Språkbehandling och datalingvistik

Projektarbeten 2004

Handledare: Pierre Nugues och Richard Johansson

LUNDS UNIVERSITET

Institutionen för Datavetenskap

http://www.cs.lth.se

Printed in Sweden
Lund 2005

Innehåll

LUNDS UNIVERSITET

Institutionen för Datavetenskap

http://www.cs.lth.se

Identification of rhetorical words in swedish

Petter Bergman, c01pb@efd.lth.se

January 17, 2005

1 Introduction

For some words in Swedish, two distinct usage patterns can be seen. Like, for instance, in the following sentences, the words we shall discuss are written in italics.

1. taket är mörkbrunt *liksom* dom enkla borden
2. jo det skulle *liksom* vara jag det
3. dom är av samma *typ*
4. vi skulle *typ* gå dit och hämta honom

Sentences 1 and 3 examplifies what we would call "normal" use of the words, this use is commonly seen as more "correct" Swedish. Sentences 2 and 4 contain a different use, the word in question is used more as a pause. Note that in sentences 1 and 3, removing the words distort the sentences. In sentences 2 and 4, the removal of the words will leave the meaning of the sentence intact.

From here, the use in sentences 1 and 3 is called "grammatical", and the one in 2 and 4 "rhetorical". Can we classify the words automatically from examining its context in a sentence?

1.1 Part-of-Speech

In sentence 1, "liksom" is a conjunction, in sentence 3 it is an adverb: In sentence 2, "typ" is a noun, in sentence 4 it is an adverb[2].

Using this information we could classify the words, but then we would have to know it's Part-of-Speech.

The task of classifying the words can be seen as a subtask of disambiguating in a Part-of-Speech tagger.

1.2 Spoken vs Written language

Rhetorical words occur almost exclusively in spoken language, and at least grammatical "liksom":s occur almost exclusively in written language. So are we just disambiguating between written and spoken language (or rather, spoken language translated into written language, which has it's own problems)?

2 The Method

To find the use of a word in a text is to find the u which maximizes:

$$P(u|W_1, W_2)$$

$$u \in \{Grammatical, Rhetorical\}$$

because:

$$P(A|B)P(B) = P(B|A)P(A)$$

we can instead maximize:

$$P(u)P(W_1, W_2|u)$$

we pretend that $P(Grammatical) = P(Rhetorical)$ and that words occur independently:

$$\text{argmax} \prod_{(w_1,w_2)\in W_1,W_2} P(w_1, w_2|u)$$

We estimate $P(w_1, w_2|r)$ by counting words from a manually tagged corpus. We would get a lot of zero counts so we use Laplace estimates to cope with the sparse data.

3 Implementation

The implementation consists of three tools written in O'Caml:

collect counts occurences of a word in it's grammatical/rhetorical use from a hand-annotated file.

tag tags occurences of a word using collected data

eval evaluates the output by comparing it to a file hand-annotated with the correct use.

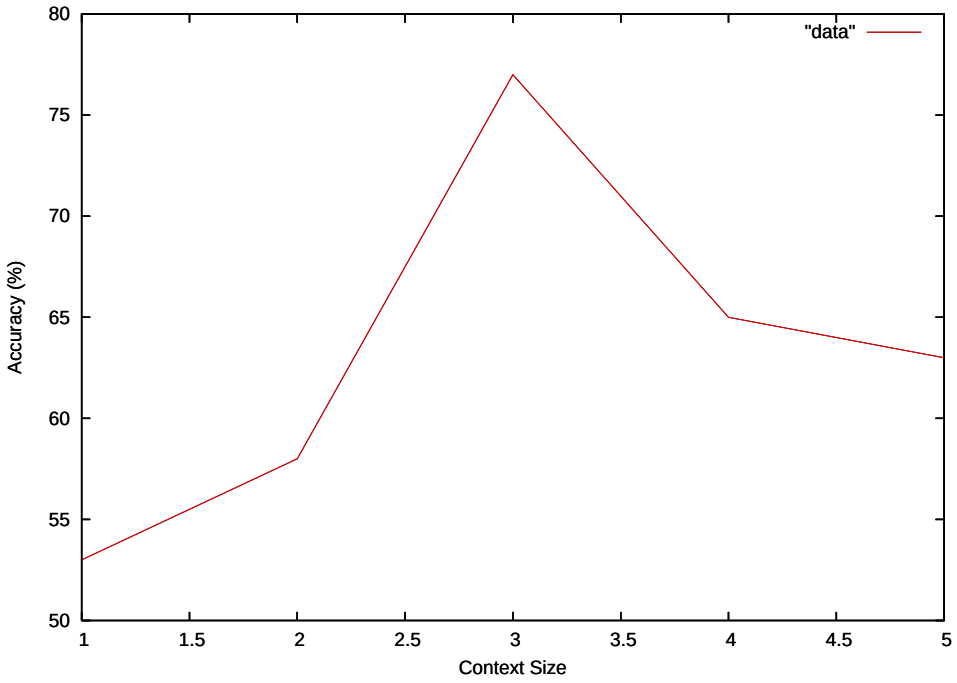


Figure 1: Results of the test-run

4 Results

For evaluation, a training set was compiled from [4] and [3] in such a way that it contained 50 occurences of liksom, hand annotated. Data was then collected from this text, using different context sizes.

In a similiar way, the (not annotated)test set was compiled. The results of running the tagger on the test set for different context sizes is shown in figure 1.

The behaviour is as expected, although slightly better than expected for such sparse data.

5 Final Comments

In spite of the result from the test run there are some concerns.

We make the asumption that "liksom" occurs an equal number of times in the rhetorical use as in the grammatical use, we then proceed to collect a test and training set making this true. What is a balanced corpus in this case? it should be a sample of the body of all swedish, written or spoken. Spoken swedish is is hard to sample, do we mean all utterances ever made in swedish? or just contemporary swedish?

We could look at the POS of the context instead of the words themselves

and reduce the sparse-data problem. But if we know the POS of every word in the text, we also know if our examined word is rhetorical or not.

References

[1] Daniel Marcu and Abdessamad Echihabi, "An Unsupervised Approach to Recognizing Discourse Relations" Information Sciences Institute and Department of Computer Science, University of Southern California

[2] "Nordstedts Svenska Ordbok" Nordstedts Förlag 1990

[3] Allwood J., Björnberg M., Grönqvist L., Ahlsén E., Ottesjö C. "The Spoken Language Corpus at the Linguistics Department" Göteborg University in Forum Qualitative Social Research, vol 1, no 3 - Dec 2000

[4] Philipp Koehn "Europarl: A Multilingual Corpus for Evaluation of Machine Translation" Draft, Unpublished

Statistical Noun Group Detector

Antonio CALZADA
Department of Computer Science
Lund University
dat95jca@hotmail.com

Abstract

Statistical noun group detectors and chunkers provide powerful means of detecting syntactically correlated non-overlapping parts of sentences.

This report describes discoveries made exploring statistical noun group detection based on Support Vector Machines (SVM) applied on data from the CoNLL-2000 shared task.

1 Introduction

Text chunking consists of dividing a text in syntactically correlated parts of words. For example, the sentence *He reckons the current account deficit will narrow to only # 1.8 billion in September* . can be divided as follows:

[NP He] [VP reckons] [NP the current account deficit] [VP will narrow] [PP to] [NP only # 1.8 billion] [PP in] [NP September] .

The shared tasks of CoNLL provide excellent reference material and results.

In the shared task of CoNLL-2000 full phrase part chunking is explored.

In shared task of CoNLL-1999 NP bracketing is explored. This consists in identification of all noun-phrase structures allowing multiple levelled groups where a for example a noun-phrase may be identified as being decomposable into smaller noun-phrases.

Noun group detection. Also known as noun phrase (NP) chunking is a simple and robust alternative to full parsing for segmenting a text into syntactically correlated parts.

While this report specifically explores detection of noun groups, many times the same methods can be applied to other group detection problems like full phrase part detection and identification of names of companies and people in texts.

Because statistical methods and learning algorithms are used instead of for example hand made rules, the implementation can easily be adapted to different languages and types of text.

2 Segmentation and labelling

Segmentation and labelling are two of the most common operations in natural language processing. These two operations are strongly related. While segmentation divides a stream of characters into linguistically meaningful pieces, labelling classifies those pieces.

There are many different ways a text can be segmented, most notably: bulletins, pages, sections, sentences, phrase parts, words and word stems.

Segmentation might be done at more than one level. When classifying news bulletins it might suffice to have two levels of segmentation. First the text would be segmented into bulletins and then into sub-segments of keywords and non-keywords. This would contrast full parsing, where text is segmented into hierarchical structures of unlimited depth.

Labelling is characterized by the set of labels used and their meaning. It may range from sentence identification by enumeration to tagging using an extensive set of part-of-speech (POS) tags.

3 SVM

Support Vector Machines (SVMs) are used for solving classification and regression problems, very much like neural networks.

They are trained using data sets of attributes (features) and corresponding target value (class label). A trained SVM model can then be fed sets of features that it will attempt to classify correctly.

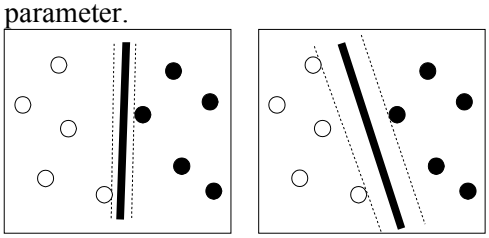
The SVM model training works by mapping the training vectors into a higher dimensional space. During training the SVM engine attempts to maximize the margin between critical values and a separating hyperplane. See (Drawing 1) for an illustration where the dark line represents a projection of the hyperplane dividing the dataset. The thinner dotted lines mark the distance between the plane and the closest data point.

There are a number of configuration parameters that can be tuned for the task at hand, the most common are kernel function, γ and C.

The heart of the SVM engine is a pluggable kernel function controlling the creation of the hyperplane. Some examples are: linear, polynomial, radial basis function (RBF), and sigmoid.

Depending on the type of the function, a number of configuration parameters may be available. The γ parameter is common to most kernel functions.

Since it may not be possible to place the plane so it correctly classifies the training data it is allowed to incorrectly classify training values but at the cost of a penalty that is to be minimized. The severity of this penalty is controlled by the C parameter.



Drawing 1 SVM Classification optimization

4 NP identification using SVM

The basic steps for applying SVM to NP detection are:

- Selecting appropiate features.
- During training:
  - Scale and encode features for train data.
  - Select kernel function and trim parameter.
  - Train SVM model.
- During testing:
  - Scale and encode features for test data.
  - Let SVM model classify test data.
  - Decode classification labels.

Since SVM works with points in a mathematical space the words and tags in the natural language material needs to be encoded into numbers.

SVM like most learning algorithms thrive on information. But it's important that both the training data and the encoding is not biased fooling SVM into identifying patterns that are not applicable to the test data.

5 Test, training and evaluation

The material I used is the same as used at CoNLL-2000 and in turn originally produced by (Ramshaw and Mitchell, 1995).

The corpora contain one word per line and each line contains six fields of which the first three fields are relevant: the word, the part-of-speech tag assigned by the Brill tagger and the correct IOB tag showing phrase par limits.

Words can be inside a NP (I) or outside a NP (O). In the case that one NP immediately follows another NP, the first word in the second base NP receives tag B.

The source corpora of the data is sections of the Wall Street Journal (WSJ), part of the Penn Treebank (Marcus et al., 1993). Sections 15-18, 211727 tokens are used as training material and section 20, 47377 tokens as test material.

Three values are used to measure performance:

- precision, percentage of detected noun phrases that are correct ,
- recall, percentage of noun phrases in the data that were found by the classifier,
- and F-beta, provides a collected measurement of the previous two values evaluated as (2*precision*recall)/ (precision+recall)

Results are measured up against a baseline value provided by a simple unigram tagger (tagging IOB tags instead of POS tags).

6 The Demo Program

The purpose of the demo program is to demonstrate how an arbitrary text provided the user is tagged and NP chunked by the implemented methods.

The demo is implemented as an interactive console program. Once it is started an introductory message and a prompt is shown. The prompt accepts the commands described in (Table 1).

The tagger used is a simple back-off tagger composed of in order of preference: a trigram tagger, a bigram tagger, a unigram tagger, and as a last result defaulting to the NN tag.

Table 1Demo program commands

command	Description
chunk	Chunks the provided text. Tagger and SVM engine has to be loaded.
create_svm [word count]	Creates a SVM engine and trains it using CoNLL-2000 train data. A word count can be provided to limit the size of the corpora used.
create_tagger [word count]	Creates a tagger and populates it with data from the CoNLL-2000 train data. A word count can be provided to limit the size of the corpora used.
evaluate_tagger	Evaluates the currently loaded tagger.
exit	Exits the program.

help [command]	Shows either available commands or information about a command if provided.
history	Provides a list of executed commands.
load_svm	Loads a svm model from provided filename or svm.pkl if none given.
load_tagger	Loads a tagger from provided filename or tagger.pkl if none given.
save_svm [filename]	Saves a svm model to provided filename or svm.pkl if none given.
save_tagger [filename]	Saves a tagger to provided filename or tagger.pkl if none given.
shell [command]	Executes the provided command in a spawned shell.
tag text	Tags the provided text using the loaded tagger. Also saves the output to the file tagged.txt.
test	Creates a tagger and a SVM engine and then tags and chunks a test text.
! [command]	Same as the shell command.

7 Results

Building on the baseline (F-beta = 79.99) implementation by using n-tagger showed showed some improvements that quickly tapered off with tagger complexity.

Using SVM on half the training set with C=64 and γ=64 produced F-beta=80.99 when not using context based features. When adding the previous context based POS tag to the features F-beta improved to 86.60.

8 Conclusion

Although my results are not that great, they show that just adding some context data shows a good improvement. Providing more an better features would give very good results.

9 References

Dimitrios Kokkinakis and Sofie Johansson Kokkinakis. 1999. *A Cascaded Finite-State Parser for Syntactic Analysis of Swedish*, In: "Proceedings of EACL'99", Bergen, Norway.

Taku Kudo and Yuji Matsumoto.2001. *Chunking with Support Vector Machines*, In: "Proceedings of NAACL 2001", Pittsburgh, PA, USA.

Mitchell P. Marcus, Beatrice Santorini, Mary Ann Marcinkiewicz. 19993. *Building a large annotated corpus of English: the Penn Treebank*, Computational Linguistics.

Lance A. Ramshaw and Mitchell P. Marcus. 1995. *Text Chunking Using Transformation-Based Learning*. In: "Proceedings of the Third ACL Workshop on Very Large Corpora", Association for Computational Linguistics.

Requirements Analysis

Stephane Clinchant
ENSEEIHT
Lund University
stephane.clinchant@netcourrier.com

Abstract

This paper presents the attempts to adapt the minimal edit cost algorithm for sentences in order to compare two requirements. These requirements are two short texts strongly related we want to compare. What is similar ? What is new ? are the questions we want to answer.

1 Introduction

One of the first and fundamental activities in a software process is requirement analysis. The system's services constraints and goals are established by consultation with system users and defined in a manner which is understanbled by boh users and developpement staff. Domain understanding , Requirement collection , Classification , Conflict Resolution , Prioritization and requirement validation are main activities in requirement analysis. This project is related to classification: this activity takes the unstructured collection of requirement and organized them into coherent clusters. In market driven software development there is a strong need for support to handle congestion in the requirement enginneereing process. To meet the market demands it is important to have an effective and efficient requirement engineereing process to deal with a numerous flow of incoming requirements.

2 Aim of the project

This project was suggested by Obigo a mobile phone software company. An analyst work on 2 sets of requirements: the old requirements and the new requirements, which could be for example :

> **Old**: SMS messages may contain up to 140 characters
> **New**: SMS messages must contain 240 characters including ISO characters.

A part of the analysts' work is spent on matching pairs of old and new requirements and identifying changes. His first task is to link similar requirements, to detect requirements referring to the same need , functionnality. It is a classification activity to find the structure and the relations between the requirements. The second task is to find the change , the new constraints , the difference between the related requirements. In the example, the important changes here are *may* $\rightarrow$ *must*, 140 $\rightarrow$ 240 and a new constraint: *including ISO characters.*

This project aims at improving requirement analysis and saving time of Obigo's analysts. We focused on sequential sets of requirements and compare the new and the old ones. To streamline analysis , the following process was proposed :

- Use the tool of Johan Natt och Dag, ReqSimile to find the similar requirements
- Find the difference, the change between the pair of requirements :this is the aim of this project.

The language processing project was to implement an algorithm to detect and quantify the changes between two similar requirements. Thus, the analysts would be able to explore quicly the sets of requirements, find relations and detect the changes.

3 Minimun Edit Cost Algorithm and Alignments

The idea to explore in order to compare two short texts was to try to adapt the minimun edit cost algorithm for sentences and provide an alignment to compare them. Alignments and the minimun edit cost algorithm (MCA) are strongly related since the minimun edit cost produce an alignment. A definition of an alignement could be a sequence of operation which

transforms a sequence of symbols (the source) into an other (the target). Operations acts on symbols , can be copy (of symbols), insertion, deletion, substitution and have a cost. Letters are the symbol type for the classical minimun edit cost algorithm. A sequence of symbols is a word and the algorithm measure the distance between two words. Symbols can also be ADN bases in order to compare genomes. In this project symbols are words in the same language. But what is important to keep in mind is that an alignment is a way of matching elements of the source with the target. Here is an example of a possible alignment for the previous requirements

> [copy(sms),copy(messages),subs(may,must),
> del(up),del(to),subs(140,240),
> copy(characters),ins(including),ins(iso),
> ins(characters)]

The MCA is part of dynamic programming and gives alignments with the minimal cost. It is significant to underline the fact that there exists most of the time several possible alignments for one source and one target.The MCA equations stand for the possibilities a sequence source can be transformed into the sequence target. Basically there are 3 ways to do this. Suppose we have two sequences $[a_1, \ldots, a_n]$ and $[b_1, \ldots, b_n]$. First we can suppress a_n and transform $[a_1, \ldots, a_{n-1}]$ into $[b_1, \ldots, b_n]$. A second way to proceed is : if $a_n = b_n$ or if we substitute a_n by b_n we only need to transform $[a_1, \ldots, a_{n-1}]$ into $[b_1, \ldots, b_{n-1}]$. Lastly , if we know how to transform $[a_1, \ldots, a_n]$ into $[b_1, \ldots, b_{n-1}]$, we just need to insert b_n.

More formally, if E is the edit cost of two sequence and $c(.)$ the cost of an operation The minimun edit cost algorithm is defined by these equations:

$$\mathrm{E}([a_1, \ , a_n], [b_1, \ , b_n]) =$$

$$\min \begin{cases} \mathrm{E}([a_1, \ , a_{n-1}], [b_1, \ , b_n]) + c(\mathrm{del}(a_n)) \\ \mathrm{E}([a_1, \ , a_{n-1}], [b_1, \ , b_{n-1}]) + c(\mathrm{subs}(a_n, b_n)) \\ \mathrm{E}([a_1, \ , a_{n-1}], [b_1, \ , b_{n-1}]) + c(\mathrm{ins}(b_n)) \end{cases}$$

$$\mathrm{E}([a_1, \ , a_k], [\,]) = k \quad \mathrm{E}([\,], [b_1, \ , b_k]) = k$$

The costs of operation in the classical algorithm are

$$cost(ins) = 1 \quad cost(del) = 1$$
$$cost(copy) = 0 \quad cost(subs) = 2$$

To get the alignment , a matrix is filled with the cost of intermediary transformed sequences. Then , an alignment is obtained by a path in this matrix.

4 Hypothesis

To investigate the adaptation of MCA , I first limit my approach by not doing any linguistic process but just to use and apply the the basic algorithm to observe basic results. For example, we could say that the insertion of the word "and" costs zero because it does not bring significant information in the new requirement.So I did not try to tune this algorithm linguistically. One of the most important observation, is that there are several possible alignments.When I developped my algorithm, my goal was to construct all the possible alignments and to choose the best one with an heuristic.There is one simple reason for multiple alignment. In the example of requirements, the word "characters" occurs two times in the new requirement. So the "character" from the old requirement can be linked in two different places in the new requirement.

5 Factorization

Another reason is if we use the classical MCA we will obtain similar alignments which leads to the same matching of words. If x and y are two symbols, inserting x then deleting y is the same than deleting y then inserting x which is the same than substituting y by x.

$$ins(x), del(y) \sim del(y), ins(x) \sim subs(y, x)$$

We get the same matching of words but we are only intereted in a reduced form, the shortest alignment among its similarity class. So I decided to suppress the substitution operation from the algorithm so that it produces sequences of operations which are only insertion, deletion or copy. The next step was to factorize , to reduce the alignment in order to find the set of possible and interesting alignment. $copy(x)$, $subs(y, w)$ is the same than $copy(x)$, $ins(w)$, $del(y)$ but the first form is the best to keep. If we want to reduce an alignement, we have to consider more complicated similarity forms. As I suppressed the substitution operation, I can obtain alignement whose parts are such that:

$$ins(x_1), del(y_1), ins(x_2), del(y_2), \ \ldots \ , ins(x_k), del(y_k)$$

I need to factorize it and the problem was there was several way to do it. For example if we

have:

$$ins(x_1), del(y_1), ins(x_2), del(y_2)$$

This can be factorized in two ways:

$$subs(y_1, x_1), subs(y_2, x_2)$$

$$ins(x_1), subs(y_1, x_2), del(y_2$$

and we can not know which form is the best. If F is the Factorization function, the simple case are

$$F(Copy :: tail) = Copy :: F(tail)$$

$$F(Ins :: Ins :: tail = Ins :: F(Ins :: tail)$$

$$F(Del :: Del :: tail = Del :: F(Del :: tail)$$

$$F(Del :: Ins :: a :: tail = Subs :: F(a :: q)$$

where a = Copy or Ins

$$F(Ins :: Del :: a :: tail = Subs :: F(a :: q)$$

where a = Del or Copy
The complicated case is whenever there is a sequence of ins, del or a sequence of del,ins. Wether the number of couple ins,del be pair or not , there is always two possibilities to understand and to factorize this sequence:

- Take the first pair of ins,del (or del,ins) and group the other pairs after. If the number of sequence is unpair there is one operation left.
- Take the first operation and form the first pair with the second and the thrid operation. If the number of sequence is pair there is one operation left.

6 Results

So I developped a component for reducing alignment and I obtained 18 factorized alignment for the requirement example. I should have worked on heuristic at this time but I developped a simple GUI to display alignment and the matching of words with color codes. I had one idea about a heuristic at this time: alignments wih insertions at the end are more likely to be better than the others because people tend to add new elements at the end of a requirement when they rewrite them.

Then, I ran my algorithm on real requirements from Obigo which were much longer and I obtained something like 8 thousands alignments ! It was a exponential and combinatorial problem. To face this, I had to think to new ways to proceed:

- The first idea is to "to divide and conquer" and split the requirement in sentences and align sentences first and then align words.
- Drop the idea of finding the "best" alignment among the set of all possible alignment but just take one alignment with an heuristic.

I only had time to explore the second idea: my heuristic was to favorize insertion at the end. I had also a new idea : highlight the new words in the new requirement. New words in the requirement show the analyst which part of the requirement has been changed and need to be read. All the copy operation in the alignment show what is identical in the requirements.

7 Conclusion

Aligning two sentences is a very difficullt task if we do not take into acccount linguistic information. In translation where they also align sentences they exploit the structure of the sentence and linguistic information to find an alignment. But what the analyst is interested in , is to see what part of the requirement are identical and what is new. Taking one alignment is a simple and effective solution but there is work to do on how to display graphically the difference and the similarity of two text so that it is easy to see for the human eye.

8 Acknowledgements

I would like to thank Pierre Nugues, Johann Natt och Dag from Departement of Computer Science at Lund Universoty and Sven Olof Karlsson from Obigo for their help and advices during this project.

References

K.Yamada C.Goutte and E.Gaussier. Aligning words using matrix factorisation. *Xerox Research Centre Europe.*

C.Brockett C.Quirk and W.Dolan. Monolingual machine translation for paraphrase generation. *Natural Language Processing group Microsoft Research.*

S.Brinkkemper J. Natt och Dag, V.Gervasi and B. Regnell. Speeding up requirements management in a product software company.

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

H.Garcia-Molina S.Chawathe, A.Rajaraman and J.Widom. Change detection in hierarchically structured information. *Departement of Computer SCience Stanford University.*

Word sense disambiguation using WordNet and the Lesk algorithm

Jonas EKEDAHL
Engineering Physics, Lund Univ.
Tunav. 39 H537, 223 63 Lund, Sweden
f99je@efd.lth.se

Koraljka GOLUB
KnowLib, Dept. of IT, Lund Univ.
P.O. Box 118, 221 00 Lund, Sweden
koraljka.golub@it.lth.se

Abstract

Word sense disambiguation is the process of automatically clarifying the meaning of a word in its context. It has drawn much interest in the last decade and much improved results are being obtained.

In this paper we take the so-called Lesk approach. In our case, definitions of the senses of the words to be disambiguated, as well as of the ten surrounding nouns, adjectives and verbs, are derived and enriched using the WordNet lexical database.

Two possible implications of this project could be that the results are dependent on the characteristics of a test document and on the characteristics of glosses, which needs to be further investigated. The average precision performed worse (0.45) than baseline precision (0.60) which was based on always selecting the most frequent sense. However, the presented approach has several limitations: a small sample, and a big number of fine senses in WordNet, many of which are not that distinguishable from each other. The future work would include experimenting with different variations of the approach.

1 Introduction

Word sense disambiguation is the process of automatically clarifying the meaning of a word in its context. For example, the word *contact* can have nine different senses as a noun, and two different senses as a verb.

Word sense disambiguation has drawn much interest in the last decade and much improved results are being obtained (see, for example, (Senseval)). It can be important for a variety of applications, such as information retrieval or automated classification (for an example of the latter, see Jones, Cunliffe, Tudhope 2004).

Different approaches to word sense disambiguation have been taken. Many are based on different statistical techniques. Some require corpora that are tagged for senses and others employ unsupervised learning. In this paper we take the so-called Lesk approach (Lesk 1986), which involves looking for overlap between the words in given definitions with words from the text surrounding the word to be disambiguated. In our case, definitions of the senses of the words to be disambiguated, as well as of the ten surrounding nouns, adjectives and verbs, are derived and enriched using the WordNet lexical database (WordNet). The sense definition chosen as correct is the one that has the largest number of words in common with the definitions of the surrounding words. A version of Lesk algorithm in combination with WordNet has recently been reported for achieving good word sense disambiguation results (Ramakrishnan, Prithviraj, Bhattacharyya 2004).

In this paper we conduct a pilot experiment, which is a part of a larger project that employs word sense disambiguation for improving accuracy of automated classification.

In the following chapter (2 Methodology) the approach is described in detail. Results are presented and the third chapter (3 Results), and in the last chapter conclusions are given and the future work is suggested.

2 Methodology

2.1. Introduction

In the paper a pilot experiment is conducted, that is a part of a larger project in which this word sense disambiguation approach would be applied for improving accuracy of automated classification.

The Lesk algorithm has first been implemented in its simple form by M. Lesk (1986). It is based on the assumptions that when two words are used in close proximity in a sentence, they must be talking of a related topic and, if one sense can be used by each of the two words to refer to the same topic, then their dictionary definitions must use some common words (Banerjee 2002, p 1). This approach involves looking for overlap between the words in dictionary definitions with words from the text surrounding the word to be disambiguated. The problem of this approach is that dictionary definitions often do not have enough words for this algorithm to work well, which can be overcome by using the WordNet lexical database (WordNet) (ibid.), because it contains different types of relationships between words, such as, for example, syononymy and hyper/hyponymy.

2.2. Creation of glosses from WordNet

In the research conducted by G. Ramakrishnan, B. Prithviraj and P. Bhattacharyya (2004), different types of relationships in WordNet have been experimented with. It showed that the best results are obtained when concatenating the descriptions of word senses with the glosses of its first- and second-levels hypernyms (ibid., p. 218). We adopted their approach. For example, the word *contact* in WordNet has nine senses for the noun, and two senses for the verb:

The noun *contact* has 9 senses in WordNet:

1. contact -- (close interaction; "they kept in daily contact"; "they claimed that they had been in contact with extraterrestrial beings")
2. contact -- (the state or condition of touching or of being in immediate proximity; "litmus paper turns red on contact with an acid")
3. contact -- (the act of touching physically; "her fingers came in contact with the light switch")
4. contact, impinging, striking -- (the physical coming together of two or more things; "contact with the pier scraped paint from the hull")
5. contact, middleman -- (a person who is in a position to give you special assistance; "he used his business contacts to get an introduction to the governor")
6. liaison, link, contact, inter-group communication -- (a channel for communication between groups; "he provided a liaison with the guerrillas")
7. contact, tangency -- ((electronics) a junction where things (as two electrical conductors) touch or are in physical contact; "they forget to solder the contacts")
8. contact, touch -- (a communicative interaction; "the pilot made contact with the base"; "he got in touch with his colleagues")
9. contact, contact lens -- (a thin curved glass or plastic lens designed to fit over

the cornea in order to correct vision or to deliver medication)

The verb *contact* has 2 senses in WordNet:

1. reach, get through, get hold of, contact -- (be in or establish communication with; "Our advertisements reach millions"; "He never contacted his children after he emigrated to Australia")
2. touch, adjoin, meet, contact -- (be in direct physical contact with; make contact; "The two buildings touch"; "Their hands touched"; "The wire must not contact the metal cover"; "The surfaces contact at this point")

For each sense, we take the description given in the brackets, e.g. for the seventh noun sense it is:
(electronics) a junction where things (as two electrical conductors) touch or are in physical contact; "they forget to solder the contacts."

Then we extract two nearest hypernym levels of the word. The resulting gloss for the seventh sense of the noun *contact* would be:

contact, tangency --
((electronics) a junction where things (as t
wo electrical conductors) touch or are in p
hysical contact; "they forget to solder the c
ontacts")
=> junction, conjunction --
(something that joins or connects)
=> connection, connexion, connect
or, connecter, connective --
(an instrumentality that connects; "he sold
ered the connection"; "he didn't have the ri
ght connector between the amplifier and th
e speakers")

Words in the form *bank_building* have been converted into their components, i.e. in this example into *bank building* for easier later comparison.

Finally, while comparing, all words containing three characters and less are left out. This was done in order to leave out frequent words such as articles or pronouns; when there were more than one occurrences of a word, only one was retained. The final gloss for the seventh sense of the word contact would be:

amplifier between conductors conjunction connecter connection connective connector connects connexion contact contacts didn't electrical electronics forget have instrumentality joins junction physical right solder soldered something speakers tangency that they things touch where

The glosses were prepared using Prolog, since WordNet is available in Prolog (Obtaining WordNet).

2.3. Pre-processing the documents

Fifteen documents were selected and downloaded from the World Wide Web. They had to be prepared for the algorithm. First, they were converted into .txt format. Then they were pre-processed into Penn Treebank (Penn Treebank project) tokens using a sed Unix script (Tokenizer.sed). The part-of-speech tagger was MXPOST (MXPOST). Finally, regular expressions were used to put one word per line.

2.4. Comparing for overlapping words

From the pre-processed document, words to be disambiguated were extracted, together with senses of surrounding words. The surrounding words were simply five nouns or adjectives or verbs preceding the word to be disambiguated, and five nouns or adjectives or verbs following it. If a noun/adjective/verb was not in the WordNet, the next closest one was chosen.

Every sense of the word to be disambiguated was compared to each sense of the surrounding words. A number of combinations was derived

and scores were assigned to them, based on the number of the overlapping words. For example, if a word to be disambiguated had two senses, and it was surrounded by two words, one having three different senses, and the other having two different senses, the number of derived combinations was 12, out of which six were for the first sense of the word to be disambiguated, and the other six were for the second sense of the word to be disambiguated. The sense chosen was the one in which group of six there was the combination with the highest score out of all the 12 combinations.

The Lesk algorithm itself was implemented in Prolog.

2.5. Sample

Three words to be disambiguated have been selected: bank, contact, and m/Mercury. Although all of these words have more than two senses, the aim of this pilot experiment was to disambiguate between the two major senses:

bank:

1) depository financial institution (two documents in the sample)
2) sloping land, especially the slope beside a body of water (three documents in the sample)

contact:

1) close interaction between people (two documents in the sample)
2) a junction where things (as two electrical conductors) touch or are in physical contact (three documents in the sample)

m/Mercury:

1) mercury: Hg, metallic element (three documents in the sample)
2) Mercury: the planet. (two documents in the sample)

For each word five documents have been manually selected, out of which two of them had one main meaning, and three another.

3. Results

On our small sample, the average precision performed worse (0.45) than baseline precision (0.60) which was based on always selecting the most frequent sense. However, this result should not be taken for granted, since the sample of three words and 15 documents is too small for any trustworthy results.

Instead, we could use some qualitative analysis:

1) The word bank has 18 senses in WordNet. The precision for all the five documents was relatively bad: 0.25, 0.16, 0.27, 0.30, and 0.5. In all the documents the often assigned sense was that of a piggybank, which might have to do with the fact that its gloss contains a lot of frequent words, such as usually, with, that, from, some.
2) The word contact has 11 senses listed in WordNet. The precision for the five documents was the following: 0.08, 1, 0.6, 0.625, and 0.92. This good result is partly due to the fact that we merged together two rather closely related senses, that of contact as communicative interaction, and that of contact as close human interaction. We were able to do this since the main aim of the experiment was to distinguish

between two totally unrelated senses of contact (see 2.5). While in one example we obtained 23 correct senses out of 25 occurrences, in another only 3 out of 38 were correctly assigned and in this case the extracted senses were not related to the topic of electrical contact.

3) The word m/Mercury has four senses listed in WordNet. The precision for the five documents was the following: 0.82, 0.5, 0.66, 0, and 0.05. The three first numbers are quite good results and all refer to discovering the sense of mercury as a metallic element. Not-so-good results in one of the other two documents is due to the fact that the document was discussing the temperature of the planet of Mercury, which produces the third sense of the word mercury in WordNet, about temperature.

4. Conclusion

Two possible implications of this project could be that the results are dependent on the characteristics of a test document and on the characteristics of glosses, which needs to be further investigated. However, the presented approach has several limitations: a small sample, and a big number of fine senses in WordNet, many of which are not that distinguishable from each other.

In order to determine which solution is best, the future work would include conducting experiments with:

- WordNet preparation and document pre-processing (create a collection-specific stop-word list, apply stemming, do part-of-speech tagging on WordNet glosses, exclude examples from glosses which are in quotation marks, replace the ten-surrounding-word frame with a paragraph/sentence frame; experiment with different combinations of WordNet relations);
- modify algorithm (the role of *tfidf* in precision, taking into account the number of words per gloss, experiment with different similarity measures); and
- utilize WordNet Domains (Domain Driven Disambiguation), a file that contains synsets annotated by domain labels, such as Medicine, Architecture and Sport.

References

Desire : Development of a European Service for Information on Research and Education. http://www.desire.org/.

Domain Driven Disambiguation. http://wndomains.itc.it/download.html

Ganesh Ramakrishnan, B. Prithviraj, Pushpak Bhattacharyya. A Gloss Centered Algorithm for Word Sense Disambiguation. Proceedings of the ACL SENSEVAL 2004, Barcelona, Spain. P. 217-221.

Jones I., Cunliffe D., Tudhope D. 2004. Natural Language Processing and Knowledge Organization Systems as an aid to Retrieval. Proceedings 8th International Society of Knowledge Organization Conference (ISKO 2004), UCL London. (Ed: Ia C. McIlwaine), Advanced in knowledge Organization, 9, Ergon Verlag. P. 351-356.

Lesk, Michael. 1986. Automatic sense disambiguation: How to tell a pine cone from an ice cream cone. In Proceedings of the 1986 SIGDOC Conference, pages 24−26, New York. Association for Computing Machinery.

MXPOST : Maximum Entropy Part-Of-Speech Tagger, and MXPARSE: (local) Maximum Entropy Parser. http://www.cis.upenn.edu/~adwait/pentools.html#Tools

Obtaining WordNet. http://www.cogsci.princeton.edu/~wn/obtain.shtml

The Penn Treebank project. http://www.cis.upenn.edu/~treebank/

Satanjeev Banerjee. 2002. Adapting the Lesk algorithm for Word Sense Disambiguation to WordNet. Master's thesis. Dept. of Computer Science, University of Minnesota, USA. http://www.d.umn.edu/~tpederse/Pubs/banerjee.pdf

Senseval : evaluation exercises for Word Sense Disambiguation. http://www.senseval.org/

Tokenizer.sed. http://www.cis.upenn.edu/~treebank/tokenizer.sed

WordNet : a lexical database for the English language. http://www.cogsci.princeton.edu/~wn/

HMS2005: Predictive text entry using bigrams

Myrtille Dedianne and Robert Nilsson

17th January 2005

Abstract

Nowadays, hundred millions of SMS are sent everyday all around the word and become common in our everyday-life. Then, the efficiency of text entry method in mobile phones is more and more important. As a previous team project already worked on, in order to improve it by using bigram prediction, we decided to continue their work and improve it. In this paper, we describe the system, which is called HMS[3], and how we improved it. This improved version will be called HMS2005. Firstly, the code was reviewed to make it work in English or in whatever language. Secondly, the code was reviewed to make it work quicker and usable. Finally, we added a key, which stops the prediction and falls back to T9 when typing. For the tests, we involved 7 international persons and measured the time and the number of key pressed needed to entry a text. We showed that keys pressed needed were reduced of 20%, but time-consuming was increased of 15%. However, we noticed a difference between people who were or not trained by the new system, which can false the final results. This could be measure in good conditions of a real experience.

1 Introduction

To enter SMS with our mobile phones, we use to use methods we are offered, like T9. But these are not perfect yet and can be optimized to be more usable, more flexible, more efficient, easier to learn, quicker type a SMS message, with less stress etc.

We chose to work on this subject in order to improve the way to write SMS messages. A previous team of 3 Swedish students (Hasselgren, Montnemery, Svensson) have had already worked on it in this course and their system was named HMS [3]. We studied how they built it and how to improve it. Our improved version is called HMS2005 and can be found as an applet at this address: `http://www.orbstation.com/hms2005`.

After, we analysed the different methods already existing and began to code.

2 Evolution of different methods

With a keyboard of 12 keys, we can measure the efficiency of different methods using the number of keystrokes per character or KSPC[4] and the time-consuming to entry the text.

Since the beginning of SMS messages, several predictive text entry methods were developed, in order to improve the efficiency of the multi-press method. In this method, the user has to press once a key to type the first letter of the key, twice for the second, three times for the third et cetera. It is still used but requires more than one of keystroke per character and takes time. We will present 3 other predictive text entry methods.

2.1 T9, by Tegic

This method is a single-press method using unigrams. The user presses once key per character and the program matches the sequence to words in a dictionary. In many cases, only one word is possible given the sequence, otherwise, a list with other possibilities is suggested and the user chooses. The KSPC is then reduced roughly to 1 and is less time-consuming, whereas the beginning of using it is quite disturbing. Many mobile phones use this method nowadays.

But other implementations are iTAP by Motorola[2] and eZiText by Zi Corporation[1], which suggest the next word you intend to type.

2.2 HMS (for Swedish), Lund Institute of Technology, Sweden

This method is a single-press method using unigrams and bigrams, i.e. two consecutive words[3]. It was developed by 3 students of LTH, Lund (Sweden), and uses context. Typing a sequence of keys, the system considers the previous constant word typed, matches it in the dictionary of bigrams (which gives the most frequent words which can follow this word), and gives in real-time the entire word it could be. In this implementation, the bigrams are always prioritized over the unigrams. Available for Swedish, this method reduced KSPC of 7% on SMS messages and 13% on News[3].

2.3 Other methods

There are other methods, like LetterWise[5] or Less-tap[6].

LetterWise is a system that does not use a stored dictionary of words, but a small database of prefix information to disambiguate user keystrokes (Eatoni Corporation, 2003). Its published KSPC is 1.1500[5].

Less-tap method uses a remapped keyboard as a complement for single-press or multi-press methods. Since the keyboard is built on the alphabetical order, it does not take in account the frequency of the most common characters used in different languages. For example in English, "e" is the most common character used, but is in the second place on his key and shares it with "d" and "f". The keyboard could be remapped, mixing all characters and their order, to reduce the KSPC required to 1.4412[6]. However, this proposal could be difficult to be accepted by people who use to use the actual keyboard.

3 Dictionary and corpus

3.1 Dictionary compilations

Our first aim was to make it work in English, and independent of the language. To achieve this it was important that the data files were stored in an internationalizable and platform independent way. Hence we, quite naturally, chose to use Unicode. After that we collected a dictionary of English unigrams. We could have chosen between 2 dictionaries available on the Oxford's documents which are publically available. The first was small (254 kb, about 27000 words) and contained most common abbreviations, places and names, but did not contain all the inflected words. The second was big (3 Mb, 10 millions words), and was a mixed file of all Moby's dictionaries available, with all inflected words, abbreviations, places and names. The first was too small to be used, not enough complete, and the second was too big, requires too much memory and given too many not common words. So we decided to use one between both as a compromise. We found a dictionary on the web (UK English wordlist v1.01, from the website of Brian Kelk, Cambridge, UK: `http://www.bckelk.uklinux.net/`) containing inflected forms, and we combined it with the small one, containing most common abbreviations, names and places. The final dictionary is now 686 kb for 67 485 words.

3.2 Corpus collection and statistics calculation

A suitable corpus for text entry on mobile phones should contain mostly everyday English. However, must corpora are compiled from either news or literature and the language use from these two sources can differ quite much from everyday use. Therefore we decided to collect our own corpus. After some contemplation we decided that Usenet contains large volumes of text which quite close to everyday use. The problem with Usenet is that it also contains a large amount of, for us, unwanted content such as spam and binaries. However, certain news groups are more likely to contain usable text than others and hence we decided to limit our sampling to 118 subgroups of *alt.politics*, *alt.society* and *soc*. From these groups we collected 57 181 messages over a period of nineteen days. This was then compiled into a corpus of roughly ten million words.

First the uni- and bi-gram statistics were calculated from the corpus at runtime. Obviously this slowed down the application considerably as it could take over 2 minutes to compile all statistics with all n-grams present, see section 3.3. In order to avoid this lag when the application started it was decided to precompute all statistics. This was done through the use of a Python script which read in the normalized corpus and created one data file for

each desired n-gram, such as uni-, bi- and so forth. In our case we decided to only use uni- and bi-grams as the frequency of trigrams and higher was too low to yield any noticable effect. The application and its support scripts are however designed in such a way that adding support for higher n-grams is easy to do, see section 5.1.

3.3 Memory considerations

We noticed early on that this application requires a lot of memory. As a matter of fact in our initial revisions of the dictionary and statistics files it required up to 300 MB of RAM. It is important to note here, also, that the memory requirements are largely due to the fact that the data structures are written more with the aim of being easy to understand and maintain rather than to optimize memory consumption. Furthermore Java in itself creates a fairly large overhead when it comes to memory usage. However, we decided that we needed to reduce the amount of data used in the application. This was done by both limiting the size of the dictionary to only include more common words, less pronouns and so fourth. Furthermore a frequency cut-off was applied to both the unigrams and bigrams. Entities with a frequency of xx and yy respectively were removed. This measure ensured that the application consumed less than the 96 MB limit which is standard for Java applets.

Naturally the reduction of the dictionary and the cut-off reduces the accurary of both the uni- and bi-gram prediction but it is our distinct impression that it does not degrade the performance of the application by a great deal. One would of course have to conduct more thourough investigations of this matter to draw a more firm conclusion. Furthermore we believe that the fact that the application can run without problems both as an applet and from the command line is more important than the accuracy gained from lowering the cut-offs and increasing the dictionary size.

4 Combining bigram statistics, prediction and T9

First, we made a system using bi-gram statistics, prediction and unigram statistics at the same time, for each key pressed. The bigrams were obviously prioritized over the unigrams. But a problem was high-lighted: many times when typing a short word, longer words were given instead of having words with only the length of the number of key pressed, because of the prediction. For example, typing "of" after "no", the words given before "of" were "next", "need", "means", "news", "mention", "new", "mercy" and "official". Itincreased a lot the number of KSPC, to go down in the list in order to select it, whereas using T9, "no" appears first in the list. Then we decided to add a key, the "yes" key, which allows stopping the prediction and restricts the length of words given to the number of key already pressed. In the previous example, pressing this key would have reduced the list of words given to "me", "ne" and "of", because we pressed only 2 keys.

This kind of key is not natural at the beginning to using it, but permits to combine advantages of both HMS and T9 methods, i.e. prediction with bigrams and statistics of unigrams without prediction.

Moreover, we added two checkboxes, which are "prediction" and "context" and work in real-time. With them, the system is flexible and the user can choose to use the different advantages of HMS. In that sense, we can differentiate the 2 advantages of HMS compare to T9. With both check-off, the system works like T9. It is useful, then, to be able to choose the right method for the right word. The user can switch from one to the other when typing, to take the best advantage of the different methods (for example, we will choose "context-on" and "prediction-off" to use T9 with context, or choose "context-on" and "prediction-on" to write a long sentence in a good language, or choose "context-off" and "prediction-off" for short words, et cetera).

Finally, we added a functionality usually well appreciated by the users: the ability to learn. That means that each time a word is selected, its frequency is increased by 1. Then, more and more, the most common words used by the user become the first in the list and it is easier and quicker for him, decreases the KSPC.

However, as the system is settled and reloaded each time, it is not possible to save the scores of each user.

5 Implementation

As mentioned earlier one of our goals was to enable the application to be as flexible as possible when it comes both the depth of n-grams used and to have structures that are language independent. Of these the latter is the largest importance if this is do be deployed in real use. This is achieved by making the data structure in the application abstract enough to allow for different key pad encodings without having to rewrite any code. How this is done is discussed in section 5.1. One prerequisite for the data structure to work in this manner is to be able to translate a press on a button into a set of characters, e.g. the button labeled "2" on the keypad, see *figure 1*, represents the characters "a", "b", "c" and "2". By reversing this, e.g. saying that "a" maps onto the button "2", we can encode a word as a series of key presses given a certain encoding scheme, or key map. By stating that the buttons each have an index and that the first button is "1" with index 0. By using this the word "hello" would in English be encoded as the following sequence 3-2-4-4-5.

Another important aspect of any application is its user interface. In our case There was not really much choice as to how the application should look as its task was to mimic the front, or user interface, of a mobile phone and there is already a de facto standard for this. Hence the GUI is laid out with a text area above a keypad which the user can press with the mouse. One addition is the list of proposed words to the right of the text area and keypad. Furthermore we added two checkboxes to turn word prediction and context awareness on and off. The user interface of HMS2005 can be seen in *figure 1*.

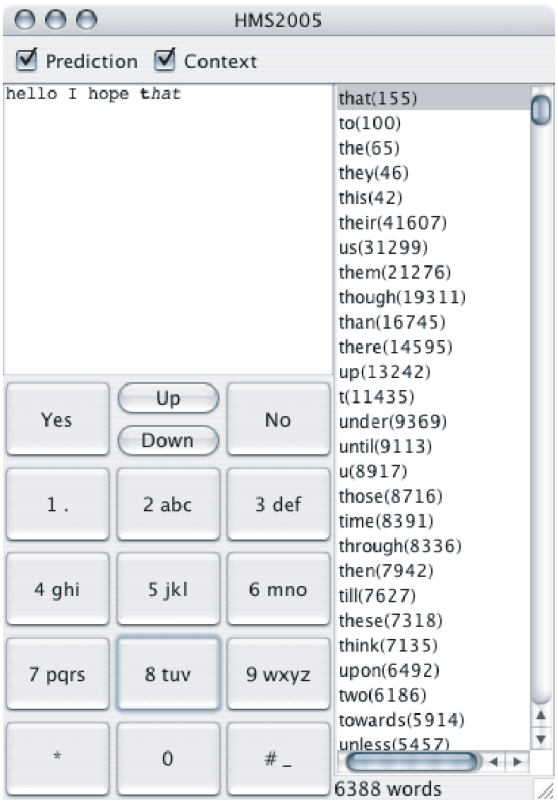


Figure 1: The user interface of HMS2005

5.1 Data structure

The application stores its data about words and their probabilities (unigrams) as well as bigrams in a common structure, a word tree. In the tree the nodes are connected by arcs which represents one of the buttons labeled 1 through 9, see *figure 1*. Each node also can contain a list of words which this node is said to represent. Furthermore the tree can optionally contain a link to a different word tree for each word, more on this later. And last, and quite naturally a node can have up to nine references to subtrees. Hence if we take the example with "hello" from above we would, starting from the root node, take the fourth arc followed third and so forth. After the sequence has been followed we would have reached the node containing the word "hello". With this solution word prediction just becomes the task of compiling a list of words from the subtree of the current node as well as the list of words contained in the node.

We can use this basic structure to include bigram information as well by, as hinted earlier, for each word in a node also link to another word tree, the bigram tree. When a word has been accepted by the user the bigram tree corresponding to that tree is stored. As the user types the next word the application traverses both trees with the same input and when the prediction lists are generated the list from the bigram tree is prepended to the unigram list. Of course this method could be repeated an arbitrary number of times to provide which ever n-gram depth desired. A schematic view of the data structure can be seen in *figure 2*.

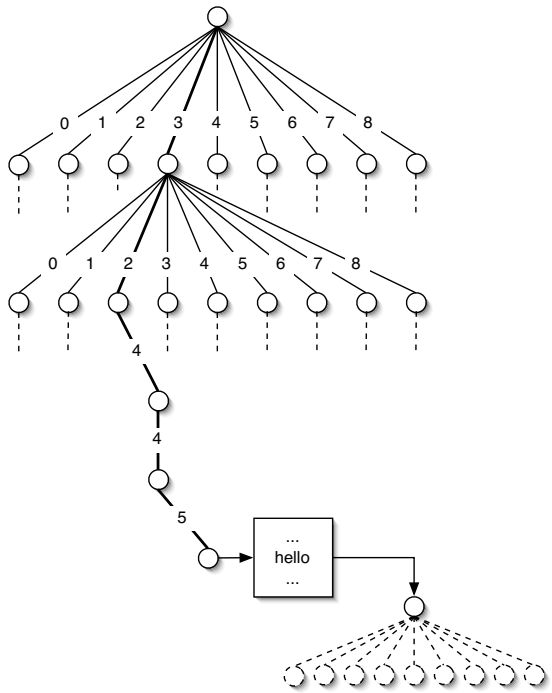


Figure 2: The data structure used in HMS2005 with the path leading to "hello" highlighted

The data for the application is stored as plain text lists with one entity per line and where each line consists of either just a word, for the dictionary, or the frequency followed by one or more words for uni- and bi-grams. When the application starts these are added to the tree in a such a way that only words in the dictionary are allowed, this to avoid unwanted words from the corpus to enter the tree. Of course this approach slow the loading process if there are many unwanted words but as the loading time was acceptable after cut-offs were applied, see section 3.3, we decided to keep this approach as it simplified our work.

The time consumption of this data structure depends largely of the length of the words stored. Of course going from one node to another can be done in constant time hence going from the root node to a certain node and finding a word within that node can be done in $O(n+m)$ time where n is the length of the word and m is the number of words within the destination node. When building a word prediction list the application has to first visit each node in the subtree and for each node a word list has to be compiled consisting of the words in that node concatenated with the words of that node's subtree. If we assume that adding a word to a list is $O(1)$ this phase will run in $O(nm)$ time where n is the number of nodes and m the number of words. The next phase will be sorting the list according to some criterion, usually frequency but it could also be alphabetically, and depending on the sorting algorithm used this can vary greatly. If we employ an fairly efficient sorting algorithm this phase can be done in $O(n \log n)$ time which would yield a total running time of $O(nm + n \log n)$. This can be optimized further of course, for example storing the words pre-sorted within the structure, but as the aim of this implementation is clarity and demonstration rather than efficiency and speed this must be considered adequate.

6 Evaluation

To evaluate our new method, we chose 2 factors of measurement: number of KSPC and time to entry the text.

We did first a "different participants for the same sentence" evaluation, i.e. each participant entries the same sentence in order to compare between participants, and secondly a "same participant for different sentences" evaluation, i.e. each participant entries 2 different sentences in order to compare between sentences.

A total of 7 participants were involved, typing firstly a sentence both in HMS and T9 (in this order), and secondly another sentence in the same way. The sentences were "I study at the University of Lund" and "Hello I hope you're fine and don't forget our meeting in Lund tomorrow morning".

Typing 2 different sentences was important because for the first one, people were disturbing by the new system, using it for the first time and trying to learn how it worked. There were friendlier with it for the second sentence.

The results were quite positive. Firstly, in terms of keystrokes per character, we highlighted an improvement of 20% (that means that you need 20% less key pressed to write the message). We can consider it as an average because we kept all the mistakes done by the users, which sometimes increase the KSPC instead of reducing it.

Secondly, in terms of time, we highlighted a weakness of 15% (that means that you need 15% more time to write the message). The problem is that the system disturbs the user at the beginning, mostly because of short words and because you have to remember the letters you have already typed.

Comparing trained and not trained people (both of us and other users), we noticed that for the more experienced users the time consumption of HMS and T9 were roughly the same although the KSPC for HMS was lower. Furthermore, we noticed a big difference in time consumption when it came to trained and untrained people. We measered a 200% speed increase for trained people and this can probably be explained by the fact that as you get used to using the bigram prediction you get more confident that the system will predict the correct word. Naturally this is probably also true for new users of T9.

But the most important thing which reduces the efficiency of the system is the length of words. HMS reduces a lot the KSPC for long words but not for short words. This can be seen in table 1.

word	**T9**	**HMS**
tomorrow	8 kp	4 kp
in	2 kp	3 kp

Table 1: Key press Comparision

The problem is that when typing short words, long words are proposed first and you have to press one key more (the "Yes" key or the "T9" key) to restrict the length or stop the prediction. This increases KSPC and time consumption and could be optimized. It can be seen from the following two figures, *figure 3* and *figure 4*, how the KSPC changes as the word length increases.

7 Conclusions

As we have seen, we tried to improve the HMS system, which is a predictive text entry method using context. We first made it works in English, collecting an English dictionary and a corpus of the closest language from SMS language we could. After, we improved it by adding a key which can stop the prediction and come back to the T9 method. This

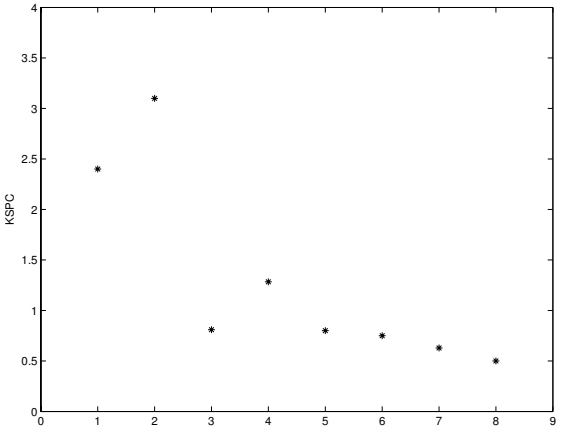


Figure 3: Word length versus KSPC for HMS

key permits to take both advantages of HMS and T9 methods when typing. Then we made the system intelligent by learning the words the most used by the user (without reload the system). After an evaluation, we showed an improvement of 20% using the KSPC as the measurement, and a decrease of 15% using the time as the measurement.

7.1 Improvements

But of course, this system can be optimized and improved again. Firstly, to be closer to the existing mobile phones, numbers, punctuation and special characters could be added in the keyboard. This could be used to do longer and more complex experiments and could be closer to the reality. Secondly, our results are not as closed to the reality as they could be because of the corpus we used to calculate the bigrams. We wanted to use free corpora of SMS but none exists yet today, so we used corpora from Usenet. This kind of corpora of SMS are collecting by researchers just now, because the technology is new and these corpora are needed to extend the researchs. We found one corpus of english SMS from Singapore, but not really useful because of strange words or names sometimes. Thus, we think that in a few years, more SMS copora will be available for researchs and evaluation will be closer to the reality.

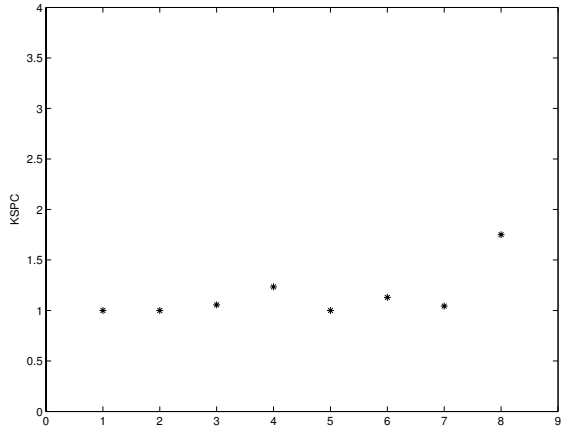


Figure 4: Word length versus KSPC for T9

References

[1] Zi Corporation. *eZiText*, 2002.

[2] Lexicus Devision. *iTap*. Motorola, 2001.

[3] Jon Hasselgren, Erik Montnemery, Pierre Nugues, and Markus Svensson. HMS: Predictive text entry. In *DAT171*, 2003.

[4] I. Scott MacKenzie. KSPC (keystrokes per character) as a characteristic of text entry techniques. In *http://www.yorku.ca/mack/hcimobile02.PDF*, 2002.

[5] Scott MacKenzie, Hedy Kober, Derek Smith, Terry Jones, and Eugene Skepner. Letterwise: Prefix-based disambiguation for mobile text input. In *http://www.eatoni.com/research/lwmt.pdf*, 2001.

[6] Andriy Pavlovych and Wolfgang Stuerzlinger. Less-tap: A fast and easy-to-learn text input technique for phones. In *http://www.cs.yorku.ca/ andriyp/papers/GI2003_Less-Tap.pdf*, 2003.

Named entity recognition using statistical methods

Stefan Ekenberg
Department of Computer Science
Lund University
d01se@efd.lth.se

Abstract

This report is the result of the project part of a course in computer linguistics. The assignment was to develop a java program that tags proper nouns using statistical methods. To do this, a Support Vector Machine was used and the main part of the project was to find out what parameters to send into the SVM to get a good result.

To evaluate the system the precision and recall values of a corpus were calculated.

1 Introduktion

Syftet med denna rapport är att beskriva hur statiska metoder kan användas i ett system för att tagga egennamn i en text. En träningstext används för att lära systemet vilka taggar som skall användas och i vilken kontext som dessa oftast befinner sig i. Informationen kan sedan användas för att tagga en liknande text.

Till skillnad från tidigare arbete som ”Name Extraction in Car Accident Reports for Swedish”[1] används inga gazetteers[2] eller regler som till exempel reguljära uttryck. Detta ger främst de två fördelarna att systemet blir generellt och kan på så sätt användas i många olika sammanhang samt att den implementerade koden blir mycket enklare att felsöka och få översikt över. Koden med reguljära uttryck blev mycket svårhanterlig och det var mycket svårt att hitta fel samt att lägga till nya uttryck. Dock har även detta nya system utvecklats för att fungera så bra som möjligt på texter om trafikolyckor, precis som tidigare nämnda arbete. Detta har påverkat en del val av parametrar som används för att känna igen kontexten för en viss tagg.

Parametrarna som plockas ut från det ord som skall taggas och dess kontext matas in i LIBSVM som är ett programpaket för att känna igen mönster. Uppgiften i projektet var att hitta de parametrar som skulle matas in i LIBSVM för att få bästa möjliga resultat.

Först kommer denna rapport nämna möjliga användningsområden för systemet och därefter beskrivs LIBSVM kortfattat. Sedan kommer de olika parametrarna som skickas in i detta programpaket behandlas noggrant och till sist görs analys av hur storleken på systemets träningskorpus påverkar resultatet samt en evaluering av korrektheten vid taggning som systemet producerar.

2 Bakgrund

System för att tagga egennamn kan till exempel användas till program som behandlar texter och behöver information om de olika orden i texten för att sedan på något sätt kunna tyda vad meningen betyder. CarSim[3], som är ett text-till-scenomvandlingsprogram för trafikolyckor, är ett exempel på program som har användning av att

[1] Ett projekt som tidigare utförts på LTH i kursen ”Språkbehandling och Datalingvistik”, precis som detta projekt. Se referenser.

[2] Gazetteer – lista med ord och motsvarande tagg som systemet har lagrad.

[3] Pierre Nugues, 2004. Se referenser.

veta vilka ord som är egennamn. Då kan det ta reda på vem som är inblandad i olyckan, var den inträffade och så vidare.

3 LIBSVM

LIBSVM bygger på Support Vector Machines som är en statistisk metod för att känna igen mönster. För att klassificera försöker SVM hitta en hyperyta i rymden av möjliga indata. Denna hyperyta försöker dela de positiva exemplen från de negativa. Delningen görs på ett sådant sätt att hyperytan får så långt avstånd som möjligt till de närmsta av de positiva och negativa exemplen. På så sätt kan en klassificering göras korrekt för testdata som är nära men inte identisk till träningsdatan. För mer information om SVM se ”Chih-Chung Chang and Chih-Jen Lin, LIBSVM: a library for support vector machines, 2001”, ”A Tutorial on Support Vector Machines for Pattern Recognition” och en hemsida från Microsoft Research CCSP Group på adressen http://research.microsoft.com/~jplatt/svm.html.

4 Parametrar

För att få idéer på vilka parametrar som kan ge bra resultat har bland annat artiklarna ”Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition”, ”Named Entity Recognition through Classifier Combination” och ”Named Entity Recognition with a Maximum Entropy Approach” studerats. Dessa användes som en utgångspunkt för vilka parametrar som med stor sannolikhet borde ingå. Därefter har felaktiga taggningar i resultatet studerats för att komma på nya parametrar som skulle kunna användas. Dessa har sedan testats för att se om de gör en positiv eller negativ inverkan på resultatet för att kunna avgöra om de skall användas respektive förkastas. På så sätt har en iterativ metod använts för att komma fram till slutresultatet.

I Tabell 1 finns en sammanställning av de parametrar som gjorde en positiv inverkan på resultatet och därför ingår i systemet.

Tabell 1. De olika parametrarna som används och deras relativa inverkan på resultatet. Vid framtagningen av värdena i tabellen användes en träningskorpus på 9502 ord och en testkorpus på 3805 ord. Värden större än 1 betyder att då parametern används så ökar värdet. Om en parameter är bra så skall den alltså ha värden som är större än 1 i kolumnerna ”Korrekta taggningar”, ”Precision” och ”Recall” medan värdet för ”Feltaggningar” skall vara mindre än 1 (förklaring till precision och recall finns i stycke 6).

Parametertyp	Korrekta taggningar:	Feltaggningar:	Precision:	Recall:
Ordets suffix	1,12	0,82	1,10	1,12
- Längden 4	1,02	0,93	1,03	1,02
- Längden 3	1,01	1,01	0,99	1,01
- Längden 2	1,03	0,95	1,02	1,03
- Längden 1	1,01	0,98	1,01	1,01
Föregående och efterföljande ord	1,07	0,84	1,11	1,07
- Ordet innan föregående	1,00	0,95	1,02	1,00
- Föregående ord	1,02	0,95	1,02	1,02
- Efterföljande ord	1,04	0,93	1,03	1,04
Inledande stor bokstav	1,06	0,89	1,02	1,06
Ordklassinformation	1,04	0,93	1,03	1,04
Frasinformation	1,03	0,95	1,02	1,03
Efterföljande är ett nummer	1,03	0,95	1,01	1,03
Efterföljande ord har stor bokstav	1,04	0,91	1,05	1,04

4.1 Ordets suffix

Suffix av det ord som skall taggas gav positivt resultat upp till längden fyra. Hela ordet används även som parameter men suffix kan få fram generella mönster på orden för de olika taggarna som inte enbart själva ordet kan. En möjlig anledning till att suffix av längden fem inte gav något förbättrat resultat är att inte träningsdatan var tillräckligt omfattande för att LIBSVM skulle kunna känna igen mönster av längden fem.

Suffix av längden ett (sista bokstaven) skulle man kunna tro var för generell för att användas. Dock var så inte fallet som visas i Tabell 1. Detta kan förklaras med samband som till exempel att gatunamn ofta slutar på bokstaven n vilket är sista bokstaven i ”gatan” och ”vägen”.

Suffix av längden två hade en större inverkan på resultatet men någon direkt förklaring till detta är svårt att ge. Dock är ju denna mindre generell än endast en bokstav vilket gör att mönster är lättare att upptäcka.

Suffix av längden tre är bra för att upptäcka mönster i efternamn vilka ofta slutar på ”son”, som till exempel ”Andersson” och ”Johnson”.

Suffix av längden fyra gav även ett positivt resultat. En möjlig förklaring till det positiva resultatet är till exempel samband som att städer ofta slutar på ”borg” som i ”Helsingborg” och ”Ludviksborg”.

Suffix av längden fem gav som tidigare inget förbättrat resultat. Dock skall det observeras att så är fallet när även alla de andra suffixen används. Om alla de andra ej används medan suffix av längden fem används så ger denna en positiv inverkan på resultatet jämfört med om inget suffix alls används.

4.2 Föregående och efterföljande ord

För att kunna utnyttja ett ords kontext används föregående och efterföljande ord som parametrar. Dessa medför att programmet tar hänsyn till i vilket sammanhang som ordet befinner sig i.

Testning har visat att systemet fungerar bäst då de två föregående orden var för sig samt det efterföljande ordet används som parametrar. Träningsdatans begränsade storlek kan ses som en anledning till att systemet inte fungerar bättre med ännu fler föregående och efterföljande ord som parametrar. Om det skall vara möjligt att se mönster på ett längre avstånd från ordet som skall taggas så måste systemet tränas mer eftersom orden får ett mindre och mindre samband med ordet som skall taggas ju längre avståndet är.

Ett exempel på en mening där kontexten har stor betydelse är ”Mannen och kvinnan åkte till Malmö.” som innehåller typiska ord för en viss sorts tagg innan ordet ”Malmö”. Om orden ”åkte” och ”till” befinner sig innan ett ord är det stor sannolikhet att ordet skall taggas med en tagg som LOCATION eller CITY, beroende på vilka taggar som systemet har tränats med. Ett exempel på en ytterligare mening är ”Jag vill ha glass, sade Johan” där ordet ”sade” innan ”Johan” kan hjälpa systemet att tagga rätt eftersom ord efter ”sade” med stor sannolikhet skall ha en tagg av typen PERSON.

Ett försök med bigram som parameter, det vill säga de två föregående orden tillsammans, gjordes även med ett misslyckat resultat som följd. Att skicka in orden var för sig som parametrar gav mycket bättre resultat. Återigen borde detta bero på träningsdatans storlek eftersom det finns många fler variationer av bigram än av orden var för sig.

Efterföljande ord kan även ha stor betydelse för att systemet skall ha möjlighet att tagga rätt. I meningar som ”De med svårast skador flyttades till Falu lasarett.” kan system har svårt att tagga ”Falu lasarett” som LOCATION om det inte vet om att ordet ”lasarett” kommer efter ”Falu”. Tabell 1 visar även att det efterföljande ordet inverkar mer på resultatet än de två föregående var för sig.

4.3 Inledande stor bokstav

En typisk egenskap för ett egennamn är att det inleds med stor bokstav. Därför används en parameter som kan anta värdena ”true” och ”false” som anger om ordet inleds med stor bokstav eller inte. Tabell 1 visar dock att parametern inte har så stor betydelse som man skulle kunna tro. Detta beror till största del på att den inte säger så mycket vilken tagg som skall sättas utan mer att någon tagg överhuvudtaget skall sättas. Att ett ord inleds med stor bokstav avslöjar inte om det är en stad eller ett personnamn.

Något som också måste observeras är att alla egennamn inte inleds med stor bokstav. Till ex-

empel ordgruppen ”riksväg 13” som möjligtvis skall taggas som ROAD har inte stor bokstav i något av orden.

4.4 Ordklassinformation

Om texten som skall taggas även innehåller information om ordets ordklass hjälper detta även systemet att tagga korrekt. Ett egennamn har då oftast ordklasstaggen ”pm.nom” vilket hjälper systemet på samma sätt som inledande stor bokstav. Dock kan det förekomma fel i ordklasstaggarna eftersom dessa antagligen är satta av en automatisk ordklasstaggare som inte är hundraprocentig. Detta har nackdelen att man får in fler felkällor i resultatet men som Tabell 1 visar så har denna parameter ändå en positiv inverkan på resultatet.

Den positiva inverkan som ordklassinformationen för med sig måste sättas i relation till den extra beräkningskraft som krävs för att köra ordklasstaggaren. Resultatet visar att ordklassinformationen inte har en så pass stor inverkan att den är nödvändig och därför kan systemet klara sig utan den då den ej finns tillgänglig.

4.5 Frasinformation

Precis som med ordklassinformationen kan texten som skall taggas först gå igenom en frastyptaggare som sätter ut frastypsinformation. Eftersom egennamn oftast taggas som substantivfras kan detta hjälpa systemet att identifiera att en tagg skall sättas, dock ej vilken. Frastypinformationen medför även den att det finns en felkälla eftersom en frastyptaggare inte heller alltid taggar korrekt. Denna parameter förbättrar ändå resultatet, dock är inte förbättringen inte lika tydlig som med ordklassinformationen.

Precis som med ordklassinformationen så visar resultatet i Tabell 1 att den extra beräkningskraft som krävs för att sätta ut frasinformationen inte alltid är motiverad. Frasinformation är alltså inte heller en nödvändighet men har en positiv inverkan på resultatet.

4.6 Efterföljande är ett nummer

Systemet kontrollerar om den grupp av tecken som befinner sig efter ordet som skall taggas är ett nummer. Denna information används som en parameter som kan anta värdena ”sant” eller ”falskt”. Taggen används för att kunna märka upp taggar som ROAD där dessa ofta skrivs som till exempel ”riksväg 13”. För att systemet skall ha möjlighet att tagga ”riksväg” korrekt behövs informationen att ”13” är ett nummer. Detta är en parameter som är lite specialanpassad till den korpus med trafikolyckstexter som användes för att träna och testa systemet och har ingen märkbar betydelse på andra typer av korpus.

4.7 Efterföljande ord har stor bokstav

Denna parameter ökar systemets förmåga att tagga personnamn korrekt. Till exempel ”Joakim” i ”Joakim Palmkvist” har större sannolikhet att få rätt tagg när denna parameter används. Däremot så hjälper den ju inte när till exempel endast förnamnet finns utskrivet i texten. Precis som parametern med kontrollen om efterföljande teckengrupp är ett nummer, så består även denna av antingen värdet ”sant” eller ”falskt” då efterföljande ord har inledande stor bokstav respektive ej inledande stor bokstav.

4.8 Föregående ords egennamnstagg

Innan denna parameter användes hade systemet stora problem med det gjorde konstiga taggningar som till exempel för ”Udo Theil”:

```
Udo I-PERSON
Theil I-CITY
```

Det vill säga taggningar som skulle innefatta flera ord fick istället olika taggar för de enstaka orden. Genom att även använde föregående ords tagg som parameter i systemet kunde taggningens kvalitet öka avsevärt vilken syns tydligt i Tabell 1. Det är ju mycket mindre sannolikt att två ord efter varandra har olika taggar än att de båda har samma tagg. Då denna parameter används får ”Udo Theil” sin korrekta taggning, det vill säga:

```
Udo I-PERSON
Theil I-PERSON
```

4.9 Kommentarer till val av parametrar

Tester har även gjorts med lite mer avancerade parametrar. Försök att komma till rätta med problemet att personnamn inte fick någon tagg då inte både för- och efternamn fanns utskrivet har gjorts, tyvärr med misslyckat resultat. Denna parameter använde det faktum att då ett namn nämns i en artikel så står nästan alltid hela namnet utskrivet först, det vill säga med både för- och efternamn och dessa kunde systemet tagga korrekt som PERSON. Dessa namn sparades sedan undan i en lista och för varje ord som skulle taggas så kontrollerades det mot denna lista för att se om det tidigare taggats som namn. I så fall fick parametern värdet ”sant”, annars värdet ”falskt”. Resultatet blev dock att många taggningar där både för- och efternamn fanns med nu istället blev felaktiga och systemet blev inte heller bättre på syftet med taggen – att tagga personnamn som endast bestod av för- eller efternamn.

Orsaken till detta är antagligen att parametern inte alltid är antingen ”sant” eller ”falskt” då systemet tränas eftersom första gången namnet förekommer är den ”falskt” och nästa gång är den ”sant”. På så sätt förvillar istället parametern eftersom den inte har samma värde för alla ord som skall taggas som PERSON.

En lösning på problemet hade varit att inte gå via SVM och använda informationen som en parameter utan att själv tagga ord som PERSON då de finns med i listan. Då förloras dock syftet med systemet eftersom det inte längre blir generellt och all information som finns i de andra parametrarna tas ingen hänsyn till. Det behöver ju inte vara ett personnamn bara för att det finns med i listan. Till exempel ”Berg” kan först finnas med i ”Johan Berg” och sedan inleda en mening som ”Berg är vackra att se på!”.

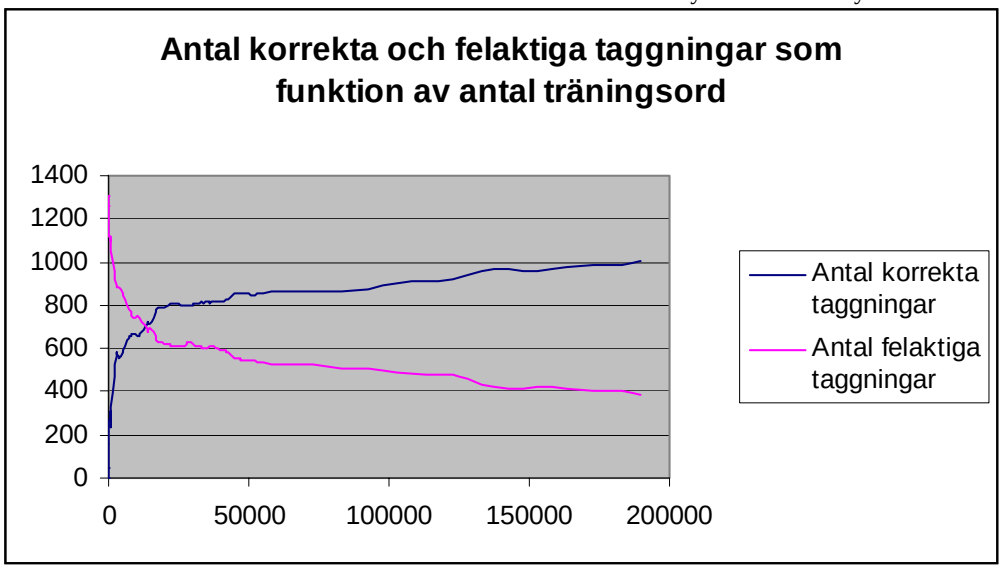
Läxan som kan läras av detta är att man inte skall anstränga sig för mycket vid val av parametrar och hitta på komplicerade samband, utan ta med generella saker som finns i ordets närhet, det vill säga den typen av parametrar som finns med i Tabell 1.

5 Träningskorpusens storlek

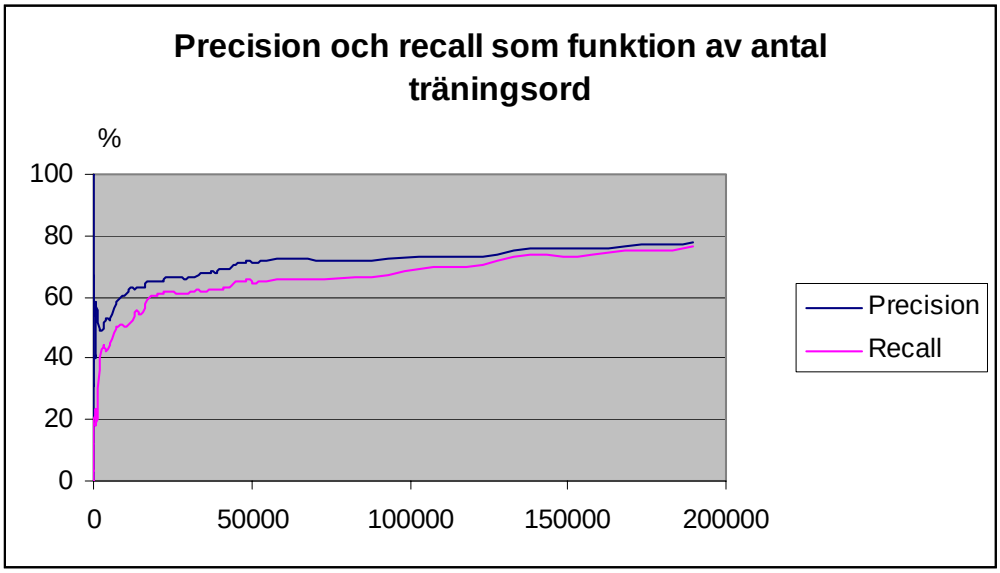
Det är inte bara parametrarna som har en stor betydelse för hur resultatet blir utan även storleken på träningskorpusen har en stor inverkan. Detta illustreras i Figur 1 och Figur 2 som visar hur antalet korrekta och felaktiga taggningar beror av antal träningsord respektive hur precision[4] och recall[5] beror av antalet träningsord.

I figurerna syns att kurvorna fortfarande vid 190000 ord inte har stabiliserat sig vilket betyder att en ännu större träningskorpus hade gett ett bättre resultat. Tyvärr fanns inte större träningskorpus tillgänglig för att vidare utforska hur stor den måste vara för att en ytterligare storleksök-

[4] Vad precision betyder beskrivs i stycke 6
[5] Vad recall betyder beskrivs i stycke 6



Figur 1. Visar hur antalet korrekta och felaktiga taggningar beror av träningsdatans storlek



Figur 2. Visar hur precision och recall beror av träningsdatans storlek

ning inte skall påverka resultatet.

6 Resultat

För att kunna evaluera systemet har en testkorpus med trafikolyckstexter, som innehåller 3805 ord, taggats manuellt vilket gav 182 taggar. Denna taggning jämfördes sedan med systemets producerade taggning. Innan dess tränades systemet med en annan trafikolyckskorpus innehållandes 9502 ord och 436 taggar. Dock var denna träningskorpus maskinellt taggad vilket gör att det inte är helt korrekt och detta påverkar systemets resultat negativt. Dessutom är det inte tillräckligt stort för att systemet skall ha en chans att lära sig de olika taggarnas egenskaper. Med en större korpus som är korrekt taggat hade alltså resultatet varit bättre.

Vid utvärderingen används följande vokabulär:

- **Answer file** – texten, maskinellt taggad av vårt program.
- **Key file** – texten, manuellt taggad. Denna text utgör definitionen på korrekt taggning.
- **Recall** – antalet korrekta taggar i Answer file dividerat med det totala antalet taggar i Key file.
- **Precision** – antalet korrekta taggar i Answer file dividerat med det totala antalet taggar i Answer file.

Systemet gav följande resultat:

```
Antal korrekta taggningar:
144
Antal feltaggningar: 55
Precision: 77.84%
Recall: 79.12%
```

Eftersom kvaliteten på den träningskorpus som användes inte var den bästa kan inte alltför stora slutsatser dras av detta resultat. Det ger dock en fingervisning på hur pass bra systemet är.

Systemet har även testats på en korpus som innehåller ekonomiska texter på engelska. Träningskorpusen innehöll ca 190000 ord och testkorpusen innehöll ca 23000 ord. Tyvärr var både tränings- och testkorpus maskinellt taggade vilket återigen gör att resultatet endas kan ses som en fingervisning. Då erhölls följande resultat:

```
Antal korrekta taggningar:
1001
Antal feltaggningar: 383
Precision: 77.96%
Recall: 76.41%
```

Resultatet visar att systemet även fungerar väl på texter skrivna i andra språk än svenska samt andra typer av texter. Som en jämförelse kan man studera resultatet från CoNLL-2003 Shared Task[6]

[6] Se referenser.

som gick ut på att tagga egennamn. Där fick det bästa systemet resultatet

Precision: 88.99%
Recall: 88.54%

på engelska texter vilket är ett betydligt bättre resultat. Dock använde detta system, precis som de flesta andra som ställde upp, gazetteers vilka ej skulle användas i detta system eftersom det skall vara så generellt som möjligt.

7 Slutsatser

Vikten av valet av bra parametrar kan inte nog påpekas när man skall utveckla ett system som använder en statistisk metod som detta system. För att ta fram parametrar som fungerar bra måste man testa sig fram. Dock kan man alltid ha en tanke i bakhuvudet som säger att generella parametrar som ligger i ordets kontext fungerar bäst. Man skall inte försöka hitta komplicerade samband.

Systemet som implementerades med hjälp av enbart statistiska metoder får tyvärr anses inte vara tillräckligt bra för kommersiell användning. Då måste precision och recall säkert upp till 97 % innan det kan anses intressant. Dit har systemet en lång väg. För att komma dit måste antagligen knep som gazetteers användas på bekostnad att systemens generallitet.

Tack

Jag vill tacka Richard Johansson för implementeringen av interface till LIBSVM, allmänna tips samt stödet under projektets gång.

8 Referenser

Lisa Persson and Magnus Danielsson. 2003. *Name Extraction in Car Accident Reports for Swedish.* Projektarbete på Lunds Tekniska Högskola i kursen "Språkbehandling och Datalingvistik".

Pierre Nugues. 2004. *Development of a Text-to-Scene Converter for Vehicle Accident Reports*, http://www.lucas.lth.se/lt/carsim.shtml

Chih-Chung Chang and Chih-Jen Lin. 2001. *LIBSVM: a library for support vector machines*. Mjukvara tillgänglig på internet-adressen http://www.csie.ntu.edu.tw/~cjlin/libsvm.

Christopher J.C. Burges. 1998. *A Tutorial on Support Vector Machines for Pattern Recognition*. Bell Laboratories, Lucent Technologies.

Microsoft Research. *Support Vector Machines*. http://research.microsoft.com/~jplatt/svm.html.

Erik F. Tjong Kim Sang and Fien De Meulder. 2003. *Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition*. CNTS – Language Technology Group University of Antwerp.

Radu Florian, Abe Ittycheriah, Hongyan Jing and Tong Zhang. 2003. *Named Entity Recognition through Classifier Combination*. IBM T.J. Watson Research Center 1101 Kitchawan Rd, Yorktown Heights, NY 10598, USA.

Hai Leong Chieu and Hwee Tou Ng. 2003. *Named Entity Recognition with a Maximum Entropy Approach.* Department of Computer Science National University of Singapore 3 Science Drive 2 Singapore 117543.

Investigating an implementation of Joakim Nivre's algorithm for projective dependency parsing of Swedish text.

Jörgen Hartman
Lund University Computer Science
jorgen.hartman.398@student.lu.se

Abstract

This paper presents some statistics on an implementation of Joakim Nivre's algorithm. The implementation in Prolog have a coverage of 100% because of the backtracking mechanism in Prolog. The rank of the correct graph within all produced graphs are not very good with this implementation. This paper also shows that the rank can be improved at the cost of the coverage.

1 Introduction

How hard is it to find the correct dependency graph from Swedish sentences? I will investigate an implementation of the algorithm described by Nivre [1]. The implementation itself was made by Pierre Nugues at Lunds Tekniska Högskola. I will try the implementation on an annotated Swedish Treebank called Talbanken. Talbanken is tagged using a probabilistic part-of-speech-tagger trained on the Stockholm Umeå Corpus (SUC). Talbanken consists of about 5000 sentences.

2 The algorithm.

Nivre's algorithm uses a basic shift reduce algorithm extended with some more parse actions:

- Left arc - Adds an arc from head to left dependent if there is a dependency rule that allows it.
- Right arc - Adds an arc from head to right dependent if there is a dependency rule that allows it.
- Reduce - Pops the node on top of the stack if it has a head.
- Shift - Pushes next word of the input onto the stack.

The algorithm makes sure that the graph will be acyclic, connected, and projective. The algorithm will always find a graph for any sentence provided there are lexical rules for all dependencies in the sentence. The algorithm and its different actions are further described in Nivre [1].

2.1 The implementation of the algorithm.

The implementation is done in Prolog programming language. Prolog uses a backtracking mechanism that allows the algorithm to produce a new alternative graph until the correct one is found or all alternatives are found. For example; if the dependency between two words can be determined by two different rules, Prolog will try the first rule, and if it does not produce a correct graph it will go back and try the other rule.

3 Investigation of the implementation.

The idea was to see if the algorithm always could find the correct graph. Prolog uses a backtracking mechanism that makes it possible to find several different graphs that can be generated from a single sentence. It would be interesting to see the rank of the correct graph within all the generated graphs.

I started with a Treebank called Talbanken. It's an annotated Treebank with Swedish sentences. It contains information like index and class of the words, index for the sentence, and dependencies between words. I used that information to extract the dependency graph from Talbanken and compare that graph with the graph produced by nivre's algorithm. I wrote a perl script that extracts the sentences from Talbanken into a file that can easily be read by prolog, and I made a script that extracted the graphs as well. The algorithm needs a set of dependency rules that covers the dependencies in the sentences. If a rule was missing you would not be able to find the correct graph. Since all information is available in Talbanken, I wrote a perl script that extracts all the dependency rules as well.

The test was then done in prolog. The testing program reads the first sentence and then uses the implementation of Nivre's algorithm to produce all possible dependency graphs. It then compares this

list of graphs with the dependency graph taken from Talbanken and writes the rank of the correct graph to a result file. If no correct graph was found, the rank would be set to zero.

I immediately ran into problems with stack overflows. Making prolog do a list containing all possible graphs for a sentence required a lot of memory, especially if the sentence was long. I made changes in all three perl scripts so you could set a maximum sentence length. It would now be easy to only extract sentences up to a specific length along with the corresponding graphs and dependency rules.

Tests with sentences of maximum length five showed that the coverage (number of correct graphs found) was 100%, and 74% of the correct graphs was ranked number one.

I found that some of the sentences in Talbanken were only one word long. The reason is that there are for example titles inside the annotated text in Talbanken. One-word sentences will of course always produce the correct dependency graph since it is equal to an empty list. I decided to implement an option to skip over one-word sentences when extracting information. With the same sentences but excluding those consisting of only one word, I still got 100% coverage, but only 47% of the graphs were ranked number one. So I will from now on exclude one-word sentences.

I tried with maximum sentence length ten, but it was not able to complete the algorithm due to memory shortage. The number of graphs generated from a sentence with ten words would be huge. I now understood that I would not be able to get ALL the graphs. I needed a way to limit the calculations. I started looking in the dependency rules direction, since every new rule would generate a large amount of graph combinations.

I experimented with the 1098 first sentences from Talbanken and got these results:

Maximum sentence length eight generated 252 sentences and graphs, and 110 rules.

100% coverage and 18% of the graphs ranked number one (average rank 387).

The same 252 sentences and rules extracted for sentences of maximum 7 words, I got 93 rules, 93% coverage and 20% of the graphs ranked number one (average rank 233).

The same 252 sentences and rules extracted for sentences of maximum 6 words, I got 84 rules, 88% coverage and 22% of the graphs ranked number one (average rank 202).

The result shows that if you have fewer rules the rank will get better, but the coverage will decrease.

Since talbanken is tagged by a probabilistic POS-tagger, some of the dependencies might be incorrect. Statistically, faulty dependency rules would be more common for longer sentences than for shorter.

The dependency rules can be either left or right oriented and when checking for a matching rule inside the algorithm you will try both rules. This generates a lot of backtracking in prolog, and requires a lot of memory. Extracting the dependency rules from shorter sentences will provide the algorithm with fewer rules to match. This will of course lower the coverage of the algorithm since some rules might be missing completely, but it will increase the speed, lower the memory usage of the implementation, and increase the rank of the graphs that are correct.

When you build a sentence you first make the core of the sentence, for example 'bilen röd'. Then you apply different rules to make the sentence more readable, for example linking the noun and adjective like 'bilen är röd'. You also add determiners like 'Den bilen är röd'. One of the last things you do before the sentence is complete is topicalization to restructure the phrases like 'bilen den är röd'. That would generate a dependency rule; 'From noun to determiner where determiner is to the right of the noun'. This looks like a strange rule, and it's not common at all. It would cause the implementation in prolog to make a lot of extra graphs for all sentences containing determiners. Topicalizations are more common in longer sentences than in shorter ones. This shows why the rank can be improved at the cost of the coverage by extracting the dependency rules from shorter sentences only.

Another way of limiting the number of dependency rules would be to split up the data. Using smaller chunks of text from Talbanken, I suspected I could increase the rank. I also suspected that the coverage would go down since each rule would percentage wise be a larger part of the rules needed for 100% coverage.

With 500 sentences from Talbanken and maximum sentence length eight, the perl script generated 102 sentences and 83 dependency rules. The test result I got now was:

100% coverage, 22% was rank one, average rank was 344. The same sentences but with rules extracted for sentences of maximum 7 words, I got 61 rules, 84% coverage, 31% was rank one, average rank was 91. The same sentences but with rules extracted for sentences of maximum 6 words, I got 54 rules, 79% coverage, 35% was rank one, average rank was 81.

With 200 sentences from Talbanken and maximum sentence length eight, the perl script generated 38 sentences and 42 dependency rules. The test result I got now was: 100% coverage, 39% was rank one, average rank was 88. The same sentences but with rules extracted for sentences of maximum 7 words, I got 32 rules, 79% coverage, 53% was rank one, average rank was 38. The same sentences but with rules extracted for sentences of maximum 6 words, I got 25 rules, 66% coverage, 60% was rank one, average rank was 23.

Ultimately, you could extract the rules dynamically from Talbanken, giving a set of rules for each sentence. The algorithm would then use only the rules attached to a specific sentence. That would give 100% coverage and a very good rank. Of course this would not apply to real life cases, because you then need to have all rules available at all times, but it would give a good idea on how robust the algorithm is.

There are more ideas on how to improve the rank of the dependency graphs. One of them would be to implement a probability check for the dependency rules within the algorithm. If the algorithm had two rules to choose from, it would choose the one which are most often used.

Talbanken is tagged with features in addition to the part-of-speech tag. The feature is more granular than the part-of-speech tag and describe for example tense (present, preterite, supinum, and infinite) and degree (positive, comparative, and superlative). Keeping these features will make each dependency rule apply to less words. It would therefore be interesting to investigate the rank when keeping the features within the dependency rules.

4 Conclusion

The implementation of Nivre's algorithm is very robust and will always produce a graph provided that there are dependency rules covering the word classes in the sentences. The correct graph will always be produced because of the backtracking mechanism in prolog. Too many rules or faulty rules will make the implementation produce lower (worse) ranked graphs.

5 Acknowledgements

Pierre Nugues, Lunds Tekniska Högskola: Project leader, algorithm implementation.

Klas Sigbo, Lunds Tekniska Högskola: Answering Prolog questions.

Richard Andersson, Student at Lund University Cognitive Science: Linguistic discussions.

References

[1] Joakim Nivre: An efficient algorithm for projective dependency parsing. School of Mathematics and Systems Engineering, Växjö University.

[2] Pierre Nugues: An Introduction to Language Processing with Perl and Prolog, August 2004. Unfinished book used as course material, Lunds Tekniska Högskola.

[3] Joakim Nivre and Mario Scholz: Deterministic dependency parsing of English text. School of Mathematics and Systems Engineering, Växjö University.

Appendix:
Users manual.

Files needed:
conv_xml2sent.perl
conv_xml2graph.perl
conv_xml2drules.perl
calcResult.perl
nivre_2.pl
readfiles.pl
TalbankenMalt.xml

TalbankenMalt.xml can be downloaded from http://w3.msi.vxu.se/~nivre/research/talbanken.html

Make sure you have an untouched version of the file TalbankenMalt.xml or at least a part of the original file that contains the sentences you wish to work with.

If you want to limit the tests to only work with sentences up to a specific length, you need to edit the value of the variable sentence_length_limit. This is done in all three perl scripts (conv_xml2...). Please note that the script that extracts the sentences and the graphs need to have the same value.

if you want to exclude sentences that are only one word long (they will always give correct graph) from the tests, you need to change the variable named $removeSingleWordSentences. The value true will make the perl script skip all one-word sentences. This variable needs to be changed in both conv_xml2sent.perl and conv_xml2graph.perl. This variable is not found in the dependency rules extraction script because there are no dependencies in a one-word sentence.

NOTE: Make sure you work with the SAME input file when running the three different perl scripts.

Convert the xml to sentences that nivres algorithm can read in prolog:
perl conv_xml2sent.perl TalbankenMalt.xml
This will produce a <Sentences> file.

The dependency rules needed by Nivre's algorithm are extracted by a perl script:
perl conv_xml2drules.perl TalbankenMalt.xml
This will produce a <Dependency rules> file.

The correct graphs as given by TalbankenMalt.xml is extracted by a perl script:
perl conv_xml2graph.perl TalbankenMalt.xml
This will produce a <Graphs> file.

start prolog with:
pl -G20m
This increases the global stack size to 20 Mb (4 Mb default). You might have to increase even more for longer sentences.

consult needed files (algorithm, dependency rules, help predicates):
consult([nivre_2,drules,readfile]).

The command:
readfiles(<Sentences>,<Graphs>,<Result>).
should perform the tests and write the result to the <Result> file.

Run the perl script called calcResult.perl to calculate the coverage, rank, and other statistics for the result:
perl calcResult.perl result
This will give the statistics in the terminal window.

Naive Bayes Spam Filtering Using Word Position Attributes

Johan Hovold
Department of Computer Science, Lund University
Box 118, 221 00 Lund, Sweden
johan.hovold.363@student.lu.se

Abstract

This paper explores the use of the naive Bayes classifier as the basis for personalized spam filters. Various machine learning algorithms, including variants of naive Bayes, have previously been used for this purpose, but the author's implementation using word position based attribute vectors gives very good results when tested on several publicly available corpora.

The effect of various forms of attribute selection—removal of frequent and infrequent words, respectively, and by using Mutual Information—is investigated. It is also shown how n-grams, with $n > 1$, may be used to boost classification performance. Finally, a weighting scheme for cost-sensitive classification of variable length attribute vectors is introduced.

1 Introduction

The problem of unsolicited bulk e-mail, or *spam*, gets worse for every year. The vast amount of spam being sent wastes resources on the Internet, wastes time for users and may expose children to unsuitable contents (e.g. pornography). This development has stressed the need for automatic spam filters.

Early spam filters were instances of *knowledge engineering*, using hand-crafted rules (e.g. the presence of the string "buy now" indicates spam). The process of creating the rule base requires both knowledge and time, and the rules were thus often supplied by the developers of the filter. Having common and, more or less, publicly available rules made it easy for spammers to construct their e-mails to get through the filters.

Recently, a shift has occurred, as more focus has been put on *machine learning* for the automatic creation of personalized spam filters. A supervised learning algorithm is presented with e-mails from the users mailbox and outputs a filter. The e-mails have previously been classified manually as spam or non-spam. The resulting spam filter has the advantage of being optimized for the e-mail distribution of the individual user. Thus it is able to use also the characteristics of non-spam, or *legitimate*, e-mails (e.g. presence of the string "machine learning") during classification.

Perhaps the first attempt of using machine learning algorithms for the generation of spam filters was reported by Sahami et al. (1998). They trained a *naive Bayes classifier* and reported promising results. Other algorithms have been tested but there seems to be no clear winner (Androutsopoulos et al., 2004). The naive Bayes approach have been picked up by end-user applications such as the Mozilla e-mail client[1] and the free software project SpamAssassin[2], where the latter is using a combination of both rules and machine learning.

Spam filtering differs from other text categorization tasks in at least to ways. First, one might expect a greater class heterogeneity—it is not the contents per se that defines spam, but rather the fact that it is unsolicited. Similarly, the class of legitimate messages may also span a number of diverse subjects. Secondly, misclassifying a legitimate message is generally much worse than misclassifying a spam.

In this paper the results of using a variant of the naive Bayes classifier for spam filtering, will be presented. The effect of various forms of *attribute selection*, will be explored, as will the effect of considering not only single tokens, but rather sequences of tokens, as attributes. A scheme for cost-sensitive classification will also be introduced. All experiments have been conducted on several publicly available corpora, thereby making a comparison with previously published results possible.

The rest of this paper is organized as follows:

[1] http://www.mozilla.org/
[2] http://www.spamassassin.org/

section 2 presents the naive Bayes classifier; section 3 discusses the benchmark corpora used; the experimental results are presented in section 4; section 5 gives a comparison with previously reported results and in the last section some conclusions are drawn.

2 The Naive Bayes Classifier

In the general context, the instances to be classified are described by attribute vectors $A = \langle a_1, a_2 \ldots, a_n \rangle$. Bayes' theorem says that the posterior probability of an instance A being of a certain class c is

$$P(c|A) = \frac{P(A|c)P(c)}{P(A)}. \quad (1)$$

The naive Bayes classifier then assigns to an instance the most probable, or maximum a posteriori, classification from a finite set C of classes

$$c_{MAP} \equiv \operatorname*{argmax}_{c \in C} P(c|A).$$

By noting that the prior probability $P(A)$ in Equation (1) is independent of c, we may rewrite the last equation as

$$c_{MAP} = \operatorname*{argmax}_{c \in C} P(A|c)P(c). \quad (2)$$

The posterior probabilities $P(A|c) = P(a_1, a_2 \ldots, a_n|c)$ could be estimated directly from the training data, but are generally infeasible to estimate unless the available data is vast. Thus the *naive Bayes assumption*—that the individual attributes are conditionally independent of each other, given the classification—is introduced:

$$P(a_1, a_2, \ldots, a_n|c) = \prod_i P(a_i|c).$$

With this strong assumption, Equation (2) becomes the naive Bayes classifier:

$$c_{NB} = \operatorname*{argmax}_{c \in C} P(c) \prod_i P(a_i|c) \quad (3)$$

(Mitchell, 1997).

In text classification applications, one may choose to define one attribute for each word position in a document. This means that we need to estimate the probability of a certain word w_k occurring at position i, given the target classification c_j: $P(a_i = w_k|c_j)$. Due to training data sparseness, we introduce the additional assumption that the probability of a specific word w_k occurring at position i is identical to the probability of that same word occurring at position m: $P(a_i = w_k|c_j) = P(a_m = w_k|c_j)$ for all i, j, k, m. Thus we estimate $P(a_i = w_k|c_j)$ with $P(w_k|c_j)$. The probabilities $P(w_k|c_j)$ may be estimated with *maximum likelihood estimates*, using Laplace smoothing to avoid zero probabilities:

$$P(w_k|c_j) = \frac{C_j(w_k) + 1}{n_j + |Vocabulary|},$$

where $C_j(w_k)$ is the number of occurrences of the word w_k in all documents of class c_j, n_j is the total number of word positions in documents of class c_j and $|Vocabulary|$ is the number of distinct words in all documents (Mitchell, 1997).

Note that during classification the index i in Equation (3) ranges over all word positions containing words also in the vocabulary, thus ignoring so called *out-of-vocabulary* words. For a more elaborate discussion of the text model used see Joachims (1997).

3 Benchmark Corpora

The experiments were be conducted on the PU corpora[3] and the SpamAssassin corpus[4]. The four PU corpora, dubbed PU1, PU2, PU3 and PUA respectively, have been made publicly available by Androutsopoulos et al. (2004) in order to promote standard benchmarks. The four corpora contain private mailboxes of four different users in encrypted form. The messages have been preprocessed and stripped from attachments, HTML-tags and mail headers (except `Subject`). This may lead to overly pessimistic results since attachments, HTML-tags and mail headers may add useful information to the classification process. For more information on the compositions and characteristics of the PU corpora see Androutsopoulos et al. (2004).

The SpamAssassin corpus (SA) consists of private mail, donated by different users, in unencrypted form with headers and attachments retained[5]. The fact that the e-mails are collected from different distributions may lead to overly optimistic results, e.g. if (some of) the

[3]The PU corpora may be downloaded from http://www.iit.demokritos.gr/skel/i-config/

[4]The SpamAssassin corpus is available at http://spamassassin.org/publiccorpus/

[5]Due to a primitive mbox parser, e-mails containing non-textual or encoded parts, i.e. most e-mails with attachments, are ignored completely in the experiments.

spam messages have been sent to a particular address, but none of the legitimate messages have. On the other hand, the fact that the legitimate messages have been donated by different users may lead to underestimates since this should imply greater diversity of the topics of legitimate e-mails.

The sizes and compositions of the five corpora are shown in Table 1.

corpus	messages	spam freq
PU1	1099	44%
PU2	721	20%
PU3	4139	44%
PUA	1142	50%
SA	6047	31%

Table 1: Sizes and spam frequencies of the five corpora.

4 Experimental Results

As mentioned above, misclassifying a legitimate mail as spam ($L \rightarrow S$) is in general worse than misclassifying a spam message as legitimate ($S \rightarrow L$). In order to capture such asymmetries when measuring classification performance, two measures from the field of information retrieval, called precision and recall, are often used. Denote with $|S \rightarrow L|$ and $|S \rightarrow S|$ the number of spam messages classified as legitimate and spam, respectively, and similarly for $|L \rightarrow L|$ and $|L \rightarrow S|$. Let N_S and N_L be the total number of spam and legitimate messages, respectively. Then *spam recall*(R) and *spam precision*(P) are defined as

$$R = \frac{|S \rightarrow S|}{N_S} \quad \text{and} \quad P = \frac{|S \rightarrow S|}{|S \rightarrow S| + |L \rightarrow S|}.$$

In the rest of this paper spam recall and spam precision will be referred to simply as recall and precision. Intuitively, recall measures effectiveness and precision gives a measure of safety. One is often willing to accept lower recall (more spam messages slipping through) in order to gain precision (fewer misclassified legitimate messages).

Sometimes *accuracy* (Acc) is used as a combined measure

$$Acc = \frac{|L \rightarrow L| + |S \rightarrow S|}{N_L + N_S}.$$

All experiments have been conducted using 10-fold cross validation, i.e. the messages have been divided into ten partitions[6] and at each iteration nine partitions have been used for training and the remaining tenth for testing. The reported figures are the means of the values from the ten iterations.

4.1 Attribute Selection

It is common to apply some form of attribute selection process, retaining only a subset of the words—or rather tokens, since punctuation signs and other symbols are often included—found in the training messages. This way the learning and classification process may be sped up and memory requirements are lowered. Attribute selection may also lead to increased classification performance, e.g. since the risk of overfitting the training data is reduced.

Removing infrequent and frequent words, respectively, are two possible approaches. The rationale behind removing infrequent words is that this is likely to have a significant effect on the size of the attribute set and that predictions should not be based on such rare observations anyway. Removing the most frequent words is motivated by the fact that common words, such as the English words "the" and "to", are as likely to occur in spam as in legitimate messages. Furthermore, this has the effect of making sure that very frequent tokens do not dominate Equation (3) completely.

Another possibility—used by Sahami et al. (1998), Androutsopoulos et al. (2000) and Androutsopoulos et al. (2004)—is to rank the attributes using *Mutual Information*(MI), and to keep only the highest scoring ones. $MI(X;C)$ gives a measure of how well an attribute X discriminates between the various classes in C, and is defined as

$$\sum_{x \in \{0,1\}} \sum_{c \in C} P(x,c) \log \frac{P(x,c)}{P(x)P(c)}$$

(Cover and Thomas, 1991). The probability distributions are estimated using maximum likelihood estimates with Laplace smoothing.

In the experiment tokens occurring less than $n = 1, \ldots, 15$ times were removed. The results indicated unaffected or slightly increased precision at the expense of slightly reduced recall, as n grew. The exception was the PU2 corpus, where precision dropped significantly. The rea-

[6] The PU corpora come prepartitioned and the SA corpus has been partitioned according to the last digit of the messages decimal id.

son for this may be that PU2 is the smallest corpus and contains many infrequent tokens. On the other hand, removing infrequent words had a dramatic impact on the vocabulary size (see Figure 1). Removing tokens occurring less than three times seems to be a good trade-off between memory usage and classification performance, reducing the vocabulary size with 56–69%. This selection scheme was used throughout the remaining experiments.

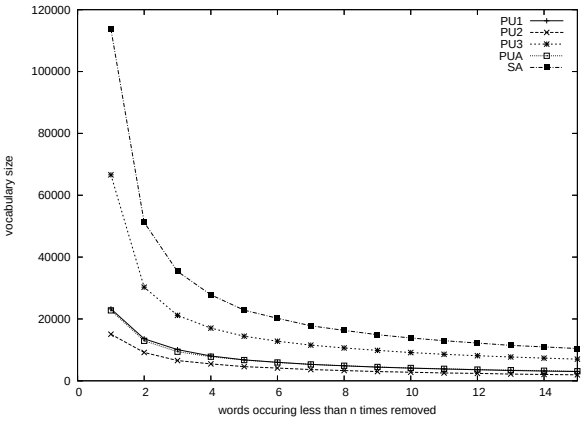


Figure 1: Impact on vocabulary size when removing infrequent words.

Removing the most frequent words turned out to have a major effect on both precision and recall (see Figure 2). This was most significant on the largest and non-preprocessed SA corpus where recall increased from 77% to over 95% by just removing the hundred most common tokens, but classification gained from removing the 100–200 most frequent tokens on all corpora. Removing too many tokens reduced classification performance—again most notably on the smaller PU2 corpus.

In the last attribute selection experiment MI-ranking was used instead of removing the most frequent tokens. Although the gain in terms of reduced memory usage was high—the vocabulary size dropped from 7000–35000 to the number of attributes chosen to be kept, e.g. 500–3000—classification performance was significantly reduced (see Figure 3). Since learning and classification time is mostly unaffected—MI still has be calculated for all attributes—I see no reason for using MI-ranking, if memory usage is not crucial[7].

[7]Androutsopoulos et al. (2004) reaches the opposite conclusion.

4.2 n-grams

Up to now each attribute has corresponded to a single word position, or unigram. Is it possible to obtain better results by considering also token sequences of length two and three, i.e. n-grams for $n = 2, 3$? The questioned was raised and answered partially in Androutsopoulos et al. (2004). Although many bi- and trigrams were shown to have very high information contents, as measured by MI, no improvement was found.

There are many possible ways of extending the attribute set with general n-grams, e.g. by using all available n-grams, by just using some of them or by using some kind of back-off approach. The attribute probabilities, $P(w_i, w_{i+1}, \ldots w_{i+n}|c_j)$, are still estimated using maximum likelihood estimates with Laplace smoothing

$$\frac{C_j(w_i, w_{i+1}, \ldots, w_{i+n}) + 1}{n_j + |Vocabulary|}$$

(see Section 2). Note that extending the attribute set in this way will result in a total probability mass greater than one. Fortunately, this need not be a problem since we are not estimating the classification probabilities explicitly (see Equation (3)).

It turned out that adding bi- and trigrams to the attribute set increased classification performance on all the PU corpora, but not on the SA corpus. The various methods for extending the attribute set all gave similar results and I settled on the simple version which just considers each n-gram as an independent attribute[8]. The results are shown in Table 2.

The precision gain was highest for the corpus with lowest initial precision, namely PU2. For the other PU corpora the precision gain was relatively small or even non-existing. At first the significantly decreased classification performance on the SA corpus came as a bit of a surprise. The reason turned out to be that when considering all bi- and trigrams in the non-preprocessed SA corpus, a lot of very frequent attributes, originating from mail headers and HTML, are added to the attribute set. This had the effect of giving badly discriminating attributes (e.g. some mail headers) and HTML, a too dominant role in Equation (3). By removing

[8]This is clearly not true. The three n-grams in the phrase "buy now"—"buy", "now" and "buy now"—are obviously not independent.

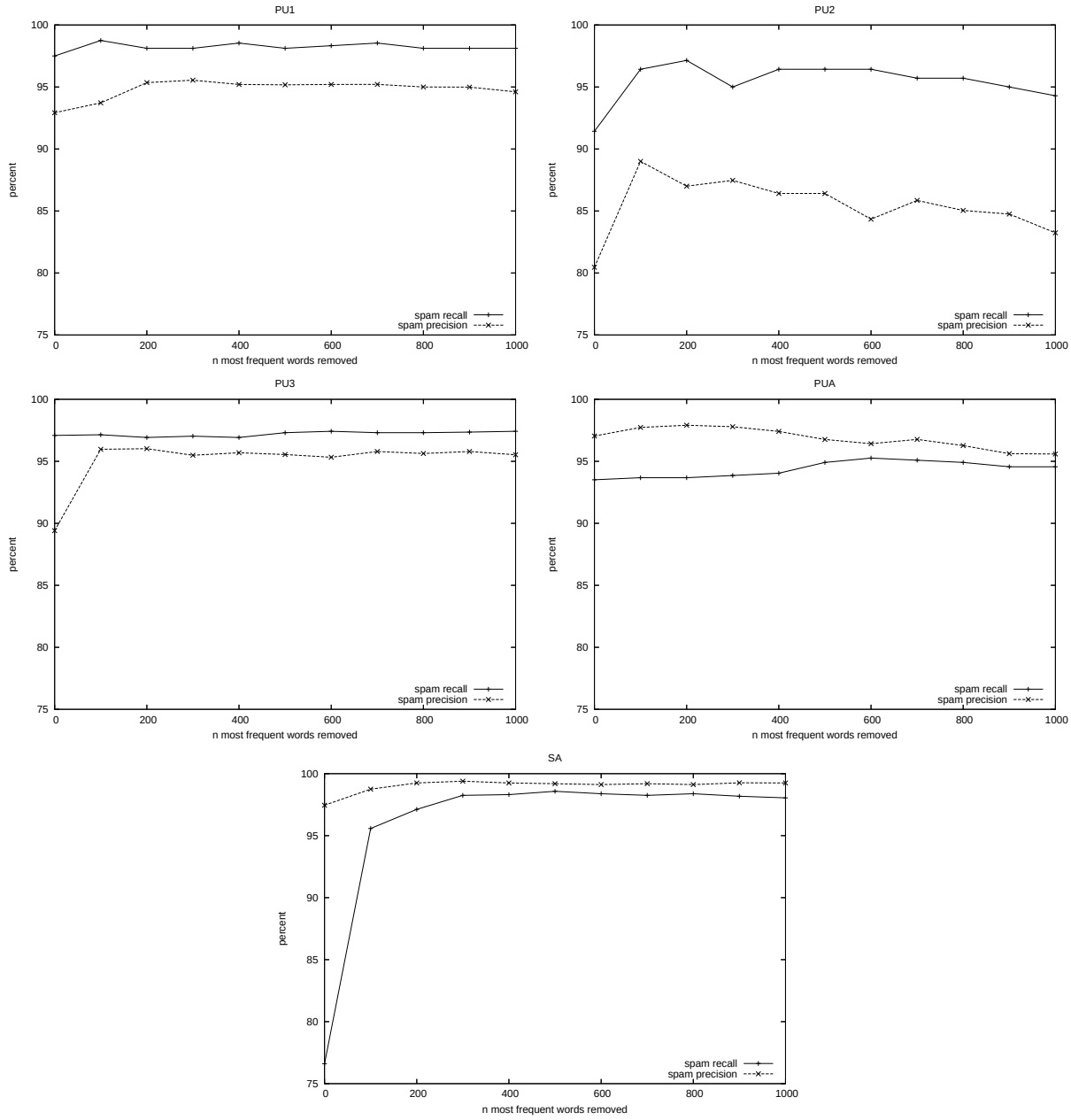


Figure 2: Impact on spam precision and recall when removing the most frequent words.

more of the most frequent words, classification performance was increased also for the SA corpus (see Table 3). The conclusion to be drawn is that mail headers and HTML, although containing useful information, shouldn't be included by brute force. Perhaps some kind of weighting scheme or selective inclusion process would be appropriate.

Finally, considering that extending the attribute set with bi- and trigrams has a dramatic effect on the vocabulary size, the gained classification performance is unlikely to compensate for the increased memory requirements.

4.3 Cost-Sensitive Classification

Generally it is much worse to misclassify legitimate mails than letting spam slip through the filter. Hence, it would be desirable to be able to bias the filter towards classifying messages as legitimate, yielding higher precision at the expense of recall.

One way of biasing the filter is to multiply the prior probability of legitimate messages by some factor $\lambda > 1$ (Androutsopoulos et al., 2000;

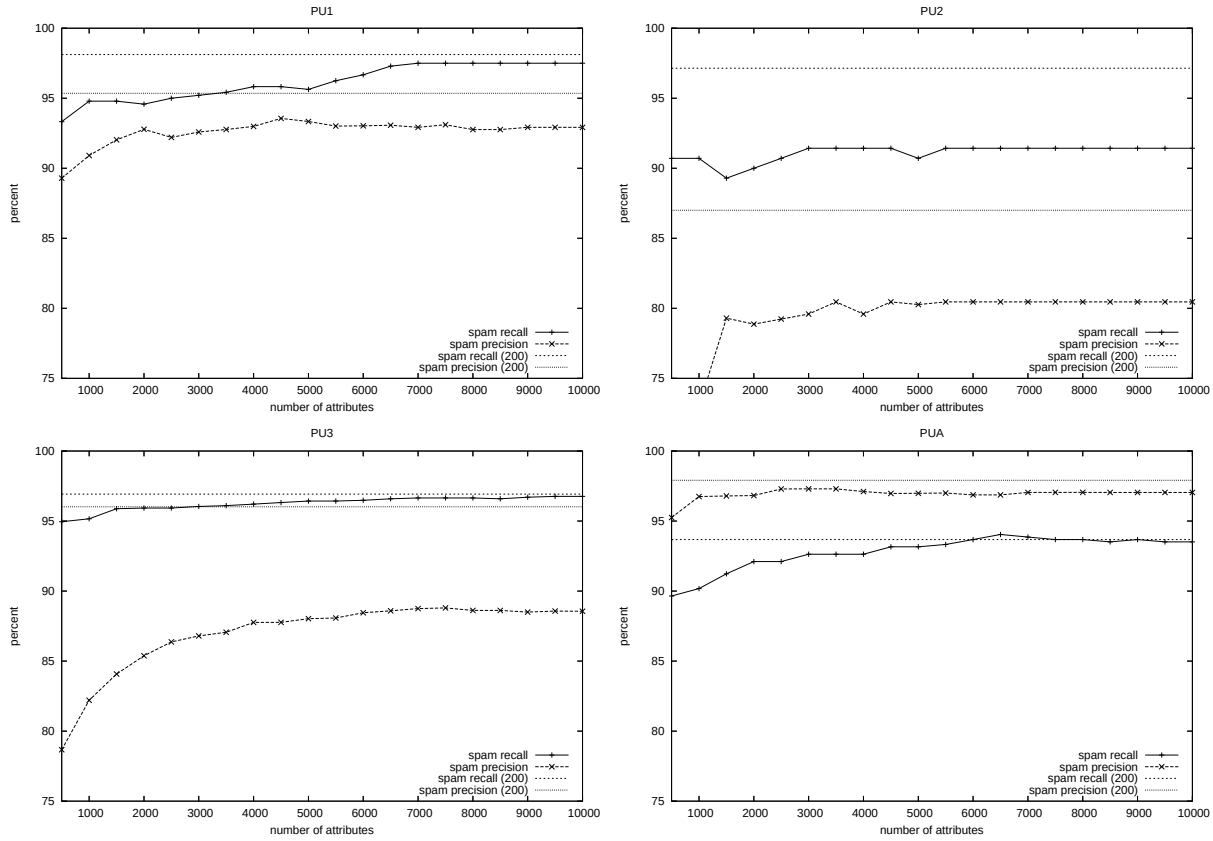


Figure 3: Attribute selection using Mutual Information on the PU corpora—spam recall and precision versus the number of retained attributes. Included is also the precision and recall figures when only the 200 most frequent words have been removed.

Androutsopoulos et al., 2004). This turns out to have a very limited effect, since the expression in Equation (3) is dominated by the posterior probabilities. Another problem is that this weighting scheme is inappropriate to use with word position based attribute vectors, as the impact of the cost factor λ will vary with the length of the document being considered.

To overcome these problems the following simple weighting scheme was used; each posterior probability $P(w_i|c_{legit})$ in Equation (3) was multiplied with a weight $w > 1$. The result of using this "tuning knob" can be seen in Figure 4.

5 Evaluation

Many different machine Machine Learning algorithms besides naive Bayes, such as C4.5, k-Nearest Neighbor and Support Vector Machines, have previously been used in spam filtering experiments. There seems to have been no clear winner, but there is a difficulty in comparing the results of different experiments, since the used corpora have rarely been made publicly available (Androutsopoulos et al., 2004). This section gives a comparison with the implementation and results of the authors of the PU corpora.

In Androutsopoulos et al. (2004), a variant of naive Bayes was compared with three other learning algorithms; Flexible Bayes, LogitBoost and Support Vector Machines (SVM). All of the algorithms used real valued word frequency attributes. The attributes were selected by removing words occurring less than five times and then keeping the 600 words with highest Mutual Information (see Section 4.1). As can be seen in Table 4, the word position based naive Bayes implementation of this paper achieved significantly higher precision and better or comparable recall on all four PU corpora. The results were also better or comparable to the results of the best-performing algorithm on each corpus.

In Androutsopoulos et al. (2000), the authors used a naive Bayes implementation based on boolean attributes, representing the pres-

n-grams	R	P	Acc
PU1			
$n = 1$	98.12	95.35	97.06
$n = 1, 2, 3$	99.17	96.19	97.89
PU2			
$n = 1$	97.14	87.00	96.20
$n = 1, 2, 3$	95.00	93.12	96.90
PU3			
$n = 1$	96.92	96.02	96.83
$n = 1, 2, 3$	96.59	97.83	97.53
PUA			
$n = 1$	93.68	97.91	95.79
$n = 1, 2, 3$	94.56	97.90	96.23
SA			
$n = 1$	97.12	99.25	98.95
$n = 1, 2, 3$	92.26	98.70	97.42

Table 2: Comparison of classification results when using only unigram attributes and uni-, bi- and trigram attributes, respectively. In the experiment words occurring less than three times and the 200 most frequent words have been removed.

n-grams	f	R	P	Acc
$n = 1$	200	97.12	99.25	98.95
$n = 1, 2, 3$	200	92.26	98.70	97.42
$n = 1, 2, 3$	5000	98.46	99.66	99.46

Table 3: Comparison of classification results on the SA corpus when using only unigram attributes and uni-, bi- and trigram attributes, respectively. In the experiment words occurring less than three times and the f most frequent words have been removed.

ence or absence of a fixed number of words. The attributes were selected using Mutual Information. In their experiments three different cost scenarios were explored. Table 5 compares the best results achieved on the PU1 corpus[9] for each scenario, with the results achieved by the naive Bayes implementation of this paper. Due to the difficulty of relating the two different weights, λ and w, the weight w has been selected in steps of 0.05 in order to get equal or higher precision. The authors deemed out the $\lambda = 999$ scenario because of the low recall figures.

[9]The results are for the *bare* PU1 corpus, i.e. the stop-list and lemmatizer have not been applied. The number of attributes have been optimized for each cost scenario.

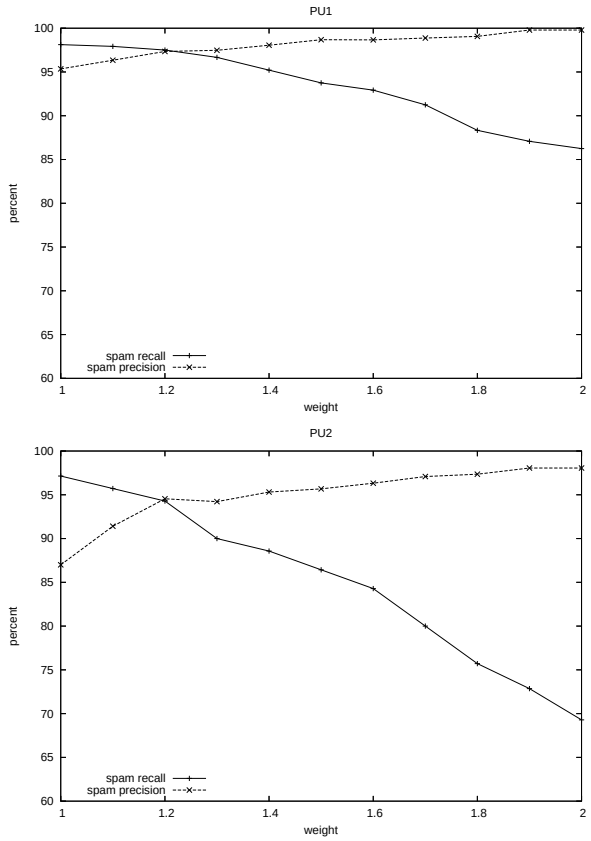


Figure 4: Cost-sensitive classification on the PU1 and PU2 corpora—spam recall and precision versus classification weight.

6 Conclusions

In this paper it has been shown that it is possible to achieve very good classification performance using a word position based variant of naive Bayes. The simplicity and low time complexity of the algorithm, thus makes naive Bayes a good choice for end-user applications.

The importance of attribute selection has been stressed—memory requirements may be lowered and classification performance increased.

By extending the attribute set with n-grams ($n = 1, 2, 3$), better classification performance may be achieved, although at the cost of significantly increased memory requirements.

With the use of a simple weighting scheme, precision may be boosted further, while still retaining a high enough recall level—a feature very important in real life applications.

7 Acknowledgments

The author would like to thank Pierre Nugues for inspiring comments during this work. Many

learner	R	P	Acc
PU1			
Androutsopoulos	99.38	89.58	94.59
Hovold	98.12	95.35	97.06
Flexible Bayes	97.08	96.92	97.34
PU2			
Androutsopoulos	90.00	80.77	93.66
Hovold	97.14	87.00	96.20
Flexible Bayes	79.29	90.57	94.22
PU3			
Androutsopoulos	94.84	93.59	94.79
Hovold	96.92	96.02	96.83
SVM	94.67	96.48	96.08
PUA			
Androutsopoulos	94.04	95.11	94.47
Hovold	93.68	97.91	95.79
Flexible Bayes	91.58	96.75	94.04

Table 4: Comparison of the results achieved by naive Bayes in Androutsopoulos et al. (2004) and by the author's implementation. In the latter, attributes were selected by removing the 200 most frequent words as well as words occurring less than three times. Included is also the results of the best-performing algorithm for each corpus, as found in Androutsopoulos et al. (2004).

thanks also to Peter Alriksson and Pontus Melke for useful suggestions regarding this paper.

References

Ion Androutsopoulos, John Koutsias, Konstantinos Chandrinos, and Constantine D. Spyropoulos. 2000. An experimental comparison of naive bayesian and keyword-based anti-spam filtering with personal e-mail messages. *CoRR*, cs.CL/0008019.

Ion Androutsopoulos, Georgios Paliouras, and Eirinaios Michelakis. 2004. Learning to filter unsolicited commercial e-mail. Technical Report 2004/2, NCSR "Demokritos". Revised version.

Thomas M. Cover and Joy A. Thomas. 1991. *Elements of Information Theory*. Wiley.

Thorsten Joachims. 1997. A probabilistic analysis of the Rocchio algorithm with TFIDF for text categorization. In Douglas H. Fisher, editor, *Proceedings of ICML-97, 14th International Conference on Machine Learning*, pages 143–151, Nashville, US. Morgan Kaufmann Publishers, San Francisco, US.

Tom M. Mitchell. 1997. *Machine Learning*. McGraw-Hill.

learner	R	P
Androutsopoulos ($\lambda = 1$)	83.98	95.11
Hovold (unigram, w=1)	98.12	95.35
Hovold (n-gram, w=1)	99.17	96.19
Androutsopoulos ($\lambda = 9$)	78.77	96.65
Hovold (unigram, w=1.20)	97.50	97.34
Hovold (n-gram, w=1.05)	99.17	97.15
Androutsopoulos ($\lambda = 999$)	46.96	98.80
Hovold (unigram, w=1.65)	91.67	98.88
Hovold (n-gram, w=1.30)	96.04	98.92

Table 5: Comparison of the results achieved by naive Bayes on the PU1 corpus in Androutsopoulos et al. (2000) and by the author's implementation. Results for the latter are shown for both unigram and n-gram ($n = 1, 2, 3$) attributes. In both cases, attributes were selected by removing the 200 most frequent n-grams as well as n-grams occurring less than three times. For each cost scenario, the weight w has been selected in steps of 0.05 in order to get equal or higher precision.

Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. 1998. A bayesian approach to filtering junk E-mail. In *Learning for Text Categorization: Papers from the 1998 Workshop*, Madison, Wisconsin. AAAI Technical Report WS-98-05.

Hidden Markov Models in Spoken Language Processing

Björn Johnsson
dat171
Sveaborgsgatan 2b
21361 Malmö
dat02bjj@ludat.lth.se

Abstract

This is a report about Hidden Markov Models, a data structure used to model the probabilities of sequences, and the three algorithms associated with it. The algorithms are the forward algorithm and the Viterbi algorithm, both used to calculate the probability of a sequence, and the forward-backward algorithm, used to train a Hidden Markov Model on a set of sequences, raising the propabilites of these and similar sequences.

1 Introduction

The theories and methods of spoken language processing have evolved much over the last years and the field is one of the most interesting of computer science. This report will explain Hidden Markov Models, a graph-based data-structure used to model and calculate probabilities of sequences. The Hidden Markov Model is used in almost every speech-recognition environment.

2 The Model

2.1 The Markov Chain

The Hidden Markov Model data-structure is based on a data-structure called the Markov Chain. The Markov Chain is a directed, weighted graph, where each node contains a symbol from the output alphabet to wich it is applied, and an initial probability. The graph is complete i.e all nodes have vertices to all nodes, including itself. The weight of each vertice is the probability of a transition. Given a Markov Chain and a sequence it is then easy to calculate the probability of the given sequence by taking the product of the initial propablity of the node associated with the first symbol of the sequence and all the transition-propabilites to the nodes associated with the following symbols in the sequence. That is, for sequence S with of length n, the propabilite of sequence S is

$$prop(S) = init(S_1) * \prod_{i=2}^{n} (S_{i-1}|S_i)$$

where $(i|j)$ is the transition-probability from state i to state j.

2.2 The Hidden Markov Model

In the Hidden Markov Model there is no one-to-one relation between the alphabet and the nodes as in the Markov Chain, instead each node contains every symbol in the alphabet and relates each symbol with a probability. As a consequence there is no longer a "true" path through the graph corresponding to a sequence, as there often exists several possible paths. Therein lay one of the problems of the Hidden Markov Model, as the evaluation no longer is as simple as in the Markov Chain. In most cases is there even O^t possible ways through the graph corresponding to the same sequence, making the calculations extensive.

3 Algorithms

In the Hidden Markov Model there are basically two problems, the evaluation and the training. The evaluation problem is solved by two different algorithms giving different probabilities of a sequence. The first, the forward algorithm, gives the sum of the probabilities of all possible paths through the graph. The Viterbi algoritm gives the probability of the path with the highest probability. The training is performed by the forward-backward algorithm, wich trains the Hidden Markov Model to give a set of similar sequences a higher probability.

3.1 The Forward Algorithm

The forward algorithm calculates the probability of a sequence by adding up the sum of probabilities of all possible paths giving the right outputsequence. To do this in an easy way I first define the forward probability.

3.1.1 The forward Probability

Given a Hidden Markov Model, Φ, and a sequence X, it is possible to calculate the forward probability α. $\alpha_t(j)$ is defined as the probability of being at node j at time t giving the output in the sequence. Using a recursion this is fairly easy to calculate using the formula:

$$\alpha_t(j) = \left[\sum_{i=1}^{N} \alpha_{t-1}(i)a_{ij}\right] b_j(X_t)$$

$$\alpha_1(i) = \pi_i b_i(X_1)$$

3.1.2 The Final Step

Having calculated the forward probability the sum of all probabilities is easy to calculate.

$$P(X|\Phi) = \sum_{i=1}^{N} \alpha_T(i)$$

This gives that $P(X|\Phi)$ is the sum over all nodes, of being at that node after sending out the output given in the sequence, ie the sum of the probabilities over all possible path in the graph.

3.2 The Viterbi Algorithm

The Viterbi algorithm calculates the probability and the nodesequence of the most likely traversion through the graph giving the expected outputsequence. This is implemented in my Hidden Markov Model but I have not investigated it more closely as I rarely use it.

3.3 The Forward-Backward Algorithm

The forward backward algorithm trains the Hidden Markov Model by raising the probability of the given sequence and thereby all sequences similar to this. To calculate the new values for a and b, labeled $\hat{a}$ and $\hat{b}$, I first define two new probabilities, the backward probability, β, and the transition probability, γ.

3.3.1 The backward probability

Just as I defined the forward probability, I can also defin the backward probability $\beta_t(i)$ that is, the probability that after being at node i at the time t, the model outputs the sequence $X_{t+1}...X_T$. Similar to the forward probability, this can be calculated using a recursion as follows.

$$\beta_t(i) = \left[\sum_{j=1}^{N} a_{ij}b_j(X_{t+1})\beta_{t+1}(j)\right]$$

$$\beta_T(i) = \frac{1}{N}$$

3.3.2 The transition probability

The transtion probability $\gamma_t(ij)$ is the probability of taking the transition from node i to node j at time t given a Hidden Markov Model and an outputsequence. Using the prior defined forward and backward-probabilities it can be calculated as follows.

$$\gamma_t(ij) = \frac{\alpha_{t-1}(i)a_{ij}b_j(X_t)\beta_t(j)}{\sum_{k=1}^{N}\alpha_T(k)}$$

This is interpreted as the forwardprobability of being at node i after giving the output $X_1...X_{t-1}$ multiplied by the probability of taking the transition to node j and there give the output X_t multiplied by the probability of going from node j and give the output $X_{t+1}...X_T$ and dividing the hole product by the sum of all possible paths giving that outputsequence.

3.3.3 Calculating $\hat{a}$

To calculate $\hat{a}_{ij}$, the new values meant to replace the prior value at a_{ij},Itake the sum of all transitions between node i and node j, at all possible times t and divide it with the sum of all possible transitions from node i at all possible times t.

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T}\gamma_t(ij)}{\sum_{t=1}^{T}\sum_{k=1}^{N}\gamma_t(ik)}$$

3.3.4 Calculating $\hat{b}$

It is possible to calculate $\hat{b}_j(k)$ in a similar way as the sum of all transitions to node j if $X_t = O_k$ and dividing it by the sum of all transitions to node j.

$$\hat{b} = \frac{\sum_{t \in (x_t = O_k)} \sum_{i=1}^{N} \gamma_t(ij)}{\sum_{t=1}^{T} \sum_{i=1}^{N} \gamma_t(ij)}$$

3.3.5 Trainging on a set of sequences

These calculations can easy be extended to train on a set of sequences instead of a single sequence. To do soIadd another dimension to the γ so that $\gamma_t^m(ij)$ is the probability of going from node i to node j at time t given the m'th sequence of the trainingset. he calculations are still the same. Howether, the calculations for $\hat{a}$ and $\hat{b}$ has to be changed, so they are based on the hole set. $\hat{a}$ and $\hat{b}$ should instead be calculated as follows.

$$\hat{a}_{ij} = \frac{\sum_{m=1}^{M} \sum_{t=1}^{T} \gamma_t^m(ij)}{\sum_{m=1}^{M} \sum_{t=1}^{T} \sum_{k=1}^{N} \gamma_t^m(ik)}$$

$$\hat{b} = \frac{\sum_{m=1}^{M} \sum_{t \in (x_t = O_k)} \sum_{i=1}^{N} \gamma_t^m(ij)}{\sum_{m=1}^{M} \sum_{t=1}^{T} \sum_{i=1}^{N} \gamma_t^m(ij)}$$

3.3.6 Post calculations

After calculating $\hat{a}$ and $\hat{b}$ you just exchange a and b by them. In theory. On computers this will be a problem as almost all evaluation then will become zero as the values go beyond the reach of a double floating point variable. To cope with that, the easiest way is to define that the values in a and b are not allowed to go beneath a value, in my case 1e-20. As computers calculating values of this size almost always does errors I normalize the values in a and b so that the probabilities sum up to one.

3.3.7 Result of the forwardbackward algorithm

Using this algorithm on fairly large trainingset, it can train a Hidden Markov Model in about five iterations and after that recognize sequences similar to those in the trainingset.

4 Appendix

4.1 Notation

The following notations are used in the calculations:
N is the number of nodes in the Hidden Markov Model
O is the number of symbols in the outputalphabet
T is the number of symbols in a particular outputsequnce
a_{ij} is the transitionprobability between node i to node j
$b_i(x)$ is the probability of the output x at node i
X_t is the output at time t in sequence X
π_i is the initial probability of node i, ie the probability that the sequence starts at node i
$\alpha_t(i)$ is the forward probability defined in chapter 3.1.1
$\beta_t(i)$ is the backward probability defined in chapter 3.3.1
$\gamma_t(ij)$ is the transition probability defined in chapter 3.3.2
$\hat{a}_{ij}$ is the new transitionprobability used temporary in the training, defined in chapter 3.3.3
$\hat{b}_{ij}$ is the new outputprobability used temporary in the training, defined in chapter 3.3.4
M is the number of sequences in a trainingset

4.2 The use of Hidden Markov Models in Speech Recognition

Speech and sound is in computers often represented as PCM data or Pulse CodeModulation. It is a sequence of values representing the porition of the membran on either the speaker or the microphone. Unfortunatly this sequence contains to much data to be evaluated by a Hidden Markov Model so it has to be transformed before it is used this data structure. Usually the PCM data is divided in frames of approxomatly 20 milliseconds and each frame the converted to single number or a small vector of numbers. There are multipla ways to do this step.

- Fast Fourier Transfom
  A Fast Fourier Transform, or a FFT, is a fast but inaccurate way of calculating the energy of the sound at different frequences. Picking the numbers of the right frequences, this is a good way of turning the PCM into a useful value.
- Linear Prediction Coding
  The Linear Prediction Coding, or the LPC,

is a way of creating a polynomial that, given the prior values of the data, try to predict the coming value.

- Energy
  Adding up the absolute value of the PCM data gives the total energy of a frame. This is a useful as such, but if one take the difference between the frames insted hte values give a better representation of the sound. Especially if all the frames energies are subtracted with the energy of the frame with the highest energy, thus removing the possible error of different recording volumes.

After this is done, the values are made discrete using a codebook to spread the values over all the discrete symbols in the output alphabet. This gives us a sequence of symbols, in my case, integer between 0 and 255. It is then possible to do this for several recordings of the same command, getting a set of sequences that can be used to train a Hidden Markov Model. The Hidden Markov Model can the be used to calculate the probability of sequence extracted from a sound recording i the same way as the trauning set, and using a treshold, determine whether the sound was the command or not. Using several Hidden Markov Models, one per command, this can be used to controll a computer using speech commands.

5 References

Huang, Acero and Hon. 2001. Spoken Language Processing, A Guide to Theory, Algorithm, and System Development. Prentice Hall PTR. ISBN 0-13-022616-5

Grammar Checker

Markus Malmsten
Department of Computer Science
Lund Institute of Technology
Sweden,
e99mm@efd.lth.se

Simon Klasén
Department of Computer Science
Lund Institute of Technology
Sweden,
e99sk@efd.lth.se

Abstract

The goal with our project was to implement a grammar checker prototype. The work was influenced by the paper intelligent writing assistance (Heidorn, 2000), which describes the Microsoft word grammar checking technique. Our implementation uses a Perl script for text formatting and the Charniak parser for part of speech and syntactic tagging. The analyzing part was implemented in java.

1 Introduction

The goal with our project was to implement a grammar checker prototype. The purpose with a grammar checker is to check a text for grammatical errors that a grammar book would discuss. A grammar Checker can also include support for style checking (good writing style), but this is not part of our system.

One of the first widely used grammar checker was Writers Workbench (Macdonald et al., 1982) which was developed for Unix systems about 25 years ago. Today the built-in grammar checker in Microsoft Word probably is the most widely used one. It is based on the work that was started by the natural language processing (NLP) group at Microsoft Research in 1992. Our work was influenced by the paper intelligent writing assistance (Heidorn, 2000) which describes the Microsoft Word grammar checking technique.

The biggest difference between the Microsoft Word and our solution is that Microsoft Word is a total solution where all the necessary parts for grammar checking are all built in, while in contrast our solution is divided into three main steps. The steps are text formatting, parsing and analyze, which is taken care of by different tools.

To start with we must tag the input data(the text that should be analyzed); this is done by a simple Perl script that just delimits the sentences with a tag which makes the text ready for parsing by the Charniak Parser.

The Charniak parser takes the tagged input data and performs part of speech and syntactic tagging based on the Penn Treebank (Marcus et al., 1993) tagset. The result is a parse-tree delimited by parenthesis.

Finally is the rule-based analyzing part implemented in Java. We also developed a simple GUI which simplifies the usage of the system. Basically there is one input area for a Charniak parse-tree and an output area that displays the original text including suggested corrections.

2 Implementation

2.1 Overview

We have used three programs in our project; one Perl-script for formatting the original text, one parser that produces a tagged tree and our own Java program which reads the output from the parser, builds a tree and applies the implemented rules and presents the result in a GUI.

2.2 Perl-script

The Perl-script delimits each sentence by adding the <s> ... </s> tags. This format is required by the Charniak parser.

2.3 Charniak parser

We chose the Charniak parser to do the part-of-speech and syntactic tagging. Collins was the other suggested parser but that was neglected due to its much longer running time and the fact that it was 10 times larger. The Charniak parser takes the delimited sentences supplied by the Perl-script and produces a tree in text form where the branches and nodes are enclosed by parentheses.

2.3.1 Penn Treebank style

The tagset used by the Charniak parser is the one constructed by the Penn Treebank project (Marcus et al., 1993), which is a large annotated English corpus. The Penn Treebank tagset is

based on the pioneering Brown Corpus which consists of 87 tags. Other tagsets uses up to around 200 tags. The Brown tagset was however pared down considerably. A key strategy in reducing the tagset was to eliminate redundancy by taking into account both lexical and syntactic information. The resulting tagset consists of 48 part-of-speech (see Appendix A, table 1) tags and 14 syntactic tags (see Appendix A, table 2).

2.4 Grammar Checker

2.4.1 System

Our program consists of 3 classes and one main-method. The main-method creates an instance of a GUI-object which includes two event handlers. The event handlers are bound to buttons, one for choosing a file and the other for executing the implemented rules.

It is also in this event handler that the tree is constructed through the method buildTree() in the class CorpusHandler. This function returns a PennNode-object which is the root of the tree. By invoking methods on this root node different rules can be applied and the modified content can be requested which is then displayed.

2.4.2 Building the tree

The program starts with storing the text produced by the Charniak parser in a string. The parentheses structure of the string is then used to decide when new nodes should be created and what they should contain. When a left parenthesis encountered, a new child is created and it becomes the current node. The tag type for the new node is the following word. There are now three possibilities; if the next character is a left parenthesis then a new node is created as above, if its a right parenthesis then this closes the node and the parent node is set to be current node and if its neither of these then the word is the content of the node i.e. a word in the input sentence. In each node we also store which depth in the tree it is in. This can then be used in the search algorithms.

2.4.3 Rules

The rules are applied on each sentence and are recursive. A finite state machine is used to keep track of what to search for and when a correction should be suggested. A special self-constructed node type, FLAG, is inserted if an error is found and it contains a text explaining to the user what can be corrected. Our rules are applied sequentially but they do not affect each other. However the inserted FLAG-nodes must be taken into consideration during the implementation of further rules.

2.4.4 GUI

There are two events that can be triggered in our program. First a file can be chosen by clicking the File-button and secondly the Submit-button which creates tree, runs the algorithms on it and finally prints the resulting tree and text to the lower window. You can chose which rules you want to apply by using the checkboxes at the top. There is also an option to hide the tree structure. See Apendix B for a screenshot of the GUI when the system analyzes a simple sentence.

3 Evaluation

3.1 Testing

Testing of the rules was done in parallel with program construction, one rule at a time. Our initial test samples were just single sentences which had the errors that a specific rule should trig on. After some modifications of the rules, the system was behaving as expected for the single sentences. Everything seemed to work fine.

The real problems started when we tried a larger test corpus. It turned out our rules were to general and was trigged not only when there was an error, but also when the sentences were grammatically correct. The main reason for the behavior was that we were analyzing too small parts of the sentences; we focused at the part of speech tags. This was solved by adding a depth value to all nodes in our parse tree, which enabled us to adjust the rules in respect to bigger text blocks, e.g. subordinate clauses.

The size of the test data was not large enough to make any statistic analyses of the system accuracy. A randomly chosen corpus downloaded from internet indicated the system to work correct, finding the errors and in the same time not trigging on any correct sentence.

4 Conclusions

Even though our grammar checker was inspired by the one found in Microsoft Word, the methods are not the same. The one found in Microsoft Word utilizes a recursive algorithm applied to top nodes. Depending on the type of the node, it applies the subset of rules that are applicable for that type of top node. Our approach on the other hand uses a finite state machine that steps through the text word by word,

in order. To determine whether we still are inside the same clause we use the depth attribute. We check one rule at a time, sequentially, which means we keep one object containing the current state of our search. This object differ depending on the rule, and the type of node we search for depend on the current state. When the final state is reached and the requirements are met, a "FLAG"-node is inserted and the state machine is reset, or set to a specific state, depending on the rule. Our approach is of course more expensive, but for our purpose if was sufficient and resulted in code that is easier to understand and maintain.

A problem we encountered that would have made it even more difficult to utilize the Microsoft Word approach was that for incorrect sentences the whole structure of the tree produced by the Charniak parser was altered. This means that it is not sufficient to look at a correct sentence for patterns to which to look for. We got around some of those issues by looking at the text in order.

References

George E Heidorn. 2000. Intelligent writing assistance.

NH Macdonald, LH Frase, P Gingrich, and SA Keenan. 1982. The writer's workbench: Computer aids for text analysis.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of english: The penn treebank.

Appendix A

1.	CC	Coordinating conjunction	25.	TO	*to*
2.	CD	Cardinal number	26.	UH	Interjection
3.	DT	Determiner	27.	VB	Verb, base form
4.	EX	Existential *there*	28.	VBD	Verb, past tense
5.	FW	Foreign word	29.	VBG	Verb, gerund/present perticiple
6.	IN	Preposition/subord. conjunction	30.	VBN	Ver, past participle
7.	JJ	Adjective	31.	VBP	Verb, non-3rd ps. sing. present
8.	JJR	Adjective, comparative	32.	VBZ	Verb, 3rd ps. sing. present
9.	JJS	Adjective, superlative	33.	WDT	*wh*-determiner
10.	LS	List item marker	34.	WP	*wh*-pronoun
11.	MD	Modal	35.	WP$	Possessive *wh*-pronoun
12.	NN	Noun, singular or mass	36.	WRB	*wh*-adverb
13.	NNS	Noun, plural	37.	#	Pound sign
14.	NNP	Proper noun, singular	38.	$	Dollar sign
15.	NNPS	Proper noun, plural	39.	.	Sentence-final punctuation
16.	PDT	Predeterminer	40.	,	Comma
17.	POS	Possessive ending	41.	:	Colon, semi-colon
18.	PRP	Personal pronoun	42.	(	Left bracket character
19.	PP$	Possessive pronoun	43.	)	Right bracket character
20.	RB	Adverb	44.	”	Straight double quote
21.	RBR	Adverb, comparative	45.	‘	Left open single quote
22.	RBS	Adverb, superlative	46.	“	Left open double quote
23.	RP	Particle	47.	’	Right close single quote
24.	SYM	Symbol (mathematical or scientific)	48.	”	Right close double quote

Table 1: The Penn Treebank POS tagset

	Tags	
1.	ADJP	Adjective phrase
2.	ADVP	Adverb phrase
3.	NP	Noun phrase
4.	PP	Prepositional phrase
5.	S	Simple declarative clause
6.	SBAR	Clause introduced by subordinating conjunction or *0* (see below)
7.	SBARQ	Direct question introduced by *wh*-phrase
8.	SINV	Declarative sentence with subject-aux inversion
9.	SQ	Subconstituent of SBARQ excluding *wh*-word or *wh*-phrase
10.	VP	Verb phrase
11.	WHADVP	*Wh*-adverb phrase
12.	WHNP	*Wh*-noun phrase
13.	WHPP	*Wh*-prepositional phrase
14.	X	Constituent of unknown or uncertain category
	Null elements	
1.	*	"Understood" subject of infinitive or imperative
2.	0	Zero variant of *that* in subordinate clauses
3.	T	Trace-marks position where moved *wh*constituent is interpreted
4.	NIL	Marks position where preposition is interpreted in pied-piping context

Table 2: The Penn Treebank syntactic tagset

Appendix B

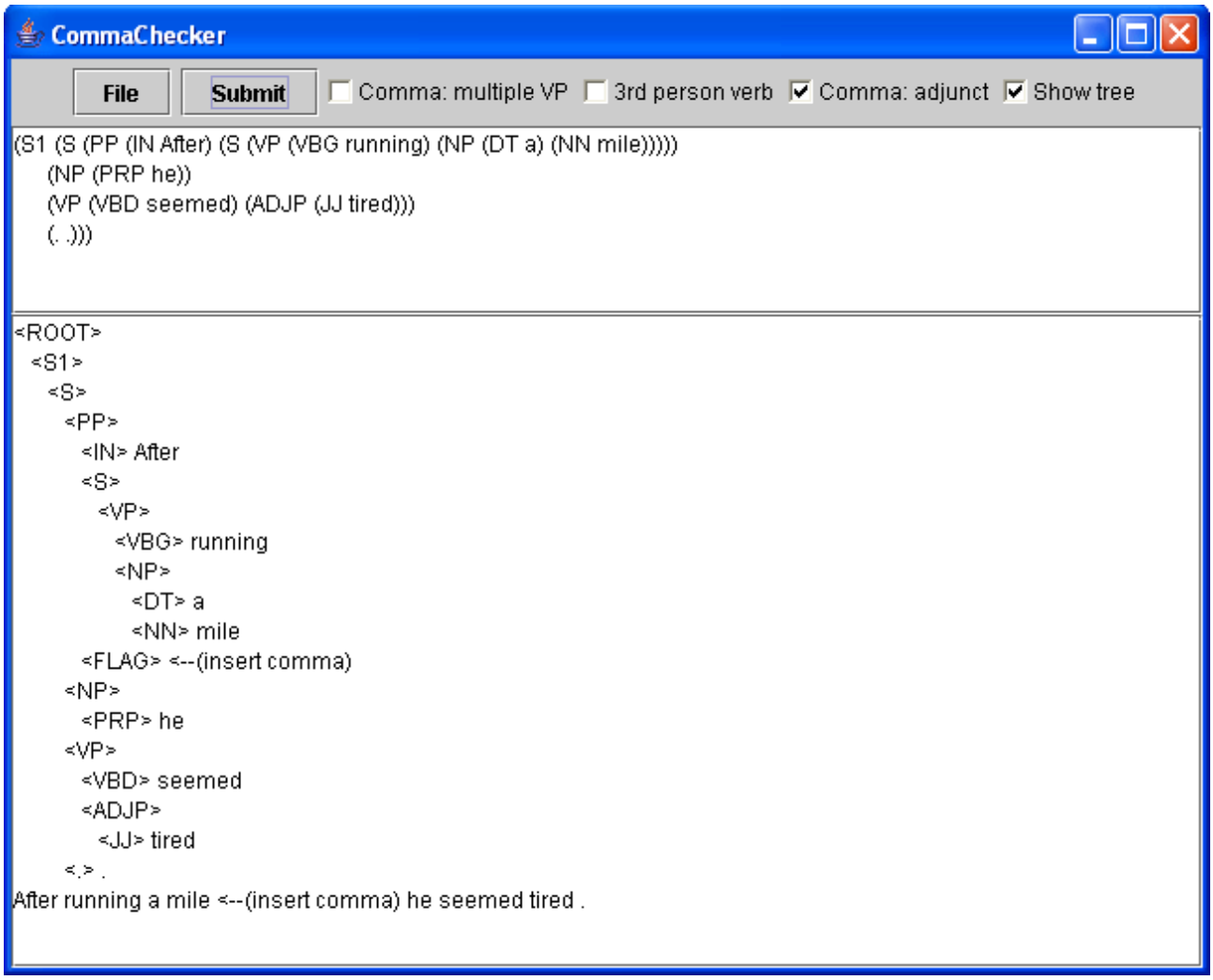


Image 1: Screenshot from the Java GUI.

Automatic learning of dependency rules from corpora

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

Tomas Rutegård e99, Bibi Sandberg e00 and Johan Larsson dat00

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 2004. The work develops and assesses a new dependency parser for Swedish, based on decision trees learned from corpora. To assess the parser, it is trained and tested on a subset of the MALT corpus and found to perform fairly well considering its stage of development.

1 Introduction

1.1 Project purpose

The given purpose of the project presented in this paper was to construct a software system for the automatic learning of grammar rules used by a certain text parser: the Nivre parser. The Nivre parser is a dependency parser developed by Joakim Nivre [1]. The Nivre parser uses a special kind of D-rules, namely directed D-rules, for parsing.

As it happened, we, the authors of this paper, decided to instead develop our own parser, a parser also using rules derived from corpora, but rules of a different kind, used in a different way. The automatic learning algorithm of our parser is decision tree induction algorithm, simple yet not powerless.

2 Short introductions of theory

2.1 Dependency Grammars

According to the book *An Introduction to Language Processing with Perl and Prolog* [3] dependency grammar is used for describing the structure of a language. It is especially good for describing languages where the word order is flexible. This is the case for Latin and Russian but not for English and Swedish. The rules can be helpful in translations or just for understanding the language.

Every word in a sentence is the dependent of one head with one exception. This exception is the head of the sentence, also called the root, which only have dependants. The head of the sentence is generally the main verb but can also in rare cases be a noun. These dependency rules are marked as arrows from the dependant to the head. The root is marked with an arrow pointed at the top of the screen.

The basic dependency rules are that a dependant links to its noun and a subject noun links to its main verb. Other rules are that determiners and adjectives are dependents to their noun and adverbs to their adjectives. One example is Figure 1.

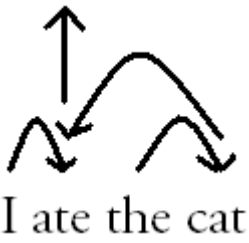


Figure 1. *An example of how dependency grammar can look like*

Here *"ate"* is the main verb and also called the root. The two nouns *"I"* and *"cat"* are the dependants of *"ate"* and the determiner *"the"* is the dependant of its noun *"cat"*.

2.2 Decision tree induction

2.2.1 The decision tree structure

Consider some object or situation to which some set of attributes is related. A decision

tree is a structure associating with each possible set of values of the attributes some value, thus constituting a function from the set of possible sets of attribute values to an arbitrary set. If the set of values associated with possible sets of attribute values is discrete, the values associated with possible sets of attribute values are called classifications.

A decision tree is a tree data structure. Each non-leaf node of a decision tree represents a test of one of the attributes of the attributes set and each outbound branch from a non-leaf node represents one of the possible values of the attribute tested in that node. Each leaf node of a decision tree represents a value assigned to some subset of the set of possible sets of attribute values. The value associated by a decision tree with a given set of attribute values is the value represented by the leaf reached when traversing the tree from root to leaf, in each node choosing branch according to the value of the attribute tested in that node.

Decision trees are simple yet somewhat expressive. Any Boolean function can be written in the form of a decision tree, though by necessity some, for example the majority function, are quite large in the form.

2.2.2 The induction algorithm

Let an example of a function be a member of the domain of the function and the associated member of the codomain of the function. A set of sets of attribute values and values associated with these sets of attribute values can be regarded as a set of examples of some function whose domain comprise the attribute values and whose codomain comprise the associated values. The forming of a function consistent with the examples approximating the function exemplified is referred to as inductive inference. The formed function is called a hypothesis.

The algorithm in Figure 1 [2] forms a consistent hypothesis in the form of a decision tree for any set of examples which are examples of a classifying decision tree or some function that can be written in the form of a classifying decision tree. In Figure 2, the goal predicate is an attribute whose value for any set of attribute values is the value associated with that set of attribute values. The algorithm applies Ockham's razor and prefers the simplest hypothesis of a set of hypotheses all consistent with the examples. Forming the smallest consistent hypothesis is an intractable problem, though. The algorithm forms a smallish one. CHOOSE-ATTRIBUTE chooses the attribute which provides the most information, in the mathematical sense.

```
function DECISION-TREE LEARNING(examples, attributes, default) returns a
decision tree
        inputs:     examples, set of examples
                    attributes, set of attributes
                    default, default value for the goal predicate
        if examples is empty then return default
    else if all examples have the same classification then return the
classification
    else if attributes is empty then return MAJORITY-VALUE(examples)
    else
        best ← CHOOSE-ATTRIBUTE(attributes, examples)
        tree ← a new decision tree with root test best
        m ← MAJORITY-VALUE(examples_i)
        for each value vi of best do
            examples_i ← {elements of examples with best = v_i}
            subtree ← DECISION-TREE-LEARNING(examples_i,
        attributes – best, m)
            add a branch to tree with label v_i and subtree subtree
        return tree
```

Figure 2. *An algorithm forming a smallish decision tree consistent with a set of examples.*

3 The parser

3.1 The decision trees generated and the parser algorithm

This section presents the parsing algorithm we have made. It uses four different decision trees in order to classify the correct part-of-speech tags in a sentence.

We will first describe how the decision trees have been generated.

3.1.1 Is-root

This tree is used to find the most probable root in the sentence. It has learned by looking at the part-of-speech tag and a context of two words on each side. In the example shown in Figure 3, a correct classified sentence is shown. The word ”är” is root in the sentence. So for every word in the sentence, we will add its part-of-speech tag and its context to the database together with a true/false that tells if its the root. Later in the parsing algorithm, the decision tree tries to classify when something is root and not.

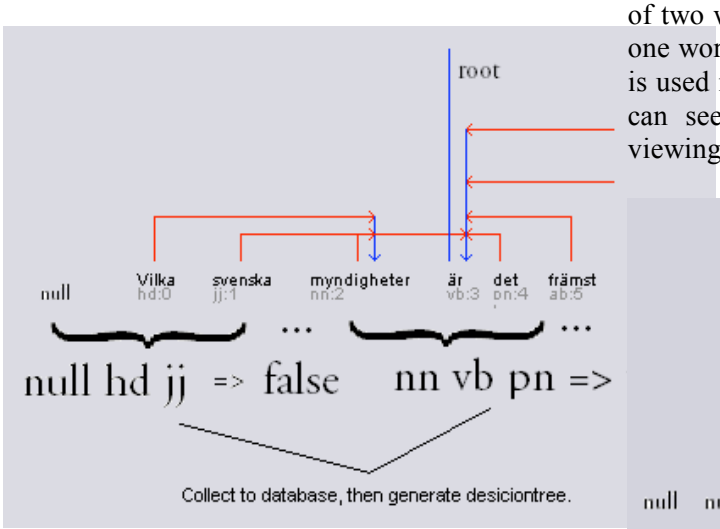


Figure 3. *Is-root*

The tree classifies 80% of all sentences (on an unseen domain) correct. When it fails, there are more than one candidate to be root (the most likely will be chosen).

3.1.2 Find-head-1

This decision tree is trained by taking the part-of-speech (pos) tag and a context of one word on each side, and trains it to find the correct head to the pos-tag. The example in Figure 4 shows how the selection has been made. This is added for every word except root (since it doesn't have any parent).

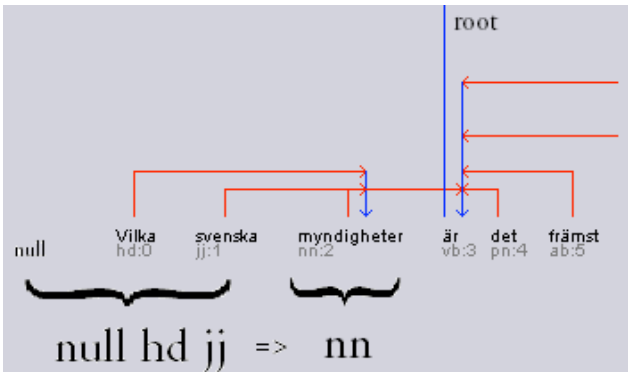


Figure 4. *Find head 1*

3.1.3 Find-head-2

This is an extension of Find-head-1 that is trained by looking at the pos-tag, a context of two words on each side, plus a context of one word on each side of the head. How this is used in practice is explained later, but you can see what the database looks like by viewing the example in Figure 5.

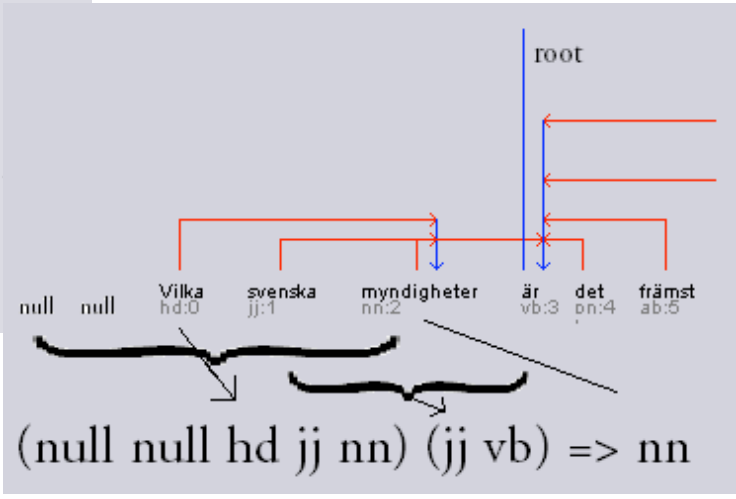


Figure 5. *Find head 2*

3.1.4 Find-Direction

The database that is used to train this tree looks almost the same as the one used to train Find-head-2. In addition it also has the direction of the arrow (right/left) that is used as goal-attribute in the decision tree learning algorithm. This makes it able to classify the most probable direction.

3.2 The parsing algorithm

This algorithm is used to demonstrate how we can use decision trees to classify all words/pos-tags in a sentence. It uses the four decision trees explained above. In practice it works in three different phases. It is at the moment quite simple, and could easily be expanded to use more decision trees and more advance functions. The following explains the three phases of the algorithm.

3.2.1 Phase 1

The first phase uses only the *find-root* and *find-head-1* trees described above. Its main goal is to find the most probable root, and all possible candidates for head for all words. The easiest way to show how this works is by using an example.

Lets say we want to find the head for the Swedish word "inkomsterna" (see Figure 6), this is a noun, and its closest neighbors is a determiner and a verb. We can ask *find-head-1* with this information, and it will answer by providing a list of the most probable neighbors, in this case verb (56.5%) and preposition (43.5%). The next thing we do is to go though all the words looking for prepositions and verbs, we add this to a list of *potential* parents. In this sentence the preposition at position zero, the verb at position three and seven would be added to the list of potential parents for our noun. We do this for all the words in the sentence.

Figure 6.

3.2.2 Phase 2

We now have a list of head-candidates for every word except the most probable root. The next step is to go through all the candidates in the list (and we do this for every word) and ask *find-head-2* and *find-direction* how probable that candidate is. Since *find-head-2* is more accurate, its result gets higher ranked when choosing the candidate. In the example above, the preposition at position zero would get a score of 105, the verb at position three a score of 6.3 and the verb at position seven a score of 1.8. This means we will chose the preposition at position zero as our most probable head.

3.2.3 Phase 3

After phase 2, the algorithm is almost complete, however there may still be easy found errors in the complete graph. For example, we know that two arrows may never cross each other (see Figure 7 "A crossing link"). If they do, at least one of them is pointing wrong. We use a simple method to find and remove all crossing arrows.

We also look for cycles in the graph (see Figure 8 "A cycle"), and use a method to break these. This step is not optimized, but works quite well.

Figure 7. *A crossing link*

ycle

3.2.4 The complete graph

On this page the program and its GUI is explained, and an example of a complete graph is shown in Figure 9.

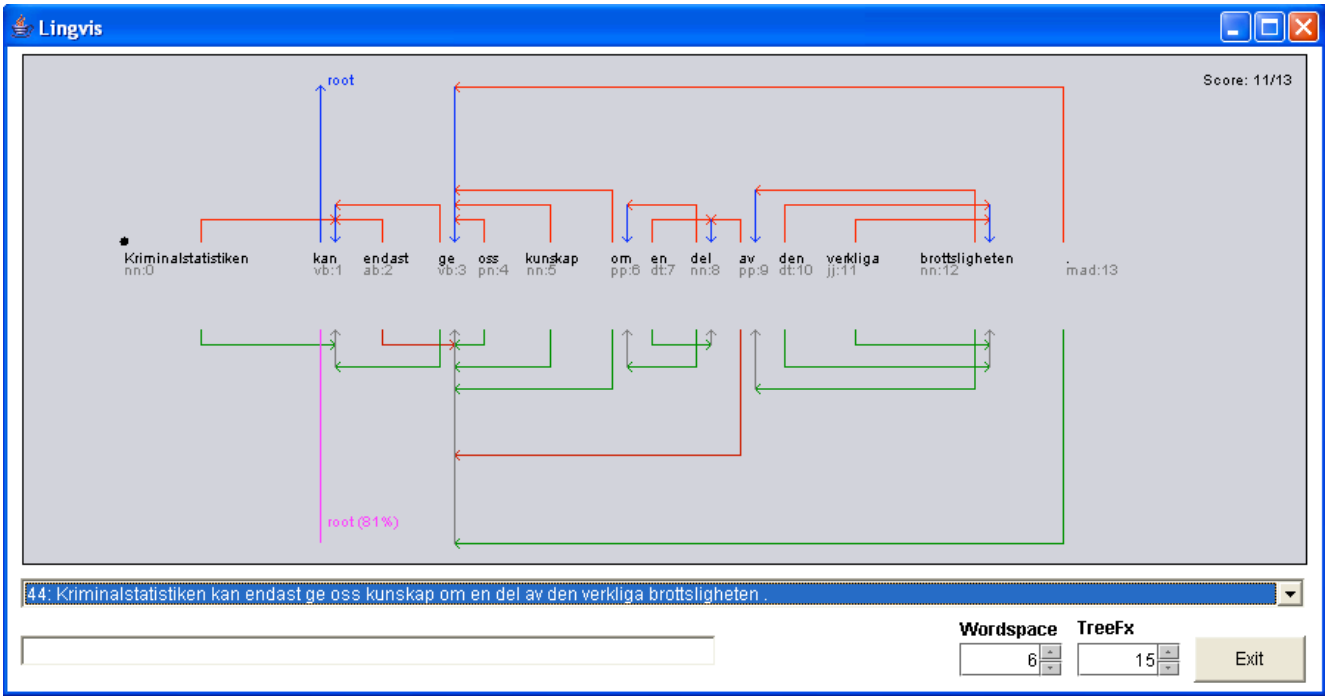


Figure 9. *The complete graph*

3.3 The program and how to use it.

This is what the GUI for our program looks like. In the list a sentence from the loaded XML file can be chosen. When loading a sentence to classify, the correct answer is shown at the top (red and blue arrows), and the result of our algorithm is shown at the bottom (green arrows for correct classifications, and red arrows for errors). So in this sentence, 2 errors are present ("endast" -> "ge", and "av" <- "ge"). There is also another window (not showing here) that is used as log-tool. If we press a word, for example "av" the following info will show in the log-tool.

The first row in Figure 10 shows the chosen form, in this case "av". The next tells us what "av" links to (in this case "ge") and the "x-link: false" says its not crossing any other arrow. The tree shows all children (and their children) and also tells us if there are any cycles present. The next list shows all potential parents for "av". As you can se, the verb at position 3 has gotten the highest score together with the verb at position 1). The noun at position 8 (the correct one) has gotten just slightly less score. The "pr: false" tells us if the arrow goes past the root, in that case, it will get a penalty since its quite unlikely. The last list (rules) shows a more a abstract view of likely parents, telling us its is most likely to link to a verb.

```
Form: av
Links to: ge (x-link: false)
-tree----------------------------------------
<pp:9>
   <nn:12>
      dt:10
      jj:11
-----------------------------------------------
Pot.Parents: pp:9
   vb:3 (106.33559) (pr: false)
   vb:1 (106.33559) (pr: false)
   nn:8 (105.58499) (pr: false)
   nn:12 (61.73145) (pr: false)
   nn:5 (5.5849915) (pr: false)
   nn:0 (0.60047567) (pr: true)
-rules---------------------------------------
vb (56.89655)
    left (100.0)
nn (41.37931)
    left (99.60318)
    right (0.39682543)
rg (1.7241379)
    right (100.0)
```

Figure 10.

4 Performance analysis and conclusions

4.1 The performance of the program

The program use approximately 800 sentences as a practice set and about 300 sentences as a test set. Best results are given when the program is training and testing on the same set of sentences i.e. the test set is 300 sentences that are selected out of the 800. In 300 sentences there are approximately 4000 arrows (there are approximately 8700 arrows in 800 sentences) and the program gets about 86.7 % of them correct and 73.5 % correct if the test set is new to the program. Sentences where all the arrows are correct is about 37.6 % when the program is training and testing on the same set and 19.3 % else. The training set was later expanded to 5000 sentences and the test set to 1300 sentences. But the result turned out to decrease. When testing on new sentences it gets about 70 % of the arrows correct which is a decrease by 3.5%. This probably has to do with the decision trees that get overfitted. A solution to this problem could be to improve the pruning of the trees.

4.2 Difficulties on the way

We started to use the neighbors of the dependant word to improve the program. Two neighbors to the right and two to the left were showing to be the most efficient so far. After this improvement we also added neighbors to the head word. The statistic improved by roughly 15 %.

To find the correct root in the sentence was harder then first expected. And when the program choused the wrong word as the root it generated more errors to the other arrows in the sentence. Improvements can still be done here.

Another problem that maybe is not that important, but contributes to lower the statistics, is that the arrow from the dot in the sentence more often is wrong than right.

In the beginning the program sometimes made loops which are not acceptable in dependency grammar. To solve the problem temporary an algorithm was made. The algorithm solves the problem with the loops (Figure 2) but not in the most efficient or most correct way. Here big improvements can be done.

Figure 2. *An example of a loop*

Crossing arrows was also a problem in the beginning but was easily fixed with a small algorithm.

4.3 Future aspects, if more time was given

To find a better way to classify the root is a good start. The root is in most cases the main verb and to locate that easier it maybe would help to add some more attributes to the words. Such as how the verb is intransitive, transitive or ditransitive.

An improvement of the loop algorithm could be done by making the algorithm go trough all possible arrow changes before choosing. At the moment the algorithm picks out the first possible fit.

To examine how much improvement a larger training set will give has not given as good results as hoped for. This is because of the decision trees that get overfitted. A solution to this problem is maybe to get better pruning of the trees.

At the moment only the lexical categories are used as attribute to the words. To improve the program it could also examine how the most common words in a sentence relate to the other words. For example instead of using the attribute determiner to a very common word like *"the"* you could use the fact that the word *"the"* in most cases has the first noun to its right as the head.

To try the program on other languages would be an interesting project and not so hard to accomplish. The program is capable to read xml documents in a certain

predefined type and if given, it could train and test on a different xml database in a new language.

5 References

[1] Nivre, J. (2003) An Efficient Algorithm for Projective Dependency Parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 03)*, Nancy, France, 23-25 April 2003, pp. 149-160.

[2] Russel, S, Norvig, P. (2003) Artificial Intelligence: A Modern Approach. Second edition. Prentice Hall series in artificial intelligence.

[3] Nugues, P. (2004) An Introduction to Language Processing with Perl and Prolog. Lecture notes for the course Language Processing and Computational Linguistics given at Lund School of Technology.

Part-of-Speech Tagging Using the Brill Method

Maria Larsson and Måns Norelius
Lund Institute of Technology
Lund, Sweden
d00ml@efd.lth.se, d00mno@efd.lth.se

Abstract

Part-of-speech tagging is the process of associating each word in a text with it's part-of-speech category and possibly a set of morphosyntactic features. This information is represented by part-of-speech tags. This paper describes an implementation of a part-of-speech tagger for Swedish based on the Brill method. The basic idea is to apply a set of rules to an initial annotation achieved using a simple algorithm. The rules are found using transformation-based learning applied to a manually tagged training corpus. The paper also addresses the problem of tagging unknown words, i.e. words that don't appear in the training corpus.

1 Introduction

1.1 Part-of-Speech Tagging

The first step in implementing a part-of-speech tagger is to build a lexicon, where the part-of-speech of a word can be found. Unfortunately many words are ambiguous, and each word can therefore have several classifications. As an example, the word "note" can be either a noun or a verb. It is the object of the part-of-speech tagger to resolve these ambiguities, using the context of the word. Another problem is the handling of words that have no entries in the lexicon.

There are basically two approaches to part-of-speech tagging: rule-based tagging and stochastic tagging. This paper describes an implementation using the rule-based approach, where the rules are generated using transformation-based learning.

1.2 Transformation-Based Learning

Transformation-based error-driven learning is a machine learning method typically used for classification problems, where the goal is to assign classifications to a set of samples. An initial classification is produced using a simple algorithm. In each iteration the current classification is compared to the correct classification and transformations are generated to correct the errors. The output of the algorithm is a list of transformations that can be used for automatic classification, together with the initial classifier.

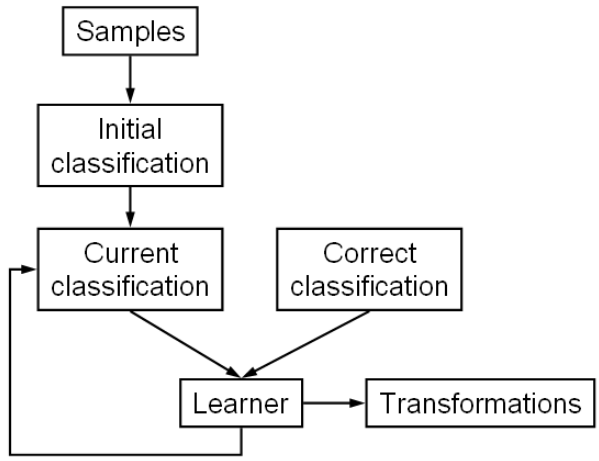


Figure 1: Transformation-based learning

There are two components of a transformation: a rewrite rule and a triggering environment. The rewrite rule says what should be done (e.g. change the class from A to B) and the triggering environment says when it should be done (e.g. if the preceding sample is of class C).

Transformation-based learning is used in many different areas, and has proven to be a very successful method in the field of natural language processing. The algorithm was introduced for POS tagging by Eric Brill in 1995.

2 Brill's Learning Algorithm

The algorithm first assigns every word it's most likely part-of-speech, i.e. the most common tag for that word. This initial annotation is compared to a hand-annotated corpus, and a list of errors is produced. For each error, rules to cor-

rect the error are instantiated from a set of rule templates. Each instantiated rule is evaluated by computing it's impact on the whole corpus. The rules are compared by assigning each rule a score, which is the difference between the number of good transformations and the number of bad transformations the rule produces. The rule with the highest score is applied to the text and added to the result list. The transformed corpus is then used to generate a new rule in the next iteration. The algorithm stops when a certain criteria has been fulfilled, e.g. the error rate is below a specified threshold. The algorithm is outlined in figure 2.

```
while (nbr of errrors > threshold)
  for (each error)
    for (each rule r correcting the error)
      good(r) = nbr of good transformations
      bad(r) = nbr of bad transformations
      score(r) = good(r) - bad(r)
  Apply the rule with highest score and
  append it to the rule list
```

Figure 2: Pseudo code for Brill's learning algorithm

2.1 Rule Templates

The learning algorithm instantiates rules given a set of templates. The rules change the tagging of a word based on the tagging of the neighbouring words, and are therefore called contextual rules. The rule templates proposed by Brill are presented below.

Change tag $\boldsymbol{a}$ to tag $\boldsymbol{b}$ when:

1. The preceding (following) word is tagged $\boldsymbol{z}$
2. The word two before (after) is tagged $\boldsymbol{z}$
3. One of the two preceding (following) words is tagged $\boldsymbol{z}$
4. One of the three preceding (following) words is tagged $\boldsymbol{z}$
5. The preceding word is tagged $\boldsymbol{z}$ and the following word is tagged $\boldsymbol{w}$
6. The preceding (following) word is tagged $\boldsymbol{z}$ and the word two before (after) is tagged $\boldsymbol{w}$

where $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{z}$ and $\boldsymbol{w}$ are tag variables.

Instantiating a rule using one of these templates means that the variables are assigned tags corresponding to the specific error to be corrected.

3 Tagging of Unknown Words

A simple way of dealing with unknown words is to assign them the most common part-of-speech and then rely on the contextual rules to correct the errors. There are however more sophisticated methods that can be used to achieve higher accuracy. Brill suggests a special set of rules to apply only to the unknown words, generated in basically the same way as the contextual rules. These rules are applied after the initial tagging and before applying the contexual rules.

3.1 Rule Templates

The rule templates used only for unknown words changes the tag based on properties of the actual word, and not based on the tagging of neighbouring words. The following list describes the templates designed by Brill.

Change the tag of an unknown word from $\boldsymbol{a}$ to $\boldsymbol{b}$ when:

1. Deleting the prefix (suffix) $\boldsymbol{x}$ results in a word
2. The prefix (suffix) of the word is $\boldsymbol{x}$
3. Adding the prefix (suffix) $\boldsymbol{x}$ results in a word
4. The preceding (following) word is $\boldsymbol{w}$
5. The character $\boldsymbol{c}$ appears in the word

where $\boldsymbol{x}$ is a string of length 1 to 4.

4 Implementation

4.1 Corpus

The corpus used is the SUC 1.0 corpus, which was developed as part of a joint research project between the Departments of Linguistics at Stockholm University and Umeå University respectively. The POS tags used in the implementation are identical to the tags used in the SUC project.

4.2 Previous Work

As a project for last year's course a Java implementation of the Brill tagger was written by François Marier and Bengt Sjödin. It contained the basic Brill algorithm and the contextual rule templates, but was too slow for practical use. This program has worked as a foundation for

our implementation. Shorter running time and handling of unknown words are the main improvements in this year's implementation.

4.3 Optimization

Because of unexpected low accuracy of the implemented tagger, it was suggested that the program of last year might contain a bug. A set of tests was performed, which resulted in the conclusion that the program worked correctly. The reason for the bad results was instead that the training algorithm was too slow. It was only able to work on small texts, which could not be expected to generate very good results. Our first goal was therefore to optimize the program to be able to run it on a large training corpus.

4.3.1 Problems Identified and Solved

After analyzing the code and extracting profile information, we were able to identify the main reasons for the slow running time of the learner. First we found that a large percentage of the running time was spent on string comparisons. This was because the POS tags were represented by strings, and the algorithm contains many comparisons between tags. The problem was solved by representing the tags by integers, which are faster to compare, and introducing a new class to handle the translation.

Secondly, we discovered that the rule evaluation was done in an inefficient way. Rules were evaluated by actually applying them to the text and then counting the number of errors in the resulting tagging. This meant that the whole corpus had to be copied for every rule evaluation. In our implementation we count the number of good and bad transformations without applying it. Only the best rule is applied and hence there is no need for copying.

Finally, a number of rules were instantiated in linear time with respect to the number of words in the training corpus. This had a great effect on the total time complexity, and increased the running time when using larger texts. In the final implementation all rules are instantiated in constant time.

To further improve the time complexity, all rules are generated in the beginning of the program, and not in each iteration. This means we only consider rules that corrected errors in the original tagging. Consequently, there is a risk that selected rules introduce new errors that no rule is able to correct. However, the output of the algorithm was not affected by this change, which indicates that these missed rules should have been disgarded anyway.

4.3.2 Test Results

In order to demonstrate the process of the optimization, a comparison has been made between the running times of the learning algorithm of the original program and three optimized versions. The versions have an increasing level of optimization and the main changes added to each version are presented below.

1. The tags are represented by integers instead of strings.
2. All rules are instantiated in constant time. All rules to consider are generated once.
3. The rules are evaluated by counting the number of good and bad transformations. Only the best rule is applied, which removes the need for copying.

The programs were tested on a training corpus of 23 000 words extracted from 10 files of the SUC corpus. There are no unknown words, since the dictionary was generated using the whole SUC corpus. The algorithm was stopped after 100 learned rules and the result was the same for all programs.

All tests were performed on a Pentium M 1.4 GHz with 512 MB of RAM and Windows XP Professional operating system.

Program version	Running time
Original	585 min
Optimized 1	121 min
Optimized 2	14 min
Optimized 3	3 min

As seen in the table above, the final optimized version is almost 200 times faster than the original program. Since the time complexity has improved as well, this ratio increases as the training corpus grows. Figure 3 and 4 illustrate the difference in time complexity between the two programs. Running time is plotted as a function of training corpus size.

4.4 Handling of Unknown Words

The second part of the project was to develope a system for handling unknown words. In the original program all words not present in the dictionary were marked as unknown and ignored. A first approach was to found out which tag was the most common and tag every unknown word with this tag. Disgarding

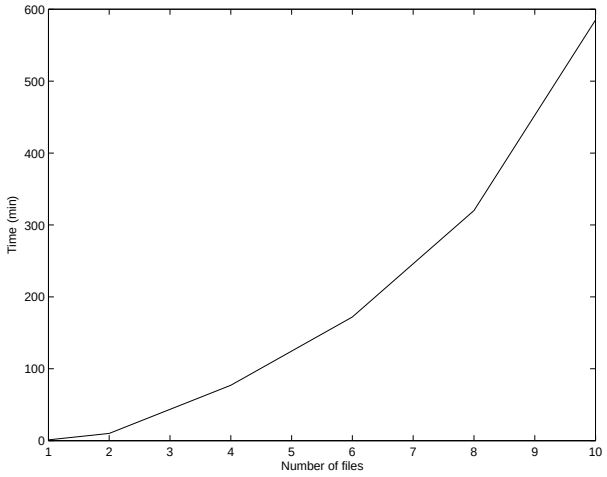


Figure 3: Running time of the original program

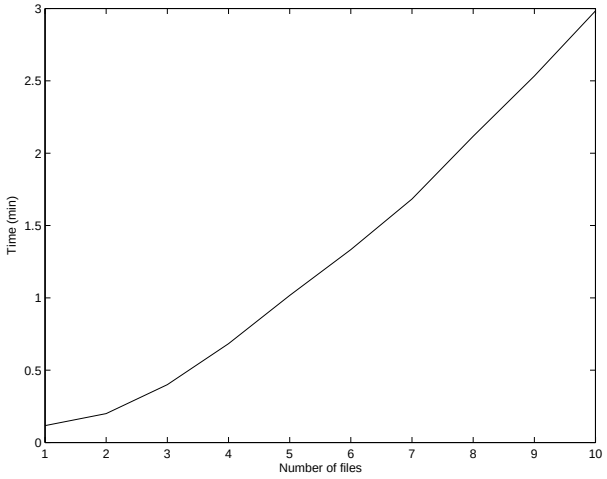


Figure 4: Running time of the optimized program

closed word classes, the most frequent tag was concluded to be "NN UTR SIN IND NOM", which represents a type of noun (see appendix for details). This simple step correctly classified about 15 % of the unknown words.

4.4.1 Initial Tagging

Although a few steps have been added to the first approach, the initial tagging is still very simple. The unknown words are divided into three groups: numbers, proper nouns and common nouns. If the word contains digits it is tagged as a number. If the it is capitalized and not in the beginning of a sentence it is tagged as a proper noun. All other words are tagged with the most common tag, i.e. common noun.

Also more sophisticated methods were tested to improve the initial tagging. For instance, we tried to conclude some features based on word endings. Although this improved accuracy, we later decided to keep the initial tagger simple. During the developement work of the learner of unknown word rules, it became clear that these types of problems could be better solved by generated rules than by our manually written rules.

However, one extra feature is part of the final implementation. It was found that many of the unknown words are compound words. As a result, we try to divide the unknown word in such a way that the last part forms a known word. If it succeeds, the unknown word is tagged with the default tag for that word. As an example, the unknown word "solstol" is tagged in the same way as the known word "stol".

4.4.2 Learning Rules for Unknown Words

The handling of unknown words was done in the way suggested by Brill. A program was written for generating rules instantiated from the rule templates described in section 3.1. The learning algorithm is very similar to the algorithm described in figure 2, with the difference that only errors concerning unknown words are considered. Another small difference is the way of counting good and bad transformations of a rule. The good and bad counts should only increase once for each unique unknown word. This prevents a rule that only corrects the tagging of one unknown word from getting a good count just because there are many occurrences of that word. Also, the training corpus must of course contain unknown words and should therefore not be part of the corpus used to create the dictionary.

The rules for unknown words process words instead of tags, and therefore the algorithm for unknown word rules is slower than the algorithm for contextual rules. In addition, tests showed that it is necessary to generate the rules in each iteration in this case.

4.4.3 Tagging

The tagging is done in three steps.

1. Initial tagging
2. Application of unknown word rules
3. Application of contextual rules

4.5 System Overview

The class diagram below presents an overview of the system.

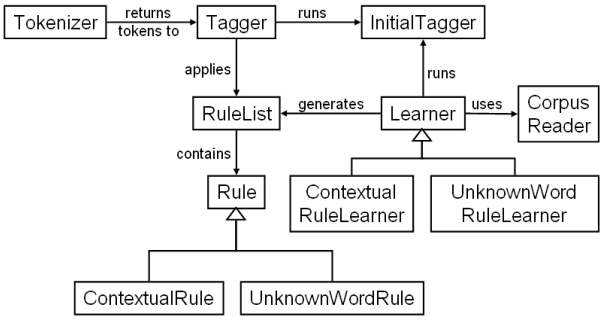


Figure 5: Class diagram

The system can be divided into two programs. First the learning algorithm must be run (one time for each type of rules). After that the rules are generated and can be used in the program performing part-of-speech tagging. **Learner** is the main class of the learning program and **Tagger** is the main class of the tagging program. As can be seen in the class diagram, many classes are used by both programs.

For clarity reasons some classes are omitted in the diagram. These include the classes for specific rule templates along with the WordDictionary and TagDictionary classes used by most other classes. The rule templates for contextual rules are represented by 13 subclasses to the class ContextualRule. In the same way the unknown word rule templates are represented by 9 subclasses to the class UnknownWordRule.

4.6 Class Descriptions

Learner is the main class of the learning program and contains the general learning algorithm. It is an abstract class that requires the subclasses to implement some parts of the algorithm.

ContextualRuleLearner is a subclass of Learner and is responsible for the contextual rule learning.

UnknownWordRuleLearner is also a subclass of Learner and is responsible for learning the unknown word rules.

Rule is the superclass of all rules. It contains the abstract methods instantiate, predicate, evaluate and apply.

ContextualRule is the superclass of all contextual rules.

UnknownWordRule is the superclass of all unknown word rules.

RuleList is a class for maintaining a list of rules.

CorpusReader is responsible for extracting words and tags from the manually annotated corpus.

InitialTagger is responsible for the initial tagging of each word with it's most common tag. Unknown words are tagged according to a few simple rules.

Tagger is the main class of the tagging program. It takes an untagged text as input and produces a tagged text as output.

Tokenizer is used by the Tagger to divide the input text into tokens.

WordDictionary contains all words of the training corpus. It is used for finding the most likely tag for a word, investigating if a word exists and searching for prefixes or suffixes of words.

TagDictionary is responsible for the translation between the string and integer representation of the part-of-speech tags.

4.7 User Instructions

Before running the learner och tagger programs, the dictionaries must be created. This is done by running the two Dictionary classes with the directory containing the training corpus passed as an argument. This creates the files word_dict.dat and tag_dict.dat which are used by the other programs. If the corpus is large, it might be necessary to increase the heap size. An example is shown below. Note that the package name

```
se/lth/cs/BrillTagger
```

has been omitted in the examples to save space.

```
java -Xmx256M WordDictionary dir
java -Xmx256M TagDictionary dir
```

Now the two learning programs can be run. The directory of the training corpus must be passed as an argument. It is important that the training corpus for UnknownWordRuleLearner has not been used when creating the dictionary. Files containing the generated rules are created.

```
java -Xmx256M ContextualRuleLearner dir
java -Xmx256M UnknownWordRuleLearner dir
```

Finally the part-of-speech tagger is ready for use. The argument is a file containing the untagged text. The output is printed to standard out.

```
java Tagger file.txt
```

In order to test the accuracy of the tagger, a small testing program has been developed. It works like the Tagger but takes a manually annotated corpus as input. That way it can compute the accuracy of the tagging. The test program is started with the following command, where the argument can be either a file or a directory.

```
java Test dir
```

5 Test Results

The implemented tagger has been evaluated by computing the accuracy of the tagging on a test corpus of 120 000 words, both with an open and closed vocabulary. The time to learn the rules have also been recorded. The tests were performed on a Pentium M 1.4 GHz with 512 MB of RAM and Windows XP Professional operating system.

5.1 Rule Learning

The contextual rules were learned in 9 hours using a training corpus of 470 000 words. The learner was stopped when 200 rules had been generated. The dictionary was built using the whole SUC corpus.

The learner of unknown word rules used a training corpus of 230 000 words and was finished after 11 hours and 30 minutes. The whole SUC corpus except the texts in the training corpus was used to generate the dictionary. The algorithm was stopped after 100 generated rules.

Figure 6 and 7 show the first ten rules learned by each learner.

5.2 Closed Vocabulary Test

A closed vocabulary means that there are no unknown words in the text to tag. Therefore the whole SUC corpus could be used to build the dictionary.

The results are presented as the percentage of correctly tagged words after initial tagging, which is called the baseline, and after applying the rules. Only the contextual rules are applicable here, since there are no unknown words. As a comparsion it can be mentioned that an accuracy of 97.0 % was reported by Brill, making the closed vocabulary assumption.

From tag	To tag	Condition
IE	SN	Tag 1, 2 or 3 after is VB PRS AKT
PN NEU SIN DEF SUB/OBJ	DT NEU SIN DEF	Tag 1, 2 or 3 after is NN NEU SIN DEF NOM
IE	SN	Tag 1, 2 or 3 after is VB PRT AKT
JJ POS UTR/NEU PLU IND/DEF NOM	JJ POS UTR/NEU SIN DEF NOM	Tag 1, 2 or 3 before is DT UTR SIN DEF
JJ POS UTR/NEU SIN DEF NOM	JJ POS UTR/NEU PLU IND/DEF NOM	Next tag is NN UTR PLU IND NOM
NN NEU PLU IND NOM	NN NEU SIN IND NOM	Tag 1 or 2 before is DT NEU SIN IND
HP - - -	KN	Tag 1 or 2 after is NN UTR SIN IND NOM
DT UTR SIN DEF	PN UTR SIN DEF SUB/OBJ	Next tag is VB PRS AKT
DT UTR/NEU PLU DEF	PN UTR/NEU PLU DEF SUB	Next tag is VB PRS AKT
PN NEU SIN DEF SUB/OBJ	DT NEU SIN DEF	Tag 1 or 2 after is NN NEU SIN IND NOM

Figure 6: The first ten contextual rules

From tag	To tag	Condition
NN UTR SIN IND NOM	NN UTR PLU DEF NOM	Suffix is “rna”
DT UTR SIN IND	NN UTR SIN DEF NOM	Suffix is “en”
PM NOM	PM GEN	Suffix is “s”
UO	NN NEU SIN DEF NOM	Suffix is “et”
NN UTR SIN IND NOM	PC PRS UTR/NEU SIN/PLU IND/DEF NOM	Suffix is “ande”
UO	NN UTR PLU DEF GEN	Suffix is “rnas”
AB	NN UTR SIN DEF GEN	Suffix is “ens”
DT UTR/NEU PLU DEF	VB PRT AKT	Suffix is “de”
JJ POS UTR SIN IND NOM	NN UTR SIN DEF NOM	Suffix is “n”
PM NOM	NN UTR PLU DEF NOM	Suffix is “rna”

Figure 7: The first ten rules for unknown words

Baseline: 91.92 %
Final accuracy: 95.18 %

5.3 Open Vocabulary Test

To test the performance of the tagger with an open vocabulary, the test corpus could not be part of the corpus used to build the dictionary. The ratio of unknown words in the test corpus were 7.44 %. The figures below show the percentage of correctly tagged known and unknown words after the three main steps of tagging.

Accuracy after initial tagging (baseline)
Known words: 90.78 %
Unknown words: 59.67 %
All words: 88.46 %

Accuracy after applying unknown word rules

Known words:	90.78 %
Unknown words:	73.37 %
All words:	89.48 %

Accuracy after applying contextual rules

Known words:	94.41 %
Unknown words:	74.70 %
All words:	92.94 %

This can be compared to the accuracy claimed by Brill. With a baseline of 92.4 %, he reported accuracies of 96.3 % for all words and 82.0 % for unknown words.

6 Conclusions

6.1 Optimization

The optimization of the original tagger was very successful, resulting in a running time almost 200 times faster when using a training corpus of 23 000 words.

6.2 Handling of Unknown Words

Unfortunately, most of the rules for unknown words turned out to give very poor results. A possible reason is that the rule templates were developed with the English language in mind. The rule template that gave by far the best results was the one that changes the tag depending on the suffix of the word. It resulted in very reasonable rules like "Change from nominative to genitive if the word ends with *s*".

6.3 The Results

The final result of our work can be summarized in the figures showing the accuracy of the tagger. The resulting accuracy was computed to 95.18 % and 92.94 %, with a closed and open vocabulary respectively. These figures are significantly lower than the ones reported by Brill. The main difference between our result and that of Brill is the baseline, which is much lower in our implementation (88 % compared to 92 %). The difference between the baseline and the resulting accuracy is about 4 percentage points for both implementations. However, it may be difficult to make comparisons between the two implementations since our is for the Swedish language and Brill's is for English.

7 Further Work

For future extenders of this work, the baseline will presumably be the focal point of attention. It may be interesting to examine why the baseline in this and Brill's implementation differs and see if improvements can be made.

A way to further increase the accuracy of the tagger, would be to introduce the lexicalized rules also suggested by Brill. It is a set of contextual rule templates that make reference to words instead of tags. However, according to Brill, these rules only improve accuracy slightly (0.2 percentage points).

Another interesting task would be to investigate in what ways the rule templates for unknown words could be adjusted to make them more suitable for the Swedish language.

8 Acknowledgements

We would like to thank Pierre Nugues for valuable help and suggestions during our work. We also wish to acknowledge the work of François Marier and Bengt Sjödin, which resulted in a clear and useful Brill tagger implementation.

References

Eva Ejerhed and Gunnel Källgren and Ola Wennstedt and Magnus Åström. 1992. *The Linguistic Annotation System of the Stockholm-Umeå Project.*

Pierre Nugues. 2004. *An Intruduction to Language Processing with Perl and Prolog.*

Eric Brill. 1995. *Transformation-Based Error-Driven Learning and Natural Language Processing: A Case Study in Part-of-Speech Tagging.*

Grace Ngai and Radu Florian. 2001. *Transformation-Based Learning in the Fast Lane.*

Johan Carlberger and Viggo Kann. 1999. *Implementing an efficient part-of-speech tagger*

François Marier and Bengt Sjödin. 2003. *A part-of-speech tagger for Swedish using the Brill transformation-based learning*

Appendix - The SUC 1.0 Corpus

Text Categories

The corpus consists of 500 text files, with approximately 2000 words each. Each file has a unique name, containing information of which category the text falls under. There are ten main text catagories and each of them has a number of subcategories. The distibution of files over the main categories are presented below.

Category	Number of files
A. Press, Reportage	44
B. Press, Editorials	17
C. Press, Reviews	27
E. Skills, trades and hobbies	58
F. Popular lore	48
G. Biographies, essays	26
H. Miscellaneous	70
J. Learned and scientific writing	83
K. Imaginative prose	127

Format

The SUC 1.0 corpus is available in two different formats called SUC1A and SUC1B. The format used in the project and described here is the SUC1A format.

The corpus is divided into text elements generally called tokens. Tokens are normally words, but also include punctuations, numbers etc. Each token is tagged with it's part-of-speech category along with a number of morphosyntactic features. The base form of the word is also part of the tag. Below is an example of a tokenized and tagged sentence, with a reference number for each token. Note that the swedish letters å, ä and ö are encoded }, { and |.

```
("<Det>" <1142>
    (PN NEU SIN DEF SUB/OBJ "det"))
("<{r>" <1143>
    (VB PRS AKT "vara"))
("<viktigt>" <1144>
    (JJ POS NEU SIN IND NOM "viktig"))
("<att>" <1145>
    (IE "att"))
("<inte>" <1146>
    (AB "inte"))
("<st|ra>" <1147>
    (VB INF AKT "st|ra"))
("<f}glarna>" <1148>
    (NN UTR PLU DEF NOM "f}gel"))
("<under>" <1149>
    (PP "under"))
("<h{ckningstiden>" <1150>
    (NN UTR SIN DEF NOM "h{ckningstid"))
("<.>" <1151>
    (DL MAD "."))
```

Part-of-Speech Categories

All tags begins with one of the two letter codes representing the part-of-speech.

Code	Swedish category	Example	English translation
AB	Adverb	inte	Adverb
DT	Determinerare	denna	Determiner
HA	Relativt adverb	när	Relative Adverb
HD	Relativ determinerare	vilken	Relative Determiner
HP	Relativt pronomen	som	Relative Pronoun
HS	Relativt possessivt pronomen	vars	Relative Possessive
IE	Infinitivmrke	att	Infinitive Marker
IN	Interjektion	ja	Interjection
JJ	Adjektiv	glad	Adjective
KN	Konjunktion	och	Conjunction
NN	Substantiv	pudding	Noun
PC	Particip	utsänd	Participle
PL	Partikel	ut	Particle
PM	Egennamn	Mats	Proper Noun
PN	Pronomen	hon	Pronoun
PP	Preposition	av	Preposition
PS	Possessivt pronomen	hennes	Possessive
RG	Grundtal	tre	Cardinal number
RO	Ordningstal	tredje	Ordinal number
SN	Subjunktion	att	Subjunction
UO	Utländskt ord	the	Foreign Word
VB	Verb	kasta	Verb

Morphosyntactic Features

Parentheses show that a feature only applies to some members of the part-of-speech or that not all the values of a feature are applicable.

Feature	Value	Legend	POS where feature applies
Gender	UTR	Uter (common)	DT, HD, HP, JJ, NN, PC, PN,
	NEU	Neuter	PS, (RG, RO)
	MAS	Masculine	
Number	SIN	Singular	DT, HD, HP, JJ, NN, PC, PN,
	PLU	Plural	PS, (RG, RO)
Definiteness	IND	Indefinite	DT, (HD, HP, HS), JJ, NN, PC,
	DEF	Definite	PN, (PS, RG, RO)
Case	NOM	Nominative	JJ, NN, PC, PM, (RG, RO)
	GEN	Genitive	
Tense	PRS	Present	VB
	PRT	Preterite	
	SUP	Supinum	
	INF	Infinite	
Voice	AKT	Active	
	SFO	S-form[1]	
Mood	KON	Subjunctive[2]	
Participle form	PRS	Present	PC
	PRF	Perfect	
Degree	POS	Positive	(AB), JJ
	KOM	Comparative	
	SUV	Superlative	
Pronoun form	SUB	Subject form	PN
	OBJ	Object form	
	SMS	Compound [3]	All parts-of-speech

Morphological parser for Latin

Alexander Malmberg
LTH
d00am@efd.lth.se

Abstract

Morphology describes how words are formed in a language, for example by adding suffixes or prefixes to existing words. In some languages, this process is very productive, and it is thus important for computational linguistics to be able to handle this. The purpose of a morphological parser is to extract information from the morphological structure of a word. In this paper, we examine this problem and briefly look at the standard two-level morphology approach of handling it. We also present a basic but working morphological parser for Latin.

1 Introduction

Morphology is the study of how words are formed. In many languages, the processes by which new words are formed are very common. For example, in English, one can form compound words, and it is common that plural forms of words are formed by adding "s" to the singular form. Other languages use other sets of prefixes and suffixes to form new words from other words, sometimes with phonological changes (such as "morphologies", where "ys" turns into "ies"). Some languages use infixes or other exotic methods for forming words.

Systems that want to process text in a language need to understand all these words. A simple and straightforward approach is to make a dictionary that lists all words. However, this is ugly from a theoretic point of view. Many of the methods that form new words are regular, and it should be possible to build a model of these methods and use it.

It is also impractical to list all words, especially in languages with rich morphological processes. For example, nearly every Latin verb has approximately 150 forms, but these can usually be formed from just three stems. Even in English, which has relatively poor morphological processes, listing all words is unlikely to work in practice. An interesting example(Sproat, 1992) involved Associated Press newswire text from a 10 month period. Even when the words from all the texts expect those of the last day of the period were collected in a dictionary, there were still many words on the final day that weren't in the dictionary. Many of these involved new forms of words that were in the dictionaries.

Thus, morphology aims at modelling how words are formed, and the job of a morphological parser is to extract information from words using this model. There are many applications of this, and different applications need different types of information. One type would be information about gender, number, tense, etc., which could be used to find the meaning of a word, or to aid part-of-speech tagging. Other applications include spell checking, or text-to-speech, where morphology can provide information about morpheme boundaries and pronunciation.

2 Two-level morphology

One standard way of writing a morphological parser is to use so called two-level morphology. This was originally done by Koskenniemi in the KIMMO system for Finnish.

The first level in two-level morphology is, roughly, a "dictionary" with idealized morphological rules. The second level is a set of phonological rules for rewriting idealized forms of words into their real forms.

The "dictionary" can be represented as a web of tries. A trie is a tree where each node has a child for each letter. This makes it possible to find the node for a word effi-

ciently: just start at the root and recursively go to the child corresponding to the next letter. Morphological rules are handled by connecting many tries; the node for a stem won't have children for all the possible endings. Instead, it will have a link to a separate trie that contains these endings. This way, only one trie is needed for each (idealized) paradigm, and it is still possible to find the node for a complete word efficiently.

The phonological rules are represented as finite state automatons that accept or reject a pair of strings. One of the strings would be a real form of a word, and the other would be an idealized form as found in the dictionary. The automaton would accept the pair if the dictionary form matches the real form.

When parsing words, these two levels run in parallel. The dictionary trie is searched recursively starting at the root. At each node, the idealized form (so far) is compared to the real form using the automaton to see if the idealized form might correspond to the real form. If it doesn't, the search need not continue below that node. (Since the phonological rules might include large changes, the system might have to search a few levels down dead-ends before the automaton can reject the pair.)

There are many practical details in implementing such a system, but this is only a brief description. A more extensive description can be found in my source for this section (Sproat, 1992).

3 Morphology in Latin

Morphology in Latin is extensive: nearly every word indicates number and gender, there are many cases, many paradigms, and many obscure forms of verbs.

However, the structure is fairly simple: words are formed by adding suffixes to a stem. There are no phonological rules (except some vowel length changes, but since I'm working with written texts, that doesn't affect my parser). Completely new words are can be formed using prefixes, but these were included in my dictionary and thus didn't cause any problems. Stems are formed in more complex ways, but again, listing all stems isn't hard (e.g. a verb may need 3–4 stems, but no more).

(It is perhaps worth noting that classic Latin is a language where it would be possible to simply build a list of all words. Being a dead language, no new texts will be written in it, so if you collected all words in all texts, you'd trivially get perfect coverage on all texts.)

During the work on my parser, I used primarily two Latin grammar references: Grammatica Nova (Larsson and Plith, 1992), and Latin Grammar (Conrad, 2004).

4 Morphological parser for Latin

I wrote a morphological for Latin. It is based on the dictionary level of the two-level morphology and doesn't include any phonological rules. The trie structure is defined in trie.h and the main source is in latin1.c.

When the program is, it reads the data files specified on the command line. Each data file defines a trie: the words contained in it, which other tries it links to, and some other interesting information (e.g. the meaning of a stem, and tense/case/etc. information for endings).

The parser function is parse(). The parse is done in three steps. First, the tries are searched for the unmodified word.

If no parses are found and the word ends in "que", the "que" is removed and the search is attempted again. "que" is a word that is sometimes attached to other words as a suffix. Since it can be attached to all kinds of words, it was convenient to special case this word here.

If no parses are found in the second search, the parser tries to parse the word as a roman numeral.

Writing and debugging the parser was fairly easy. Most of the time in the project was spent gathering and working on the data that the parser uses.

4.1 Data

When the parser is run, it is given a list of files with data that is used to build and link together the tries. There are two basic kinds of tries: stems and endings.

4.1.1 Endings

The ending tries were built by hand by me using Latin grammar resources (Larsson and

Plith, 1992) (Conrad, 2004). While it would have been possible (and straightforward) to simply make long lists of all endings from a grammar book, there are many regularities in the endings, and I tried to exploit this.

As an example, almost all verb forms use one of three sets of endings to indicate person. Thus, instead of having to list 6 endings for each combination of verb conjugation, tense, active/passive, etc., only the first part of the ending is listed along with a link to the trie with person endings corresponding to this combination. (In fact, in some cases I cheat and do this even when some forms don't follow one of the three patterns. In those cases, I also list the exceptional forms, so the parser still recognizes all valid forms; the drawback is that it will also recognize some ill-formed words.)

With some support for handling phonological rules, it would have been possible to exploit even more near-regularities. Unfortunately, the near-regular endings don't seem to follow regular phonological rules. For example, the ending for both nominative plural and genitive singular second declension nouns is "-i". For second declension nouns whose stems end in "i", such as "gladius", the "ii" in genitive singular is contracted to a single "i", "gladi", while the "ii" in nominative plural isn't, "gladii".

To handle this in a two-level morphology, it would have been necessary to introduce new "magic" letters, e.g. several variants of "i", identical except that some would combine with other "i":s and some wouldn't. Thus, you still wouldn't really be able to exploit the regularities since you'd have to explicitly list which "i" would be used in different endings. To me, this doesn't appear to be any nicer than simply listing all endings from a theoretical point of view.

4.1.2 Stems

The stems were collected from a dictionary built from the word list of another morphological parser for Latin (Whitaker, 2004). This dictionary included over 30000 entries, and while it was written in traditional dictionary form, it included enough information about the words to extract the stems and connect them to my ending tries.

The program gen_roots_dict_1 parses the dictionary and builds the data files used by my parser. The program can handle about 25000 of the entries in the dictionary. Extending the coverage is straightforward but, at this stage, time consuming since the remaining words are spread across many small paradigms.

Author	Number of words	Coverage
Caesar	51624	91%
Vergilius	63748	77%

Table 1: Parser results

5 Results

I tested my parser on "Commentariorum Libri VII de Bello Gallico" by Caesar, and the Aeneid by Vergilius (both texts from "Corpus Scriptorum Latinorum" (Camden, 2003)). The results are in table 1.

While developing the parser, I tested and analyzed the results of the parser on parts of the first chapter of the text by Caesar. These results were used to guide the development; they told me which words and paradigms that would increase coverage the most. Since the results from the Caesar text was used for this, this is likely part of the reason why the coverage is much better on the Caesar text. Another reason could be that the dictionary I extracted stems from was, according to its documentation, originally built using Caesar's texts.

On average, there were about 2.5 parses of each successfully parsed word. Many of these seem to be cases where several different genders/cases of a word have the same ending.

I have examined correctness by manually examining some randomly selected words, and by systematically testing some paradigms, and to the limits of my knowledge of Latin, all parses are valid (and usually, if a word is parsed at all, the correct parse is among the parses found).

References

David Camden. 2003. Corpus scriptorum latinorum, http://www.forumromanum.org/literature/index.htm

Eric Conrad. 2004. Latin grammar, http://www.math.ohio-state.edu/ econrad/lang/latin.html.

Lars A. Larsson and Håkan Plith. 1992. *Grammatica Nova*. Bonniers.

Richard Sproat. 1992. *Morphology and Computation*. ACL-MIT Press.

William Whitaker. 2004. Words 1.97, http://users.erols.com/whitaker/.

POS Tagger for Spanish

Carlos Miguel Gómez Gracia **Héctor Yela Reneses**

A continuación vamos a mostrar el trabajo obtenido a partir de la realización del citado proyecto en la Universidad de Lund (Suecia) la asignatura de Language Processing and Computational Linguistics (EDA 171).

1. Introducción:

En primer lugar, vamos a dar la idea general sobre la que se ha desarrollado nuestro proyecto. Para hacer eso, lo primero que debemos decir, es que un llamado POS Tagger es un programa cuya principal función es la de extraer información de un Corpus (texto de gran dimensión en el que cada palabra presenta información adicional de su estructura) para la realización de unas estadísticas que podrá utilizar en nuevas aplicaciones.

En nuestro caso, dichas estadísticas han sido usadas para encontrar las posibles etiquetas de cada palabra en nuevos textos (en nuestro caso, las etiquetas serán las distintas categorías gramaticales) y elegir la más adecuada en cada caso.

Gracias a esta herramienta, mediante un entrenamiento adecuado de nuestra base de datos, deberíamos ser capaces de etiquetar de forma adecuada cualquier frase que pretendamos evaluar. Ese entrenamiento será después clave en nuestro proyecto, porque podremos observar como difieren bastante los porcentajes de éxito si el fragmento de texto que se pretende etiquetar pertenece o no al Corpus que ha servido para entrenar.

Una vez dadas estas pinceladas que dan una idea general del propósito de nuestro proyecto, pasamos a explicar de un modo más detallado el mismo en los siguientes apartados.

2. Desarrollo del proyecto:

Nosotros hemos tratado, en primer lugar, nuestro Corpus. Éste procede del departamento de lenguaje natural de la UPC (Universidad Politécnica de Cataluña).

La estructura del mismo constaba de una palabra por línea en la que aparecían de modo sucesivo la propia palabra original del texto, la palabra raíz de la que procede la misma y un conjunto de etiquetas en el que se podían identificar algunas de las propiedades de las palabras, como son su categoría gramatical, el género al que pertenecen o su número. En el caso de que la palabra fuese una forma verbal, también podíamos identificar su propio tiempo verbal. Algunos ejemplos de esta forma original del Corpus que estamos explicando, podrían ser:

No	No	RN
Quiero	querer	VMIP1S0
Decir	decir	VMN0000

En cualquier caso, para el propósito que le hemos asignado a este proyecto, nos ha sido suficiente con utilizar solamente las primera letra de cada una de estas etiquetas originales, que es la que nos indica la categoría gramatical a la que pertenece la palabra presente en el texto. De este modo, hemos podido distinguir los diferentes tipos de palabras que, según su etiqueta, nos encontraremos a la hora de hacer el análisis. Han quedado como sigue:

A	Adjetivo
S	Preposición
N	Nombre
V	Verbo
R	Adverbio
D	Determinante
Y	Abreviatura
I	Interjección
P	Pronombre
Z	Números
C	Conjunción
F	Puntuación
X	Desconocido

W	Fechas

Una vez que fue definida con precisión la parte del Corpus sobre la que íbamos a trabajar, se procedió a recoger datos que resultaran de interés para los posteriores cálculos de probabilidades. De este modo, fueron cuatro los elementos que decidimos almacenar en nuestra base de datos:

- C(w): Número de ocurrencias de la palabra w.
- C(t): Número de ocurrencias de la etiqueta t.
- C(w, t): Número de ocurrencias de la palabra w etiquetada con t.
- C(t1, t2): Número de ocurrencias del bigrama de etiquetas (t1, t2), que consiste en la aparición de una palabra etiquetada con t1, seguida de otra etiquetada con t2.

Algunos ejemplos del cálculo del número de palabras en el Corpus son estos:

3 corral
5 corre
3 correcto
1 corrector
10 corredores
3 corren
3 correr
1 correrse
9 corresponde

También mostramos a continuación un ejemplo de la distribución de las etiquetas contabilizadas en un fragmento del Corpus, donde se puede ver como algunas de ellas aparecen muchas mas veces que otras.

A 7886
C 6660
N 22254
P 6249
R 5514
S 14216
V 13795
W 207
Z 233

Una vez que fuimos capaces de realizar correctamente la contabilización de palabras y etiquetas, procedimos a aplicar alguna de las fórmulas que nos habían sido proporcionadas durante el curso para el cálculo de diversas probabilidades. De este modo, y debido a que tras una reunión con el profesor que ejercía como tutor de nuestro proyecto decidimos usar el algoritmo de Viterbi, las probabilidades que calculamos fueron las siguientes:

- Probabilidad de transición:
  $P(ti \mid ti\text{-}1) = C(ti\text{-}1, ti) / C(ti\text{-}1)$
- Probabilidad de estado para palabras conocidas:
  $P(wi \mid ti) = C(wi, ti) / C(ti)$
- Probabilidad de estado para palabras desconocidas:
  $P(wi \mid ti) = 1$

Este algoritmo de Viterbi del que hemos hablado, determina el camino subóptimo que debe seguir para cada nodo en el autómata, descartando el resto de nodos, mientras lo atraviesa. Esta autómata es el que se construye teniendo en cuenta las posibles etiquetas que pueden presentar las palabras ambiguas del texto.

En el proyecto desarrollado, la probabilidad de transición que hemos usado se ha calculado mediante bigramas, por lo que nuestro porcentaje de acierto cuando aparecen palabras desconocidas ronda el 80%. En el caso de que hubiésemos usado trigramas para su cálculo, este resultado hubiese mejorado hasta el 90% aproximadamente.

Además de los cálculos de los que he hablado, para elevar el porcentaje de acierto al etiquetar las palabras, hemos usado también un conjunto de simples reglas que en castellano se pueden aplicar de un modo sencillo, pero que serían muy difíciles de identificar en el tipo de análisis que habíamos desarrollado. Unos de los casos más complejos de este tipo que hemos encontrado es el de la palabra “que”. Es un caso muy común su uso en cualquier texto en castellano y no resulta nada fácil distinguir con simples estadísticas cuando debemos identificarlo como un pronombre o como un determinante. Por ello, al hacer un análisis de cuáles eran las palabras en las que más errores se producían, pudimos ver que ésta era con bastante diferencia la que más problemas daba. Aún así, y a pesar de la introducción de estas reglas todavía sería necesario hacer un análisis gramatical mas profundo para su correcto etiquetado.

3. Análisis del código

Básicamente lo que hace nuestro código es analizar el corpus adquiriendo estadísticas, uno de los programas analiza todo el corpus mientras que el otro omite analizar la parte que luego usaremos como comprobante.

Una vez obtenidas estas estadísticas, que luego servirán para calcular probabilidades, utilizaremos el algoritmo de Viterbi para ir etiquetando las palabras con su correspondiente etiqueta.

Finalmente comprobamos con la parte del texto que hemos elegido, el porcentaje de acierto y, en el caso de que existan palabras desconocidas, calculamos el porcentaje para los dos tipos de palabras.

Todo este código del que estamos hablando ha sido estructurado de un modo claro y ordenado, de modo que cualquier posible ampliación que se desee realizar sobre el mismo no precise mas que un simple vistazo a los comentarios que en cada fragmento del mismo hemos situado. Este hecho también nos ha sido de gran ayuda durante la realización del proyecto, para tener claro en todo momento el lugar en el que realizábamos cada operación sobre el Corpus.

Sirva como ejemplo de la adecuada estructura del código alguna de sus partes:

```
($ini, $end) = @ARGV;
@mytags = ('A', 'R', 'D', 'N', 'V', 'I', 'Y', 'S', 'P', 'Z', 'C', 'F', 'X', 'W');
open(FILE, "corp.txt") || die "Could not open file corp.txt";
$cnt = 0;
$auxiliar = ""; #fragmento con las palabras sin etiquetar
$comprobante = ""; #fragmento para comprobar tags


#COMPROBAMOS QUE LOS ARGUMENTOS SEAN CORRECTOS
if($ini < 0 || $end >= 106124 || $ini >= $end){
	print "There is a wrong with the arguments\n";
}

else{
	#RECORREMOS EL FICHERO, OBTENEMOS EL TROZO A ANALIZAR Y
LLENAMOS LA BD
	while ($line = <FILE>){
		@linea = split(/ /, $line);

		#CAMBIAMOS LAS MAYUSCULAS POR MINUSCULAS EN LA PALABRA
		$linea[0]	=~	tr/A-ZÅÀÂÄÆÇÉÈÊËÎÏÔÖÙÛÜ/a-
zåàâäæçéèêëîïôößùûüÿ/;

		#OBTENEMOS EL TAG DE LA PALABRA
		@palabra = split(/ */, $linea[2]);

		#OBTENEMOS LA FRECUENCIA DE CADA PALABRA
		if(!exists($words{$linea[0]})){
			$words{$linea[0]} = 1;
		}
		else{
			$words{$linea[0]}++;
		}

		#OBTENEMOS LA FRECUENCIA DE CADA TAG
		if(!exists($tags{$palabra[0]})){
			$tags{$palabra[0]} = 1;
```

```
		}
		else{
			$tags{$palabra[0]}++;
		}

		#OBTENEMOS LA FRECUENCIA DE LA DUPLA PALABRA TAG
		if(!exists($wordandtag{"$linea[0] $palabra[0]"})){
			$wordandtag{"$linea[0] $palabra[0]"} = 1;
		}
		else{
			$wordandtag{"$linea[0] $palabra[0]"}++;
		}

		#OBTENEMOS LAS FRECUENCIAS DE BIGRAMAS DE TAGS
		if($cnt != 0){
			if(!exists($bigramtag{"$tagant $palabra[0]"})){
				$bigramtag{"$tagant $palabra[0]"} = 1;
			}
			else{
				$bigramtag{"$tagant $palabra[0]"}++;
			}
		}

		#OBTENEMOS LAS FRECUENCIAS DE TRIGRAMAS DE TAGS
		if($cnt > 1){
			if(!exists($trigramtag{"$tag2ant $tagant $palabra[0]"})){
				$trigramtag{"$tag2ant $tagant $palabra[0]"} = 1;
			}
			else{
				$trigramtag{"$tag2ant $tagant $palabra[0]"}++;
			}
		}

		#OBTENEMOS LAS FRECUENCIAS DE (W1,T1,T2)
		if($cnt != 0){
			if(!exists($mixgramtag{"$wordant $tagant $palabra[0]"})){
				$mixgramtag{"$wordant $tagant $palabra[0]"} = 1;
			}
			else{
				$mixgramtag{"$wordant $tagant $palabra[0]"}++;
			}
		}

		#LLENAMOS EL FRAGMENTO SI ES NECESARIO
		if($cnt >= $ini && $cnt <= $end){
			$comprobante .= "$palabra[0]\n";
			$auxiliar .= "$linea[0] ";
		}

		#GUARDAMOS LA ULTIMA PALABRA DEL FRAGMENTO PARA SABER SI
ES UN PUNTO
		if($cnt == $end){
			$ultima_palabra = $linea[0];
		}

		$tag2ant = $tagant;
		$tagant = $palabra[0];
```

```
		$wordant = $linea[0];
		$cnt = $cnt + 1;
	}


	#METEMOS EN WORDANDTAG PARA CADA PALABRA LA LISTA DE TAGS POSIBLES
	#CUIDADO LUEGO CON LAS PALABRAS SIN TAG; LAS UNKNOW!!!!!!!
	foreach $word (sort keys %words){
		$lineatags = "";
		for ($j = 0; $j <= $#mytags; $j++){
			if(exists($wordandtag{"$word $mytags[$j]"})){
				$lineatags .= "$mytags[$j] ";
			}
		}
		$tagsforword{$word} = $lineatags;
	}
```

4. Resultados

Una vez explicado como hemos calculado las diversas probabilidades para realizar un etiquetado correcto y el algoritmo usado, ya podemos explicar que han sido 2 los programas que hemos elaborado en nuestro proyecto.

Por una parte, el primero lo que hace es introducir todo el Corpus en la base de datos para, posteriormente, proceder a analizar una parte del mismo en el que no se van a encontrar palabras desconocidas. Es el fichero llamado POSinc.pl y los resultados obtenidos por él se acercan al 98% de etiquetados correctos. Para invocar a dicho programa es necesario hacerlo del siguiente modo:

perl –w POSinc.pl lineIn lineOut > exit.txt

En el otro programa llevado a cabo, el texto es entrenado por una amplia parte del Corpus para después analizar la parte restante, apareciendo de este modo palabras desconocidas. El porcentaje de éxito roza el 80% y esta en el fichero llamado POSdestex.pl. La manera de invocar este programa es:

perl –w POSdestex.pl lineIn lineOut >exit2.txt

Este último además nos devuelve el porcentaje de aciertos de las palabras conocidas y el de las desconocidas por el trainer. Para las palabras conocidas el porcentaje ronda el 88% y para las desconocidas el 25%.

5. Conclusiones y posibles mejoras

De nuestro experimento y de los resultados obtenidos podemos deducir que un POS que funciona con bigramas y sin casi tratamiento para las palabras desconocidas es una buena herramienta pero no óptima, ya que con un tratamiento sobre esas palabras desconocidas de las que solo acertamos una cuarta parte ahora y con una leve mejoría sobre las conocidas (posiblemente con trigramas, aunque se podria dar el caso que los porcentajes no mejorasen acordes con el esfuerzo y la complejidad requerida para utilizarlos) podríamos aumentar nuestra efectividad quizás hasta un 90%.

Todo y así necesitaríamos de un corpus mayor para conocer el alcance de nuestros resultados y poder trabajar en una posible mejora, habría también que comparar los resultados de nuestro proyecto empleado sobre otras lenguas, ya que quizás el castellano es una lengua más sencilla de Taggear que el inglés.

En cualquier caso, la conclusión que nosotros podemos sacar tanto de la realización del proyecto en sí, como del conjunto de la asignatura, es muy positiva, ya que ninguno de los dos habíamos tenido anteriormente experiencia alguna con el lenguaje de programación PERL y en España no se utiliza de un modo común.

De hecho, las herramientas que habíamos podido llegar a usar para tratar textos eran algunas como Lex, Yacc o Visón, y siempre en asignaturas que tenían una estrecha relación con el mundo de los compiladores, nunca con el fin que le hemos aplicado en este proyecto.

6. Agradecimientos

Por último, nos gustaría agradecer a CLiC Centre de Llenguatge i Computació (UB) el hecho de habernos permitido la utilización de su extraordinario Corpus de aprendizaje en español, ya que sin él, no habría habido proyecto, ni resultados ni nada.

Por supuesto, también debemos acordarnos y dar las gracias tanto a la persona que se ha encargado de tutorarnos este proyecto, Richard Johansson, como al profesor encargado de las clases de teoría, Pierre Nugues. Nos han ayudado a sacar adelante esta tarea, especialmente en la parte que nos resultaba más nueva para nuestros conocimientos, todo lo relacionado con el algoritmo de Viterbi. También agradecer a aquella gente de la Universidad de Lund que permite tantas facilidades a los alumnos para trabajar cómodamente, tanto a nivel de medios como de horarios.

Ya para acabar, sólo queda presentarnos de un modo mas formal. Este proyecto ha sido realizado por:

- Carlos Miguel Gómez Gracia, alumno del Centro Politécnico Superior (Zaragoza, España).
- Héctor Yela Reneses, alumno de la Universidad Politécnica de Cataluña (Barcelona, España).

Part-of-Speech Tagger for Swedish

Simon STÅHL
Computer Science, Lund University
sys03sis@ludat.lth.se

Abstract

A Part-of-Speech (POS) tagger is a tool that automatically resolves the ambiguities that would occur if a text was tagged with the help of a dictionary. Automatic tagging of texts is used in many applications (grammar checkers, etc.), and quite high accuracy can be achieved.

This document describes a stochastic POS tagger that uses a unigram version of the Viterbi algorithm. The overall idea behind the stochastic POS tagger and the Viterbi algorithm is also described.

The unigram tagger is evaluated using a small corpus of just below 100 000 words, and the results indicate that a larger corpus would have yielded greatly improved accuracy.

1 Introduction

There are many applications that needs a text to be tagged with Parts-of-Speech (POS), the lexical categories of words and symbols in a text, and there are several ways to do this tagging. The most primitive approach is to determine the POS of a word by looking it up in a dictionary. This unfortunately leaves us with a lot of ambiguities, since many words have more than one possible POS.

Early POS taggers resolved these ambiguities by using hand coded rules, but writing these rules is both time demanding and complex. Newer rule based taggers derives rules automatically from a hand annotated corpus.

Other POS taggers uses stochastic models. The ambiguities is resolved using statistics, derived from a hand annotated corpus. The probabilities of the possible tag sequences of a given word sequence is calculated, and the one with the highest probability is chosen. This is approximated using N-grams (bigrams and trigrams mainly), since statistics of long sequences is impossible to obtain.

The Viterbi algorithm can be used to optimize the probability calculation of the tag sequences by discarding sub-sequences that can not be part of the tag sequence with the highest probability. This makes the stochastic POS tagger less time and memory consuming than if it would have to calculate all possible paths.

This document describes a stochastic POS tagger, using the Viterbi algorithm. The current implementation uses only unigrams, which makes the result noticeably less correct than a bigram or trigram implementation.

Chapter two describes the POS tagger with its statistics and probabilities. In the third chapter the Viterbi algorithm is described, and in chapter four the results are discussed.

2 The POS tagger

2.1 Statistics

All statistics are derived automatically from a hand annotated corpus. This training of the POS tagger only needs to be done once, since the statistics is saved to file to be used when tagging. The statistics derived is:

C_n – The number word tokens.
$C(w,t)$ – Occurences of word *w* tagged with *t*.
$C(t)$ – Occurences of the tag *t*.
$C(t_1,t_2)$ – Occurences of the tag bigram t_1,t_2.
$C(t_1,t_2,t_3)$ – Occurences of the tag trigram t_1,t_2,t_3.

Since the current implementation only uses unigrams, no bigram or trigram statistics is saved, as would be the case in a more advanced implementation.

2.2 Probabilities

The probability of a tag sequence *T* for a given word sequence is:

$$P(T)P(W|T)$$

That is, the probability of the tag sequence in it self multiplied with the probability of the word sequence knowing the tag sequence. The tag sequence with the highest probability is chosen to tag the word sequence with.

The probabilities $P(T)$ and $P(W|T)$ are approximated using N-grams, most often trigrams, backing off to bigrams (and unigrams) in the case

of missing data. The trigram approximations of the probabilities are:

$$P(T) \approx P(t_1)P(t_2|t_1)\prod_{i=3}^{n} P(t_i|t_{i-2}, t_{i-1})$$

$$P(W|T) \approx \prod_{i=1}^{n} P(w_i|t_i)$$

These probabilities are estimated with the statistics from the hand annotated corpus:

$$P(t_i) = \frac{(C(t_i))}{C_n}$$

$$P(t_i|t_{i-1}) = \frac{(C(t_{i-1}, t_i))}{(C(t_{i-1}))}$$

$$P(t_i|t_{i-2}, t_{i-1}) = \frac{(C(t_{i-2}, t_{i-1}, t_i))}{(C(t_{i-2}, t_{i-1}))}$$

$$P(w_i|t_i) = \frac{(C(w_i, t_i))}{(C(t_i))}$$

Since the current implementation only uses unigrams, the probability *P(T)* is approximated further, on the expense of less correct tagging. The unigram approximation of *P(T)* is:

$$P(T) \approx \prod_{i=1}^{n} P(t_i)$$

The resulting unigram approximation of a tag sequence is:

$$P(T)P(W|T) \approx \prod_{i=1}^{n} P(t_i)P(w_i|t_i)$$

The unigram approximation only selects the most common tag of a word, and does not take any previous tags into consideration when selecting tags.

3 The Viterbi algorithm

The Viterbi algorithm is a dynamic programming algorithm that optimizes the tagging of a sequence, making the tagging much more efficient in both time and memory consumption.

In a naïve implementation we would calculate the probability of every possible path through the sequence of possible word-tag pairs, and then select the one with the highest probability. Since the number of possible paths through a sequence with a lot of ambiguities can be quite large, this will consume a lot more memory and time than neccesary.

Since the path with highest probability will be a path that only includes optimal subpaths, there is no need to keep subpaths that are not optimal. Thus the Viterbi algorithm only keeps the optimal subpath of each node at each position in the sequence, discarding the others.

4 Results

The tagger was tested with a corpus of just below 100 000 Swedish words. The original idea was to test it with the full corpus (with approximately 1.2 million words), but this unfortunately proved to be too time demanding, both when training the tagger and when the statistics was loaded before tagging.

A test set of just below 25 000 words was tagged by the tagger, that had been trained on the small (100 000 words) corpus, and the results were compared with a copy of the test set that was hand annotated. The tagger was able to tag 74.6% of the words correctly, which is way below what could be expected. As a comparison, Sjöbergh (2003) report an 87.3% accuracy when using an unigram tagger trained on a corpus of 1.1 million words.

The reason for this difference in accuracy is probably mainly because of sparse data, many words in the test set are not found in the small corpus, and this is not handled by the tagger. 71.6% of the erroneous taggings were, as a matter of fact, tagged with the tag UKN, which means that the tagger was not able to find any possible tags for that word. This indicates that a larger corpus would have given a result closer to that of Sjöbergh (2003).

5 Conclusion

The implemented POS tagger only uses unigram probabilities, which means that it never takes any sequence of tags into consideration, only selects the most common tag of a word. It does not try to tag unknown words in any way, it just tags them as UKN, unknown, and it was trained on a small corpus of just below 100 000 Swedish words. But with the limitations of both the tagger and the small corpus in mind, it gave relatively good results, 74.6% correct tagged words.

The large percentage, 71.6%, of the erroneous taggings, that were tagged with the tag for unseen word, indicates that a larger corpus would give a higher percentage of correctly tagged words. A faster version of the tagger would have been able to be trained on the larger corpus, but the current version would have taken days to process it.

Extending the tagger to use bigrams and trigrams would also improve the correctness, and would be the next natural step. This would take sequences of tags into consideration, which is

important in natural languages.

The handling of unseen words is non-existent in the current implementation, and even a naïve algorithm to handle this would improve the results.

References

Pierre Nugues. 2004. *An Introduction to Language Processing with Perl and Prolog.* Lund University, Lund, Sweden.

Johan Carlberger and Viggo Kann. 1999. *Implementing an efficient part-of-speech tagger*. Royal Institute of Technology, Stockholm, Sweden.

Jonas Sjöbergh. 2003. *Combining POS-taggers for improved accuracy on Swedish text.* KTH Nada, Stockholm, Sweden.

University of Leeds, Viterbi algorithm: http://www.comp.leeds.ac.uk/roger/HiddenMarkovModels/html_dev/viterbi_algorithm/s1_pg1.html

Wikipedia, Viterbi algorithm: http://en.wikipedia.org/wiki/Viterbi_algorithm

Luz Abril Torres Méndez. 2000. *Viterbi Algorithm in Text Recognition.* McGill University, Montreal, Quebec, Canada: http://www.cim.mcgill.ca/~latorres/Viterbi/va_main.html

Collocations Computed from the Web

Tomasz Adam Maksymilian WYSOCKI
Engineering Physics, Lund Institute of Technology
Ehrensvärdsg. 22, 212 13 Malmö, Sweden
pogodno@gmx.de

Abstract

This paper describes a prototype system implemented for verifying the correctness of all verb-preposition-collocations found in a given text. The verification is done using statistics from the world's largest corpus - the Internet. The tool used for obtaining these statistics is the Google Web APIs service[1]. The probability of correctness is computed according to the concepts of proportional score, t-score and mutual information.

1 Introduction

1.1 Preface

Almost each and every one of us has tried to learn a foreign language at some point in our lifes, being more or less successful in doing so. The obstacles that one has to conquer in the process of learning a natural language are more or less complex, depending on ones mother tongue, learning ability, other languages of which one already has command etc. What should a person do when exposed to a new, unfamiliar expression? Trying to express something in German, linguistic rules valid for English could always be applied. A word-for-word translation could also be tried. The attained result might be acceptable, but in most cases it would not. Regarding fixed expressions, or collocations, one can almost be sure of the non-feasibility of a literal translation. Collocations have often a very long history and can be very specific for the given language. They are linguistic entities that one has to learn by heart, or continue to be ignorant. In this paper I concentrate on the collocations of a verb and a preposition, e.g. *"depend on"*. The following sections describe the prototype of an application helping the user in checking if such a collocation was written correctly or not. The lexicon used for verification is the Internet and the tool employed is Googles Web APIs service. As measures for the probability of correctness, the concepts of proportional score, t-score and mutual information are applied.

In the following, whenever the word *collocation* is stated, it will have the meaning of *verb-preposition-collocation* only.

1.2 Background

In order to verify the correctness of a collocation, one could ask a native speaker of that language, but after a while he would get tired of being asked these things over and over again. Instead, one might try consulting a dictionary, or checking the frequency of the collocation in a large text corpus. This last alternative is what the application implemented in this project makes use of. The corpus used for verification is the largest cohesive and searchable existing for the time being - the Internet. Using any search engine on the Web, one can look up the collocation in doubt, and check the amount of hits generated. Then the process could be repeated, substituting another preposition for the original one. Applying the concept of proportional score, the conclusion would then be that the proper preposition to be used in that collocation is the one generating most hits. In the next section this very method, together with two others, will be described.

Google, known for its popular search engine, provides a service for software developers, allowing the applications of each of them to state 1000 automated queries per day. This service offers many of the options found in the original Web-based search engine of Google's. In this project, the Google

[1] Google Web APIs service.

Web APIs service is used constructing a java-application for checking the collocations in a text.

2 Approach

2.1 System

The sentences one would like to verify should be saved in a text file and tagged using a part-of-speech tagger (e.g. the MXPOS Tagger[2], as done in this project). The input fed to the java-program implemented in this project is a tagged text file. The application recognizes as verbs all the words labeled with either of the tags _VBZ, _VBN, _VBG, _VBP and as prepositions all the words labeled with the tag _IN (these are the tags used by the MXPOST). The probability for each collocation found is then calculated using one of the three scoring methods described below and the figures obtained using Google's Web APIs service.

The result is then sent to the standard output. The printout presents the probability for the collocation with the original preposition. Additionally, if there is a preposition giving higher probability, it is suggested as the more proper one. Otherwise, the second best preposition is presented along with the original one.

2.2 Obtaining Probabilities

When calculating the probability of a collocation (e.g. *"depends on"*) being correct, the following figures have to be known:

c(v), the amount of hits for the verb *"v"* (*"depends"*);

c(p_i), the amount of hits for the preposition *"p_i"* (*"on"*);

c(v, p_i), the amount of hits for the expression *"v p_i"* (*"depends on"*).

The collocation (consisting of the verb v and the preposition p_i) considered to be the most correct one is the one having the largest probability π_i, which is defined as

$$\pi_i = \frac{score(v,p_i)}{\sum_j score(v,p_j)}$$

where

$$\sum_i \pi_i = 1$$

The list of prepositions considered by the application is

$$\begin{aligned} L \quad = \quad & \{as, at, by, for, from, in, of \\ & off, on, than, to, upon, with\} \end{aligned}$$

Figures for each of those prepositions are obtained just once, in one of the initial steps of the algorithm.

The number N, representing the total number of words in English texts on the Internet is roughly estimated to equal around 11 billion. This figure is obtained by checking the frequency of such words as "*in*", "*on*" and "*of*" in a large, fixed-size text corpus, then checking the amount of those words on the Internet, multiplying the both figures for each word, and taking the average among all of them.

2.3 Scoring Methods

The application uses one of the three most common scoring methods for calculating the probabilities.

2.3.1 Proportional Score

The proportional score is the same as the amount of hits obtained for the collocation, that

$$score_p(v, p_i) = c(v, p_i)$$

2.3.2 T-Score

The t-score is defined by

$$score_t(v, p_i) = \frac{c(v,p_i) - \frac{1}{N}c(v)c(p_i)}{\sqrt{c(v,p_i)}}$$

The t-score shows in what extent the association between two words v and p is non-random. In case of a high t-score, the result can be assumed to be quite confident.

[2]Maximum Entropy Part-Of-Speech Tagger.

2.3.3 Mutual Information

The mutual information is defined by

$$score_t(v, p_i) = \log_2 N \frac{c(v,p_i)}{c(v)c(p_i)}$$

The mutual information puts the probability of observing the collocation *"v p"* in comparison with the probabilities of observing v and p independently. This implies that if the collocation occurs often compared with the occurrence of the words v and p, it should be considered as probable.

3 Examples

The following sentences are examples of those used for verification of the system.

1. *The weather depends on the climate.*
2. *The rate depends of the initial values.*
3. *The health of children depends at least partially on their access to health services.*
4. *Lato was substituted for Maradona.*
5. *The luxurios champagne was substituted with less expensive, but even more sofisticated bavarian wheat beer.*

4 Results

The final results obtained for the sentences from the previous section were the same for each method, although the probabilities differed widely.

1. *The weather depends on the climate.*

Method	c	t	m
Probability (%)	77	48	27

Second best: *depends upon*

Method	c	t	m
Probability (%)	16	22	23

2. *The rate depends of the initial values.*

Method	c	t	m
Probability (%)	1	4	8

Suggestion: *depends on*

Method	c	t	m
Probability (%)	76	49	27

3. *The health of children depends at least partially on their access to health services.*

Method	c	t	m
Probability (%)	0	1	2

Suggestion: *depends on*

Method	c	t	m
Probability (%)	76	48	27

4. *Lato was substituted for Maradona.*

Method	c	t	m
Probability (%)	50	29	18

Second best: *substituted by*

Method	c	t	m
Probability (%)	18	18	15

5. *The luxurios champagne was substituted with less expensive, but even more sofisticated bavarian wheat beer.*

Method	c	t	m
Probability (%)	12	14	14

Suggestion: *substituted by*

Method	c	t	m
Probability (%)	49	29	18

The most time-consuming part of the program is the communication with Google. If this procedure could be made faster, it would actually be feasible to include this feature into some word-processing software in order to make it easier to process a larger mass of text.

5 Discussion

As seen in the previous section, all the three methods delivered exactly the same results. The single issue that differs between them

is the confidence in the result being unquestionably correct.

Using the proportional score, the result space contains in each case a single significant peak, making it easy to distinguish the correct (according to this method) preposition.

The mutual information delivers several peaks having similar values, the highest of them still being the correct solution.

The level of confidence of the t-score method lies somewhere in-between the two others.

The third sentence is actually erroneous, although the system finds it to be correct. This is due to the fact that the sentence consists of a principal clause (*The health of children depends on their access to health services.*) and a subordinate clause (*at least partially*) and the commas are left out in the source file. Even if the commas would be there, Google does not make any difference between "*depends at*" and "*depends, at*", thus the results would be identical.

It is free for anyone to construct a webpage of his own, in the language and with the contents of his choice. Due to this nature, the Internet contains a lot of noise. This contributes to the fact that much of the virtual substance, the collocations in general, contains grammatical errors.

The errors observed in the contents of a webpage are correlated with the native language of its author. It his highly probable, that the members of a language group make the same mistakes, applying for instance the concept of literal translation of expressions and collocations. If the language group is large, the texts produced by its members could cause significant noise, as it is correlated. The largest noise peaks will probably origin from such large languages as Spanish, French and Chinese, whereas the noise produced by members of minor language groups would not be significant, although noticeable. Since the mass of English text constructed by its native speakers is much larger than the single native groups', this noise should not reach the amplitude of the real, correct signal.

The situation would look somewhat different if our language of interest would be other than English. A small language is always strongly exposed to such noise, because one false entry makes a large contribution to the total corpus in this tongue.

There are always errors made by the natives as well, but this should not be correlated enough to give peak noise, it would rather be an amplification of the basic noise level. If an error convicted by the native speakers would become significant, it should rather be considered as synonym with the original one rather than incorrect.

The system is partly optimized in order to delimit the processing time and the amount of requests to Google. The check of the amount of hits for the single prepositions is done only once, at the startup of the system. Still, one run is performed for the verb and the collocation for each sentence, even if they have been checked before. The system could be modified to remember previous searches in order to save time and enquiry-credits. Such a procedure would though increase both the memory space needed and the internal running time of the system. However, the large limitation of the costly server connection-time would cause the net save in running time to be positive.

The system is easily extended to cover other scoring methods. Constructing a subclass to the already written one and adding the desired methods is all that is needed.

Since Google's search service is not limited to searches in the English language only, the system could easily be modified in order to handle almost any other tongue. The issue of tagging the text remains, though part-of-speech taggers for several different languages are available on the Internet itself. The tags indicating the verbs and the prepositions would have to be the same as used by the MXPOS tagger, otherwise the method extracting the collocations would have to be overwritten.

6 Conclusions

As we could see, the system arrives at the correct conclusions for each of the examples used for verification. Although, the example space is small and extended testing is required in order to obtain the confidence-statistics for the system.

It is also necessary to develop a model, which would combine the results yielded by all the methods. For example, in case of the three methods having the same outcome (as in the examples presented in section 5), the delivered answer could be considered to be certain.

The essential conclusion is that, using the Internet as language source and Google's APIs service as searching tool, it is possible to construct an effective linguistic assistant, not only for English, but also for almost any language. Such an application must also not be limited to handle mere verb-preposition-collocations, but also other expressions, or even spell checking. To make such a system practical, one would have to incorporate the system into word-processing software. This would make it much easier and faster to access. To increase the speed even further, the server connection time would have to be delimited in order to yield a feasible running time. This could be solved by keeping a local record storing previous searches. It would also be necessary to higher the limit of searches allowed, as a standard-size text would require many more than 1000 queries.

7 References

Google Web APIs service, 2004, http://www.google.com/apis/

Maximum Entropy Part-Of-Speech Tagger (MXPOST), 2004, http://www.cis.upenn.edu/~adwait/statnlp.html

NuguesPierre, Language Processing and Computational Linguistics, Lecture Notes, LTH, 2004

LUNDS UNIVERSITET

Institutionen för Datavetenskap

http://www.cs.lth.se

Produktion: Jonas Wisbrant • 2005