the first experiment, i.e. with a relatively high threshold of 1. This allows the user to quickly exclude obvious false positives on the type level and thus to refine the overall quality of the returned result set. In this experiment, I acted as the user myself which is not optimal, because I have also collected the human gold standard against which the different approaches are evaluated in this paper. I therefore assume that the results obtained in this experiment are better than those for an average user that has not worked on variation in Low Saxon for some months. However, as far as possible, I tried to base my decisions on the first impression of the list of matching lexicon terms ignoring knowledge gained during the collection of the gold standard data, i.e. I did not exclude plausible candidates although I knew that they were false positives in my test corpus. Unfortunately, time constraints and lack of speakers in the local area prohibited a long user test. In this experiment, I allowed one refinement cycle without actually looking at the retrieved documents themselves. The results I obtained are: 82.02 % type-level precision (variance 5.05 %), 61.63 % type-level recall (variance 9.84 %), and 68.51 % type-level f-measure. On the token level, the experiment resulted in an average precision of 88.19 % (variance 4.67 %), a recall of 74.96 % (variance 10.86 %), and an f-measure of 78.72 % (variance 7.81 %). In my opinion, these results are quite encouraging.

Finally, I would like to give the reader an overview over the time and space consumption of the different approaches. Table 4 shows the amount of space needed for the index into the lexicon in addition to the size of the Lucene index built for the test document collection. As expected, storing multiple parses requires the highest amount of additional space almost tripling the size of the original plain Lucene index. Additional space could be saved by not using code search as coarse search method which makes it unnecessary to save codes for the parses of the lexicon terms.

Table 4. Space requirements

	Only Lucene	Letter Parse	Simple Parse	Multi Parse
Space	3.07 MB	6.29 MB	5.91 MB	8.51 MB
Percent Lucene	100 %	205 %	193 %	277 %

Table 5 shows the average time in milliseconds needed for all six combinations from experiment one, and the two approaches from experiments 2 and 3. The fastest approach uses the Levenshtein Distance on strings that are simply split at each letter. The more effective approaches consume a lot of time and are currently at the limit of acceptability with an average of 15 seconds for the combination LS/MP.

Table 5. Average computing time

Test	Time in ms	Test	Time in ms
LS/LP	2692 ms	LD/LP	350 ms
LS/SP	5855 ms	LD/SP	427 ms
LS/MP	14561 ms	LD/MP	1350 ms
exp. 2	12399 ms	exp. 3	17465 ms

5.2 Discussion of results

The results described in section 5.1 show that linguistic knowledge does indeed increase retrieval performance. Most importantly, a graphemic parsing of the compared strings – even quite a crude one – already leads to a substantial increase in retrieval quality.

Whether a linguistically motivated cost function increases retrieval performance is not as clear. The results in the preceding section seem to support this conclusion. However, the difference in the f-measure between Levenshtein Distance and Low Saxon Distance is not very big. This is largely due to a lower precision exhibited by the combinations using the Low Saxon Distance. The lower precision is a consequence of the crudeness of the information encoded in the graphemic and phonemic classes: Firstly, the current version of the Low Saxon Distance collapses short and long vowels, because they often cannot easily be distinguished. However, this distinction is crucial for many graphemic and dialectal alternations. Secondly, another weakness of the current graphemic and phonemic classes is that they assume alternations to be independent of the position of the relevant segments in the word or the syllable. This assumption is clearly false and leads to a considerable loss of precision by often allowing variants that contain a certain alternation at the beginning of the syllable which in reality only occurs at the end of a syllable and vice versa. This also seems to be the reason why using a coarse search based on the graphemic class of the first segment of the query terms as in experiment 3 increases precision. Consonants at the beginning of Low Saxon words unlike those at the end of a syllable are usually quite stable, i.e. show little variation across dialects and writing systems. As a large majority of the test query terms begin with a consonant, experiment 3 which did not allow pronunciation variants for the first segment (it was based on graphemic classes) returned less false positives and still almost matched the recall of LS/MP in the first experiment. All this suggests that the linguistic knowledge used for the calculation of the Low Saxon Distance is often still too crude. One idea would be to use a coarse search based on the starting segment for query terms that start with a consonant and code search for those that begin with a vowel.

The answer to the question of whether one should use a simple or more elaborate edit distance algorithm as well as the question of whether one should use nondeterministic graphemic parsing largely depends on time and space constraints. Unless more effective ways of calculating a linguistically informed edit distance such as the Low Saxon Distance can be found, the faster retrieval speed is an argument in favor of more simple approaches.

5.3 Proposal for usable system

As a demo, I implemented a web interface using Java servlets which allows users to try out the combination: code matching for coarse searching and Low Saxon Distance calculation on strings split by multi parse for fine searching. As a practical system, it allows users to enter a query term and choose from four different threshold values, the lowest of which means that the query is send on to Lucene as is without searching for variants. On the generated result page, a list of all matched variants is displayed. Below that all documents that have been returned as hits by Lucene are listed, each with a list of the matched terms it contains. Departing from this initial result, the user can choose to exclude matched terms from the result by deactivating checkboxes which are provided for each matched term. The user can thus step by step refine the set of retrieved documents by examining some of them and consequently excluding more matched terms if necessary. In my opinion, this system is already quite useful for exploring the orthographic and dialectal variation of Low Saxon or simply for finding documents dealing with certain issues. However, as I also state on the demo webpage, users should be a little more patient and tolerant while trying out the demo than when they use commercial search engines.

6. CONCLUSION

The results obtained in section 5, clearly show the usual trade-off between precision and recall. The question that should be investigated is thus whether for the application that this paper envisages, i.e. a web search engine for Low Saxon, more importance should be put on recall or on precision. The best way to decide this question would be to conduct a study of user satisfaction.

The results described in the preceding section seem promising but they are very likely suboptimal and further research should be able to improve on these initial results. Currently, I see two main routes of improving the system.

First, a good way to estimate optimal cost functions and thresholds has to be found. Probably, the length of the compared strings should be taken into account for calculating the threshold value. The calibration of the threshold values performed by me for the different experiments was based on the overall impression of the results obtained with each different threshold. Clearly, the whole process of deciding whether a term in the lexicon approximately matches a query term can be considered a classification problem. Therefore, the next step should be to use some kind of machine learning to obtain optimal cost functions in order to make false positives and false negatives as separable as possible and to find an optimal threshold for the classification. Alternatively, one could try to devise a method of combining the usual document relevance ranking with a ranked distance of the matched terms in the document from the query term(s).

Second, if one believes that more linguistic knowledge can help to further improve retrieval performance, one should try to find a better way of assigning costs to substitutions, insertions and deletions. For example, as already remarked in section 5.2, the current system does not distinguish between the positions of segments (i.e. graphemes or phonemes) in a word or in a syllable. Many of the alternations encoded in the CharacterClasses data structure used by the Low Saxon Distance cost function totally disregard these distinctions which results in a substantial decrease in precision. Finding a way to include more linguistic knowledge into the cost calculation could help to remedy this problem. But at the same time, it is absolutely vital to speed up the retrieval for these more elaborate matching algorithms in order to reach acceptable retrieval times.

I would be quite happy if some time in the future a robust web search engine for Low Saxon could be implemented for the benefit of the whole Low Saxon web community and possibly as a model for other non-standardized languages or dialects.

7. ACKNOWLEDGMENTS

I would like to thank Wang Lam for valuable advice on how to use Java servlets for the web interface, Hinrich Schütze for helping me with linear algebra, the Linguistics Department at the University of Bochum, Germany, for hosting the search engine prototype on their web server, and especially André Halama for setting it up.

8. REFERENCES

[1] The Ethnologue. http://www.ethnologue.com/.

[2] Erikson, K. (1997): Approximate Swedish name matching – survey and test of different algorithms. Nada report TRITA-NA-E9721 and Master's thesis in computer science, Stockholm.

[3] Zobel, J., and Dart, P.W. (1995): Finding Approximate Matches in Large Lexicons. Software --- Practice and Experience, Vol. 25 No. 3, pp. 331 – 345.

[4] Zobel, J., and Dart, P.W. (1996): Phonetic String Matching: Lesson from Information Retrieval. In: Proceedings of the 19th International Conference on Research and Development in Information Retrieval (Zurich, Switzerland). ACM Press, pp. 166 – 172.

[5] Jurafsky, D., and Martin, J.H. (2000): Speech and Language Processing. An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Prentice Hall, Upper Saddle River.

[6] Manning, C.D., and Schütze, H. (1999): Foundations of Statistical Natural Language Processing. The MIT Press, Cambridge Massachusetts.

Test query terms used during the evaluation:

Randomly chosen: *barg* (mountain), *bekannt* (known), *blau* (blue), *bruun* (brown), *dodenvagel* (bird of death), *fell* (fur), *foot* (feet), *geld* (money), *gruglichs* (horrible), *herrn* (lords, gentlemen), *heven* (sky, heaven), *hübsch* (pretty), *hymne* (hymn), *krediet* (loan), *leeigen* (bad), *leeive* (dear), *nooborship* (neighborhood), *pappa* (daddy), *riemels* (short poems), *schinken* (ham), *schoyn* (beautiful), *tügen* (witnesses), *upptog* (procession), *wiehnachten* (Christmas). **Picked from a single Radio Bremen news report**: bundesregeern (federal government), egaalweg (always), fredag (Friday), gendarmes (police men), kark (church), künnig (known), orrig (quite), rebeed (area), verdeffendert (defends), vörstand (chair person).