Språkbehandling och datalingvistik

Projektarbeten 2005

Handledare: Pierre Nugues och Richard Johansson

LUNDS UNIVERSITET

Institutionen för Datavetenskap

http://www.cs.lth.se

Printed in Sweden
Lund 2006

Innehåll

LUNDS UNIVERSITET

Institutionen för Datavetenskap

http://www.cs.lth.se

Protein Name Extraction

Xavier Adam
adamxavi@utt.fr

Olle Rundgren
Department of Computer Science
Lund University
olle.rundgren.913@student.lu.se

Abstract

This article describes a simple information extraction system using pattern recognition. The intended use of the program is to extract protein names from research articles. The article gives a short introduction to the background of our project and then goes on describing the implementation and structure of our system. Finally we evaluate our system and discuss what conclusions can be made from our work.

1 Introduction

Within the area of biotechnology and the life sciences there are vast amounts of information generated every day. The need for organizing this information is therefore of biggest concern and any method of automating this task is a welcome one. The FetchProt[1] and the Yapex[2] projects of the Swedish Institute of Computer Science (SICS[3]) adresses this need of automation and they were the starting point of our project. The FetchProt project is an ongoing project building a complete system. It will be able to extract proteins and their functions and inserting these into a database for wider use, both academic as well as industrial use. The Yapex project is the protein name tagger that FetchProt uses for the extraction of the protein names. We were interested in the information extraction part of the project and therefore decided to build a small system similar to the Yapex project. Our aim was to build a system that matched the results of the Yapex system using a pretty simple approach of pattern recognition.

[1]http://www.sics.se/humle/projects/fetchprot/
[2]http://www.sics.se/humle/projects/prothalt/
[3]http://www.sics.se/

2 Implementation and Structure

2.1 Language

We used Java for our implementation partly because of personal preferences but mostly because of its wide use and possibilty to find free modules to use from within our program. As it turned out we only used standard Java but found resources that would have been interesting to incorporate into our program. A specifically useful package was the standard package for regular expressions.

2.2 Corpus

The Corpus we used was downloaded from the Yapex site and was relatively small. Since it was already tagged we had to clean it from all tags before we could use it. The Corpus was originally collected from research articles from the MEDLINE resource and consists of the abstracts of randomly chosen articles. The training collection contains 1745 protein names and the test collection contains 1966 protein names. All of the abstracts were originally annotated by domain experts connected to the Yapex project.

2.3 System

Our system has a direct and simple structure. When given an input text we create a new object that we chose to call a BioText which holds a dictionary that can store protein names and a buffer to store the text. On this object we can call all methods associated with the different tasks. If the text needs to be cleaned from tags we call the cleaning method and store the new clean text in the buffer instead. For the task at hand (tagging of Protein names) we have two different methods that both takes regular expressions written as strings as parameters and which both fills the dictionary with protein names. The two methods differ in that one of them is for applying more specialized patterns catching few protein names with a high precision while the other method is for applying a broad pattern cathing lots of protein names. Because of this implementation choice we could experiment with the patterns to make them better in a fast and easy way. We also have some methods to help us in analyzing the effectiveness of the patterns we write. For example we have a contextwriter that writes the context of each of the tagged protein names so that we can analyze what needs to be done. For the evaluation we had methods that could be applied directly to our BioText objects to compute recall and precision.

2.3.1 Different types of patterns

The most important part of our program is the patterns which detects the protein names to be tagged. They are all applied using the Java standard package for regular expressions though we wrote convenient methods to be able to just write the patterns as strings. At first we tried to apply all-catching patterns that catches a lot of protein names and we got a lot mismatches and didn't get all the names either. Therefore we realized that we had to build different kinds of patterns to get better results. The method we settled for was to have one broad pattern whith a filter to get rid of mismatches and several high precision patterns that catches pretty few names but with a higher precision (fewer mismatches). This choice resulted in two different methods which made it easy to experiment with different patterns. The first method is used for the high precision patterns and is typically used to detect multiword protein names. The second method is used to apply the broad pattern and also includes a filter to avoid common mismatches of names that look like protein names but in fact are not proteins for example DNA, RNA, UV and PH. This way to implement the pattern recognition was pretty fruitful since it was easy to write new patterns and to test and analyze these new patterns. The ability to test new patterns fast was very important to us since we are no experts on proteins and the structure of the names are not completely standardized. There were also the occurence of similar names that had to be taken into account like DNA and the introduction of the filter was a very important improvement and could easily be extended if other recurring mismatches were found. This iterative process was typical for our workflow. Both of the methods also had functionality to avoid tagging the same proteins more than once.

2.3.2 Main Parts of the Biotext class - The dictionary and the Buffer

As mentioned shortly in an earlier section each Biotext object holds a dictionary to store each occurence of protein names and a buffer to store the text to be tagged and later the tagged text. The dictionary was implemented with a map and has the protein names as keys and the number of occurences as values. When a pattern is applied with the appropriate method each protein that the pattern catches is stored in the dictionary. The Biotext objects also incorporates a buffer to store the text to be tagged. After each pattern applied the buffer changes accordingly with the proteins that the pattern has caught tagged in the buffer. When all patterns are applied the dictionary contains all the protein names that has been tagged and the buffer contains all the tagging of these protein names. Then different methods can be called to compute recall, precision and the result file can be written to a file. This makes the system easy to extend, improve and test and was also done several times before we reached the final results. The actual running of the program was as implied done from a seperate testclass.

3 Evaluation

For the evaluation we used methods for computing of recall, precision and using those two it was easy to make a printout for the harmonic mean (which is a combination of the two). We first built a baseline case which only tagged proteins in the test corpus which had been stored in the dictionary after applying patterns on the training corpus. That is we didn't apply any patterns at all on the test corpus. This gave us pretty poor results which was as expected. After that we applied patterns to the testcorpus as well and the results got better so we knew we were on the right track. We then wrote better patterns for the training corpus . The more patterns added the better it got and we used the evaluation methods to see how effective new patterns were. The wide pattern was the one with the biggest increase of recall and writing of the specialized patterns only made smaller increases but still good increase. It was also during early evaluation that we became really aware of deficiencies in our patterns that tagged names that shouldn‘t be tagged and therefore the evaluation process became an iterative process done several times before acceptable results were reached. Finally we settled for a couple of high precision patterns and the filtered broad pattern and ran them on both the training corpus and the test corpus. The results including the original baseline case can be viewed in the table below.

	recall	precision
base	10.04%	45.08%
train	56.19%	52.24%
test	55.5%	52.6%

Table 1: Results

For further evaluation it would have been interesting to run some arbitrary but relevant articles through the system as well but due to time restrictions that was never done.

4 Conclusions

As with any project you are left with lots of thoughts about things that could have been done better and things that you are pretty happy about afterwards. With the methods we used we don't think we could do much better. Yes, we could write more of the specialized patterns but there wouldn‘t be any bigger point since the system isn‘t going to be used and increases after a while becomes very small. If the system were to be used for a ”real” project we realize that we would have to do better than we have done so far but in that case we would need other methods to combine with the ones we have used. We looked into adding some kind of part-of-speech tagger and had actually started writing some methods for that as well. After a while though we felt that it was better to concentrate completely on one method since we didn't think we would make it in time otherwise. That was probably a wise choice. We experienced some trouble with the reading and understanding of the texts that we were working on. It was not an easy task to read and analyze text so out of context and so highly specialized. Finding effective patterns therefore became more difficault than we first thought. All in all it was very good training with the whole project since we got to adress problems that is probably very common in natural language processing and to work freely with something always inspire. It was also interesting to see that we could reach fairly good results with rather simple means.

5 Acknowledgements

We would like to thank both Pierre Nugues and Richard Johansson for the help we received during the project.

6 References

- http://www.sics.se/humle/projects/prothalt/
- http://www.sics.se/humle/projects/fetchprot/
- An Introduction to Language Processing with Perl and Prolog, Pierre Nugues

Building a Swedish rhyme dictionary

Rasmus Arnling Bååth
Department of Computer Science
Faculty of Science, Lund University
rasmus.baath@gmail.com

Staffan Åberg
Department of Computer Science
Institute of Technology, Lund University
carnevorum@gmail.com

January 17, 2006

1. Abstract

This report describes an attempt to build a Swedish rhyme dictionary. The project is part of the course *Language Processing and Computational Linguistics* given at *Lund university* the spring term 2005. To be able to rhyme with a word you have to know the pronunciation and so the authors conclude that a phonetic lexicon is needed. Since no comprehensive phonetic lexicon is readily available the first step is to build one out of LEXIN, a small but free lexicon. The second step is to implement a search engine and the requirements of such an engine are discussed.

2. Introduction

A rhyme is when two or more words partially sounds the same. There are many categories of rhymes, and what category a rhyme belongs to depends on what part of the words are similar. When there's a similarity in the beginning one speaks of alliteration. Alliteration is common in old Norse writings and here is a famous stanza from the poetic Edda:

“**V**red **v**ar **V**ing-Tor,
när han **v**aknade,
och sin **h**ammare,
han saknade.“

When there's similarity in the middle one speaks of assonance, e.g. *g**å**ta-m**å**la.* The most common type of rhyme is when there's a similarity in the end. This is called a tail rhyme and it's the type of rhyme this report deals with. For the rest of this report a rhyme is always a tail rhyme. Rhymes can be both monosyllabic and polysyllabic, *gr**is**-par**is*** being the former and ***österrikare**–tr**österikare*** being the latter. Rhyming serves two purposes. It gives a text rhythm and flow, hence pop lyrics nearly always rhyme, and it makes a text easier to remember, the fact that nursery rhymes often are very old is proof of this. Rhyming is an old practice and there are examples of rhymed Christian hymns dating back to 300 ad.

Rhyming is not always easy and sometimes you would want a rhyme dictionary. This report describes an attempt to build one, supposed to be comprehensive, easy to use and accessible over the web. There already exists a number of rhyme dictionaries on the web, but they are all flawed in the same way. They only consider the spelling of a word when searching for rhymes. This is an understandable design choice. It's easy to implement, you just write an algorithm that picks out words from a word list that matches the end of a given word. It's easy to extend, you just add new words to your word list. Though good in many ways it also gives rise to some problems. Even though Swedish spelling is relatively consistent, at least compared to English, there are many exceptions. A rhyme dictionary that only considers spelling would miss perfectly good rhymes such as *fl**eece**-gr**is*** and include *dam-skam*, which doesn't rhyme. When rhyming it is also important consider stress, something that's hard to deduce from the spelling of a word. To cope with such problems one has to consider the pronunciation of the words, not only the spelling. The easiest way to do this is by using a phonetic lexicon when searching for rhymes. One could also attempt to devise rules to derive the pronunciations from the spelling, but this is difficult and a perfect solution is probably AI-complete.

Since no free and comprehensive phonetic lexicon for Swedish is readily available, the task to build a rhyme dictionary is twofold. First one has to get hold of a phonetic lexicon then one has to build a search engine to search it for rhymes. The rest of this document is organized in the following way: Section 3, describes how we take LEXIN, a lexicon aimed at emigrants learning Swedish, and by means of several Perl scripts make it into a phonetic lexicon implemented as a MySQL database. Section 4 discusses the requirements and the implementation of the search engine. Section 5 shortly discusses the interface of the rhyme dictionary. Then follows an evaluation and some concluding comments. Finally there's an appendix consisting of a short explanation of the phonetic alphabet used, an excerpt from LEXIN and a picture of the web interface of the lexicon.

3. The Phonetic Lexicon

3.1 . Lexin

Swedish is a small language and it's not strange that Swedish linguistic resources aren't as great as for larger languages, such as English. If you're looking for an English phonetic lexicon you would find Moby Pronunciator[1], a large and free lexicon consisting of 175,000 words with corresponding pronunciation. A similar lexicon for Swedish doesn't exist. What does exist is LEXIN. LEXIN is a lexicon built by the Institution of Swedish language at the university of Gothenburg and it's free to download from Språkbanken[2], an online collection of linguistic resources. LEXIN is intended as an aid for emigrants learning Swedish and consists of roughly 20,000 entries. Every entry consists of a lemma, a pronunciation, a part-of-speech tag, a list of inflections, a list of compound words and possibly a description of the usage of the word. LEXIN is stored in the form of an XML document, see appendix A for a short excerpt and appendix B for a description of the phonetic alphabet used.

Before we can use LEXIN as a phonetic lexicon some changes has to be made. First of all LEXIN has to be made into a MySQL database. XML is a good format for structured data, but by using a MySQL database you gain access to powerful tools such as fast search mechanisms and consistency checks. Secondly LEXIN has to be cleaned up. The way the spelling and the pronunciation is notated is inconsistent and the XML document is not valid nor well-formed. There's also a lot of information in LEXIN that we don't need. We don't need to know what part-of-speech a word belongs to or the usage of a word. The phonetic notation is also unnecessarily complicated. Thirdly LEXIN has to be extended. LEXIN's mere 20,000 entries are not enough, for instance the dictionary compiled by the Swedish Academy contains over 120,000 entries.

3.2 . Making LEXIN into a MySQL database

To make LEXIN into a MySQL database we use Perl since it has a good database interface and good tools for working with text. First we transform LEXIN into a flat file, this being easier and faster to work with than a XML file. Unnecessary information such as information about what part-of-speech a word belongs to is not included. The flat file is then made into a MySQL database and at the same time parentheses in the spelling and pronunciation of a word, notating alternate spelling or pronunciation, are removed. The LEXIN database, hereafter only called the database, now consists of four tables. One table consisting of lemma's with corresponding pronunciation, two tables with inflections and compounds, respectively, without corresponding pronunciation and one empty table to hold inflections of compounds.

1 http://www.dcs.shef.ac.uk/research/ilash/Moby/

2 http://spraakbanken.gu.se/

The database now has to be cleaned up and made more consistent. The notation of words and pronunciations contains elements that we don't need. Eg. "~" that separates parts of multiwords is removed and in the pronunciation ":" after consonants is removed. ":" after a consonant means that the consonant is long and this is not important to consider when rhyming. Monosyllabic words, that for some reason aren't stressed, are made stressed. Inflections can both be complete words, as in [*bagare :: bagaren*], or partially spelled out, as in [*bagare :: -n*]. We want all inflections to be complete words and we therefore complete partially spelled out inflections.

3.3 . Extending the Database

The database now is in the form we want it to be in but it's still not very comprehensive and we would want to extend it. To add completely new words is out of the question since this can't easily be automated. What can be automated is adding pronunciation to the roughly 37,000 inflections. The information we have at our disposal is this: the spelling of the lemma, it's pronunciation and the spelling of it's inflections. We first produces a list of all the differences, hereafter called suffixes, between an inflection and it's lemma. E.g. [*bagare :: bagaren*] would produce "*n*" and [*fadern :: fäderna*] would produce "*äderna*". It turns out that this list consists of under 200 different suffixes and so it's possible to enter their pronunciation by hand. The script for producing the inflections' pronunciations then works in the following way. For each inflection pick out the suffix, the spelling in common with the lemma and the rest of the lemma, e.g. [*driva :: drivor*] would produce *("or", "driv", "a")*. Then extract the common pronunciation by removing the pronunciation corresponding to the rest of the lemma from the lemma's pronunciation, in this case *"drI:va"* would become "*drI:v*". Then the pronunciation corresponding to the suffix is concatenated with the common pronunciation, and so *"drI:v"* + *"or"* becomes *"drI:vor"*. This simple method works, but some extra rules are needed. In for example [*dö :: dött*] the *ö* is long in *dö* but short in *dött,* so it's not enough just to add the pronunciation of the suffix *"tt"*. This problem can be solved by checking if adding the suffix to the common spelling leads to new double consonants, if this is the case make the preceding vowel short. Other tricky inflections that has to be dealt with are the ones ending with *"orer"*. E.g. [*dator :: datorer*] are pronounced respectively *"2dA:tor"* and *"2da:tO:rer"*. Not only is the *"o"* made long, it is also made stressed.

The method works incredibly well and produces 36,000 new inflection-inflection pronunciation pairs. When checking 100 randomly picked inflection-inflection pronunciation pairs only one was not correct.

Each lemma also has a number of compound words associated with it. The compound words can be divided into two categories, those that ends with it's lemma and those that starts with it's lemma. We can't produce the full pronunciation of the compound words but we can produce a partial by using the lemma's pronunciation. Since our database is going to be used as a rhyme lexicon we are mostly interested in the pronunciation of the end of the words. We therefore only produces partial pronunciation for the compound words that ends with it's lemma. We introduce *"§"* as a wild card sign indicating where we don't know the pronunciation. A compounds pronunciation is then simply it's lemmas pronunciation with a "*§*" added in front, e.g. [*väggalmanacka :: §almanaka*] and [*överansträngning :: §anstre@ni@*]. This produces 3,500 new word-pronunciation pairs. We make the uppercase letter, indicating stress, into lowercase in all the partial pronunciations since the stress most often lies in the first word of the compound words. We also generate the inflected forms of all the compounds with partial pronunciation producing another 8,000 word-pronunciation pairs. We now have a phonetic database consisting of 63,000 word-pronunciation pairs.

4. Searching for rhymes

4.1 . Requirements of a search engine.

The strict definition of a rhyme is a word that corresponds with another one from and including the last stressed syllable. A search engine using this definition could work as follows: Accept a word as input, get that word's pronunciation in the database, remove everything from the pronunciation up to the last uppercase vowel and return a list of words which pronunciation ends in the same way as whats left of the search word's pronunciation. But this engine is naïve in many ways and wouldn't suffice. Here's a list of features a search engine should have.

1. You should be able to change the strictness of the search. The strict definition of a rhyme is in many cases to strict. Sometimes, especially when writing lyrics for a song, the stress doesn't matter that much.
2. The search engine should be able to handle the fact that many words are spelled the same but pronounced differently. E.g. [*banan :: 2bA:nan*], the one you can safely walk on and [*banan :: banA:n*], which you wouldn't want step on.
3. You should be able to search for rhymes for a word not currently in the database. This could be solved by making the search engine accept a phonetic transcription as input, but this approach would make the search engine less user friendly.
4. The search result should be presented in a perspicuous way. Often you now how many syllables the rhyme your looking for should have. A search engine returning a randomly ordered list or a list sorted in lexical order would make it hard to find rhymes with a certain number of syllables.
5. The search engine should be able to handle an incomplete database. Searching for rhymes with a word the database only has partial pronunciation for should still produce a passable result.

4.2 .Implementation of the search engine

Our search engine will be implemented in Perl and will use the MySQL database as it's source of information. The search engine is not intended to be used by end users, and will later be made easier to use by means of a web interface. To understand the choices of design we have made one must first know in what way the output is sorted. First of all the output is sorted by how good it matches the search word, that is how many syllables matches the search word's. E.g. if the search word was *analogi*, *astro**logi*** would be before *byråkrat**i***. Then the output is sorted by number of syllables and finally in lexical order.

Our search engine will take a word as input and have four customizable options: strict_rhyming_matters, stress_matters, vowel_length_matters and table_to_use. If the first, strict_rhyming_matters, is on, the search engine will produce rhymes according to the strict definition of a rhyme. If it's off, any word that has any syllables up to the end of the word pronounced the same way as the search word will be considered to rhyme. If this option is on it would mean that *anan**as**-kaleb**ass*** would be considered a rhyme, even though it is the first syllable of *ananas* that is stressed. The words still has to be stressed the same way, thus *ananas-hölass* wouldn't rhyme. If the second option, stress_matters, is off, the search engine won't consider stress. A*nan**as**-höl**ass*** would be considered a rhyme and so would *mak**aker**-gröns**aker***. The third option, vowel_length_matters, decides whether the vowel length should matter when rhyming. If vowel_length_matters is of, word pairs than normally aren't considered rhymes,because of differences in vowel length, are. For example *kan**el**-pens**el*** would be considered a rhyme. The last option, table_to_use, governs which parts of the database to use when searching for rhymes. By changing this you can for example make the search engine search only for lemmas or inflections.

The search consists of two steps. First a phonetic transcription corresponding to the search word is extracted from the database. Then the phonetic transcription is used to search the database for words that rhyme.

4.2.1 .Getting the phonetic transcription

When trying to extract the phonetic transcription there are two scenarios. First, if the search word is already in the database, the corresponding pronunciations are returned. Secondly, if there's no perfect match, the search engine will try to match with any word that ends with the search word. If there's still no match, the search engine will successively shave away letters from the beginning of the search word and search after words ending with the now shortened search word until there is a match. If the match returns many words the word with the length most similar to the search word is chosen. That word's pronunciation is then shortened to have as many syllables as the now shortened search word. That pronunciation is then returned. An example: The word *mandelmassa* doesn't match anything in the database and so letters are shaved away from the beginning until a match is made. When *mandelmassa* has become *massa* it matches three words in the database. The word with the most similar length is [*pappersmassa ::2pAper+smasa*]. Syllables are removed from it's pronunciation to match the number of syllables in *massa* and thus *masa* is returned. Instead of doing this second scenario search one could devise rules that derives the pronunciation from the search word. But since it's the pronunciation of the end of the words that matters when rhyming and we already have a large database containing information about how words sound in the end, this would be a bad approach. Words spelled the same in the end nearly always sound the same in the end.

4.2.2 .Getting the rhymes

Now when we have the pronunciations of the search word we're going to use them to search the database for rhymes. For each of the pronunciations the following is done. If the pronunciation wasn't the result of a perfect match, strict_rhyming_matters and stress_matters are switched off. That's because the stress of the partial pronunciation probably isn't correct. We then search for words with pronunciations that ends like the search word's pronunciation from the first vowel. The search result is then sorted in lexical order and by number of syllables in ascending order.

If the search word's pronunciation is partial we will probably have unwanted rhymes in the search result. E.g. when searching with [*mandelmassa :: masa*] we will get everything rhyming with *assa* when we really would like to get everything rhyming with *andelmassa.* Our solution to this problem is to remove every word from the search result that doesn't match, from the first vowel, what's left of the search word when we remove from it, from the end, as many syllables as there is in it's pronunciation. In the case of [*mandelmassa :: masa*] *mandelmassa* is shortened to *mandelm* and every word that doesn't match, from it's first vowel, with *andlem* is removed. This approach is a return to the method to use a words spelling to search for rhymes, but since we only have a partial pronunciation this is the best we can do.

We then remove the first syllable from the search word and remove syllables from the beginning of it's pronunciation until it has equal or less syllables. We then search once more. If strict_rhyming_matters is on the search stops when there's no stressed syllable left in the search words pronunciation. Else we continue searching until there's no syllables left in the search word. The result of the searches are consecutively added to a list. When the search is finished we have a list of rhymes sorted by number of matching syllables in descending order, number of syllables in ascending order and lexical order. Finally the search word and duplicate words are removed from the list .

This method of search seems to work pretty well, though a problem is that it's not very fast. When searching with a search word that already is in the database, the database is accessed about 3-6 times. When searching with a search word that's not in the database, the database could be accessed as many times as the number of letters in the search word. Accessing the database is slow and therefor the search method is slow.

The strategy to use the spelling to weed out rhymes when the search word only has a partial pronunciation can be discussed. There are many cases when a word isn't pronounced as it's spelled but in contrast to only using the spelling to search for rhymes we here don't use it to match the end of the word. There we use the partial pronunciation. This approach will never remove a correct rhyme, the rhyme will only occur in the wrong place in the search result.

5. The Interface

We now has a working rhyme dictionary but it's not accessible nor very user friendly. To use it one would have to set up ones own MySQL database spending a long time building it by running various perl scripts. Then one have to learn rudimentary perl syntax to use it, something nobody should be forced to learn. Therefore we made a web interface (see appendix C). The interface is a HTML form with an underlying cgi script that handle the search queries. The form consists of a field for entering text and two options. The text field is of course used for entering the word to rhyme with. The first option governs the strictness of the search, that is it governs which of the three options strict_rhyming_matters, stress_matters and vowel_length_matters that should be switched on. Here follows the different setups of the four levels of strictness :

strikt = (true, true, true)
normal = (false, true, true)
utan betoning = (false, false, true)
nödrim = (false, false, false)

We could of course have allowed the three options to be changed directly, but we feel that our approach is more intuitive. The other option governs what database to use when searching for rhymes. The user can choose between *standard* and *utökad. Standard* contains all the words which pronunciations are complete, that is all the original lemmas and their inflections. *Utökad* is like *standard* plus all the words with partial pronunciation, that is the compound words and their inflections. The *Utökad* database also contains words submitted through the homepage.

The search result is presented in columns, one for every number of syllables that was matched. If a search is made with the strictness level *normal* it turns out that the column containing words that only matched one syllable contains far to many words to be easily browsed. The use of columns allows the user to first check the column with the "best" rhymes and then, if necessary dive into the other "worser" columns.

The homepage also contains a form that allows the user to add new words to the database. At the moment users can't change words already in the database. This functionality should be added in the future so that users could fully participate in extending and correcting the lexicon.

6. Evaluation

We started this report by complaining about that existing on-line lexicons uses spelling and not pronunciation to search for rhymes. We argued that the former approach misses many rhymes and includes words that doesn't rhyme. The question one ask oneself now is; how good does our rhyme dictionary perform in comparison other dictionaries. Though it's hard to find anything to measure some things could be pointed out.

The fact that our database only consists of 63,000 entries cripples our dictionary. 63,000 may sound a lot but one should remember that most of these are inflected forms of the 20,000 lemmas. Other on line rhyme dictionaries, e.g. DbLex[3], have databases with over 400 000 words. Our dictionary also suffers from some blind spots, e.g. our database contains no names of geographical locations. Where our lexicon excel is when the *strikt* search mode is used. A *strikt* search produces only rhymes according to the strict definition of a rhyme, that is rhymes that rhyme perfectly. No other rhyme dictionary on the web is able to do something similar.

7. Conclusion

This project has led to the creation of a working rhyme dictionary that uses the pronunciation of a word when searching for rhymes. This makes our dictionary unique(we believe) since all other Swedish rhyme dictionaries uses the spelling of a word when searching for rhymes. The dictionary's phonetic database was created out of Lexin, a small but free Swedish dictionary. Considerable amounts of time have been spent transforming Lexin into a usable database. This was not anticipated when the project started. Even though Lexin isn't very comprehensive we feel that our dictionary is usable and that it perform well in comparison to other existing dictionaries.

One thing that could be done to improve our dictionary further is to make it into a proper wiki. This could help solving the problem with our insufficient database. At the moment it's just possible to enter new words, not alter old ones.

Our dictionary is currently not accessible over the Internet since we haven't found anywhere to put it. The license of Lexin may also prohibit us from making our dictionary accessible over the

3 http://www.dblex.com/

Internet. If our dictionary finds somewhere to live a link will probably be found on the projects course homepage, http://www.cs.lth.se/DAT171/. The source code of the project and directions on how to set up your own rhyme dictionary should also be available from there.

8. Acknowledgments

We wish to thank Pierre Nugues for his support during this project.

9. References

Svenska ord/LEXIN vid Språkbanken, Göteborgs universitet. http://spraakbanken.gu.se/

Fredrik Hansson and Lennart Nilsson. 1996. *Rimlexikon*. ICA Bokförlag.

Bengt Sigurd. 1991. *Språk och språkforskning*. Studentlitteratur.

Staffan Bergsten. 2002. *Rim & reson*. Nationalencyklopedin. www.ne.se.

Appendix A

Lexin's DTD:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!--                                                            -->
<!--                     LEXIN                                  -->
<!--              Svenska ord, 2:a uppl.                        -->
<!--       Copyright Språkdata, Göteborgs universitet           -->
<!--       XML Markup: Yvonne Cederholm, Susanne Mankner        -->

<!DOCTYPE lexin [
<!ELEMENT lexin (lemma-entry+)>

<!ELEMENT lemma-entry        (form, pronunciation, inflection, pos, lexeme*)>

<!ELEMENT  form                     (#PCDATA)>
<!ELEMENT  pronunciation     (#PCDATA)>
<!ELEMENT  inflection        (#PCDATA)>
<!ELEMENT  pos               (#PCDATA)>
<!ELEMENT  lexeme     (lexnr?, definition?, usage?, comment?,
                             valency?, grammat_comm?, definition_comm?,
                             example*, idiom*, compound*)>
<!ELEMENT  lexnr             (#PCDATA)>
<!ELEMENT  definition        (#PCDATA)>
<!ELEMENT  usage             (#PCDATA)>
<!ELEMENT  comment           (#PCDATA)>
<!ELEMENT  valency           (#PCDATA)>
<!ELEMENT  grammat_comm             (#PCDATA)>
<!ELEMENT  definition_comm   (#PCDATA)>
<!ELEMENT  example           (#PCDATA)>
<!ELEMENT  idiom             (#PCDATA)>
<!ELEMENT  compound          (#PCDATA)>
```

A sample of Lexin:

```
<lemma-entry>
       <form>geometri</form>
       <pronunciation>jeometrI:</pronunciation>
       <inflection>geometri(e)n</inflection>
       <pos>subst.</pos>
       <lexeme>
       <definition>vetenskapen om de matematiska rumsstorheterna</definition>
       </lexeme>
</lemma-entry>
<lemma-entry>
       <form>ger med sig</form>
       <pronunciation>je:rmE:(d)sej</pronunciation>
       <inflection>gav gett (el. givit) ge(!)</inflection>
       <pos>verb</pos>
       <lexeme>
       <definition>acceptera något (efter påtryckning), foga sig</definition>
       <valency>A &amp;</valency>
       </lexeme>
</lemma-entry>
<lemma-entry>
       <form>gerilla</form>
       <pronunciation>2gerIl:a</pronunciation>
       <inflection>gerillan gerillor</inflection>
       <pos>subst.</pos>
       <lexeme>
       <definition>motståndsrörelse, icke reguljära trupper</definition>
       <compound>gerilla~soldat -en</compound>
       <compound>gerilla~verksamhet -en</compound>
       </lexeme>
</lemma-entry>
<lemma-entry>
       <form>gest</form>
       <pronunciation>$es:t</pronunciation>
       <inflection>gesten gester</inflection>
       <pos>subst.</pos>
       <lexeme>
       <definition>(hand)rörelse, åtbörd</definition>
       <usage>bildligt "handling avsedd att visa en persons känslor etc"</usage>
       <example>livliga gester</example>
       <example>en tom gest</example>
       <compound>försoningsgest</compound>
       </lexeme>
</lemma-entry>
```

Appendix B

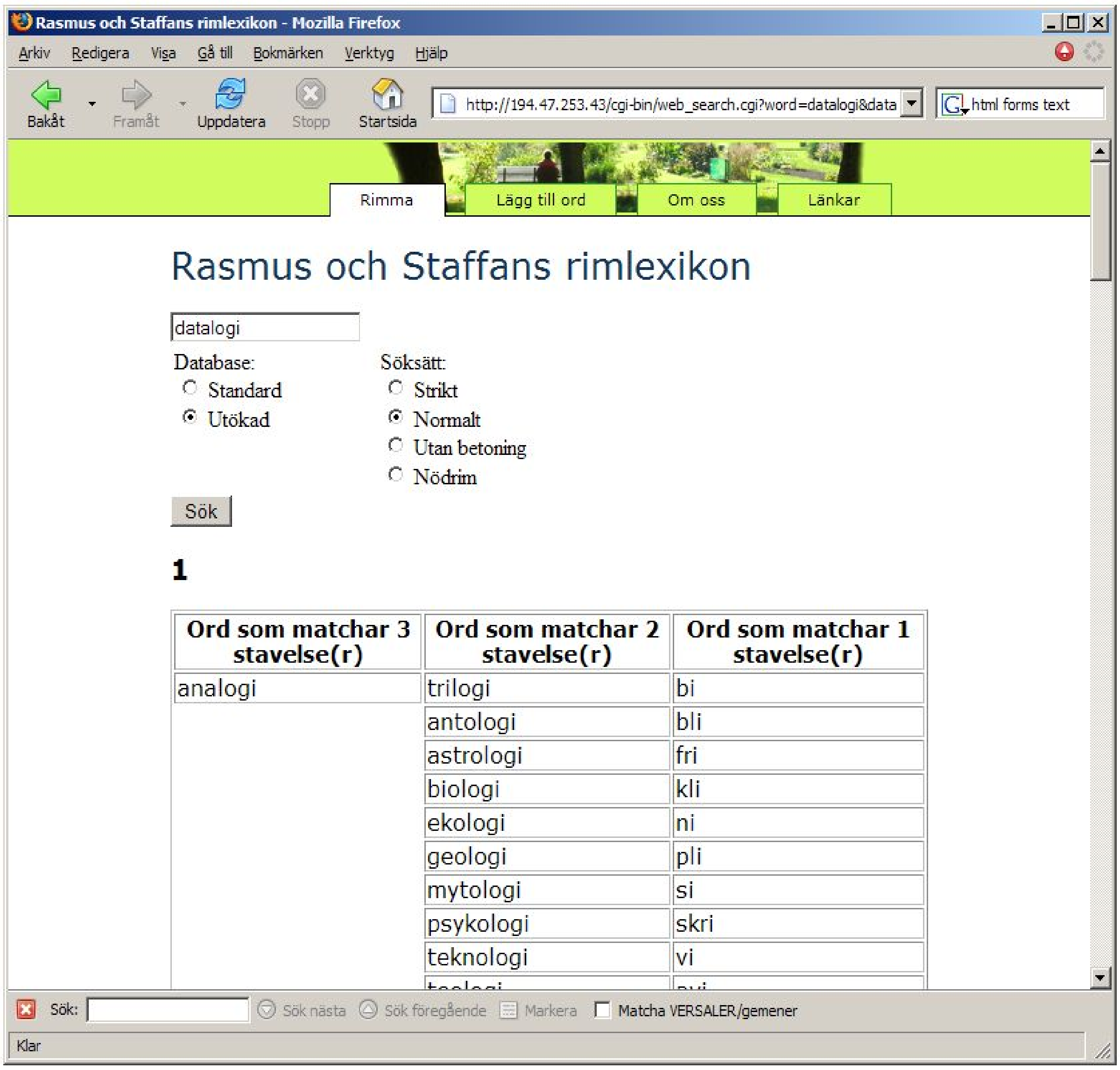
Here follows a brief description of the phonetics used in our dictionary.
The following consonants have the pronunciation usually associated with them:

[*b, d, f, g, h, j, k, l, m, n, p, r, s, t, v*]

Consonants missing are: [*c, q, w, x, z*]
The pronunciation of these consonants can be expressed using the consonants above.
E.g. [*cirkus :: sIrk*us], [*yxa :: Yksa*] and [*zoo :: sO:*].
The following vowels have the pronunciation usually associated with them:
[*a, o, u, å, e, i, y, ä, ö*]
A vowel can be long or short. A [:] after a vowel indicates the former, a vowel without [:] indicates the latter. E.g. [*ful :: fU:l*] and [*full :: fUl*].
There are three symbols that represent pronunciation not usually associated with them: [*c, $, @*].
[*c*] represent the way the *k* sounds in ***k****är*. [*$*] represent the way the *stj* sounds in **stj***ärna*. [*@*] represent the way the *ng* sounds in *spri***ng***er*.
[*2*] in the beginning of a word indicates grave accent.
[+] between two symbols indicates that the two symbols pronunciation is melted together. E.g. between two vowel this would indicate a diphthong.
What syllable in a word that is stressed is indicated by a capital vowel.

These phonetics are nearly identical to the phonetics used by Lexin, with one major exception. In Lexin [:] could also be used after consonants to indicate long pronunciation.

Appendix C


Rasmus och Staffans rimlexikon - Mozilla Firefox
Arkiv Redigera Visa Gå till Bokmärken Verktyg Hjälp
Bakåt Framåt Uppdatera Stopp Startsida
http://194.47.253.43/cgi-bin/web_search.cgi?word=datalogi&data
html forms text
Rimma
Lägg till ord
Om oss
Länkar
Sök: Sök nästa Sök föregående Markera Matcha VERSALER/gemener
Klar

Rasmus och Staffans rimlexikon

datalogi

Database: Standard [✓ Utökad]

Söksätt: Strikt [✓ Normalt] Utan betoning Nödrim

Sök

1

Ord som matchar 3 stavelse(r)	Ord som matchar 2 stavelse(r)	Ord som matchar 1 stavelse(r)
analogi	trilogi	bi
	antologi	bli
	astrologi	fri
	biologi	kli
	ekologi	ni
	geologi	pli
	mytologi	si
	psykologi	skri
	teknologi	vi

The result of a search using our rhyme dictionary.

Detection of similarity between documents

Axel Bengtsson
Department of Computer Science
University of Lund
axel.bengtsson@gmail.com

Ola Olsson
Department of Computer Science
University of Lund
ola@matematik.nu

Abstract

This document describes an implemented GUI application for detection of syntactic similarities between documents.

1 Introduction

Similarities between documents is interesting in many different kind of areas. The purpose can stretch from different kind of areas such as:

- Let the application choose articles such that we don't read the same kind of article twice.
- Easily detect cheating at assignments.
- Easily detect changes between two revisions of papers, source code etc.

2 How to detect syntactic similarities between documents

To detect similarities, we choose to implement a vector based algorithm called the cosine similarity. This algorithm lets all document represent a vector in the space. To see if two texts are equal or near equal, they should have a cosine similarity as near 1 as possible.

3 The program

Our program consists of three modules, Topic detection and tracking, TDT which is a module that counts the document vectors, LCS, the longest common subsequence counter and the third module which is the GUI.

The first module TDT, reads the articles and counts the word frequency. Then it calculates the weights of the words and counts the cosine similarity described above. All the words are included in this calculation except words described in "stoplist.txt". It returns an ordered list with all the pairs of articles in descending order of the rank. [1] A XML-file will be created at this time with all the weights and word frequency that the document contains. The name of the XML-file is the same as the filename and then the suffix ".XML" is added.

The GUI starts up and calls the LCS with the two documents with highest rank. Every time the up/down button is pushed, a new call to LCS will be made.

3.1 Class diagram

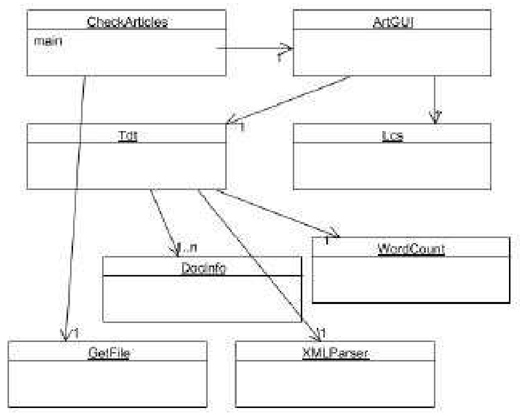


Figure 1: *Representation of the classes.*

3.2 TF-IDF

To give every word in the text a weight, we implemented the TF-IDF term frequency, inverted document frequency algorithm. This is seen as (Downie, 1997)

$$Weight(w_{ij}) = f_{w_{ij}} \times log_2 \left(\frac{N}{n_{w_i}} \right)$$

Term	Meaning
w_{ij}	i:th word in document j
$f_{w_{ij}}$	Frequency for word w_{ij}
N	Total number of documents
n_{w_i}	Number of documents w_i occurs in

[1] This is the vector analysis number, where 1 is very close to each other and 0 is not.

This is trivial math but does explain some effects of the formula:

- If a word occur in every text, the weight for that word will be zero in every document.
- Two words can have different weights, it depends on which document it is in. The log term is constant in this sense but the frequency of the word in the document may differ.
- If a word only occur in one text that certain word will (of course) get $f_{ij} \times log(N)$ which is the biggest weight a word can get.

This means that, the more a word appears in all the texts, the less weight it will get. To get a high weight, the word should appear very often in as few texts as possible. When the TF-IDS has selected the two articles which is most equal, we are running these articles through a LCS algorithm. This algorithm detects which words are the same in both articles and paints these red in the GUI. Observe that the words found in the algorithm doesn't have to be consecutive. We have slightly modified the well known recursive algorithm in two ways, first we have made it iterative and secondly, we are running it through words instead of single characters.

3.2.1 Cosine similarity rate

After the TF-IDF algorithm, every word in every document has got a weight. Now we define the similarity between two documents as:

$$Rate(X,Y) = \frac{\sum_{w_i \in (X \cap Y)} x_{w_i} \times y_{w_i}}{\|X\| \times \|Y\|}$$

Term	Meaning
w_{ij}	i:th word in document j
X	A document
Y	A document
x_{w_i}	The weight of word w_i in X
y_{w_i}	The weight of word w_i in Y
$\|X\|$	$\sqrt{x_{w_1}^2 + x_{w_2}^2 + \ldots}$
$\|Y\|$	$\sqrt{y_{w_1}^2 + y_{w_2}^2 + \ldots}$

First, the program looks up all words which appears in both documents. The word weights are multiplied together and sums up for each word. This sum is divided by the product of both document norms. This formula will run through all $\binom{n}{2}$ pairs of documents.

3.2.2 LCS

The LCS is often based upon a recursive algorithm. Imagine two strings, $X =< x_1,...,x_i >$ and $Y =< y_1,...,y_j >$. The recursive solution is based on the fact that if x_i and y_j is the same character, then the LCS of the two strings is the same as the LCS of $x_1,...,x_{i-1}$ and $y_1,...,y_{j-1}$ and then add x_i to the solution.
If $x_i \neq y_j$, we say that the LCS of X and Y is the maximum length of one of the two subproblems $LCS(X, y_1,...,y_{i-1})$ and $LCS(x_1,...,x_{i-1},Y)$. This means that if the last two characters are not equal, the problem can be reduced to two subproblems, one subproblem that runs LCS with whole Y and deletes the last character from X and the other subproblem runs LCS with whole X and deletes the last character from Y. This follows cause we are comparing two strings and if their characters don't match, then one of the characters is worthless.

The base of the recursion is when the argument to the algorithm is the empty string, then the algorithm returns the empty string.

$$LCS(i,j) = \begin{cases} ”” & if \quad i=0 \quad or \quad j=0 \\ LCS(i-1,j-1)+x_i & if \quad x_i = x_j \\ max(LCS(i-1,j),LCS(i,j-1)) & otherwise \end{cases} \quad [2]$$

However, this method is really slow because of the recursive steps. The worst case happens when $i = j$ and when the strings doesn' t have a common character at all. Of course, in this case the strings can have a maximum length of the number of characters in the alphabet. Anyway, the worst case calls the last row in (3.2.2) all the time which splits the problem in two parts. In the worst case, this call will be made i times. This means that the time complexity of the solution is $O(2^n)$ [3] which is totally unacceptable in our program. Of course some of these subproblems are the same and can easily be treated by memoization (probably using a hash) but we agreed on implement it iteratively. To find out how to make an iterative solution we can gain information from the recursive solution. If we think of the problem as a matrix:

	S	T	R	I	N	G	1
S							
T							
R							
I							
N							
G							
2							

The recursive solution starts at the right-bottom corner of the matrix and checks whether the charac-

[2] In this document and context the operator plus is overloaded as concatenation of strings, and the function max returns the argument which has the longest string.

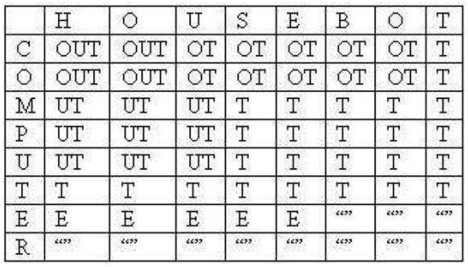
[3] Here, n is the length which is equal to i and j

ters are the same or not. If they are the same, the cell$_{i,j}$ is equal to cell$_{i+1,j+1}$ + whats in cell$_{i,j}$. Else, the cell$_{i,j}$ will be equal to max$(_{i+1,j}\ _{i,j+1})$.

If we make a matrix of the two strings and follow the recursive solution of this matrix, then we simply see that the recursive solution can be written as two for-loops which starts in the right-bottom corner and works its way up to the left-upper corner where the solution will be stated.

```
for(int x=i;x>=0;++x)
{
for(int y=j;y>0;++y)
{
if (X[x]==0 || Y[y]==0) ResultMatrix[x][y]="";
if (X[x]==Y[y])
ResultMatrix[x][y] = ResultMatrix[x+1][y+1] + X[x];
else
ResultMatrix[x][y] = max(ResultMatrix[x][y+1],ResultMatrix[x+1][y]);


}
}
```

This means that a certain cell in the matrix is either a 0 which means that the letters are not equal, or the letter itself + the letter (or string) in the cell down one step and then one step to the right. This makes the solution a $O(n^2)$ in time which is much better than $O(2^n)$. An example of how our algorithm calculates the LCS of the strings "HOUSEBOT" and "COMPUTER".



	H	O	U	S	E	B	O	T
C	OUT	OUT	OT	OT	OT	OT	OT	T
O	OUT	OUT	OT	OT	OT	OT	OT	T
M	UT	UT	UT	T	T	T	T	T
P	UT	UT	UT	T	T	T	T	T
U	UT	UT	UT	T	T	T	T	T
T	T	T	T	T	T	T	T	T
E	E	E	E	E	E	""	""	""
R	""	""	""	""	""	""	""	""

Figure 2: *Note that the result from this algorithm does not provide us a valid result for substrings of these strings. To get that, we should run the algorithm forward instead of backward.*

4 Screen shots and test runs

The program requires Java 2 Standard Edition 5.0 and can be started as follows:

```
java CheckArticles
```

A file chooser window will appear. You choose your multiple articles by pressing the control key (or shift). [4] The program reads and calculates data and a progress bar will guide you through this step. After this, a GUI will appear where the words appearing in both articles in same order are marked red. The articles that will be shown at the start of the GUI are the 2 articles of $\binom{n}{2}$ [5] that got the highest rating with our TF-IDF algorithm. When using the up/down buttons you will go through the documents in ascending/descending rank order.

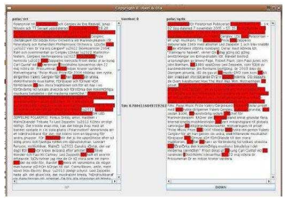


Figure 3: *Two documents are shown and the red marked text is the LCS.*

5 Quality assurance

To see whether our functions are work as they should we made some test cases and predicted the result before running them on our program. The main items we wanted to test is the cosine similarity function and the LCS.

To test them, we made three files containing this information [6]

File1

```
Hello everybody. this is a test Ola and Axel.
```

File2

```
Hello anyone. What may be the deal.
```

File3

```
Hello anybody. Great program Ola and Axel.
```

Before we ran these files in the program we expected four things.

1. The word "Hello" must get word weight 0 in every document because $log_2 1 = 0$.
2. File1 and File3 is the pair of files which should get the highest cosine similarity rate. This is because of the words "Ola and Axel".
3. File1 to File2 and File2 to File3 are the rest of pairs and these pairs are worth nothing.

[4]Files have to be chosen from the same directory.

[5]If n is the total number of documents, this will be the number of pairs

[6]Dont care about the semantics of the sentences, it is just test cases.

4. The LCS of File1 and File3 should be ”Hello Ola and Axel” because the other words isnt contained by the other string respectively.

We ran all the tests and the predicted results were correct.

Another test to run through the LCS is at text which contains a word only one, against the same text but reversed. The LCS algorithm should display one word in red. Imagine the text:

```
This is a test.
```

If we run this text against the reverse

```
.test a is This
```

, the red marked word could be any of the words in the two sentences depending on how one has implemented the *max* function. The important thing is to understand that not two words will be red. This is because of the reversing text. Suppose that the strings are build like $X =< x_1, x_2, ..., x_n >$, the other string Y will look like $< x_n, x_{n-1}, .., x_1 >$. Suppose we find a word which must be in our LCS. Say x_k where $k \in 1 \to n$. Then, this word must be the word y_{n-k} in Y. After some iteration we find a new word to be in our LCS, say x_l where $l > k$. This means that this word is found in y_{n-l} where $l < k$. Because $l < k$, it means that the second word is before the first word which is not possible in LCS.
The word which got red in our program was: ”a” This is because we didnt implement a max function ourselves.

6 Statistics

To get a good understanding of how good our application is, we tried it on several documents, some of them were intended to give some result while other documents were copied directly from newspapers. We chose two subjects from the newspapers that have been pretty large in the last weeks, namely ”wilma” and ”the bird disease”. 10 documents from each subject were collected from DN, Aftonbladet, Expressen, Sydsvenskan, TT.se and other big news sources. First, both of us, independently of each other ranked all the pairs of documents based upon the LCS and give the pairs a grade between 1-5. After that we compared our ratings and they were exactly the same but one or two pairs. We ran the articles through the program and the result was that almost every document got a TDT rating very good compared to our LCS rating except those pairs where one of the document was much larger than the other one.

7 Conclusions

We think that the assignment was perfect in time measure. We also liked the subject and we had absolutely no problem with coding whatsoever. The problem was to get the idea of how the TF-IDF works but Pierre explained it very good.

What we could conclude from the statistics is that the program is very good at finding similar documents when the papers are approximately in the same length. Otherwise, if one of the documents are a proper subset of the other but far more smaller, the TDT will be very low. Maybe it is good, maybe not, it depends on the purposes. It is now possible for a student to copy another students paper and keep on writing without the notice of the TDT, but the LCS will cover it, if the teacher will search through all the pairs graphically.

Something notable is also that it was the first time we used CVS and that worked very good as well.

8 Acknowledgments

We would like to thank Pierre for the help and for assistance with books and Rolf Karlsson who sent us the lecture notes of Dynamic Programming (Karlsson, 2005).

References

J. Stephen Downie. 1997. Term weighting: tf*idf. Webpage, September24 . http://instruct.uwo.ca/gplis/601/week3/tfidf.html.

R. Karlsson. 2005. Lecture 5: Dynami programming.

Automatic Article Generator from Extracted Databases

Ianick Boudreault
Lund University, Sweden
ianboudreault@hotmail.com

Abstract - The primary goal of an automatic content generator is to bring available new information to the public by the means of search engines. In fact, nowadays there is a rush for accumulating a lot of data which is not always "humanly readable", mostly formatted into databases. The recent success of search engines has revolutionized our way to search for information, allowing them our trust in delivering us the most relevant results to our search requests. The goal of this paper is to present a tool to manipulate into articles such unformatted sources of data so it can be indexed by search engines ready for the end users to consult through their searches.

1. Introduction

Humanly readable information: this seems to be the new fashion in nowadays best search engines to deliver what they believe to be the most relevant results to its users. Wondering why they have attained such standards makes us wonder what the end user really wants. Will a database containing, for example thousands of species of insects be a better alternative then reading a websites presenting articles on each of these species? The answer to this question makes us believe that articles are much more convenient to read. However the main advantage of articles is that search engine will "like" such formatted data and will rank them in their results. This information will then be available to searches through specific keywords like "Hymenoptera bugs" or "Orthoptera insects". The optic of this tool is to get available information to the public that wouldn't be available through the World Wide Web.

2. The ContentBot tool

Contentbot is the name given to the article generator tool built here. Its goal is to build unique articles automatically with a template built by the user with its inbuilt template creator tools. It mainly works in two main operations: a template sentence creator containing "holes" to put the database entries and a synonym creator to make the articles more unique. An XML derived syntax is used to insert the desired information at the right places in the template sentences. An extraction tool is also available to extract web information aimed at creating database from online websites. In this paper, examples will be made with a beer database containing 1600 entries extracted from a list of beers taken on the web.

2.1 The ContentBot Interface

ContentBot has been built as a web application tool to guide the user through the steps of creating new articles from its raw database. The tool has been built to ease the creation of this template that will then be used to generate the articles. It handles adding, modifying and deleting sentences and synonyms while presenting the information in a way to make it easy to the user to write his articles. All the information is stored in a database on the server until the articles are generated and downloaded to the user. In this way, a user can create many projects and finish them later on. When the process is finished and the articles generated, the result can be viewed through the ContentBot result navigator and then be downloaded to the client station in different formats.

2.2 The ContentBot Extraction tool

The beer database presented as an example in this paper has been extracted with the ContentBot extraction tool. This tool is a script that basically finds unique HTML information before and after the desired information. This unique code has to be identical through each pages of the website to extract the same information in each beer page. Doing so for each data desired, we run the script to extract each information for each page of the website. This has been done on a beers website to furnish this database of 1600 beer type, having seven relevant information of each beer: the beers name, the style and sub style, the country, the brewery, the level of alcohol and a score assigned to the beer. For the moment, the database needs to be in the form of a single table, it will later be extended to support relational databases.

2.3 The Sentence Creator Tool

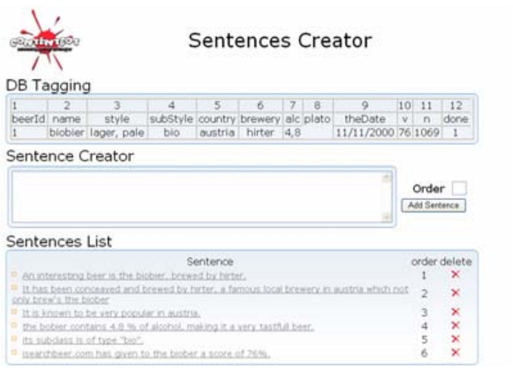
This is the first step to create the template. The tool first presents the first row of the database on top of the page as a resource for writing the sentences. For the beer database, we have the following items:

- Name: biobier
- Style: lager, pale
- Sub style: bio
- Brewery: Hirter
- Country: Austria
- Alcohol level: 4.8 %
- Score: 76 %

We then write an article as if it was written about the current row. Sentences are added one by one as they will be treated separately, for example:

> *"An interesting beer is the biober being classed as a lager, pale."*

The sentence creator tool is presented here with already composed sentences.



Sentences Creator

DB Tagging

1	2	3	4	5	6	7	8	9	10	11	12
beerId	name	style	subStyle	country	brewery	alc	plato	theDate	v	n	done
1	biobier	lager, pale	bio	austria	hirter	4,8		11/11/2000	76	1069	1

Sentence Creator

Sentences List

Sentence	order	delete
An interesting beer is the biobier, brewed by hirter.	1	✗
It has been conceaved and brewed by hirter, a famous local brewery in austria which not only brew's the biober	2	✗
It is known to be very popular in austria.	3	✗
the bobier contains 4.8 % of alcohol, making it a very tastfull beer.	4	✗
its subclass is of type "bio".	5	✗
isearchbeer.com has given to the biober a score of 76%.	6	✗

The next step is to add the tags related to the database so that the sentences don't depend anymore on the first row but on the assigned column. Each column is identified by a unique id. Adding the id to the tag is of the form:

<-DB item=2 /DB->

During the final generation of the article, this tag will make the system replace the tag for the correct entry in the database. Here it would change *<-DB item=2 /DB->* for biobier. Replacing each tag on the example sentence will look like:

"An interesting beer is the <-DB item=2 /DB-> being classed as a <-DB item=3 /DB->"

2.4 The Synonym Creator tool

The article that would be produced from the only use of the Sentence creator tool would result in 1600 articles having the same text for each beer. That would result in a really bad website and our main goal to rank in search engines would fail since all major search engines use what is called "similar content filters" on websites. This means that having 1600 sentences with a high percentage being the same text would be detected and the website containing the articles would probably be dropped. This is where the synonym creator tool gets interesting as it is used to catch sections of a text and allocate synonyms. Having many synonyms for different parts of the sentence will result in a high level of diversity for each sentence during the final generation.

The Synonym generator asks the user to insert in the sentences "synonym tags" enclosing the desired part to work with. Using the same sentence as before for the example, the tag would look like this:

"<-syn item=a1->An interesting beer<-/syn-> is the <-DB item=2 /DB-> being classed as a <-DB item=3 /DB->"

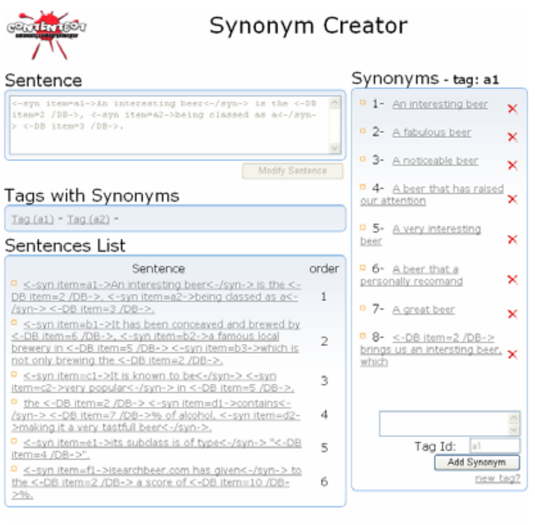
The id of the tag can be anything as long as it is made in one word. The synonym tool now lets the user add as much synonyms for the "a1" tag as wanted. We could for example add:

- *A fabulous beer*
- *A noticeable beer*
- *A beer that has raised our attention*
- *A beer that we would recommend*

We can add an infinite amount of synonym in a sentence. The more synonyms there is the highest are the chances of having all different sentence. The different possibilities then go exponential. For example for a sentence having three synonym tags in a sentence having eight synonyms each, this would make:

$$8^3 = 512$$

512 possibilities for this sentence alone. The interface of the synonym creator tool with the completed tagged sentences is presented below:



```
<-syn item=a1->An interesting beer<-/syn-> is the <-DB item=2 /DB->, <-syn item=a2->being classed as a<-/syn-> <-DB item=3 /DB->.
```

Sentence	order
<-syn item=a1->An interesting beer<-/syn-> is the <-DB item=2 /DB->, <-syn item=a2->being classed as a<-/syn-> <-DB item=3 /DB->.	1
<-syn item=b1->It has been conceaved and brewed by <-DB item=6 /DB->, <-syn item=b2->a famous local brewery in <-DB item=5 /DB-> <-syn item=b3->which is not only brewing the <-DB item=2 /DB->.	2
<-syn item=c1->It is known to be<-/syn-> <-syn item=c2->very popular<-/syn-> in <-DB item=5 /DB->.	3
the <-DB item=2 /DB-> <-syn item=d1->contains<-/syn-> <-DB item=7 /DB->% of alcohol, <-syn item=d2->making it a very tastfull beer<-/syn->.	4
<-syn item=e1->its subdass is of type<-/syn-> "<-DB item=4 /DB->".	5
<-syn item=f1->isearchbeer.com has given<-/syn-> to the <-DB item=2 /DB-> a score of <-DB item=10 /DB->%.	6

1- An interesting beer
2- A fabulous beer
3- A noticeable beer
4- A beer that has raised our attention
5- A very interesting beer
6- A beer that a personally recomand
7- A great beer
8- <-DB item=2 /DB-> brings us an intersting beer, which

The example sentence with the proper synonym tags would like this:

"<-syn item=a1->An interesting beer <-/syn-> is the <-DB item=2 /DB->, <-syn item=a2->being classed as a <-/syn-> <-DB item=3 /DB->."

Several tools are available on the internet to help finding synonyms and ways to say things differently such as the Prinstons wordnet[1] tool and synonym dictionaries.

2.5 The generation of the articles

The generation tool then uses a random number to select the good synonym for

[1] WordNet : http://wordnet.princeton.edu/

each tag and adds for each of the articles the correct database entry. In our example, we have a total of six sentences, having a total of eleven synonym tags. To find out the amount of different sentence possibilities we multiply the amount of synonym per synonym tags together to find:

$$7 \times 5 \times 6 \times 5 \times 4 \times 3 \times 5 \times 6 \times 4 \times 4 \times 5 = 30\,240\,000$$

This makes a total of more than 30 million possibilities. Having 1600 articles to generate the chances a same article appears twice is

$$30\,240\,000 / 1600 = 18\,900$$

This makes a probability of 0.06 % which is extremely low.

3. Conclusion

The results generated by ContentBot are very interesting as each article really seems like they have been written by a real author. These articles can then be added to a website that will be indexed by the search engines, ready for user consulting. A parallel project of mine will actually use the information generated by this beer example. This project called ISearch[2] is using article formatted information to attach to its search engine and making the information searchable through the pages of a new built website. The information is then available in the isearch site, which will be called www.isearchbeers.com. The articles will finally be easley spiderable through the major search engines and will be available to be found using keywords in the articles. Now someone searching for "*merlin's pilsener beer*" will probably find our article on the merlin's pilsener!

[2] Examples of Isearch at:
www.isearchquotations.com
www.isearchjokes.com
www.isearchbible.com

Annex A: Example of the beer articles generated

A beer that a personally recommend is the pilsner urquell, knowing to be a lager, pale. We appreciate this beer, thanks to plzn (pilsen), a famous local brewery in czech republic well known for its other conception than the pilsner urquell. This famous brewery is known to be loved in Czech Republic. the pilsner urquell contains an amount of 5,0% of alcohol, which makes it an interesting beer. It is also described as a "pilsen". isearchbeers.com has given to the pilsner urquell a score of 80%.

A beer that has raised our attention is the merlin's pilsener, knowing to be a lager, pale. All the credits are accorded to bextrim, a talented brewery from germany which is not only brewing the merlin's pilsener. This famous brewery is known to be very successful in Germany. The Merlin's pilsener contains an amount of 4,9% of alcohol, making it a very tasteful beer. It is also characterized as a "pilsen". We have accorded to the Merlin's pilsener a score of 70%.

An interesting beer is the dachsenfranz kellerbier, characterized as a lager, pale. It is actually brewed by Herbert Werner, an award winning brewery coming from Germany having much more to know than the dachsenfranz kellerbier. This beer is known to be very successful in Germany. The dachsenfranz kellerbier is powered by an amount of 5,2% of alcohol, enough for a good night of fun.

isearchbeers.com has granted to the dachsenfranz kellerbier a score of 70%.

A noticeable beer is the warsteiner, knowing to be a lager, pale. All the credits are accorded to warsteiner, a famous local brewery in Germany which is not only brewing the warsteiner. It is known to be loved in Germany. The warsteiner includes about 5,0% of alcohol, enough to cheer you up. isearchbeers.com has granted to the warsteiner a score of 86%.

Automatic Identification of Participants in Discussion Groups

Jakob Carlsson
Väpplingv 7
227 38 Lund
Sweden
dat04jca@student.lu.se

Bobo Wieland
Transtigen 39
262 41 Ängelholm
Sweden
bobo@bitbob.biz

January 23, 2006

Abstract

The idea behind this project was to see if you could create a system that could tell you who wrote a certain piece of text. The method that we used for this was to encode the text as numerical data with an id for each word followed by it's frequency in the text. The numerical data was then fed to an SVM that predicted the author of the text. This report briefly discusses information extraction from the internet and describes the thoughts behind the java application RUU.

1 Introduction

The project idea was to see if there was any possibility to create a system that could tell you who wrote a certain piece of text. It was an interesting area that not too many people had thought of. We early decided that the best way to get data to test the system was to download it from discussion forums on the internet. We also thought about how to implement the system and found that the easiest way would be to use Support Vector Machines (SVM) for the classification of the text.

At this point we started collecting test and training data for the project from discussion forums on the internet. At an early stage we decided that our first goal would be to have a system that could see the difference of two people in a discussion between only these two; our second, and final goal to see the difference between several people in a discussion between these and other people as well. At this point we also started writing our system and we needed a name for it, after a while the system was named RUU (pronounced: Are You You) because it can tell if a text realy is written by a certain person.

1.1 Support Vector Machines

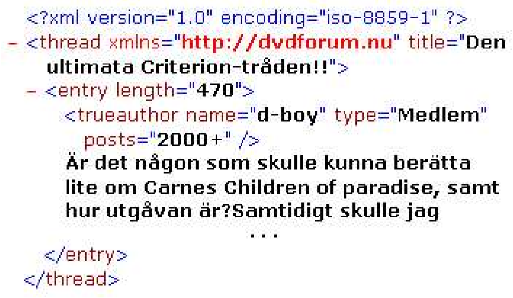
SVM is a method to classify data using vectors and mathematical models. We downloaded LIBSVM [1] and used it in the project because it is quite big to write your own SVM. Since we are beginners on using SVM for data classification we had to read through the beginners guide [3] to get a good grip of how to use SVM.

2 Application Structure

In the following section we will first briefly explain how we gathered and formated our test data (Section 2.1). We will then move on to explain in more detail how our main application - RUU - works (Section 2.2).

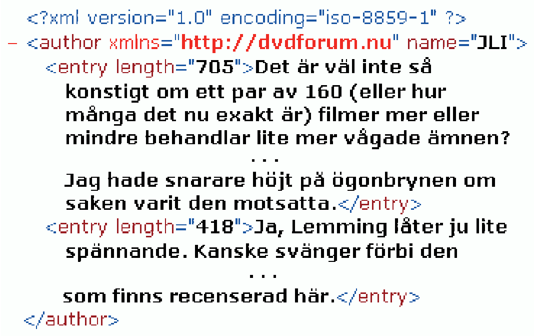
2.1 Information Gathering

We had some different web forums in mind when we started to work on this project but soon decided to use the swedish spoken forum on dvdforum.nu [2]. dvdforum.nu is a web site for movie enthusiasts and has many active members in their forums. It's been online since 1996 and is to some extent a closed



```
<?xml version="1.0" encoding="iso-8859-1" ?>
- <thread xmlns="http://dvdforum.nu" title="Den
    ultimata Criterion-tråden!!">
  - <entry length="470">
      <trueauthor name="d-boy" type="Medlem"
        posts="2000+" />
      Är det någon som skulle kunna berätta
      lite om Carnes Children of paradise, samt
      hur utgåvan är?Samtidigt skulle jag
                          . . .
    </entry>
  </thread>
```

Figure 1: Example of Criterion.xml



```
<?xml version="1.0" encoding="iso-8859-1" ?>
- <author xmlns="http://dvdforum.nu" name="JLI">
    <entry length="705">Det är väl inte så
      konstigt om ett par av 160 (eller hur
      många det nu exakt är) filmer mer eller
      mindre behandlar lite mer vågade ämnen?
                          . . .
      Jag hade snarare höjt på ögonbrynen om
      saken varit den motsatta.</entry>
    <entry length="418">Ja, Lemming låter ju lite
      spännande. Kanske svänger förbi den
                          . . .
      som finns recenserad här.</entry>
  </author>
```

Figure 2: Example of JLI.xml

domain since most discussions concerns movie related subjects. It suited our needs perfectly since we needed a long forum thread with many posts for our tests and as many posts as possible from forum members active in that particular thread.

After finding a suitable test thread, Den ultimata Criterion-tråden!! (The ultimate Criterion[1] thread!!) with >1100 posts over a period of four years, we created a simple PHP script to parse the thread data. The script simply looked at the HTML source and split the text by the tag pattern of the code. After stripping out all HTML tags the script saved the data in a convenient XML format as Criterion.xml (Figure 1).

By counting the number of posts from different users in Criterion.xml we could easily single out our test subjects. We will reference our four test subjects in the remaining of this rapport by their screen names - d-boy, JLI, ola-t and von Krolock.

When we knew the screen names of our test subjects it was easy to modify our existing PHP script to loop through the threads at dvdforum.nu, catching all posts by either subject and append the post to a specified XML file; one for each of our five subjects with their screen names as file names (Figure 2). At this point we also decided not to use posts consisting of fewer than 250 characters.

While being an easy and simple way of gathering data it wasn't the most efficient one. We started our loop counter at the then most recent thread id and went backwards in time from that point on, looping through each thread, existing or non-existing, and each page of each thread (in case it was spread out over many pages due to it's number of posts). After letting the script run for 24-hours straight and looping through approximately 50.000 threads we hoped that we had gathered enough data and aborted the information gathering.

After doing some labour-some manual edits to our test files, removing the signature each subject put last in all of their posts that unfortunately was impossible to remove automatically (at least with our simple PHP script), our information gathering was complete.

[1]The Criterion Collection is a line of authoritative consumer versions of "classic and important contemporary films" on DVD (and on Laser Disc pre DVD era). The quality of these releases - from picture and sound to packageing and included extras - are always top-notch.

2.2 RUU

RUU is our main java application that converts our XML files to files that can be used with SVM. It does not, however, simply convert from one format to the other but tries to format the text in the XML files to maximize the final svm prediction rate by apllaying some simple rules.

RUU does two passes over the supplied data, first building a dictionary of tokens and the in the second pass generating the output.

We will now explain in more detail four parts of the application; The Sink (Section 2.2.1), The Tokenizer (Section 2.2.2), the Dictionary (Section 2.2.3) and the svmFileCreator (Section 2.2.4).

2.2.1 Sink

RUU uses a SAX parser to parse the XML files. We choose to use a SAX parser rather than a DOM parser since we, at the time of the decision, didn't know how large our XML files where to be and if they would be well-formed or not. In contrast to a DOM parser a SAX parser reads the XML files a bit at the time. This means that the whole XML document will never be in the computers memory (which could cause problems if the documents are huge) and it also guaranties that until the parser encounters an error in the XML syntax it will parse the data. A DOM parser would abort the attempt to read the XML document immediately.

The second reason to use SAX rather than DOM was the simplicity of our XML files and the knowledge that we would only use the parser to read the data - not to manipulate it.

The Sinks main purpose in our application is to handle the data sent from the SAX parser. In it's basic form it had three important methods for handling XML data; one for handling data sent when a XML tag is opened, one for handle data when a tag is closed and one for handling character data.

The actions the Sink takes is different depending upon what stage of the process RUU are in. In the first pass that RUU does over the files the Sink sends the data from all the documents to the Dictionary. In the second pass it sends the data to the svmFileCreator instead. In both cases all character data is processed by the Tokenizer.

2.2.2 Tokenizer

A regular tokenizer breaks a character stream into tokens - separate words - and sentences [4]. Our tokenizer breaks the string of words supplied by the Sink into tokens, but sentences are not generally taken into account. Before storing a token our tokenizer turns all regular characters into lower case and on top of this scans the input string for specific patterns and possibly add some of three special tokens;

1. #SMILEY# - Some regular ascii smilies - i.e. :) or ;-(- are recognized and are replaced with this token.

2. #NET_SHORT# - The most regular internet short forms for different expressions - such as lol (laughing out loud) or isf (swedish for i så fall (in that case)) - adds this token to the list. It does not, as with #SMILEY#, replace the old token.

3. #NON-CAPITAL# - While sentences are of no interest to us in general, the Tokenizer does check ff the first character of a sentence is in lower case and if that is the case adds this token to the list.

This special treatment of the input string is done since it will, supposedly, help SVM to predict who is who more accurate. To explain our thoughts behind this we have to see ourselves as long users of the internet. We've grown accustom to the way people express themselves and it feels naturally to categorize peoples use of words and symbols.

Some users use a lot of smilies in their post - some use non. Some use a lot of internet abbreviations - some, again, use non.

Also, to abuse the use of capitalized letters means, in net-language, to shout. And excessive shouting is often followed by excessive amounts of angry replays to the point where the original poster learns to fear Caps-Lock. This is why we treat the few capitalized words as a non-capitalized word since it won't be a net users regular way of writing.

Finally, many - but far from all - regular forum posters have the bad habit to, more often than not, forget to start sentences with a capitalized letter. If this is because of laziness or fear of Caps-Lock, we won't elaborate further on #SMILEY#.

2.2.3 Dictionary

In the first pass that RUU does over the XML files all tokens returned by the Tokenizer is sent to the Dictionary. The Dictionary gives each token a unique label - starting with 1 for the first token, 2 for the second and so on - and keeps track of the total number of times - it's frequency - a token is used throughout the XML documents.

We've also added the functionality to store bigrams (word pairs) instead of unigrams (single tokens), or to store both bigrams and unigrams at the same time, in the Dictionary. Since bigram and n-gram predictions themselves can be used to check authorship of texts this seemed a resonable thing to do. However - later tests showed us that using bigrams instead of unigrams gave a great (huge!) performance hit and a much worse final results, so we won't say much more on this matter.

Number of subjects	Method used	T-limit	C / g(e-5)	CV rate	Correct/ Tot(Valid)
2	Unigrams	1	128 / 12.21	94.84	71/83
2	Bigrams	1	32 / 12.21	92.46	42/83
2	Uni+Bi	1	32 / 12.21	94.05	42/83
2	Unigrams	2	32 / 48.83	94.84	70/110(83)
3	Unigrams	2	512 / 03.05	87.09	80/110
4	Unigrams	2	512 / 03.05	81.10	101/242
4	Unigrams	2	512 / 03.05	81.10	118/1090(242)

Table 1: Final results

2.2.4 svmFileCreator

After the Dictionary is populated RUU parses the XML files a second time. This time the Sink keeps track of the author of the entries (forum posts) in our test files giving them unique labels, once again starting at 1 for the first author, 2 for the second (and so on and so forth)...

After character data has been tokenized the tokens for that particular user and entry is stored along with the frequency for each of the tokens. In this case the frequency is the number of times the token has appeared in the current entry and not the total frequency stored in the Dictionary.

When the Sink encounters the end element of an entry the svmFileGenerator is used to generate an SVM suitable representation of the stored data for the entry in question. This data is appended to a train file, later to be used by SVM to build a prediction model.

When all test files are parsed and the train file is completed, the procedure is repeated once more for the XML file that should be used to test our SVM model. When the Sink encounters a entry in this file that has not been written by one of our test subjects it - depending on what we've chosen - either completely discards the entry or generates a new unique label for it (the Sink does not keep track of other authors of our test files so two entries posted by he user bitbob will not get the same label - in fact we increment the label value by one each time we come across a post that is not from one of our test authors)).

As before, when encountering the end element of an entry the svmFileGenerator comes into play; generating an SVM suitable representation of the data that gets saved to disc.

With the two files created - the train file for building the SVM model and the test file to test the model on - RUU is done and it is time to let easy.py give the CPU a run for it's money.

2.3 easy.py

Throughout the project we've followed the guide lines given in the paper A Practical Guide to Support Vector Classification [3] and their suggestion to use easy.py for fast and easy, as well as good, results.

easy.py is a python script that comes with the LIBSVM package. It simplifies the use of SVM by automatically fine tune your files and finding suitable values for constants used by the RBF kernel function. It creates the SVM model and and runs the model on a test file, if specified.

All of our result data is data given by easy.py

3 Results

While generating train and test files in RUU, we've tried some different approaches, just to see what gave the best result. In most of our early tests we've used only two of our test subjects; JLI and von Krolock (since we had most data from these two). After getting some initially poor final results we decided to skip the test on Criterion.xml and use a subset of JLI's and von_Krolock's test files instead. We'd gotten a fairly good CV rating[2] on them (between 80- and 95%) so it felt like a good

[2]CV stands for Cross Validation and the CV rating is the probability that SVM guesses right when it uses parts of the training file as test data (used when easy.py tries to find the optimal constants)

place to start. We cut out 25% of each file and put randomly each entry of the data in a new test file.

After sorting out a small error (causing a major drop i prediction accuracy) in our code we could see a fairly good result with a prediction rate approximately 10% lower than our CV value.

From this point we concentrated on tweaking our variables in RUU for best results. As mentioned before we had an idea that maybe using bigrams instead of unigrams would give a better result. This wasn't the case at all and we noticed a huge performance hit as well as really poor results.

We also, at one point, tried to raise the frequency of each token by a power of 2 or even 3. While not taking SVM much longer to process it gave a small decrease in prediction accuracy (minus 1- or 2%). So, as the with the case with bigrams, we soon discarded this idea.

The final modification we tried was to put a lower limit on how many times, in total a token had to be used in the text to be considered. Trying different values we decided to use 2 as the lower limit, making RUU discard all tokens that was only mentioned once in the text. This will include non-standard miss-spellings, unusual names and other rare character combinations.

Table 1 shows some of our test results. The left side of the tables is values we've decided in RUU (T-limit being the lower limit of tokens mentioned in the previous paragraph) and the right hand side is the values easy.py calculated for us; C and g being the constants used by the RBF kernel function and CV being the cross validation accuracy. The last column is the final result of applying the train model on the test file. Correct is the number of correctly predictions and Tot being the total number of predictions made (or total number of rows in the test file). However, when supplied with a test file that had entries from others than our test subjects SVM allways predict each row as one of the known subjects. In these cases the number of valid entries (entries written by any of our test subjects used in the test) is written in parenthesis.

4 Conclusion

One of the most important things that came up during the project was that it wouldn't be possible to use this implementation in a real-time system, but that was never our intention with RUU. To have this in a real-time system we must have a model created before and a direct link between RUU and the SVM so that we don't need to create any files or run an external program in any part of the process. Another thing is that it takes a lot of CPU and time to create the model and when finished the model is quite large so you don't want to create a new model to often.

Finally; our test results show us that the more people that are involved in a disscusion the more our final results will suffer. Trying to separate and keeping track of hundreds of people at once probably will prove to be impossible simply using SVM.

Acknowledgements

We would like to thank Richard Johansson for all the support and help with the project. Special thanks to Chih-Jen Lin for fast response and invaluable help and advice.

References

[1] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[2] dvdforum.nu. www.dvdforum.nu. A swedish discussion forum on `http://www.dvdforum.nu`.

[3] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. *A Practical Guide to Support Vector Classification*. `http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`.

[4] Pierre Nugues. *An Introduction to Language Processing with Perl and Prolog*. Springer, 2005.

Information extraction for classified advertisements

A project in the course
EDA171/DAT171 Language Processing and Computational Linguistics

Johan Eriksson

Abstract

This paper presents work done in project form in the course Language Processing and Computational Linguistics given at Lund School of Technology during the fall of 2005. The goal of the project has been to implement a simple information extraction tool from the domain of classified advertisments about apartments.

Introduction

The desire to find things on the web has lead to the development of search engines like Altavista, yahoo! and many more. The basic idea of these is to allow the user to find pages that contain certain words and do not attempt to understand or make sense of any web pages. Whenever they do take steps towards understanding web pages they risk losing their generality. Google, for instance, did not support stemming in the beginning but does so now and it is described here: http://www.google.com/help/basics.html#stemming . Searching for ”google stemming” currently, 2006-01-11, gives 716,000 results and browsing through the results one can see that there are a lot of discussions surrounding the peculiarities of Google stemming.

Another category of search engines is the shopping agents who index web shops. These are specialized to index the semistructured data of web shops where product data and prices usually are displayed in table-like structures. A technology that can be used for such applications is described in ”A Scalable Comparison-Shopping Agent for the World-Wide Web”[3]
Examples of sites using this technology include Froogle(http://froogle.google.com) and Pricerunner (http://pricerunner.com).

A third category, which index text that is unstructured, is what this paper is about. An example of this is http://eniro.jobsafari.se which indexes job ads. I have chosen to work with apartment ads from the site www.blocket.se which is a Swedish categorized ad site. I have implemented a simple information extraction engine using hand crafted patterns.

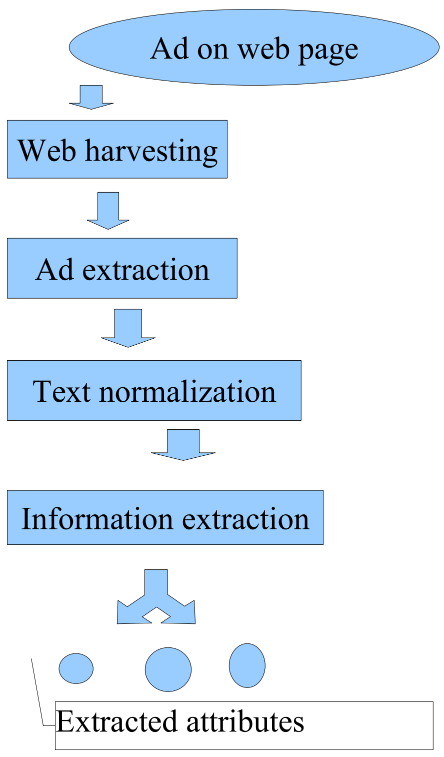
Implementation

I chose to split my system into three different logic parts: the robot that gathers ads from the site blocket.se, the web interface where a user can make queries and the information extraction engine.

From ad to information

Figure 1 shows how the information from an ad in a web page becomes extracted. The differrent steps are then described in detail.

Figure 1: From ad to information



Web harvesting
An ad starts out being a web page that is gathered by the robot.

Ad extraction
The actual ad is extracted from the markup of the web page.

Text normalization
Ad text is split into chunks that are well fitted for running patterns against. I choose to split the text at the sentence level.

Information extraction
Regular expressions are run against the sentences in an ad to retreive the desired information.

Nature of ads

Sometimes the data of interest have what I would like to call labels while other times context is used to tell the reader about the data. To exemplify we can look at apartment rental ads. My translations of the examples appear in paranthesis after each sample and might not be exact, but they suffice for the point I am trying to make.

For "Hyran ligger på 3207" (The rent is 3207) or "pris 4136 Kr/mån" (Price 4136 crowns/month), the labels are "Hyra" and "pris" respectively.
In other cases the data is described by context like
"2100:-/månad (förstahandskontrakt) inflytt 1/10" (2100 crowns/month (first hand lease) available 1/10) where the fact that it is a price/month and the mentioning of type of lease tells us that it is about rent. Another example is the very short: "900kr/mån" (900 crowns/month) which really only consists of a measure and a unit. The available information, out of context, only tells us that it is a price, but in the context of an ad for an apartment an unlabeled price is likely to be the price of the apartment.

In my observations it seams like if a price refers to something other than what one would expect for a typical item of this kind it usually carries a label: "Jag kommer börja arbeta i Köpenhamn och kommer ha en månadslön på ca 25 000 danska" (I will start working in Copenhagen and I´ll have a monthly salary of about 25000 danish[crowns]), which carries the label “månadslön” or “Hyra 4828:- Deposition 7500:-” (rent 4828 crowns deposit 7500 crowns) which uses the labels ”hyra” ”deposition”.

It seems like for different kinds of ads, or maybe texts to be more general, there exist some kind of defaults which tell us what unlabled data is about. If you look at ads for bikes, it seems like the only time you would find any mentioning of the number of wheels is when it is different from the default 2-wheel. It would be interresting to examine to what extent an information extraction system needs to have knowledge about such defaults and what influence cultural differences have on this matter, but there was no time to dwell deeper into this area.

regular expressions sub language

In an attempt to maintain some order in the chaos that emerged from testing a lot of regular expressions and having cleanup or transformation code that should be run after a successful match I experimented with creating a new regular expression definition language. The main features are that such a regular expression can

- inherit from another regular expression
- define cleanup/transformation
- add things that must match

Example. Price pattern:

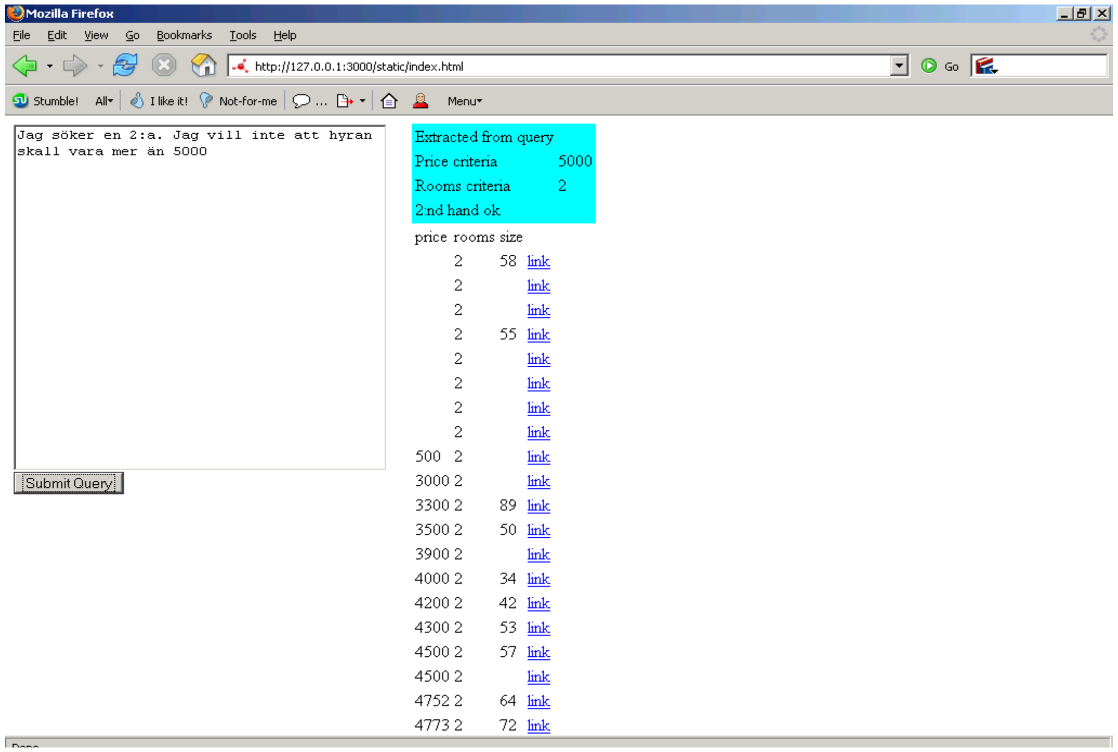
```
# base pattern 'number_free'
'number_free' => {
  'pattern' => q{\b(\d+[\d., ]*)},
  postprocessing' => [q{=~s/(\s|\.)+//g}]
}



# pattern 'rent' which inherits from
'number_free'
'rent' => {
  'ISA' => 'number_free',
  'preceeded_by'=>q{hyra.*?}
}

# pattern 'price' which also inherits
from 'number_free'
'price' => {
 'ISA' => 'number_free',
 'followed_by'=>q{ ?(kr\b|:-|\/[md])}
}
```

First I define the pattern 'number_free' which matches a wide range of numbers including “2 000” and “2.000”. I also define a some post processing which removes space or dot characters turning both “2 000” and “2.000” into ”2000”.

Then I create two sub patterns of 'number_free'. The first sub pattern is able to match the kind of prices that are preceeded by a label, as mentioned in 'Nature of ads'. It is called 'rent' and is a 'number_free' preceeded by the label 'hyra'. The second pattern can match a price. It is called 'price' and also inherits from 'number_free' and narrows the matching by saying that 'price' is a 'number_free' followed by a (Swedish) money unit.

Web interface

The web interface consists of a text box in


Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://127.0.0.1:3000/static/index.html
Go
Stumble! All I like it! Not-for-me Menu
Submit Query

```
Jag söker en 2:a. Jag vill inte att hyran
skall vara mer än 5000
```

Extracted from query	
Price criteria	5000
Rooms criteria	2
2:nd hand ok	

price	rooms	size	
	2	58	link
	2		link
	2		link
	2	55	link
	2		link
	2		link
	2		link
	2		link
500	2		link
3000	2		link
3300	2	89	link
3500	2	50	link
3900	2		link
4000	2	34	link
4200	2	42	link
4300	2	53	link
4500	2	57	link
4500	2		link
4752	2	64	link
4773	2	72	link

Figure 2 The web interface

the left side where the user can enter a query and then press 'Submit Query' after which the results will be displayed in the right part of the browser. It is shown in figure 2. The results page starts with displaying the values that have been extracted from the query and continues with displaying information from the 'For rent' ads that match the query. The web application was created using the Catalyst[4] web framework.

Evaluation

Scoring

I have used a simple way of scoring, namely for each attribute to be extracted:
+1 for correct value
0 for incorrect value
The learning set has not been included in these test runs.

For rent ads

The following table show the results for the 'for rent' ads.

attribute	**correct/ total**	**percentage correct**
price	114/133	85%
number of rooms	91/133	68%
size	113/133	85%

Wanted ads

The following table show the results for the 'wanted' ads

attribute	correct/ total	percentage correct
price	109/115	95%
wanted rooms range	71/115	62%

”Price” here is maximum price, ”wanted rooms range” is the number of rooms the advertiser wants expressed as a range like ”2 to 4”.

Comment

The closer you desire to come to being able to extract all information correctly using this method the more domain and language knowledge you have to add to the system.
This fact makes it not feasible to use this method on a larger scale(across many different domains). When building this system I came up with the idea that it would be nice to have a system which discovers the patterns itself and that maybe something like n-grams and manual tagging could be used to find the kind of labels I mentioned before.
This might be a bit naive and would not solve the problem for data that does not carry labels, but it could be a starting point for diving into this problem.
A simple example would be "The rent is $500" and "The rent, including electricity, is $1500". Having a corpus of such sentences and the price tagged, you would probably get "the" and "rent" as candidate labels. The system could then try "the", which would probably get too many false hits since it is such a common word and "rent" which would prove to be a good candidate.

Something similar had already been done and one example I found was the AutoSlog[1] system which ”automatically builds dictionaries of extraction patterns" and "uses an annotated corpus and simple linguistic rules". Another would be the TIMES[2] system.

Conclusion

For a simple information extraction task within one or few domains, handcrafting patterns might be the right way to go. But as the task grows larger and the domains increase the need for more sophisticated tools emerges like the aforementioned AutoSlog and TIMES.

References

[1] Ellen Riloff:Automatically Constructing a Dictionary for Information Extraction Tasks(http://citeseer.ist.psu.edu/riloff93automatically.html)
[2] AMIT BAGGA, JOYCE CHAI and ALAN BIERMANN: Extracting Information from Text (http://citeseer.ist.psu.edu/588088.html)
[3] Robert B. Doorenbos, Oren Etzioni, Daniel S. Weld: A Scalable Comparison-Shopping Agent for the World-Wide Web (http://citeseer.ist.psu.edu/doorenbos97scalable.html)
[4]Catalyst web framework http://catalyst.perl.org/

Information extraction for classified advertisements

David FAURE
david.faure.401@student.lu.se

Claire MORLON
claire.morlon.612@student.lu.se

1. Abstract

This report describes our work on the project part of the course Language Processing and Computational Linguistics. It presents a java written program that extracts six important pieces of information from French job advertisements. The inputs of the system are advertisements taken from the internet and converted as text files.
The results presented in this paper show that the extraction mechanism is reliable and robust.

2. Introduction

Everyone entering the job market knows how time consuming the search for a job is. The main contribution of this waste of time is the time spent while reading the advertisements in order to see if the proposed job can fit with one's competences and requirements. From this observation, we guess how useful a program that extracts automatically the interesting information from these job advertisements could be.

This paper will present a java written program whose role is to extract six pieces of information from some internship advertisements found on the Internet. The six characteristics of interest for the internships are: its subject (`subject`), its duration (`duration`), the required study level (`studylevel`), the company (`firm`) and the place (`city`) where it takes place and finally the date when it starts (`beginning`).

This paper is organized as follows: After this short introduction, section 3 presents the source texts that can be used with the program, section 4 deals with the functioning of the written java program. The results obtained with this program as well as its evaluation are presented in section 5.

3. Source texts

The program was developed to extract information from French advertisements. The advertisements used throughout the project were taken from the internet: at first, we took them on some companies' web sites. As some information were not explicit in this case (the name of the company was for example supposed to be known), we finally chose to work with "general" web sites for job advertisements. The advertisements, chosen such that they had at least two pieces of interesting information, where then converted into text files (.txt) in order to get rid of all web formats (tables, headers …). These files were finally used as inputs for the java program.

We used 15 advertisements as a working set during the development of the program. We then applied the final version on 14 other unread texts in order to evaluate the program more objectively, and to measure how independent from the texts the results were. All these 29 texts were hand marked to make easier the comparison between the theoretical words of interest and those found by the program.

4. The information extraction program

1. Mechanism

The information extraction mechanism in our program works in two steps:

- **Division of the text into blocks**

  The first task consists in dividing the text into smaller blocks corresponding to the different pieces of information (subject, beginning…), to make the search of useful information easier and quicker.

  In order to do that, we first look for keywords in the text, which will delimit the different blocks. The program runs through a list of keywords, and performs pattern matching for each of them.

  For instance, let's consider a text composed by the sentence:

Nous recherchons deux futurs ingénieurs pour une durée de <duration>6 mois</duration> sur notre site de production dans le centre ville de <city>Quimper</city>.

Three key words are found in this text : *durée* (duration), *site* (site) and *ville* (city). The first one will thus introduce a block where there is a high probability to find an information of duration. The two other ones are both related to the `city` information; they will thus delimit two blocks corresponding to `city.` The blocks obtained for this example are finally:

1. *durée de 6 mois sur notre*
2. *site de production dans le centre*
3. *ville de Quimper.*

Since several keywords can be found for the same piece of information (such as `city` in the example), all the blocks corresponding to a same information are stored into a table. Finally, we store all these tables into a hashtable whose keys are the different kind of information that are looked for. The final hashtable corresponding to the example is represented in Table 1.

information	blocks
`duration`	*durée de 6 mois sur notre*
`city`	*site de production dans le centre*
	ville de Quimper

Table 1: Hashtable listing the different blocks found in the example and the pieces of information they are related to

The idea behind this system is to try to isolate the relevant information to avoid searching the whole text. But it is also a good way to chose the better solutions in a list of several propositions, which leads to a better accuracy. For example, if several names of cities are present in the whole text, a global search would normally detect all of them. But then, how to choose the one where the internship really takes place? With our solution, only the names detected in the blocks related to `city` will be kept, as it's very likely that the relevant city is mentioned in those blocks.

- **Pattern matching**

Secondly, the useful information has to be detected in the relevant blocks. This is done with pattern matching. The first step of this stage consists in removing the keywords at the very beginning of the blocks. Then, for each piece of information, we look for patterns inside the corresponding block(s). If one pattern matches, we keep it and store it in a hashtable. The next section presents some of the patterns used in the program. If no pattern is found in any of the blocks, the same pattern matching is applied in the whole text, in case the block division would be inappropriate. It's however good to

keep in mind that the patterns found in the blocks give in general a better accuracy than the ones found after a search in the whole text.
Since several patterns can be found for one piece of information, only the first matched pattern is stored in the hashtable. Indeed, the useful and relevant information are often present at the beginning of the advertisement.

The resulting hashtable corresponding to the example is represented in Table 2.

Information	**Result**
`duration`	*6 mois*
`city`	*Quimper*

Table 2: Hashtable listing the final results for the example

2. Some patterns used

Here are some examples of the patterns we used to find the relevant information in the blocks or in the text.

- In order to find the location of the internship, we used an alphabetical list of all the French cities. The program runs through this list and for each city, looks in the block or in the text if it can find it.
- To find the `beginning` information, one of the pattern we used was "*month 200[0-9]*" where *month* was one of the twelve months of the year.
- For the `studylevel` information, we took advantage of the French system of qualification which has the type {*bac +number between 0 and 7*} (Ex.: *bac* +5). We thus used as pattern : *"bac.{1,5}[0-9]"*
- We noticed that the `subject` information corresponds often to a complete sentence introduced by a `subject` keyword (*"sujet : "*, *"intitulé : "*, …). Therefore, we took the first sentence of the `subject` block if there is such a block, or the first sentence of the whole text in the opposite case.
- To find the `duration` information, we used the pattern *"[0-9].mois"* (where "mois" means month in French).
- To find the name of the company, we took the first line of the firm block, the name included in the email address (pattern: "*@(.*?)*\\."), or in the web address (pattern: "*(www*\\.*)(.+)(*\\.*com*|\\.*fr)*").

5. Results

In order to have a quantitative evaluation of the performance of our program, we computed two parameters (precision and recall) for each of the six desired pieces of information.
The precision gives an evaluation of the correctness of the proposed instances; it is defined by:

$$\text{Precision} = \frac{\text{nbr of instances proposed correctly}}{\text{nbr of proposed instances}}$$

The recall measures how well the program finds what was expected; it is defined by:

$$\text{Recall} = \frac{\text{nbr of instances proposed correctly}}{\text{nbr of instances possible to find}}$$

1. Results for the working set of texts

The precision and recall average values for each piece of information obtained with the 15 texts of the working set are presented on Fig. 1 and Fig. 2.
We can notice that the `city` and the `firm` information get good results (precisions over 80% and largest recalls). For the firm information, this can be explained by the fact that the name of the company is often well isolated in the advertisement and introduced by regular keywords; thus our system of blocks works quite well for this

information. For the city information, this shows that the use of the city list is very efficient.

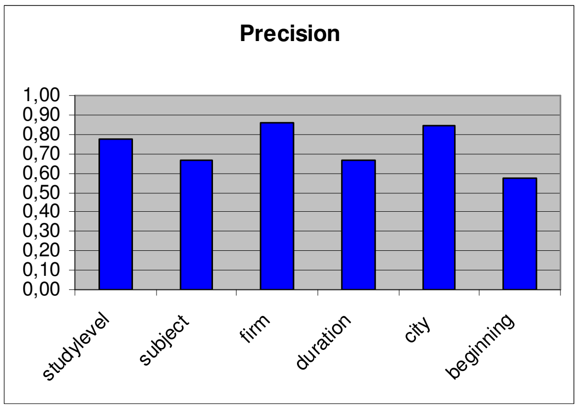


Fig. 1: Precisions obtained with the 15 working texts for the 6 parameters of interest

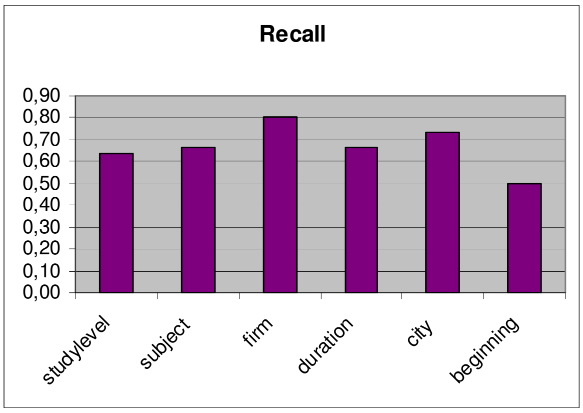


Fig. 2: Recalls obtained with the 15 working texts for the 6 parameters of interest

The other parameters get less good results mainly because of the difficulty to isolate and delimit very precisely the correct answers. Indeed, when we have a look to the proposed instances, we notice that the program sometimes proposes either only a part of the correct answer or a bit more than it.
Fig. 3 and Fig. 4 represent two lightly modified versions of the precision in order to quantify more precisely the amount of "inexact" answers.
In the case of precision 1, an answer is considered as correct if the instance proposed by the program contains the theoretical value. We see on the graph that only the subject information is concerned by this case. Indeed, it's easy to find the beginning of the `subject` (with keywords) but difficult to know where it ends. Consequently, the program sometimes takes more words that needed.
At the opposite, precision 2 is obtained by considering an answer as correct if the instance proposed by the program is included in the theoretical value. In this case, the information `studylevel`, `duration` and `beginning` are concerned, especially because the pattern matching system is too rigid.

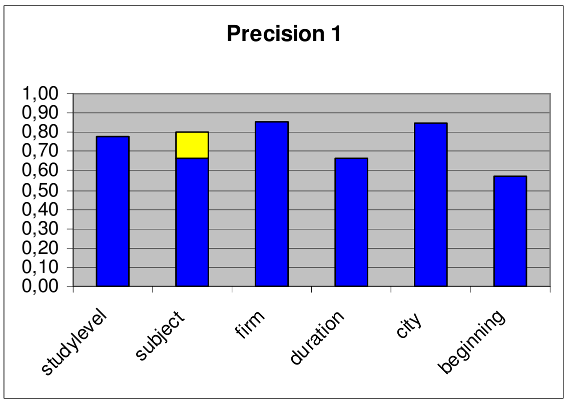


Fig. 3: Precisions obtained if "unprecise" solutions are accepted

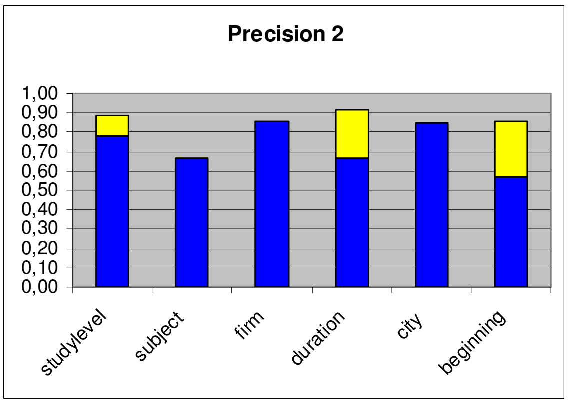


Fig. 4: Precisions obtained if partial solutions are accepted

For instance, for the `beginning` information, the program finds "janvier 2006" when the expected answer was "fin janvier 2006". Likewise, the study levels Bac +4/+5 are partially detected (we only obtain Bac +4). However, a more precise pattern matching system which could have improved the results and detected those kind of missed words would have be too "hard wired". It was indeed very difficult to find general rules that could have taken into account those exceptional cases.

2. Results with new texts

When applied on unread texts, the program gives the results on Fig.5 and Fig. 6. The results obtained with the new texts are plotted in green on the graphs, while those considered previously are still in blue.

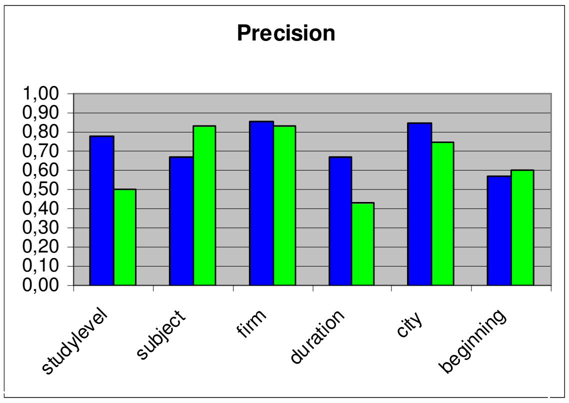


Fig. 5: Precisions obtained with the working texts (blue) and the unread texts (green) for the 6 parameters of interest

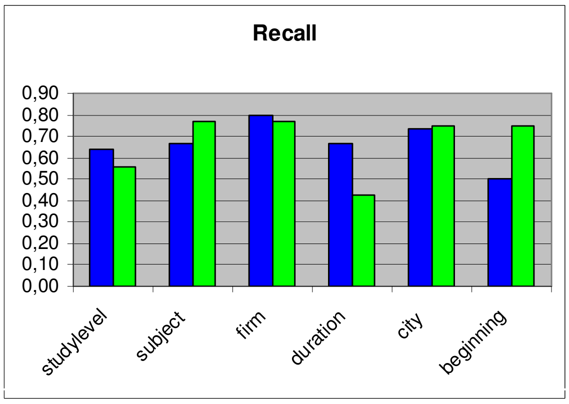


Fig. 6: Recalls obtained with the working texts (blue) and the unread texts (green) for the 6 parameters of interest

We can notice that the results are very similar, at least on average: some pieces of information obtain larger precision and recall with the working texts, whereas others have better results with the new texts. This shows that the results are quite independent on the texts, and this can prove than our program is robust enough to be applied on unknown texts.

The small differences in the results for the two sets of texts are probably due to the fact that we used in fact a small amount of texts, for both development and evaluation. The results would consequently be much more reliable if a larger amount of advertisements was used.

6. Conclusion

According to the results, the program that we implemented is reliable enough to get a good overview of a French job advertisement. However, all the useful pieces of information are not always found, especially because each advertisement has its own format. It's therefore very difficult to establish general rules for pattern matching. Some improvements, such as taking advantage of the HTML format (exploiting the tables to delimit easier the blocks, using headers to find the most important information such as subject and company,…), could have been done if we had more time.

References

Hugo Etiévant 2004. *Expressions régulières en Java avec l'API Regex* http://cyberzoide.developpez.com

Informationsextraherare – Ölrecensioner

Hugo Forss och Henning Norén

1 Introduktion

1.1 Prolog

När det här projektet påbörjades så var vårt syftet att skriva en informationsextraherare. Att det blev för just ölrecensioner beror främst på två aspekter. Dels var det ett personligt intresse - det var en chans att lära sig mer om ölkultur och att få utveckla ett verktyg som faktiskt kunde vara användbart. Dels fanns det en begränsat mängd information att hämta och en begränsad vokabulär (en sanning med viss modifikation).

Det första problem vi ställdes inför var att finna en korpus som var stor nog. Vad vi fann var en sida vid namn Sidan RateBeer.com. Den kan i princip ses som en gigantisk databas med ölrecensioner, skrivna av sidans medlemmar, sorterade under respektive öl. Att det därmed redan framgår vilket öl det är frågan om gjorde att vi fick mindre information att försöka plocka fram. Å andra sidan gav det oss material nog att prova en annan idé - att slå samman informationen från de olika recensionerna och få fram den allmänna uppfattningen om varje öl (och i slutänden få fram något liknande posterna i Systembolagets kataloger).

Vidare gav sidan mycket nyttig information i form av artiklar om ölprovning för nybörjare samt en nybörjarvokabulär som för oss utgjorde en ypperlig startpunkt. Vad vi genast noterade var att vokabulären var uppdelad enligt 4 olika kategorier. Då många av recensionerna snyggt och prydligt tog upp kategorierna en efter en föll det sig naturligt att försöka fånga in hela längre fragment innehållande ord tillhörande samma kategori.

1.2 Öltermer

Ett öl bedöms efter fyra egenskaper: utseende, arom, smak och gomkänsla. Utseende handlar om färgen på vätska och skum, huruvida vätskan är klar eller grumlig och om skummet är tjockt och långlivat eller inte. De tre smakerna är sött, surt eller beskt (öl brukar i regel inte vara salt). Arom är det vi tar upp med näsan och brukar allt som oftast kallas för smak det också. Här handlar det i första hand om att fri association och därför finns det också en närmast obegränsad mängd ord inom denna kategori. Gomkänsla är slutligen huruvida ölet upplevs som torrt, oljigt, stickigt med mera.

2 Informationsextrahering

2.1 Förberedelser

Ett Perl-script loggar in på sidan RateBeer.com och hämtar samtliga recensioner för ett givet öl baserat på dess id-nummer. Därefter snyggar ett annat script recensionerna och utför inledande taggning.

2.2 Regler

Parsning och taggning sköts av SloppyTagger. Redan i ett tidigt stadium bestämde vi oss för att inte använda ordklasstaggning. Istället arbetar taggern i flera pass där den matchar och taggar allt mer komplexare strukturer. Reglerna är skrivna med en regex-inspirerad syntax som låter oss matcha både klartext och xml-taggar samt att infoga nya taggar på valfri plats i de matchade sekvenserna.

2.3 Ord

I det första passet taggas de ord som vi har definierat i vår ordlista. Dessa kan delas in i fyra kategorier:

- Kategoriord är ett ord eller mönster som anger kontext.
- Nyckelord är ord som bär på information. De flesta nyckelord kan klassificeras direkt i ordlistan, men några - speciellt färg - blir helt beroende av kontext.
- Modifierare är ord som (förstärker/försvagar/inverterar) ett nyckelords betydelse. De kan också användas för att hitta okända nyckelord. Modifierare är egentligen väldigt svåra då det för det mesta inte går att gissa recensentens intention. Eftersom vi inte vet om en modifierare syftar på ett eller flera ord så delar vi helt enkelt in dem i två kategorier, de som står före och de som står efter sitt nyckelord.
- Länkord är alla typer av bindeord. De hjälper ytterligare vid kontextbestämning.

En lockande tanke vore att helt slopa ordlistan så när som på kategoriorden och därmed låta programmet själv identifiera nyckelord. Men så som informationsextraheraren ursprungligen var tänkt att fungera behövde vi kunna ange betydelser för flertalet ord (som modifierarnas vikter).

I vissa fall finns en så pass liten och väl vedertagen vokabulär att vi bedömt det onödigt att leta efter ytterligare uttryck. Ett annat skäl kan också vara att det råder en tydlig relation mellan de olika orden. Färger är ett tydligt exempel och modifierare än mer så. I vår ursprungliga design placeras dessa på en slags skala och i slutänden är det tänkt att informationsextraheraren ska presentera ett slags genomsnitt av dessa värden.

2.4 Enheter

Nyckelord och modifierare bildar tillsammans enheter där nyckelordet anger grundbetydelse och modifieraren ordets vikt. Enheterna förenklar på många sätt kommande steg. Dels ger det oss möjligheten att placera nyckelord tillhörande mer specifika kategorier i enheter med mer generella namn. Med detta uppnår vi att vi på ett enkelt sätt kan matcha ord från olika kategorier med en gemensam regel.

Med enheternas hjälp kan vi också tagga kringliggande ord. Många gånger är recensionerna byggda som uppräkningar av olika egenskaper och därför är det naturligt att misstänka att intilliggande ord också kan vara nyckelord.

2.5 Fragment

Fragment är ytterligare ett sätt att förstärka ordens kontext. Här taggas sammanhängande sjok av redan taggade ord som ofta inleds eller avslutas med ett kategoriord. Dessutom hänger vissa kategorier naturligt samman och återfinns ofta tillsammans åtskilda av ett länkord. Andra separeras mer naturligt av meningsgränserna.

Dessa regler kommer inte att finna några nya nyckelord, tvärtom används de för att plocka bort enheter som inte ingår i något fragment och som därför kan ha taggats felaktigt. Fragmenten används också för att lösa vissa oklarheter. Exempelvis genom att skilja de färger som beskriver ölets kropp (själva vätskan) från de som beskriver dess skum.

2.6 Sammanslagning

Ett problem med språk som engelska är dess många böjnings- och avledningsformer. På något vis ska de olika formerna räknas samman. Enklast är att utnyttja storleken hos vår textmassa. I en speciell ordlista finns ett antal kända ändelser angivna. Orden matchas mot de olika ändelserna som isåfall plockas bort/ersätts med sin grundform. Det nya 'hypotetiska' ordet testas därefter mot textmassan för att se om det förekommer någonstans.

Av de ord som taggats och klassificerats väljs de mest frekventa ut. På så sätt kommer en stor del av alla felklassificeringar att sorteras bort.

3 Utvärdering

För att kunna bedöma hur pass väl informationsextraheraren fungerar så har vi för hand plockat ut information ur ett flertal öl. Två jämförelser görs: dels övergripande proportionerna falska positiver och negativer, dels hur många av de topprankade nyckelorden den lyckas pricka rätt.

Ovanstående tabell visar ration för antalet falska negativer (sådana vi inte taggat men som vi borde) samt falska positiver (sådana vi felaktigt taggat) för testet mot våran testmängd. Fler falska negativer än positiver tyder på att våra regler är alltför strikta och borde skrivas mer generella. Kategorierna palate, head och body lider av att det är rätt få recensenter som tar upp dem i sina recensioner. De kan också ses som mer abstrakta än de andra kategorierna. Vad färg, smak och arom beträffar så brukar recensenterna vara betydligt mer ense.

Kategori	Falska Negativa	Falska Positiva
AROMA	0.602	0.061
FLAVOUR	0.767	0.029
PALATE	0.388	0.511
HEAD	1.289	0.130
BODY	1.308	0.111
HEADCOLOUR	0.256	0.400
BODYCOLOUR	0.366	0.516
TOTAL	0.611	0.157

Som man ser så är precisionen inte särskilt upplyftande om man inte plockar fram majoritetsvärdena.

Ovanstående tabell visar på precisionen när vi enbart tar med majoritetsvärdena för kategorierna. Varje kategori har med sitt majoritetselement utom aroma som har med sina 3 största majoritetselement. Ökar man antalet majoritetselement till fyra får vi 92% precision samt 87% för 5. Samma mönster som syntes ovan kommer tydligare fram nu när vi bara har majoritetselementen. Head samt Palate har vi vissa problem med och Body som både är ganska ovanligt i beskrivningarna samt saknar tydliga mönster för beskrivningen har vi väldigt dåligt resultat på.

4 Slutsats

Vi har inte nått fram till vårat slutmål, att producera kort läsbar sammanfattning av ölet men från våra resultat så bör det inte vara några större problem. Det som t ex systembolaget brukar ha med är tre-fyra aromer, färg samt flavour. Dessa visar ovanstående att vi med god sannolikhet kan plocka ut ur en tillräckligt stor mängd recensioner. Det finns dock mer att göra. Vikter för modifierare och färg bör kunna tas in för att ge mer nyans och precision i beskrivningen. Förbättringar av matchning av Body, Head samt Palate bör kunna göras. Vår bristfälliga suffix-hantering borde bytas ut mot en riktigt transducer, anpassad för den typ av något kaotiska data som vi jobbar med, vilket antagligen ökat precisionen ännu mer. Vi anser att med viss anpassning så skulle vårat system kunna köras mot en levande recensionsdatabas för öl och automatiskt plocka fram majoritetsuppfattningen av de mer populära kategorierna med tillräckligt stor precision för att vara intressant.

A Användarhandledning

För att systemet skall fungera måste man skapa två underbibliotek från där man har programmen;
`reviews` - här hamnar alla recensioner man tankar hem, döpta efter ölet. Töm detta bibliotek mellan körningarna eller tag och använd en backup mellan momenten om du vill köra flera gånger då alla programmen utom `comparer.pl` och `stats.pl` är destruktiva och förändrar innehållet i detta bibliotek
`creviews` - här skall alla handtaggade data ligga, döpta efter ölet med `WORKED` i slutet av namnet.

A.1 autologin

`autologin.pl` - autologin är en automatiserad recensionsinsammlare för vår databas (ratebeer.com). Programmet tar ett eller flera argument i form av idn för ölet man vill hämta ner. Man kan även begränsa så att programmet bara hämtar ner den första sidan genom att som första argument använda `-f`. Autologin använder ett hårdkodat användarnamn och lösenord till ratebeer.com men man kan enkelt ändra det genom att ändra variablerna `$username` samt `$password` på rad 12 resp. 13.

A.2 formatreviews

`formatreviews.pl` - formatreviews formaterar, rensar och bygger xml-träd av den råa datan som autologin har tankat hem. Programmet tar inga argument.

A.3 sloppytagger

`sloppytagger.pl` - sloppytagger är själva taggaren som tar som argument filnamnet för en regelsamling och tillämpar den på recensionerna. Generellt kör man detta program i flera pass med

Kategori	Top X element	Andel rätt
AROMA	3	100%
FLAVOUR	1	100%
PALATE	1	50%
HEAD	1	50%
BODY	1	33%
HEADCOLOUR	1	100%
BODYCOLOUR	1	100%

olika regelsamlingar för att på så sätt fånga mer och mer komplexa mönster.

A.4 collector

`collector.pl` - collector samlar ihop data från de taggade recensionerna och stoppar in dem i de olika kategorierna. Formatet är detsamma som de handtaggade recensionerna är i så detta är slutprodukten av själva insamlingen av information. Collector tar inga argument.

A.5 comparer

`comparer.pl` - comparer jämför slutprodukterna från reviews med de handtaggade filerna. Programmet kräver att underbiblioteket `reviews` innehåller alla de recensioner som finns i `creviews`. Programmet tar inga argument och skriver ut resultatet på skärmen, ölvis och kategorivis och sedan ett slutresultat som avslutas med det kryptiska TOP TOTAL. Nästa rad skriver ut hur många ölsorter/filer som behandlats och sedan 7 heltal som representerar de olika kategorierna. Värdena för hur många majoritetselement som skall vara med är hårdkodat till 3 för aroma och 1 för resten och ordningen på dem är: aroma, flavour, palate, head, body, headcolour, bodycolour. Varje heltal representerar hur många majoritetselement som stämmer med de handtaggade och för 100% så skall alltså `antal filer` $\times$ `antal majoritetselement` vara lika med värdet som skrivs ut. En nolla innebär att inga matchade. Progrmmet är inte destruktivt så man kan köra om utan att skydda `reviews` eller `creviews`.

A.6 stats

`stats.pl` - stats skriver ut en sammanfattning för vilka ord som klassats i vilken kategori för varje öl som finns taggat. Programmet tar inga argument och och är inte destruktivt så det kan upprepas utan att man behöver skydda `reviews`

A.7 Exempelkörning 1

Vi plockar hem ett sex stycken ölsorter och kör dem genom hela systemet för att avsluta med att skriva ut en sammanfattning om dem.;

```
  $>./autologin.pl 12492 32111 36624 51539 6115 8484 &&
./formatreviews.pl && ./sloppytagger.pl ordlista  &&
./sloppytagger.pl enheter && ./sloppytagger.pl kandidater &&
./sloppytagger.pl fragment && ./collector.pl && ./stats.pl

<---------- Valley_Brew_Uberhoppy_IPAWORKED ---------->

AROMA: hops(10) malty(7) alcohol(5) caramel(4) toast(3) hoppy(2) resin(2)
woody(1) ginger(1) yeasty(1) lemon(1) grass(1) chocolate(1) floral(1)
citrus(1) fruity(1)
FLAVOUR: sweet(5) bitterness(2) bitter(2) sweetness(1)
PALATE: bodied(3) carbonation(3) thin(1)
HEADCOLOUR: tan(2) white(1) yellow(1)
...
$>_
```

A.8 Exempelkörning 2

Efter att vi kört exempel 1 så vill vi jämföra resultatet mot de handtaggade filer vi har, som råkar stämma precis med de ölsorter vi har tankat hem och jobbat med;

```
$>./compare.pl

<---------- Valley_Brew_Uberhoppy_IPAWORKED ---------->

<--- AROMA --->
MATCH: 30       malt(6) alcohol(4) hops(4) toasty(3) hoppy(2) resin(2)
woody(1) yeasty(1) ginger(1) caramelly(1) grass(1) chocolate(1) floral(1)
citrus(1) fruity(1)
F_NEG: 55       hops(6) hop(4) caramel(4) grapefruit(3) caramelly(3) pine(3)
citrusy(3) malts(2) bready(1) piney(1) milk(1) spice(1) mango(1) tropical(1)
resiniousness(1) amarillo(1) orange(1) cookie(1) gingersnap(1) malt(1)
alcohol(1) peppery(1) grassy(1) flesh(1) blackberryish(1) grassiness(1)
maltiness(1) grapefruity(1) spiciness(1) juiciness(1) berryish(1) cascade(1)
roastiness(1) papaya(1) rye(1)
F_POS: 4        resin(2) lemon(1) citrus(1)
false negative rate: 1.618, false positive rate: 0.047
<--- FLAVOUR --->
MATCH: 7        sweet(3) bitter(2) sweetness(1) bitterness(1)
F_NEG: 3        sweet(2) bitterness(1)
F_POS: 1        bitter(1)
...
<--- BODY --->
false negative rate: 1.308, false positive rate: 0.111
<--- BODYCOLOUR --->
false negative rate: 0.366, false positive rate: 0.516
<--- TOTAL --->
false negative rate: 0.611, false positive rate: 0.157
<--- TOP TOTAL --->
of 6 files:  18 6 3 3 2 6 6
$>_
```

Dependency Parsing

Johan Hellström and Ian Kumlien

January 23, 2006

Abstract

This paper presents an implementation of a deterministic parsing algorithm for dependency grammar in Swedish Natural Language. The pursued implementation is based upon the Java programming language and not Perl or Prolog commonly used in this field of research. This project constitutes a part of the undergraduate course *Language Processing and Computational Linguistics* by the department of Computer Science, Lund University.

1 Introduction

An essential part of processing natural language is to properly understand and determine the hierarchy of dependence. This means, given an arbitrary sentence, to find the main word and how all the other words come to depend upon this one word. The main word by convention or almost without exception turns out to be the main verb or the most central proper noun, and the dependences to this word naturally can be expressed as a tree structure. In this structure the main word is elevated to the root of the tree allowing for two branches, left and right, designed to express the dependencies of the words found on either side, still maintaining the original order of words, creating sub-trees for dependencies.

2 The Model of Our Parser

The art of dependency parsing of Swedish texts has been expertly explored by Joakim Nivre [2], and it was his work on parsing which inspired us to implement a similar parser designed in Java, not Perl or Prolog which most often is selected for natural text parsers today.

2.1 Nivre's Principal Parser

The principles of this kind of parser was developed by Joakim Nivre[1], and has many similarites to the basic shift-reduce algorithm for context-free grammars[2] The principal parser make use of a rule set composed of suggested word class pairs ordered in expected frequency and four methods; Right-arc (RA), Left-arc (LA), Shift and Reduce as presented in table 1.

[1]principles and outline by Nivre [1].
[2]extensively defined by Aho et al [4].

Initialization	$\langle \mathbf{nil}, W, \emptyset \rangle$	
Termination	$\langle S, \mathbf{nil}, A \rangle$	
Left-Arc	$\langle n\|S, n'\|I, A \rangle \rightarrow \langle S, n'\|I, A \cup \{(n', n)\} \rangle$	$\texttt{LEX}(n) \leftarrow \texttt{LEX}(n') \in R$ $\neg \exists n''(n'', n) \in A$
Right-Arc	$\langle n\|S, n'\|I, A \rangle \rightarrow \langle n'\|n\|S, I, A \cup \{(n', n)\} \rangle$	$\texttt{LEX}(n) \rightarrow \texttt{LEX}(n') \in R$ $\neg \exists n''(n'', n') \in A$
Reduce	$\langle n\|S, I, A \rangle \rightarrow \langle S, I, A \rangle$	$\exists n'(n', n) \in A$
Shift	$\langle S, n\|I, A \rangle \rightarrow \langle n\|S, I, A \rangle$	

Table 1: Formal description of Nivre's Parser [1]

3 Implementation Outline

For a matter of practicality, offering attractive possibilities of reuse and customizing, the parser application is actually composed of three semi-independent parts, intended to be executed in sequence. The source text, a fully annotated collection of Swedish natural language called "Talbanken MALT" [6], first was processed to determine the frequency of different pairs of word groups appearing in typical. The 100 most frequent, in ascending order, were selected as set of rules, considered to well enough represent typical Swedish natural language in general. Next, the parser principles of Joakim Nivre were consulted [1]. While using an unaltered syntactic approach we introduced different choices of classes, native or optimized for Java performance. Our parser made use of the annotation tagging offered in the text-source as far as word classes were concerned, but ignored the sentence structure tagging during this phase. Finally, the outcome of our parsing was compared with the previously neglected sentence structure tagging available in the source text. Statistics were created based upon word correctness and full sentence completeness.

3.1 Rule Extraction

The rules used in the parser are a list of pairs of word classes ordered by how frequently they occur. By pre-parsing the entire source of natural text for every single pair of words and keeping score of what combinations of word classes is most frequent, the most proper account of how frequent word pairs in typical Swedish language are is obtained. This list will control the order in which word class combinations are considered as well as determine the time effectiveness of parsing since the list will be consulted top-to-bottom until the specific pair is matched or else the search will fail

3.2 Parsing Dependencies

The basic implementation consists of four different operations, in order they are: Left-Arc, Right-Arc, Reduce and Shift. In our implementation we added a extra top priority shift operation that makes sure that there are elements on the stack.

First construct a string from the word classes involved, elements: the first element on the stack and current element.

Here is some documented pseudocode, this is all done in a iterative way, this the continue statements below.

```
if stack.isEmpty then:
  shift

Left-Arc:
  if stack.peek.isDone is false then: /*if the element hasn't been handled*/
    if tmp in rules_left then:        /*if it's in the list of rules      */
      stack.peek.setDone_LeftArc      /*the element is a left arc         */
      stack.pop                       /*remove the element from the stack */
      continue                        /*continue the iteration            */
    else:
      if element in rules_root then:  /*if it's in the list of rules      */
        stack.peek.setDone_Root       /*the element is a root element     */
        stack.pop                     /*remove the element from the stack */
        continue                      /*continue the iteration            */

Right-Arc:
  if stack.peek.isDone is false then: /*if the element hasn't been handled*/
    if tmp in rules_right then:       /*if it's in the list of rules      */
      element.setDone_RightArc        /*the element is a right arc        */
      stack.push element              /*push the element to the stack     */
      continue                        /*continue the iteration            */
    else:
      if element in rules_root then:  /*if it's in the list of root rules */
        element.setDone_Root          /*the element is a root element     */
        stack.push element            /*push the element to the stack     */
        continue                      /*continue the iteration            */

Reduce:
  if stack.peek.isDone is true then:  /*If the element on the stack is done*/
    stack.pop                         /*then remove it                    */
    continue                          /*continue the iteration            */

Shift:
  stack.push element                  /*push current elementto the stack  */
```

Each rule continues the iteration and thus terminates any additional parsing done.

3.3 Correctness Evaluation

Since the source text is fully annotated, even when sentence structure is considered, the process of determining the correctness of the parsing is simply a matter of processing the source text yet another time, comparing the parser's findings with the tags of annotation. Since the annotation is available word-by-word, the correctness statistics can be calculated by word as well as by sentence

4 Results and Conclusions

By parsing the corpora, keeping score of the occurrences of word class pairs, we select the top 100 of these for rule set. This is then used in the Nivre style (Left, Right, Reduce and Shift) dependency parser. When a correctness score parse finally is performed, our findings are that 70,515 words from the corpora of 103,939 words are correctly determined. This would then give an average of 67.84%. For the completion of entire sentences, in essence a 100% correctness of words per sentence, the result is the somewhat modest count of 546 out of 6,316 sentences, or 8.64%.

5 Comments and Future Improvements

We need to point out that no further guides or pre-processing steps were taken, besides the general top 100 pair-of-word-classes statistics.
Further improvements, would surely be made by recognizing subclasses of certain word classes, say for instance making the parser sensitive to specific prepositions linked to certain verbs. But the path along this thinking is long and by these steps the method is no longer as general as now is the case. Different language parsers would of course have all different subclasses and the top-100 count would be even more dependent upon the nature of the training corpora. By applying this subclass distinction, the results would improve at the cost of narrowing the range of application.
Neither did the parse allow for different context recognition. Certainly, one would not consider the often short-and-to-the-point chapter headlines to be as fluent and be composed of as colourful language as the rest as the text. By adding methods to make this distinction, one would in fact have even more valid statistics from the word class frequency parse and probably raise the score. But once more, general application would be sacrificed. The types of headline composition vary as the nature of the corpora does and also, most likely the trends of headlining is different in different languages.
In our code there is a small race condition. If Right-Arc is triggered and we get a Left-Arc right after, it can set the *setDone_right* element to be root as well, while this is intended behavior it still wrong. Any given sentence should only have one root element. This is easily fixed with some post processing or better rules, but it was not one of our goals due to time constraints.

6 Acknowledgements

We would like to thank Pierre Nugues for his patience and guidence during this project, which initally was a challenge due to some confusion about the details of the algorithm development. It was by invaluable guidence and much effort the completion of this project came to be.

References

[1] Joakim Nivre (2003), *An Efficient Algorithm for Dependency Parsing*, School of Mathematics and Systems Engineering, Växjö University, 12 p.

[2] Joakim Nivre (2005), *Inductive Dependency Parsing of Natural Language Text*, School of Mathematics and Systems Engineering, Växjö University, 209 p.

[3] Pierre Nugues (August 2005), *An Introduction to Language Processing with Perl and Prolog*, Department of Computer Science, Lund University, 544 p.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman (1986), *Compilers: Principles, Techniques and Tools*, Addison Wesley.

[5] (1997), *Stockholm Umeå Corpus*, Produced by Department of Linguistics, Umeå University and Department of Linguistics, Stockholm University.

[6] (1997), *Talbanken MALT*,
Available at Nivre's website; http://w3.msi.vxu.se/ nivre/research/maltDT.html
(Certified jan 2006)

This paper was printed using $\LaTeX\,2_\varepsilon$.

Comparing methods for Coreference solving

By Jonas Henrikson, 2006

Abstract

In 2005, Magnus Danielsson created a coreference solver for a project called Carsim. The purpose of this project was to automatically analyze news reports about traffic accidents, extract the relevant details of the events in an accident, and recreate a simulation of these events. In analyzing the news reports, an important part of the program is the ability to understand when seperate references to e.g. a car or a person involved in the accident are actually references to the same car or person; i.e. the ability to understand when coreference occurs in the text. In the coreference solving program by Magnusson, the method used to make final decisions of whether two nouns (or, more precisely, two noun phrases) are indeed coreferential was the ID3 decision tree algorithm. The purpose of my project is to generalise the code to make it possible to substitute the ID3 algorithm with other solver methods, so as to be able to compare the efficiency and accuracy of different methods.

Original program structure

Instead of changing the coreference implementation in the actual Carsim program, I used an evaluation version of the coreference solver. This evaluator selects a number of random documents from a collection of accident reports, performs the coreference calculations, and compares the results to manually identified coreferences. The final results are shown as three percentages: recall, precision and the so-called f-measure. Recall is the percentage of how many of the actual coreferences are identified by the solver, precision is the percentage of identified coreferences that actually *are* coreferences, and the f-measure is a sort of mean value calculated from the first two percentages.

The evaluator includes a lot of code for identifying noun phrases and their grammatical features, etc, but fortunately most of these processes are carried out before the final result of whether a pair of noun phrases are coreferential or not is calculated. Not that much of the code was originally dependent on the decision tree algorithm, and most of the code concerning the ID3 solver method were contained in a seperate directory: the "decisiontree" packade. My first step was to identify what segments of code outside the decisiontree package were specifically dependent on the ID3 solver.

First of all, the Evaluator class is where most of the central work is done, and this is where the ID3 object was created. Each document used in the evaluation is represented in the Evaluator by an instance of the CorefDocument class, and since this is where the information about the noun phrases are stored as private vectors, and since these vectors are the data to be "analyzed" by the decision tree algorithm, the Evaluator sent the ID3 object to each CorefDocument object which in turn used a TreeExecutor object to execute the desicion tree solver with regard to the noun phrases in the document. The TreeExecutor class was part of the decisiontree package, and this class is where the final call to the ID3 algorithm occured, to decide whether a given pair of noun phrases actually corefers or not.

So, in short, the only classes outside the decisiontree package which contained specific uses of classes inside the decisiontree package were the Evaluator and CorefDocument classes.

My changes – Step 1: Generalisation

What I did first was to make the CorefDocument independent of the decisiontree package. Instead of having a method in the CorefDocument class to which the Evaluator sent the ID3 tree, and which used the document's private noun phrase vectors and the TreeExecutor class, I added simple "get"-methods making it possible for the Evaluator to access the otherwise private noun phrases, so that the uses of decisiontree package classes could be contained in the Evaluator class. (See the appendix for specific examples of this and other changes to the code.) By doing this, I could move the use of the TreeExecutor class to the Evaluator, and hence the CorefDocument class became completely free of ties to the decisiontree package.

Next, I discoreved that a couple of the classes contained in the decisiontree package were actually not specifically dependent on any of the other decisiontree package classes, and therefore I moved them out of the package and put them among the rest of the standard evaluator classes. These were the NPFeatureTransferer and ApplyHardConstraints classes. Only one line was changed in each of these files: the first line of the file saying which package the classes were part of.

Then I looked closer at the TreeExecutor class and noticed that this class was only minimally dependent on the decision tree solver method. What was happening at this point in the changes being made was that the Evaluator created an ID3 tree and a TreeExecutor, sent the ID3 tree to the TreeExecutor, and the Treeexecutor at one (and only one) point in the code asked the ID3 object whether a pair of noun phrases coreferred or not. What I did here was to create a simple interface called CoreferenceSolver, and a simple class called TreeSolver which implements the CoreferenceSolver interface. I moved the creation of the ID3 object to the TreeSolver class, and I changed the TreeExecutor class so that instead of using the ID3 tree directly, it now used the CoreferenceSolver inferface. By doing this, the TreeExecutor class became fully independent from the decision tree solver method, and because of this I thought it to be apropriate to change the name of this class from TreeExecutor to SolverExecutor, and to move it out of the decisiontree package and put it among the other standard evaluator classes, just as with the NPFeatureTransferer and ApplyHardConstraints classes mentioned above.

Now there was only one line left in the Evaluator class that specifically used the decision tree solver method: the line where the TreeSolver object was created (which of course was represented by and used as a CoreferenceSolver object in all other places). No other files outside the decisiontree package now had any ties to the ID3 solver method, except the new and simple TreeSolver class. Generalisation complete!

My changes – Step 2: Implementing another solver method

The alternative solver method I implemented was an SVM classifier. Only two things were needed to make this work. First, a new class implementing the CoreferenceSolver interface had to be created, which I naturally called SVMSolver, and second, the one line in the Evaluator class which created a TreeSolver object was changed to instead create an SVMSolver object. The only issue here was the difference in format of the input to the two solver methods. While the ID3 solver method used two seperate vectors to build the decision tree – one with positive examples of coreferences and one with negative examples – the SVM solver method used only one vector to build the SVM classifier. This one vector had the same content as both of the positive and negative examples combined, but with a boolean value in the beginning of each example to flag it as positive or negative. Also, while the ID3 solver used standard Vector objects, the SVM solver used ArrayList objects. I decided to handle this type conversion in the constructor for the new SVMSolver class, so that nothing else needed to be changed in any other class. Once this type conversion was implemented, the SVM solver method fit perfectly into the program and could be tested and evaluated in exactly the same way as the original ID3 solver method.

Results / Evaluation / Conclusion

To compare the two methods of coreference solving, I ran two Evaluations of each method. One evaluation actually contains four iterations of the whole process, and shows a total result. The resluts from these evaluations are as follows.

	Recall	**Precision**	**F-measure**
Total results for **ID3**, 1'st and 2'nd run:	*86.1%*	*81.6%*	*83.8%*
Total results for **SVM**, 1'st run:	*86.1%*	*79.6%*	*82.7%*
Total results for **SVM**, 2'nd run:	*85.3%*	*79.8%*	*82.4%*

Interestingly, the results from the two ID3 evaluations were identical, even though the documents used are randomly selected. I have no explanation for this. As you can see, the SVM solver method can be equal to the ID3 solver method in terms of recall, although it shows bigger variation in this value. When it comes to precision, however, the ID3 solver takes the lead. This 2% drop in precision on the SVM's part might look like very slight, but this difference becomes much more striking when formulated as a 10% increase in incorrectly identified coreferences.

Philosophical comment on coreference

In the philosophy of language, coreference remains a problem in the sense that it is difficult to explicate just how coreference works and is comprehended by language users. However, it is not a central problem. The question of what coreference is is not as problematic as the question of how it works, and the latter question is of much bigger interest in computational linguistics. Up to and including my candidate course in theoretical philosophy, coreference has hardly been mentioned as a difficulty, while on the other hand you are probably already aware that coreference is a very difficult problem in computational linguistics. Apparently, there is a difference in how this issue is viewed within these two disciplines. In philosophy, it is generally thought that the comprehension of coreference is highly dependent on a wide base of knowledge about the world. In computational linguistics, by contrast, one usually tries to find a solution by focusing on grammatical features and basic semantic category information. Some mean that it is important to realize that while this approach is perfectly legitimate, a big part of the problem remains unsolved.

Appendix – Relevant code segments

It is recommended that you are somewhat familiar with the source code when reading this appendix.

Originally, the ID3 tree was created in the Evaluator class with these three lines:

```
ID3 id3 = new ID3(yesVector, noVector, "true", "false");
String generatedTree = id3.getResultTree();
DTreeNode treeInMemory = id3.getRootNode();
```

With the creation of my new CoreferenceSolver interface and TreeSolver class, these lines were removed and replaced by the following line. This line is the only thing that needs to be changed in the Evaluator class if one wants to switch solver methods.

```
CoreferenceSolver coreferenceSolver = new TreeSolver(yesVector, noVector);
```

To use the SVM solver instead, one would simply change the above line to this:

```
CoreferenceSolver coreferenceSolver = new SVMSolver(yesVector, noVector);
```

This method was removed from the CorefDocument class:

```
public void addToEvaluation(EvalMaker localEv, EvalMaker globalEv, DTreeNode treeInMemory) {
            TreeExecutor t = new TreeExecutor(nounPhraseFeatures, treeInMemory);
            Vector postiveChains = t.getPositiveChains();
            localEv.addValues(chainsAsInts, postiveChains);
            globalEv.addValues(chainsAsInts, postiveChains);
}
```

In its stead, these two methods were added:

```
public Vector getChainsAsInts() {
            return this.chainsAsInts;
}
public Vector getNounPhraseFeatures() {
            return this.nounPhraseFeatures;
}
```

Because of these changes to the CorefDocument class, this line was removed from the Evaluator class:

```
doc.addToEvaluation(localEvaluator, globalEvaluator, treeInMemory);
```

...and was substituted by the following lines. Note that these are very similar to the contents of the removed addToEvaluation method shown above.

```
Vector nounPhraseFeatures = doc.getNounPhraseFeatures();
SolverExecutor solverExec = new SolverExecutor(nounPhraseFeatures, coreferenceSolver);
Vector chainsAsInts = doc.getChainsAsInts();
Vector postiveChains = solverExec.getPositiveChains();
localEvaluator.addValues(chainsAsInts, postiveChains);
globalEvaluator.addValues(chainsAsInts, postiveChains);
```

Here is the very simple new CoreferenceSolver interface:

```
public interface CoreferenceSolver {
  public boolean corefers(Vector features);
}
```

And here is the TreeSolver class which implements it. Again, note the similarities with the first three lines mentioned in this appendix, which were removed from the Evaluator class.

```
public class TreeSolver implements CoreferenceSolver {
            private ID3 id3;
            private DTreeNode root;

            public boolean corefers(Vector features) {
                          return ExecutingTree.corefers(features, root);
            }
            public TreeSolver(Vector yesVector, Vector noVector) {
                          this.id3 = new ID3(yesVector, noVector, "true", "false");
                          this.root = id3.getRootNode();
            }
}
```

The new SVMSolver class is a bit too big to be included in this appendix, while other changes are too small to deserve mention. Please refer to the source code for further code reading pleasure.

A compiler for phonological rules

Hans Jansson
Department of Computer Science, Lund University
Box 118
221 00 Lund,
Sweden,
hans.jansson.667@student.lu.se

Abstract

This is a report of an implementation of a compiler for phonological rules. The implementation processes files written in lexc (Karttunen, 1993) annotation and produces data suitable for processing with SWI-Prolog (Wielemaker, 2005). The output is a transducer (a Mealy machine, to be more precise), a finite-state machine which not only accepts but also translates its input. Such machines can be used to perform spell checking, morphological analysis etc. The project resembles the lexicon compiler from Xerox[1].

1 Introduction

Implementation of a compiler for phonological rules is nothing new. Xerox has already a set of adequate tools for working with such rules, these are, however, not open source and the free of charge copy is limited and for non-commercial use only. Yet, they have gained popularity and thusly, their formalisms have too. Other formalisms and tools for phonological facts and rules (e.g., XML-annotated corpora) exists alongside those from Xerox, even some GPL-licensed ones such as SFST (Schmid, 2001), but many suffer from a common mistake: everyone likes standards, and thus a new such is borned. This project aims to be a prototype for a generic tool, able to generate descriptions of transducers provided phonological rules and easy to attach new front- and backends to.

2 Design

To achieve a modular, easily extendable compiler, I chose to build front- and backends around a abstract model of a transducer consisting of start node marker, nodes, labeled arcs and end node markers. Unfortunately, I have already found a design flaw in the representation of the nodes; they are objects, and as such exist uniquely in memory and are therefore already easily distinguishable, but I added a unique non-negative integer to each of them, usable only in the final output. Such details are backend-specific and should of course be implemented in the appropriate backends. The needed correction is however quite small and has no impact on the design as a whole.

3 Tools

The JastAdd (Ekman et al., 2005) package was used in the implementation, simply because I was already familiar with it. It was used on top of JavaCC (Viswanadha, 2006), a Java compiler compiler, and it added advanced features of which I only used the parts providing aspect orientation and abstract grammar. By using aspect oriented code, no Visitor pattern was needed to traverse the AST. The choice of JavaCC turned up to be sinister; when I was about to implement the lexc frontend I discovered that JavaCC simply couldn't tokenize certain parts correctly.

4 Frontend

There is only one frontend in the prototype application, for lexc source files. The lexc formalism is fairly easy to parse, there is a catch though: sometimes a string of characters should be interpreted as one token and sometimes each character alone should be considered a token. This can be solved in two ways, by means of scanner/parser co-operation, where the scanner asks the parser for the correct interpretation, or using a stateful scanner, where the interpretation depends on the current state of the scanner. The first alternative gives no reliable result within JavaCC, because the scanner may be several tokens ahead of the parser, and thus a stateful scanner was made.

There are more quirks though. A normal lexc file (see listing 1) declares multicharacter symbols before the lexicon part begins, i.e.,

[1] lexc

strings of characters intended to be interpreted together as a single token within the context where these characters normally should be considered as several one-character tokens. Since there are no characters reserved for marking the start and end of such multicharacter symbols, the only solution is to rely on the scanner's feature of choosing the longest possible match when tokenizing the character stream. All you have to do is to modify the symbol table of the scanner by adding the new symbols as soon as they are defined at the top of the source file, and the scanner will do the rest as usual. But because of efficiency issues, JavaCC constructs new scanners as finite-state automata and no symbol table is ever used. There is no way to add new keywords to the scanner at runtime. The suggested solutions to this problem is to run JavaCC and compile a new compiler at runtime (Viswanadha, 2004) (or simply implement that part of the scanner by hand). I decided to omit multicharacter symbol support in the frontend.

The lexc notation includes regular expressions, though they are not as frequently used as the multicharacter symbols. The regular expressions forms a sublanguage by themselves and while they play an important part in compactly expressing complex parts of finite automata, I found the benefit of including support for them lower than the cost, since they wouldn't contribute significantly to the usability of the prototype and yet require work corresponding to that of implementing a whole new frontend.

Listing 1 A simple lexc snippet

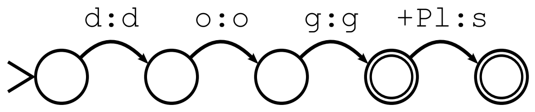
```
Multichar_Symbols +Pl

LEXICON Root
dog Noun;

LEXICON Noun
+Pl:s #;
#;
```

Some pitfalls exists regarding the generation of arcs and nodes from the lexc notation. As listing 1 shows, an entry in a lexicon may have no data and an end-of-word-token ("#"). The absence of data in an entry means that no arcs will be generated. No arcs means no nodes to mark as end nodes, which is a problem if the end-of-word-token is encountered. The solution is to always track the last node generated, even if it originated in an other lexicon, and to avoid generating nodes in advance, before it is known whether they will be used. By storing the first node each lexicon became connected to, the problem of finding nodes from preceding lexicon is solved along with another problem, the detection of cycles in the transducer. The first thing done when a lexicon is entered is to store the previous node encountered, unless that reference already points to a node, in which case a cycle has been detected and the processing of that particular lexicon should return (the lexicon has obviously been entered earlier).

5 Internal representation

The internal representation should as far as possible reflect the transducer as mathematical idea. In purifying it, the modularity of the compiler is preserved and the transducer becomes available for optimizing algorithms. No such algorithms have yet been implemented, but the possibility to combine transducers for re-writing rules with those handling lexicon is essential for any practical use of a phonological compiler, and a natural extension of the prototype presented would consist of algorithms for combining and optimizing transducers, together with a frontend for re-write rules.

Figure 1 Abstract representation of a transducer



6 Backend

One backend has been implemented, producing Prolog predicates suitable for SWI-Prolog. The intention is to load the predicates together with a Prolog interface program bundled with the compiler, but the external program could have been included in the file with the predicates without any problems. The predicates consists of start node marker, arcs and end node markers, more or less a pretty-printing of the internal representation. The node representation has changed though, the interesting parts in the transducers are the labeled arcs and very little information is connected to the nodes, thus there is no need to build predicates around the nodes. These are instead represented by unique non-negative integers.

When building a Prolog program from several lexc files, it is convenient to allow several different transducers in the same program, one for each lexc file. To avoid mixing up the enumeration of the nodes, a Singleton pattern is used, producing one single sequence of unique integers during the run of the compiler. Alas, this enumeration had been put in the internal representation as mentioned above.

The backend simplifies the syntax of the output to make it more readable. A general arc is encoded as `arc/4`, with start node, end node, lexical form and surface form being its arguments. If an arc contains the same symbol in both the lexical form and the surface form, it is encoded as `arc/3`, where the two forms are represented by one argument.

Note that multicharacter symbols is no issue for the backend, if they exist in the internal representation, they will show up in the Prolog predicates. On the other hand, the interface program must be able to recognize these symbols in the user input. A solution in this particular combination of frontend and backend would be to store a list of the multicharacter symbols in the internal representation or passing them immediately to the backend, then encoding them as predicates; `multicharsymbol/1`. This would contaminate the internal abstract model and break the modularity of the compiler. A better solution would be to scan through all arcs in the backend and construct the list of all multicharacter symbols before encoding them as before. This strains the Prolog interface; it must find the longest match when tokenizing user input. It could be extended but that would require unnecessary effort. Implementing a new interface has its advantages, I am free to choose whatever formalism I prefer, thus, I force the user to surround each multicharacter symbol with percentage signs, following the principle of KISS[2]. No need to encode any symbols as predicates or juggle tokenization in Prolog, the (relatively small) burden is laid upon the user instead.

The user interface in Prolog provides three predicates:

down(*+Atom*) finds all possible translations of *Atom* from lexical (upper) form to surface (lower) form and prints them.

up(*+Atom*) finds all possible translations of *Atom* from surface (lower) form to lexical (upper) form and prints them.

listall lists all possible translations. It does not work for transducers containing cycles, for obvious reasons.

Listing 2 Prolog representation of a transducer

```
startnode( 0 ).
arc( 0, 1, 'd' ).
arc( 1, 2, 'o' ).
arc( 2, 3, 'g' ).
arc( 3, 4, '+Pl', 's' ).
endnode( 3 ).
endnode( 4 ).
```

7 Conclusions

Starting with an optimistic idea about implementing frontends for both lexc and twolc (Karttunen and Beesley, 1992), I found out I had to cut down on my ambition a bit.

The implementation of the backend and the associated user interface turned out to be easy to accomplish, mainly because I was able to rely on Prolog's backtracking instead of implementing an engine on my own. The cost of using backtracking is that speed is lost, even if the transducer could be treated like a deterministic machine, it will be treated like a non-deterministic dito (which is slower).

The lexc formalism seemed straight-forward to me, but as I soon discovered, it would have been easier to implement a scanner of my own rather than twisting and bending JavaCC for a task it was not designed to handle. Needless to say, I did not realize in the beginning that regular expressions were a whole language on their own.

It would have been interesting to test the efficiency of the compiler's output by compiling a huge database but that would require a new frontend, I have not seen such a database in lexc notation.

Some features which should be implemented if the prototype is extended:

- Multicharacter symbol support in scanner.
- Regular expression support.
- Algorithms for combining transducers.
- Algorithms for optimizing transducers.

[2]Keep It Simple, Stupid!

References

Torbjörn Ekman, Görel Hedin, and Eva Magnusson. 2005. Jastadd. `http://jastadd.cs.lth.se`.

Lauri Karttunen and Kenneth R. Beesley. 1992. Two-level rule compiler. Technical Report ISTL-92-2, Xerox Palo Alto Research Center, October. available at `http://www.xrce.xerox.com/competencies/content-analysis/fssoft/docs/twolc-92/twolc92.html`.

Lauri Karttunen. 1993. Finite-state lexicon compiler. Technical Report ISTL-NLTT2993-04-02, Xerox Palo Alto Research Center, April. available at `http://www.xrce.xerox.com/competencies/content-analysis/fssoft/docs/lexc-93/lexc93.html`.

Helmut Schmid. 2001. SFST. `http://www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html`.

Sreenivas Viswanadha. 2004. [javacc] changing keywords at runtime. answer in a support forum `https://javacc.dev.java.net/servlets/ReadMsg?list=users&msgNo=408`.

Sreenivas Viswanadha. 2006. Javacc. `https://javacc.dev.java.net/`.

Jan Wielemaker. 2005. SWI-Prolog. `http://www.swi-prolog.org/`.

Lemmatiserare för okända ord

Boel MATTSSON
Lunds Tekniska Högskola
Lund, Sverige
d03bm@efd.lth.se

Abstract

En lemmatiserare hittar grundformen för ett böjt ord. Detta projekt har går ut på att göra en lemmatiserare för okända ord. En klassificerare har använts för att lösa problemet.

Rapporten beskriver bakgrunden till problemet och ger en kort beskrivning av hur en klassificerare fungerar. Träningsmängder och testmängder till klassificeraren har tagits från SUC-korpusen. Träningsmängder med olika antal ord har använts.

Resultaten visar att fler ord i träningsmängden ger bättre resultat och att denna metod fungerar bra.

1 Introduktion

Ibland stöter man på ord som inte påträffats tidigare. I det svenska språket kan man bilda många nya ord genom t ex sammansättingar. Orden *kräm* och *bluffen* kan sättas samman till ordet *krämbluffen*. Att hitta lemmat (grundformen) till ordet *bluffen* är enkelt eftersom *bluffen* är ett känt ord som finns i flera korpus. Detta gäller däremot inte för *krämbluffen*.

Syftet med projektet är att göra ett program som gissar lemmat för okända ord. Jag har begränsat mig till de öppna ordklasserna - substantiv, verb och adjektiv – eftersom det är i dessa ordklasser nybildning av ord oftast sker.

Metoden för att lösa lemmatiseringsproblemet har varit att formulera det som ett klassificeringsproblem. Det innebär att ett ord som *krämbluffen* betraktas som ett ord som tillhör den klass där man ska ta bort ändelsen *"-en"* för att få lemmat *krämbluff*. Syftet har varit att undersöka om detta angreppssätt är lämpligt.

Denna rapport är upplagd på följande vis. Avsnitt två beskriver vilka hjälpmedel jag använt mig av. Avsnitt tre går igenom implementationen och metoden jag använt. Resultat och slutsatser återfinns i avsnitt fyra respektive fem.

2 Hjälpmedel

I projektet har jag använt mig av en annoterad korpus och en statistisk klassificerare.

2.1 Korpus

Den korpus jag använder är SUC (1), Stockholm Umeå Corpus, som består av 1 miljon ord. För varje ord finns information om böjd form, ordklass och lemma. (se Tabell 1).

Särskilt	ab	särskilt
smygrustningen	nn.utr.sin.def.nom	smygrustning
vad	ha	vad
gäller	vb.prs.akt	gälla
missiler	nn.utr.plu.ind.nom	missil
oroar	vb.prs.akt	oroa
.	mad	.

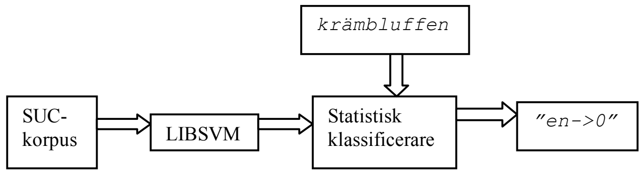
Tabell 1: Utdrag ur SUC-korpus

Texterna i SUC är skrivna under 1990-talet och hämtade från olika genrer.

2.2 Klassificerare

En klassificerare är en funktion som givet ett antal inparametrar ger en klass t ex "*en->0*" i fallet *krämbluffen*. För att skapa en klassificerare har jag använt mig av programmet LIBSVM (2) och hjälpprogrammet SimpleSVM (3). LIBSVM tar in ett antal exempel, träningsmängd, och använder sig av en statistisk algoritm för att generera en klassificeringsfunktion som passar så bra som möjligt med en träningsmängd. Hjälpprogrammet SimpleSVM förenklar användningen av LIBSVM.

3 Implementation



Figur 1: Översikt över implementationen

Figur 1 visar hur SUC-korpusen, eller rättare sagt delar av den, används som träningsmängd till LIBSVM för att träna en klassificerare. Därefter visas att en klassificerare skapas. Klassificeraren kan sedan användas för att klassificera ett valfritt okänt ord. SUC-korpusen har i projektet även använts som testmängd. I nedanstående avsnitt går jag närmare in på varje steg.

3.1 Träningsmängder och Testmängd

Som träningsmängd och testmängd har jag använt ett blandat urval ur SUC bestående av substantiv, adjektiv och verb. Testmängden består av 50 000 ord. Träningsmängden har bestått av mellan 1000 och 10 000 ord. Följande storlek på träningsmängden har använts:

- 1000 ord
- 2000 ord
- 5000 ord
- 10 000 ord

3.2 Inparametrar

För varje ord i träningsmängden behöver LIBSVM den korrekt klassificerade klassen – för ordet smygrustningen är den ”*en->0*” – och ett antal inparametrar. Jag har använt följande sex inparametrar:

- Ordets suffix upp till fem bokstäver
- Ordets ordklass

På formatet som hjälpprogrammet SimpleSVM använder blir indata för ordet *smygrustningen*:
`en>0|n|en|gen|ngen|ingen|nn.utr.sin.def.nom`

3.3 Träna klassificeraren

Orden i träningsmängden skrivs om på ovanstående format och sparas i en fil `data.txt`. Hjälpprogrammet läser in textfilen och omvandlar den till ett numeriskt format (`data.txt.processed`). Dessutom skapas ett kodningsobjekt (`data.encoding`). Jag har tränat klassificeraren med anropet:

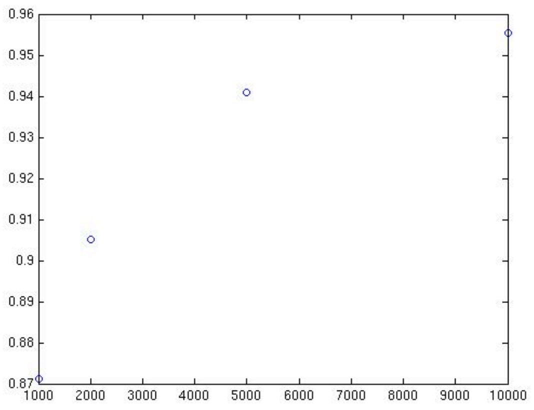
```
svm-train -c 256 -g 0.0025
data.txt.processed data.model
```

som skapar ett modellobjekt (`data.model`).

Kodningsobjektet och modellobjektet behövs när man använder klassificeraren.

4 Resultat

För de fyra olika träningsmängderna har jag mätt andelen korrekt klassificerade ord (se figur 2).



Figur 2: Andel korrekta klassificeringar som funktion av antalet ord i träningsmängden.

Jag har försökt ta reda på vilka grammatiska former som var svårast att klassificera. Eftersom inte alla former förekommer lika ofta i korpusen, varifrån test- och träningsmängderna är tagna, är det svårt att redovisa vilka ord som är svårast att klassificera. Jag har undersökt det fall då träningsmängden bestod av 10 000 ord. I tabell 2 redovisar jag de fem former som fick högst antal klassificeringsfel per antal förekomster i testmängden. Jag tar inte med former som förekom mindre än 70 gånger i testmängden eftersom jag antar att de inte förekommit så ofta i träningsmängden heller.

ordklass	**andel fel**
nn.utr.plu.def.gen	38%
vb.prt.sfo	31%
nn.neu.sin.def.gen	29%
nn.utr.plu.def.nom	28%
nn.neu.sin.def.nom	13%

Tabell 2: Ordklasser som var svåra att klassificera.

Exempel på ord i ordklassen nn.utr.plu.def.nom som blev felklassificerade är *pojkarna* och *gaffeltruckarna*. Dessa ord skulle enligt klassificeraren ha lemmorna *pojkare* respektive *gaffeltruckare* medan de rätta lemmorna är *pojke* respektive *gaffeltruck*.

Ett annat exempel är *barkborrarnas* (nn.utr.plu.def.gen) som enligt klassificeraren skulle ha lemmat *barkborr* men enligt korpusen har lemmat *barkborre*. Dock skulle ju klassificeringen kunna vara korrekt eftersom man kan tänka sig att det finns ett redskap som används för att borra i bark, en barkborr.

Ett sista och positivt exempel är ordet jag tog som exempel i inledningen, *krämbluffen* (nn.utr.sin.def.nom). Detta ord blev korrekt klassificerat. Så var även fallet för majoriteten av orden i denna ordklass. Bara 4% blev felaktigt klassificerade.

5 Slutsatser och diskussion

Slutsatsen man kan dra av resultaten i figur 2 är att ju fler ord träningsmängden har desto bättre blir klassificeraren. Med en träningsmängd på 10 000 ord blev 96% av orden i testmängden korrekt klassificerade. Kurvan ser ut att plana ut efter detta antal. Jag har inte undersökt resultaten för träningsmängder med fler ord eftersom det tog för lång tid att köra programmet då.

För att metoden ska ge ett helt korrekt resultat skulle man antagligen behöva fler inparametrar till klassificeraren. Ord som *pojkarna* och *gaffeltruckarna* har ju samma ändelser i böjd form men ska klassificeras olika. Dock är det svårt att säga vilka ytterligare inparametrar som behövs. Dessutom visar exemplet med *barkborrarnas* att det inte är möjligt att få en hundraprocentigt korrekt klassificering eftersom det finns ord som kan tolkas på olika sätt.

Slutligen kan man förstås tänka sig andra sätt att angripa problemet. Ett sätt hade kunnat vara att dela upp det okända ordet i kända ord. *Krämbluffen* skulle då delats upp i *kräm* och *bluffen*. Lemmat för *kräm-bluffen* skulle antas ha samma ändelse som lemmat för *bluffen*.

6 Tack

Tack till min handledare Richard Johansson och min föreläsare Pierre Nugues.

Också ett tack till min vän och diskussionspartner Kajsa Lindén.

Referenser

1. Eva Ejerhed, Gunnel Källgren, Ola Wennstedt och Magnus Åström: "The Linguistic Annotation System of the Stockholm-Umeå Project", 1992.
2. Chih-Chung Chang and Chih-Jen Lin: "LIBSVM: A Library for Support Vector Machines", 2001.
3. Richard Johansson: "SimpleSVM: A Java-based Wrapper Library for LibSVM". Software available at http://www.df.lth.se/~richardj/simplesvm.

Evaluation of the Tanaka-Iwasaki-algorithm for word clustering

Mats Mattsson
p02mm@efd.lth.se

Jonas Åström
p02jas@efd.lth.se

Abstract

We have implemented and tested the algorithm described in (Tanaka-Ishii and Iwasaki, 1997).

It is about clustering words based on the co-occurence graph by using transitivity.

We find similiar, but less exact, results. However we have been unable to test the algorithm on a corpus of the same size.

1 Introduction

1.1 Equality relation

A relation can be represented as a graph where vertices a and b are said to be related if there is an edge from a to b. It can be written aRb.

An equality relation (R) is reflective ($aRa, \forall a$), symmetric ($aRb \Rightarrow bRa$) and transitive ($aRb, bRc \Rightarrow aRc$).

1.2 Co-occurrence graph

A graph can be formed from words that co-occur in a corpus. Words are represented as vertices. An edge between two vertices indicate that they co-occur.

This graph can be viewed as an equality relation. Partitioning the graph would give groups of words connected to one topic. Such groups can be used for construction and validation of a thesaurus and clustering of documents.

Both reflectivity and symmetry are guaranteed in the co-occurence graph. Transitivity is usually not present.

1.3 Loosening constraints for subgraph extraction

We loosen the requirement of transitivity for the subgraph. I.e it no longer needs to be a complete graph. Instead of an edge between any two vertices, we only require each vertex in the subgraph to be a part of a complete graph of four vertices. E.g. figure 1.

In (Tanaka-Ishii and Iwasaki, 1997) more theory around the loosening of constraints is discussed.

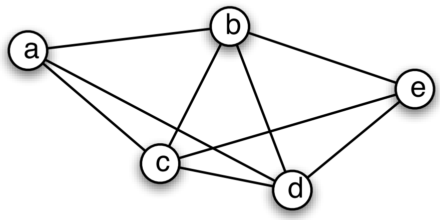


Figure 1: To be a transitive graph an additional edge between vertices a and e is required. After loosening the constraints this graph will be considered transitive.

1.4 Algorithm for clustering

We extract a subgraph A from the co-occurance graph G.

Step 1 Starting from edge e. Put a triangle graph including e into A.

Step 2 For a branch $e' \in A$: If there exists nodes $v \in G$ and $v' \in G$ both forming a triangle with e' and connected to each other, put v and all edges connected to v into A.

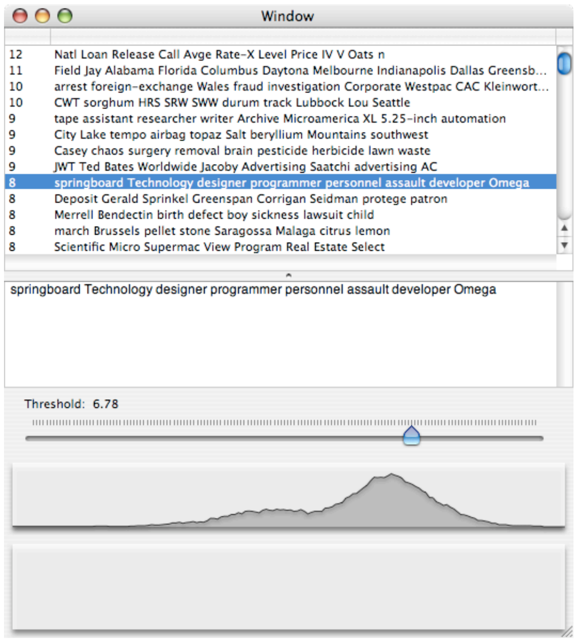
Step 3 Repeat step 2 until A cannot be extended any more.

By starting from every triangle in G we will find all subgraphs.

By limiting our output to maximal subgraphs we only have to start from edges not already included in previously calculated subgraph. Some extracted graphs may be parts of others so this needs to be checked.

2 Co-occurance measure

We use the notion of mutual information similar to (Church and Hanks, 1990), which is used in (Tanaka-Ishii and Iwasaki, 1997). Our co-frequency measure is symmetrical and we also



```
12    Natl Loan Release Call Avge Rate-X Level Price IV V Oats n
11    Field Jay Alabama Florida Columbus Daytona Melbourne Indianapolis Dallas Greensb...
10    arrest foreign-exchange Wales fraud investigation Corporate Westpac CAC Kleinwort...
10    CWT sorghum HRS SRW SWW durum track Lubbock Lou Seattle
9     tape assistant researcher writer Archive Microamerica XL 5.25-inch automation
9     City Lake tempo airbag topaz Salt beryllium Mountains southwest
9     Casey chaos surgery removal brain pesticide herbicide lawn waste
9     JWT Ted Bates Worldwide Jacoby Advertising Saatchi advertising AC
8     springboard Technology designer programmer personnel assault developer Omega
8     Deposit Gerald Sprinkel Greenspan Corrigan Seidman protege patron
8     Merrell Bendectin birth defect boy sickness lawsuit child
8     march Brussels pellet stone Saragossa Malaga citrus lemon
8     Scientific Micro Supermac View Program Real Estate Select

springboard Technology designer programmer personnel assault developer Omega

Threshold:  6.78
```

Figure 2: Screenshot of the resuts presented by our implementation. The graph displays the number of clusters for different co-occurance thresholds.

use a finite weighted window

$$w_i = \exp(-\alpha|i| - \beta i^2).$$

As most texts change the subject between paragraphs we add extra distance between them, i.e. the distance between the last word of the previous paragraph and the first word of the current is 7 instead of 1.

To form the graph we set a threshold for the mutual information between two words and say they co-occur when it is above the given threshold.

3 Implementation

We have implemented the algorithm in Objective-C++ as a Cocoa application for Mac OS X. See figure 2 for a screenshot.

3.1 TreeTagger

We use TreeTagger (University of Stuttgart, 2005) to mark the part of speech each word has and find the lemma (e.g. am → be) for words.

We only consider nouns when running the algorithm.

4 Results

As corpora we have used different texts collected from the internet and a part of Reuters-21578 from from the Reuters newswire 1987.

We have only been able to use the algorithm on about 4% of the 15MByte Reuters corpus. Example of the largest clusters:

12 Natl Loan Release Call Avge Rate-X Level Price IV V Oats n

11 (Cities, States) Field Jay Alabama Florida Columbus Daytona Melbourne Indianapolis Dallas Greensboro Jacksonville

10 (Economic crime) arrest foreign-exchange Wales fraud investigation Corporate Westpac CAC Kleinwort Benson

10 CWT sorghum HRS SRW SWW durum track Lubbock Lou Seattle

9 (Writing) tape assistant researcher writer Archive Microamerica XL 5.25-inch automation

9 (Mining) City Lake tempo airbag topaz Salt beryllium Mountains southwest

9 (Cultivation) Casey chaos surgery removal brain pesticide herbicide lawn waste

9 JWT Ted Bates Worldwide Jacoby Advertising Saatchi advertising AC

8 (Designer) springboard Technology designer programmer personnel assault developer Omega

8 Deposit Gerald Sprinkel Greenspan Corrigan Seidman protege patron

8 (Children's disease Merrell Bendectin birth defect boy sickness lawsuit child

8 march Brussels pellet stone Saragossa Malaga citrus lemon

8 Scientific Micro Supermac View Program Real Estate Select

8 Governor Exchequer Nigel Lawson Geoffrey Howe Robin Leigh-Pemberton

8 (Iranian army) attack Iranian Army Revolutionary guard Corps Third commander

About half of the groups have a possible topic, even though there are some noise present.

5 Discussion

In (Tanaka-Ishii and Iwasaki, 1997) a 30MByte corpus from Wall Street Journal is used. They achieve good results for 39 clusters of sizes from 8 to 105 words. There are some noise present but much less than we have encountered.

As the co-occurrence measure by mutual information is more noise resistant for large corpora this may explain the difference between our results and those in the original article.

The running time of our implementation is between 5 and 30 seconds for one clustering of 800kBytes, depending on the mutual information threshold for generating the co-occurance graph.

We have not made much effort to analyze the time complexity of the algorithm. The running time growths worse than linear and probably at least quadratic with the input size.

References

K.W. Church and P. Hanks. 1990. Word associaton norms, mutual information, and lexicography. *Computational Linguistics*, 16(1):22–29.

Kumiko Tanaka-Ishii and Hideya Iwasaki. 1997. Clustering co-occurrence graph based on transitivity. *5th Workshop on Very Large Corpora*, pages 91–100.

Institute for Computational Linguistics University of Stuttgart. 2005. Treetagger. *http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/DecisionTreeTagger.html*.

RoboLinguistics

Ett textförståelseprogram

Henrik Palmér, d01hp

Inledning

För att styra robotar räcker det inte att som i science-fictionfilmer bara säga till dem vad de ska göra. De största kraftansträngningarna inom AI för industrirobotar har inte lagts på att utveckla ett begripligt användargränssnitt gentemot dem, utan snarare på deras funktion. Det har gjort att vi nu sitter här med flexibla robotar som fungerar mycket bra så länge de ska utföra sina förprogrammerade arbetsmoment, men som ofta kräver en tekniker för att programmeras om.

RoboLinguistics är ett försök att komma förbi detta. Tanken med programmet är att man ska kunna instruera roboten på naturlig engelska. För att göra detta möjligt måste programmet inte bara kunna grammatiskt strukturera meningar med olika uppbyggnad, utan även kunna lösa koreferens och slutligen förstå vad olika kommandon betyder rent praktiskt.

Programmoment

Programmet är uppdelat i olika moment som är ansvariga för olika delar av textbehandlingen. Först skapar Charniaks parser ett träd av satsdelar som beskriver meningen grammatiskt. Detta är nödvändigt för att det andra delmomentet ska fungera. Den semantiska modulen söker upp agenter, predikat och deras argument.

Slutligen bearbetar den del av RoboLinguistics som jag har skrivit denna lista och finner verb roboten är programmerad att känna igen. För verbgrupper där dessa ingår går programmet igenom alla argument och jämför dem med dem som tidigare nämnts för att förstå vilka de är.

Den semantiska modulen

Det den semantiska modulen finner de olika predikaten i utdatan från charniakparsern. För varje predikat söker den sedan efter agenten och argumenten. När den funnit argumenten måste den dessutom klassificera dem, så att agenter och direkta objekt får rätt etiketter. Till sin hjälp har den PropBank, ett corpus på många tusen ord, där olika betydelser av verb och deras argument finns utmärkta.

Innan denna modul kan användas måste den instrueras vilka predikat den ska söka efter. Därför kommer okända verb att helt ignoreras av RoboLinguistics.

Utdatan från den semantiska modulen är ett meningsträd. Detta innehåller element med en klassificering, pekare till var i texten de förekommer samt signaturträd i de fall de är argument till verb.

Detta program hanterar två sorters signaturträd: substantivgrupper och prepositioner. En substantivgrupp har som rot huvudordet i gruppen och dess satsdel: *the red ball* kommer till exempel att ha *NP/ball/NN* som rot. Denna beteckning innebär att signaturträdet är en substantivgrupp (NP) och *ball* är ett substantiv i singular (NN).

Ett signaturträd kan också vara en samling av flera signaturträd. Detta sker vid konjunktioner som *the red ball and the blue ball* och relationer som *the book on the shelf*. I det senare fallet kommer det andra underträdet vara av typen preposition. Dessa har som underträd signaturträdet för substantivet, i fallet från den föregående meningen *the shelf* . Som rot kommer denna att ha *PP/on/IN*.

Textförståelsedelen

Trots att texten nu delats upp i sina verbgrupper och argumenten till verben till stor del har identifierats är datan ännu inte redo för roboten. För det första är datan i utdatastrukturen i princip bara text, och för det andra är ingen koreferens ännu löst. Utdata från textförståelsedelen är en lista av kommandon, som *put*, *drill* och *remove,* som alla har varierande antal argument och en lista på alla *entiteter* som nämnts i texten.

Det första programmet gör är att gå igenom listan som den semantiska modulen skapat. Denna kommer normalt att bestå av en rad argument följd av ett verb.

Argument

Argument som RoboLinguistics förstår är de som kallas *An* i PropBank (där *n* är ett nummer). När dessa dyker upp läggs deras signaturträd i en hashtabell med argumentnamnet som nyckel. Denna hashtabell är unik för varje verb.

Verb

Textförståelsedelen representerar all kommandon den känner till med klasser. Till exempel representeras verben *get* och *take* av klassen *Get*. Innan ett kommando kan användas av programmet måste systemet få reda på att det existerar. Detta görs genom att lägga till en referens till deras klass tillsammans med namnet på verbet i en hashtabell. Varje klass kan därför kopplas till flera verb.

RoboLinguistics söker upp vilken klass som är kopplad till det aktuella verbet. Om ingen hittas genereras ett felmeddelande och all vidare bearbetning av detta verb och dess argument avbryts, annars skickas namnet på verbet, hashtabellen med argument och listan på tidigare nämnda entiteter till konstruktorn med hjälp av Java-reflektion.

Vad som sedan händer beror på vilket kommando det handlar om. De mest komplicerade som är med i RoboLinguistics för tillfället är de som tar två argument: direkta objekt och platser, som är prepositioner följda av substantivgrupper. Kommandot *remove* tillsammans med några andra gör detta.

Först tar kontruktorn reda på signaturträderna för det direkt objektet och för platsen. Om den finner båda dessa innebär det att den semantiska modulen fungerat felfritt. Detta sker dock sällan för mer komplicerade substantivgrupper med både adjektiv och prepositioner, för vilka både objekt och plats hamnar som samma argument. I dessa fall försöker kontruktorn bryta upp objektargumentet i en del före prepositionen och en med prepositionen och dess substantivgrupp.

Konstruktorn försöker sedan lösa koreferenser för att komma fram till vilka entiteter som ska läggas i dess argumentlista. För argument som kan vara olika prepositioner, som till exempel *put on* och *put into* skapas en instans av en underklass till *Preposition*, som innehåller en referens till en entitet och en specifikation, men om prepositionen är entydig läggs bara entiteten till för enkelhets skull. Vad som görs bestäms entydigt av kommandot.

Koreferens

Att lösa koreferenser är RoboLinguistics största utmaning. Klassen som har hand om detta är *EntityContainer*. Denna upprätthåller en lista över alla entiteter som någonsin nämnts. Entiteter representeras av klassen *Entity* med attribut för med vilken signaturträd den introducerades, vid vilka andra signaturträder den benämnts och som vilket argumentnummer den senast förekom, eller dess underklasser *RelationalEntity*, som är substantivgrupper som *the lid of the box* eller *the book on the table* med attribut för vilken entitet den är relaterad till och hur, och *MultipleEntities*, som är uppräkningar av flera substantiv, som *the small gerbil and the brown hamster*.

Om alla substantiv i uppräkningen har samma huvudord sätts signaturträden för entiteten till plural för detta. Vad plural av detta ord är bestäms genom att det först jämförs med ord som helt förändras, som *index*/*indice*. Om ordet inte finns i denna lista jämförs det med ord som får en förändrad ändelse, som *cactus*/*cactii* eller *box*/*boxes*. Om det inte återfinns här heller läggs helt enkelt ett s till slutet.

För att komma runt problemet att den semantiska modulen inte alltid lyckas hitta platsargument har en förenkling gjorts i *MultipleEntities*; om en uppräkning av entiteter slutar med en *RelationalEntity* antas alla entiteter i uppräkningen vara av samma klass och med samma relation. Detta innebär att *the lid and the contents of the box* förstår som *the lid of the box and the bottom of the box*, men också att *the lid of the box and book on the table* förstås som *the lid on the table and the book on the table*.

EntityContainer har en metod *findEntity()* som tar som parameter en signaturträd och vilket argument till ett kommando denna har. Därefter skapas en lista av entiteter baserad på denna. Om signaturträden inte är en konjunktion av substantivgrupper kommer listan bara att innehålla ett element.

Dessa element jämförs sedan med de redan existerande entiteterna. Om signaturträden för entiteten är ett pronomen jämförs först numerus för de olika entiteterna. Om dessa stämmer ges en grundpoäng som sedan ökas beroende på om den existerande entiteten någon gång tidigare benämnts vid detta pronomen och i så fall hur länge sedan. Detta behövs för meningar som den i *Exempel 1*, där det sista kommandot annars skulle ha blivit *glue the screw to the small box*.

När denna poäng slutligen räknats ut divideras den med ett tal i storleksordningen av det argumentnummer den existerande entiteten tidigare varit. Detta görs eftersom programmet antar att ett pronomen oftare refererar tillbaka till ett direkt objekt än platsargument. Utan detta hade det andra kommando i *Exempel 1* blivit *put a screw in the big box*.

Om signaturträden istället är en substantivgrupp jämförs först dess huvudord med dem för de redan kända entiteterna. Om dessa stämmer jämförs varje ord av typen substantiv och adjektiv i den kortaste signaturträden med alla ord i den längre. För varje matchande ord ökas den totala poängen, medan den minskas för varje ord som inte återfinns i den längre listan. Därefter delas poängen med antalet jämförda ord.

Om båda entiteterna är relationsentiteter jämförs även dessa, och utfallet av denna jämförelse får större vikt än den tidigare, eftersom programmet måste se skillnad på *the big red lid of the box* och *the big red lid of the coffin*. Slutligen delas poängen med ett tal i storleksordningen roten av avståndet i entiteter mellan den nyskapade entiteten och den tidigare kända.

Om ingen tidigare entitets poäng överstiger ett tröskelvärde behålls den nyskapade entiteten och läggs först i listan på kända entiteter, annars läggs istället den matchande entiteten först. På så sätt ligger alltid den senast nämnda entiteten först.

I de fall då den semantiska modulen inte funnit ett platsargument och istället slagit ihop det med objektargumentet uppstår ett problem; istället för att skapa en entitet och ett prepositionsobjekt skapas istället en relationsentitet, och roboten kommer att tro att den ska *ta ”boken från hyllan”* och inte *ta ”boken” från hyllan*. För att råda bot på detta finns möjligheten att bryta upp en relationsentitet till en enkel entitet och en signaturträd som beskriver relationen. Detta görs i kontruktorerna till vissa kommandon om vissa villkor uppfylls.

För det första måste relationen vara den senast introducerade entiteten. Om *the book on the* shelf redan nämnts kan det inte handla om att lägga boken på hyllan. För det andra måste relationen vara av en typ som är vettig för kommandot. För kommandot *open* är till exempel relationen *the door with the key* vettig.

Framtida utvidgningar

Det stora felet med koreferenslösningen är att konjunktionera av relationer begränsas till att vara relationer till samma entitet, och detta måste åtgärdas i kommande versioner av programmet.

Jag har medvetet valt att göra programmet så lite beroende av de underliggande modulerna som möjligt. Det innebär att förutom att dela upp relationer försöker det inte att korrigera misstag från charniakparsern eller den semantiska modulen. Det är härifrån i princip alla felaktiga tolkningar kommer. Till exempel lyckas programmet inte se sambandet mellan *the prickly cactus and the tall cactus* och *the cactii* eftersom charniakparsern tror att *the cactii* är singular.

Programmet antar att varje verb som förekommer i texten är en uppmaning. Detta anser jag vara ett rimligt antagande, eftersom poängen med det är just att finna kommandon i en löpande text. Alternativet vore att ignorera verb som inte är uppmaningar, men det skulle leda till att information från texten gick förlorad. Detta leder dock till fel som i *Exempel 3*.

De fel som stammar från det sista steget beror till största delen på att entiteterna bara minns hur de benämnts och vilka roller de spelat och inte till vilka verb de spelat dessa roller. Detta illustreras av *Exempel 4*. För att lösa detta måste den modularitet där det är lätt att lägga till nya kommandon ge vika för ett system där kommandon känner till varandra och vet hur de ska förhålla sig till varandras argument. I denna övervägning prioriterar jag enkelhet.

Något som däremot måste läggas till om programmet ska få någon nytta är en databas med objekt tillgängliga för roboten. I nuläget representeras kommandon av klasser som är kopplade till en handling, medan entiteterna bara beskrivs av strängar.

Appendix

Exempel 1

”Remove the lid from the big box and put a screw in it, and then glue it to the small box.”

```
remove the lid (E1) from the big box (E2).
put a screw (E3) in the lid (E1).
glue the lid (E1) to the small box (E4).
--------------------------------------------------------------------------------
Known entities:
the lid (E1)
  it
  the lid
the small box (E4)
  the small box
a screw (E3)
  a screw
the big box (E2)
  the big box
```

Exempel 2

”Open the big box and remove a screw from it. Put the screw in the center of the lid of the box and place the lid on the box. Please glue a handle to the lid of the small box. Open the small box and put an apple in it and then close the box.”

I detta exempel finns specificeraren *in the center* i den andra meningen. I det tredje kommandot har därför *the center* inte något ID; det existerar inte som en entitet. Den tredje meningen dyker inte upp överhuvudtaget, vilket beror på att charniakparsern felaktigt identifierat *glue* som ett substantiv: ”(S1 (S (NP (NN Glue)) (DT a) (VP (VB handle) (PP (TO to) (NP (NP (DT the) (NN lid)) (PP (IN of) (NP (DT the) (JJ small) (NN box)))))) (. .))).”

```
open the big box (E1).
remove a screw (E2) from the big box (E1).
put a screw (E2) in the center of the lid (E3) of the big box (E1).
put the lid (E3) of the big box (E1) on the big box (E1).
open the small box (E4).
put an apple (E5) in the small box (E4).
close the small box (E4).
--------------------------------------------------------------------------------
Known entities:
the small box (E4)
  the box
  it
  the small box
an apple (E5)
```

```
  an apple
the lid (E3) of the big box (E1)
  the lid
the big box (E1)
  the box
  it
  the big box
a screw (E2)
  the screw
  a screw
```

Exempel 3

”He opened the door.”

```
open the door (E1).
--------------------------------------------------------------------------------
Known entities:
the door (E1)
  the door
```

Exempel 4

”Open the box with the key and then open the door with it.”

I detta exempel illustreras hur programmet förvirras av grammatiska konstruktioner som en människa lätt genomskådar. Eftersom systemet inte känner till att man vanligtvis öppnar saker med nycklar och inte lådor säger dess regler åt den att välja lådan som instrument för att öppna dörren.

```
open the box (E2) with the key (E1).
open the door (E3) with the box (E2).
--------------------------------------------------------------------------------
Known entities:
the door (E3)
  the door
the box (E2)
  it
  the box
the key (E1)
  the key
```

A quick approach to Summarizing

Magnus Skog, Jacob Persson

23rd January 2006

Abstract

This project deals with the difficult problem of automatic text summarization. This is an interesting and open-ended problem that has a multitude of uses. We attack the problem by using both a Linguistic Summarizer written in Prolog and a simpler sentence pruning algorithm written in Perl. The basic idea is to first cut the unnecessary parts out of the sentences and then prune the less important sentences creating a summary. The results were evaluated by a group of testers that compared the original articles with the summaries. The overall results were satisfying but there is more work to be done.

1 Introduction

Text summarizing is an interesting and difficult problem. In this project we attempt to create good summaries of shorter texts of about 1000 characters. We based this project on the work of Polanyi,Culy, van den Berg, Thione and Ahn(2004) and the the work of Thione, van den Berg, Polanyi and Culy(2004). We also use a parser created by Collins(1998). The system is divided into two parts: A discourse parser and a sentence pruner.

2 Linguistic summarizing

This part of our system uses a linguistic approach to remove overflow information inside sentences. The advantage of the linguistic models is that they often rely on both syntactic and semantic knowledge.

A linguistic model for summarizing usually builds up a tree/structure of the discourse to be summarized according to a discourse model. The actual summarizing is then performed by pruning branches with less interesting information.

2.1 Linguistic discourse models

A model we looked at was the linguistic discourse model(LDM) by Polanyi and Scha (1984, 1988). It starts with segmenting the discourse into basic discourse units (BDU) using a set of rules. Then a a tree of DCU's(discourse constituent unit) are build up from the BDU's and according to the structure of the discourse.

Between the DCU's are relations that can be divided into three classes: discourse coordination, discourse subordination and n-ary constructions. Discourse coordinations are relations where the parts have equal importance, like *"he shot the president and he got away with it"*. Discourse subordination express relations where something is a elaboration of something, for example *"jurgen is the VD of Janco, which is a large company on Iceland"*. N-ary construction are lists or enumerations of things.

A similar model is that of the rhetorical structure theory(RST) by Mann and Thompson (1988). There the discourse is segmented into nucleuses and satellites, where the nucleus is the main segment and the satellites hold some relation to it. Relations can for example be elaboration, cause, motivation and concession.

2.2 A simplified linguistic discourse model

Our model starts from a tree created by the August parser, which output a tree in the Penn tree-bank style. We don't segment our discourse instead segmenting is included into the rules.

Since we also use a statistical summarizer on a per sentence basis we decided to let our LDM handle only one sentence at a time. This means of course that it can't solve co references or deal with any relationships that span across sentences.

With these simplification made the difficulty in implementing this model is to write the rules that find relations between discourse segments.

2.3 Implementation

We choose to implement our LDM in Prolog because its build in search mechanism make rules easy to apply.

The output from the August parser is converted to a tree of Prolog lists. On this tree we do a bottom up search and match rules. The relations we are interested in are mostly those of the elaboration type and since this is the last and only step we prune the tree at the same time.

A rule for identifying a elaboration can look like this. Here the elaboration is identified by the cue word 'which'. You can also see how we prune the tree by only saving what's matched to X.

```
   match(Tree, ['NP', X]) :- ['NP' | Y] = Tree, append(X, [[(',',),
[(',',)]], ['SBAR', ['WHNP', ['WDT', [which]]], _], [(',',), [(',',)]]],
Y).
```

Taking the leafs of the pruned tree in a inorder walk results in the summarized sentence. This is applied to every sentence in the discourse. A couple of post processing steps are finally used to give a proper output.

3 Sentence Pruner

3.1 Introduction

Unlike the Prolog summarizer, the sentence pruner only works with whole sentences. It prunes unwanted sentences using an algorithm that is based on word frequency and the length of the sentences

3.2 The Algorithm

The algorithm starts by splitting the text into sentences. It also makes a list of all the words in the text together with a score which is equivalent to the number of times the word appears in the text. Each sentence is then processed by adding the scores for all the words in the sentence. If the sentence contains words that appear in the title of the text, then the score of the sentence is improved. Also, the first sentence in the text is given a bonus to it's score due to the general importance of the very first sentence after the title. When this has been done for all the sentence in the text, the algorithm then prunes the unwanted sentences. Input to the algorithm is a percentage. This is the amount of sentences that the algorithm won't prune. The top scoring sentences are kept while the rest are removed.

3.3 Implementation

We chose to implement the algorithm in Perl. Perl is the natural choice for this kind of simple text manipulation.

3.4 Improvements

The algorithm does several passes through the text. It might be possible to score the sentences and count the words at the same time, thus saving time by doing less passes. Also, the implementation uses regular expressions to prune the sentences. It does this one by one, thus doing several unnecessary passes through the text. A good improvement would be to make one pass during the text that prunes all unwanted sentences.

4 Evaluation

4.1 Introduction

The program was evaluated using a test-set of the corpus. Five texts were selected from the text-set and we ran the program on them. We pruned 75 percent of the sentences. The original text and the results were posted on a website where a number of people read the texts and scored them. An interesting result was that we accidentally forgot to mention that the texts were not summarized by hand, so most testers believed that the summaries were done by a human.

4.2 Expected Results

From the results of the training corpus we expected some trouble with co-references. The program would prune sentences containing a name of a person and then later on keep a sentence containing âHe went to the marketâ and no reference to what âheâ was could be found in the summary. We found this to be a major problem but we simply did not have the time to implement a co-reference solver to fix

4.3 Results

As expected, most of the negative comments were that there were numerous problems with co-references. Most texts contained these problems. Other than that, most testers were fooled by the program and still believed that the summaries had been written by hand. One tester commented, and I quote: 'It looks like you have been a bit lazy when you summarized these texts'. All testers thought that the readability of the texts were good to excellent. Most of them, if not all, were very surprised when they later found out that the summaries were actually made by a computer program

4.4 Comments

It seems as the expected results were indeed reasonable. A co-reference solver would have remedied most of the problems and improved readability significantly.

5 Conclusion

The overall performance of our system was very good. The summaries were most of the time very easy to read and they often looked hand-made. In order to create better summaries we would have to start by implementing a co-reference solver. That is the beyond the scope of this short project. Another problem is the extremely bad performance of the summarizer. This is however caused by the parser. A faster parser would improve performance but this could never be used in real-time application such as websites. This is unfortunate because a website would have been a very interesting application for this project. We could for example summarize news articles for quick and easy reading. This would however require a much faster parser and most likely a dedicated high-performance server.

6 References

Gian Lorenzo Thione, Martin van den berg, Livia Polanyi and Chris Culy. 2004. Hybrid Text Summarization: Combining External Relevance Measures with Structural Analysis

Livia Polanyi, Chris Culy, Martin van den Berg, Gian Lorenzo Thione, David Ahn. 2004. A rule based Approach to Discourse Parsing

Pierre Nugues. 2005. An introduction to Language Processing with Perl and Prolog

7 Appendix A: Running Instructions

7.1 Requirements

The program requires SWI-Prolog and Perl to be able to run

7.2 Instructions

In the root directory of the program there is a Perl script called ”run.pl”. The script runs the entire program, automizing most of the tedious longer run scripts. The corpus is available in the corpus directory. The parser must be available in the root directory of the program.
The usage is ”Perl run.pl text percentage ”title words””.

Example 1: Perl run.pl corpus/dev-set/prohibition 25 ”prohibition”
Example 2: Perl run.pl corpus/test-set/wikipedia 15 ”wikipedia”

You don’t have to put in any title words but the results will be better if you do.

8 Appendix B: Links

The project files,including the parser and all of the articles used in the evaluation, are available at:
http://www.snakeeyes.nu/project/

Identification of time expressions, signals, events and temporal relations in texts

Cyril Perrig
ETH Zurich,
Switzerland,
perrigc@student.ethz.ch

Abstract

In this project an approach to time and event annotation is presented. First time expressions, signals and events are identified in texts. Using these extracted features temporal relations between events are identified. Furthermore the performance of event recognition with increasing corpus size is investigated.
We use the widely accepted *TimeML* specification language as an annotation scheme. The identification tasks are solved by a machine learning technique called *Support Vector Machines (SVMs).*
Finally we report the results we obtained and discuss conclusions and open problems.

1 Introduction

The goal of this project is to detect all the features needed to identify temporal relations in texts and at last to identify these temporal relations. In other words we want to determine a temporal relation between two events. For this we first have to detect time expressions, then signals and finally events. Due to time reasons the focus of this project is set on the detection of events while the identification of temporal relations itself is not fully explored. The results of the event recognition is also compared to a paper from IBM (Boguraev and Ando, 2005).
The order of detection is important because detections of a specific item might be used as a feature input for another detection. Before explaining these identifications in more details a short overview of TimeML is highlighted. Then the machine learning tool used for the tasks and the experimental setup is reviewed. In the next section all tasks are evaluated step by step. Finally the results, conclusions and open problems are discussed.

2 TimeML

TimeML 1.1[1] is a robust specification language for events and temporal expressions in natural language. It is an XML-based language and provides all tags needed for our analysis. Amongst other things it is designed to address the problem of ordering events with respect to one another. Besides all these reasons for choosing TimeML it is widely accepted and used in the language processing research.
In the following subsections all the relevant tags for this project are briefly explained and illustrated with an example. TimeML is by far more complex and therefore the interested reader is referred to the TimeML specifications 1.1 (Sauri et al., 2004) for more details.

2.1 Time Expressions

Time expressions in TimeML are tagged with TIMEX3 tags. This tag is primarily used to mark up explicit time expressions, such as times, dates, durations, etc.
An example TIMEX3-tag (if "today" is the 2006-01-17):

<TIMEX3 type="time" value="2005-01-16T10:00">yesterday at 10 am</TIMEX3>

2.2 Signals

The tag SIGNAL is used to annotate sections of text, typically function words, that indicate how temporal objects are to be related to each other. This can either be an indicator of temporal relations (e.g. *during, when, etc.*) or an indicator of temporal quantification (e.g. *twice, three times, etc.*).

2.3 Events

An EVENT is typically described by a verb, although event nominals, such as "crash" in "...killed by the crash", will also be annotated as events. Further events have a tense, an aspect (both optional), and a *class*. The

[1]http://www.timeml.org

class classifies the type of event, wheter it is e.g. a state or an intentional action. All in all TimeML distinguishes seven different event types. The attributes tense and aspect can each have four different values, *past, present, future, none* and *progressive, perfective, perfective_progressive, none* respectively. In Table 1 an example with tense value *present* is illustrated.

Verb group	Aspect
teaches	*none*
is teaching	*progressive*
has taught	*perfective*
has been teaching	*perfective_progressive*

Table 1: Example with tense value *present*

2.4 Temporal Relations

Temporal relations are marked with TLINK tags. It represents the relation between two temporal elements. Such links can connect time expressions with events or pairs of events. There are thirteen different temporal relation types, e.g. *before, after, includes etc.*
Let's consider the following sentence:
"Peter came home at ten o'clock and after eating, he went to bed."
Obviously there are three events and one time expressions in this example. Between the first event *(came)* and the time expression *(ten o'clock)* the tlink tag is of type *simultaneous.* The second temporal relation connects the two events *eating* and *went to bed* with an after-relation.

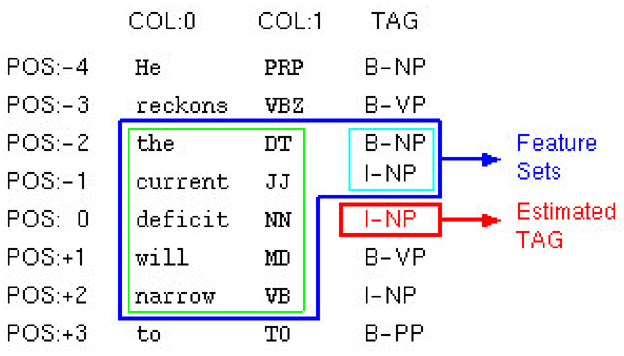
2.5 TimeBank

TimeBank 1.1 is an illustration and proof of concept of the TimeML specifications 1.1. It is a set of 186 news report documents annotated with the 1.1 version of the TimeML standard for temporal annotation. We use this freely provided illustration as a corpus in this project. With around 75'000 tokens one has to be aware that this corpus is rather small.
To end this section an example sentence from TimeBank 1.1 containing two events is presented:

On the other hand, it's <EVENT eid="e1" class="OCCURRENCE" >**turning**</EVENT>out to be another <EVENT eid="e84" class="STATE" >**very bad**</EVENT>financial <TIMEX3 tid="t83" type="DURATION" >week</TIMEX3>for <ENAMEX TYPE="LOCATION">Asia</ENAMEX>.

3 YamCha

YamCha[2] (Yet Another Multpurpose CHunk Annotator) is a generic, customizable, open source text chunker oriented towards a lot of NLP tasks. Further it provides a lot of useful features which can easily be used by changing the standard input parameters. If not stated otherwise we use the standard input parameters of YamCha. Concerning the feature sets (window-size) the default setting is illustrated in Figure 1.
YamCha is using a state-of-the-art machine learning algorithm called Support Vector Machines (SVMs). This classification algorithm provides a high generalization performance independent of feature dimension. Combinations of multiple features can be trained by using a Kernel Function. In YamCha only polynomial kernels are supported.



	COL:0	COL:1	TAG
POS:-4	He	PRP	B-NP
POS:-3	reckons	VBZ	B-VP
POS:-2	the	DT	B-NP
POS:-1	current	JJ	I-NP
POS: 0	deficit	NN	I-NP
POS:+1	will	MD	B-VP
POS:+2	narrow	VB	I-NP
POS:+3	to	TO	B-PP

Figure 1: Default feature set (window-size) setting in YamCha

4 Experimental Setup

First of all the corpus has to be tokenized to get a "YamCha-compatible" format. Additionally all relevant information belonging to a token, such as e.g. tense, has to be extracted. Unfortunately the POS (Part-of-Speech) of a token is not provided in the TimeBank-corpus. Therefore we used MXPOST[3] (Maximum Entropy Part-of-Speech Tagger) to get the POS of the tokens.
Another issue we have to be aware of is that sometimes more than one token is inside an entity (or tag). The first token of such an entity is often not very relevant for the detection. An example is illustrated in the time expression

[2]http://chasen.org/ taku/software/yamcha/
[3]http://www.cogsci.ed.ac.uk/j̃amesc/taggers/MXPOST.html

"*the last twenty hours*" where the determiner *the* is certainly not an indicator for a time expression. Therefore we use the IOB-model (Inside Outside Beginning) which distinguishes tokens that begin, are inside or are outside an entity. In Table 2 the end of the previous example sentence is presented.
To evaluate the results for the different tasks the measures *recall, precision and F-measure* are used. In the following subsection a short recall of these measures is given. Furthermore we use a 5-fold cross validation to get an "averaged" F-measure. First the corpus is divided into 5 equally sized parts. Then each part is once used as a test set (20% of corpus) and the others as a training set (80% of corpus). This means that each training set contains around 60'000 tokens. At the end the average F-measure of the 5 runs is taken as a reference measure.

Token	**POS**	**Event**	**TimeExpr**
turning	VBG	B-*occurrence*	O
out	RP	O	O
to	TO	O	O
be	VB	O	O
another	DT	O	O
very	RB	B-*state*	O
bad	JJ	I-*state*	O
financial	JJ	O	O
week	NN	O	B-*timex3*
for	IN	O	O
Asia	NNP	O	O
.	.	O	O

Table 2: Example illustrating the IOB-model for the features *POS, Event and Time Expressions*

4.1 F-measure

As mentioned before the measures recall, precision and F-measure are used. They are common when evaluating performance of algorithms in computational linguistics. The definitions can easily be explained with the Figure 2 where the sets A, B and C are shown. Recall measures how many of the relevant entities that were found. It is defined as *Relevant items retrieved / All relevant items* = $\frac{B}{A+B}$. Precision is a measure of how many of the retrieved entities are relevant. It is defined as *Relevant items retrieved / All retrieved items* = $\frac{B}{B+C}$. Recall (R) and precision (P) are combined into the F-measure which is the harmonic mean of both numbers:

$$F = \frac{2 * P * R}{P + R}$$

For more information about these measures the reader is referred to (Nugues, 2005).

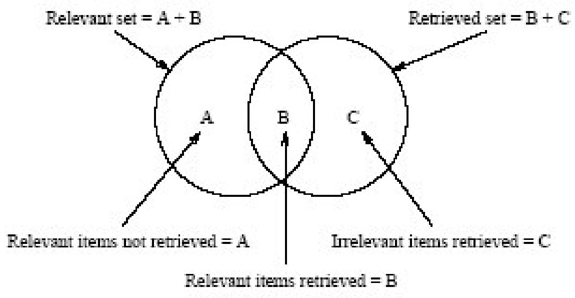

Relevant set = A + B
Retrieved set = B + C
A
B
C
Relevant items not retrieved = A
Irrelevant items retrieved = C
Relevant items retrieved = B

Figure 2: Precision and Recall

5 Experiments

5.1 Time Expressions

Extracting time expressions from texts comes at the first place of our analysis tasks. The only features used for training are the token itself, the POS and of course the time expression. In Table 3 the results for the 5 runs are listed. The average F-measure is 0.791 which is quite high considering that only approximately 2'200 time expressions are contained in each training set. This means that around 3.7% of all tokens of each training set are tagged as time expressions.

Recall	**Precision**	**F-measure**
0.590566	0.894286	0.711364
0.718033	0.914405	0.804408
0.723842	0.917391	0.809204
0.791878	0.921260	0.851683
0.770355	0.783439	0.776842

Table 3: Results for Time Expressions

5.2 Signals

For identifying Signals we even have less information in the training sets (1'800 Signals per set). As we already extracted time expressions these entities are now of use for the next task. Nevertheless we take all features used for a specific task from TimeBank and not from an output of our own detection tasks. This fact holds

throughout this project.
The experiment using only the token itself and the POS results in a F-measure of 0.576. Adding the time expressions to the features improves the F-measure to 0.616. One reason for this moderate result is certainly the small corpus size.

5.3 Events

As noted before TimeML provides 7 different class types for events. Obviously this makes this analysis task very difficult. On this account we evaluate the experiment for two different cases. Once events are recognized without considering the class type *(untyped case)* and once we want to determine the events together with their class type *(typed case)*. Each training set contains approximately 7'300 events with unequally distributed class types.
In Table 4 the F-measure for 7 different experiments are presented. The first 3 rows differ only in the feature set. In the last row the window size parameter is increased while using the same features as in the third row. This experiment is only evaluated for the untyped case because of the long run-time for training. Further the untyped case shows that increasing the window size doesn't improve the performance. The evaluation of the typed case using the features token, POS, TimeExpr and Signal takes around one hour per run on an Intel Mobile Pentium 4 with 1.7 GHz. Thus the real run-time for an experiment, using a 5-fold cross validation, is around 5 hours.
As expected the best results are obtained using the features token, POS, TimeExpr and Signal together with the standard settings in YamCha. These results are only around 3% lower than the results from (Boguraev and Ando, 2005). Though their feature vector representation is much more complex and contains a lot more features, e.g. word uni- and bi-grams based on subject-verb-object and preposition-noun constructions. Compared to our SVM-approach they use a classification framework based on a principle of *empirical risk minimization* called *Robust Risk Minimization* (RRM).
Finally we investigated the performance of event identification with increasing corpus size. In the first step the training set size is only 20% of the corpus and therefore of the same size as the test set. Then the training set size is increased until it reaches 80% of the corpus as in the experiments before. Because of run-time reasons we disclaim the 5-fold cross validation and only evaluate one run per training set size. In Figure 3 and 4 the F-measure is plotted against the percentage of the corpus size for both, the untyped and the typed, case. The irregularity in Figure 4 is probably caused by the fact that we do not use 5-fold cross validation. Apart from this and that the F-measure is higher in the untyped case the two plots illustrate the same logarithmic trend. They also show that the point where the performance begins to stagnate is not yet reached. Therefore a larger corpus can still improve the performance significantly.

	typed	untyped
Token/POS	0.551	0.734
Token/POS/TimeExpr	0.557	0.735
Token/POS/TimeExpr/Signal	0.573	0.754
larger window size	-	0.751

Table 4: Results for Events

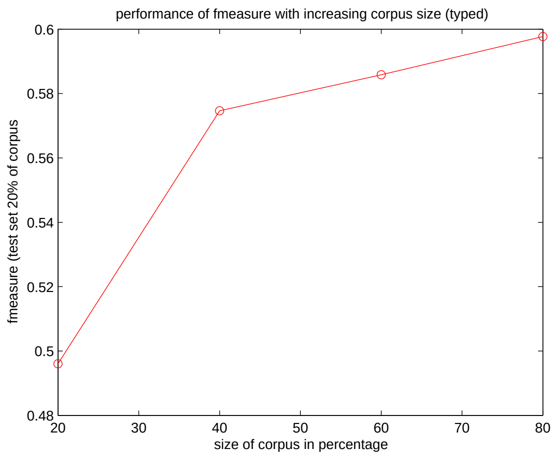


Figure 3: Increasing corpus size (typed case)

5.4 Temporal Relations

Determining temporal relations is even a more complex problem than event recognition. In fact TimeML provides 13 different temporal relation types. To simplify the problem we focus only on six temporal relation types (*before, after, includes, is_included, simultaneous and identity*). Furthermore we only consider temporal relation between events.
We use a similar approach to the one in (Berglund, 2004) that builds a feature row considering two adjacent events. All in all 15 features are used for the temporal relations

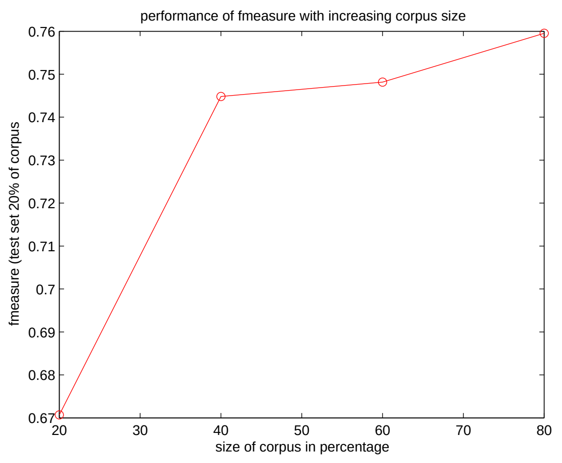


Figure 4: Increasing corpus size (untyped case)

analysis. In the following listing the features are presented:

- 1. - 5. Token/POS/event/tense/aspect of first event
- 6. - 10. Token/POS/event/tense/aspect of second event
- 11. temporal signals between events: [several, none, signal_token]
- 12. distance measured in tokens: [1, 2t3, 4t6, 7t10, gt10]
- 13. distance measured in sentences: [0, 1, . . . , gt4]
- 14. distance measured in punctuation signs: [0, 1, . . . , gt5]
- 15. temporal relation

This feature representation is illustrated on the following example:
The cat ate some cheese. Then, the dog saw the cat and chased it. Cheese is good for you.
Three events (e1:ate, e2:saw and e3:good) and one signal (signal:then) are contained in this example. This gives us three pairs of adjacent events with the following representation:

1. ate VBD OCCURRENCE PAST NONE saw VBD OCCURRENCE PAST NONE THEN 7t10 1 2 BEFORE
2. saw VBD OCCURRENCE PAST NONE chased VBD OCCURRENCE PAST NONE NONE 4t6 0 0 BEFORE
3. chased VBD OCCURRENCE PAST NONE good JJ STATE NONE NONE NONE 4t6 1 1 NONE

If we only consider adjacent events each training set size contains around 6'650 pairs of events where around 1'220 pairs possess a temporal relation. To get more pairs of events *i_before and i_after (i for immediately)* are mapped to the temporal relations *before and after*.
The results of the 5 runs are listed in Table 5 which gives us an average F-measure of 0.195. The five runs show a quite poor performance (especially the recall is very low) and a high variance.

Recall	**Precision**	**F-measure**
0.087227	0.237288	0.127563
0.130159	0.284722	0.178649
0.219355	0.441558	0.293103
0.16	0.360902	0.221709
0.134752	0.183575	0.155419

Table 5: Results for Temporal Relations

6 Conclusions and Open Problems

We showed that SVM is a very powerful algorithm for this kind of text analysis. With a much easier feature representation we could almost reproduce the results from (Boguraev and Ando, 2005).
The main bottleneck could easily been located with the very small corpus size. Another problem is that the entity types are often unequally distributed, e.g. the TimeBank 1.1 contains 4'452 events of type *occurrence* and only 51 of type *perception*. In the future better results could definitively be reached with a much larger corpus.
In the last case of event ordering the results are quite poor. The task of identifying temporal relations is known to be very difficult, especially if it is about domain-independent text. But the results could be easily improved by creating a larger corpus containing more event pairs. This can be realized by considering a larger event window size and not only considering adjacent events. The best strategy is to take the transitive closure to build event pairs. The number of related pairs increases and therefore the training sets provide more training information.
Another approach to improve the results could be realized by tuning the parameters for the SVMs. Although in an experiment for the event recognition task using the 3nd degree of polynomial kernel no improvement could be observed.

7 Acknowledgements

First of all I would like to thank my supervisor Richard Johansson for his patience, support and good advices. Additionally, I would like to thank Pierre Nugues for giving me the opportunity to work on this project.

References

A. Berglund. 2004. Extracting temporal information and ordering events for swedish.

B. Boguraev and R.K. Ando. 2005. Timeml-compliant text analysis for temporal reasoning. *IJCAI-05*, pages 997–1003.

P. Nugues. 2005. *An Introduction to Language Processing with Perl and Prolog.*

R. Sauri, J. Littmann, R. Gaizauskas, A. Setzer, and J. Pustejovsky. 2004. Timeml annotation guidelines, version 1.1.

LUNDS UNIVERSITET

Institutionen för Datavetenskap

http://www.cs.lth.se

Produktion: Jonas Wisbrant • 2006