Collocation statistics to check verb-preposition usage in Swedish

Brice Jaglin
Lund University / Sweden
ex05bj8@student.lth.se

Felix Nairz
Lund University / Sweden
nairz@sbox.tugraz.at

Abstract

Non-native speaker have always difficulties to know which prepositions to use with common verbs. This can be predicted by considering verb-preposition frequencies in a corpus. Our goal was to develop a small Java application that tell the user which pairs are frequent and which pairs are not, to signal where there may be a mistake. Our software uses an external tagger to identify relevant pairs of words, and rate them according to likelihood algorithms relying on statistics from the biggest corpus on earth, the Web.

1 Introduction

Verb-Preposition pairs like ”going to”, ”meeting at” are one of the hardest things to learn in a new language. They normally do not follow any rules and even if there exists some rules, there are many exceptions. When we both started to learn Swedish we realized those difficulties again. It was therefore a pleasant opportunity to develop a Verb-Preposition checker for the Swedish language during our ”Language Processing and Computational Linguistics” course.

The aim of our project was to help beginners in Swedish to check their texts in order to correct Verb-Preposition pairs. But the approach is not limited to the Swedish language and can therefore be extended to other languages with very little effort. During the implementation of a prototype our focus was on a good design of the software with easy changeability of all components.

The result of our programming is a fully working prototype with a graphical user interface. The user can type some Swedish text and query the web to evaluate the likelihood of a Verb-Preposition pair. Depending on the computed likelihood are the pairs displayed in different sizes to distinguish between common and uncommon pairs. Some more advanced options are also available but will be described later.

In the following section we will give you a brief overview of our system-design followed by a section describing the external tools we used for our project. After this we will get more into detail in the implementation section followed by a description of the algorithms used to evaluate the likelihood. At this point the reader has enough knowledge to comprehend the discussions section where we talk about various problems we faced and how our prototype could be further improved. Finally we close with our conclusions.

2 System Overview

The main steps to get from a raw text to a highlighted text that takes probability into account are as follows:

- Identify verb-preposition pairs in the text using a tagger
- Get statistics for verb, preposition and bigram[1]
- Compute the likelihood of those pairs
- Normalize the likelihood in order to display them graphically

Having those steps in mind makes it easy to understand the system design. We start with a overview

[1]A bigram is a pair of two words. It is a special case of N-grams, with N=2. In the report, it is a synonym of verb-preposition pair, but can be extended to any pair of words.

of the project in this section and get into more detail about the external tools and the implementation in section 3 and 4. The whole system can roughly be separated into four parts: our project, the external tools, the external database and the internal database. This separation can better be seen in figure 1.

Starting from the graphical user interface that you can see in figure 2, the user types the text to check into a text box and presses the check-button. This leads to the first step in the processing chain described before - the tagging. The tagging is done using an external tagger, that will be described in more detail in the next section. For now is it enough to know that the tagger returns the a text where each word is tagged. It is our task to create a list of bigrams out of this text that will be further processed. For each bigram in the list is it our goal to compute a likelihood. This is done by querying the web using the Google-API. The API returns among other things the number of Google-hits for the search term. The final likelihood for the bigram is then computed using different algorithms on the number of hits for the verb, preposition and the bigram. In the last step this likelihood is shown graphically in the user interface so that the user can distinguish between frequent and non-frequent bigrams. For performance reasons a cache is implemented that stores the number of hits. Whenever new words are in the user-text the size of the cache increases making the application faster the more often it is used.

3 External tools

In this section we will give a introduction to the external tools we used.

3.1 Granska

Granska[2] is a part-of-speech tagger developed at KTH, in the department of Numerical Analysis and Computer Science (NADA). We used the old version of Granska, written in C++[3]. It takes a text file as an input and returns on the standard output the list of words associated tags, what makes it easy to interface it with any programming language, such as Java in our example. It is fast and efficient (97% of correct tags).

Granska uses a probabilistic model, combined to an external POS lexicon generated thanks to the Stockholm Umea Corpus[4].

3.2 Google SOAP-API

Google SOAP-API[5] is an API developed by Google that allows search-queries from inside a program without having to start a browser. You get the results for your query as you get them when using a browser. We do not make use of any other feature than the number of hits for each query. It is further important to let the API only list results in the Swedish language, otherwise the results will be wrong.

To be able to use the service, one has to get a free license-key[6]. Each key is limited to 1000 queries a day, but we could have asked for several keys to increase the number of allowed queries up to several thousands. This remains a static limitation though, whose probability to occur has been decreased by implementing a cache (see section 4.3).

We have experienced strange problems when querying the API. It is sometimes necessary to loop and send again the query until we finally get an answer. This issue does not slow down the whole process really much, since the number of tries has never been greater than 4.

4 Implementation

In this section we describe the processing chain from a raw text to version with highlighted Verb-Preposition pairs in a bit more detail than in section. A class diagram describing only the objects directly related to the language aspect of our software is presented in figure 3.

4.1 User Interface

The user-interface already presented in figure 2 was developed using the Java Swing Gui. We did not put extensive value on the user interface. It just fulfills its basic requirements.

The progress-bar is not really accurate because the time required for processing each bigram is considered as equal, whereas it is not: depending on whether the bigram statistics have been cached, time for processing one bigram can vary from several milliseconds to seconds.

[2] `http://skrutten.nada.kth.se/`

[3] Note that the code compiles well with g++3.4, but does not seem to compile with g++4.x.

[4] `www.ling.su.se/staff/sofia/suc/suc.html`

[5] `http://code.google.com/apis/soapsearch/`

[6] Since December 2006, it is no longer possible to get new license keys

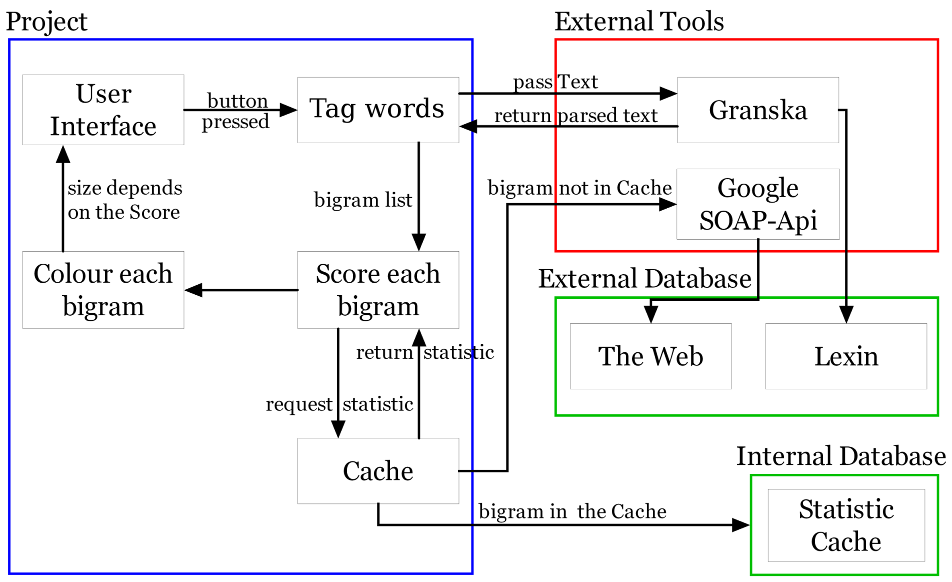
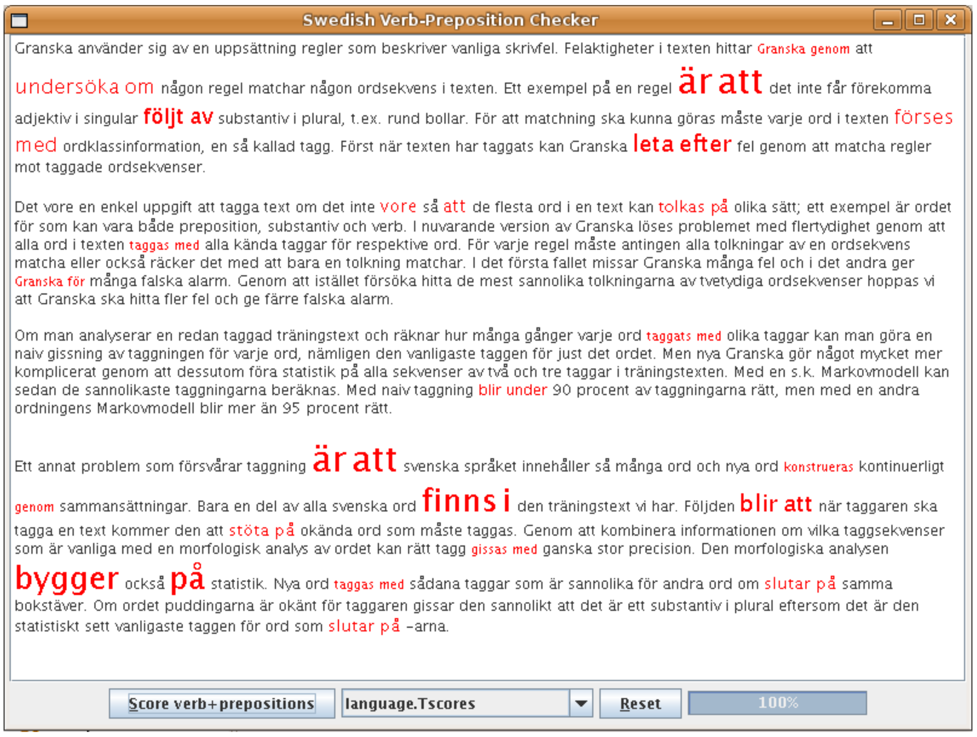


Figure 1: System design. The whole system can be separated into: our project, external tools, external database and internal database.



Granska använder sig av en uppsättning regler som beskriver vanliga skrivfel. Felaktigheter i texten hittar Granska genom att undersöka om någon regel matchar någon ordsekvens i texten. Ett exempel på en regel är att det inte får förekomma adjektiv i singular följt av substantiv i plural, t.ex. rund bollar. För att matchning ska kunna göras måste varje ord i texten förses med ordklassinformation, en så kallad tagg. Först när texten har taggats kan Granska leta efter fel genom att matcha regler mot taggade ordsekvenser.

Det vore en enkel uppgift att tagga text om det inte vore så att de flesta ord i en text kan tolkas på olika sätt; ett exempel är ordet för som kan vara både preposition, substantiv och verb. I nuvarande version av Granska löses problemet med flertydighet genom att alla ord i texten taggas med alla kända taggar för respektive ord. För varje regel måste antingen alla tolkningar av en ordsekvens matcha eller också räcker det med att bara en tolkning matchar. I det första fallet missar Granska många fel och i det andra ger Granska för många falska alarm. Genom att istället försöka hitta de mest sannolika tolkningarna av tvetydiga ordsekvenser hoppas vi att Granska ska hitta fler fel och ge färre falska alarm.

Om man analyserar en redan taggad träningstext och räknar hur många gånger varje ord taggats med olika taggar kan man göra en naiv gissning av taggningen för varje ord, nämligen den vanligaste taggen för just det ordet. Men nya Granska gör något mycket mer komplicerat genom att dessutom föra statistik på alla sekvenser av två och tre taggar i träningstexten. Med en s.k. Markovmodell kan sedan de sannolikaste taggningarna beräknas. Med naiv taggning blir under 90 procent av taggningarna rätt, men med en andra ordningens Markovmodell blir mer än 95 procent rätt.

Ett annat problem som försvårar taggning är att svenska språket innehåller så många ord och nya ord konstrueras kontinuerligt genom sammansättningar. Bara en del av alla svenska ord finns i den träningstext vi har. Följden blir att när taggaren ska tagga en text kommer den att stöta på okända ord som måste taggas. Genom att kombinera informationen om vilka taggsekvenser som är vanliga med en morfologisk analys av ordet kan rätt tagg gissas med ganska stor precision. Den morfologiska analysen bygger också på statistik. Nya ord taggas med sådana taggar som är sannolika för andra ord om slutar på samma bokstäver. Om ordet puddingarna är okänt för taggaren gissar den sannolikt att det är ett substantiv i plural eftersom det är den statistiskt sett vanligaste taggen för ord som slutar på -arna.

Figure 2: User Interface. Prepositions are highlighted in red. A large font size means that the bigram is quite common according to the chosen algorithm.

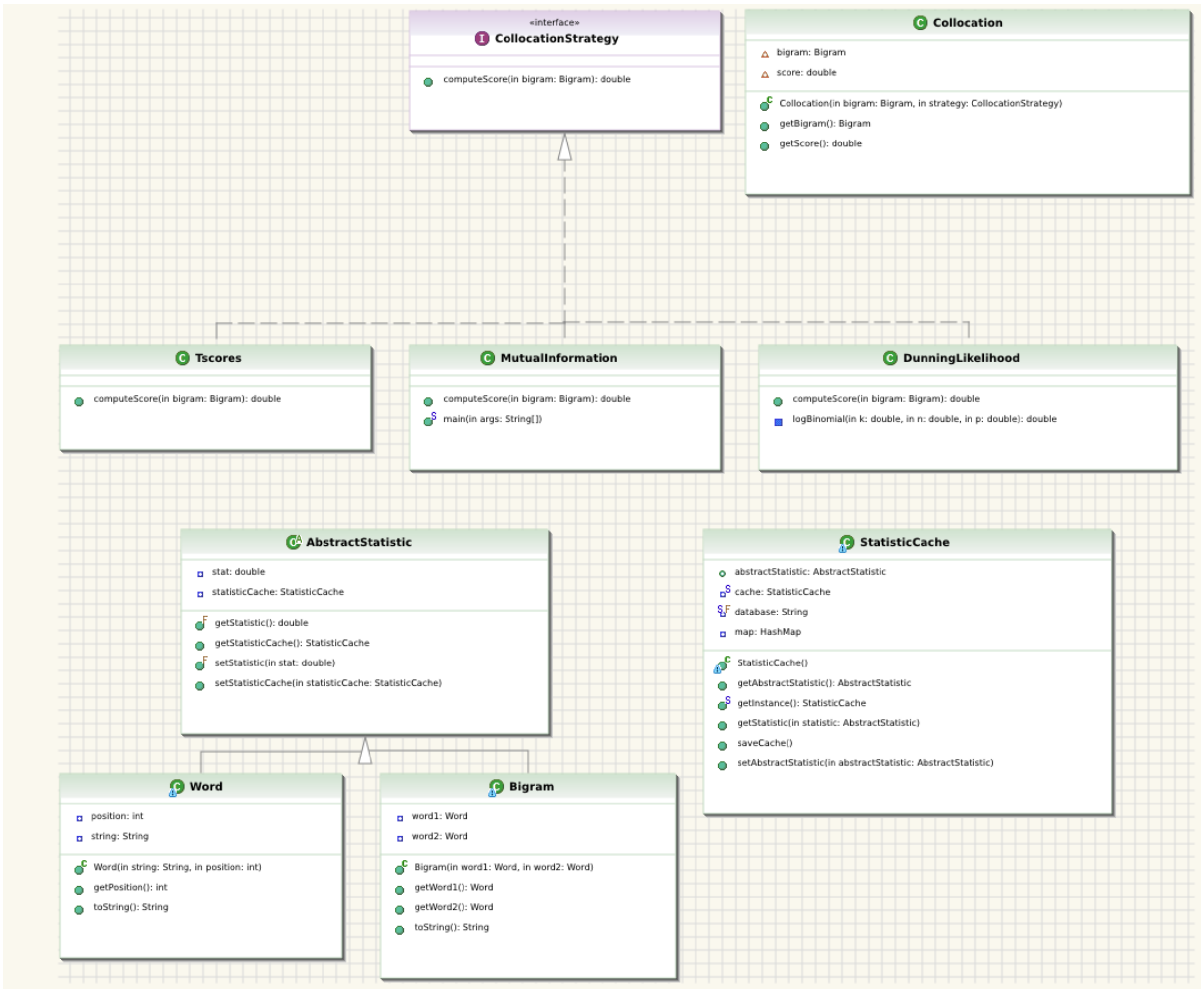

«interface»
CollocationStrategy
computeScore(in bigram: Bigram): double
Collocation
bigram: Bigram
score: double
Collocation(in bigram: Bigram, in strategy: CollocationStrategy)
getBigram(): Bigram
getScore(): double
Tscores
computeScore(in bigram: Bigram): double
MutualInformation
computeScore(in bigram: Bigram): double
main(in args: String[])
DunningLikelihood
computeScore(in bigram: Bigram): double
logBinomial(in k: double, in n: double, in p: double): double
AbstractStatistic
stat: double
statisticCache: StatisticCache
getStatistic(): double
getStatisticCache(): StatisticCache
setStatistic(in stat: double)
setStatisticCache(in statisticCache: StatisticCache)
StatisticCache
abstractStatistic: AbstractStatistic
cache: StatisticCache
database: String
map: HashMap
StatisticCache()
getAbstractStatistic(): AbstractStatistic
getInstance(): StatisticCache
getStatistic(in statistic: AbstractStatistic)
saveCache()
setAbstractStatistic(in abstractStatistic: AbstractStatistic)
Word
position: int
string: String
Word(in string: String, in position: int)
getPosition(): int
toString(): String
Bigram
word1: Word
word2: Word
Bigram(in word1: Word, in word2: Word)
getWord1(): Word
getWord2(): Word
toString(): String

Figure 3: Class diagram for language-related objects.

4.2 Tagging

Tagging each word is the first task to do for our software when processing a text. From a list of pairs word/POS, our software identifies the pairs verb/preposition using the following rules[7]:

- A word identified as a *verb*,
- One or several words tagged as *adverbs*,
- A word identified as a *preposition*, a *particle*, a *subjunction*, or a *infinitive mark*.

The adverbs are not saved, but must be present in the pattern to identify for example negation forms of prepositional verbs (tycker *inte* om).

4.3 The cache

Even for smaller texts the number of queries to the Google-API can be very high. For every verb-preposition pair in the text we have to query Google three times (once for the verb, the preposition and the bigram). Since one query took approximately three seconds it was therefore very important to implement a cache. The cache stores the word and the number of hits returned by Google API. The more often you use the application and the more different Verb-Preposition pairs are included in the text, the better the cache performance.

As you can see in figure 3, the `StatisticCache` consists in a hashmap of `AbstractStatistic` objects. Verbs, preposition and bigram statistics are thus cached indifferently. As Java comes with already implemented functions to save/restore an object on/from disk, our cache is saved from one start to another.

4.4 Algorithms

To compute the likelihood of verb-preposition pairs, three algorithms were implemented, from the most simple to the most advanced: *Mutual Information*, *T-scores*, and *Log likelihood ratio*, as shown in figure 3. We won't go into detail about the formulas of those algorithms in this paper and therefore refer to (Nugues, 2006) for more information.

The problem is that all these algorithms compute the probability based on *occurrences* statistics, while we only get *hits* statistics using a search engine such as Google. However, according to (Lapata and keller,), it is reasonable to approximate occurrences with hits. An underlying issue was to determinate the size of the corpus, i.e. in our Web case the total number of pages indexed. Extrapolating results returned when searching common words such as *the*, several studies have shown that this number was around $8.1 * 10^9$. This may be not really accurate and up-to-date, but it is sufficient for our prototype where we only need an order of magnitude.

It is difficult to estimate which algorithm is the best, but it seems that the more complex the algorithm is, the better it is. Independently of the log effect, the *Log likelihood ratio* maps its values on the whole interval, and is more robust to large statistic variation within the same bigram.

4.5 Display of Results

The score returned by the algorithms is just a absolute number. To be able to display the likelihood graphically it is necessary to transform those absolute score into a relative one. In our first version of the project we have chosen to take the smallest absolute number as the relative minimum and the biggest absolute number as the relative maximum and linearly interpolate all values between. In the next project iteration we added the option to interpolate not only linearly but also logarithmic which lead to better results in some cases.

The final likelihood is displayed through variations of the font-size. The smallest relative number is set as the minimum font-size we defined and the biggest relative number is set as the maximum font-size. All values in between are interpolated either linearly or logarithmic and then rounded to an even number.

5 Discussions

The Google-API is a nice and powerful tool but has some limitations as well. Every user has to register for this service through the Google webside to get a unique key. This key has to be added to every search query, so that only registered users are allowed to use this service. But the probably biggest limitation is the maximum of 1000 queries within 24 hours per key. There is no way to increase this number, not even if you pay for it. Another drawback is that the reaction time seems to be throttled down and you sometimes get no answer. But since we do not know another compara-

[7]Part-of speech-codes can be found in the SUC manual

ble service we decided to still use it. In addition the whole software is designed flexible enough and with clear interfaces, so that every single part can be replaced by another one very easily.

In our basic version of the cache we did not care about entries getting out of date. It could for example happen that the number of Google-hits for a word has changed since it is placed in the cache. For such cases a ”refreshment” after a certain amount of time would be good.

The normalization before displaying the results works only well for larger texts. If you have only one Verb-Preposition pair then it is the minimum and the maximum at the same time. One possibility would be to safe the likelihood for all bigrams and compare the current bigram likelihood to the ”global” likelihood and chose the font-size corresponding to this likelihood.

It could be possible to improve accuracy of likelihood values by querying not only the given form of the verb, but also other tenses. For example, if a verb is used in perfect in the checked text, statistics about present and past tenses should also be collected. This would however result in slowing down the process because of the increased number of queries.

6 Conclusion

We have shown with this project that is was possible to use statistics generated from a search engine in order to highlight possible mistakes in verb-preposition usage. As any frequency-based heuristic analysis, this method suffers from false positives and false negatives issues: rare verb-preposition pairs do exist, and the most common pairs are not always the correct ones. The prototype is however a good start for a beginner willing to get rid of the most frequent mistakes.

Java classes were designed in a flexible way, so it should be easy to use a different parser or a different corpus by adding some classes implementing existing interfaces. A possible extension to our prototype would be a module of suggestion. A preposition in a pair with a low likelihood value may thus be replaced by another one, more common.

References

Pierre Nugues. *An Introduction to Language Processing with Perl and Prolog*. 2006. Springer.

Mirella Lapata and Frank Keller. *Web-based Models for Natural Language Processing*.