



UNIVERSITY OF OXFORD

COMPUTING LABORATORY

Interpretation of Noun Compounds

Dissertation for the degree MSc in Computer Science

Author: Lilit Darbinyan

Supervised by: Professor Stephen Pulman

August, 2010

Table of contents

List of tables	- 5 -
List of figures	- 5 -
1. Introduction	- 6 -
2. Background and the problem in context	- 7 -
2.1. Motivation	- 7 -
2.2. Related research.....	- 8 -
2.2.1 Proposed strategies	- 8 -
2.2.2. Rationale for utilizing verb-based paraphrasing	- 9 -
2.3. Problem	- 10 -
2.3.1. Formulation of the problem	- 10 -
3. Method	- 12 -
3.1 Description of the method	- 12 -
3.2 Verb phrase extraction	- 12 -
3.2.1 The British National Corpus.....	- 12 -
3.2.2 Main assumption	- 13 -
3.2.3 Extracting even more verb phrases: WordNet hypernyms	- 14 -
3.3 Validation	- 15 -
3.3.1 Web based validation	- 15 -
3.3.2 Mutual information based validation.....	- 16 -
3.3.3 N-gram probability based validation	- 16 -
3.3.4 Using a bit of everything.....	- 16 -
3.4. Evaluation.....	- 17 -
3.5. Review of the chapter	- 17 -

4. Verb phrase extraction	- 18 -
4.1 Searching in the corpus.....	- 18 -
4.1.1 BNC structure	- 18 -
4.1.2 Grammatical relations	- 19 -
4.1.3 Implementing the main assumption	- 19 -
4.1.4 Why BNC?	- 21 -
4.2 WordNet hypernym integration	- 22 -
4.2.1 WordNet senses.....	- 23 -
4.2.2 Identifying the relevant sense	- 23 -
4.2.3 Method of unrelated hypernym elimination.....	- 23 -
4.2.4 Discussion	- 24 -
4.3 Implementation details.....	- 24 -
4.3.1 General structure.....	- 25 -
4.3.2 The execution package	- 25 -
4.3.3 The structures package	- 26 -
4.3.4 The utils package.....	- 26 -
4.4 Review of the chapter	- 27 -
5. Validation	- 28 -
5.1 Web based validation.....	- 28 -
5.1.1 Yahoo API.....	- 28 -
5.1.2 Query construction	- 29 -
5.1.3 Implementation details	- 31 -
5.1.4 Review of web based validation	- 31 -
5.2 Mutual information based validation	- 31 -

5.2.1 Definition of mutual information	- 31 -
5.2.2 Adjusting to our needs.....	- 32 -
5.2.3 Implementation details	- 32 -
5.2.4 Review of mutual information based validation.....	- 33 -
5.3 N-gram probability based validation.....	- 33 -
5.3.1 N-gram models	- 33 -
5.3.2 Ranking	- 34 -
5.3.3 Implementation details	- 34 -
5.3.4 Comparison of N-gram and mutual information based validations.....	- 35 -
5.4 Sequences of validation methods	- 36 -
6. Evaluation and results	- 37 -
6.1 Data	- 37 -
6.2 Human judge feedback.....	- 37 -
6.2.1 Scoring mechanism.....	- 37 -
6.2.2 Agreement level.....	- 38 -
6.3 Metrics and results.....	- 40 -
6.3.1 Defining metrics	- 40 -
6.3.2 Results.....	- 41 -
7. Conclusions	- 42 -
References	- 44 -
Appendix A - Output on the test set.....	- 46 -
Appendix B - Source code	- 48 -

List of tables

2.1. Suitable verb phrases for the compound <code>sea breeze</code> and the corresponding paraphrases.....	-11-
3.1. Queries and ranks corresponding to each verb phrase extracted for the compound <code>apple cake</code>	-15-
4.1. Descriptions of grammatical relation tags.....	-19-
5.1. Noun compound query examples	-29-
5.2. Queries for <code>immigrant minority</code> and <code>consist of</code> and their web counts.....	-30-
6.1. Example of the format presented to human judges for evaluation	-38-
6.2. Scores used for evaluation by human judges	-38-
6.3. Alpha measure macro output.....	-39-
6.4. Percentage based metrics	-40-
6.5. Results obtained using different metrics and different target scores.....	-41-

List of figures

4.1. Package view of the system.....	-25-
--------------------------------------	------

1. Introduction

Noun compounds are sequences of two or more nouns acting as one noun in a sentence. Peach tree, office employee and lunch break are all examples of noun compounds. Compounding is an effective tool for describing new concepts in a concise manner and this might explain why noun compounds are so frequent in English language.

The frequency of noun compounds draws special attention to them. Any system that deals with some kind of semantic analysis of written text needs to be able to semantically interpret the noun compounds as well. To interpret a noun compound would mean to reveal the hidden relationship that underlies between the nouns of the compound. Office employee is an employee that works in the office, whereas office computer is a computer that belongs to the office.

While humans don't see any difficulty in uncovering this hidden relation, the task is much harder for a computer program. In order to complete the interpretation task, it has to develop knowledge about the nouns and the relations that can hold between them.

This thesis focuses on the problem of noun compound interpretation. Its aim is to propose a method and, based on that, implement a system which interprets noun compounds using verbs and prepositions. The proposed method consists of two phases. In the first phase possible verb phrases are extracted from a large text corpus. The role of the second phase is to rank the verb phrases produced in the first phase, and eliminate the ones that have low rankings. Therefore, given a noun compound as an input, the system outputs a ranked list of up to three verb phrases that paraphrase the compound. For example, construction materials was paraphrased by the system in the following ways: materials used in construction, materials used for construction and materials needed for construction.

The thesis consists of seven chapters. Chapter 2 gives background information, presents the previous work in this direction and provides rationale for this research. It then formulates the problem to be tackled and the motivation behind it. Chapter 3 briefly describes the proposed method. The next two chapters provide detail on the two phases of noun compound interpretation: Chapter 4 explains the verb phrase extraction phase and Chapter 5 focuses on the validation phase. Evaluation of the system and the final results are presented in Chapter 6. Chapter 7 concludes the thesis and proposes future directions.

2. Background and the problem in context

This chapter aims to demonstrate the importance of the problem of noun compound interpretation, the complexity of the problem, the approaches used to tackle it, and the advantages and drawbacks of each approach. After setting up the background, the specific problem to be tackled in this thesis is defined and explained.

2.1. Motivation

Importance of noun compounds in the language

Nakov (2007) defines compounding as a mechanism of producing a new word by putting two or more existing ones together. Noun compounds, as defined by Downing (1977), are sequences of two or more nouns acting as one noun, e.g. `peach tree` or `world war`; I stick to this definition throughout the thesis. It is worth taking a moment and realizing how often English language makes use of noun compounds. According to Baldwin & Tanaka (2004), 2.6% of words in the British National Corpus are parts of some noun compound. This is not a surprise though; in English language compounding is known to be used as a highly effective tool to describe new concepts in a concise manner.

Importance of interpreting noun compounds for semantic analysis of text

In any noun compound, there is a hidden relation that holds between the nouns, and in fact, the compound is created by removing this relation. For instance, instead of saying “a `professor that teaches linguistics`”, we remove the relation and use the compound “a `linguistics professor`” instead. It is quite easy for humans to uncover this relation when they need to understand the meaning of the compound. This is because humans have a great knowledge about the world and the types of relations that can hold between the given nouns. However, if this task is to be performed by a computer, then it has to be able to closely represent human knowledge, and only then, use this knowledge to complete the interpretation task.

Because noun compounds are so frequent in written text, systems that deal with semantic analysis of text cannot ignore those. And because the meaning of the compound cannot be directly obtained from the nouns, the system should have some kind of way of interpreting it. This clarifies the need and significance of methods that are able to disambiguate and explain the semantics of a compound.

2.2. Related research

This section presents the data from the literature on approaches of noun compound interpretation suggested by different researchers and analyzes the advantages, drawbacks and potential for practical application of each approach.

2.2.1 Proposed strategies

Since noun compound interpretation is such an important aspect in text processing, it has gained growing interest among many researchers. Different approaches have been suggested to address this problem. Butnariu & Veale (2008) group those approaches into two broad strategies: top-down and bottom-up.

In the top-down strategy, the problem of noun compound interpretation can be viewed as a classification problem. The main assumption here is that there are only a finite number of ways in which any two nouns can be connected. The goal of compound interpretation turns into that of classifying a given compound into the correct class. The majority of researchers following this strategy use abstract relations as classes, and try to assign any given compound to one of the abstract relations. Girju et al. (2005) suggest 21 classes of abstract relations, including POSSESSION, IS-A, INSTRUMENT, SOURCE, ACCOMPANIMENT, etc. Nastase & Szpakowicz (2003) propose five high-level classes of relations (CAUSALITY, PARTICIPANT, QUALITY, SPATIAL and TEMPORALITY), each of which includes more concrete relations, resulting in thirty fine grained relations. In this classification, for example, `flu virus` falls under CAUSALITY and `home town` is under SPATIAL.

Another approach for classifying noun compounds is utilized by Lauer (1995) who makes use of eight prepositions (`about`, `at`, `for`, `from`, `in`, `of`, `on` and `with`) instead of using abstract relations. According to this classification, `adventure story` belongs to the class `about`, while `baby chair` belongs to the class `for`. Although Lauer uses parts of language instead of abstract relations, nevertheless, due to the limited number of proposed classes, his approach still falls within the top-down strategy.

The second broad strategy to interpret noun compounds is the bottom-up strategy in which noun compounds are being interpreted through paraphrasing those using suitable words. According to views of a number of recent authors such as Nakov & Hearst (2006) and Butnariu & Veale (2008), the parts of the language most suitable for paraphrasing noun compounds are verbs and prepositions, because they are able to clearly represent the relationship that holds between the nouns in a compound. A number of methods were suggested by researchers following the bottom-up strategy and utilizing verbs for paraphrasing noun compounds. Nakov (2007) constructs wildcard queries from the noun compound and obtains the missing verb from the web-search results. The model suggested

by Butnariu & Veale (2008) first learns the relational possibilities of each noun from large text corpora, and then combines those to get the most likely paraphrase.

2.2.2. Rationale for utilizing verb-based paraphrasing

This subsection discusses the advantages and drawbacks of top-down and bottom-up strategies as well as the suitability of verbs for paraphrasing noun compounds. The purpose of the discussion is to justify the usage of bottom-up strategy and verb-based paraphrasing in this thesis project.

As it was shown in the previous sub-section, the top-down strategy assumes the existence of a finite set of relations between nouns within compounds and classifies the compounds into finite number of classes based on those relations. As a result, this strategy has a major drawback. Downing (1977) shows that the types of relations that can occur between nouns are infinite; therefore, when such a diversity is reduced to a finite set of relations, loss of detail is unavoidable. The number of verbs in English language, in contrast, is infinite (Downing, 1977). Furthermore, verbs are able to represent fine details of relations that hold between nouns. Butnariu et al. (2009) demonstrate the advantages of the bottom-up strategy using the following example: noun compounds `sleeping pill` and `headache pill` are both classified under the abstract class `FOR` in Levi's classification (Levi, 1978), who follows the top-down strategy; however, they have very different meanings. Verbs, in contrast, can easily capture this difference: `aid` perfectly paraphrases `sleeping pill`, whereas `prevent` paraphrases `headache pill`.

As a result of disadvantages of the top-down strategy, its usefulness is doubtful. In fact, in the available literature there aren't many data concerning possible applications of this strategy. Paraphrasing using verbs and prepositions, on the other hand, has many practical implications in areas of natural language processing. Applications such as information retrieval, information extraction, machine translation and question answering all would benefit from the ability to paraphrase noun compounds. For example, if an information retrieval system is given a query that contains a noun compound and if it can paraphrase the noun compound using a verb, then it can increase its recall by searching not only for the nouns, but also for the paraphrase including the verb.

Thus, the existing literature shows that the bottom-up strategy utilizing verbs and prepositions to paraphrase and interpret noun compounds has many advantages and applications. This strategy is relatively new; there is still a lot to be investigated in this direction.

2.3. Problem

In the previous sections it was shown why the problem of semantic interpretation of noun compounds is important. Further, based on the existing literature two main strategies utilized to solve the problem were presented. A number of methods suggested by researchers following each strategy, as well as the advantages and drawbacks of each strategy were described.

This section aims to define the specific problem to be tackled, the approach to be utilized to solve the problem, and the outcomes to be reached within the scope of this thesis project.

2.3.1. Formulation of the problem

In the previous chapters we have defined a noun compound to be a sequence of two or more nouns. This thesis is dedicated to the interpretation of noun compounds comprised of exactly two nouns; it is outside of the scope of this thesis to look at more complex compounds.

Further, based on the discussion presented in subsection 2.2.2, in this thesis the bottom-up strategy is utilized as the one having major advantages and many practical implications in areas of natural language processing; as for which parts of the language to be used for paraphrasing noun compounds, the verbs (followed by one or more prepositions if needed¹) are chosen as the most suitable for that purpose. The problem of interpretation in this thesis, therefore, narrows down to a task of finding suitable verb phrases that are able to paraphrase the compound.

The next step was to choose a format in which the noun compounds were going to be paraphrased. The decision on the format to be utilized was based on the findings of Gagné & Shoben (1997), demonstrating that relational possibilities in noun compounds are more frequently suggested by the modifier noun (i.e., the first noun). Based on that, the task of this thesis project was further narrowed down to searching only for such verb phrases that can paraphrase the noun compound in the following format: “noun2 verb_phrase noun1²”. Table 2.1 demonstrates how the compound *sea breeze* can be paraphrased in this format. The first column contains three verb phrases that can be used to create a paraphrase; the second column contains the actual paraphrases that are made up using the verb phrases.

¹ In the rest of the thesis, for conciseness purposes, we will use the term “verb phrase” when talking about “a verb followed by one or more prepositions, if needed”.

² Versus also searching for the format: “noun1 verb_phrase noun2”.

Verb phrase	Paraphrase
blow from	breeze that blows from the sea
blow off	breeze that blows off the sea
come from	breeze that comes from the sea

Table 2.1: Suitable verb phrases for the compound sea breeze and the corresponding paraphrases

As the example shows, it is very likely for a noun compound to have more than one suitable verb phrase. Therefore, in my work I do not limit the number of verb phrases to be extracted, but rather produce a ranked list of verb phrases, and provide the top three as a final output.

To sum up, the problem to be tackled in this thesis is as follows: given a noun compound comprised of two nouns as an input, to provide a list of up to three verb phrases that can be used to paraphrase the compound.

3. Method

In the previous chapter it was shown that in this thesis noun compound interpretation comes in the form of paraphrasing it using suitable verb phrases. This chapter briefly explains how exactly this goal is achieved.

3.1 Description of the method

In order to obtain suitable verb phrases for a noun compound, I am using unsupervised learning. While in supervised learning there is a teacher who has the correct outputs for a given training set, in unsupervised learning examples of correct outputs are not required. Extrapolating this to our case, there is no training set of noun compounds for which the correct paraphrases are known and from which the rules could be learned.

The full cycle of paraphrase generation can be viewed as a two-step process:

1. Verb phrase extraction
2. Validation

The verb phrases are obtained from the British National Corpus, which is a large text corpus comprised of English written text. The concrete method of choosing suitable verb phrases from the corpus is covered in Section 3.2.

After the verb phrases are obtained, different types of validation techniques are used with the purpose to rank the verb phrases by confidence level and to eliminate those that do not make sense. The details are provided in Section 3.3. The final output of the system is a set of (maximum) three top ranked verb phrases, which paraphrase the compound.

To evaluate how good the system is, it is executed on a fresh set of noun compounds. The output is provided to native English speakers who are asked to score the paraphrases. Different measures are then applied to the scored data to measure the quality of the system.

The following sections aim to give a general idea of how each phase of paraphrase generation works.

3.2 Verb phrase extraction

3.2.1 The British National Corpus

Search for suitable verb phrases is performed on the British National Corpus. The following quotation, which explains what the corpus contains, is extracted from the British National Corpus official website:

“The British National Corpus (BNC) is a 100 million word collection of samples of written and spoken language from a wide range of sources, designed to represent a wide cross-section of current British English, both spoken and written.”

For the purposes of this thesis, we are only interested in the written part of it, which comprises 90% of the full corpus. In its original version, the written text is tagged with part of speech (POS) tags generated by the CLAWS tagger. However, as it will be shown later, the grammatical relations between the words in a sentence are also needed. For these purposes, the whole written part of the corpus has been parsed with the C&C parser. Throughout the rest of the thesis, the term BNC will be used to refer to the grammatically parsed version of the corpus. The original version will no more be referred to.

3.2.2 Main assumption

The suggested method builds on a central assumption that defines which verb phrases might be suitable to paraphrase the noun compound. The assumption is the following:

“Given the noun compound $(n_1 \ n_2)$, if we can find an occurrence of n_1 with a verb phrase v_p such that it is the verb’s *object*, and an occurrence of n_2 with the same verb phrase v_p such that it is the verb’s *subject*, then, v_p *might* be suitable for paraphrasing the compound in the format: ‘ n_2 that v_p n_1 ’.”

Let’s look at an example to understand the common sense under the assumption. Say the compound `apple cake` is given. Somewhere in the corpus the system comes across the following two sentences:

This **cake** contains a lot of sugar.

The boxes in the shop contain oranges and **apples**.

Because in the first sentence n_2 (`cake`) appears with a verb phrase `contain` as its subject, and in the second sentence n_1 (`apple`) appears with the same verb phrase as its object, then the system assumes that the verb phrase `contain` *might* be able to paraphrase the compound. Indeed, in this case it is true that an `apple cake` is a cake that contains apples.

Note that the system doesn’t assume that if this condition holds, then the verb phrase is a definite paraphrase for the compound. Indeed, with frequent verbs such as `be`, `have` or `do`, for example, it is highly likely that the condition of the assumption will hold for most of the compounds. It is easy to imagine that at least one occurrence of each of the following types of sentences can be found:

The **cake** is ...

... are apples.

It doesn't, however, follow, that an apple cake is a cake that is an apple. Later in Section 3.3, when describing the validation phase, it will be shown how the system eliminates the non-sensible verb phrases.

3.2.3 Extracting even more verb phrases: WordNet hypernyms

For some noun compounds applying just the above mentioned condition produces very good results already: many sensible verb phrases are in the output list of this phase. However, for many compounds, the output list of this phase is very short; it doesn't contain many or even any sensible verb phrases. There are different reasons for that.

One important reason is that for compounds such as olive oil, where the two nouns are highly correlated, it is very rare to see the word olive occurring in a sentence by itself. More often it appears as a part of the compound olive oil, in which case a verb phrase for olive cannot be obtained.

Another reason can be that some words are very rare in general. Let's consider the compound jute product. The word jute has only 80 occurrences in the BNC, therefore the chances of it appearing with a suitable verb phrase (which will also appear with the word product) are very low.

Therefore, in order to obtain more verb phrases, I am using the same central assumption for the *hypernyms* of both nouns. A hypernym of a word is its more generalized concept. For example, the hypernym of olive is fruit, and the hypernym of jute is fiber. Chapter 4 will detail on the extraction of hypernyms; for the moment it's enough to know that for each noun its two most frequent hypernyms are used.

The main assumption slightly changes to accommodate the search for hypernyms:

“Given the noun compound $(n_1 \ n_2)$, if we can find an occurrence of n_1 **or any of its top two hypernyms** with a verb phrase v_p such that it is the verb's *object*, and an occurrence of n_2 **or any of its top two hypernyms** with the same verb phrase v_p such that it is the verb's *subject*, then, v_p *might* be suitable for paraphrasing the compound in the format: ' n_2 that v_p n_1 '”.

After this phase, the system is left with a list of verb phrases which have evidence of usage with the two nouns, or their hypernyms. This list is passed to the next important phase of paraphrase generation, which, as mentioned above, is validation.

3.3 Validation

The purpose of validation is to produce a ranked list of verb phrases and to eliminate those that are less likely to paraphrase the compound. There are different types of validation methods that have been explored in this thesis. These are explained below.

3.3.1 Web based validation

The idea underlying web based validation is that if a paraphrase is a good one, then it is highly likely to appear on the web. Moreover, the more it appears on the web, the better paraphrase it is. The paraphrases that have absolutely no appearance in the web are eliminated from the list of possible paraphrases. The number of documents on the web that contain the paraphrase is used as a rank for the verb phrase.

Let's look at a small artificial example. Table 3.1 shows a list of verb phrases that have been inputted to the validation phase for the noun compound `apple cake`. The second column represents the query that has been entered into Yahoo! search engine. A more detailed explanation of how exactly the queries are formed given the verb phrase and the compound, will be given in Chapter 5. The third column provides the web-based rank of the verb phrase.

Verb phrase	Query	Rank
<code>serve</code>	<code>cake serves apples</code>	0
<code>be made with</code>	<code>cake made with apples</code>	289
<code>be like</code>	<code>cake is like an apple</code>	2
<code>be made of</code>	<code>cake made of apples</code>	212
<code>be filled with</code>	<code>cake filled with apples</code>	281
<code>be without</code>	<code>cake is without apples</code>	0
<code>drop</code>	<code>cake drops apples</code>	0

Table 3.1. Queries and ranks corresponding to each verb phrase extracted for the compound `apple cake`

After web based validation, the system will know that the most probable verb phrases for `apple cake` are `be made with`, `be filled with` and `be made of`.

Web based validation is very useful from the view of first instance filtering. It is capable of filtering out most of the non-sensible verb phrases.

Web validation on its own is not perfect though. One drawback is that the search engine doesn't regard punctuation as separate tokens; therefore often the retrieved documents don't contain the exact phrase that has been searched, but include punctuation as well.

Why is this bad? Let's say we have the compound `love song`, and one of the extracted verbs for it is `say`. In this case, the web ranking for the paraphrase "`song says love`" is 2,270. But as we go through the documents retrieved, we see that they contain lines like

```
Their song says "love who you love" without any hidden agenda ...
```

```
The funny thing is my song says "Love don't matter anymore" ...
```

This, however, is not what we want because these search results cannot be taken as evidence of the fact that `song says love` is a valid sentence that appears on the web. What we want are documents that contain the exact phrase. The bad news is - this is not possible to implement with nowadays web search engines.

What we can do, though, is filter out the verb phrases that generally have frequent occurrence with any noun. This is what the next validation method is capable of doing.

3.3.2 Mutual information based validation

Mutual information is a measure between two words that describes how correlated the words are. The term was introduced by Fano (1961). It compares the number of times the words appear together with the number of times they appear independently. The higher the measure, the more correlated the words are.

The detailed explanation of how exactly the mutual information is calculated for the purposes of this thesis will follow in Chapter 5. At this point it is important to know that some sort of mutual information is calculated between the first noun and the verb phrase, and the second noun and the verb phrase. The idea is to check whether the verb phrase is specific to the two nouns, or it is just a high frequency verb phrase that appears with most of the nouns in the corpus.

Mutual information based validation eliminates the verb phrases with high frequency if they do not seem to be extremely correlated to the nouns in the noun compound.

3.3.3 N-gram probability based validation

Another way of ranking the paraphrase according to its validity is to calculate the probability of the paraphrase appearing in a corpus. Paraphrases with high probability get high rankings. The probability calculation is based on the bigram model which will be covered in detail in Chapter 5.

3.3.4 Using a bit of everything

As it was shown in the previous sections, web validation is good for initial filtering, the mutual information based measure helps to eliminate the high frequency verbs that are not

well correlated with the nouns, and the N-gram probability measure provides an alternative approach to measure the validity of the paraphrase based on its estimated probability. In this thesis, I am experimenting with all types of validation techniques. Tests have shown that the following sequence of validation techniques produces the best results:

- Web-based validation → Mutual information based validation → Final web-based re-ranking

After the first two phases the system has done an initial filtering and has eliminated the high frequency verb phrases. But it needs some final re-ranking because mutual information has spoiled the order that represented the characteristics of the whole paraphrase. Therefore, another web-validation can be performed, which basically reorders the list of verb phrases based on their web rankings.

3.4. Evaluation

In order to evaluate the quality of the system, 50 random noun compounds have been extracted from the SemEval2010 Task 9 test set. The system has produced results on the test set. After that, human subjects have been recruited and asked to provide scores for each paraphrase. Different measures have been applied to the scored data to allow performing fine analysis. Questions such as “What percentage of noun compounds has a 1st ranked paraphrase with a score less than 1.5 if averaged across the scores of all human judges” have been answered after the analysis.

3.5. Review of the chapter

In this chapter the method that has been used to solve the noun compound interpretation problem has been described. In this view, noun compound paraphrasing is a two stage process; in the first stage verb phrases that are likely to form a paraphrase for the compound are extracted from the BNC. The second stage receives the list of likely verb phrases and uses a combination of validation techniques to eliminate the non-sensible paraphrases and rank the sensible paraphrases based on their likelihood of being a good paraphrase.

A brief description of the evaluation that has been carried out in order to identify the quality of the system was also provided.

The following chapters will go into the implementation details of each of the paraphrasing stages discussed in this chapter.

4. Verb phrase extraction

4.1 Searching in the corpus

As already mentioned in Chapter 3, a large text corpus is needed for obtaining the verb phrases for a given noun compound. The British National Corpus is used as such. The following sections describe the structure of the BNC, the grammatical relations that we will be searching for in the corpus, and as a conclusion, a brief discussion of why I have chosen the BNC and not another text corpus.

4.1.1 BNC structure

For the purposes of this thesis we are interested in the grammatically parsed version of the BNC. Below is an extract from the BNC showing the structure of the corpus.

```
(det play_1 A_0)
(ncmod _ read_6 aloud_7)
(aux read_6 be_5)
(xmod _ demands_3 to_4)
(dobj demands_3 -it_8)
(ncmod _ demands_3 really_2)
(xmod _ play_1 demands_3)
(ncsubj demands_3 play_1 _)
(det sound_11 the_10)
(ncmod _ voice_15 human_14)
(det voice_15 the_13)
(dobj of_12 voice_15)
(ncmod _ sound_11 of_12)
(dobj needs_9 sound_11)
(dobj bring_17 it_18)
(ncmod _ bring_17 alive_19)
(xmod _ needs_9 to_16)
(ncsubj needs_9 play_1 _)
<c> A|a|DT|I-NP|O|NP[nb]/N play|play|NN|I-NP|O|N really|really|RB|I-
ADVP|O|(S\NP)/(S\NP) demands|demand|VBZ|I-VP|O|(S[dc1]\NP)/NP
to|to|TO|I-VP|O|(S[to]\NP)/(S[b]\NP) be|be|VB|I-
VP|O|(S[b]\NP)/(S[pss]\NP) read|read|VBN|I-VP|O|S[pss]\NP
aloud|aloud|RB|I-ADJP|O|(S\NP)\(S\NP) -it|-it|JJ|I-ADJP|O|N
needs|need|VBZ|I-VP|O|(S[dc1]\NP)/NP the|the|DT|I-NP|O|NP[nb]/N
sound|sound|NN|I-NP|O|N of|of|IN|I-PP|O|(NP\NP)/NP the|the|DT|I-
NP|O|NP[nb]/N human|human|JJ|I-NP|O|N/N voice|voice|NN|I-NP|O|N
to|to|TO|I-VP|O|(S[to]\NP)/(S[b]\NP) bring|bring|VB|I-
VP|O|(S[b]\NP)/NP it|it|PRP|I-NP|O|NP alive|alive|JJ|I-
ADJP|O|(S\NP)\(S\NP) .|.|.|O|O|.
```

This extract represents the following sentence:

A play really demands to be read aloud - it needs the sound of the human voice to bring it alive.

Each sentence is preceded by a set of grammatical relations that hold between different components of the sentence. Then, each word in the sentence has a number of tags associated with it. Following the word is its lemma (e.g. *cat* for *cats*, *go* for *went*, etc). The next token is the part of speech (POS) tag of the word. The C5 tag set is used, which contains 57 basic POS tags plus 4 punctuation tags. We are not interested in the other tokens but the last token, which provides corresponding information if the verb is in passive voice.

4.1.2 Grammatical relations

Chapter 3 explained that we are interested in identifying two types of grammatical relations: a noun being the subject of a verb phrase, and a noun being the object of a verb phrase. Table 4.1 summarizes the tags that we are interested in for obtaining those relations.³

Tag	Description	Example
nsubj	encodes binary relations between non-clausal subjects (NPs, PPs) and their verbal heads	The boy eats chocolate. (nsubj eats boy)
dobj	is a binary relation between verbal or prepositional head and the head of the NP to its immediate right	She gave it to Kim. (dobj gave it)
iobj	is a binary relation between a head and the preposition of a PP argument when the PP complement is a NP	Kim flew to Paris. (iobj flew to)
obj2	is a binary relation between verbal heads and the head of the second NP in a double object construction	She gave Kim toys. (obj2 gave toys)
ncmod	encodes binary relations between non-clausal modifiers and their heads	This is happening on stage. (ncmod _ happening on)

Table 4.1 Descriptions of grammatical relation tags

As we'll see in the next subsections, these tags are used to obtain relationships between different parts of speech: between a noun and a verb, between a noun and a preposition, and between a verb and a preposition.

4.1.3 Implementing the main assumption

4.1.3.1 Subject search

³ The descriptions of the tags and some examples are taken from Briscoe (2006).

Given the noun compound, we are searching for verb phrases that appear with the second noun of the compound as a subject. This means, first of all, we are looking for a verb which is in `nsubj` relation with the noun. The following extract contains what we need:

```
(nsubj happening play)
(ncmod _ happening on)
The play is happening on the stage.
```

From this extract we obtain the verb `happen` for the noun `play`. However, we are looking for a verb phrase. If the verb is associated with any prepositions, we want to obtain the latter as well. With the tag `ncmod` the second line of the extract signals the presence of a preposition. Therefore, we conclude that the noun `play` comes as a subject to the verb phrase `happen on`.

As shown in Table 4.1, the tag `iobj` can also signal a presence of a preposition associated with a verb.

An important thing to take into account is that verbs can come in passive voice. As soon as the verb is obtained, we check whether the last token contains the keyword `[pss]`. If it does, then we store the verb in a form corresponding to passive voice. For example, if the verb `make` is found to be in passive voice, then instead of storing `make`, we store `be made`.

To conclude, in order to obtain a verb phrase that comes with the second noun of the compound as a subject, we go through the following steps:

1. Look for verbs such that the following holds:
`(nsubj verb second_noun)`
2. If found, look for prepositions such that any of the following holds:
`(iobj verb prep)`
`(ncmod _ verb prep)`
3. Check for the keyword `[pss]` to appear with the verb. If it does, store the verb in passive voice.

4.1.3.2 Object search

We also need to search for verb phrases appearing with the first noun of the compound as their object. Unlike the case with the subject, here the case when the verb comes with a preposition is treated a bit different from the case when it comes without a preposition.

If the verb doesn't have a preposition, then either `dobj` or `obj2` will join the verb with the noun.

However, if the verb does have a preposition, then instead of the verb being connected to the noun directly via some tag, the preposition will be connected to the noun, and then, the verb to the preposition. To illustrate this, let's have a look at the following example:

```
(dobj on stage)
(ncmod _ happening on)
The play is happening on the stage.
```

Say, we have the compound `stage performance`. The first line signals that there is a preposition that has `stage` as its direct object. We take the preposition and look further to see if there are any verbs associated with it with either `ncmod _` or `iobj`. If there are, then the full verb phrase is obtained. If not, the preposition is dropped and the search continues.

Note that in both cases (with or without a preposition) passive voice determination is done in the same way as discussed for the subject case.

To recap, two parallel sequences of steps are needed in order to find relevant verb phrases for the first noun of the compound. The first sequence finds those verbs that appear without prepositions.

1. Look for verbs such that any of the following holds:
(dobj verb first_noun)
(obj2 verb first_noun)
2. Check for the keyword `[pss]` to appear with the verb. If it does, store the verb in passive voice.

The other sequence finds verb phrases that do contain prepositions.

1. Look for prepositions such that any of the following holds:
(dobj prep first_noun)
(obj2 prep first_noun)
2. If found, look for verbs such that any of the following holds:
(ncmod _ verb prep)
(iobj verb prep)
3. Check for the keyword `[pss]` to appear with the verb. If it does, store the verb in passive voice.

4.1.4 Why BNC?

At this point it is clear how exactly the grammatical relations suggested by the C&C parser are used to find relevant phrases that fit the main assumption. But let's stop for a second

before going into hypernym integration, and understand why I have chosen the BNC as a corpus to learn the relations and not another corpus.

In Chapter 3 we saw that the BNC is a quite large corpus, containing 90 million⁴ words. But there is another large text corpus, Google N-grams. According to the official Google research blog, the five-gram corpus contains 1,176,470,663 five-grams. Why not use this corpus?

During my work I have attempted using this corpus as well. Since the corpus containing the five-grams is extremely large, it would have probably been practically impossible (in terms of execution time) to run the C&C parser on the whole corpus. Even if it were possible, the parsed data would need a couple of times more space on disc because of all the grammatical/POS tags that the parser adds.

In order to avoid this, I had implemented an initial search through the five-grams which copied to a new location only those five-grams which contain either of the nouns in the compound. This made the data to parse and process relatively manageable. Then, the system gave the chosen five-grams to the C&C parser to perform POS tagging and grammatical parsing. At this point I could've used this output in the exact same way as the BNC.

However, the parsed output showed that many words have been assigned to a wrong part of speech. Statistics showed that more than 50% of the words get a wrong part of speech. And if the part of speech is wrong, then the grammatical parsing performs even worse.

The reason behind this is that five-grams are not complete sentences; they are just extracts from a sentence. This makes it hard for the tagger to determine the part of speech, because it takes into account the beginning of a sentence, the end of a sentence and the relationship between all words in the sentence in order to determine the part of speech of a word.

The BNC, in contrast, as stated in the User Reference, has an error rate of 1.14%. This explains the reason for leaving Google N-grams aside for this task, and going on with the BNC.

4.2 WordNet hypernym integration

In Chapter 3 it was mentioned that in order to get more verb phrases in the stage of verb phrase extraction, WordNet hypernyms were integrated into the system. This chapter will explain the details of hypernym integration and unrelated hypernym elimination.

⁴ The full corpus contains 100 million words, but the written part comprises 90% of that.

4.2.1 WordNet senses

The hypernyms of nouns are automatically obtained from WordNet, which is a lexical database of English and is described in detail by Fellbaum (1998). For each noun it can provide the corresponding hypernyms for the different senses that the noun has. For example, the word `jute` has two senses in WordNet:

Sense 1: a plant fiber used in making rope or sacks

Sense 2: a member of a Germanic people who conquered England and merged with the Angles and Saxons to become Anglo-Saxon

For each sense, WordNet provides the hypernym of the word. For Sense 1 the hypernym is `fiber`, whereas for Sense 2 it is `European`. Since the senses are ordered in WordNet by frequency, we are putting a limitation of retrieving only the top two hypernyms, i.e. the hypernyms of the two most frequent senses of the nouns.

4.2.2 Identifying the relevant sense

If the noun has more than one sense, as in the case of `jute` (`fiber` and `member of Germanic people`), then a technique is needed to be able to determine which sense is more appropriate for the given compound. Within the compound `jute product` it's obvious that `fiber` is the appropriate sense, whereas `member of Germanic people` is not that probable. The problem is to make the system answer this question.

It would, of course, be possible to go with the verb phrases obtained for both senses of each noun, and hope that validation will anyways leave out the verb phrases that are not appropriate. However, it's much better to eliminate as much verb phrases as possible in earlier stages, in order not to carry too much trash to the validation phase, which would result in slower performance.

Word sense disambiguation is a large area in computational linguistics by itself. Obviously, this thesis doesn't aim to cover this area. There are many different approaches to solve the disambiguation problem in general. In this thesis a simple technique is applied that proves to perform the needed job quite well.

4.2.3 Method of unrelated hypernym elimination

The idea that the technique follows is to identify the hypernym that has more common verb phrases with the other noun of the compound. In case of `jute product`, the idea is to find out which of the hypernyms of `jute` (`fiber` or `European`) has more common verb phrases with `product`. The chosen measure doesn't directly calculate the number of verb phrases in the intersection. This is because if one hypernym is very frequent in the

language compared to the second hypernym, then it would have many neutral (not revealing any characteristics) verbs in common with the other noun, and would get a higher mark. Our measure rather has some similarities with the Dice coefficient, which is calculated by

$$s = \frac{2|X \cap Y|}{|X| + |Y|}$$

where, in our case, X is the list of verb phrases that appeared with the hypernym, and Y is the list of verb phrases that appeared with the other noun. The Dice coefficient penalizes the words that are too frequent in general, by dividing the number of common verb phrases for the hypernym and the noun by the sum of the number of verb phrases the noun and the hypernym have separately.

Because $|Y|$ is the same for all hypernyms of a given noun (the opposite noun always stays the same), and because we are not interested in absolute measures but only relative marks, we are using the following simplified formula to score each hypernym:

$$s = \frac{|X \cap Y|}{|X|}$$

Hence, for each noun, the hypernym that gets the highest score is preserved, whereas the other hypernyms are eliminated. Evidently, the verb phrases associated with the eliminated hypernyms are also eliminated. With this, the unrelated hypernym elimination phase is over.

4.2.4 Discussion

It's worth stopping here for a moment and understanding why it was necessary to extract all those verb phrases for all hypernyms, and then throw some of them away after elimination. The reason is that if we do not have the verb phrases associated with the hypernyms in the first place, how will we calculate the mentioned measures? We need the verb phrases to be able to determine the similarity of context between the hypernyms and the noun.

4.3 Implementation details

In this section we will look at the structure of the system from the implementation point of view and describe each of the components. We will, however, skip the implementation details of the validators, as well as the `utils.GoogleNGrams` package. These will be discussed in the next chapter when validators are introduced.

4.3.1 General structure

The system is developed in Java 1.6. It is a standalone application that can run on any platform. The classes are separated into logical packages for easy usage. Figure 4.1 shows the structure of the packages. As we can see, there are four top-level packages. The `utils` package provides utilities for dealing with different types of data, such the BNC or Google N-grams. The `structures` package contains data structures needed for representing language, such as verbs or noun compounds. Validation methods are collected into the `validators` package, and the top-level execution methods are in `execution` package.

The full code of all classes can be found in Appendix B.

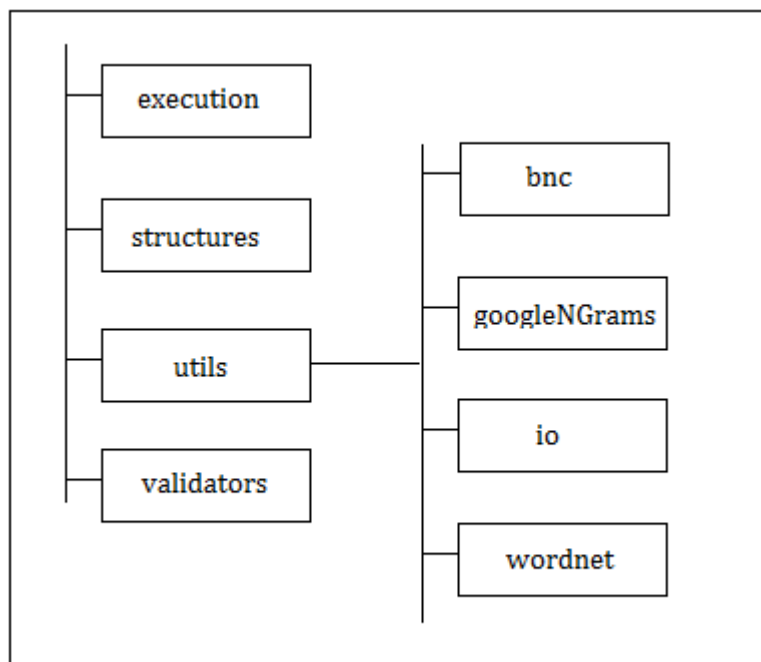


Figure 4.1 Package view of the system

4.3.2 The `execution` package

A single run of the system consists of a number of calls to different methods. As we saw in the previous sections, first the BNC is being processed, after which the unrelated hypernyms are eliminated, and then the intersection of the nouns' verb phrases is identified. We also briefly discussed that there are different validation methods that can be used in any sequence.

Since, especially during testing, it is crucial to be able to run separate components independently, the `execution` package contains a class `ExecutionPoints`, which provides all sorts of methods for running the program from any point, till any point. For example, calling the method `processBNC()` results in only searching the BNC for suitable

verb phrases and writing those into the corresponding files, whereas `startFullCycle()` invokes the whole cycle including validation. This results in high flexibility because if something goes wrong in the last stage of the processing, then instead of restarting the whole cycle, we can restart only the part that went wrong.

This package also contains the start point of the program, which is the `Executor` class. Running the program is all about modifying this class to define the compound and choosing which execution point to use.

4.3.3 The structures package

In order to represent verbs and noun compounds and their features in the system, a class is allocated to each of them.

Thus, `NounCompound` encapsulates the features of noun compounds in which we are interested throughout this project. It can answer simple questions such as which noun takes the role of the subject, or which are the hypernyms of the first noun of the compound.

During BNC processing stage, a file is created for each noun and for each hypernym, and the verb phrases corresponding to each of those are written into the relevant file. Writing into a file makes it possible for other execution points to start directly from the processed verbs, instead of having to rely on this stage all the time. Creating relevant files and disposing them is also one of the responsibilities of `NounCompound`.

The `Verb` class stores information about a single verb phrase and its ranking. During each validation verb phrases get a new ranking. It then provides a method to compare two verb phrases based on their rankings, which is used when ordering the list of verb phrases.

4.3.4 The utils package

This package includes sub-packages, each of which has knowledge about a certain data type. It also includes direct classes that are general utility classes.

`FileNameUtils` provides methods for retrieving the correct file name. For example, given the compound and a hypernym, it returns the full path of the file containing the verb phrases for that hypernym. The class contains constants for the root directories of the corpora and for the output of the program. When running the program on a different platform, these constants need to be changed to adjust to the file system of the platform.

Methods in `GeneralUtils` deal with operations such as calculating the intersection between two lists of verbs. `GrammarUtils` has knowledge about English grammar; therefore questions like what is the plural of a noun or what is the third form of a verb are answered. Note that identifying the third form of a verb is about following some simple

rules; therefore this is implemented in the system. However, finding the plural of a noun is not that straightforward, and I haven't attempted to solve this problem; in this thesis I have found the irregular nouns in the test set by hand and have provided their plural forms in a hard-coded format.

utils.io

This package contains utilities for dealing with input/output (io). The `FileReader` class is there to aid reading large files, which cannot be read with a simple `Scanner`. `IOUtils` provides methods for reading files in specific formats (such as verb followed by a tab followed by its frequency) and writing data into files.

utils.bnc

The knowledge about the format of the BNC is encapsulated into this package. The BNC is read file by file, and each file is processed sentence by sentence. For each sentence, the nouns of the compound and their hypernyms are being searched. Each occurrence is treated independently. For each occurrence, an appropriate verb is searched by following the logic described in Section 4.1.3.

utils.wordnet

This package provides access to the WordNet electronic dictionary and is used to retrieve hypernyms of the nouns. We are using the `RiTa.Wordnet` package developed by Daniel C. Howe to access the dictionary.

4.4 Review of the chapter

In this chapter we looked at the verb phrase extraction phase. The reader was introduced to the structure of the BNC and to the tags for grammatical relations that we are interested in. Then the method of extracting the verb phrases was explained, and finally the reason behind using the BNC as a corpus was revealed.

The second section of the chapter reminded the reader of the need to use WordNet hypernyms as a way of retrieving more verb phrases. The detailed description of unrelated hypernym elimination was then provided.

The final section focused on the implementation details of the whole project, and gave detailed descriptions of the packages used in verb phrase extraction phase.

5. Validation

Verb phrase extraction phase, which was described in the previous chapter, is capable of returning a large number of verb phrases, sometimes even reaching hundreds. However, it is obvious that not all of them are good paraphrases. For example, for the compound `company director` it is highly likely that the verb phrase `talk about` will appear at least once with both nouns. It doesn't mean though, that `company director` is a `director that talks about the company`. This explains the need of being able to rank the verb phrases in a way that the phrases that do paraphrase the compound get a higher rank.

In this chapter we will look at three ways of ranking the paraphrases and then see how they were combined to provide reasonable results.

5.1 Web based validation

The idea of web based validation has been suggested by Butnariu & Veale (2008). As stated in their paper, a paraphrase may be considered to be meaningful if one can find evidence of prior usage of the paraphrase. Web is an excellent place to look for prior usage, because it has enormous amount of text and searching in Web is much faster and convenient than in any text corpora. Nakov (2007) also uses this assumption, but instead of only validating the paraphrases on the web, he actually retrieves the paraphrases from the web using wildcard queries.

In this thesis, web-based validation is one of the validation types that I am using. I am using the assumption that if the paraphrase has appeared on the web, then it might be a good paraphrase, and if it hasn't, then chances are low that it is a sensible one. As we will see later, however, this is not as black and white as it may seem.

5.1.1 Yahoo API

In order to be able to search for paraphrases on the web automatically, we are in need of a search engine and an API that lets us do that. The most popular search engines nowadays are Google and Yahoo. They both provide the possibility to search for exact phrases⁵ using double quotes, and this is what we need.

Google, however, doesn't support an API through which we could send queries to the server. The way to go with Google would be to send the query string directly through a request object. This, however, is not desirable; it reminds me more of hacking the system.

⁵ As we will see later, it's not absolutely exact – both Yahoo and Google allow punctuation marks to appear anywhere inside the phrase.

Yahoo, on the other hand, provides the opportunity to send queries to their server via a simple API, called `WebSearchService`. As a developer, you can register and get a special key, which will allow up to 5,000 queries to be sent from a single IP address during a day.

Since Yahoo is one of the top search engines and provides a flexible API to use, I am using it for web validation purposes.

5.1.2 Query construction

There are some questions which arise when wanting to perform web based validation. How exactly should the query be constructed given the compound and the verb phrase? Should only one query per verb phrase be sent, or maybe multiple queries?

The first idea is to search for the paraphrase the way we think about it, e.g. “minority that consists of immigrants” for immigrant minority and consist of. However, since this format is a more explanatory format, it is less likely to appear in text⁶. Instead, one would expect the format “minority consists of immigrants” to be more frequent in text, and yet, it still does the job we want: it shows evidence of the verb phrase consist of being used to connect the nouns immigrant and minority. To give some idea of numbers, one can easily check that the query “minority that consists of immigrants” gives 0 results, whereas “minority consists of immigrants” returns 6 documents. Hence, we will stick to the non-explanatory, direct format.

In order to understand another difficulty, let’s look at different noun compounds and verb phrases presented in Table 5.1.

Noun compound	Verb phrase	Queries to be searched
immigrant minority	consist of	minority consists of immigrants
house cleaner	clean	cleaner cleans the house
automobile factory	produce	factory produces automobiles factory produces the automobile

Table 5.1 Noun compound query examples

At first glance, there might be nothing different between the queries. Indeed, all of them are made using the rule “noun₂ + verb_phrase + noun₁” presented earlier. However, a closer look shows that the first query doesn’t have any articles and the noun `immigrant` is in plural form, whereas in the second query `house` is in single form and has a definite

⁶ We would expect “minority that consists of immigrants” to appear only in some texts that actually define some specific type of minority.

article the. Furthermore, for automobile factory and produce we can create two queries, which both make sense.

This example shows that for different compounds and verbs there can be neat differences of how exactly the query is constructed. Although the difference is small, it greatly affects the number of documents retrieved from the web search engine. Indeed, Table 5.2 shows different variations of queries that can be constructed from immigrant minority and consist of. The second column shows the number of documents retrieved for each query.

Query	Count
minority consists of immigrants	6
minority consists of the immigrants	0
minority consists of immigrant	0
minority consists of an immigrant	0
minority consists of the immigrant	0

Table 5.2 Different queries for immigrant minority and consist of and their web counts.

If we ignored all types of articles and plural forms, then we would get ranking equal to 0 for the verb phrase consist of. However, that fact is that it is an appropriate paraphrase and coming up with the correct query can result in a rank equal to 6 rather than 0.

A possible solution could be to search for all possible combinations of articles appearing with single/plural forms. The obvious drawback of this approach is that too many queries will be searched per compound, which is not desirable.

The following technique has been used around the problem. During the verb phrase extraction phase, which is performed on the BNC, whenever a verb phrase is found to appear with the first noun of the compound as its object, the noun and the verb phrase are stored in a special map, where the key is the verb phrase and the value is a list of strings. Each string is a concrete version of how the noun appeared, e.g. with or without a definite article, plural or single form. For the compound afternoon rain and the verb phrase fall during the following strings were obtained during verb phrase extraction: “the afternoon” and “the afternoons”. The query will be constructed using only these forms of the noun. As we can see, this saves us the time that we would otherwise waste on searching for all other possible combinations.

For the cases where more than one query is obtained, the sum of the number of retrieved documents for each query is assigned to the verb phrase as its rank.

5.1.3 Implementation details

The class `YahooWebValidator` inside the `validators` package deals with preparing queries from a paraphrase, searching the queries using the Yahoo API and providing a list of web-ranked verb phrases given the compound and a list of verb phrases as an input.

5.1.4 Review of web based validation

Web based validation constructs queries based on the retrieved paraphrases, searches the queries in Yahoo and assigns a rank to each verb phrase, which is equal to the number of documents retrieved. The queries have the format “`noun2 + verb_phrase + noun1`”; whether `noun1` should come in single or plural form, or whether with any articles or not, is based on its previous occurrences in the BNC.

The nice thing about web based validation is that it eliminates⁷ most of the inappropriate paraphrases. However, paraphrases obtained from high frequency verbs are still likely to appear on the web, resulting in a high rank. Another reason why inappropriate paraphrases can get high rank is because the search engine doesn't perform exact phrase matching; it rather allows punctuation marks to be inside the searched phrase. The query “`song says love`” returns 2,250 results, but browsing through the retrieved documents shows that most of those actually contain something like “`song says `love ...'`”.

The next validation method tries to deal with high frequency verbs.

5.2 Mutual information based validation

Mutual information based validation tries to identify how strong the verb phrase is associated with both nouns of the compound. The stronger the association, the higher rank it gives to the paraphrase. High frequency verbs that are not particularly associated with both nouns receive lower ranks as we will see.

5.2.1 Definition of mutual information

As defined by Fano (1961), mutual information between two words `x` and `y`, that have probabilities `P(x)` and `P(y)`, is calculated by

$$I(x, y) = \log_2 \frac{P(x, y)}{P(x)P(y)}$$

This measure compares the probability of the words `x` and `y` appearing together (`P(x,y)`) with the probability of them appearing independently. If `x` and `y` are highly associated, then `P(x, y)` will be much greater than `P(x)P(y)`, and therefore `I(x, y)` will be much greater than 0.

⁷ That is, gives a rank equal to 0.

However, when x and y are slightly associated or not associated at all, then $I(x, y)$ will be around 0 or less than 0.

Church & Hanks (1990) suggest calculating the independent probabilities $P(x)$ by counting the number of times x appears in the corpus, $C(x)$, and normalize by the number of words in the corpus, N . Similarly, the joint probability $P(x, y)$ can be computed by counting the number of times x and y appear together, $C(x,y)$, and normalizing by the size of the corpus, N . Plugging in these number into the formula, we get the following:

$$I(x, y) = \log_2 \frac{C(x, y)}{N} \frac{N^2}{C(x)C(y)} = \log_2 \frac{N * C(x, y)}{C(x)C(y)}$$

5.2.2 Adjusting to our needs

How do we apply the mutual information measure to give a ranking to the verb phrase consist of for the noun compound immigrant minority? Since we are interested in the level of association of the verb phrase with both nouns, we calculate the mutual information between the verb and each of the nouns, and then multiply the two.

$$I(\textit{paraphrase}) = I(n_1, v) * I(n_2, v)$$

Since our goal is to rank the paraphrases according to their mutual information measure, we are not interested in the absolute values. The size of the corpus and the counts for the nouns are the same for all pairs, which is why we can remove them from the formula. Furthermore, logarithm is a monotonic function; therefore its absence will not influence the relative ranking. The simplified formula to be used for mutual information between two words is:

$$I(x, y) = \frac{C(x, y)}{C(y)}$$

5.2.3 Implementation details

Within the package `validators` the class `MutualInfoValidator` is responsible for calculating the mutual information measure for a paraphrase.

The $C(x, y)$ joint counts are readily available after the verb extraction phase, because every time a verb is seen to occur with one of the nouns, its counter is incremented.

The $C(x)$ counts are obtained from the BNC simply by counting the number of times the word appears. Since the BNC contains many tags and grammatical relations, which are not useful when counting, I have written a script which produced the “clean” version of the

BNC. This version doesn't contain any tags or relations; it only contains the lemmas for each word. For example, here is a sentence from the "clean" processed BNC:

there be other hazard for the reader of catalogue .

All word counts are obtained from the "clean" BNC to reduce execution time.

5.2.4 Review of mutual information based validation

The formula of $I(x,y)$ makes it clear that if the verb y is an extremely frequent one ($C(y) \gg 0$), then in order for it to get a high score, the frequency of the noun and the verb (x and y) appearing together should be quite high too. Mutual information measure of a paraphrase is high when the verb phrase is strongly associated with both nouns, and low, if not. This helps to penalize high frequency verbs such as *be*, *do* and *have*, if they are not indeed attached to the compound.

5.3 N-gram probability based validation

The last validation method that I have looked at in this thesis is based on a probabilistic model called N-gram model. The probability for each paraphrase is calculated, and more likely paraphrases get a higher rank.

5.3.1 N-gram models

In this section we will present the techniques used to calculate the probability of a sequence of words w_1, \dots, w_n using the bigram model⁸.

The first thing to do is to decompose $P(w_1, \dots, w_n)$ using the chain rule of probability:

$$P(w_1, \dots, w_n) = P(w_1)P(w_2 | w_1) \dots P(w_n | w_1, \dots, w_{n-1}) = \prod_{k=1}^n P(w_k | w_1 \dots w_{k-1})$$

As described by Jurafsky & Martin (2009), the N-gram model assumes that the probability of a word given the entire history can be approximated by the history of only the last few words. The bigram model, which we will be using, approximates this probability by the history of the preceding word only. That is,

$$P(w_k | w_1 \dots w_{k-1}) \approx P(w_k | w_{k-1})$$

The bigram assumption turns the probability of a sequence into the following:

$$P(w_1, \dots, w_n) \approx \prod_{k=1}^n P(w_k | w_{k-1})$$

⁸ Bigram model is a subtype of the N-gram model when $N = 2$.

In order to estimate the bigram probabilities, we use maximum likelihood estimation (MLE). It is achieved by simple word count in the corpus, followed by normalization.

$$P(w_k | w_{k-1}) = \frac{C(w_{k-1}w_k)}{C(w_{k-1})}$$

5.3.2 Ranking

For each paraphrase the probability of the following sequence is estimated using the bigram model:

noun₂ + verb_phrase + noun₁

where the verb_phrase is in its third form. Note that here we do not care about articles or plural forms as in web-based validation. The reason is that if we accounted for articles, for example, then the probability would look something like this:

$$P(\text{paraphrase}) =$$

$$P(\text{verb_phrase} | \text{noun}_2) * P(\text{article} | \text{verb_phrase}) * P(\text{noun}_1 | \text{article})$$

Note that this is not desirable, because it relies on computing the probability of an article appearing after a verb phrase, and a noun appearing after an article. These terms do not reveal the association level of the verb phrase with the nouns themselves. If we do not include articles, though, we will have a nicer product of probabilities:

$$P(\text{paraphrase}) = P(\text{verb_phrase} | \text{noun}_2) * P(\text{noun}_1 | \text{verb_phrase})$$

As we can see, this product reveals the true relationship between the verb phrase and both nouns.

5.3.3 Implementation details

The `NGramValidator` class inside the `validators` package is responsible for calculating the N-gram based probability for a paraphrase. The calculations are simplified to computing word and bigram counts in a corpus.

We could have used the BNC for the counting purposes. However, this is where Google N-grams are much more convenient. This corpus consists of five parts: unigrams, bigrams, trigrams, four-grams and five-grams. For each N-gram, a frequency of appearing on the web is provided. Below is an extract of the trigram corpus taken from the Official Google research blog.

```
ceramics collectables collectibles 55  
ceramics collectables fine 130
```

```
ceramics collected by 52
ceramics collectible pottery 50
ceramics collectibles cooking 45
ceramics collection , 144
ceramics collection . 247
ceramics collection </S> 120
ceramics collection and 43
```

Although the size of the corpus is much bigger than that of the BNC, searching for an N-gram is much faster. The N-grams appear in an alphabetical order, which makes it possible to create an index file that tells which letter is in which file.

Now, to calculate $P(w_k | w_{k-1}) = \frac{C(w_{k-1}w_k)}{C(w_{k-1})}$, the only thing to be done is to search for the

bigram ($w_{k-1} w_k$) in the bigram corpus, search for w_{k-1} in the unigram corpus, and divide the frequency of the first by the frequency of the second. The search in unigram/bigram corpora is implemented in `utils.googleNGrams` package.

5.3.4 Comparison of N-gram and mutual information based validations

As we could see, the N-gram validation method ranks the paraphrases according to how likely the paraphrase is to appear using a simple bigram model.

If at this point the reader remembers the formula for calculating the mutual information based rank of a paraphrase, he will be able to see that it doesn't differ much from the formula of the N-gram probability based rank. In fact, $I(x, y)$ and $P(w_k | w_{k-1})$ both compute the correlation degree of two words and their adjusted formulas look similar. This is an interesting conclusion. Although the N-gram model only makes use of the Markov assumption and doesn't have to do anything with the concept of mutual information, when used to compare sequences of words rather than get the absolute values, both measures tend to do more or less the same thing.

Conceptually, they do reach the same result. However, in this thesis I have used different implementation details for each of the validation methods, which make them behave a little bit different and produce somewhat different results.

Firstly, the mutual information validator always calculates $I(x, y)$ twice – for the verb phrase with each of the nouns. In the N-gram model, however, $P(w_k | w_{k-1})$ is calculated for all adjacent words: that is, if a verb phrase consists of two words (a verb and a preposition), then it will actually involve calculating three probabilities. For instance, for the compound `immigrant minority` and the verb phrase `consist of`, the mutual information validator calculates $I(\text{immigrant}, \text{consist of})$ and $I(\text{minority},$

consist of). N-gram probability validator computes three probabilities: $P(\text{consist} \mid \text{minority})$, $P(\text{of} \mid \text{consist})$ and $P(\text{immigrant} \mid \text{of})$.

Secondly, for mutual information measure we calculate $I(n_1, v)$ and $I(n_2, v)$, whereas in the N-gram model the order is different: it's always $P(w_k \mid w_{k-1})$.

Finally, the BNC is used for obtaining counts for the mutual information measure. $C(v, n)$ the number of times that v appears with n as its object/subject (depending on whether it's the first or the second noun of the compound). In the N-gram model, in contrast, the counts are obtained from Google N-grams, and $C(x, y)$ is simply the bigram count obtained from the corpus.

Because of these differences, the two methods gave slightly different results in my experiments. However, the analysis showed that in average both methods perform equally well and therefore the final evaluation and analysis are only performed using one of them: the mutual information measure.

5.4 Sequences of validation methods

Web validation is good for eliminating paraphrases that don't appear to have prior usage, and therefore, are not likely to be sensible. It also creates a good ranking based on web evidence. Mutual information based validation eliminates high frequency verb phrases which are not strongly attached to the noun compound. Hence, each validation method is useful in a certain, unique way.

Instead of trying to identify which validation method produces better results, I use both methods in a sequence in order to get the benefit of both. The sequence is the following:

- Web-based validation → Mutual information based validation → Final web-based re-ranking

As a first step, web validation is performed. Verb phrases that show no prior usage (i.e. get a rank equal to 0) are eliminated. Out of the rest of the verb phrases, top ten are passed on to the next validation phase, which is mutual information validation. Re-ranking is performed, where high frequency verbs are eliminated. The produced list of five verb phrases is again resorted based on web evidence, and a list of top three verb phrases is sent as an output.

Appendix A contains a list of all test noun compounds together with the output that the system generated for them.

6. Evaluation and results

Previous chapters described the method used to solve the proposed problem of noun compound interpretation. Now it is necessary to decide whether the system did a good job or not, otherwise there is little value in the work done. This chapter's goal is to present what techniques were used to evaluate the system, and what results were obtained.

6.1 Data

SemEval 2010 Task 9, proposed by Butnariu et al. (2009), comes with a set of noun compounds to use for training, and another set to use for testing. Both sets contain 250 noun compounds. In this thesis the corresponding sets are used for training and testing purposes respectively.

The test set hasn't been viewed until the final tests. During final testing, 50 noun compounds were randomly chosen from the set of 250, and the system was executed, having those 50 compounds as input.

The sample size of test noun compounds was limited by logistical constraints. As it will be shown in the next section, human judges were involved in manually rating the paraphrases for each compound, therefore presenting a list of 250 compounds to a human judge wouldn't be plausible. It would be too tedious for a human to judge $250 * 3 = 750$ paraphrases⁹, hence the quality of judgment would drop.

Therefore, the number 50 was chosen as a compromise. It's small enough for human judges not to get bored of scoring, but it's large enough to ensure proper evaluation.

6.2 Human judge feedback

After the output is obtained for the 50 test noun compounds, it's time to judge the paraphrasing ability of the system.

Since there is no readily available list of paraphrases with "ideal" ranking for a given noun compound, it is somewhat hard to evaluate the generated paraphrases. But many authors acknowledge that a paraphrase can be considered good if people generally agree that it's good. Therefore, for the purposes of this thesis I am using human feedback to evaluate the system.

6.2.1 Scoring mechanism

Twenty native English speakers have been recruited for the evaluation task. The list of compounds and their paraphrases have been given to each rater in a format identical to

⁹ For each compound up to 3 top paraphrases are returned as output.

that of the example in table 6.1. Raters have been given clear instructions on how to interpret the paraphrases and how to rate those, as well as a number of rated examples.

apple cake
contain
be made of
be made from

Table 6.1 Example of the format presented to human judges for evaluation

The rating system that has been proposed consists of three scores. Table 6.2 lists the scores and the way they are interpreted.

Score	Interpretation
1	perfect paraphrase
2	somewhat acceptable
3	absolutely inappropriate

Table 6.2 Scores used for evaluation by human judges

Having more than three scores is usually confusing for a human brain. Imagine having a scale from 1 to 4. What would be the difference between the scores 2 and 3 then? Having only two scores – positive and negative, in contrast, is too few for this task. Suppose the compound `software developer`. The verb `produce` is absolutely fine, and the verb `buy` is inappropriate, but what about the verb `make`? Maybe some people would think that it's a good paraphrase, however more strict people would want to have some score in between black and white to say that the verb phrase might make sense, but it's not a perfect paraphrase. Hence, a three level scale was found to be the most appropriate for this task.

6.2.2 Agreement level

Researchers tend to agree that some data generated by a number of raters is reliable if the scores given by different coders generally agree with each other. If there is little agreement between the raters, then the conclusion is that the rating scheme is inappropriate for the given task and/or the instructions were not clear to the raters. High agreement level between the raters shows that the rated data is reliable and consistent.

Different agreement measures have been suggested in the literature. Some of them measure only agreement between two raters; others put restrictions on the type of task being performed by the raters. For my task I have chosen to use the alpha measure suggested by Krippendorff (1980, 2004), which accounts for agreement expected by

chance, is suitable for any number of raters and any level of measurement.¹⁰ We are interested in interval scale because it assumes that the scores 1 and 3 are further than the scores 1 and 2 or 2 and 3, which is exactly what we need.

The alpha measure for the rated data was calculated using the SPSS macro provided by Hayes & Krippendorff (2007). The output of the macro is shown in Table 6.3.

```
Run MATRIX procedure:

Krippendorff's Alpha Reliability Estimate

Interval          Alpha      Units      Obsrvrs      Pairs
Interval          .6479    146.0000   20.0000  27721.0000

Judges used in these computations:
Columns  1 - 8
  JUDGE1  JUDGE2  JUDGE3  JUDGE4  JUDGE5  JUDGE6  JUDGE7  JUDGE8
Columns  9 - 16
  JUDGE9  JUDGE10  JUDGE11  JUDGE12  JUDGE13  JUDGE14  JUDGE15  JUDGE16
Columns 17 - 20
  JUDGE17  JUDGE18  JUDGE19  JUDGE20

Examine output for SPSS errors and do not interpret if any are found

----- END MATRIX -----
```

Table 6.3 Alpha measure macro output

Whether the produced alpha is good or bad has been an active topic in research. Many researchers have tried to identify which level to call an adequate level of agreement. For the area of medical diagnosis Landis & Koch (1977) have reported that values above 0.4 can be considered to contain moderate strength of agreement, and values greater than 0.6 indicate substantial agreement. Krippendorff (1980) has described values greater than 0.8 as highly reliable, and values between 0.67 and 0.8 as allowing “tentative conclusions”. However, this has been said for annotation tasks, and not rating tasks. Artstein & Poesio (2008) doubt the existence of a single cutoff point which would be appropriate for all tasks. Furthermore, they mention that the magical 0.67 defined by Krippendorff is often proved to be impossible to reach in computational linguistics research. Since the task in this thesis is not annotation but rating, where the subjectivity factor is higher, the obtained alpha of 0.6479 has been considered high enough to rely on the results and draw conclusions from them.

¹⁰ There are four levels of measurement – nominal, ordinal, interval and ratio.

6.3 Metrics and results

This section describes the metrics that have been used to evaluate the overall performance of the system on paraphrasing noun compounds, as well as the results obtained based on the defined metrics.

6.3.1 Defining metrics

The performance of the system has been measured based on averaging the ratings given by the judges. Since each test noun compound comes with three ordered paraphrases, averaging has been performed for each paraphrase individually in the first place:

$$S(\text{paraphrase}) = \frac{\sum_{j \in \text{judges}} \text{score}(j, \text{paraphrase})}{N}$$

where N is the number of judges, $\text{score}(j, \text{paraphrase})$ is the score given to the paraphrase by judge j , and $S(\text{paraphrase})$ is the resulting averaged score for paraphrase.

Then, a number of percentage based metrics are applied to the averaged scores to get the overall performance across all test noun compounds. For instance, the following types of questions are answered: what percentage of noun compounds has a 1st ranked paraphrase with an average score less than 1.5¹¹? What percentage of noun compounds has a 1st or 2nd ranked paraphrase which has an average score less than 1.5? Table 6.4 summarizes all metrics that have been used and their short notations.

Notation	Metric description
1st_ranked	Percentage of noun compounds whose 1st ranked paraphrase has an average score less than some x.
2nd_ranked	Percentage of noun compounds whose 2nd ranked paraphrase has an average score less than some x.
3rd_ranked	Percentage of noun compounds whose 3rd ranked paraphrase has an average score less than some x.
1st_or_2nd_ranked	Percentage of noun compounds for which either 1st or 2nd ranked paraphrases have an average score less than some x.
any_rank	Percentage of noun compounds for which at least one of the paraphrases has an average score less than some x.

Table 6.4 Percentage based metrics

¹¹ To remind the reader, low score indicates a good paraphrase. The best average score for a paraphrase is 1.0 and is reached when all judges have agreed to rate the paraphrase as a “perfect paraphrase”, i.e. have given a score equal to 1.

6.3.2 Results

The metrics described above need a target average score x , with regard to which the percentages should be computed. The best average score is 1.0 which can be reached when all judges have rated the paraphrase with a score equal to 1. Average scores below 1.5 are reached when more judges score the paraphrase with 1 than with any other score. Therefore, the goal is to find the percentage of compounds for which the average scores of the paraphrases fall between 1.0 and 1.5, because this interval indicates very good feedback from the judges.

Table 6.5 summarizes the results obtained using different metrics and different target scores.

	≤ 1.5	≤ 1.4	≤ 1.3	≤ 1.2	≤ 1.1	≤ 1.0
1st_ranked	86	84	70	62	58	42
2nd_ranked	74	70	64	64	48	28
3rd_ranked	68	68	60	48	38	20
1st_or_2nd_ranked	94	92	86	82	74	56
any_rank	98	98	94	92	82	64
Table 6.5 Percentage results obtained using different metrics (on the left) and different target scores (on the top)						

The results are indeed promising. If only the 1st ranked paraphrase for each compound is considered, then 86% of paraphrases are judged as “perfect paraphrase” by most judges. Furthermore, 42% of paraphrases are voted solid to be “perfect paraphrase”.

Even at the level of 2nd and 3rd ranked paraphrases more than half of the noun compounds have an average score ≤ 1.5 .

If the 1st and the 2nd ranked paraphrases are considered together, then the results are even better. 94% of all compounds have either a 1st or 2nd ranked paraphrase which is considered perfect by most voters; 56% of those have been given a score of 1 by all voters.

Finally, 98% of all compounds have at least one paraphrase that was rated as perfect by most voters and 64% of all compounds have at least one paraphrase that was solidly voted as perfect.

7. Conclusions

The aim of this thesis was to tackle the problem of noun compound interpretation. To accomplish the task, a two-stage method which uses verbs and prepositions to paraphrase a given compound, was proposed and successfully implemented. The first stage searches a large text corpus to find possible verb phrases that could paraphrase the compound. The output of this stage is then passed on to the second, validation phase, which filters out non sensible paraphrases and outputs a ranked list of up to three best paraphrases.

Three types of validation techniques have been researched and implemented: mutual information and N-gram model validations, which are both corpus based techniques. The third technique is web validation which checks the validity of a paraphrase on the web using a search engine.

The implementation of the system forms the substantial part of the work. The system is designed following the requirements of object oriented programming. The advantages of the system which include its usability, extendibility and flexibility can be attributed to its design. The system is subdivided into logical packages and classes. Furthermore, each unit of work, such as BNC processing or web validation, is designed to be able to act independently. The system can be executed on any platform with minimal changes involved.

In order to evaluate the quality of the system, human judges were recruited and asked to rate the paraphrases generated by the system. The results are promising. Firstly, the relatively high agreement level between the judges despite the high subjectivity of the task inspires more confidence in the ratings that they provided. Secondly, the high marks obtained by different percentage based metrics show that the automatically obtained paraphrases are generally well accepted by humans, which underscores the importance of the system.

A limitation of the system is that the system's full cycle's running time for a single noun compound is around 20 minutes on an Intel Core 2 Duo computer, with 1 GB RAM and 2.33 GHz speed. However, given that the system performs multiple string searches in a 5 GB text corpus and sends tens of queries to a web search engine, I believe that this performance is acceptable. Furthermore, if this task is to be performed in production systems, a couple of times faster computers can be used, resulting in much faster performance.

Applications of the designed system can be found in the areas of machine translation, information retrieval, information extraction and question answering. For example, as described by Butnariu et al. (2009), if an information retrieval system has a noun

compound interpretation component in it, then given a query that contains a noun compound, it can paraphrase it and use the paraphrasing verb phrase to aid page ranking.

The designed system is capable of working with any noun compounds consisting of two nouns. Further work might include extension of the system so that it works with compounds consisting of any number of nouns.

Another direction to look at for extending the system would be to treat differently noun compounds which include a nominalization. For example, given the compound `house cleaner`, the system can find that `cleaner` is derived from the verb `clean`, therefore pass directly to the validation phase and check whether `clean` or its hypernyms are valid paraphrases for the compound.

References

- Artstein, R. & Poesio, M. (2008). Inter-Coder Agreement for Computational Linguistics. *Computational Linguistics*, 34(4).
- Baldwin, T. & Tanaka, T. (2004). Translation by machine of complex nominals: Getting it right. In *Proceedings of the ACL 2004 Workshop on Multiword Expressions: Integrating Processing*, 24-31.
- Briscoe, E. (2006). *An Introduction to Tag Sequence Grammars and the RASP System Parser*, University of Cambridge, Computer Laboratory Technical Report 662.
- Butnariu, C., Kim, S., Nakov, P., Ó Séaghdha, D., Szpakowicz, S. & Veale, T. (2009). SemEval-2010 Task 9: The Interpretation of Noun Compounds Using Paraphrasing Verbs and Prepositions. In *Proceedings of the SemEval-2010 Workshop*.
- Butnariu, C. & Veale, T. (2008). A concept-centered approach to noun-compound interpretation. In *Proceedings of the 22nd International Conference on Computational Linguistics (COLING-08)*, 81-88.
- Church, K. & Hanks, P. (1990). Word association norms, mutual information and lexicography. *Computational Linguistics*, 16(1).
- Downing, P. (1977). On the creation and use of English compound nouns. *Language*, 53(4):810-842.
- Fano, R. (1961). *Transmission of Information: A Statistical Theory of Communications*. MIT Press, Cambridge, MA.
- Fellbaum, C. (1998). *WordNet: An electronic lexical database*, MIT press Cambridge, MA.
- Gagné, C. & Shoben, E. (1997). Influence of thematic relations on the comprehension of modifier-noun combinations. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 23(1):71-87.
- Girju, R., Moldovan, D., Tatu, M. and Antohe, D. (2005). On the semantics of noun compounds. *Computer Speech and Language*, 19(4):479-496.
- Hayes, A. F. & Krippendorff, K. (2007). Answering the call for a standard reliability measure for coding data. *Communication Methods and Measures*, 1, 77-89
- Jurafsky, M. & Martin, J. (2009). *Speech and Language Processing*. 2nd edition. Upper Saddle River, New Jersey, 117-125.

Krippendorff, K. (1980). *Content Analysis: An Introduction to Its Methodology*, chapter 12. Sage, Beverly Hills, CA.

Krippendorff, K. (2004). *Content Analysis: An Introduction to Its Methodology*. 2nd edition, chapter 11. Sage, Thousand Oaks, CA.

Landis, J. R. & Koch, G. G. (1977). The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174.

Lauer, M. (1995). *Designing Statistical Language Learners: Experiments on Compound Nouns*. PhD thesis, Macquarie University.

Levi, J. (1978). *The Syntax and Semantics of Complex Nominals*. Academic Press, New York.

Nakov, P. & Hearst, M. (2006). Using verbs to characterise noun-noun relations. In *Proceedings of the 12th International Conference on Artificial Intelligence: Methodology, Systems and Applications*, 233-244.

Nakov, P. (2007). *Using the Web as an Implicit Training Set: Application to Noun Compound Syntax and Semantics*. PhD thesis, EECS Department, University of California, Berkeley.

Nastase, V. & Szpakowicz, S. (2003). Exploring noun-modifier semantic relations. In *Proceedings of the 5th International Workshop on Computational Semantics*, 285-301.

The British National Corpus, version 3 (BNC XML Edition). (2007). Distributed by Oxford University Computing Services on behalf of the BNC Consortium. URL: <http://www.natcorp.ox.ac.uk/>

Appendix A - Output on the test set

This appendix contains the top three paraphrasing verb phrases outputted by the system for the final test set consisting of 50 noun compounds.

afternoon rain	entrance stair	light bulb	snow ball
1. come in	1. lead from	1. emit	1. be made of
2. begin in	2. lead to	2. provide	2. be covered in
3. continue into	3. return to	3. give	3. get
anatomy professor	exam anxiety	military assault	steel frame
1. lecture on	1. be associated with	1. be in	1. be made of
	2. be caused by	2. be committed by	2. be made from
	3. be involved in	3. be recorded in	3. be of
automobile factory	extinction theory	morning class	street scene
1. manufacture	1. explain	1. begin in	1. be filmed on
2. produce	2. lead to	2. start in	2. show
3. fuel	3. suggest	3. meet in	3. be shot in
band concert	film music	mountain country	student price
1. be played by	1. be written for	1. be surrounded by	1. be charged to
2. be performed by	2. be used in	2. be dominated by	2. be paid by
3. be given by	3. play in	3. be in	3. include
chemistry laboratory	food products	mystery novel	symphony orchestra
1. study	1. include	1. include	1. perform
2. test	2. be used in	2. read	2. play
3. include	3. be used for	3. capture	3. conduct
chest pain	furniture company	opposition coalition	terrorist activity
1. be in	1. offer	1. be formed in	1. be used by
2. go through	2. make	2. declare	2. be undertaken by
3. start in	3. sell	3. join	3. be associated with
city dweller	government agency	paper tray	theatre orchestra
1. live in	1. be established by	1. be lined with	1. play
2. move to	2. open	2. be made from	2. perform at
3. migrate to	3. be funded by	3. contain	3. be in

cold virus	hair follicle	petroleum wealth	transportation equipment
1. cause	1. produce	1. be based on	1. be used for
2. be associated with	2. grow	2. come from	2. facilitate
3. include	3. contain	3. be derived from	3. provide
company car	health problem	poultry products	vase painting
1. be made by	1. be related to	1. include	1. include
2. be produced by	2. be associated with	2. keep	2. depict
3. be built by	3. develop	3. contain	3. be in
concert hall	information source	property law	war crime
1. host	1. provide	1. require	1. be committed during
2. be used for	2. be based on	2. allow	2. be at
3. be followed by	3. contain	3. protect	3. be related to
construction materials	jute product	recreation area	welfare agency
1. be used in	1. include	1. be used for	1. be dedicated to
2. be used for		2. include	2. be concerned with
3. be needed for		3. enjoy	3. be involved in
county town	laboratory application	satellite data	
1. be situated in	1. include	1. be obtained from	
2. be in	2. be developed in	2. be collected by	
3. be scattered throughout	3. be in	3. be provided by	
desert storm	laser technology	sea monster	
1. hit	1. use	1. live in	
2. be lost in	2. include	2. be in	
3. occur in	3. be based on	3. be associated with	

Appendix B – Source code

execution.Executor

```
package execution;

public class Executor {

    public static void main(String[] args){
        NounCompound nc = new NounCompound("health", "problem");
        ExecutionPoints.startFromWebRanked(nc);
    }

}
```

execution.ExecutionPoints

```
package execution;

/**
 * Provides methods for executing different
 * units of the program, such as processing the BNC,
 * performing web validation or executing the full cycle.
 */
public class ExecutionPoints {

    /**
     * Processes the BNC and writes the output into
     * the corresponding file
     * @param nc noun compound
     */
    public static void processBNC(NounCompound nc){
        nc.createFiles();
        BNCFileProcessor.process(new File(FileNameUtils.BNC_ROOT), nc);
        nc.disposeFiles();
        IOUtils.writeObjVersionsToFile(FileNameUtils.getObjVersionsName(nc),
        BNCSentenceProcessor.objVersions);
    }

    /**
     * Performs the full cycle, including verb phrase extraction,
     * unrelated hypernym elimination and validation
     * @param nc noun compound
     */
    public static void startFullCycle(NounCompound nc){
        processBNC(nc);
        eliminateHypernyms(nc);

        Map<String, Map<String, Integer>> map = getMap(nc);
        List<String> inter = getIntersectionVerbs(nc, map);

        List<Verb> webRankedList = getWebRankedList(nc, inter);
        List<Verb> mutInfoRankedList = getMutInforRankedList(nc, webRankedList, map);
        List<Verb> doubleRankedList = getDoubleRankedList(nc, mutInfoRankedList);
        List<Verb> ngramList = getNGramRankedList(nc, doubleRankedList);
    }

    /**
     * Performs web validation and writes the output
     * into the corresponding file
     * @param nc noun compound
     */
    public static void processWebOnly(NounCompound nc){
        restoreObjVersions(nc);
        eliminateHypernyms(nc);
    }
}
```

```

        Map<String, Map<String, Integer>> map = getMap(nc);
        List<String> inter = getIntersectionVerbs(nc, map);

        getWebRankedList(nc, inter);
    }

/**
 * Performs all units except for N-gram model
 * based validation
 * @param nc    noun compound
 */
public static void processAllButNGrams(NounCompound nc){
    restoreObjVersions(nc);
    eliminateHypernyms(nc);

    Map<String, Map<String, Integer>> map = getMap(nc);
    List<String> inter = getIntersectionVerbs(nc, map);

    List<Verb> webRankedList = getWebRankedList(nc, inter);
    List<Verb> mutInfoRankedList = getMutInforRankedList(nc, webRankedList, map);
    List<Verb> doubleRankedList = getDoubleRankedList(nc, mutInfoRankedList);
}

/**
 * Performs all units except for BNC processing.
 * Prerequisite: BNC is already processed.
 * @param nc    noun compound
 */
public static void startFromBNCProcessed(NounCompound nc){
    restoreObjVersions(nc);
    eliminateHypernyms(nc);

    Map<String, Map<String, Integer>> map = getMap(nc);
    List<String> inter = getIntersectionVerbs(nc, map);

    List<Verb> webRankedList = getWebRankedList(nc, inter);
    List<Verb> mutInfoRankedList = getMutInforRankedList(nc, webRankedList, map);
    List<Verb> doubleRankedList = getDoubleRankedList(nc, mutInfoRankedList);
    List<Verb> ngramList = getNGramRankedList(nc, doubleRankedList);
}

/**
 * Performs mutual information, N-gram model and second
 * web validation.
 * Prerequisite: initial web validation is already performed.
 * @param nc    noun compound
 */
public static void startFromWebRanked(NounCompound nc){

    restoreObjVersions(nc);
    eliminateHypernyms(nc);

    Map<String, Map<String, Integer>> map = getMap(nc);

    List<Verb> webRankedList = IOUtils.readRankedFile(FileNameUtils.getWebRankedName(nc));
    List<Verb> mutInfoRankedList = getMutInforRankedList(nc, webRankedList, map);
    List<Verb> doubleRankedList = getDoubleRankedList(nc, mutInfoRankedList);
    List<Verb> ngramList = getNGramRankedList(nc, doubleRankedList);
}

/**
 * Performs only N-gram model validation.
 * Prerequisite: all other validations are performed.
 * @param nc    noun compound
 */
public static void startFromDoubleRanked(NounCompound nc){
    List<Verb> doubleRankedList = IOUtils.readRankedFile(FileNameUtils.getDoubleRankedName(nc));
    getNGramRankedList(nc, doubleRankedList);
}

private static void eliminateHypernyms(NounCompound nc){

```

```

        Map<String, Map<String, Integer>> sepMap = IOUtils.readVerbFileSeparately(nc);
        GeneralUtils.eliminateUnrelatedHypernyms(nc, sepMap);
    }

    private static void restoreObjVersions(NounCompound nc){
        BNCSentenceProcessor.objVersions =
        IOUtils.readObjVersionsFromFile(FileNameUtils.getObjVersionsName(nc));
    }

    private static Map<String, Map<String, Integer>> getMap(NounCompound nc){
        Map<String, Map<String, Integer>> map = IOUtils.readVerbFile(nc);
        return map;
    }

    private static List<String> getIntersectionVerbs(NounCompound nc, Map<String, Map<String, Integer>> map){
        return GeneralUtils.getIntersectionVerbs(nc, map);
    }

    private static List<Verb> getWebRankedList(NounCompound nc, List<String> inter){
        List<Verb> webRankedList = YahooWebValidator.getWebRankedVerbs(nc, inter);
        IOUtils.writeListToFile(FileNameUtils.getWebRankedName(nc), webRankedList);
        return webRankedList;
    }

    private static List<Verb> getMutInforRankedList(NounCompound nc, List<Verb> webRankedList, Map<String,
    Map<String, Integer>> map){

        List<Verb> mutInfoRankedList = MutualInfoValidator.getMutualInfoRankedList(nc, map, webRankedList);
        IOUtils.writeListToFile(FileNameUtils.getMutInfoRankedName(nc), mutInfoRankedList);
        return mutInfoRankedList;
    }

    private static List<Verb> getDoubleRankedList(NounCompound nc, List<Verb> mutInfoRankedList ){

        List<Verb> doubleRankedList = YahooWebValidator.getDoubleRankedList(nc, mutInfoRankedList);
        IOUtils.writeListToFile(FileNameUtils.getDoubleRankedName(nc), doubleRankedList);
        return doubleRankedList;
    }

    private static List<Verb> getNGramRankedList(NounCompound nc, List<Verb> doubleRankedList){
        List<Verb> ngramList = NGramValidator.getNGramRankedList(nc, doubleRankedList);
        IOUtils.writeListToFile(FileNameUtils.getNGramRankedName(nc), ngramList);
        return ngramList;
    }
}

```

structures.NounCompound

```

package structures;

/**
 * Represents a noun compound
 */
public class NounCompound {

    private String noun1;
    private String noun2;
    private List<String> noun1Hypernyms;
    private List<String> noun2Hypernyms;
    private Map<String, FileWriter> files;

    /**
     * Constructs a noun compound object given the two
     * nouns as input. Also, initializes the hypernym lists
     * of the nouns and creates the compound directory in the
     * filesystem.
     * @param noun1      the first noun of the compound
     * @param noun2      the second noun of the compound
     */
    public NounCompound(String noun1, String noun2) {
        super();
    }
}

```

```

        this.noun1 = noun1;
        this.noun2 = noun2;
        WordnetUtils w = new WordnetUtils();
        noun1Hypernyms = w.getHypernyms(noun1);
        noun2Hypernyms = w.getHypernyms(noun2);
        w.dispose();
        File dir = new File(FileNameUtils.getDir(this));
        if (!dir.exists()){
            dir.mkdir();
        }
        files = new HashMap<String, FileWriter>();
    }

    public String getNoun1() {
        return noun1;
    }

    public String getNoun2() {
        return noun2;
    }

    /**
     * Removes all hypernyms of the noun, except
     * for the hypernym in the input.
     * @param hyp hypernym, which is not to be removed
     * @param noun noun
     */
    public void leaveOneHypernym(String hyp, String noun){
        System.out.println(hyp + " left for " + noun);
        if (noun1.equals(noun)){
            noun1Hypernyms.clear();
            noun1Hypernyms.add(hyp);
        } else if (noun2.equals(noun)){
            noun2Hypernyms.clear();
            noun2Hypernyms.add(hyp);
        }
    }

    /**
     * Returns all hypernyms of the noun.
     * @param noun noun
     * @return list of hypernyms
     */
    public List<String> getHypernyms(String noun){
        if (noun1.equals(noun)){
            return noun1Hypernyms;
        } else if (noun2.equals(noun)){
            return noun2Hypernyms;
        }
        return null;
    }

    /**
     * Returns the noun corresponding to the hypernym.
     * @param hyp hypernym
     * @return noun
     */
    public String getNoun(String hyp){
        if (hyp.equals(noun1) || noun1Hypernyms.contains(hyp)){
            return noun1;
        }
        return noun2;
    }

    /**
     * Creates files for all nouns and hypernyms.
     */
    public void createFiles(){
        try {
            for (String noun : getAllNouns()){

```

```

        FileWriter fw = new FileWriter(new File (FileNameUtils.getVerbFileName(noun, this)));

        files.put(noun, fw);
    }
} catch (IOException e) {
    e.printStackTrace();
}
}

/**
 * Disposes all create files.
 */
public void disposeFiles(){
    try {
        for (String noun : files.keySet()){
            files.get(noun).close();
        }
        files.clear();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

/**
 * Given the noun, returns the corresponding file
 * ready for writing.
 * @param noun noun
 * @return      FileWriter object
 */
public FileWriter getFile(String noun){
    return files.get(noun);
}

/**
 * Returns a list consisting of the two nouns
 * and all their hypernyms
 * @return      list of nouns and their hypernyms
 */
public List<String> getAllNouns(){
    List<String> list = new ArrayList<String>();
    list.add(noun1);
    list.add(noun2);
    list.addAll(noun1Hypernyms);
    list.addAll(noun2Hypernyms);
    return list;
}

/**
 * Checks whether the given noun is to be treated
 * as a subject or not.
 * @param noun noun
 * @return      true, if the noun is to be treated
 * as a subject
 */
public boolean isSubject(String noun) {
    return (noun.equals(noun2) || noun2Hypernyms.contains(noun));
}
}

```

structures.Verb

```

package structures;

public class Verb implements Comparable{
    private String verb;
    private double frequency;

    public Verb(String verb){
        this.verb = verb;
        this.frequency = 0;
    }
}

```

```

}

public Verb(String verb, float freq) {
    super();
    this.verb = verb;
    this.frequency = freq;
}

@Override
public int compareTo(Object arg0) {
    Verb obj = (Verb) arg0;
    return (this.frequency < obj.frequency) ? 1 : -1;
}

@Override
public boolean equals(Object obj){
    if (obj instanceof Verb){
        Verb other = (Verb)obj;
        if (other.getVerb().equals(this.getVerb())){
            return true;
        }
    }
    return false;
}

public String getVerb() {
    return verb;
}

public void setVerb(String verb) {
    this.verb = verb;
}

public double getFrequency() {
    return frequency;
}

public void setFrequency(double frequency) {
    this.frequency = frequency;
}

public void addFrequency(float newFreq){
    this.frequency += newFreq;
}
}

```

utils.GeneralUtils

```

package utils;

/**
 * Provides general use utility functions.
 */
public class GeneralUtils {

    public static List<String> getIntersectionVerbs(NounCompound nc, Map<String, Map<String, Integer>> map){

        Set<String> verbs1 = new HashSet<String>();
        verbs1.addAll(map.get(nc.getNoun1()).keySet());

        Set<String> verbs2 = new HashSet<String>();
        verbs2.addAll(map.get(nc.getNoun2()).keySet());

        List<String> intersection = new ArrayList<String>();
        for (String verb : verbs1){
            if (verb.startsWith("@") && verb.indexOf(" ") == verb.lastIndexOf(" ")){
                continue;
            }
        }
    }
}

```

```

    }
    if (verb.contains("B") || verb.contains("-")){
        continue;
    }
    if(verbs2.contains(verb)){
        intersection.add(verb);
    }
}
return intersection;
}

public static void eliminateUnrelatedHypernyms(NounCompound nc, Map<String, Map<String, Integer>> map){

    String maxHyp = "";
    float maxCount = 0;
    for (String hyp : nc.getHypernyms(nc.getNoun1())){
        int interCount = getIntersectionCount(map.get(nc.getNoun2()), map.get(hyp));
        int hypCount = map.get(hyp).size();
        float count = ((float)interCount)/(hypCount);
        System.out.println("COUNT FOR " + nc.getNoun2() + " AND " + hyp + ": " + count);
        if (count > maxCount){
            maxHyp = hyp;
            maxCount = count;
        }
    }
    nc.leaveOneHypernym(maxHyp, nc.getNoun1());

    maxHyp = "";
    maxCount = 0;
    for (String hyp : nc.getHypernyms(nc.getNoun2())){
        int interCount = getIntersectionCount(map.get(nc.getNoun1()), map.get(hyp));
        int hypCount = map.get(hyp).size();
        float count = ((float)interCount)/(hypCount);
        System.out.println("COUNT FOR " + nc.getNoun1() + " AND " + hyp + ": " + count);
        if (count > maxCount){
            maxHyp = hyp;
            maxCount = count;
        }
    }
    nc.leaveOneHypernym(maxHyp, nc.getNoun2());
    for (String h : nc.getAllNouns()){
        System.out.println(h);
    }
}

private static int getIntersectionCount(Map<String, Integer> m1, Map<String, Integer> m2){
    int count = 0;
    for (String v1 : m1.keySet()){
        if (m2.containsKey(v1)){
            count++;
        }
    }
    return count;
}
}
}

```

utils.FileNameUtils

```

package utils;

public class FileNameUtils {
    public static final String BNC_ROOT = "C://Corpora//CCG_BNC_v1";
    public static final String SHORT_BNC_ROOT = "C://Corpora//BNC_SHORT";
    public static final String BNC_TEST_ROOT = "C://Corpora//BNC_FINAL_TEST//";
    private static final String NOUN_VERBS_TXT = "_verbs.txt";
    private static final String WEB_RANKED_TXT = "web_ranked.txt";
    private static final String MUT_INFO_RANKED_TXT = "mut_info_ranked.txt";
    private static final String DOUBLE_RANKED_TXT = "double_ranked.txt";
    private static final String NGRAM_RANKED_TXT = "ngram_ranked.txt";
}

```

```

private static final String OBJ_VERSIONS_TXT = "obj_versions.txt";

private static final String GOOGLE_ROOT = "C://Corpora//Google//";
private static final String GOOGLE_1_GRAMS = "1gms//vocab";
private static final String GOOGLE_INDEX_TXT = "2gms_index.txt";

public static String getUniGramFileName(){
    return GOOGLE_ROOT + GOOGLE_1_GRAMS;
}
public static String getGoogleIndexFileName(){
    return GOOGLE_ROOT + GOOGLE_INDEX_TXT;
}

public static String getBigramFileName(String end){
    return GOOGLE_ROOT + end;
}

public static String getVerbFileName(String noun, NounCompound nc){
    return BNC_TEST_ROOT + nc.getNoun1() + "_" + nc.getNoun2() +
        "/" + noun + NOUN_VERBS_TXT;
}

public static String getDir(NounCompound nc){
    return BNC_TEST_ROOT + nc.getNoun1() + "_" + nc.getNoun2();
}

public static String getWebRankedName(NounCompound nc){
    return getDir(nc) + "/" + WEB_RANKED_TXT;
}

public static String getMutInfoRankedName(NounCompound nc){
    return getDir(nc) + "/" + MUT_INFO_RANKED_TXT;
}
public static String getDoubleRankedName(NounCompound nc){
    return getDir(nc) + "/" + DOUBLE_RANKED_TXT;
}

public static String getNGramRankedName(NounCompound nc){
    return getDir(nc) + "/" + NGRAM_RANKED_TXT;
}
public static String getObjVersionsName(NounCompound nc){
    return getDir(nc) + "/" + OBJ_VERSIONS_TXT;
}
}

```

utils.GrammarUtils

```

package utils;

/**
 *Provides utilities for retrieving English
 *grammar information.
 */
public class GrammarUtils {

    /**
     * Returns the plural of the given noun.
     * NOTE: The plural form of irregular nouns is not implemented.
     * Such nouns that were in the test set have been given their
     * plural forms directly by hand.
     * @param noun noun
     * @return plural form of the noun
     */
    public static String getPlural(String noun){
        if (noun.equals("deer") || noun.equals("wealth") || noun.equals("hair") || noun.equals("economics")){
            return noun;
        }
        if (noun.equals("child")){
            return "children";
        }
    }
}

```

```

    }
    if (noun.endsWith("y") && !noun.endsWith("ay")){
        return noun.substring(0, noun.length()-1) + "ies";
    } else if (noun.endsWith("s")){
        return noun + "es";
    } else {
        return noun + "s";
    }
}

public static String getVerbThirdForm(String realVerb){
    String verbThirdForm;
    if (realVerb.endsWith("y")){
        char prev = realVerb.charAt(realVerb.length() - 2);
        if(prev == 'a' || prev == 'e' || prev == 'o' || prev == 'u' || prev == 'i'){
            verbThirdForm = realVerb + "s";
        } else {
            verbThirdForm = realVerb.substring(0, realVerb.length()-1) + "ies";
        }
    } else if (realVerb.endsWith("o") || realVerb.endsWith("ch") || realVerb.endsWith("sh") ||
        realVerb.endsWith("s") || realVerb.endsWith("x") || realVerb.endsWith("z")){
        verbThirdForm = realVerb + "es";
    } else if ("be".equals(realVerb) || "@be".equals(realVerb)){
        verbThirdForm = "is";
    } else if ("have".equals(realVerb)){
        verbThirdForm = "has";
    } else {
        verbThirdForm = realVerb + "s";
    }
    return verbThirdForm;
}

public static String getDeterminerForNoun(String noun){
    if (noun.startsWith("a") || noun.startsWith("e") || noun.startsWith("i") ||
        noun.startsWith("o") || noun.startsWith("u") || noun.startsWith("y")){
        return "an";
    }
    return "a";
}
}
}

```

utils.bnc.BNCFileProcessor

```

package utils.bnc;

public class BNCFileProcessor {

    public static void process(File inputFile, NounCompound nc) {
        if (inputFile.isDirectory()) {
            File[] files = inputFile.listFiles();
            for (File f : files) {
                process(f, nc);
            }
        } else {
            processSingleFile(inputFile, nc);
        }
    }

    private static void processSingleFile(File file, NounCompound nc) {
        System.out.println(file.getAbsolutePath());
        Scanner s = null;
        try {
            s = new Scanner(file);
            s.useDelimiter("\n\n");
            while (s.hasNext()){
                String sentence = s.next();
                BNCSentenceProcessor.processSentence(sentence, nc);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```

    } finally {
        if (s!=null){
            s.close();
        }
    }
}
}
}

```

utils.bnc.BNCSentenceProcessor

```
package utils.bnc;
```

```
public class BNCSentenceProcessor {
```

```
    public static Map<String, List<String>> objVersions = new HashMap<String, List<String>>();
```

```
    private static List<String> beList = new ArrayList<String>();
```

```
    static {
        beList.add("be");
        beList.add("was");
        beList.add("were");
        beList.add("is");
        beList.add("are");
        beList.add("am");
    }

```

```
    public static void processSentence(String sentence, NounCompound nc){
        sentence = sentence.toLowerCase();
        int indexOfC = sentence.indexOf("<c>");
        for (String noun : nc.getAllNouns()){
            processSentenceForNoun(sentence, nc, noun, indexOfC);
        }
    }

```

```
    private static void processSentenceForNoun(String sentence, NounCompound nc, String noun, int indexOfC){
        int index = sentence.indexOf("|" + noun + "|");
        while (index != -1){
            processConcreteOccurrenceOfNoun(sentence, nc, noun, indexOfC, index);
            index = sentence.indexOf("|" + noun + "|", index + 1);
        }
    }

```

```
    private static void processConcreteOccurrenceOfNoun(String sentence, NounCompound nc, String noun, int
indexOfC, int index){
        int j = getIndexOnLeft(' ', index, sentence);
        //this is not the real lemma, moreover, it's vice versa :D
        String lemma = sentence.substring(j+1, index);
        String pos = getPOS(lemma, sentence);
        //we are interested only in noun occurrences of the word
        if (!pos.startsWith("n")){
            return;
        }
        int topIndex = sentence.indexOf(" " + lemma + "_");
        while (topIndex < indexOfC && topIndex != -1){
            processConcreteGR(sentence, nc, noun, lemma, topIndex, pos);
            topIndex = sentence.indexOf(lemma, topIndex + 1);
        }
    }

```

```
    private static void processConcreteGR(String sentence, NounCompound nc, String noun, String lemma, int
topIndex, String pos){
        String nounPortion = getStringOnLeft('(', topIndex, sentence);
        int spaceIndex = nounPortion.indexOf(' ');
        if (spaceIndex == -1){
            return;
        }
        String grTag = nounPortion.substring(0, spaceIndex);
        if (grFits(nc, grTag, noun)){
            String wordWithNumber = nounPortion.substring(spaceIndex + 1, nounPortion.length()).trim();
            if (wordWithNumber.indexOf("_") == -1){
                return;
            }
        }
    }

```

```

    }
    String word = wordWithNumber.substring(0, wordWithNumber.indexOf("_"));
    String posOfWord = getPOS(word, sentence);
    if (posOfWord.startsWith("v")){
        obtainFullVerbGivenVerb(sentence, nc, word, wordWithNumber, noun, lemma, pos);
    } else if (!nc.isSubject(noun) && !":".equals(word) &&
        (posOfWord.equals("in") || posOfWord.equals("to") || posOfWord.equals("rp"))){
        obtainFullVerbGivenPrep(sentence, nc, wordWithNumber, word, lemma, noun, pos);
    }
}

private static void obtainFullVerbGivenPrep(String sentence, NounCompound nc, String wordWithNumber, String
word,
    String lemma, String noun, String pos){
String verb = getVerb(sentence, wordWithNumber);
if (!verb.startsWith("#")){
    String verbLemma = verb.trim() + " " + word;
    try {
        nc.getFile(noun).write(verbLemma + "\n");
    } catch (IOException e) {
        e.printStackTrace();
    }
    if (!nc.isSubject(noun)){
        addObjVersion(sentence, lemma, verbLemma, noun, nc.getNoun(noun), pos);
    }
}
}

private static void obtainFullVerbGivenVerb(String sentence, NounCompound nc, String word, String
wordWithNumber,
    String noun, String lemma, String pos){
String verbNormalForm = getLemma(word, sentence);
String aux = getAuxiliary(sentence, wordWithNumber);
if (!"".equals(aux)){
    verbNormalForm = aux;
}
if (nc.isSubject(noun)){
    String prep = getPrepositions(sentence, wordWithNumber);
    if (!"".equals(prep)){
        verbNormalForm = verbNormalForm + " " + prep;
    }
}
verbNormalForm = verbNormalForm.trim();
try {
    nc.getFile(noun).write(verbNormalForm + "\n");
} catch (IOException e) {
    e.printStackTrace();
}
if (!nc.isSubject(noun)){
    addObjVersion(sentence, lemma, verbNormalForm, noun, nc.getNoun(noun), pos);
}
}

private static void addObjVersion(String sentence, String lemma, String verbNormalForm, String noun, String
rootNoun,
    String pos){
String det = "none";
int indexOfDet = sentence.indexOf("(det " + lemma);
if (indexOfDet >= 0){
    int indexOfSpace = sentence.indexOf(" ", indexOfDet + 6);
    int indexOfDefis = sentence.indexOf("_", indexOfSpace);
    det = sentence.substring(indexOfSpace+1, indexOfDefis);
    if ("a".equals(det) || "an".equals(det)){
        det = GrammarUtils.getDeterminerForNoun(rootNoun);
    }
}
}
List<String> list ;
if (!objVersions.containsKey(verbNormalForm)){
    list = new ArrayList<String>();
    objVersions.put(verbNormalForm, list);
}

```

```

    }else{
        list = objVersions.get(verbNormalForm);
    }
    String word;
    if ("nns".equals(pos)){
        word = GrammarUtils.getPlural(rootNoun);
    } else {
        word = rootNoun;
    }
    String obj = "";
    if (!"none".equals(det)){
        obj = det + " " + word;
    } else {
        obj = word;
    }
    System.out.println(verbNormalForm + " " + obj);
    if (!list.contains(obj)){
        list.add(obj);
    }
}

private static String getVerb(String sentence, String prepWithNumber){
    int shift = 0;
    int parenthesisIndex = 0;
    int indexOfPrep = sentence.indexOf(prepWithNumber + " ");
    if (indexOfPrep != -1) {
        parenthesisIndex = getIndexOnLeft('(', indexOfPrep, sentence);
        if ("iobj".equals(sentence.substring(parenthesisIndex + 1, parenthesisIndex + 5))){
            shift = 6;
        } else if ("ncmod _".equals(sentence.substring(parenthesisIndex + 1, parenthesisIndex + 8))){
            shift = 9;
        }
    }
    if (shift != 0){
        int verbIndex = parenthesisIndex + shift;
        String verbWithNumber = sentence.substring(verbIndex, indexOfPrep-1);
        String verb = verbWithNumber.substring(0, verbWithNumber.indexOf("_"));
        String pos = getPOS(verb, sentence);
        if (pos.equals("in") || pos.equals("to") || pos.equals("rp")){
            return getVerb(sentence, verbWithNumber) + " " + verb;
        } else if (pos.startsWith("v")){
            String aux = getAuxiliary(sentence, verbWithNumber);
            if (!"".equals(aux)){
                return aux;
            } else
                return getLemma(verb, sentence).trim();
        } else {
            return "#";
        }
    }
    return "#";
}

private static String getAuxiliary(String sentence, String verbWithNumber){
    String verb = verbWithNumber.substring(0, verbWithNumber.indexOf("_"));
    if (verb.endsWith("ing")){
        return "";
    }
    int verbIndex = sentence.indexOf(" " + verb + "|") + 1;
    int spaceIndex = sentence.indexOf(" ", verbIndex);
    if (spaceIndex < verbIndex){
        spaceIndex = sentence.length()-1;
    }
    String portion = sentence.substring(verbIndex, spaceIndex);
    if (portion.contains("[pss]") && !beList.contains(verb)){
        return "@be " + verb;
    }
    return "";
}

```

```

private static String getPrepositions (String sentence, String verbWithNumber){
    int indexOfVerbIObj = sentence.indexOf("iobj " + verbWithNumber);
    int shiftNumber = 5;
    if (indexOfVerbIObj == -1){
        indexOfVerbIObj = sentence.indexOf("ncmod _ " + verbWithNumber);
        shiftNumber = 8;
    }
    String prep = "";
    String prepWithNumber = "";
    if (indexOfVerbIObj != -1){
        prepWithNumber = sentence.substring(indexOfVerbIObj + verbWithNumber.length() + shiftNumber,
            sentence.indexOf(')', indexOfVerbIObj + verbWithNumber.length() +
shiftNumber));
        prep = prepWithNumber.substring(1, prepWithNumber.indexOf("_"));
    }
    String posOfPrep = getPOS(prepare, sentence);
    if (!"in".equals(posOfPrep) && !"to".equals(posOfPrep) && !"rp".equals(posOfPrep)){
        prep = "";
    }
    return prep;
}

private static boolean grFits(NounCompound nc, String grTag, String noun){
    return (nc.isSubject(noun) && "ncsubj".equals(grTag))
        || (!nc.isSubject(noun) && ("dobj".equals(grTag) || "obj2".equals(grTag)));
}

private static String getLemma(String word, String sentence){
    int wordIndexInSentence = sentence.indexOf(" " + word + "|");
    int lemmaIndex = sentence.indexOf("|", wordIndexInSentence) + 1;
    int endOfLemmaIndex = sentence.indexOf("|", lemmaIndex);
    return sentence.substring(lemmaIndex, endOfLemmaIndex);
}

private static String getPOS(String word, String sentence){
    int wordIndexInSentence = sentence.indexOf(" " + word + "|");
    int indexOfPOS = sentence.indexOf("|", wordIndexInSentence + word.length() + 2) + 1;
    int endIndexofPOS = sentence.indexOf("|", indexOfPOS);
    return sentence.substring(indexOfPOS, endIndexofPOS);
}

private static String getStringOnLeft(char ch, int fromIndex, String sentence){
    StringBuffer strBuf = new StringBuffer();
    char c;
    for (int j = fromIndex-1; j > 0; j--){
        c = sentence.charAt(j);
        if (c == ch){
            break;
        }
        strBuf.append(c);
    }
    return strBuf.reverse().toString();
}

private static int getIndexOnLeft(char ch, int fromIndex, String sentence){
    int j;
    for (j = fromIndex; j > 0; j--){
        if (sentence.charAt(j) == ch){
            break;
        }
    }
    return j;
}

public static List<String> getObjVersions(String verb){
    return objVersions.get(verb);
}
}

```

utils.io.FileReader

```

package utils.io;

public class FileReader implements Iterable<String> {
    private BufferedReader _reader;

    public FileReader(String filePath) throws Exception {
        _reader = new BufferedReader(new java.io.FileReader(filePath));
    }

    public void close() {
        try {
            _reader.close();
        } catch (Exception ex) {
        }
    }

    public Iterator<String> iterator() {
        return new FileIterator();
    }

    private class FileIterator implements Iterator<String> {
        private String _currentLine;

        public boolean hasNext() {
            try {
                _currentLine = _reader.readLine();
            } catch (Exception ex) {
                _currentLine = null;
                ex.printStackTrace();
            }

            return _currentLine != null;
        }

        public String next() {
            return _currentLine;
        }

        public void remove() {
        }
    }
}

```

utils.io.IOUtils

```

package utils.io;

public class IOUtils {

    public static Map<String, Map<String, Integer>> readVerbFile(NounCompound nc){
        Map<String, Map<String, Integer>> map = new HashMap<String, Map<String, Integer>>();

        for (String noun : nc.getAllNouns()){
            Map<String, Integer> nounMap = new HashMap<String, Integer>();
            String n= nc.getNoun(noun);
            if (n.equals(nc.getNoun1())){
                if (map.get(nc.getNoun1()) != null){
                    nounMap = map.get(nc.getNoun1());
                }
                addVerbCount(noun, nc, nounMap);
                map.put(nc.getNoun1(), nounMap);
            } else {
                if (map.get(nc.getNoun2()) != null){
                    nounMap = map.get(nc.getNoun2());
                }
                addVerbCount(noun, nc, nounMap);
                map.put(nc.getNoun2(), nounMap);
            }
        }
        return map;
    }
}

```

```

}

private static void addVerbCount(String noun, NounCompound nc, Map<String, Integer> nounMap){
    FileReader fr = null;
    try {
        fr = new FileReader(FileNameUtils.getVerbFileName(noun, nc));
        for (String verb : fr){
            int count = 0;
            if (nounMap.containsKey(verb)){
                count = nounMap.get(verb);
            }
            nounMap.put(verb, count+1);
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (fr != null){
            fr.close();
        }
    }
}

public static Map<String, Map<String, Integer>> readVerbFileSeparately(NounCompound nc){
    Map<String, Map<String, Integer>> map = new HashMap<String, Map<String, Integer>>();
    for (String noun : nc.getAllNouns()){
        Map<String, Integer> nounMap = map.get(noun);
        if (!map.containsKey(noun)){
            nounMap = new HashMap<String, Integer>();
        }
        FileReader fr = null;
        try {
            fr = new FileReader(FileNameUtils.getVerbFileName(noun, nc));
            for (String verb : fr){
                int count = 0;
                if (nounMap.containsKey(verb)){
                    count = nounMap.get(verb);
                }
                nounMap.put(verb, count+1);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (fr != null){
                fr.close();
            }
        }
        map.put(noun, nounMap);
    }
    return map;
}

public static void writeListToFile(String inputFileName, List<Verb> list){

    try {
        FileWriter fw = new FileWriter(new File(inputFileName));
        for (Verb v: list){
            fw.write(v.getVerb() + " " + v.getFrequency() + "\n");
        }
        fw.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static long getVerbOccurrenceCount(String inputFileName, String word){
    File inputFile = new File(inputFileName);
    return getVerbOccurrenceCount(inputFile, word);
}

```

```

private static long getVerbOccurenceCount(File inputFile, String word) {
    if (inputFile.isDirectory()) {
        long n = 0;
        File[] files = inputFile.listFiles();
        for (File f : files) {
            n += getVerbOccurenceCount(f, word);
        }
        return n;
    } else {
        return getVerbCountFromFile(inputFile.getAbsolutePath(), word);
    }
}

private static long getVerbCountFromFile(String inputFileNames, String word) {
    FileReader b = null;;
    long count = 0;
    try {
        b = new FileReader(inputFileNames);
        for (String line : b) {
            if (line.contains(" " + word + " ")){
                count ++;
            }
        }
    } catch (Exception e) {
        System.out.println(inputFileNames);
        e.printStackTrace();
    } finally {
        if (b != null){
            b.close();
        }
    }
    return count;
}

public static void writeToFile(String fileName, List<String> list){
    try {
        FileWriter fw = new FileWriter(new File(fileName));
        for (String str : list){
            fw.write(str + "\n");
        }
        fw.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void writeObjVersionsToFile(String fileName, Map<String, List<String>> map){
    try {
        FileWriter fw = new FileWriter(new File(fileName));
        for (String verb : map.keySet()){
            fw.write(verb + "\n");
            for (String version : map.get(verb)){
                fw.write(version + "\n");
            }
            fw.write("\n");
        }
        fw.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static Map<String, List<String>> readObjVersionsFromFile(String fileName){
    System.out.println("reading file");
    Map<String, List<String>> map = new HashMap<String, List<String>>();
    FileReader fr = null;
    try {
        fr = new FileReader(fileName);
        List<String> versions = new ArrayList<String>();
        for (String line : fr){
            if (!"".equals(line)){

```

```

        versions.add(line);
    } else {
        List<String> objVersions = new ArrayList<String>();
        for (int i = 1; i < versions.size(); i++){
            objVersions.add(versions.get(i));
        }
        System.out.println(versions.get(0) + " " + objVersions.size());
        map.put(versions.get(0), objVersions);
        versions.clear();
    }
} catch (Exception e) {
    e.printStackTrace();
} finally {
    if (fr != null){
        fr.close();
    }
}
return map;
}
}

```

```

public static List<Verb> readRankedFile(String fileName){
    List<Verb> list = new ArrayList<Verb>();
    FileReader fr = null;
    try {
        fr = new FileReader(fileName);
        for (String line : fr){
            int lastSpaceIndex = line.lastIndexOf(" ");
            Verb verb = new Verb(line.substring(0, lastSpaceIndex));
            verb.setFrequency(Float.parseFloat(line.substring(lastSpaceIndex+1)));
            list.add(verb);
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (fr != null){
            fr.close();
        }
    }
    return list;
}
}
}

```

utils.wordnet.WordnetUtils

```

package utils.wordnet;

public class WordnetUtils {

    public static void main(String[] args) throws JWNLEException{
        WordnetUtils w = new WordnetUtils();
        w.getHypernyms("oil");
    }

    private RiWordnet wordnet;

    public WordnetUtils() {
        wordnet = new RiWordnet(null);
    }

    public void dispose(){
        wordnet.dispose();
    }

    public List<String> getHypernyms(String noun) {
        List<String> hyps = new ArrayList<String>();
        int[] senseIds = wordnet.getSenseIds(noun, "n");
        String[] hypernyms;
        try {
            hypernyms = wordnet.getHypernymTree(senseIds[0]);
        }
    }
}

```

```

        if (hypernyms != null){
            String hyp = hypernyms[0];
            if (hyp.contains(":")){
                hyp = hyp.substring(0, hyp.indexOf(":"));
            }
            hyps.add(hyp);
        }
        if (senseIds.length > 1){
            hypernyms = wordnet.getHypernymTree(senseIds[1]);
            if (hypernyms != null){
                String hyp = hypernyms[0];
                if (hyp.contains(":")){
                    hyp = hyp.substring(0, hyp.indexOf(":"));
                }
                hyps.add(hyp);
            }
        }
    } catch (JWNLEException e) {
        e.printStackTrace();
    }
    return hyps;
}
}

```

utils.googleNGrams.GoogleNGramUtils

```

package utils.googleNGrams;

public class GoogleNGramUtils {

    public static long getUnigramCount(String uniGram){
        return getNGramCount(uniGram, FileNameUtils.getUniGramFileName());
    }

    public static long getBigramCount(String biGram){
        long overallCount = 0;
        List<String> biGramFileNames = getNGramFileNames(biGram);
        for (String fileName : biGramFileNames){
            overallCount += getNGramCount(biGram, FileNameUtils.getBigramFileName(fileName));
        }
        return overallCount;
    }

    private static long getNGramCount(String nGram, String fileName){
        long count = 0;
        FileReader fr = null;
        try {
            fr = new FileReader(fileName);
            for (String line : fr){
                line = line.toLowerCase();
                if (line.startsWith(nGram+"\t")){
                    count += Long.parseLong(line.substring(line.lastIndexOf("\t")+1));
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (fr!=null){
                fr.close();
            }
        }
        return count;
    }

    private static List<String> getNGramFileNames(String ngram){
        String firstLetters = ngram.substring(0, 2);
        List<String> nGramFileNames = new ArrayList<String>();
        FileReader fr = null;
        try {
            fr = new FileReader(FileNameUtils.getGoogleIndexFileName());
            for (String line : fr){

```

```

        if (line.toLowerCase().startsWith(firstLetters)){
            line = line.substring(line.indexOf("\t")+1);
            String[] names = line.split("\t");
            for (String name : names){
                if (!nGramFileNames.contains(name)){
                    System.out.println(name);
                    nGramFileNames.add(name);
                }
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (fr!=null){
            fr.close();
        }
    }
    return nGramFileNames;
}
}
}

```

utils.validators.MutualInfoValidator

```

package validators;

/**
 * Deals with mutual information based validation and ranking
 */
public class MutualInfoValidator {

    public static List<Verb> getMutualInfoRankedList(NounCompound nc, Map<String, Map<String, Integer>> map,
List<Verb> webRankedList){
        List<Verb> topList = new ArrayList<Verb>();
        int i = 0;
        for (Verb v : webRankedList){
            if (i > 9){
                break;
            }
            if (v.getFrequency() > 0){
                topList.add(v);
            } else {
                break;
            }
            i++;
        }

        List<Verb> rankedList = MutualInfoValidator.getMutRankedList(topList, nc, map);
        return rankedList;
    }

    private static float calculateMutualInfo(String verb, int coocurrenceNumber){
        if (verb.startsWith("@")){
            verb = verb.substring(verb.indexOf(" ") + 1);
        }
        long verbCount = IOUtils.getVerbOccurrenceCount(FileNameUtils.SHORT_BNC_ROOT, verb);
        System.out.println("Verb count " + verb + " " + verbCount);
        return ((float)coocurrenceNumber)/verbCount;
    }

    private static float calculateMutualInfo(String verb, NounCompound nc,
        Map<String, Map<String, Integer>> map){
        int coOccur1 = map.get(nc.getNoun1()).get(verb);
        System.out.println("Cooccur 1 " + verb + " " + coOccur1);
        int coOccur2 = map.get(nc.getNoun2()).get(verb);
        System.out.println("Cooccur 2 " + verb + " " + coOccur2);

        float mutInfo1 = calculateMutualInfo(verb, coOccur1);
        float mutInfo2 = calculateMutualInfo(verb, coOccur2);
        System.out.println(mutInfo1 + " " + mutInfo2);
    }
}

```

```

        return mutInfo1 * mutInfo2;
    }

    private static List<Verb> getMutRankedList(List<Verb> list, NounCompound nc,
        Map<String, Map<String, Integer>> map){
        for (Verb v : list){
            v.setFrequency(calculateMutualInfo(v.getVerb(), nc, map));
        }
        Collections.sort(list);
        return list;
    }
}

```

utils.validators.NGramValidator

```
package validators;
```

```

/**
 * Deals with N-gram model based validation and ranking
 */
public class NGramValidator {

    public static List<Verb> getNGramRankedList(NounCompound nc, List<Verb> list){
        Map<String, Double> bigrams = new HashMap<String, Double>();
        Map<String, Double> unigrams = new HashMap<String, Double>();
        for (Verb v : list){
            String realVerb = v.getVerb();
            String prep = "";
            if (realVerb.contains(" ")){
                prep = realVerb.substring(realVerb.indexOf(" ") + 1, realVerb.length()) + " ";
                realVerb = realVerb.substring(0, realVerb.indexOf(" "));
            }
            String thirdForm = "";
            if (!realVerb.startsWith("@be")){
                thirdForm = GrammarUtils.getVerbThirdForm(realVerb) + " ";
            }
            String nGram = nc.getNoun2() + " " + thirdForm + prep + nc.getNoun1();
            System.out.println("NGRAM IS " + nGram);
            v.setFrequency(getNGramProbability(nGram, bigrams, unigrams));
            System.out.println(v.getVerb() + " " + v.getFrequency());
        }
        Collections.sort(list);
        return list;
    }

    private static double getNGramProbability(String ngram, Map<String, Double> bigrams, Map<String, Double>
unigrams){
        double prob = 1;
        String[] words = ngram.split(" ");
        for (int i = 1; i < words.length; i++){
            String bigram = words[i-1] + " " + words[i];
            String unigram = words[i-1];
            double bigramCount, unigramCount;

            if (bigrams.containsKey(bigram)){
                bigramCount = bigrams.get(bigram);
            } else {
                bigramCount = GoogleNGramUtils.getBigramCount(bigram);
                bigrams.put(bigram, bigramCount);
            }

            if (unigrams.containsKey(unigram)){
                unigramCount = unigrams.get(unigram);
            } else {
                unigramCount = GoogleNGramUtils.getUnigramCount(unigram);
                unigrams.put(unigram, unigramCount);
            }

            double newProb = bigramCount / unigramCount;

```

```

        prob = prob * newProb;
    }
    return prob;
}
}

```

utils.validators.YahooWebValidator

```
package validators;
```

```

/**
 * Deals with web based validation and ranking
 */
public class YahooWebValidator {

    public static List<Verb> getWebRankedVerbs (NounCompound nc, List<String> unorderedList){
        List<Verb> verbs = new ArrayList<Verb>();
        String plural = GrammarUtils.getPlural(nc.getNoun2());
        boolean searchPlural = !nc.getNoun2().equals(plural);
        for (String verbStr : unorderedList){
            Verb verb = new Verb(verbStr);
            List<String> objVersions = BNCSentenceProcessor.getObjVersions(verbStr);
            int rank = 0;
            if (objVersions != null){
                for (String objVersion: objVersions){
                    objVersion = objVersion.replaceAll(" ", "+");
                    String query = YahooWebValidator.prepareQuery(nc.getNoun1(), nc.getNoun2(),
verbStr, objVersion, false);
                    System.out.println(query);
                    rank += YahooWebValidator.search(query);
                    if (searchPlural){
                        String queryPlural = YahooWebValidator.prepareQuery(nc.getNoun1(),
plural, verbStr, objVersion, true);
                        rank += YahooWebValidator.search(queryPlural);
                    }
                }
                verb.setFrequency(rank);
                verbs.add(verb);
            } else {
                System.out.println("OBJ VERSIONS NOT FOUND FOR: " + verbStr);
            }
        }
        Collections.sort(verbs);
        return verbs;
    }

    public static List<Verb> getDoubleRankedList(NounCompound nc, List<Verb> mutInfoRankedList){
        List<String> stringList = new ArrayList<String>();
        int limit = mutInfoRankedList.size()/2 + 1;
        if (mutInfoRankedList.size() == 2){
            limit = 1;
        }
        for (int i = 0; i < limit; i++){
            stringList.add(mutInfoRankedList.get(i).getVerb());
        }
        List<Verb> doubleRankedList = getWebRankedVerbs(nc, stringList);
        return doubleRankedList;
    }

    private static int search(String query) {
        String request =
"http://search.yahooapis.com/WebSearchService/V1/webSearch?appid=zbckb4HrsM5bNQfxIKX4g169gZa5C.LfxeYDz0QGYqaeB
3iWScgwFlFKFuE-&query=\"" + query + "\"";
        String number = "0";
        try {
            URL url = new URL(request);
            InputStream in = url.openStream();
            byte[] buf = new byte[1024];
            int len;

```

```

        StringBuffer strBuf = new StringBuffer();
        while ((len = in.read(buf)) > 0) {
            for (int i = 0; i < len; i++) {
                strBuf.append((char)buf[i]);
            }
        }
        in.close();
        String allStr = strBuf.toString();
        int index = allStr.indexOf("totalResultsAvailable");
        String resStr = allStr.substring(index, index + 40);
        int firstIndex = resStr.indexOf("\"") + 1;
        int lastIndex = resStr.indexOf("\"", firstIndex);
        number = resStr.substring(firstIndex, lastIndex);
    } catch (Exception e) {
        e.printStackTrace();
        System.out.println("Web services request failed");
    }
    return Integer.parseInt(number);
}

private static String prepareQuery(String noun1, String noun2, String verb, String objVersion, boolean
plural){
    String realVerb = verb.trim();
    String prep = "";
    if (realVerb.contains(" ")){
        prep = realVerb.substring(realVerb.indexOf(" ") + 1, realVerb.length());
        realVerb = realVerb.substring(0, realVerb.indexOf(" "));
    }
    String verbThirdForm = realVerb;
    if (!plural){
        verbThirdForm = GrammarUtils.getVerbThirdForm(realVerb);
    }
    String queryPrep = "";
    if (!"".equals(prep)){
        StringTokenizer strTok = new StringTokenizer(prep, " ");
        while(strTok.hasMoreTokens()){
            queryPrep += strTok.nextToken()+" ";
        }
    }
    String queryBeginning;
    if ("@be".equals(realVerb)){
        queryBeginning = noun2 + "+" + queryPrep;
    } else {
        queryBeginning = noun2 + "+" + verbThirdForm + "+" + queryPrep;
    }
    return queryBeginning + objVersion;
}
}

```