Prepositional Phrase Attachment Disambiguation using WordNet

by

Claus W. Spitzer

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2006

Author's Declaration for Electronic Submission of a Thesis

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In this thesis we use a knowledge-based approach to disambiguating prepositional phrase attachments in English sentences. This method was first introduced by S. M. Harabagiu.

The Penn Treebank corpus is used as the training text. We extract 4-tuples of the form [VP, NP_1, Prep, NP_2] and sort them into classes according to the semantic relationships between parts of each tuple. These relationships are extracted from WordNet. Classes are sorted into different tiers based on the strictness of their semantic relationship. Disambiguation of prepositional phrase attachments can be cast as a constraint satisfaction problem, where the tiers of extracted classes act as the constraints. Satisfaction is achieved when the strictest possible tier unanimously indicates one kind of attachment. The most challenging kind of problems for disambiguation of prepositional phrases are ones where the prepositional phrase may attach to either the closest verb or noun.

We first demonstrate that the best approach to extracting tuples from parsed texts is a top-down postorder traversal algorithm. Following that, the various challenges in forming the prepositional classes utilizing WordNet semantic relations are described. We then discuss the actions that need to be taken towards applying the prepositional classes to the disambiguation task. A novel application of this method is also discussed, by which the tuples to be disambiguated are also expanded via WordNet, thus introducing a client-side application of the algorithms utilized to build prepositional classes. Finally, we present results of different variants of our disambiguating algorithm, contrasting the precision and recall of various combinations of constraints, and comparing our algorithm to a baseline method that falls back to attaching a prepositional phrase to the closest left phrase. Our conclusion is that our algorithm provides improved performance compared to the baseline and is therefore a useful new method of performing knowledge-based disambiguation of prepositional phrase attachments.

Acknowledgements

Many thanks to my supervisor, Anne Banks Pidduck, and my co-supervisor, Robin Cohen, for their support and encouragement. I would also like to extend my gratitude to Gordon Cormack, Frank Tompa, and Chrysanne DiMarco for providing guidance, insight, and access to tools and literature that helped me complete this thesis.

Claus W. Spitzer, Waterloo, November 23, 2005

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Significance of Prepositional Phrase Attachments

Prepositional phrases, such as "*with chopsticks*" and "*in the park*" do not occur by themselves in written English. They require another phrase to be attached to, for example

Joe eats with chopsticks.

The above sentence is unambiguous. It is clear that "*with chopsticks*" is attached to the verb phrase, "*eats*". However, not all prepositional phrase attachments are unambiguous. In the sentence

Carl saw the man in the park

it is not possible to determine immediately whether Carl was in the park and saw a man, or whether Carl saw a man that was in the park.

Systems like grammar checkers, parsers, and machine translators (among others) each implement or utilize some form of disambiguation algorithm. One task done by such algorithms is the disambiguation of prepositional phrase attachments. While such an

undertaking is mostly trivial to humans, an efficient algorithmic solution to it for computers remains elusive. However, disambiguation of prepositional phrase attachments is necessary for the understanding of text.

Efficient disambiguation of prepositional phrase attachments is a very difficult problem in natural language processing [Brill and Resnik, 1994] [Harabagiu, 2000]. Difficulties arising from this task are rooted in the fact that the problem encompasses not only lexical ambiguities but also semantic and thematic ambiguities (see section 1.3 for details). Prepositional phrase attachments are important to text understanding and even information extraction because they affect the meaning of sentences in a critical manner. In addition to the previous example of ambiguous meanings of prepositional attachments, one can consider how the mere presence of a prepositional phrase attachment can alter the meaning of a sentence. For example, in the sentence

> [*Nurses can reduce the likelihood of being named in a lawsuit by maintaining clinical competency*]. [Schulmeister, 1999]

the relevance of the subject is altered if the prepositional phrase attachment is left out:

> [*Nurses can reduce the likelihood of being named in a lawsuit.*][1]

The original sentence indicates that the risk of lawsuits can be reduced if nurses perform a certain action. The second one implies that just having a nurse in the area will reduce the risk of lawsuit. The thematic content of the sentence is altered by omitting the prepositional phrase attachment.

[1]This abbreviated sentence was an actual result produced by the *Medstract* system ([Pustejovsky et al., 2002b], [Pustejovsky et al., 2002a]).

1.2 Focus of this Paper

During the course of this paper an algorithm is described that performs disambiguation of ambiguous prepositional phrase attachments. Once trained, the algorithm receives as input tuples extracted from ambiguous attachments, and returns a value containing the suggested attachment type.

Throughout this paper we will consider phrases to be collections of words within a sentence that function like a single syntactic unit. The *phrase head* is the word that links each phrase to the rest of the sentence. Phrase heads are usually the first word of a phrase.

The type of sentences of interest for this paper are those whose syntactic structure contains a particular sequence of parts of speech. This sequence is indicated by the 4-tuple containing a verb phrase (VP), a noun phrase (NP), a preposition, and another noun phrase:

[VP, NP_1, Prep, NP_2].

These sentences are of interest in natural language processing because they only obey the English rule of thumb an average of 60 percent of the time. The aforementioned rule states that whenever there is a choice between two phrases to which a prepositional phrase can be attached, the rightmost should be chosen. This rule is hence known as right attachment [Kimball, 1973]. The structure of the above figure can be simplified by reducing the verb phrases and noun phrases to their phrase heads, those being a verb for the verb phrase and a noun for the noun phrase respectively.

[$Verb$, $Noun_1$, Prep, $Noun_2$].

We can justify this simplification because phrase attachments are relations between the involved phrases. Since such relations are characterized by links and the links themselves

1. Sally VP(watches NP(the bees) PP($Prep$(with) NP(her binoculars))).
2. The doctor VP(has approved) NP(the new treatment PP($Prep$(for) NP(his ailment))).

Figure 1.1: Examples of an adverbial (1) and an adjectival (2) prepositional phrase attachment

are rooted in the phrase heads, we can infer that these phrase heads are sufficient to provide the lexical information necessary to disambiguate prepositional phrase attachments.

Prepositional phrases that are attached to the verb phrase have the prepositional phrase defining attributes pertaining the theme whose action, state, or event are lexicalized by the verb phrase head. This type of attachment is called adverbial or left in this paper attachment. A prepositional phrase modifying a noun phrase, on the other hand, affects the nominal of such noun phrase by defining its attributes. When the noun phrase head is a nominalization of a verb (that is, a noun morphologically derived from that verb, such as *rejection*) then the prepositional attachment defines the thematic attributes of the phrase in a manner similar to adverbial attachments. Prepositional phrases attached to the noun phrase are called adjectival or right attachments throughout this paper. While "adjectival attachment" does not completely reflect the nature of attachments where the noun phrase head is a nominalization, we believe that it is an acceptable generalization since we do not concern ourselves with particular special cases regarding the attributes of the prepositional phrases. In other words, the labels differentiate whether the prepositional phrase is attached to the noun phrase or the verb phrase, and nothing more. Figure 1.1 provides examples of sentences containing adverbial and adjectival attachments respectively.

We have chosen to focus the input to our algorithm on the aforementioned 4-tuples and ignore other parts of the sentence (such as subject-object relations) because those tuples are formally considered to be the critical components needed for prepositional

phrase attachment disambiguation [Brill and Resnik, 1994].

Training and execution of our algorithm is unsupervised, meaning that the system does not wait for human input for all or part of its steps. Supervised algorithms, on the other hand, require human actions at certain parts of their training in order to interpret certain input.

1.3 Prepositional Phrase Attachment Ambiguities

Formal English rules indicate that whenever possible, prepositional phrases should be placed so that they attach to the nearest part of speech (right association). This principle does not perform well when applied to common English sentences, however, thus reducing the effectiveness of an always-adjectival attachment algorithm to about 60 percent [Kimball, 1973]. Identifying which preceding phrase a prepositional prhase is attached to is thus important for the understanding of a sentence.

Four types of knowledge may be employed in order to disambiguate prepositional phrase attachments. Some of these types of knowledge are more frequent in texts than others. We have listed them in order of commonality:

- *Lexical*: Lexical information is provided by the sentence structure and contains not only part of speech information but also sentence syntax. This kind of information may not always be enough to disambiguate prepositional phrase attachments, but it is a valuable asset in narrowing down the scope of the problem by delineating the valid attachment forms.

- *Semantic*: Determining the sense of the words involved in a prepositional attachment can help us in fixating the type of prepositional phrase attachment we have in a manner similar to the lexical information.

- *Thematic*: Some prepositional phrase attachments, such as "John talked to the man in the room" are correct in both adverbial and adjectival form - John may have been in the room, talking to the man, or he could have been talking to the man that was in the room. Determining whether this prepositional phrase attachment is of the adverbial or adjectival kind in this case is no longer a matter of specifying which combination is valid. Contextual information would be needed to further determine which type of attachment should be chosen.

- *World knowledge*: Some phrases, particularly composite proper nouns such as "Bank of Commerce", contain prepositional phrases whose attachment is static because of the attributed sense of the phrase, which may not always be derived from the individual senses of its components. Knowing the domain for which the text was written (business journals, for example) may provide important information on which kinds of phrases have a particular sense and attachment type differing from the one that would be determined by the independent senses of their components.

It is rare to find sources for which all four of these types of knowledge are readily available. *Lexical* information is the easiest to obtain, since it can be extracted with considerable success rates with the help of part of speech taggers as described in section 2.1. Utilization of *semantic* information embedded in text requires a considerable effort. A preexisting knowledge of the lexical structure of the text is necessary, as well as knowledge of the grammatical rules that apply to the parsed source. *World knowledge*, on the other hand, is a considerably more difficult matter. It requires an understanding of the area that the text stems from, which is challenging when performing unsupervised disambiguation. This kind of knowledge can rarely be directly extracted from the parsed text, requiring an external source to be supplied. When available, databases like gazetteers (geographical dictionaries) provide a wealth of information useful to focus on the appropriate phrases.

Thematic knowledge is usually the most difficult kind to acquire, since it depends on partial understanding of the context in which a sentence is parsed. For that reason it is normally left out of unsupervised prepositional phrase disambiguation algorithms.

Semantic information is the one which we shall focus most of our attention for this paper, because it is the most useful kind in order to perform proper disambiguation of prepositional phrase attachments. In order to facilitate this task we will be using a pre-tagged source, the Penn Treebank [Marcus et al., 1993], to acquire lexical information. The system presented emulates the approach to prepositional phrase disambiguation with the WordNet ontology [Miller, 1995] proposed in [Harabagiu, 1996] and further extends it by applying their techniques in a novel way. This is achieved by applying parts of the routines that were used to train the algorithm to the disambiguation phase, greatly improving the recall rate of the system.

Chapter 2

Natural Language Processing

2.1 Part of Speech Tagging

Part of speech tagging is the area of natural language processing that focuses on labeling of the words in a sentence. The labeling is usually done as an intermediate step towards other applications of natural language processing, such as information gathering, question answering, and shallow parsing.

The labels utilized in part of speech tagging are usually those identified in the Brown corpus [Francis and Kucera, 1979] or a variant thereof. Table 2.1 shows some commonly used tags. A typical tagging of a sentence may look like this:

> [*The-**AT** cat-**NN** chased-**VBD** the-**AT** fly-**NN** towards-**IN** the-**AT** bank-**NN**.*]

However, some measure of syntactic disambiguation is required in order to prevent incoherent results such as

> [*The-**AT** cat-**NN** chased-**VBD** the-**AT** fly-**VB** towards-**IN** the-**AT** bank-**VB**.*]

Tag	Part of Speech
AT	article
IN	preposition
JJ	adjective
VB	verb, base form
VBD	verb, past tense
NN	singular or mass noun

Table 2.1: Examples of commonly used part of speech tags

The extent of such disambiguation only goes as far as being able to produce coherent chains of word-tag pairs. Resolution of prepositional phrase attachment ambiguity for example is not a task required of part of speech taggers. It is thus said that the task of these taggers is of limited scope. However, the usefulness of taggers is not to be underestimated because of this limited scope - taggers are crucial to natural language processing applications.

Typically we run into two kinds of part of speech taggers, stochastic and rule-based, which are discussed below.

2.1.1 Stochastic Taggers

One of the easiest taggers that is possible to create is a "dumb" probabilistic tagger. In such a system, the tagger is trained on large quantities of pre-tagged text. The frequencies of the tags for each word in the training text are collected These frequencies are used for tagging new text by assigning each word the tag that is most likely for it. [Charniak et al., 1993] demonstrated that such a tagger performs at an astonishingly high level of 90% correctness. The efficiency of such a tagger indicates that statistical methods perform well for part of speech tagging. A common stochastical approach is the Markov Model, by which sentences are treated as Markov chains where the states are represented by the words' tags. The Markov Model is a left-to-right finite state model, which means

that once a word has been tagged, its tag is not altered. To be more precise, Markov chains feature two properties:

- Time invariance and
- Limited horizon.

The former is the already described property of word tags being stationary. The latter attribute is based on the notion that, knowing the present, the state of the past is irrelevant to predict the future. In a Markov Model tagger that means that, knowing the tag of the current word, it is not necessary to know the tags of the previous words. Because this model works with two words at a time (the word being tagged and its predecessor), it is said to be a *bigram* tagger. Some versions of Markov Model taggers are trigram taggers, which means that they use the tag information of the *two* preceeding words to select the tag of the current word. Church [1988], one of the most referenced publications on tagging and one of the most influential forces to drive researchers to the problem of part of speech tagging, describes a trigram tagger.

Markov Model taggers can be grouped into two categories - Visible Markov Model taggers (commonly referred to as just Markov Model taggers) and Hidden Markov Model (HMM) taggers. Both of these types perform the same tagging procedure, and differ only in how they are trained. Visible Markov Model taggers are trained on tagged texts, and the models are called visible because we can observe the state of the model as it is trained. HMM taggers, on the other hand, are handy in situations where large amounts of tagged corpora are unavailable, such as domain-specific texts and foreign language texts. They are trained with the aid of dictionaries. One class of HMM taggers assigns zero probabilities to impossible word-tag pairs, such as speaker-AT. Another type of HMM tagger groups words into classes depending on the tags that they may be paired with. For example, all words that may contain the tags NN and JJ (and only those tags) are

put into the $NN - JJ$ group. The advantage of such a method is that it is not necessary to fine-tune each word separately. It is important to note, however, that this approach is disadvantageous when enough training material is available to perform per-word analysis.

We have already mentioned one variation of Markov Model taggers, trigram taggers. Another important variation to consider is the tagging of unknown words. The efficiency of the dumb tagger in [Charniak et al., 1993] is so high because it deals with blind tagging of known words. However, not only do unknown words have no statistical data to help the tagger identify which tag it is more likely to have, but they also lack any concrete information as to which word class they belong to. A naive solution is to set the probability of such a word belonging to any word class to the same value. A small variant of this approach is to only consider open word classes, such as nouns, verbs, and adverbs. Neither of these solutions, however, prove to be sufficiently accurate, averaging at about 40% error rate. A better solution is to consider the morphological and lexical information that may be extracted from a word, such as "words ending in -ed tend to be past participles". These features are usually considered to be independent of each other in literature dealing with this problem, but that assumption is often an improper one. Interactions between word features, such as the relation between "unknown word" and "capitalized word" can be as important as the features themselves [Manning and Schütze, 1999].

2.1.2 Rule-Based Taggers

Early rule-based part of speech taggers were founded on simple inferences gained from syntagmatic information extracted from sentences. Such information usually consists of tag chains and their features. For example, the chain $AT-JJ-NN$ is common in English sentences, whereas the chain $AT-JJ-VBP$ is forbidden. Such deterministic rule-based taggers do not perform at high precision, producing a success rate of only 77% at best.

With the advent of stochastical part of speech taggers, rule-based taggers were mostly ignored. Brill [1992] introduced a rule-based part of speech tagger whose accuracy can be compared to the top stochastic models, and whose design sports several advantages over statistical models. Such advantages include robustness, automatic acquisition of rules, reduction of required stored information, a small set of meaningful rules, easy improvement, and better portability between tag sets or corpora genres.

The base model of this rule-based tagger is founded on the dumb statistical model. On its first iteration of training, the tagger labels all words in a training corpus with their most likely label, independently of their context. Unlike dumb statistical models, the Brill tagger does not stop here. After tagging all words, triples of the form $label_a, label_b, num$ are extracted from the tagged text, where $label_a$ is the label that the tagger assigned to the word, $label_b$ the label that the word should have had instead, and num the number of occurrences of this mistake. The tagger then automatically creates a list of rules using a small set of instructions. These rules are created using contextual information inferred from the incorrectly tagged triples to correct their instances. They are then individually tested against the training corpus, and the rule with the best success rate (where success rate = number of improper tags fixed - number of new words improperly tagged) is selected as a permanent rule. The procedure then continues to extract triples and formulate rules until an arbitrarily selected accuracy has been achieved. This approach is similar to boosting, which we describe in section 3.5.

2.2 WordNet

WordNet [Miller, 1995] is an online lexical reference system - in other words, an ontology - created at Princeton University, in which words are classified into synonym sets (synsets in short) that tie words to lexical concepts. Different synsets are constructed for nouns,

verbs, adjectives, and adverbs. Each synset can contain different words, and each word can occur in different synsets, depending on the sense of the word and the part of speech category (noun, verb, adjective, adverb). Various relations such as hypernymy, meronymy and antonymy create links between the synsets of each category. It is these synsets and their relations - in particular hypernymy and hyponymy - which we shall exploit in the algorithm presented in this paper.

Synonymy as implemented in WordNet is based on the weakened form of the classical definition of synonymy. The latter is generally attributed to Leibniz (via Miller [1995]), and consists of declaring two words as being synonymous if the truth value of a sentence containing one of the words does not alter if said word is replaced with the other one. The weakened definition restricts this condition to a certain linguistic context. Because of this relaxed characterization, synonymy is not a discrete concept but rather an area on a gradient where words with high similarity are clustered together. An interesting side effect of this characterization is that synonymy does not carry across word categories. In fact, until recently there was no relation at all between word classes in WordNet[1].

WordNet is an invaluable tool in applying semantic senses to phrase heads. As indicated in section 1.3, semantic information can provide a wealth of information critical to prepositional phrase disambiguation. Beginning at chapter 4 we shall further discuss the implications of the use of WordNet for the extraction of semantic knowledge from collected 4-tuples.

[1]The latest version of WordNet introduces some weak relations between certain synsets in different word classes. For the purpose of this paper, however, we shall work from the assumption that there are no relations between synsets of different word classes in WordNet. The rationale behind that is that such relations are a new introduction and may be prone to change in the future. For example synsets in the noun category have no relation to synsets in the verb category in WordNet.

```
( (S
    (NP-SBJ
        (NP (NNP Pierre) (NNP Vinken) )
    (VP (MD will)
        (VP (VB join)
            (NP (DT the) (NN board) )
            (PP-CLR (IN as)
                (NP (DT a) (JJ nonexecutive) (NN director) ))))
    (. .) ))}
```

Figure 2.1: Excerpt from a tagged and parsed Penn Treebank source

2.3 Penn Treebank

Penn Treebank is an ongoing effort to construct a large annotated corpus that can serve as the base for research in all areas related to natural language, such as natural language processing (NLP) and theoretical linguistics. It is widely utilized as a training corpus for NLP algorithms, which makes it a good choice for any research in the area since utilizing the same corpus helps in reducing possible variants that might influence the results of corpus-trained algorithms.

Two forms of annotation are featured in the Penn Treebank: Part of speech tagging, and skeletal syntactic structure. The latter is the one which gives this corpus its name: each syntactic skeleton can be described as a syntactic tree, so the collection of these forms a *bank* of *trees* (treebank). Figure 2.1 is an example of such tagged and structured text.

The latest version of the Penn Treebank contains four annotated corpora: the Wall Street Journal, the Brown Corpus, switchboard transcripts, and ATIS[2]. Our work focused on the annotated Wall Street Journal corpus as the principal training and testing set in

[2]Automatic Terminal Information Service - a service that continuously broadcasts recorded information for the purpose of improving pilot and controller effectiveness and relieving frequency congestion by automating the repetitive transmission of essential but routine information.

order to minimize the thematic diversity of our knowledge base.

Chapter 3

Prepositional Phrase Attachment Disambiguation

Since disambiguation of prepositional phrase attachments is a common but difficult problem in natural language processing, much effort has been put into developing efficient algorithms capable of performing such disambiguation. In this chapter we survey the most common approaches.

Numerous approaches have been proposed to extract the four types of information required to disambiguate prepositional phrase attachments. In recent years the focus in natural language processing in general has shifted towards stochastical methods, since they provide an easier way of achieving good results at acceptable costs when compared to rule- and corpus-based approaches. Statistical methods, however, require a larger amount of resources than other systems, and cannot handle variations in style very well. It is for that reason that in this paper we shall only briefly cover these, focusing instead on rule- and knowledge-based alternatives.

3.1 Corpus-Based Statistical Disambiguation

In [Hindle and Rooth, 1991] a method of prepositional phrase disambiguation was proposed that utilized distributional frequencies within an automatically parsed corpus to determine the relative associative strength between a preposition and verb and noun phrase heads.

The method in which the type of association is determined - whether adjectival or adverbial[1] is based on lexical association and therefore relies purely on syntactic knowledge. The training set is composed of triples of the form [VP, NN, PP]. For each triple, the number of times it occurs within a text are counted, providing the frequency information needed for the next step. Once all the distributional frequencies have been collected, disambiguation is a matter of calculating the relative strength of association between a preposition and the verb phrase head versus the likelihood of said preposition being attached to the noun phrase head, and selecting the better one. The efficiency of this method is in the 80% range.

Like many stochastic methods, this approach relies on a very large table of probabilities. Not only is such a method prone to the data sparsity problem[2], but the size of the model acquired during training makes analysis and refinements of its workings a difficult task.

[1] In their text, the authors refer to these attachments as *nominal* and *verbal*. In order to maintain the consistency in this text, we have decided to utilize the terminology previously defined in Chapter 1. However, the words *adjectival* and *adverbial* may be replaced with *nominal* and *verbal* respectively within this context without any fear of altering their sense.

[2] The *data sparsity problem* occurs when the training corpus does not contain tuples similar to the ones occurring in the test set. The lack of similar tuples results in a deficiency in frequencies for disambiguation purposes.

3.2 Statistical Backed-Off Model

This model, pioneered by [Collins and Brooks, 1995], is a more advanced approach to statistical prepositional phrase attachment disambiguation. It works by calculating the frequencies for [VP, NP_1, Prep, NP_2] 4-tuples based on the frequencies of previous attachments with the same phrase heads. In this sense it is similar to the corpus-based statistical disambiguation algorithm. It differences itself from the former, however, in that it utilizes an approach that mimics the backed-off n-gram model in how it handles data sparsity. This algorithm falls back to triples and eventually word pairs if no frequencies for similar 4-tuples can be found. Of all purely statistical methods, this one performs best at a precision of 84.5%.

3.3 Transformation-Based Error-Driven Rule Learning

This method is a wide-spread alternative to stochastical methods in natural language processing. Error-driven rule learning starts out with a base rule which is applied to the test set. Additional rules are applied to the entries in the set which were missed by the previous rules, and the best performing one of those additional rules is appended to the formal rule chain. The process is reiterated until addition of rules stops when an arbitrary precision has been reached.

An analogous method to this one was described in Section 2.1.2. In [Brill, 1992] a part of speech tagger that applies this learning technique to determine part of speech assignment for words is described. Its author has applied the same technique to rule-based prepositional phrase disambiguation in [Brill and Resnik, 1994].

The process begins by considering always-adjectival attachment as its base rule. Once that is done, transformation rules learned from a training corpus are applied in an iterative fashion.

Error-driven rule learning performs slightly better than corpus-based statistical disambiguation at 80.8% precision. Its disadvantage, hovever, lies in that this is not an unsupervised method and requires a training corpus with pre-annotated correct attachments. The advantage to this approach is that it utilizes a small number of readable rules to achieve its high accuracy. The use of these rules, in contrast to the vast frequency tables of stochastic methods, makes a straightforward analysis of this algorithm possible.

3.4 Knowledge-Based Ambiguity Resolution

In [Harabagiu, 1996], the authors propose a knowledge-based method of prepositional phrase disambiguation. This method is trained on 4-tuples of the form [VB, NN_1, Prep, NN_2] that are extracted from the Penn Treebank corpus. [VB, NN_1, Prep, NN_2] are the heads of [VP, NP_1, PP, NP_2] respectively. Since the Penn Treebank corpus is fully parsed, all these tuples have their correct prepositional attachment disambiguated. All of the extracted tuples are sorted into different classes of semantic relationship based on the WordNet synsets that their arguments belong to. For any two adjectival attachment 4-tuples A and B, A and B belong to the same class if they both have the same preposition, NN_{1A} and NN_{1B} belong to the same WordNet hierarchy and NN_{2A} and NN_{2B} also belong to the same WordNet hierarchy. Similarly, any two adverbial attachment 4-tuples A and B belong to the same class if they both have the same preposition, VB_A and VB_B belong to the same WordNet hierarchy and NN_{2A} and NN_{2B} also belong to the same WordNet hierarchy.

Two colocated heads H_1 and H_2 belong to the same WordNet hierarchy if any of the following conditions holds true (in order of relevance):

- H_1 and H_2 are synonyms. In this case, they both point to the same WordNet synset. This is the tightest semantically disambiguated relation between phrase

heads possible with this method.

- H_1 is a hypernym/hyponym of H_2. In this case the synset pointed to by H_1 is a direct parent/child of the synset pointed to by H_2.
- H_1 and H_2 share a common hypernym/hyponym. For this condition to be true the synsets pointed to by H_1 and H_2 are siblings.

An additional set of inferential rules are used to resolve any remaining ambiguities that could not be eliminated by (or arose as a product of) the sorting of 4-tuples into classes.

Once the tuples have been sorted into different classes we just need to express the disambiguation of prepositional phrase attachments in the test corpus as a constraint satisfaction problem. The unknown variable is the attachment type of a new tuple, and clusters of prepositional classes are used as constraints. A tuple is disambiguated if there is a match for it in only one of those clusters. Because prepositional attachments in these classes are disambiguated via the utilization of semantic information extracted from WordNet, the resulting byproduct is that prepositional attachments hereby disambiguated syntactically are also disambiguated semantically.

3.5 Boosting

Boosting is a method founded on the Probably Approximately Correct (PAC) model [Valiant, 1984]. Schapire [1990] first presented a provable boosting algorithm. Similar in design to error-driven rule learning, boosting allows the user to experiment with different weak instruction (rule) sets that perform slightly better than a random guessing model. Algorithms using these weak rules can, as with the previously seen rule-based error-driven learning algorithm, be "boosted" to arbitrarily strong algorithms. A major difference, however, is that boosting stops including new rules when the inclusion of the

best performing rule candidate does not improve performance over a random guessing algorithm.

Boosting by itself is not a prepositional phrase disambiguation algorithm, but it may be used to enhance other kinds of disambiguation methods.

Chapter 4

Building a Knowledge Base

The following chapters will focus on the implementation of the algorithms used in this paper. In this chapter we will describe the extraction of tuples of the form [$Verb$, $Noun_1$, Prep, $Noun_2$] from the Penn Treebank. Following that, the classification of tuples into prepositional classes is explored. Chapter 6 describes the use of the prepositional classes as constraints for the disambiguation of unknown tuples. Chapter 7 covers the analysis of the results from the algorithm constructed during Chapters 4 - 7. Finally, Chapter 8 explores changes and improvements applied to the base algorithm to improve its performance.

4.1 The Knowledge Base

The first step to producing an algorithm capable of performing prepositional phrase attachment disambiguation is to construct a knowledge base for it. One of the most important tasks to training such a knowledge base is the extraction of information from training texts into a format that is usable by the algorithm. We have decided to focus on the Treebank corpus [Marcus et al., 1993] as our main training text from which to extract the required information. The reason for this is that it is the same corpus that is

used by [Harabagiu, 1996], which we are using as the baseline for our algorithm.

4.2 Splitting the Corpus into Training and Target Sets

We have chosen the treebank corpus described in Section 2.3 to train and test our prepositional phrase attachment disambiguation algorithm. To do this we split the corpus into two sets: The target set, which is composed of 10% of the Penn treebank corpus randomly selected from its Wall Street Journal section, and the training set, which consists of the remainder of the Wall Street Journal treebank corpus. The knowledge base is built with the training corpus, and the trained algorithm is tested on the target set. The advantages of this approach are two-fold: The target set stems from the same subset of English sentences as the training corpus (business English in this case), so both sets are semantically connected; and the target set is already parsed, so we can use the same algorithm that was used to build the knowledge base to extract the solutions from this set, which can thus be used to procure vital statistics as to the precision and recall rate of our algorithm.

When selecting the random 10% from the Penn treebank corpus for our target set we had to decide how to perform the sampling. The treebank corpus is split among $\tilde{2}300$ files. It can be assumed that the target tuples are more or less evenly distributed among these files, and empirical data indicates that sampling 10% of the files composing the Penn treebank corpus is roughly equivalent to sampling 10% of each tuple set. Therefore it is possible to reduce the sampling task to selecting 230 files out of the entire treebank set. A better solution, however, is to first parse the entire treebank set, then select 10% of all tuples of the adjectival kind and 10% of the tuples of the adverbial kind. Since the sets for adjectival attachments and adverbial attachments do not overlap, we are guaranteed to again have 10% of the total set. The tools we created to do the sampling sport an adaptable design. Both methods can be applied with equal ease. Since the

second solution is more reliable in terms of producing a uniform number of tuples we decided to favor it.

Another decision to be made on the topic of sampling for the training corpus is whether we want a completely random sample or whether we want to force the sampling evenly throughout the treebank corpus (for example, by randomly selecting one file out of every ten). In the end we concluded that a completely random sample is less prone to bias error, and that the fully random sampling performs well enough at avoiding clustering of samples (which would be one reason to smooth out the sampling process).

The splitting of the input corpus was performed immediately after all tuples had been extracted from the Penn Treebank corpus, but *before* we attempted to assign each of the tuples' elements to their WordNet synset. This decision was made in order to leave tuples in the test set that contain elements that do not match any WordNet synset. Any action otherwise would have artificially inflated the recall rate of the algorithm and could have led us to a skewed analysis of the results. Out of a total of 12407 adjectival attachment tuples, 8685 were used to train the algorithm. A total of 5095 adverbial attachment tuples out of 7279 were also used for algorithm training purposes. The remainder was kept to construct the test set.

4.3 Committing Parsed Strings into Data Structures

In section 1 we indicated that we would be working with 4-tuples of the format

[$Verb$, $Noun_1$, Prep, $Noun_2$]

It is these tuples that we are interested in extracting from the Treebank corpus. However, we additionally want to acquire another kind of information that is embedded in the training text, namely the correct attachment for the prepositional phrase. As a result, the actual data structures that interest us are of the following kind:

[$Verb$, [$Noun_1$], [Prep, $Noun_2$]].

and

[$Verb$, [$Noun_1$, [Prep, $Noun_2$]]].

where the former indicates that the prepositional phrase is attached to the preceding verb phrase, and the latter indicated that the prepositional phrase is attached to the preceding noun phrase. Note that in this form, it is easy to determine the form of the prepositional phrase attachment by the level of its nesting: A third-level nesting of the prepositional phrase indicates that it is of the adjectival kind (attached to the preceding noun phrase), whereas a second-level nesting of this phrase indicates that it is of the adverbial kind (attached to the preceding verb phrase). In addition Ruby[Matsumoto, 1995] [Thomas et al., 2004], the programming language that we use, further simplifies extraction of individual elements of the nesting by offering methods of "flattening" nested arrays. Flattening nested arrays converts them into a single array, where individual elements are indexed as they are encountered during a depth-first traversal. The two structures above for example are transformed into the following one

[$Verb$, $Noun_1$, Prep, $Noun_2$].

which features the same layout as the 4-tuple mentioned at the beginning of this section.

In order to store the type of attachment of these tuples, we first decided to place the strings containing the parsed and tagged sentences into nested arrays that reflect the parsed structure of the training corpus. Fortunately this was facilitated by the division of parsed structures with parentheses in the parsed texts. The resulting data structure is in fact a tree, with leaves represented by the tokens extracted from the strings (in this case, each token is a word) and roots represented by arrays. It is therefore easy to differentiate between roots and leaves by checking the type of the object that is being handled.

Once all sentences were parsed into their respective data structures, these data structures were refined to contain only the necessary information.

4.4 Composite Verbs and Nouns

An important decision was what to do with groups of words that form a single semantic unit, such as *will be* and *interest rates*. We were faced with two choices. The first choice was whether to use all the words together as a single token. It was quickly decided to not group these words into one token, because that would greatly reduce the number of tuples that may be grouped by similarity class. The second decision to be made arose as a result of the first one, namely which of the words to choose as being representative of the group. In the first case of words grouped into a semantic unit the words involved are different parts of speech. These groups were encountered when dealing with nested "chunks" - parsed syntagmas, such as verb phrases and noun phrases - of the same type. For "*will be*", *will* is the modal and *be* is the verb. We ultimately concluded that for these cases it was sensible to assume that the best fit part of speech would be the one that identifies the chunk that it occurs in. The example above was extracted from a verb phrase, so we picked the base form, *be*, as the representative part of speech of this unit.

The second case is trickier - both words are the same part of speech and are encountered at the same level within a phrase, so there is no solid syntactic information for the parser's decision making process. In the end it was reasoned that there are two kinds of composite nouns of interest - composite proper nouns, such as *Brenda Malizia Negus*, *New York*, and *Federal Reserve*, and the rest (such as *percentage point*). The latter case is easier for us, since we concluded that the former word of such a composite noun acts as an adjectival modifier. By inference we take the latter noun to be the representative of this particular syntagma. For these kinds of composite nouns our task is reduced to

finding two consecutive noun terminals - that is, an array with two elements where the first element matches the regular expression */^NN/* - and eliminate the former.

For composite proper nouns, however, we decided that all the words composing this syntagma are representatives of it. Consequently we had to find a way to prevent these from being reduced as the other composite nouns were. The first idea was to look for capitalized nouns, under the assumption that two consecutive capitalized nouns indicates that both of them are proper nouns. This is a weak point to make decisions on, however, and better suited as a complement to a part of speech tagger than an actual parser. Parts of speech, hovever, do provide a solution to this quandary. Identification of proper nouns is easy to implement by taking advantage of the additional information provided by the tags available in the Penn treebank corpus. In this treebank, all proper nouns are tagged with the *NNP* tag. Since *NNP* is also matched by the */^NN/* regular expression it follows that composite proper nouns are a subset of the entire set of composite nouns we collected and reduced in the above algorithm. Therefore we can modify such algorithms to identify the special case of two consecutive *proper* nouns (matched by the regular expression */^NNP/*) and concatenate them instead of eliminating the first of the two. With this algorithm, proper nouns like *"Elsevier N.V."* are transformed into *"Elsevier_N.V."* (note the underscore) instead of just *N.V.*.

This solution to compressing composite nouns is robust and uses intrinsic information available in the source text to make a decision. It is also capable of handling composite nouns containing more than two words by performing a step-wise reduction. For example *"Brenda", "Malizia", "Negus"* is first transformed into *"Brenda_Malizia", "Negus"* and then into *"Brenda_Malizia_Negus"*.

4.5 Refining Parsed Sentences

All the work mentioned thus far in this chapter was done to prepare the training corpus for our parser, whose sole purpose is to identify and collect the required 4-tuples from said corpus. This section describes the different approaches that have been taken to implementing this parser, and the varying levels of success achieved by each approach. We have identified and implemented three possible solutions: One bottom-up parser and two top-down algorithms.

4.5.1 The Preemptive Bottom-Up Approach

The first idea to refining the data structures obtained from the parsed sentences is focused on approaching the solution of parsing the text by utilizing homogeneous attachment features as an anchor point. For the purposes of this application, the only homogeneously formed features are the prepositional phrases. These phrases are closest to the leaf end of the parse tree, so once identified they allow our application to retrieve the rest of the required structures by working its way up that tree. This approach is very similar to how bottom-up parsers work, and draws its name from them.

The basic feature of the algorithm is that it iterates over each element of the array that we use as a data structure, recursing into each nested array it encounters. In other words, the algorithm is instructed to descend the tree that is formed by the arrays on a preorder depth-first basis. When the algorithm encounters a preposition, the recursion stops and begins with the extraction of the preposition embedded in this phrase. The algorithm is then instructed to find the nearest attached noun phrase and to extract the phrase head from it. As has been previously discussed, extracting the phrase head for this application is equivalent to extracting the first noun. The preposition and noun thus obtained are placed into an array, forming the [Prep $Noun_2$] structure common to both

the target data structures we expect to construct. The array is then returned, putting an end to the recursion, and initiating the second phase of the information refinement algorithm.

Upon receiving a return value from a recursive call, the algorithm checks the length of this value to decide what the next course of action should be.

- If the return value is *nil* or if the length of the return value is zero, then the recursive algorithm encountered the end of the nesting of the arrays without finding a candidate structure for a 4-tuple. The algorithm then proceeded to iterate over the next elements in the current level of nesting of the arrays.

- If an array of nonzero length is returned, then the algorithm has run into a prepositional phrase down its recursion path[1]. If the length of the array returned is two or three[2], then the algorithm has to make a decision based on the state of the current nesting level:

  - If the current level is a noun phrase and the length of the returned array is *two* then the prepositional phrase is of the adjectival kind. The algorithm recurses its current nesting level to find the nearest [3] noun, and builds an array with this noun and the array containing the prepositional phrase, thus setting the base form for an adjectival attachment 4-tuple. The array is then returned to its upper level for further actions.

  - When the length of the returned array is 3 and the current level is a verb phrase, then we are dealing with the return value of the above item. The task

[1] It is interesting to note that the minimum nonzero length of the array that may be returned is two, namely the array containing a preposition and a noun.

[2] A length of four indicates that a full 4-tuple has been extracted.

[3] By finding the nearest element we refer to recursing down the array tree in a depth-first manner and terminating the recursion on the first element encountered.

at hand then is to find the nearest nested verb and finish the construction of the adjectival attachment 4-tuple, which is then returned.

- Sometimes a case is encountered where the current level is a noun phrase and the length of the returned array is *three*. It is possible that the algorithm is dealing with a composite noun in this case (commonly a proper noun), so our algorithm finds the closest nested noun and *prepends it* to the noun in the array returned. This way the related nouns are joined in one object, and the array depth is not altered. Early implementations of this approach did not perform this action and were subject to discrepancies. They would encounter situations where a 4-tuple contains nouns where a verb is expected. It was identified that nested noun phrase levels were the reason for this occurrence. Any algorithm not designed to handle this kind of event would be unable to create 4-tuples of the proper form. The changes described above are one solution to solving this issue.

- The last case to consider is the situation when the algorithm receives an array of length 2 from a recursive iteration, but is at a verb phrase level during the current iteration. This case occurs when the prepositional phrase is unambiguously attached to a verb phrase. A "dumb" algorithm that performs no checks between the return value of its received return value and the phrase that it is currently located at will assume that it is at the last steps of the process of finding an adverbial 4-tuple. Such a naive empirically-oriented algorithm would then attempt to find the nearest nested verb and noun in the parse tree and prepend them to the returned array.

  Such an assumption, however, is erroneous. The expected 4-tuple is only present if the text in fact contains an ambiguous sentence of the form [VP,

NP(The visitors) VP(eat) PP(with NP(chopsticks)).

Figure 4.1: Unambiguous prepositional phrase attachment

NP_1, Prep, NP_2]. Since it is just as likely, however, that we are dealing with an unambiguous prepositional phrase attachment, such as the one described in figure 4.1. A parser that ignores these cases will fail. As can be inferred from the above sentence, the first noun phrase is missing, thus making the second noun phrase the nearest one to the verb phrase and returning the second noun as the "first" one. In this case the naive unbound algorithm would produce a 4-tuple of form ["*eat*", "*chopsticks*", "*with*", "*chopsticks*"].

As has already been mentioned, the problem in this case is that no bound checks are performed to match the current position of the algorithm within the parse tree and the return value. In other words, the recursive algorithm which finds the respective parts of speech (noun, verb, preposition) has no delimiter as to where it should look. A possible solution to this dilemma is to have the invoking method exclude or prune the branch that contains the prepositional attachment from the data structure that the recursive find method receives. This solution, however, is not very elegant in that it provides a postorder solution to a preorder algorithm. Furthermore, by introducing a top-down solution to a problem caused by bottom-up parsing unnecessarily increases the complexity of the algorithm, introducing additional problems of interaction between recursive iterations and increasing the frailty of the algorithm in general.

- A returned array length of 4 indicated that a full 4-tuple has been extracted, and it is thus passed up along the hierarchy for collection by the driver program.

As it is currently exhibited, the above algorithm displays a number of flaws and drawbacks that indicate that a preorder bottom-up method is not the ideal approach to extracting 4-tuples from tagged and parsed sentences. Some of these issues, such as the nested noun hierarchy and the case where there is no noun phrase on an adverbial attachment, have already been addressed in the above sections. One important issue, however, has not yet been solved, namely the non-recursive nature of the algorithm after the first prepositional phrase has been encountered. Any sentence that contains two prepositional attachments will only produce a 4-tuple (if any) on the first prepositional phrase encountered, because all recursion stops at that point. This again is a problem inherent to the preorder bottom-up approach, because there is no natural way of keeping a running memory of each step. Keeping a running stack of the tuples that have already been crawled by the algorithm might be a way to maintain the needed information. It was decided not to implement a separate stack because the other approaches considered provide more elegant solutions.

The elemental reason for the unsuitability of the bottom-up approach is that, unlike context-free grammar parsers, it lacks a good set of grammar rules to reconstruct phrases from tokens. Such a set, if it were to exist, would have to be very extensive in order to provide support for the various nuances of English grammar. Bottom-up parsing is an all-or-nothing approach, and thus the least fit of the three proposed systems to perform extraction of ambiguous prepositional phrase tuples.

4.5.2 The Tiered Top-Down Approach

In the previous section it was determined that a preorder bottom-up approach to extracting 4-tuples from parse trees is an inefficient solution. It has also been concluded that while a bottom-up approach is capable of easily locating prepositional phrase attachments, top-down techniques are better fit to perform the fine-grained parsing required

for the extraction of ambiguous 4-tuples. Top-down approaches have the additional advantage of performing their recursion in the same order in which it is desired to maintain a parse history. By combining the parse history into the recursion stack the need for an additional data structure to maintain this information can be eliminated, simplifying the algorithm and making it more robust.

The first observation of the running method of top-down systems is that a such an approach will first run into the verb phrase section of the ambiguous branches. By predicting where the algorithm enters the branches it is possible to determine an approach to tackling a purely top-down algorithm. The idea is founded on the realization that each consecutive part of the 4-tuples that we are looking for is nested deeper within the verb phrase root. This would indicate that a tiered approach is better suited for the extraction of 4-tuples than the preemptive bottom-up approach. The steps involved in examining the parse tree with this method are as follows:

- The initial traversal algorithm recurses down the entire sentence structure searching for the beginning of a verb phrase. Recursion stops here, and the branch containing this verb phrase is passed on to the next part of the algorithm.

- The part of the algorithm that deals with the verb phrases is the most complex one because it is the one where there are the highest chances for the algorithm to incorrectly parse a branch.

  While the perorder bottom-up approach described in the previous section is practically blind to the structure of the branch that the prepositional phrase is located in, a top-down system has no such limitation. It is therefore possible for the parser to first perform a check and confirm whether it is possible at all for the branch in question to contain the 4-tuple that is expected by establishing the existence of the necessary branch structure. The implementation of this idea is straightforward:

The branch is scanned for the appearance of noun phrases, a prepositional phrase, and another noun phrase in a depth-first manner. Depth-first scanning ensures that the phrases are present in the correct order. Because this is intended to be a tiered algorithm, this check is performed by ignoring the actual nesting of the elements. That is, it is assumed that any sequence of the form [NP, PP, NP] will indicate the presence of either of the target tuples within the verb phrase, regardless of the elements encountered between each of the tokens of the sequence. To perform such a scan the only thing necessary was to flatten the data structure[4] and run a regular expression on it to find the three tokens.

Ignoring the nesting of the phrases within the verb phrase branch is a quick method of locating candidates for parsing, but not without problems. Several of these were encountered during initial tests of this algorithm, and close examination of the nested phrases unveiled new and unexpected sets of difficulties not encountered by the bottom-up approach. The first problem is that the presence of a noun phrase element does not always indicate the presence of a noun. The reason for this problem is that the part of speech tagger applied to the parsed texts sometimes inserts null tokens where an object has been left out of the sentence. Figure 4.2 is an excerpt of a sentence containing a typical occurrence of null tokens. The phrase that causes a problem in this case is the noun phrase. It is a null noun phrase, meaning that while it is a part of the parse tree it does not contain any words, only the *NONE* null token to indicate that it is empty. Therefore it appears to our parser as containing a sequence of [VP, NP, PP, NP] phrases, while actually being just an unambiguous sequence of [VP, PP, NP] phrases. In order to deal with this issue it was decided

[4] A flattened data structure is obtained by taking a structure composed of nested arrays and recursively extracting each of the nested arrays into their parent array. The result is a one-dimensional array of items as encountered in a preorder traversal of the nested arrays.

```
(NP
  (NP (DT the) (CD 400) (JJ taxable) (NNS funds) )
    (VP (VBN tracked)
    (NP (-NONE- *) )
    (PP (IN by)
      (NP-LGS
        (NP (NNP IBC) (POS 's) )
        (NNP Money) (NNP Fund) (NNP Report) ))))
```

Figure 4.2: Excerpt of a parsed sentence containing a null noun phrase.

that it would be necessary to not only search for a [NP, PP, NP] syntagma, but also for a parallel [NN, IN, NN] syntagma. However, the initial implementation of this solution made the did not draw any direct ties between the phrases and the parts of speech sought, resulting in mismatched phrase and phrase head sets. This mistake was quickly discovered, and a better solution was drafted. The new system was set up to include the additional tokens into the regular expression, as they are expected to occur in the vicinity of the former symbols. Therefore the set that is searched for is [NP, NN, PP, IN, NP, NN]. There is, however, still a problem with this technique. Regular expressions are greedy, meaning that they will try to find the most extensive string that matches their requirements. A much better solution would be to utilize an algorithm that tries to minimize the distance between the tokens. The catch to implementing such an algorithm is to define a measure for distance between tokens. Token counting is not a good solution, since phrases may contain a variable amount of tokens. While it is not impossible to implement a system that maintains a measure of token distance, the work that is expected to be done by the parser in order to properly determine whether there actually is a tuple in a sentence is a greater part of the effort needed to actually extract such a tuple from that sentence. The proposed preorder top-down parser

therefore may not be the most appropriate for the extraction of the needed tuples from the array tree. Like the bottom-up parser, this system became frail due to unexpected changes in the structure of the parse trees.

4.5.3 The Backed-Out Top-Down Approach

Because the preemptive bottom-up parser and the tiered top-down system were implemented before work on the backed-out top-down approach had begun, the latter was implemented with the complications of its predecessors in mind. If anything is to be learned from the tiered top-down approach, then that is that it is of key importance to minimize the syntactic distance between the tokens that compose each tuple. We have observed that the two previous attempts at creating a parser lacked any means of maintaining minimal syntactic distance. The solution to this shortcoming in the parsing process is as trivial as it is easy to overlook. By switching the system from a preorder traversal algorithm (as was utilized by the two previous systems) to a postorder one, it is possible to reduce the complexity - and thus the frailty - of the parser.

- The size of the branch that is to be parsed can be minimized via postorder traversal by working on verb phrases as the parser backs out of each branch. By looking for matches while backing out of the branches the parser maintain a minimal match pattern like the bottom-up approach. The postorder top-down system lacks the shortcoming of the bottom-up approach because it has already built a parse stack when it traversed towards the leaves. In order to eliminate duplicates with nested verb phrases the branches that have already been parsed are pruned as they are left.

- The problem of null noun phrases encountered in Section 4.5.1 and depicted in Figure 4.2 was tackled by limiting the recursive finding algorithm to only a certain

set of allowed phrases. These sets are composed of the phrase that the desired word is the determiner for (noun phrases for nouns, prepositional phrases for prepositions, and verb phrases for verbs), simple declarative clauses (S), and unlike coordinated phrases (UCP). The likelihood of UCP tokens appearing in the text is minimal - in fact it is the most unlikely phrase of all of the possible ones - but it was decided to include it for completeness' sake.

- Each of the components of the 4-tuple are searched for sequentially within the verb phrase. The parser first looks for the verb, then the first noun phrase. The part responsible for finding the first noun phrase also attempts to find an attached prepositional phrase. If one is found and we have a full tuple set (a [VB, NN, IN, NN] set with no null tokens) then we tag this tuple as being a tuple of the adjectival kind. If no prepositional phrase is found attached to the noun phrase then we attempt to find one attached to the verb phrase. As before, we check if a full tuple set is available, and in the affirmative case we tag it as being of the adverbial kind.

All in all, the approach thus described features some major advantages. It boasts a very simple and compact design by utilizing the recursive call stack as a parser stack, and has shown itself to be robust and resistant to fluctuations in the input text.

4.6 Chapter Summary

In this Chapter we have covered how the pre-tagged text in the Penn Treebank corpus is parsed to extract the needed tuples of the form [$Verb$, $Noun_1$, Prep, $Noun_2$]. First the corpus was split into two sets, a training set and a testing set. Splitting was performed by random selection. The training set is composed of 90% of the total amount of tuples of the corpus, while the rest of the tuples constitutes the test set.

The approach taken to parsing the corpus was to translate the tagged text into tree-like data structures and to explore different traversal strategies to find the best fit strategy. It was decided that a postorder traversal form brought forth the best results because it is capable of building a parse stack and analyze it in a single pass. The building stage is the traversal down towards the leaves, while the analysis stage is the return. The tokens of each tuple are kept tightly clustered by pruning used tokens during the analysis. Pruning prevents these tokens from occurring multiple times in tuples.

Chapter 5

Semantic Classes Of Prepositional Attachments

In this chapter we describe how we classify the tuples extracted with the algorithm described in Chapter 4 into prepositional classes. These classes are then used in the algorithm described in Chapter 6 as constraints for the disambiguation routines. We used the rules described in Section 3.4 as our core rules when performing this classification.

5.1 Assignment Of Synonym Sets To Terms Of Tuples

Semantic classes as described in Section 3.4 can only be built with tuples for which all words involved in an attachment can be disambiguated. This means that for each 4-tuple [VB, NN_1, Prep, NN_2] we need to find either

- The senses for VB and NN_2, if the attachment is adverbial, or
- The senses for NN_1 and NN_2 if the attachment is adjectival.

This is performed *before* any attempt is made to cluster the 4-tuples into classes of prepositional relations, but after the splitting of the input corpus into training and test sets in order to avoid any artificial skewing of the distribution of tuples.

5.1.1 WordNet Difficulties

It was decided at this point that it would be useful to collect some statistics on the results of matching tokens in tuples to WordNet synsets. A surprising fact was that the most recent algorithm resulted in an unexpected low recall rate of about 10%. An investigation into the kind of tuples that were rejected revealed that the problem resided in the algorithm passing a literal transcript of the terms as found in the Penn Treebank corpus to the WordNet library for the Ruby programming language [Matsumoto, 1995] [Thomas and Hunt, 2002] [Thomas et al., 2004] . This library turned out to contain a deficient word normalization algorithm. The normalization algorithm only looked up transformation tables to find the base forms of words, and did not perform any direct transformations by itself. In order to solve this issue, the normalization task was delegated to an external tool that made direct use of the original WordNet libraries. This modification considerably improved the recall rate of sense assignment to adjectival attachment tuples, but adverbial attachment tuples remained at what seemed to be an uncommon low 50% recall.

5.1.2 Symbols As Nominals

Our investigations revealed that another reason for the low recall in our algorithm was the way in which contractions were parsed in the Penn Treebank corpus. "$It's$" was parsed into a syntactic structure that can be roughly described as $NP(\ NN - It\ )\ VP(\ VB - 's$), "$They're$" into $NP(\ NN - They\ )\ VP(\ VB - 're\ )$, and so on.

Because of the high number of occurrences of tuples containing these cases it was decided to pre-parse the input set by transforming all occurrences of *'s* and *'re* into *is*

and *are*. When looking for further contractions we also encountered a number of cases of year contraction which we expanded as well (*'80s* to *1980s*) and three cases where the Treebank corpus contained misspelled or misparsed text. These are as follows:

- *maitre 'd*, where it should have been *maitre d'*;
- *fancy'schvartzer* where it should have been *fancy schvartzer*;
- *the'breakup* where it should have been *the breakup*

Since these were very rare (three occurrences of *maitre 'd* and one occurrence of the other two each) we decided to not fix them in the input set since doing that would not significantly improve the performance of the classification algorithm.

That covered all the ambiguous symbols with a tilde in them. Another common case was the appearance of "%" as a noun. WordNet performs no translation from "%" to "*percent*", thus leaving this symbol in an ambiguous state. We manually corrected this case because it is a common occurence.

If not considered carefully, the alterations described in this section could be regarded as changing the status of our algorithm from unsupervised to supervised. We can justify these modifications because they solve a shortcoming of WordNet, and do not require any further human interaction once implemented.

5.1.3 Polysemous Normalization

A problem that is related to semantic class polysemy can be encountered during the normalization process described in the previous sections. This problem occurs when a word can be normalized to two different synset entities. Examples are the words "$fell$" and "*days*". "$Fell$" can be the past tense of "$fall$", or it can be the infinitive of "$fell$" itself. Similarly, "*days*" may be the plural of "*day*", but it also appears in a synset as a collective noun.

1. (1) days, years – (the time during which someone's life continues; "the monarch's last days"; "in his final years")

Because we have no way of determining which of the two (or more) normalized forms of the word we should deal with, our algorithm creates a combination of all possible forms. A tuple of the form ["*fell*", "*in*", "*days*"] would thus create the following normalized combinations:

- ["fall", "in", "day"];
- ["fall", "in", "days"];
- ["fell", "in", "day"];
- ["fell", "in", "days"].

We do not handle this from the normalization as a particular problem, rationalizing this lack of preoccupation by assuming that, as with polysemous words, most of the combinations will be discarded because of data sparsity - namely the lack of other tuples matching the corresponding phrase heads' synsets. An algorithm that focuses on maximizing its precision should eventually discard most of these tuples at the cost of recall, whereas a recall-focused version of such an algorithm should attempt to maximize the amount of combinations resulting from the tuples in its input. Later in this chapter we will discuss a number of approaches regarding precision versus recall as applied to polysemous words - that is, combinations that lead to multiple senses from a single normalization of a word. The approaches thus discussed can be ported back and applied to solving the issue of aggregate word transformations from a single source.

	Attachment kind	
	Adjectival	Adverbial
Total	8685	5095
Recalls	1211	322
Normalizations	8366	5218

Table 5.1: Results of normalizations

5.1.4 Effects of normalization and WordNet lookup

After applying the enhancement to allow combinations of polynominalizations, we wanted to get some statistics regarding the usefulness of normalizations. The reason for doing so is because this tool is also utilized to determine the recall for training the disambiguation algorithm on phrase heads. Table 5.1 shows the results obtained. We began with the 8685 adjectival 4-tuples and 5095 adverbial 4-tuples resulting from the work in section 4.2. After applying the normalization, 1211 and 322 tuples were discarded from the adjectival and adverbial tuple sets respectively. These were the tuples that had one or both phrase heads which could not be matched to any sense in WordNet. The remaining row, "*Normalizations*", lists the number of tuples obtained after the normalization process.

Note that there are more normalizations of adverbial tuples than there were original tuples. This peculiarity can be traced back to the polysemous normalizations. For any tuple that has one phrase head with n normalizations, n tuples will be created. Tuples that have two polysemous phrase heads will produce $n \times m$ tuples, where n is the number of normalizations possible for one phrase head and m is the number of normalizations possible for the other head. This causes the statistics collected in this section to be less useful than they could be, but not completely unreliable. A possible alteration that would increase the value of these numbers would be a change in how they are collected - if the normalizations and recalls can be clustered by their source, the statistics presented here can be made more accurate. Such a change was not implemented because it would

require a considerable amount of retroactive changes to our system which would degrade its performance in a manner that would cause the drawbacks to outweigh the benefits.

5.2 Tuples That Can Successfully Create Classes

After the work described in Section 5.1, two sets of tuples were produced:

- One aggregation of adjectival attachment tuples for which both nouns have one or more WordNet synsets assigned to them.
- Another collection of adverbial attachment tuples which have their verb and prepositional noun paired with one or more WordNet synsets.

Even though it is possible to perform an exhaustive comparison between all tuples within each set, not all of the relations between tuples A and B - where tuple A is composed of [H_{1A}, Prep, H_{2A}] and tuple B is composed of [H_{1B}, Prep, H_{2B}] - will produce a synonymy class that is useful to our algorithm[1].

The points listed here were discovered upon our first experimental attempt at building a collection of semantic classes based on synonymy alone. The failure of that attempt led to a further investigation into the kind of tuple comparisons that may turn out problematic results. In this section we address conclusions drawn from this investigation.

A number of tuple comparisons will produce semantic classes that cannot disambiguate one or both of the phrase heads in each tuple. In other words, we cannot reduce the number of senses that are possible for these phrase heads to one sense per head.

[1]Tuples with unmatched prepositions are not considered in any of these sections because they are very likely to act on different senses of a word. We therefore only consider interactions between tuples that share the same preposition.

5.2.1 Unmatched Tuples

The most obvious of the ambiguous cases is the one where the two tuples don't fall within any of the categories defined in Section 3.4. That means that at least one of the [H_{1A}, H_{1B}] or [H_{2A}, H_{2B}] pairings don't share a synonymy hierarchy (be it a synonym, parent/child, or sibling relation). Because there is no match between co-located phrase heads, these cases fall through the algorithm and produce no class at all, so we do not have to worry about them where produced semantic classes are concerned.

5.2.2 Multiple Classes From A Pair

A more difficult case occurs when we cannot reduce the synsets that match a colocated pair of phrase heads to one. This is particularly evident when either the H_1 and/or the H_2 phrase heads in both tuples contain the same word, but it can also occur when two words occur in more than one WordNet synset. This last subject was briefly discussed from the normalization point of view in Section 5.1.3. In the next section we will discuss the implications of these classes in terms of semantic class construction.

During the experimental version of our algorithm a number of instances were encountered where exact duplicates of tuples were obtained. This particular issue was resolved by eliminating duplicate entries from the training set. A list of the tuples that were removed was kept, however, and a count of the total number of occurrences of each particular tuple. Tuple frequencies have been utilized previously to calculate the chances of a prepositional phrase attachment being affiliated to the preceding noun phrase or verb phrase. By keeping a record of the tuples with multiple occurrences, sufficient information has been retained to allow the construction of a fallback statistical algorithm that makes use of this piece of information. From the tuples resulting from the normalization process, 4573 unique adverbial attachment tuples were found, 317 of which had one or

	Tuple		
1	["rise",	"in",	"quarter"]
2	["climb",	"in",	"quarter"]
3	["fall",	"in",	"quarter"]
4	["begin",	"with",	"service"]
5	["start",	"with",	"overhaul"]

Table 5.2: A sample of ambiguous tuples

more duplicates. A total of 7268 unique adjectival classes were found, with 540 of those having duplicate tuples.

Table 5.2 shows a list of less obvious tuple combinations where combinations of phrase heads can lead to an inability to clearly pair them with a particular sense. When cases 1 and 2 are compared, the sense of the verb phrase heads can be pinned to sense number 9 of the word "*rise*"

> 9. (4) wax, mount, climb, rise – (go up or advance; "Sales were climbing after prices were lowered")

because that is the only one where "*rise*" and "*climb*" co-occur. We cannot disambiguate the noun phrase head, however, because we do not know which of the 13 senses of "*quarter*" apply. A similar issue occurs with cases 1 and 3. We can fixate "*rise*" to its first sense and "$fall$" to its 15th sense because they share a common hypernym:

> 1. travel, go, move, locomote – (change location; move, travel, or proceed; "How fast does your new car go?" "We travelled from Rome to Naples by bus"; "The policemen went from door to door looking for the suspect"; "The soldiers moved towards the city in an attempt to take it before night fell")

While the verb phrase head can be bound to one sense, the noun phrase head cannot be disambiguated. Lastly, cases 4 and 5 present a narrower ambiguity which, despite its

limited scope is probably the most important ambiguity on this subject. For these two cases exhibited in the table we are able to disambiguate the noun phrase heads to the 8th sense of "*service*",

> 8. (17) overhaul, inspection and repair, service – (periodic maintenance on a car or machine; "it was time for an overhaul on the tractor")

On the other hand, five senses out of ten of "*begin*" also include "*start*" in their synonym list, as follows:

> 1. (379) get down, begin, get, start out, start, set about, set out, commence – (take the first step or steps in carrying out an action; "We began working at dawn"; "Who will start?"; "Get working as soon as the sun rises!"; "The first tourists began to arrive in Cambodia"; "He began early in the day"; "Let's get down to work now")
>
> 2. (58) begin, start – (have a beginning, in a temporal, spatial, or evaluative sense; "The DMZ begins right over the hill"; "The second movement begins after the Allegro"; "Prices for these homes start at $250,000")
>
> 3. (27) begin, lead off, start, commence – (set in motion, cause to start; "The U.S. started a war in the Middle East"; "The Iraqis began hostilities"; "begin a new chapter in your life")
>
> ...
>
> 7. begin, start – (have a beginning characterized in some specified way; "The novel begins with a murder"; "My property begins with the three maple trees"; "Her day begins with a work-out"; "The semester begins with a convocation ceremony")
>
> 8. begin, start – (begin an event that is implied and limited by the nature or

inherent function of the direct object; "begin a cigar"; "She started the soup while it was still hot"; "We started physics in 10th grade")

Once again, we are left with an ambiguous semantic class.

5.2.3 Solving Polysemous Classes

One alternative to discarding the semantic classes for polysemous tuples that was considered is a case where one a tuple of the form ["*get*", "*with*", "*service*"] were to be encountered in addition to cases 4 and 5. A smart algorithm could infer that this tuple complements that ambiguous semantic class and utilize it to disambiguate this class. This, however, would add an extra level of complexity to the algorithm by introducing a form of backtracing. Such a form of backtracing is redundant because the target semantic class can be reached in most cases via two independent paths, namely by building a disambiguated class for the tuple and case 4 and another version of the same class for the tuple and case 5. Duplicate classes can be eliminated with a postprocessor. A special case that cannot be handled by this solution is one where we have three tuples A, B and C, each with different phrase heads (a, b and c), and four senses 1, 2, 3 and 4 where

- a and b are contained within sense 1;
- a and c are contained within sense 2;
- b and c are contained within sense 3;
- a, b, and c are contained within sense 3.

In this situation no single comparison between two tuples can yield a completely disambiguated semantic class. That would seem to indicate that the solution we seek in order to reduce the scope of these classes to one sense is non-trivial. One could attempt to tackle the situation by attempting to refine ambiguous semantic classes (and maybe

create new ones). Such an undertaking could be accomplished by running the class construction algorithm again with the extracted classes as input. This is briefly discussed in Chapter 9.

One could argue that attempting to refine these ambiguous classes is counterproductive since it could be contended that for any ambiguous semantic class, all combinations presented by the polysemous semantic class could be valid. This argument is an optimistic view of the situation and is to be considered problematic. The final part of the algorithm is tasked with determining the attachment type of a 4-tuple. To do so it casts this task as a constraint satisfaction problem, with the semantic classes acting as constraints. If these constraints are too permissive, errors will creep in and the performance of the algorithm will suffer. On the other hand, while it is entirely possible that some of these combinations may be incorrect, thus being a potential source for error in the algorithm, we speculate that the impact of incorrect combinations is of no more concern than the effect of phrase heads with a large number of senses - such as "*is*" - occurring frequently.

The bottom line is that the choice between discarding or including ambiguous classes can be cast as deciding whether to give precision or recall more weight. Tossing out ambiguous semantic classes should improve precision as it eliminates potentially false positives. On the other hand, disposing of these classes will have a considerable negative impact on recall, because the total of combinations producing ambiguous semantic classes out of tuple pairings is higher than the sum of unambiguous semantic classes produced via the same process. There may be some way to improve that ratio, but addressing that would bring us out of the scope of our problem.

For the implementation of our algorithm we decided to discard semantic classes produced from tuple pairs with headers whose words are the same, but kept the polysemous classes created from headers that do not contain the same words yet produce multiple

sense solutions.

5.2.4 Relational Transitivity

During the early experimental versions of our algorithm we produced a very low recall rate. An inspection of that algorithm led to an important observation regarding the nature of class relations in terms of transitivity.

The initial algorithm only looked up hypernym/hyponym relations from tuple A to tuple B, under the assumption that a hypernym relation from B to A would be caught as a hyponym relation from A to B. This assumption is in fact correct, but it is not precise. We discovered that the conditions under which we were running our algorithm allowed for relations between tuples A and B to exist in which the prepositional class formed by the hypernym relation from A to B did not coincide with the prepositional class formed by the hyponym relation from B to A. Neither of the classes were incorrect, but the set that resulted as a combination of both classes was larger than the set of each individual class.

The ultimate conclusion is that for the purpose of constructing prepositional classes the assumption that paired WordNet relations - hypernym/hyponym relations in particular - are transitive must not be made. Therefore, when applying this axiom to the implementation of the algorithm responsible for determining prepositional classes, we must keep in mind that mere combinations are not sufficient to exhaust all the possible relations between tuple A and tuple B. In fact the relations need to be explored in a permutational fashion in order to provide a thorough set of prepositional classes. For example, to construct a list of all the words from a hypernym relation from token a to token b, all hypernyms H_1 of a that are synonyms of b are collected, as well as all hyponyms H_2 of b that are synonyms of a. H_1 and H_2 are then combined. The resulting set, $H_{1,2}$, is the list of all words based on a and b that share a hypernym/hyponym relation.

5.2.5 Extended Implementation of Sibling Classes

The definition of sibling prepositional classes in [Harabagiu, 1996] was that of classes formed by tuples with a common hypernym, or classes formed by tuples with a common hyponym. This classification ignores classes which may be formed by pairs of tuples where the hypernym of one tuple is the hyponym of the other one. One reason for ignoring those cases is to maintain a closer semantic relationship among the phrase heads that are thus paired. If one were to plot the synonymy and hypernymy hierarchy in WordNet of words occurring in two tuples, then the graph could be divided into tiers or hierarchy levels. The tuples occurring in each level would have a similar degree of specificity. For example, both *hawk* and *vulture* have *bird of prey* as their hypernym, which in turn has *bird* as its hypernym. *Hawk* and *vulture* have the same degree of specificity in this hierarchy, which is higher than that of either *bird of prey* and *bird*. Harabagiu [1996] has reduced the semantic distance between two tuples by minimizing the difference in WordNet hierarchy tiers. The precision of prepositional classes is thus improved by limiting the set of prepositional classes to those that share common levels within the WordNet hierarchy.

The problem with that approach is that, while the semantic distance between tuples remains the same in regard to the number of edges necessary to construct a class, there is no conclusive method of weighing the edges themselves. Furthermore, it is improper to assume that semantic distances are a proper measure of similarity between tuples [Harabagiu, 2000]. In order to determine whether it would be more beneficial to our algorithm to utilize a comprehensive set of sibling classes or to limit them to classes that strictly share a common hypernym or a common hyponym, we implemented a version of each system. It was concluded from the results of these systems that the gain in recall from performing comprehensive sibling class construction outweighs the potential performance

gain of the limited system. As a result, all of the sibling prepositional classes used in our algorithm are of the comprehensive kind. This decision has an effect on nomenclature in use: Because the classes are no longer limited to sharing a hypernym/hyponym exclusively in one direction in the WordNet hierarchy, reporting them as being siblings is no longer a proper description for them. To prevent confusion by changing the terms utilized, we will continue to utilize the legacy nomenclature and refer to these classes as sibling classes.

5.3 Performance Issues

In order to build prepositional classes from the extracted information, each tuple has to be matched against every other tuple. Because there are n tuples and $n-1$ tuples to be matched against, the class building algorithm runs in $O(n^2)$ time. We applied various optimizations and pruning in order to improve the performance of the algorithm. This section discusses each of those changes.

5.3.1 Smaller Batches

An improvement that lends itself to a potentially much higher performance gain is to split the work into smaller sections. When building classes we can be certain that there is no need to compare tuples with unmatched prepositions. We can thus split the input set by clustering the tuples according to their preposition and processing each batch separately. While this process adds little performance improvement in a single-threaded program over a full comparison between tuples[2], a distributed version can benefit from this modification by running the tuple matches for different prepositions in separate threads. The performance thus gained depends on the kind of system that the parser is run on

[2]Even though we cut down significantly on the number of comparisons performed, $O((n/k)^2 * k)$ still resolves to $O(n^2)$. The operations eliminated were also simple comparisons, which are fast instructions.

and how the system is implemented. An implementation that utilizes marshaling to share large batches of information between processes is better suited for a system with remote machines, whereas a version that uses pipes as a form of inter-process communication (IPC) performs better on a multiprocessor machine or local computer clusters with low latency and high bandwidth networking. A rudimentary implementation of this concept was able to reduce the running time of our parser by about 40%[3] by parsing adjectival and adverbial attachments in separate threads. Further changes were not implemented because they would have required a large-scale restructuring of the program.

5.3.2 Elimination of Duplicate Comparisons

One easy way to gain a little performance is by eliminating unnecessary object comparisons. Finding all pairs of tuples that form a prepositional class can be cast as attempting

```
Data: tuple set
Result: All tuples have been matched against all other tuples in order to build
        prepositional classes.
classes = [ ];
foreach tuple in set do
    foreach otherTuple in set do
        /* matches() returns an array containing the classes formed by
           a match, or nil if there was no match                        */
        tmp = tuple.matches(otherTuple);
        if tmp then
            /* where << is the concatenation operator.                  */
            classes << tmp;
        end
    end
end
```

Algorithm 5.1: Simple class building algorithm

[3]This figure does not include further performance increases due to system differences such as hard drive and physical memory latency and the total amount of physical memory available. The latter have helped reduce the parser's running time by an additional 20%.

to match each tuple against all other tuples. An easy way to perform this is by placing all tuples into an array and matching each of them against a clone of that array, as shown in Algorithm 5.1. If there are n tuples in that array, then the number of computations performed by this algorithm is n^2. However, because we are dealing with combinations, not permutations, the number of computations performed can be reduced. Tuples do not need to be matched against themselves, and do not need to be matched again against tuples occurring before them in the array. A modification of Algorithm 5.1 yields better results at $n*(n-1)/2$ tuple comparison. This was achieved by eliminating elements from our data structures after the outer loop of the algorithm finished using them. Elim-

```
Data: tuple set
Result: All tuples have been matched against all other tuples in order to build
        prepositional classes.
classes = [ ];
repeat
    /* shift() removes the first element from set and returns it     */
    tuple = set.shift();
    foreach otherTuple in set do
        /* matches() returns an array containing the classes formed by
           a match, or nil if there was no match                     */
        tmp = tuple.matches(otherTuple);
        /* nil?() returns true is tmp is nil, false if it is not     */
        if tmp.nil?() then
            /* where << is the concatenation operator.               */
            classes << tmp;
        end
    end
until set.length() == 0 ;
```

Algorithm 5.2: Improved class building algorithm

ination of used elements can be done by either using range counters and incrementing them after each iteration of the outer loop, or by utilizing a dynamic data structure such as a queue or stack. There is also an additional benefit to the latter implementation in that it reduces the memory footprint of the system. The algorithm itself still remains at

$O(n^2)$, but the smaller improvements add up to a considerable performance increase. In our system the data structure version of this enhancement has been implemented. The modifications are shown in Algorithm 5.2. The array objects in Ruby are subclasses of collections and as such have inherent methods that let them act like a stack, queue, or linked list, thus requiring only trivial changes for this improvement to be implemented. These changes come at the cost of increased system overhead due to garbage collection, but the benefits still vastly outweigh this small performance loss.

5.3.3 Reducing I/O Bottlenecks

Performing synset lookup in advance for all tuples significantly reduces the time taken for each iteration, since synset lookup is bound to a BerkleyDB database and thus I/O-bound. I/O operations are expensive, and if enough physical memory is available then loading the necessary information into it beforehand signifies a considerable improvement in running time. In our algorithm we perform all the I/O-bound operations in advance and store the results. This process triples the amount of physical memory used, but has helped reduce the running time from 60+ minutes to an average of 5 minutes for the synonym class building algorithm alone.

Despite the performance gains achieved by loading everything into physical memory we cannot ignore the possibility of the process using up all the system memory. This did in fact occur when we built the hypernym/hyponym classes and sibling classes in our algorithm. It required careful planning and frequent marshaling of intermediate results to minimize memory usage to a point where the system did not suffer a hit in performance because of memory page faults.

5.4 Chapter Summary

In this Chapter we have covered the use of the tuples extracted in the previous Chapter to build prepositional classes. A prepositional class is formed when all the tokens of two tuples share a relationship in WordNet. Three levels of relationship were explored: synonym, hypernym/hyponym, and sibling (tuples with a common hypernym/hypony).

Chapter 6

Prepositional Classes as Disambiguators

In Chapter 5 the nuances of building a knowledge base were covered. This chapter takes the work performed there and presents the results of its implementation and application to the test set of tuples reserved for this purpose in section 4.2.

6.1 The Test Corpus

The set of tuples initially set aside for testing purposes was composed of 5906 tuples of adjectival/adverbial attachment type. The correct attachment is provided by these tuples, but our application will ignore that information except for the measurement of the precision of the attachment decision algorithm.

6.2 Alternate Algorithms

In order to increase the thoroughness of our examination of the attachment decision algorithms we decided to implement some simple alternatives to our disambiguation algorithm. These implementations serve a twofold purpose - they act as a predictable point of reference for the measurement of our algorithm, and provide inexpensive fallback algorithms which can be used to boost the recall figures of our own algorithm.

6.2.1 Random Attachment

This algorithm was implemented to provide an absolute lower bound to our tests. It performs at predictable 50% average precision.

6.2.2 Right Attachment

The right attachment algorithm assumes that all prepositional phrases are attached to their closest part of speech, which in this case is the noun phrase. In other words, it is an algorithm that always decides on adjectival attachment. As mentioned in section 1.3, [Kimball, 1973] predicts an average of 60 percent precision with this type of attachment decision algorithm. On our tuple set, the results precision for right attachment precision were set at 63.021%, which falls within the acceptable range for us to confirm the results in [Kimball, 1973]. This attachment decision algorithm performs with the same results upon each iteration, thus being more reliable than the random attachment algorithm. It also performs better on a constant basis. It is because of this that we decided to use it as our fallback algorithm.

6.3 Disambiguation Algorithm Implementations

In order to depict the evolution of our refined knowledge base we took snapshots of the different stages it went through during its implementation. All of these stages demonstrate the changes in precision and recall produced by the addition of consecutively more thorough tuple matching algorithms. In this section we will step through the knowledge bases as viewed by the disambiguation algorithm, and the steps that this disambiguation algorithm went through in order to disambiguate unknown tuples.

6.3.1 Internal Sorting of Prepositional Classes

All prepositional classes were sorted by their preposition, and independent of each other. This was done in response to the reasoning in Section 5.3.1, wherein we argued that performing tuple matching across dissimilar prepositions adversely affects the precision of prepositional classes in a manner that outweighs the gains of having a higher recall rate.

6.3.2 External Sorting of Prepositional Classes

A second commonality between knowledge base versions is how they are stored. Each stage had its classes sorted by their degree of separation in WordNet. Synonym, hypernym/hyponym, and sibling classes each were collected as independent sets, and each set is in turn split between tuples derived from right (adjectival) attachment and those derived from left (adverbial) attachment. The file size of these collections growed predictably in an exponential form. It is possible to compress the file size of these sets by removing duplicates, but we decided not to do so because the number of identical classes for a particular tuple may provide our algorithm with the means necessary to disambiguate tuples for which we have results as both adjectival and adverbial fields. While the number of am-

biguous tuples present is very small, the application of class frequency to prepositional attachment disambiguation has shown some improvements in certain cases at minimal costs.

6.3.3 Tuple Normalization

Prior to any application of constraints to the test set, we applied normalization to the tuples in our it in a fashion similar to the normalization performed in Section 5.1.3 when constructing the prepositional classes. WordNet was again used to construct the base forms of each term, which were collected and used at the time of constraint application instead of the inflected form.

6.3.4 Priority of Constraint Selection

The various implementations of attachment decision algorithms described in this chapter are all applications of constraint satisfaction. Each set of classes - synonyms, hypernyms/hyponyms, siblings - is viewed as a constraint on the input, which are the ambiguous tuples from the test set. Synonyms are the tightest constraint, followed by hypernyms/hyponyms, followed by sibling classes. This order is imposed by the maximum degree of separation between tuples that were used to construct the classes. This degree of separation ranges from 0 at synonym classes, to 2 at sibling classes. The degree of separation is set by the minimum number of edges necessary to connect any respecive phrase heads of the tuples in the WordNet hierarchy. Synonym classes are formed with phrase heads that share the same WordNet synsets and therefore do not need any edges to connect. Hypernym/hyponym classes share at least one set of phrase heads for which one head is the hypernym/hyponym of the other, resulting in one edge being needed. Sibling classes contain phrase heads which share a common hypernym/hyponym (but are not synonyms), resulting in a minimum of two edges to jump from one to another. Our

algorithm will settle on the first constraint that is matched exclusively. By exclusively matched constraints we refer to those which are matched either for the adjectival attachment set or the adverbial attachment set, but not both at once. If a tuple is matched by both sets it is passed on to the next level of constraints, just as if it had remained unmatched. The implementation of this system is as follows:

- If the target tuple is featured in either the adjectival synonym class or the adverbial synonym class - but not both - then the class that it is matched to is considered to be the correct attachment. The algorithm would stop at this point because the attachment is disambiguated.
- A tuple that is matched by both adjectival and adverbial constraints offers two choices to our algorithm, both of which are explored in our work:
  - The simple solution is to delegate the disambiguation to the next level of constraint. This method is more time efficient, at the cost of a small amount of precision loss.
  - The alternative is to utilize a count of the number of constraints matched in each set as collected during the external sorting of prepositional classes in order to weigh the decision towards either the adjectival set or the adverbial set.
- If the tuple is not found in either the adjectival set or the adverbial set then that signifies that it didn't properly match any of the synonym attachment constraints (which are respresented as the classes) and the decision is delegated to the hypernym/hyponym classes.
- The constraint matching is repeated for hypernym/hyponym classes, a failure in matching the constraints resulting in a further delegation down the list to sibling

classes.

- Sibling classes are the last constraints to be considered in the knowledge base.
- Tuples that match both or none hypernym/hyponym class attachments are collected as the set of tuples that could not be disambiguated. These tuples affect the recall of the algorithm.

6.4 Chapter Summary

This chapter describes how to apply the prepositional classes constructed during the previous chapter's work to the disambiguation of prepositional phrase attachments. The task was cast as a constraint satisfaction problem, with the classes acting as the constraints. Satisfaction is produced when a tuple solely matches either right attachment classes or left attachment classes. Synonym classes are tested first, followed by hypernym/hyponym classes. Sibling classes are tested last.

Chapter 7

Constraint Result Assessment

We recorded various numbers at different key stages of our algorithm. These results are shown in Table 7. Pictured are the levels of constraints used for a particular snapshot, the precision and recall of our algorithm, the precision of our algorithm when using right attachment as a fallback algorithm (bringing the recall up to 100%), and the precision of the fallback algorithm on the tuples that were matched by our algorithm. We have sorted the results by their degree of complexity.

We have split the table into three sections, each one containing the results for the versions of the algorithm using the various degrees of complexity by which we performed the external sorting of prepositional classes. We have also highlighted the features of tuples that were most desireable (in **bold**) and least desirable (in ***bold italics***). Examination of the numbers produces allows us to draw some conclusions regarding the effectiveness of the algorithm and the rate at which the the statistics change.

Constraints	Precision	Recall	Precision with Fallback	Right attachment precision
S	**90.799**	***6.976***	***64.900***	63.835
Sa	**90.799**	6.993	64.917	63.680
S,H	86.8055	14.629	66.2885	64.468
Sa,H	86.821	14.646	66.305	64.393
Sa,Ha	86.508	14.934	66.2885	***64.626***
S,H,Z	78.924	30.850	68.5235	61.087
Sa,H,Z	78.936	30.867	68.540	61.053
Sa,Ha,Z	79.105	31.036	**68.626**	**61.0475**
Sa,Ha,Za	***77.859***	**32.272**	68.371	61.280

Legend: S = Synonym, H = Hypernym/Hyponym, Z = Sibling,
Xa = Constraint X with additional frequency-based ambiguity resolution.

Table 7.1: Constraint results.

7.1 Complexity versus Precision

The first pattern that can be examined is the relation between the degrees of highest semantic distance that the prepositional classes were constructed with and the precision of our disambiguation algorithm. The second column of the table demonstrates that higher semantic distance results in lower precision. On the first change in level of constraints a drop in precision of about 4% can be observed, and an 8% drop on the second level change. There is no sufficient information available to conclusively determine that this trend will continue as more lax constraints are added, but it is most likely that another level of constraints will confirm a linear drop in precision in relation to complexity. What makes this pattern more interesting is the fact that the number of total constraints grows exponentially as more levels are added, making the precision drop at a lower rate than complexity.

7.2 Complexity versus Recall

After observing the rate at which recall increases with respect to complexity we can conclude that the rate of recall doubles with every level of constraints added. This points to an exponential growth rate of recall within the system. There is reason, however, to believe that this rate cannot be sustained. Recall growth will eventually flatten as the system approaches the limit of WordNet's knowledge base. The maximum recall possible without any additional word databases would be the percentage of tuples that do not contain words unknown to WordNet. This is estimated to be around 75% based on the results from the normalization process in Chapter 5.

7.3 Recall versus Precision

Since we have observed that there is a relation between complexity and recall as well as a relation between complexity and precision, it is now possible to draw a relation between recall and precision. These two values grow in opposite directions. It follows that maximizing one of them will minimize the other, and vice-versa. A naive observation to be concluded from this would be that there is one point where the combination between precision and recall is balanced and has the best yield in efficiency. This assumption ignores a key figure in the table, namely the precision of the right attachment algorithm on the same set of tuples as those that were matched by our algorithm. This number affects the precision of our algorithm when combined with right attachment. The lower this value is, the higher the effect of our algorithm is on the overall precision. It is possible that the combination of these *three* curves will yield more than one maximum. The present data, however, is insufficient to determine where such points will be.

7.4 Effect of Constraint Counts

The last important observation to be drawn from Table 7 is regarding the effect of utilizing constraint frequency to disambiguate those cases where we find a solution in both the adjectival and adverbial prepositional class sets. We have implemented this use of constraint counts in an incremental fashion. Three important statements can be made about this exercise:

- Using constraint counts does not cause the algorithm to be any worse off than not using constraint counts. None of the cases seen had a worse joint precision with the fallback algorithm than if the counts had been ignored.

- Constraint counts have a very small effect when applied only to synonym (zero degree WordNet separation) constraints. This can be rationalized as being caused by the efficiency with which synonym prepositional classes work, leaving no ambiguous classes to be resolved, and by the relatively small number of synonym classes in our knowledge base.

- Using counts on second degree constraints degrades the performance over using constraint counts on only up to first degree prepositional classes. A possible explanation for this effect is that the precision of hypernym prepositional classes is sufficiently large to have a greater impact on the system than the more extensive set of second degree prepositional classes.

7.5 Chapter Summary

The results from the base algorithm were analyzed in this Chapter. Four kinds of statistics were collected. These are precision of the standalone base algorithm, recall of the

standalone base algorithm, precision of the base algorithm with a right attachment fallback system, and precision of the right attachment system on the tuples matched by the base algorithm.

The statistics described above were collected while running this system with different variations of our knowledge base. Results were gathered on a corpus of only synonymy-based classes, synonymy and hypernymy/hyponymy based classes, and finally the full set of synonymy, hypernymy/hyponymy, and sibling classes. In addition, each variation was tested with incremental class count disambiguation, by which additional disambiguation is performed through selecting attachment based on the number of results.

The best results were achieved with a complete corpus and class count disambiguation up to the hypernym/hyponym level with 68.626% precision using the right attachment fallback algorithm. This indicates that prepositional class frequencies at the sibling level are too distorted to be effective in a disambiguation effort.

Chapter 8

Alterations to the Algorithm

In order to further improve the recall rate of our algorithm some changes to our disambiguation algorithm have been implemented.

8.1 Client-side Word Sense Expansion

In our base algorithm we normalized the words and used their base form for the tuple matching algorithm. While this was a step in the right direction, our algorithm could do better. We utilized the module that was created for the construction of our knowledge base to collect the synonym, hypernym, and hyponym information for the tuples in our test set. By doing this, a novel client-side use for the prepositional class creation algorithm presented by [Harabagiu, 1996] has been implemented. As was done with the reconstruction of the original algorithm, the statistics at different stages of this new implementation were recorded. The results of this collection can be seen in Tables 8.1 and 8.2. Table 8.1 contains results for algorithms using only zero and one degree constraints, whereas Table 8.2 shows results for the complete set of constraints. For convenience, we have included a copy of the base algorithm that produced the best results so that the

Constraints	Precision	Recall	Precision with Fallback	Right attachment precision
Sa,H	**86.821**	***14.646***	66.305	***64.393***
S,H + C1	80.706	20.623	**66.949**	**61.6585**
Sa,H + C1	80.722	20.640	**66.949**	61.690
Sa,Ha + C1	80.238	21.334	66.915	61.984
S,H + C2	71.296	36.454	65.7975	63.679
Sa,H + C2	71.309	36.472	65.7975	63.695
Sa,Ha + C2	***71.099***	36.675	65.713	63.758
S,H + C3	71.247	43.989	***66.153***	64.126
Sa,H + C3	71.286	43.989	66.170	64.125
Sa,Ha + C3	71.451	**44.06**1	66.238	64.140

Legend: S = Synonym, H = Hypernym/Hyponym, Z = Sibling, Xa = Constraint X with additional frequency-based ambiguity resolution, Cn = Client-side class expansion.

Table 8.1: Constraint results - Two classes.

new statistics can be compared and contrasted to it. Three stages of the client-side algorithm were focused on to collect these results: Usage of synonymy information alone, uncombined use of synonymy, hypernymy, and hyponymy information, and fully combined use of the three kinds of information. These stages have been labeled as *C1*, *C2*, and *C3* respectively. All changes were applied on a system that performs all three levels of constraint lookup. As was done with the base system, we also recorded the effects of prepositional class counts on the stages of the algorithm in an incremental manner.

8.1.1 Client-Side - Synonymy

For this system we collected only synonym information from each tuple in the test set. Each phrase head has a number of synsets that it matched to. For each of these synsets there is a word collection that lists all of the terms that match this synset. In other words, a list of synonyms. These sum of all these lists constitutes the list of words that share a semantic meaning with the phrase head. To apply a matching constraints the

Constraints	Precision	Recall	Precision with Fallback	Right attachment precision
Sa,Ha,Z	**79.105**	***31.036***	***68.626***	61.0475
S,H,Z + C1	74.261	46.969	**70.369**	59.616
Sa,H,Z + C1	74.270	46.986	**70.369**	**58.631**
Sa,Ha,Z + C1	74.385	47.528	70.488	58.675
Sa,Ha,Za + C1	71.525	52.624	69.150	59.878
S,H,Z + C2	69.939	69.336	69.0315	61.270
Sa,H,Z + C2	69.963	69.336	69.048	61.270
Sa,Ha,Z + C2	69.7975	69.404	68.930	61.283
Sa,Ha,Za + C2	69.480	70.0135	68.727	61.330
S,H,Z + C3	69.827	75.703	69.082	61.821
Sa,H,Z + C3	69.850	75:703	69.100	61.821
Sa,Ha,Z + C3	69.940	75.703	69.167	61.821
Sa,Ha,Za + C3	69.983	**76.329**	69.981	***61.890***

Legend: S = Synonym, H = Hypernym/Hyponym, Z = Sibling, Xa = Constraint X with additional frequency-based ambiguity resolution, Cn = Client-side class expansion.

Table 8.2: Constraint results - Three classes.

problem was cast as finding a non-empty set at the intersection of this collected word list and a word list from a prepositional class.

8.1.2 Client-Side - Uncombined Range

The algorithm labeled as stage two (*C2*) adds hypernym and hyponym word lists to the synonym sets in the previous stage. These three lists, however, are not mixed. This means that the system performs searches where all phrase head lists are from one and only one of the three lists. The lists are all either synonym lists, hypernym lists, or hyponym lists.

8.1.3 Client-side - Combined Information

The last stage combines synonym, hypernym, and hyponym word lists to explore all possible variations of these sets. The challenge in this case has been to determine which

```
Data: tuple targets.
Result: tokens in tuples are replaced with 3 arrays, each containing a list of
        synonyms, hypernyms/hyponyms, and siblings respectively.
foreach tuple in targets do
    /* collect synonyms, hypernym/hyponyms, and siblings.  Each set is
       wrapped in its own array [ ] .                                    */
    foreach token in tuple do
        /* where << is the concatenation operator.                      */
        token = [ token.synonyms() ] << [ token.hypernyms() <<
        token.hyponyms() ] << [ token.siblings()] ] ;
    end
end
```

Algorithm 8.1: Uncombined-range word expansion

```
Data: tuple targets.
Result: tokens in tuples are replaced with a list containing their synonyms,
        hypernyms/hyponyms, and siblings.
foreach tuple in targets do
    /* collect synonyms, hypernym/hyponyms, and siblings.  Unlike
       Algorithm 8.1, the lists are not wrapped in their own array and
       thus merged.                                                      */
    foreach token in tuple do
        /* where << is the concatenation operator.                      */
        token = token.synonyms() << token.hypernyms() << token.hyponyms()
        << token.siblings();
    end
end
```

Algorithm 8.2: Same-priority word expansion

Rank	Feature	Short form
1st	All synonyms	SSS
2nd	Two synonyms, one hypernym/hyponym	SSH
3rd	Two synonyms, one sibling	SSZ
4th	One synonym, two hypernyms/hyponyms	SHH
5th	One synonym, one hypernym/hyponym, one sibling	SHZ
6th	One synonym, two siblings	SZZ
7th	Three hypernyms/hyponyms	HHH
8th	Two hypernyms/hyponyms, one sibling	HHZ
9th	One hypernym/hyponym, two siblings	HZZ
10th	Three siblings	ZZZ

Table 8.3: Priority of client-side combined constraints.

combination is given priority. One option considered was to merge all three sets into one larger list. Algorithm 8.2 describes a possible approach to merging all three word lists. The advantage of this is that all words receive the same priority, removing chances of improper prioritization. The efficiency of the algorithm is improved as well because there is no need for multiple searches. The disadvantage of this system is that all words receive the same priority. Synonym sets should receive higher priority than hypernym/hyponym sets because of their improved precision. It is for this reason that our system explores all combinations separately.

The challenge was to determine the order in which the combinations should be approached. The final decision was to favor combinations with synonym sets over combinations with hypernym/hyponym sets, and combinations with hypernym/hyponym sets over combinations with sibling sets. The relations were made transitive, meaning that synonym set combinations were favored over sibling set combinations as well. The implementation follows a switched approach. Stronger relations are tested first, and failure to find a match leads to the next strongest relation. Because the prepositions in the 4-tuples do not change, only 3 tokens need to be matched against the prepositional classes. We know that each of these tokens has 3 possible sets from which to draw a match from -

the synonym set, the hypernym/hyponym set, and the sibling set. No token has priority over any of the other ones. Therefore, ranking of matches is performed on a collective basis. Table 8.3 illustrates the ranks used in our application.

8.2 Assessment of Results

The changes described in this section have produced interesting statistics. Some of them reinforce the conclusions that were drawn during the analysis of the base system, while others contradict trends that were observed previously.

8.2.1 Precision versus Recall

The most noticeable event that was observed is that the increase in recall rates has begun to slow down. This effect had already been predicted in the previous chapter. As the parser approaches the limit of tuples whose phrase heads can be identified by WordNet, the curve will flatten until it hits that hard limit. In conjunction with the observed increase in recall rates, the decrease in precision has nearly stalled. From these observations it is possible to conclude that as the knowledge base reached its saturation point, precision and recall become disjoint from the size of the knowledge base. More importantly, we learned that it would be wasteful to extend the knowledge base of the base system beyond second degree prepositional class relations.

8.2.2 Effect of Client-Side Class Extension

A surprising result from both tables of statistics is the effect of client-side class expansions on the test tuple set. Precision of the algorithm with fallback has in fact reached its peak with *C1* class expansions. This can be explained by comparing algorithm precision with right attachment precision on the same set. By increasing the recall rate of the algorithm

its precision is affected. It is logical to expect less precise algorithms to perform closer to right attachment after the algorithm's peak performance has been reached.

One exception that can be observed is the *C3* class expansion with full class count-based disambiguation. The large difference in recall over its predecessors is enough to offset the reduced precision.

8.2.3 Effect of Class Count

Unlike the trend observed in the base system, our modifications have altered the effect of prepositional class frequencies on precision and recall. Whereas the base system benefited noticeably from it (specially on first degree prepositional classes), our modified algorithm's precision and recall rates brought forth mixed results from its application. In both tables we can observe that class count has a detrimental effect on *C1* and *C2* algorithms. Opposing this trend, the *C3* modified algorithm sees a steady rise in performance with each additional level of count-based class disambiguation.

What can be concluded from these observed trends is that class count-based disambiguation has a positive effect on algorithms that rely more heavily on recall, while precision-focused algorithms will see detrimental effects on their performance if class count is applied to them.

8.2.4 The "Better" Algorithm

Deciding whether one algorithm is superior to another one is a subjective matter. While overall precision is possibly the best indicator of improvement, other factors also need to be considered.

The modified system with *C1* alterations is clearly ahead in general efficiency when combined with right attachment fallback. It features the highest precision when no class count disambiguation is applied to it, and has the second shortest running time (after the

base system) due to reduced complexity. *C1* performs well on both two- and three-degree constraint sets. On the other hand, the *C3* altered algorithm boasts the highest recall rate (which can be further boosted with class count disambiguation), to a point where it can be applied without the need for a fallback algorithm on sets utilizing three constraint levels. Not needing a fallback algorithm is an important milestone, and the ability to be able to perform well without any additional assistance should not be ignored. The *C2* hybrid algorithm, which attempts to maintain a compromise between higher precision and higher recall, has demonstrated the least favorable results.

Making a choice as to which algorithm is the better selection for a general purpose algorithm, we favor the *C1* modified algorithm due to its steady performance, and leave the *C3* algorithm for domain-specific applications.

8.3 Chapter Summary

This chapter covered client-side word sense expansion, an additional change that we have implemented on the system to improve its performance. The change involves the use of WordNet to extend the test set tuples with the synsets of their phrase heads.

Client-side word sense expansion were tested at three levels. For the first level, tokens were expanded to the set of their synonyms. Second, tokens were expanded to the set of their synonyms, hypernyms, and hyponyms, and looked up on each set separately. The last phrase head expansion was performed on fully combined synonym, hypernym, and hyponym sets. Both the second and the third extended systems give preference to synonym sets. Hypernym/hyponym sets are considered to be equivalent in priority.

Results for each level of client-side class expansion were recorded with two knowledge bases. The first one is composed of synonym- and hypernym/hyponym-based prepositional classes. The second knowledge base uses the complete set of prepositional classes.

Both corpora were also tested with incremental levels of class count disambiguation.

Two systems using the complete set of prepositional classes were singled out for their superior performance. Synonym-based client-side expansion achieved the highest precision with right attachment fallback at 70.369%, with no class count disambiguation or count-based disambiguation only at the synonym level. Full client-side and count-based disambiguation performed next best, at 69.983% standalone precision and 76.329% standalone recall, and 69.981% precision with right attachment fallback.

Chapter 9

Conclusions and Future Work

9.1 Summary and Research Method

In this paper we have presented and implemented a class-based algorithm for the disambiguation of prepositional phrase attachments, and enhancements to this algorithm. The base algorithm featured here is a recreation of the one described by [Harabagiu, 1996].

At the beginning of this paper the problem of prepositional phrase attachment disambiguation was identified. This was followed by a discussion on techniques and tools that aid in the process of disambiguation of prepositional phrase attachments. After that, we appraised and discussed the current methods of attachment disambiguation, and briefly discussed the reasons for choosing a class-based disambiguation algorithm over more conventional methods such as statistical or rule-based algorithms.

In Chapter 4, we described how the pre-tagged text in the Penn Treebank corpus is parsed to extract the needed tuples of the form [$Verb$, $Noun_1$, Prep, $Noun_2$]. First the corpus was split into two sets, a training set and a testing set. Splitting was performed by random selection. The training set was composed of 90% of the total amount of tuples of the corpus, while the rest of the tuples constituted the test set. The approach taken

to parsing the corpus was to translate the tagged text into tree-like data structures and to explore different traversal strategies to find the best fit strategy. It was decided that a postorder traversal form brought forth the best results because it was capable of building a parse stack and analyzing that stack in a single pass. The building stage was the traversal down towards the leaves, while the analysis stage was the return. The tokens of each tuple were kept tightly clustered by pruning used tokens during the analysis. Pruning prevented these tokens from occurring multiple times in tuples. Chapter 6 described how to apply the prepositional classes constructed during the previous chapter's work to the disambiguation of prepositional phrase attachments. The task was cast as a constraint satisfaction problem, with the classes acting as the constraints. Satisfaction was produced when a tuple solely matches either right attachment classes or left attachment classes. Synonym classes were tested first, followed by hypernym/hyponym classes. Sibling classes were tested last.

The last section of the thesis explored the results of the implemented algorithm. These were analyzed in Chapter 7. Four kinds of statistics were collected. These were precision of the standalone base algorithm, recall of the standalone base algorithm, precision of the base algorithm with a right attachment fallback system, and precision of the right attachment system on the tuples matched by the base algorithm. The described statistics were collected while running the system with different variations of our knowledge base. Results were gathered on a corpus of only synonymy-based classes, synonymy and hypernymy/hyponymy based classes, and finally the full set of synonymy, hypernymy/hyponymy, and sibling classes. In addition, each variation was tested with incremental class count disambiguation, by which additional disambiguation is performed through selecting attachment based on the number of results. The best results were achieved with a complete corpus and class count disambiguation up to the hypernym/hyponym level with **68.626%** precision using the right attachment fallback

algorithm.

The last chapter covered client-side word sense expansion, an additional change that we have implemented on the system to improve its performance. The change involved the use of WordNet to extend the test set tuples with the synsets of their phrase heads. Client-side word sense expansion was tested at three levels. For the first level, tokens were expanded to the set of their synonyms. Second, tokens were expanded to the set of their synonyms. hypernyms, and hyponyms and looked up on each set separately. Lastly, phrase head expansion was performed on fully combined synonym, hypernym, and hyponym sets. Both the second and the third extended systems gave preference to synonym sets. Hypernym/hyponym sets were considered to be equivalent in priority. Results for each level of client-side class expansion were recorded with two knowledge bases. The first one is composed of synonym- and hypernym/hyponym-based prepositional classes. The second knowledge base used the complete set of prepositional classes. Both corpora were also tested with incremental levels of class count disambiguation. Two system using the complete set of prepositional classes were singled out for their superior performance. Synonym-based client-side expansion achieved the highest precision with right attachment fallback at **70.369%**, with no class count disambiguation or count-based disambiguation only at the synonym level. Full client-side and count-based disambiguation performed next best, at **69.983%** standalone precision and **76.329%** standalone recall, and **69.981%** precision with right attachment fallback.

9.2 Limitations

The presented system performs on-line lookup of prepositional classes. This means that its running performance is affected both in terms of resources used and execution time. Optimizations have not been included in order to preserve flexibility in implementation.

These optimizations can be rolled in once the algorithm performs with sufficient precision and recall.

In order to reduce the amount of resources utilized by the application, sense disambiguation of the tuple phrase heads is not performed, and relevant sense information is discarded. Better encoding of the information and elimination of non-critical debugging information should be sufficient to free up enough resources to reintroduce semantic knowledge for each synset.

Even though the presented algorithm requires a considerably smaller training set than statistical systems, it still relies on a pre-tagged source. Such texts are difficult to find for domain-specific applications.

There is a limit as to how many of the tuples can be disambiguated, which depends on the ratio of tokens that can be found in the WordNet database to those which cannot. Estimates based on results from Chapter 5 place this limit at about 75% of the total set of tokens.

9.3 Future Work

Further improvements could be applied by creating additional semantic classes from the directly extracted classes. Synonym class clusters themselves can be combined to form additional first degree (hypernym/hyponym) and second degree (sibling) classes. This approach would be an extension of the original class-building algorithm. The advantage of utilizing synonym class clusters to build additional disambiguation classes is that this process is more exhaustive and efficient than performing individual hypernym/hyponym/sibling class lookup between 4-tuples.

The knowledge-base building algorithm can be enhanced with the aid of dictionaries, encyclopedias, and gazetteer databases in order to reduce the number of unmatched

phrase heads in WordNet and thus improve the system's recall rate. The use of proper nouns changes faster than any system - be it knowledge-based or stochastical - can keep up with. Disambiguation of tuples containing such terms has to be performed at the last moment. By utilizing a frequently-updated database of proper nouns, the client-side token expansion algorithm can be modified to attempt to perform a live match of any unmatched target token to its most likely synonym.

The presented system does not give priority to any of the tokens in the target 4-tuples. The current method utilized to disambiguate a tuple when it matches prepositional classes for both left and right attachment at a particular level is a plain frequency-based system. When comparing the frequencies for right and left attachment, however, the system ignores the fact that right attachment is more likely to occur in first place. While we have demonstrated that the use class frequencies can have a positive effect on the base system, our extended system has not benefited from it except in the case of comprehensive client-side and server-side disambiguation (see Table 8.2). By skewing the priority of tokens within the 4-tuples the positive effects of frequency-based disambiguation can be boosted. For example the match

$$[H, S, Prep, S]$$

in which the head of the noun phrase is matched against a synonym, and the head of the verb phrase is matched against a hypernym/hyponym, may be preferred over

$$[S, H, Prep, S]$$

where the matches are inverse. The reason for this particular preference is the fact that right attachment is more likely to occur. The effect of such prioritization needs to be documented.

The problem of prepositional phrase attachment disambiguation can be extended to cases which feature multiple prepositional phrases, such as

He saw the man in the park with the telescope

The decision making process is complicated in this case by the availability of two prepositional phrases instead of one. One possible approach to solving this problem is to alter the constraint selection algorithm to process both prepositional phrase attachments at the same time. Another possible approach would be to treat each phrase as an independent attachment, and handle them separately.

Class-based disambiguation is not restricted to prepositional phrase attachments only. The same principles can be applied to other forms of binary attachment disambiguation, such as object attachment in

The fishbone was stuck in her throat, and it *was swollen.*

and

Her throat had a fishbone stuck in it, and it *was swollen.*

In both sentences, *it* refers to the throat. However, the placement of the object that this is attached to has changed.

In the above case, disambiguation can be performed syntactically. By populating a knowledge-base with examples of correct object attachment, this problem can be solved in an analogous manner to prepositional phrase attachment disambiguation.

9.4 Contributions

In this paper we have replicated the system described in [Harabagiu, 1996] and added enhancements to improve its performance. Just like the original system, our algorithm relied on WordNet in order to produce prepositional classes, and was trained and tested on the Penn Treebank corpus. Prepositional classes were formed by combining tuples

whose phrase heads share a close connection within the WordNet hierarchy. This was determined by the synonymy, hypernymy/hyponymy, and sibling relations between the phrase heads.

Unlike the work described in [Harabagiu, 1996], we have chosen to not pursue sense disambiguation of every token in the 4-tuples used, in order to improve recall of the system. This is possible because sense disambiguation of tokens is not a crucial part to the disambiguation process, only a by-product.

We have extended the sibling relations and explored a greater range of tuple combinations. The extension was achieved by allowing tuples to form a prepositional class if the hypernym of one is shared with the hyponym of the other. This is opposed to the original description of the system which only considered tuples which both share a common hypernym or hyponym.

The system that we replicated and extended performs perceptibly better than right attachment (which performs at **63.021%** precision), exceeding its precision by **14.8-27.7%** on matched tuples. When combined with right attachment as a fallback algorithm to achieve 100% recall, the base system performs at a **5.8%** higher precision than right attachment alone.

We have enhanced the disambiguation algorithm by performing client-side synset expansion of tuples. This novel application of class-based lookup provides a significant improvement over the base algorithm of **15.9-35.3%** recall at an affordable cost in precision loss of **4.9-9.2%**. The precision of the new algorithm with a fallback system at 100% recall has been improved by an additional **1.8%** to **70.369%**. Furthermore, the recall rate of the extended system after the implemented changes reached **76.329%**, allowing it to perform on its own without a fallback algorithm.

Both base and extended algorithms have been additionally extended with class count disambiguation. This change has improved the recall rate of the base system by **0-2.2%**,

and the recall rate of the enhanced algorithm by **0.6-5.7%**.

We expect this system to be used as a skeleton model to create newer and better algorithms, both in terms of precision as well as recall. Client-side application of common techniques is also a method that is often overlooked, and we hope to raise the awareness of such usage throughout this paper. Because client-side token expansion is performed on-the-fly, it is advantageous over static prepositional classes in that it can use the latest knowledge repositories to improve its performance.

Glossary

antonymy: The semantic relation that holds between two words that can (in a given context) express opposite meanings, 12

fallback algorithm: An algorithm that can make an attachment decision when the base algorithm cannot, 53

hypernymy: The semantic relation of being superordinate or belonging to a higher rank or class, 11

hyponymy: The semantic relation of being subordinate or belonging to a lower rank or class, 12

marshaling: See serialization, 49

meronymy: The semantic relation that holds between a part and the whole, 11

pipes: In UNIX programming, a method of channeling the input and output of programs, 49

polysemy: The ambiguity of an individual word or phrase that can be used (in different contexts) to express two or more different meanings, 38

serialization: In programming, the process of storing objects somewhere to reconstitute them later, 49

synset: A set of one or more synonyms, 11

syntagma: A syntactic string of words that forms a part of some larger syntactic unit, 24

Bibliography

E. Brill and P. Resnik. A rule-based approach to prepositional phrase attachment disambiguation. In *15th International Conference on Computational Linguistics (COLING94)*, Kyoto, Japan, 1994.

Eric S. Brill. A simple rule-based part of speech tagger. In *Proceedings of the third conference on applied natural language processing*, 1992.

Eugene Charniak, Curtis Hendrickson, Neil Jacobson, and Mike Perkowitz. Equations for part-of-speech tagging. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*. AAAI Press/MIT Press, 1993.

K. W. Church. A stochastic parts program and noun phrase parser for unrestricted text. In *International Conference on Acoustics, Speech, and Signal Processing*, 1988.

Michael Collins and James Brooks. Prepositional Attachment through a Backed-off Model. In David Yarovsky and Kenneth Church, editors, *Proceedings of the Third Workshop on Very Large Corpora*, pages 27–38, Somerset, New Jersey, 1995. Association for Computational Linguistics.

W. N. Francis and H. Kucera. *Brown Corpus Manual*, 1979. Manual of information to accompany A Standard Corpus of Present-Day Edited American English, for use with Digital Computers.

Sanda M. Harabagiu. An application of WordNet to prepositional attachment. In *34th Annual Meeting of the Association for Computational Linguistics*, 1996.

Sanda M. Harabagiu. Patterns of Prepositional Attachments - Where Dictionary Semantics Meets Corpus Statistics. In *International Journal of Pattern Recognition and Artificial Intelligence*, volume 14, pages 809–838. World Scientific Publishing Company, 2000.

Donald Hindle and Mats Rooth. Structural Ambiguity and Lexical Relations. In *Meeting of the Association for Computational Linguistics*, pages 229–236, 1991.

J. Kimball. Seven principles of surface structure parsing in natural language. In *Cognition*, volume 2. Elsevier Science, 1973.

Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, Massachusetts, 1999. URL `citeseer.ist.psu.edu/635422.html`.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of English: the Penn treebank. *Computational Linguistics*, 19(2): 313–330, 1993. SPECIAL ISSUE: Special issue on using large corpora: II.

Yukihiro Matsumoto. The Ruby Programming Language, 1995. Available at www.ruby-lang.org/en.

George A. Miller. WordNet: a lexical database for English. *Communications of the ACM*, 38(11), 1995.

J. Pustejovsky, J. Castano, J. Zhang, M. Kotecki, and B. Cochran. Robust relational parsing over biomedical literature: Extracting inhibit relations. In *Pacific Symposium on Biocomputing (PSB) 2002*, 2002a.

J. Pustejovsky, J. Castao, R. Saur, A. Rumshisky, J. Zhang, and W. Luo. Medstract: Creating Large-scale Information Servers for Biomedical Libraries. In *ACL 2002 Workshop on Natural Language Processing in the Biomedical Domain.*, Philadelphia, PA., 2002b.

Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2), 1990.

L. Schulmeister. A complication of vascular access device insertion. A case study and review of subsequent legal action. *J Intraven Nurs*, 21(4), 11 1999.

Dave Thomas and Andy Hunt. Ruby in a Day. In *OOPSLA 2002*, number 39, 2002.

Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby, Second Edition*. Pragmatic Bookshelf, 2004.

L. G. Valiant. A theory of the learnable. In *Proceedings of the sixteenth annual ACM symposium on theory of computing*, 1984.