DEVELOPING METHODS FOR WORD SENSE DISAMBIGUATION

by
Önder Eker

Submitted to the Department of Computer Engineering
in partial fulfilment of the requirements
for the degree of
Bachelor of Science
in
Computer Engineering

Boğaziçi University
2007

TABLE OF CONTENTS

Abstract

Project Name : Developing Methods for Word Sense Disambiguation

Project Team : Önder Eker

Supervisor : Asst. Prof. Tunga Güngör

Term : 2006/07 II.Semester

Keywords : word sense disambiguation, decision lists, unsupervised learning, corpus linguistics

Summary

Word sense disambiguation (WSD) is the task of assigning a meaning to an ambiguous word given the context in which it occurs. WSD serves as an intermediate step for many computer science applications such as machine translation, information retrieval, hypertext navigation, content and thematic analysis, speech processing. Therefore, it has been a central problem since the earliest days of computational studies of natural language. WSD algorithms can be classified into three general categories [1]: *(i)* AI-based methods, *(ii)* Knowledge-based methods, *(iii)* Corpus-based methods which gather knowledge from corpora—collection of edited and published texts used for linguistics purposes. *Supervised learning* requires manually disambiguated words, whereas *unsupervised learning* does not. Difficulties involved in hand-tagging training corpora bring about the infamous roadblock known as *knowledge acquisition bottleneck.*

This project draws on 'decision lists' introduced by Rivest [2]. Decision lists are particularly useful for a set of non-independent evidences as in the case of word sense disambiguation and performs comparable to or better than other approaches such as N-gram taggers or Bayesian classifiers [3]. Pseudowords were employed to experiment with both supervised and unsupervised methods. Yarowsky algorithm [14] was implemented and its behavior was analyzed with respect to program parameters. The algorithm proceeds in an iterative bootstrapping fashion in the training phase. It starts with a manually-specified seed collocation for each sense of the target word. At each step, new collocations are discovered and then used to disambiguate other occurrences in the training corpus. The training continues until the convergence condition when no more occurrences can be added into sense classes.

1. LITERATURE SURVEY

1.1. Introduction

Word sense disambiguation (WSD) is the task of assigning a meaning to an ambiguous word given the context in which it occurs. WSD serves as an intermediate step for many computer science applications such as machine translation, information retrieval, hypertext navigation, content and thematic analysis, speech processing. Therefore, it has been a central problem since the earliest days of computational studies of natural language.

WSD requires a set of meanings for each word to be disambiguated and a means to choose the correct one from that set. It is a common practice to use the word sense distinctions of a machine readable dictionary; particularly, the use of WordNet for this purpose has become a standard. Consequently, WSD algorithms almost always deal only with the latter task.

WSD methods utilize external knowledge sources such as machine readable dictionaries (MRD), thesauri, tagged or untagged corpora. Longman Dictionary of Contemporary English (LDOCE) has been the most widely used MRD. LDOCE is a highly structured dictionary that allows for building taxonomies easily. It has fine sense distinctions which are ordered by frequency. The first machine implemented knowledge base was Roget's Thesaurus in 1957. Merriam Webster Seventh Collegiate Dictionary, prepared between 1966—68, was the first machine-readable MRD. Other MRDs exist such as Collins English Dictionary.

WordNet [4] is a freely available lexical database. It specifies hierarchical relations between word senses. The relations include synonymy, antonymy, is-a, and part-of. Related concepts are grouped into synonym sets. A dedicated class of algorithms has been developed to exploit the WordNet structure.

A major classification of WSD methods is based on whether human assistance is required. S*upervised* algorithms are trained on correctly sense tagged examples. However, hand tagged corpora are rather limited due to the general problem of *knowledge acquisition bottleneck* which states that human undertaking can only provide a small portion of the vast volume of examples required for computational studies of language. SemCor, developed by the same team of WordNet, is a popular sense tagged corpus. On the other hand, *unsupervised* WSD algorithms do

not require human assistance. Therefore they are more feasible and scalable despite their own idiosyncratic problems.

As noted by eminent scholars, the field of word sense disambiguation—and computational linguistics in general—has seen much repetition in 60 years of research. Although this may, in part, be attributed to surprising achievements of early researchers in spite of primitive hardware and software resources of the time; the main cause seems to be unfamiliarity with other researchers' work—especially those of previous generations. Few number of qualified surveys attests to this problem. My humble desire is that this survey—though very limited in scope—serves as an introduction to many brilliant ideas developed over years in the field of word sense disambiguation and invites anyone to make her contribution.

1.2. Word Sense Disambiguation Methods

1.2.1. Methods Based on the Context Window of the Target Word

These methods constitute the earliest WSD examples. The set of words to the left and right of the target word in the context, called *window,* is used for disambiguation. Several researchers recognized the importance of window and carried out experiments to determine the optimal window size [1]. In bag-of-words method other features such as distances, syntactic relationships, etc. are not considered.

1.2.2. AI-based Methods

AI-based methods rose to popularity in 1960's following the advances in theoretical linguistics; namely, the discovery of formal rules and transformational theories. The distinctive feature of this class of algorithms is the claim to model human language understanding. However, there is no set of objective criteria to verify these claims and the proposed methods are

not supported by empirical evidence. Given the arbitrariness in choosing human cognition model, it should come as no surprise that methods themselves vary greatly.

A notable case is the Word Expert Parser (WEP) developed by Adriaens & Small [5] until early 1980's but discontinued since then. WEP methodology is a radical departure from conventional parsing models in that it is a semantic parsing program with a strong emphasis on disambiguation mechanisms. No central decision mechanism is assumed; program control is carried out by words which are considered as active agents running in parallel. These agents—or experts—interact with the following knowledge sources: the words in its immediate context, the concepts processed so far or expected locally, knowledge of the overall process state, knowledge of the discourse, and real world knowledge. Eventually, the experts are expected to agree on the meaning of the text fragment. WEP faced serious implementation problems; nevertheless, it marks a unique approach in the history of WSD research.

In striking contrast to WEP's parallelism, Milne [6] carries out lexical ambiguity resolution by a strictly deterministic parser without backtracking. He tries to avoid special mechanisms as much as possible to be compatible with human understanding processes. However, terms such as elegance and simplicity are not open to scientific inquiry and therefore the connection with human cognition models is loosely defined. The parser is implemented in Prolog. All parts of speech which a word can take on in a sentence are listed in the corresponding entry for that word in the dictionary. A three-word look-ahead window is used. The restricted window size causes a commonly encountered problem with some sentences. For example, "I told the girl (that) (the) (boy) hit the story" and "I told the (that) (the) (boy) will kiss her" are indistinguishable from the point of view of the program.

Yuret [7] proposes the method of 'lexical attraction' which can be defined as the likelihood of two words being related. The model has an intuitive appeal for explaining language acquisition of children where they learn the relations between words before forming grammatically error-free sentences. Neither grammar nor a lexicon with parts of speech is provided. The training is based on a bootstrapping procedure. Sentence relations are represented as 'dependency structures', an example of which is figure 1.1, as opposed to the more conventional 'phrase structures'. The program analyzes the relation between content words, based on the idea that that the lexical attraction betweeen content words directly determines the meaning of the sentence. The likelihood of syntactic relation is parallelled by a decrease in entropy from an information theory perspective. The probabilities are estimated by counting the occurences of each pair in the same relative position. The program consists of the processor and

the memory. The goal of the processor is to find the dependency structure that assigns a given sentence a high probability. In return, memory element stores the frequency of pairs. The two aspects of the program are intertwined. Test results indicate 60% precision and 50% recall on a 200 sentence test which is said to be better than the previous work in unsupervised language acquisition when no initial knowledge is given.

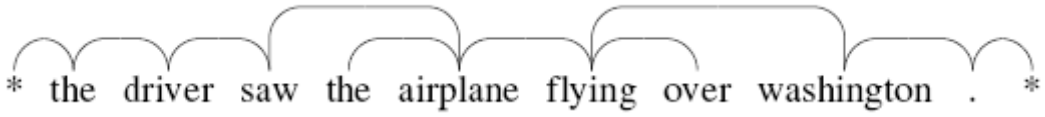

Figure 1.1. A dependency structure

1.2.3. Knowledge-based Methods

Knowledge-based methods utilize lexical and semantic knowledge bases such as machine-readable dictionaries (MRDs), thesauri, computational lexicons. Despite the efforts to automatically create knowledge bases, WordNet, the most widely used one, was created by hand. A brief introduction of WordNet is in order since almost all recent work on WSD has used WordNet in some way or another. WordNet was developed by Princeton University. Like an ordinary dictionary it contains definitions—glosses—of words; however, its distinctive feature is semantic relationships which form hierarchical structures of words. One such basic structure is *synset*—i.e. synonym set—which represents a single concept. Nouns, verbs, adjectives, and adverbs are classified separately in WordNet; with nouns constituting the richest and the most useful part. Hypernymy/hyponymy relation between nouns forms a hierarchical tree structure. A hyponym (subname) is a kind of its hypernym (supername). Meronymy (part-name) and holonymy (whole-name) are "part of" relations. Verbal hierarchy is based on troponymy which shows a "a manner of" relation; like stroll/walk. Attributive adjectives are organized according to the antonymy relation, whereas relational adjectives point to nouns. Adverbs are organized according to synonymy and antonymy relations.

Given the types of information contained in lexical knowledge bases, there have been two traditions differing in the way they utilize these resources. One focuses on glosses of word

meanings while another focuses on hierarchical structure as that of WordNet described above. However, this classification by no means is intended to imply a dichotomy since the two approaches are complementary and recent research usually combines them.

Lesk's algorithm [8] compares context of the target word with the gloss of each sense. The context words are treated in a bag-of-words approach. The decision is based on the maximum amount of overlap. The method achieves 50-70% accuracy with fine sense distinctions. The performance is highly sensitive to the choice of particular words in glosses. Despite its shortcomings, Lesk's algorithm has opened up the path for subsequent work.

Pedersen et al. [9] generalizes Lesk-style methods into the concept of "semantic relatedness". All nine methods taken as a measure of semantic relatedness in this article makes use of the observation that words that occur together in a sentence should be related to some degree. Rada defines the conceptual distance between any two concepts as the shortest path through a semantic network. Leacock and Chodorow scales this edge distance by the length of the longest path from a leaf node to the root node of the hierarchy. Wu and Palmer use the distance from a concept to the root node. Resnik refines their measure by introducing the concept of *lowest common subsumer*—the most specific node that intersects the path of the two concepts in the is–a hierarchy. Hirst and St. Onge consider relations other than is-a relation as well and have four levels for the strength of the relation. Resnik measures semantic relatedness using *information content* of the noun concept in the hypernymy/hyponymy hieararcy. Information content indicates the amount of specificity; for example, *sheep dog* has a higher information content than *animal.* Jiang and Conrath define a similar measure to that of Resnik but they subtract the information content of the lowest common subsumer from the sum of information contents of individual concepts. Lin divides the information content of the lowest common subsumer by the sum of information contents of individual concepts. The extended gloss overlap measure developed by the authors extends Lesk's algorithm by including the WordNet relatives in the comparison. In gloss vectors measure devoped by the authors, word vectors are created from gloss co-occurrence data and a gloss vector is created as the average of vectors of words appearing in the gloss. The disambiguation algorithm runs independently of the relatedness measure which is a function that takes as input two input senses and outputs a number. The algorithm is expressed concisely in figure 1.2.

$$argmax_{i=1}^{m_t} \sum_{j=1, j \neq t}^{n} max_{k=1}^{m_j} relatedness(s_{ti}, s_{jk})$$

Figure 1.2. Formula for the disambiguation algorithm based on maximum semantic relatedness

Rigau et al. [10] correctly state that most WSD algorithms have been developed as stand-alone and investigate the possibility of combining them. The methods in the study include those used by Pedersen et al. and some baseline methods such as using the most frequent sense. The contribution of each method to the final decision is normalized according to its weight in the interval of 0 and 1. Test results indicate approximately 8 % increase in precision for the combination of disambiguation methods.

1.2.4. Corpus-based methods

Corpus-based methods grew in importance after the public availability of large-scale digital corpora. A corpus is a collection of—preferably published—texts used for linguistics purposes. Texts should be selected across a variety of domains to cover different word senses since domain usually restricts words to one sense only; for example, in a paper on economics, the word "interest" may be used exclusively for the economics-related sense. Corpora provide vast volume of information regarding language usage; therefore they are especially well-suited for statistical or empirical methods.

The way disambiguation methods utilize corpora forms a classification. S*upervised learning* requires manually disambiguated words, whereas *unsupervised learning* does not. Since there is no equivalent of Moore's law for human workforce, hand-tagging training corpora creates the infamous roadblock known as *knowledge acquisition bottleneck.* Several researchers have tried to circumvent this problem. Agirre and Martinez [11] try to derive automatically trained data from the web. They use monosemous synonyms and glosses in WordNet to construct queries for search engines. A fixed number (100) of documents for each sense of the word are retained. Then the target words are substituted for synonyms. However, their results are disappointing with test results being close to random baseline. Leacock et al. [12] used WordNet relations such

as hypernymy, hyponymy, and part-of to find monosemeous relatives near the target word in the sentence. They report the performance of the unsupervised training only marginally behind supervised training. Other work-around solutions for building sense-tagged data for unsupervised training include using aligned bilingual corpora, creating pseudowords by combining two words like “escorted/abused”, using homophones like “seller/cellar”.

Word sense diambiguation can be thought as consisting of two successive stages. First step is *sense discrimination* where the occurrences of a word are mapped into a number of classes depending on the sense they belong to. The second step, *sense labelling,* then assigns a sense to each class, hence to each word in that class. Schutze’s [13] method is interesting in that it deals solely with the first step, eliminating the need to refer to any outside knowledge base such as MRD’s. Thus it can be considered as a purely corpus-based method. The method relies on “second-order co-occurence” derived from the words that the context words of the target word co-occur with in the training corpus. The underlying assumption is that humans make use of context similarity information for the disambiguation task. There are three types of entities in the program: *word vectors, context vectors,* and *sense vectors.* A vector for word w is derived from the words that co-occur with w in the context of w in the corpus. The entry for word v in a word vector denotes the number of times that word v occurs close to w in the corpus. A context vector (see figure 1.3) is the centroid (or sum) of the vectors of the context words. The final abstraction is the sense vectors which are derived in turn from context vectors via EM clustering algorithm. Results indicate that performance varies according to word types where words that are relatively independent of context are discriminated worst.

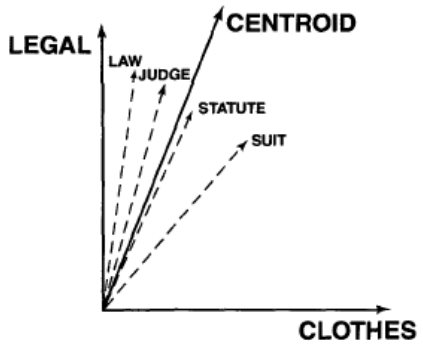

Figure 1.3. (from Schutze, 1998) An example context vector for the word *suit*

A notably successful unsupervised method is Yarowsky's [14]. His work builds upon two regularities observed in natural languages: (1) words tend to have one sense in a given discourse, and (2) one sense per collocation. The algorithm proceeds according to a bootstrapping procedure starting from seed collocations arriving at a convergence condition where each word in the corpus is assigned a sense. For each sense of the word, either a small number of training examples or some collocations are given to the program to populate the sets corresponding to different senses. Iteratively, the most probable collocation in each set is selected based on *decision lists* and other occurences with the new collocations are added into sense sets until they cover all the corpus. An accuracy of 95.5 % is achieved on a test corpus consisting of 460 million words.

1.3. Problems in WSD

Since its very first inception, WSD efforts have been plagued by deep-rooted problems. Consequently, a major portion of research has revolved around developing solutions for these problems. In fact, most of the work referenced above address the same issues in their own way. *Data sparseness* problem is related with corpus-based methods. Some senses of a word are used very rarely as stated by Zipf's law; therefore a corpus may contain few, or worse, no examples regarding the infrequent sense. A system trained on that corpus cannot produce meaningful answers for that sense of the word. *Inter-annotator agreement* problem is not related with methods but the nature of the problem itself. Put succinctly, it states that: *Which of the human informants should the computer agree with, if the humans cannot agree among themselves?* Murray et al. [15] have come up with surprising results that contradict the common practice. Their findings show that level of agreement cannot be used as an indicator of correctness because subjects who performed badly in verbal tests gave rather similar, but not necessarily correct, answers. Other cross-annotator studies indicate agreement levels as low as 60% which shed further doubts on the formulation of the problem and granularity of sense divisions.

2. ALGORITHM

2.1. Pseudowords

Pseudowords have been used in various natural language processing applications since its introduction by Schütze in 1992 [16]. In WSD, we make use of pseudowords in the following way. Two arbitrary words are selected and merged to create a hypothetical word, or a pseudoword, which is nonexistent previously. This pseudoword has two meanings corresponding to two original words. In the corpus, the occurrences of the original words are replaced by the pseudoword. The correct meaning of the pseudoword is the original word in the sentence. Below are example sentences for the 'banana_sky' which is made up of separate words 'banana' and 'sky'. It should be noted that a natural ambiguous word has typically more than two senses which are semantically more related. Therefore pseudowords provide coarse-level sense distinctions with more optimistic test results.

Corpus sentence:	The sky is blue.
Creating the pseudoword:	The banana_sky is blue.
Correct sense:	sky
Corpus sentence:	I love eating bananas.
Creating the pseudoword:	I love eating banana_skys.
Correct sense:	banana

Figure 2.1. Example corpus occurrences for the pseudoword 'banana_sky'.

2.2. Creating the Decision List

Decision lists are formed as a collection of evidences ordered according to their logarithmic likelihood values. Each evidence has a sense tag to be assigned to the ambiguous word. Three evidence types were used: (i) the surface and lemma forms of words immediately to the right or left of the target word, (ii) the surface and lemma forms of two words immediately to the right of

left of the target word or one to the left and one to the right. (iii) the surface and lemma forms of any other word in the sentence. The sum of occurrences of an evidence type on a set of sentences where the target word indicates the most frequent sense are divided by the sum obtained over the other sense set. The logarithm of this number is calculated to give the logarithmic likelihood. The evidence is then inserted into decision list together with the logarithmic likelihood value and the sense tag. Larger logarithmic likelihood values indicate that the evidence is observed more frequently on sentences where the target word is used with the chosen sense while it is observed less frequently on sentences where other senses are intended. In this respect, evidences with large logarithmic values have better discerning capacity. Similarly, in the disambiguation step, evidences are extracted from the sentence containing the target word. Starting from the topmost rule in the decision list, it is checked whether a match occurs. If this is the case, the sense indicated in the matching rule is returned as the answer. A separate decision list is created for each target word.

$$logarithmic\text{-}probability \leftarrow Log\left(\frac{number\_of\_occurrences\_over\_the\_sense\_set}{number\_of\_occurrences\_over\_the\_other\_sense\_set}\right)$$

Figure 2.2. Logarithmic probability calculation

2.3. Supervised Learning

Supervised and unsupervised learning methods differ based on whether the correct senses of the target words in the training corpus are given or not. In supervised learning, training sentences are partitioned according to the given sense tags. These sets are then used to create the decision list as explained in the previous section.

2.4. Unsupervised Learning

Supervised learning requires many training sentences for each word. Bearing in mind that even in English, for which the most extensive research has been carried out historically, the sense tagged corpora are rather limited. It is a crying necessity to make better use of untagged corpora to be able to perform word sense disambiguation for any word in a running text. The proposed iterative algorithm relies on the paradigm "one sense per collocation" first discovered by philosophers and linguists. This paradigm—as expressed in Wittgenstein's words "the meaning of a word is its use in the language" and in Wirth's words "You shall know a word by the company it keeps"—has been shown to reach 96 percent validity [17]. A collocation is specified for each sense of the target word. Collocation is not defined in the strict sense as used by linguists, which entails being adjacent in a sentence or forming an idiom; but as being together in a sentence with a probability higher than expected. First, the training sentences are partitioned into sense sets based on the given collocations. For example, the collocation *fruit* is given for the *banana* sense of the pseudoword *banana_sky* and *blue* for the *sky* sense. Based on these sense sets, an initial decision list is created. Iteratively, training sentences are partitioned into sense sets using the decision list and the decision list is created anew using these sense sets. After each iteration, we expect a larger set and a more accurate decision list. The redundancy property of natural languages and the iterative nature of the algorithm allows for avoiding classification errors rising from using a single collocation, and an accuracy level close to that of supervised learning.

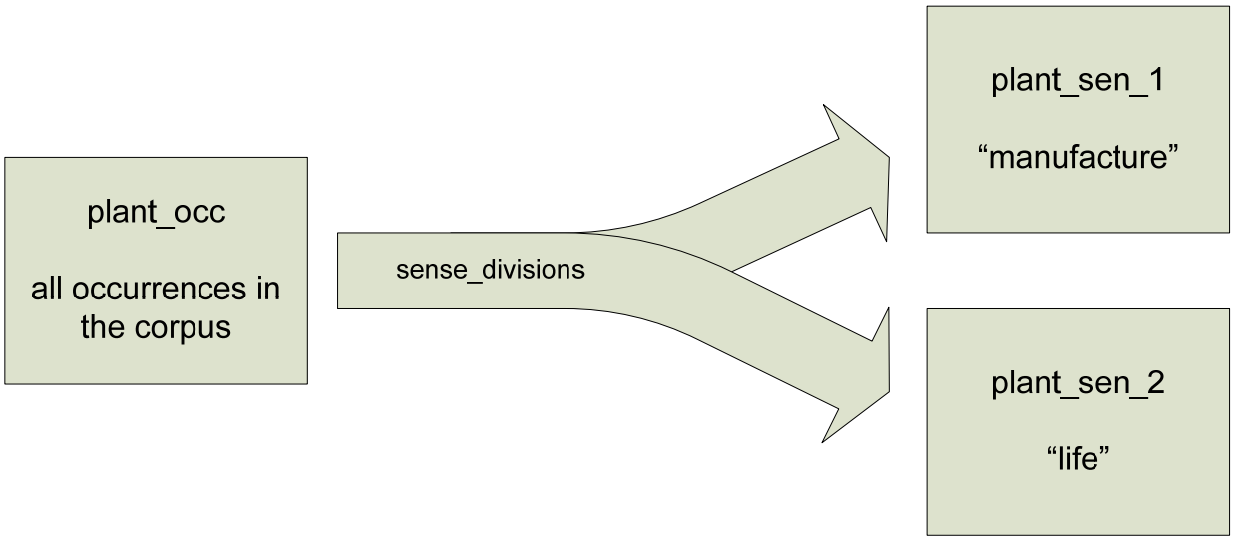

Figure 2.3. The *sense_divisions* subroutine partitions the occurrences in the corpus

2.5. Post-Pruning

Post-pruning is applied after the decision list has been created in supervised learning and after each iteration in unsupervised learning. The post-pruning step consists of removing those rules which cause more wrong answers than correct ones as evaluated on the held-out set. Post-pruning provides a more concise decision list besides a small improvement in accuracy.

3. EVALUATION

3.1. Setup

Lower and upper bounds help us put the WSD results into perspective and give a crude idea of what to expect. Choosing the most common sense is a simple method to obtain a lower bound. In this project, test sets include equal number of instances for each of the two senses of pseudowords; therefore the lower bound is 50 %. Inter-annotator agreement is accepted as the upper bound. For fine-grained sense distinctions, inter-annotator agreement is 75—80 %, and for coarse-grained sense distinctions it is 85—90 %. Accuracy is measured with precision and recall.

$$Precision = \frac{\text{number of instances which have been correctly tagged}}{\text{number of instances which have been tagged}}$$

$$Recall = \frac{\text{number of instances which have been correctly tagged}}{\text{number of test instances}}$$

Figure 3.1. Accuracy Parameters

The sentences in the corpus containing the target word were partitioned into training, validation, held-out and test sets; numbers being 2000/500/500/500 respectively. Parameters were tuned on the validation set. Post-pruning was carried out on the held-out set.

3.2. The Effect of Logarithmic-Likelihood Threshold

This parameter indicates the threshold while inserting new rules into the decision list. For example, a new rule with a log-likelihood value of 1.5 is not admitted when the threshold is 2. The figure 3.2 was obtained for the pseudoword of *banana_sky* using supervised learning and threshold values varying between 1 and 4.

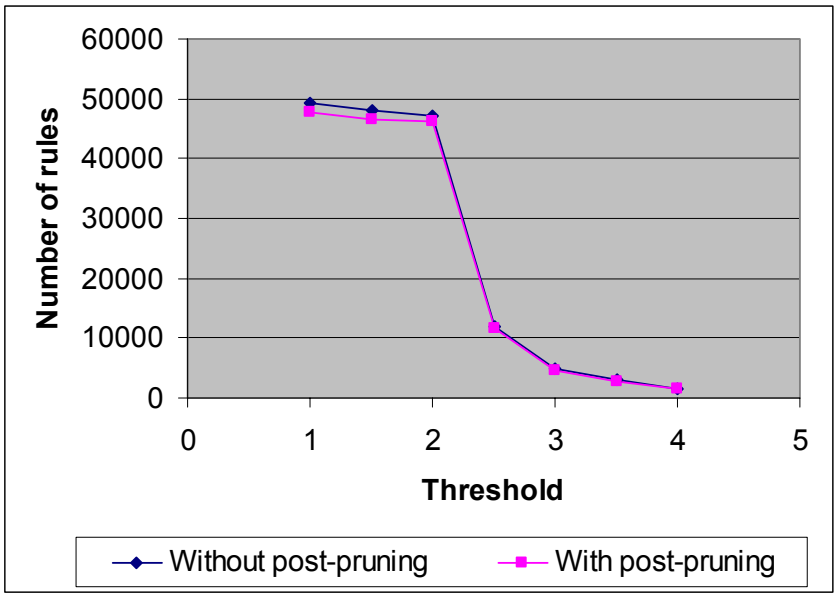

Figure 3.2. Decision list size in relation to the threshold

A considerable reduction in the decision list size was obtained by using larger threshold values. The run-time and space complexity diminish as the decision list gets smaller. Post-pruning caused a further 5 % decrease.

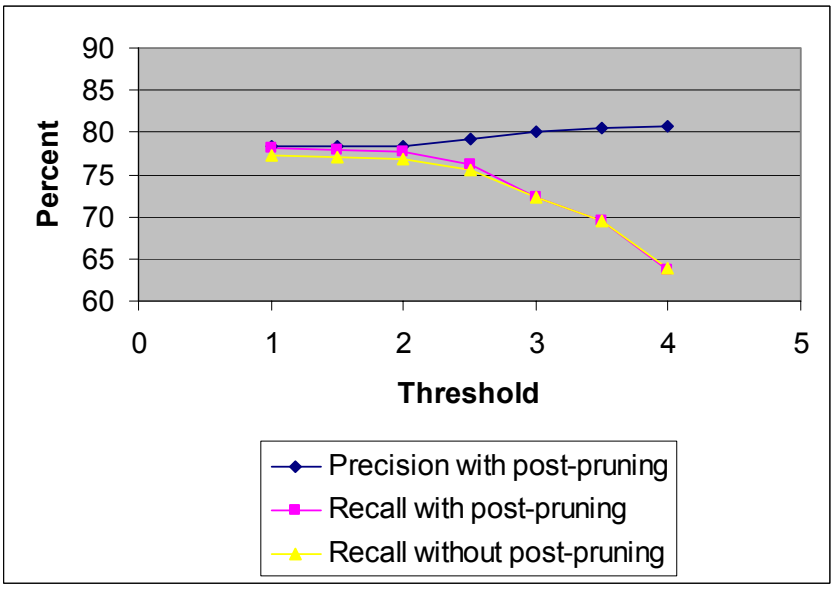

Figure 3.3. Accuracy in relation to the threshold

With a larger threshold, recall dropped and precision slightly improved. These results can be interpreted as follows. In the case of a larger threshold, new rules are scrutinized more;

therefore only better ones qualify to be inserted into the list. The ratio of correct answers out of all answers, thus precision, increases. However, the number of cases where the system does not return an answer increases, resulting in a drop in recall. Taking both graphs into consideration, we see a tradeoff between complexity/precision/recall.

3.3. The Effect of Number of Iterations in Unsupervised Learning

To evaluate the accuracy of unsupervised learning, the method described previously was applied for the pseudoword *banana_sky* using collocations *fruit* and *blue.* With a threshold value of 2, the following plot was obtained.

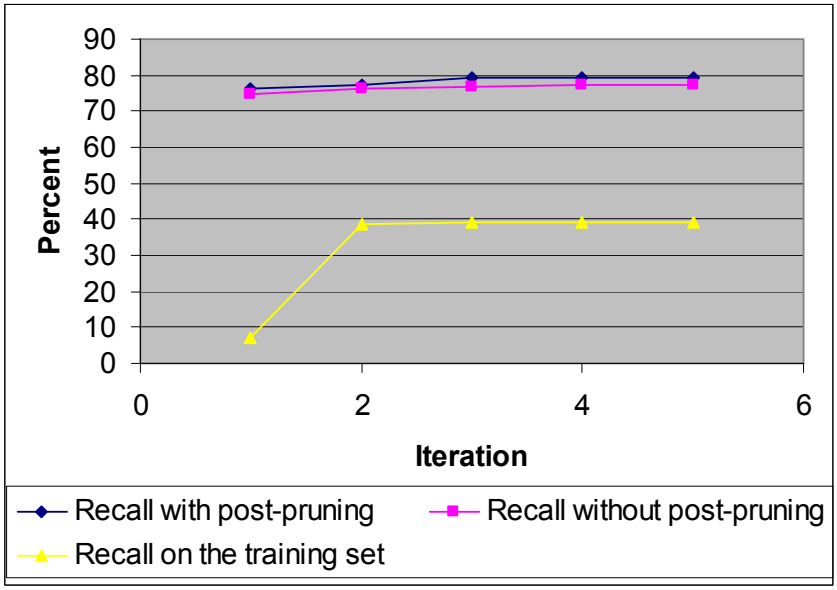

Figure 3.4. Accuracy of the Iterative Algorithm

Unsupervised learning reached 74% recall rate on the test set. The result, close to 77.7% value obtained by supervised learning, demonstrates the applicability and robustness of the iterative approach. Recall rate on the training set shows the portion that has been used for creating the decision list. This number is 100% for the case of supervised learning. Although the initial decision list of the unsupervised algorithm constructed based solely on collocation words used 7.22% of the training set, with successive steps using as high as 40%; high accuracy values were obtained. Post-pruning provided 2% improvement.

The iterative algorithm based on pseudowords is a standardized method for contructing and evaluating a word sense disambiguator, thus seems independent of chosen words' characteristics. For example, a recall rate of 74.70% was obtained for the pseudoword *lecture_lesson* consisting of two semantically related words *lecture* and *lesson*.

4. PROGRAM SPECIFICATIONS

4.1. Overview

The program and the data are organized under their corresponding directories. Since the algorithm accesses the data files at runtime to read and write back intermediate results, it is important to maintain the directory structure as seen in figure 4.1. for proper execution.

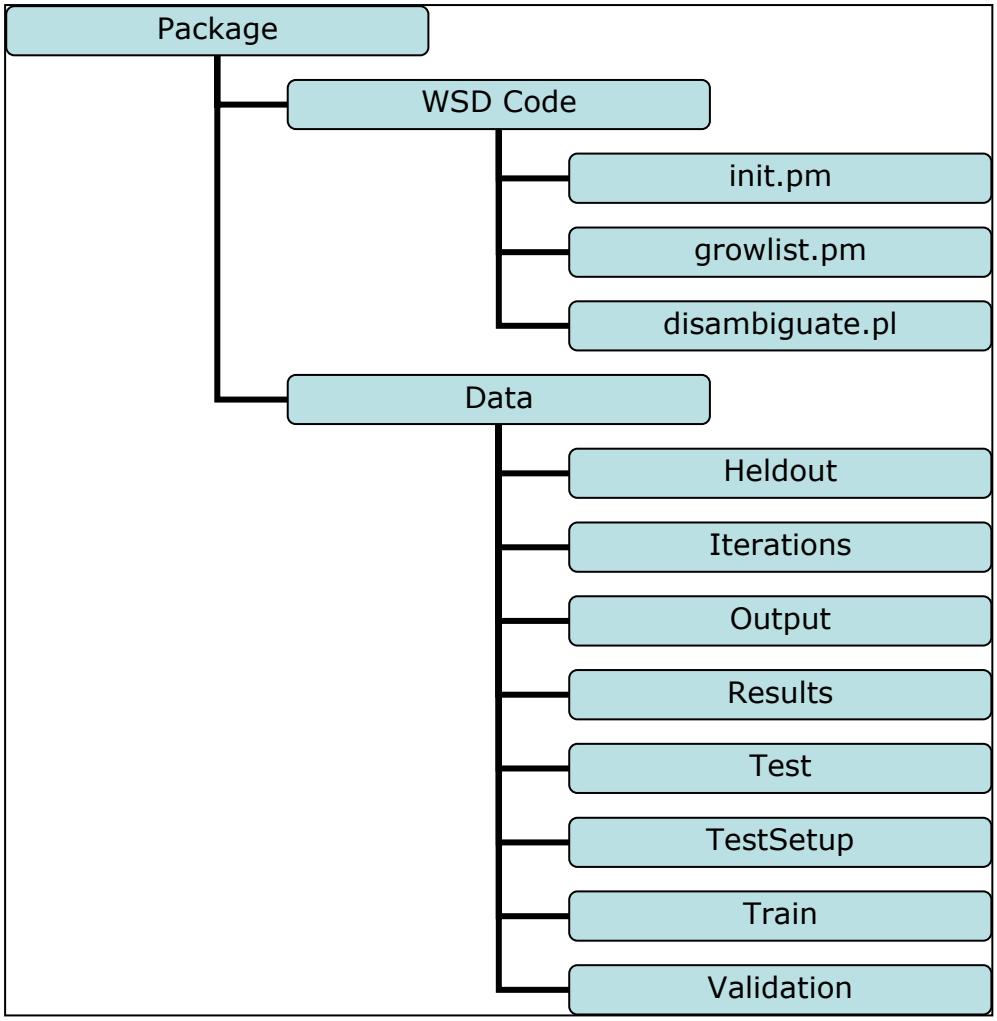

Figure 4.1. Directory Structure

The functions are packaged in two Perl modules: *init.pm* and *growlist.pm*. The *init.pm* module contains the functions *sense_divisions* and *sense_divisions_supervised* for creating the initial sense sets, using unsupervised and supervised methods respectively. The *growlist.pm* module contains functions to construct the final decision list, either iteratively for the

unsupervised method or in a single step for the supervised case, and the post-pruning function. It also contains functions for measuring accuracy on training, test, and heldout sets. The call graph is given in figure 4.2.

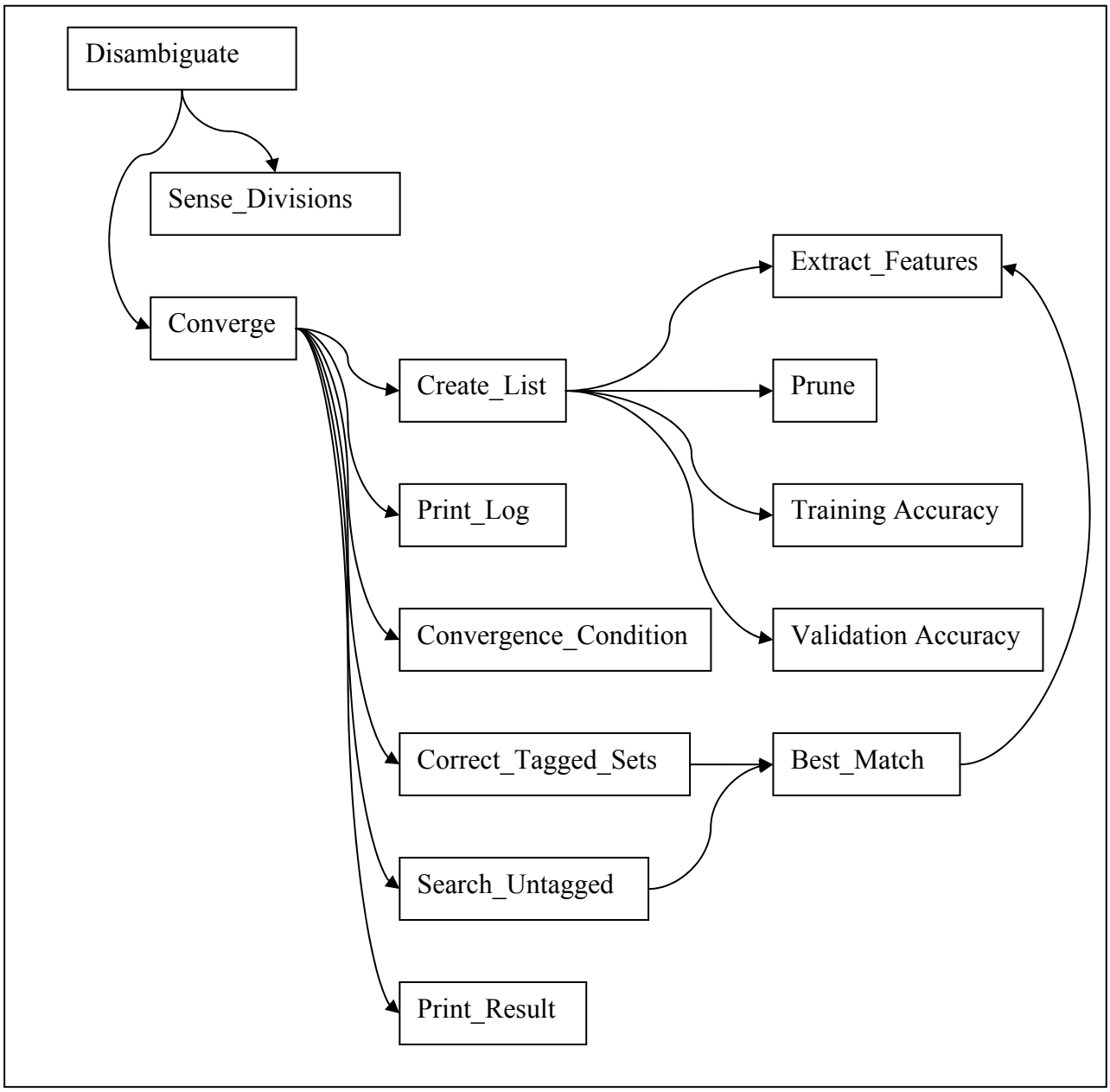

Figure 4.2. Call Graph

	Heldout	Iterations	Output	Results	Test	TestSetup	Train	Validation
Sense_Divisions		✓				✓	✓	
Sense_Divisions_Supervised		✓					✓	
Accuracy					✓			✓
Training_Accuracy		✓						
Prune	✓							
Print_Log			✓					
Print_Result				✓				
Correct_Tagged_Sets		✓						
Search_Untagged		✓						
Create_List		✓						

Figure 4.3. List of Directories Accessed by each Function

4.2. Init.pm Module

```
function Sense_Divisions_Supervised(target_word)
    inputs: target_word
    local variables:
        senses, an array storing the words that constitute the pseudoword
        sense_id, a hash storing a numerical id for each sense

    senses ← split the target word around the underscore
    remove all the existing files under the folder Data\Iterations
    sense_id ← starting with '1', assign numerical values to senses
    for each sentence in the occurrences file Data\Train\target_word do
        label ← extract the initial word which indicates the correct sense of the  target word in the
                    following sentence
        id ← look up label in the sense_id hash
        append the sentence to the file Data\Iterations\target_word_sen_id
```

Figure 4.3. Pseudocode of Sense_Divisions_Supervised

```
function Sense_Divisions(target_word)
    inputs: target_word
    local variables:
        senses, an array storing the words that constitute the pseudoword
        sense_id, a hash storing a numerical id for each sense
        collocations, an array storing one collocation for each sense

    senses ← split the target word around the underscore
    remove all the existing files under the folder Data\Iterations
    collocations ← read collocations from the file Data\TestSetup\setup
    for each line in the occurrences file Data\Train\target_word do
        target_index, sentence ← first token is the position of the target word the rest of the line is
                                 the sentence containing the target word.
        remove non-content words in the sentence
        for each token in the sentence do
            calculate the distance to the target_index
            if surface form or lemma matches one of the collocations then
                if distance is smallest among the tokens seen so far then
                    matching_sense ← the sense to which the collocation belongs
                    id ← look up matching_sense in the sense_id hash
        if a match has occurred then
            append the sentence to the file Data\Iterations\target_word_sen_id
        else
            append the sentence to the file Data\Iterations\target_word_untagged
```

Figure 4.4. Pseudocode of Sense_Divisions

4.3. Growlist.pm Module

```
// Returns precision and recall on the training set
function Training_Accuracy(training_word, iteration_number)
    inputs:
        target_word, iteration_number as input arguments
        senses, an array imported from the init.pm module storing the constituting words of the
                pseudoword
    local variables:
        hits, misses, no_answers, counter variables used in calculating precision and recall

    for each sense file such as target_word_sen_1, etc under Data\Iterations do
        for each line in the file do
            correct_sense ← extract the first word in the line
            chosen_sense ← senses[file_number] where file_number is one less than the filename
                           suffix; e.g. 0 for target_word_1
            if correct_sense and chosen_sense are equal then
                increment hits by 1
            else
                increment misses by 1
    for each sentence in the file target_word_untagged under Data\Iterations do
        increment no_answers by 1

    precision ← 100 * hits / (hits + misses)
    recall ← 100 * hits / (hits + misses + no_answers)
    if iteration is 0 i.e. the first iteration then
        reset the global arrays tr_precision and tr_recall
    tr_precision[iteration] ← precision
    tr_recall[iteration] ← recall

    return precision and recall
```

Figure 4.5. Pseudocode of Training_Accuracy

```
//Performs post-pruning
function Prune(target_word)
    inputs:
        target_word
    local variables:
        hits, misses, arrays showing number of hits and misses respectively for the corresponding
                the rules in the decision list

    for each line in the file Data\Heldout\target_word do
        correct_sense, target_index, sentence ← extract the first token, second token and the rest
                                                of the line
        context_features ← populate the array by calling function extract_features
        for each rule in the decision list starting from the topmost one do
            for each feature in the context_features do
                if feature and rule match then
                    found_sense ← as indicated by the current rule in the decision list
                    match_found ← true
                    break out of the inner loop
            if match_found then
                if found_sense is equal to correct_sense then
                    increment hits[rule_id]
                else
                    increment misses[rule_id]
    for each rule in the decision list do
        if misses[rule_id] > hits[rule_id] then
            delete the rule
```

Figure 4.6. Pseudocode of Prune

```
//Returns precision and recall on either validation or test set
function Accuracy(target_word, iteration_number, option)
    inputs:
        target_word
        iteration_number
        option, string with the value "validation" or "test"
    local variables:
        hits, misses, no_answers

    if option is "validation" then
        open the file Data\Validation\target_word
    else if option is "test" then
        open the file Data\Test\target_word

    for each line in the file do
        correct_sense ← extract the first token in the line
        best_sense ← call the function best_match(target_word, whole_line)
        if best_sense is defined
            if correct_sense and best_sense are equal then
                increment hits by 1
            else
                increment misses by 1
        else
            increment no_answers by 1

    precision ← 100 * hits / (hits + misses)
    recall ← 100 * hits / (hits + misses + no_answers)
    if option is "validation" then
        if iteration is 0 i.e. the first iteration then
            reset the global arrays vl_precision and vl_recall
        vl_precision[iteration] ← precision
        vl_recall[iteration] ← recall

    return precision and recall


function Validation_Accuracy(input_arguments)
    append the string "validation" to the list of input_arguments and call accuracy

function Test_Accuracy(input_arguments)
    append the string "test" to the list of input_arguments and call accuracy
```

Figure 4.7. Pseudocode of Accuracy

```
//Creates the decision list iteratively
function Converge(target_word, threshold, enable_pruning)
    inputs:
        target_word
        threshold, indicates the threshold for the decision list
        enable_pruning, whether pruning is enabled or not

    iteration ← 0
    create_list(target_word, iteration, threshold, enable_pruning)
    print out the iteration information into the log file
    until Convergence_Condition do
        increment iteration by 1
        correct_tagged_sets(target_word)
        search_untagged(target_word)
        create_list(target_word, iteration, threshold, enable_pruning)
    print out the final iteration information and the decision list
```

Figure 4.8. Pseudocode of Converge

```
function Converge_Supervised(target_word, threshold, enable_pruning)
    inputs:
        target_word
        threshold, indicates the threshold for the decision list
        enable_pruning, whether pruning is enabled or not
    iteration ← 0
    create_list(target_word, iteration, threshold, enable_pruning)
    print out the final iteration information and the decision list
```

Figure 4.9. Pseudocode of Converge_Supervised

```
//Returns true if convergence condition is satisfied
function Convergence_Condition
    global variables:
        vl_recall, array of validation recall values for successive iterations

    number_of_iterations ← size of the vl_recall array
    if number_of_iterations > 14 then
        condition ← true
    else if number_of_iterations > 4 then
        if vl_recall  did not improve more than 1% in any of last 3 iterations then
            condition ← true
    else
        condition ← false
    return condition
```

Figure 4.10. Pseudocode of Convergence_Condition

```
//Examines occurrences and relocate them to different sense files if necessary
function Correct_Tagged_Sets(target_word)
    inputs:
        target_word

    for each sense file such as target_word_sen_1, etc under Data\Iterations  do
        for each line in the file do
            best_sense ← best_match(target_word, line)
            if best_sense is defined then
                write the line to the file target_word_sen_best_sense_bak
            else
                write the line to the file target_word_untagged

    rename *_bak files as original files
```

Figure 4.11. Pseudocode of Correct_Tagged_Sets

```
//Disambiguates occurrences in the untagged set
function Search_Untagged(target_word)
    inputs:
        target_word

    for each line in the file Data\Iterations\target_word_untagged do
        best_sense ← best_match(target_word, line)
        if best_sense is defined then
            write the line to the file target_word_sen_best_sense
        else
            write the line to the file target_word_untagged_bak

    rename the file target_word_untagged_bak as target_word_untagged
```

Figure 4.12. Pseudocode of Search_Untagged

```
//Returns the best matching sense for the given target word and the its context
function Best_Match(target_word, line)
    inputs:
        target_word
        line

    target_index, sentence ← extract the first token and the rest of the line
    context_features ← populate the array by calling the function extract_features

    for each rule in the decision list starting from the topmost one do
        for each feature in the context_features do
            if feature and rule match then
                best_sense ← the sense indicated by the matching rule
                break out of the outer loop

    return best_sense
```

Figure 4.13. Pseudocode of Best_Match

```
//Creates a decision list based on the current sense sets
function Create_List(target_word, iteration, threshold, pruning)
    inputs:
        target_word
        iteration, the current iteration number as used in iterative learning
        threshold, indicates the threshold for the decision list
        pruning, whether post-pruning is enabled or not
    local variables:
        evidences, a hash whose keys are formed from concatenation of a context feature and the
                   chosen sense of the target word along with a delimiter character in the
                   middle. The values of the hash are simply the counts of the their keys.
    global variables:
        decision list, a hash where keys are features and values are log-likelihoods
                    sorted_decision_keys, an array for decision list rules ordered according to
                log-likelihood values

    for each sense file such as target_word_sen_1, etc under Data\Iterations  do
        for each line in the file do
            target_index, sentence ← extract the first token and rest of the line
            context_features ← as returned by the function extract_features
            for each feature in context_features do
                key ← concatenate feature, "\0", sense_file_number
                increment evidences{key} by 1

    sorted_keys ← sort keys of the evidences hash in descending order into an array
    for each key  in sorted_keys do
        feature, most_frequent_sense ← split the key string by the delimiter
        in_score ← evidences{key}
        for each sense other than the most_frequent_sense do
            out_score ← out_score + evidences{feature . sense}
        if out_score is 0 then
            out_score ← 0.1
        log_likelihood ← log(in_score / out_score)
        if log_likelihood > threshold then
            decision_list{key} ← log_likelihood

    sorted_decision_keys ← sort the keys of the decision_list hash
    if pruning is enabled then
        prune(target_word)

    training_accuracy(target_word, iteration)
    validation_accuracy(target_word, iteration)
```

Figure 4.14. Pseudocode of Create_List

```
//Returns either one of four coarse-level POS tags or the undefined value
function Pos_Type(argument)
    inputs:
        argument, a Penn treebank POS tag such as NN, VB, etc.

    if argument is either NN, NNS, NNP, NNPS or FW then
        pos ← noun
    else if argument is either VB, VBD, VBN, VBZ, VBG or VBP then
        pos ← verb
    else if argument is either JJ, CD, JJR or JJS then
        pos ← adjective
    else if argument is either RB, RBR, RBS then
        pos ← adverb
    else
        pos ← undef
    return pos
```

Figure 4.15. Pseudocode of Pos_Type

```
//Extracts all the features in the given context
function Extract_Features(target_word, target_index, context_tokens)
    inputs:
        target_word
        target_index, the position of the target word in the context
        context_tokens, an array for all the words in the context including non-content words. Each
                        element of the array is a triplet made up of the concatenation of the
                        word, its POS, and its lemma form.
    global variables:
        features, a hash whose keys are feature types such as "k_word", "left_lemma" etc. each
                  having distinct numerical values
    local variables:
        context_features, an array holding all the extracted features

    remove non-content words from context_tokens using the function pos_type
    for each token in context_tokens do
        index ← position of the token
        word, pos, lemma ← split the token by the delimiter
        distance ← index – target_index
        context_features ← push the k_word and the k_lemma features
        if distance is -1 then
            context_features ← push the left_word and the left_lemma features
            if there is at least one word to the left then
                context_features ← push the left_left_word and the left_left_lemma features
                                   using the words at positions at (-2) and (-1)
        else if distance is 1 then
            context_features ← push the right_word and right_lemma features
            if there is at least one word to the left of the target word then
                context_features ← push the left_right_word and the left_right_lemma features
                                   using the words at positions (-1) and (+1)
        else if distance is 2 then
            context_features ← push the right_right_word and right_right_lemma features using
                               the  words at positions (+1) and (+2)

    return context_tokens
```

Figure 4.16. Pseudocode of Extract_Features

5. REFERENCES

1. Ide, N., & Veronis, J., "Word Sense Disambiguation: The State of the Art," Computational Linguistics, No. 24, pp.2-40, 1998.

2. Rivest, R. L., "Learning Decision Lists," Machine Learning, Vol. 2, No. 3, Springer Netherlands, November 1987.

3. Yarowsky, D., "Decision Lists for Lexical Ambiguity Resolution: Application to Accent Restoration in Spanish and French," Proceedings of the 32nd Annual Meeting, pp. 88-95, Las Cruces, NM: Association for Computational Linguistics, 1994.

4. Miller, G. A., "WordNet: An On-line Lexical Database," Communications of the ACM, Vol.38 No. 11, 1995.

5. Adriaens, G., & Small, S. L., "Word Expert Parsing Revisited in a Cognitive Science Perspective". In S. L. Small et al. (Ed.), Lexical Ambiguity Resolution: Perspectives from Psycholinguistics, Neuropsychology, and Artificial Intelligence, pp. 13-43, 1988.

6. Milne, R., "Lexical Ambiguity Resolution in a Deterministic Parser". In S. L. Small et al. (Ed.), Lexical Ambiguity Resolution: Perspectives from Psycholinguistics, Neuropsychology, and Artificial Intelligence, pp. 45-74, 1988.

7. Yuret, D., "Discovery of Linguistic Relations Using Lexical Attraction," Ph.D. Dissertation, MIT, 1998.

8. Lesk, M., "Automatic Sense Disambiguation Using Machine Readable Dictionaries: How to Tell a Pine Code From an Ice Cream Cone," Proceedings of the 5th Annual International Conference on Systems Documentation, pp. 24–26, ACM Press, 1986.

9. Pedersen, T., Banerjee, S., Patwardhan, S., "Maximizing Semantic Relatedness to Perform Word Sense Disambiguation," Supercomputing Institute Research Report,

University of Minnesota, Minneapolis, MN., 2005.

10. Rigau, G.; Atserias, J. ; Agirre, E. (1997). Combining unsupervised lexical knowledge methods for word sense disambiguation. Proceedings of the 35th annual meeting on Association for Computational Linguistics, pp.48-55.

11. Agirre, E., Martinez, D. "Exploring Automatic Word Sense Disambiguation With Decision Lists and the Web," Procedings of the COLING 2000 Workshop on Semantic Annotation and Intelligent Content, 2000.

12. Leacock, C., Chodorow, M. and Miller, G.A. "Using Corpus Statistics and WordNet Relations for Sense Identification," Computational Linguistics, No. 24, pp.147-165, March 1998.

13. Schutze, H., "Automatic Word Sense Discrimination," Computational Linguistics, No. 24, pp.97-123, March 1998.

14. Yarowsky, D., "Unsupervised Word Sense Disambiguation Rivaling Supervised Methods," Proceedings of the 33rd annual meeting on Association for Computational Linguistics, pp.189-196, Cambridge, Massachusetts, 1995.

15. Murray, G. C., & Green, R., "Lexical Knowledge and Human Disagreement on a WSD Task," Computer Speech and Language, No. 18, pp. 209-222, 2004.

16. Schütze, H., "Context Space," In Working Notes of the AAAI Fall Symposium on Probabilistic Approaches to Natural Language, pages 113—120, Menlo Park, CA. AAAI Press.
17. Yarowsky, D., "One Sense per Collocation," Proceedings of the ARPA Human Language Technology Workshop, 1993

18. Liu, Hugo (2004). MontyLingua: An end-to-end natural language processor with common sense. Available at: web.media.mit.edu/~hugo/montylingua

APPENDIX A. USER GUIDE

A.1. Using Auxilliary Funtions to Prepare Data

The *Misc* directory contains two functions to collect occurrences from the corpus and partition those occurrences into training, validation, heldout and test sets. The function *corpus_collect* searches the corpus files recursively starting from the root directory. The words to be searched and the root directory are specified by assigning appropriate values to their corresponding variables at the top of the file. The occurrences—i.e. the sentences in which a given word occurs—are then written to the Data\Initial\{target_word}_occ file. The corpus is presumed to be preprocessed with MontyLingua package [18]. The corpus sentences are in the following format:

```
    The/DT/The first/JJ/first election/NN/election in/IN/in
the/DT/the reunified/VBN/reunify Germany/NNP/Germany was/VBD/be
in/IN/in 1990/CD/1990 ././.
```

Figure A.1. Preprocessed Sentence Format in the Corpus

After occurrences for two words are collected, a pseudoword can be built out of these words and training, validation, test and heldout sets can be created by the function *create_sets*. This program accepts the pseudoword to be created as the command line argument, e.g. call create_sets with *banana_sky* argument.

A.2. An Example Disambiguation Program

Creating and evaluating a decision list for a pseudoword is straightforward. The two modules are imported in the beginning of the file. The functions are called with the required set of parameters. A simple program is listed in figure A.2.

```
use init qw/ &sense_divisions/;
use growlist qw/ &converge /;
&sense_divisions('banana_sky');
my($threshold, $pruning)=(2, 1);
&converge('banana_sky', $threshold, $pruning);
```

Figure A.2. An Example Disambiguation Program