Using the ADIOS Algorithm for Grammar Induction and Psycholinguistic Comparison

by

Joe D. McFall

(Under the direction of Michael Covington)

Abstract

This thesis explores the field of natural language grammar induction as applied to psycholinguistic comparison. It specifically concentrates on one algorithm, the ADIOS algorithm. After a discussion of language, grammar and grammar induction methods, it describes the ADIOS algorithm, and presents experiments done on the induction and comparison of grammars from small corpora, with the goal of inferring linguistic manifestations of learning disabilities. Results show that the grammars used by the groups in question are largely similar, with one exception indicating possible differences in the grammatical consistency of normal vs. learning disabled populations. Suggestions are made regarding the future usefulness of such methods in the study of cognition and cognitive differences.

Index words: Grammar Induction, Grammatical Inference, Syntax, Learning Disabilities

Using the ADIOS Algorithm for Grammar Induction and Psycholinguistic Comparison

by

Joe D. McFall

B.A., Emory University, 1998

M.A., The University of Georgia, 2004

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Master of Science

Athens, Georgia

2007

Using the ADIOS Algorithm for Grammar Induction and Psycholinguistic Comparison

by

Joe D. McFall

Approved:

Major Professor: Michael Covington

Committee: Walter D. Potter
Khaled Rasheed

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2007

Dedication

This thesis is dedicated to my family, for all of the support they have offered over the years.

Acknowledgments

I would like to thank my major advisor, Dr. Michael Covington, for teaching me computer programming, as well as his continued support during the pursuit of this degree. Under his guidance, I have acquired skills that will serve me throughout my life, and I am indebted to him for this.

I would also like to thank Sam Fisher his friendship and his encouragement throughout my graduate studies in linguistics and artificial intelligence. For over a decade, Sam has been my intellectual collaborator, my academic partner-in-crime and a good friend. He has continually inspired and challenged me to be creative and disciplined in my scholarly pursuits

Sybil E. Smith also deserves special thanks, for her support and patience throughout my degree program.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This paper is an investigation into how cognition is manifested in language, and how that manifestation can be analyzed using techniques from natural language processing, computational linguistics and machine learning. It is fundamentally a study in comparative psycholinguistics, attempting to show how the results of algorithms for natural language grammar induction as applied to the language of individual speakers can be used to compare groups with known cognitive differences.

The paper will proceed as such: this first chapter will discuss the notions of language and grammar. Grammar induction methods will be discussed in Chapter 2, with Chapter 3 focusing on a particular method, the ADIOS (Automatic Distillation of Structure) algorithm (see, e.g., Solan et al. 2002; 2003a; 2003b; Horn et al. 2004).

The ADIOS algorithm for natural language grammar induction is discussed in detail in chapter 3, followed by a description of the experiments performed using the ADIOS algorithm on a natural language dataset in Chapter 4. Conclusions are drawn regarding the effectiveness of grammar induction, and potential uses and future work in linguistic measurement and psycholinguistic comparison are discussed in the final chapter.

1.1 What is a language?

A language, formally speaking, is a set of sentences, each generated as a sequential string from a given set of terminal symbols (Hopcroft & Ullman 1969). In natural human language, the terminal symbols are typically understood to be morphemes, the smallest linguistic unit that has meaning. However, human languages hardly adhere to formalities; a given human

```
string InfinitelyLongSentence()
{
    string sentence = "The boy is next to the tree";
    while(true)
    {
        sentence += " which is next to that other tree";
    }
    sentence += ".";
    return sentence;
}
```

Figure 1.1: C# Method for generating an infinitely long sentence: "The boy is next to the tree which is next to that other tree which is next..."

language is rarely, if ever, defined by such formal terms. While it would not necessarily be inaccurate to refer to the English language, for instance, as the set of all sentences ever spoken or written by English speakers, this is not an adequate definition of a language for laypersons, and certainly not for linguists. Human language, and indeed many formal languages, have implicit or explicit structure that guides or determines the formation of novel sentences. English, as such, is not simply a set of sentences, or a lexicon, or a set of morphemes; these elements alone cannot account for the production of novel sentences. Furthermore, human languages have structural guidelines (we will not, at the moment, call them *rules*) that theoretically allow for an infinite number of novel productions. Of course this notion is absurd in practice. Humans lack the cognitive abilities (and, frankly, the time) to produce or comprehend, for example, an infinitely long sentence. However, the fact that an infinitely long, meaningful sentence can be generated (see, e.g. Figure 1.1), shows that there must at least be such guidelines, or, perhaps, rules, that not only mediate between hierarchical logical structure and the sequential surface form of sentences, but also allow for recursive semantic hierarchies to be transduced into sequences of symbols.

1.2 What is a grammar?

A grammar is simply a representation or a description of a language. According to Chomsky (1965:24),

> A grammar can be regarded as a theory of a language; it is *descriptively adequate* to the extent that it correctly describes the intrinsic competence of the idealized speaker. The structural descriptions assigned to sentences by the grammar, the distinctions that it makes between well-formed and deviant, and so on, must, for descriptive adequacy, correspond to the linguistic intuition of the native speaker (whether or not he may be immediately aware of this) in a substantial and significant class of crucial cases.

The notion of an idealized speaker deserves review. For Chomsky, an idealized speaker-listener is one who has perfect knowledge of her language, lives in a homogenous speech community, and isn't burdened by factors such as distraction, limitations on memory, or errors (1965:3). This idealized speaker has a certain *intrinsic competence*, an idealized mental representation of the language as a grammar. Even the idealized speaker thus has a *theory of a langauge*, i.e., a finite symbolic model representing the potentially infinite surface forms of sentences. Linguistic competence, for Chomsky, is thus grammatical competence. The idealized speaker-hearer has the purest model, as it were, of the language of the homogenous speech community in which she lives.

A grammar is *descriptively adequate* insofar as it describes the idealized speaker's linguistic competence; a grammar thus describes a language only indirectly, through an imagined, perfect competence. This suggests that a language doesn't have a grammar per se, but that individual speakers have grammars; an individual speaker has a *theory or theories of language* that describe, with degrees of adequacy varying among speakers, the sets of sentences from the languages to which the speaker has been exposed in her lifetime.

An important point here is that a grammar is descriptive of linguistic input. The notion of a grammar as a theory highlights the fact that a grammar, a finite representation of a potentially infinite set of sentences in a language, must be induced from input data. Furthermore, this representation, as a theory, is constantly subjected to revision in order to account for new linguistic input. Hence Chomsky's notion of an idealized speaker-hearer; the ideal speaker-hearer lives in a static, unchanging speech community in which input data is structurally uniform. Even in an idealized situation, linguistic input must be primary, and the induction of an adequate grammar, or the acquisition of grammatical competence on the part of the speaker, derives from this primary input.

The automatic induction of adequate grammars from linguistic data has long been an issue in linguistics. Indeed, no perfectly adequate grammar has ever been formalized for any human language. Newmeyer (1986) describes the goal of the dominant, pre-Chomskyan empiricist approach to linguistics as such: "[T]o 'discover' a grammar by performing a set of operations on a corpus of data." As Newmeyer suggests, the Chomskyan revolution in linguistics of the late 1950s stemmed, at least in part, from the abandonment, or outright rejection, of this goal. With the advent and refinement of new machine learning techniques in the subsequent decades, and with the ever-increasing computational power available to researchers, grammar induction has been making a resurgence. At the same time, many linguists, psychologists, neuroscientists, cognitive scientists and other researchers are questioning the validity of the Chomskyan paradigm, and are seeking techniques for linguistic analysis that do not rely on Chomksy's assertions regarding the neuro-cognitive basis of language. It now seems possible to revisit the empiricist goal of automatic induction of linguistic structure from natural language data.

Chapter 2

Grammar Induction

2.1 The problem of natural language grammar induction

Grammar induction, also referred to as *grammatical inference*, is a non-trivial problem in linguistics and formal language theory. The problem is usually stated as such:

- Given a set of sentences that represent either positive and/or negative examples from a language,
- construct a grammar that generalizes in such a way that it can adequately describe the structure of new sentences,
- and can produce novel sentences that are acceptably included in the set of sentences belonging to the language.

There is at least one other criterion that cognitive scientists would find important in a grammar induction algorithm: that the algorithm itself, regardless of the form of the resultant grammar, is cognitively plausible. For linguists and other cognitive scientists, this criterion is perhaps the most important, and most dependent on the researcher's model of language and cognition. The Chomskyan model of human grammar presupposes an innate *Universal Grammar* (UG), one that accounts for all sentences of all human languages (Chomsky 1965). On Chomsky's model, grammars specific to individual languages are derived from universal templates which are encoded in the human genome; language acquisition thus proceeds by switching between various grammatical parameters based on input data.

For some researchers, this model presents a problem. Notably, Chomksy's proposed grammar module has yet to be found in the human brain, or in the human genome. Furthermore, Chomsky's model was intended to be explanatory; i.e., the notion of UG is supposed to explain linguistic phenomena such as language universals, rapid acquisition of language by human children, language learning windows, etc. However, many researchers find fault with Chomsky's explanation of these phenomena, and point to alternate explanations that tend to weaken the entire basis of UG (see, e.g. McFall 2004, Sampson 1997, Tomasello 1990 and Tomasello 1995 for reviews of the language instinct debate, and near-excoriations of the nativist position). [1]

The criterion of cognitive plausibility is thus dependent on the researchers conception of language. Given problems with Chomsky's innatist model, this paper will give more credence to methods that involve only minimal assumptions about what a resultant grammar should look like; that is, an algorithm will be considered more plausible if it does not require any built-in knowledge about a resultant grammar. Furthermore, the input data itself should be taken into consideration in order to determine cognitive plausibility.

2.2 Input data for grammar induction

The original problem is framed, in part, in terms of its input data. Some algorithms for grammar induction use both positive and negative examples of sentences from the target language as input; for natural langauge, this seems to fail the criterion of cognitive plausibility. Human children acquire language almost exclusively from positive examples; thus, an algorithm that induces a grammar strictly from positive examples should be more plausible than one that induces from both positive and negative examples.

[1] Other researchers, notably Lappin & Shieber (2007), point to machine learning techniques for grammar induction as vindications of UG.

2.3 Measuring the accuracy of a grammar induction algorithm

The two performance metrics of a grammar induction algorithm are typically referred to as *precision* and *recall*. Often, these measurements are combined into a single measurement, the *F measure*, which is the harmonic mean of precision and recall. The terms precision and recall are borrowed from the field of information retrieval. In information retrieval, precision is defined as "the proportion of selected items that the system got right," and recall is defined as "the proportion of the target items that the system selected" (Manning & Schütze 1999: 268-269).

Although used similarly in grammar induction, the meanings of these borrowed terms shift slightly to be relevant to the domain. According to Jurafsky and Martin, precision and recall, as used in information retrieval, are

> antagonistic to one another since a conservative system that strives for perfection in terms of precision will invariably lower its recall score. Similarly, a system that strives for coverage will get more things wrong, thus lowering its precision score. (2000:604)

This is not necessarily the case as these terms are used in grammar induction, because of the slightly different meanings given to them. Additionally, both of these concepts become more cumbersome with the introduction of the cognitive plausibility criterion, as noted below.

2.3.1 Recall

For grammar induction algorithms, recall is typically defined as the proportion of sentences from a target corpus that a candidate grammar accepts. Recall, alone, is clearly not an adequate measure of performance of an induced grammar. A grammar, for instance, could be constructed that will accept any string of randomly placed terminal symbols from a given alphabet; this is clearly undesirable if a target grammar is known or assumed to be structured non-randomly in some way. So even though such a grammar would accept all

positive examples from a target corpus, it would also accept all negative examples. This presents a problem for an algorithm that fits the criterion of cogntive plausibility; with no negative examples in the training data, there can be no false positives, which would typically factor into the recall measurement.

2.3.2 Precision

Precision, as a measurement of grammar induction accuracy, is perhaps even more problematic. Precision is often measured as the number of sentences generated from a candidate grammar that are accepted by a target grammar. While this is adequate for artificial languages, where a target grammar can be specified and a training corpus can be generated from the target grammar, in natural language grammar induction there are no existing target grammars. There are various ways to cope with this.

First, native speakers of the target language can participate as judges for sentences generated from a candidate grammar. This is clearly infeasible in many cases, especially if the algorithm used for generating candidate grammars uses evolutionary techniques in which the fitnesses of candidate grammars, each generating perhaps hundreds of sentences, need to be evaluated each generation. An evolutionary algorithm might run for thousands of generations, making it nearly impossible to use humans as judges of candidate grammar fitness.

Another way around the problem of precision measurement is to compare sentences generated from a target grammar to sentences in either the original target corpus or in a testing corpus of sentences from the same language. This tends to risk false negatives, where the candidate grammar generates a sentence that would be accepted by the hypothetical target grammar, but happen to not appear in the testing corpus.

2.4 Grammar Induction methods

There have been many different approaches to grammar induction. Grammar induction, as it has been practiced in computer science, has often been applied to artificial data sets, in

order to induce deterministic finite-state automata (DFA). This class of problems has often been referred to as *Abbadingo-style* problems, following a 1997 DFA learning competition (Cicchello & Kremer 2003). Abbadingo-style problems typically involve learning a grammar, as a DFA, from data generated from a random DFA, where the alphabet consists of 0s and 1s, and where the target DFA consists of any number of states. Cicchello and Kremer (ibid.) survey ten years of research into induction of DFAs from such data. Colin de la Higuera (2005) also offers a comprehensive study of research done with DFAs. Although much research has been done on Abbadingo-style problems, especially in the field of computer science, natural language grammar induction, a long-time goal of computational linguistics, has only recently been attempted with any degree of success.

Evolutionary Methods

Many evolutionary methods have been attempted for natural language grammar induction. Most notable has been Koza's (1992) genetic programming (GP) approach, and many similar approaches, which evolve trees for grammatical structure, using lexical items from raw text corpora as terminal nodes, and boolean values as non-terminal nodes. Koza's GP method uses a LISP representation of trees for evaluation of grammars on natural language text corpora, with some apparent success. Although Koza introduces this method in the work cited, he does not give much detail on its use in natural language grammar induction, devoting more time to its use in bioinformatics.

ABL (Alignment-Based Learning)

Alignment-based learning (ABL) (van Zaanen 2000) has its roots in Harris' (1954) notion of distributional structure. ABL consists of two phases: *alignment learning* and *selection learning*. The alignment learning phase consists of finding constituents in a corpus, while the selection phase selects the best constituents generated from the alignment learning phase.

The result probabilistically converges on sets of substitutable constituents which are thus represented as a grammar. For instance, in the two sentences:

- *What is a family fare?*
- *What is the payload of an African swallow?*

the phrases *a family fare* and *the payload of an African swallow* occur in the same syntactic context. ABL thus considers these phrases constituents of the same type.

EMILE

The EMILE algorithm (Adriaans 1992; Adriaans 1999; Vervoort 2000) is also based largely on Harris' distributional model. EMILE consists of two phases: *clustering* and *rule induction*, all based on the notions of *contexts* and *expressions*. The general idea is that expressions that occur in the same context can cluster together, and are thus substitutable in similar contexts. Given a sufficient corpus, we can expect to find classes of expressions that cluster together, and are then able to induce rules from their contexts.

A comparison of ABL and EMILE by van Zaanen and Adriaans (2001) showed that EMILE scores higher on precision and recall than ABL, while EMILE is the slower learner of the two. The authors note, however, that precision and recall are measurements that are designed for supervised methods, while both of these methods (and, in fact, most methods of natural language grammar induction) are unsupervised learning methods. The following chapter deals with the method of grammar induction on which this paper will focus: the ADIOS algorithm.

Chapter 3

The ADIOS algorithm

This chapter will take an in-depth look at a particular algorithm for grammar induction, the ADIOS algorithm.[1]

3.1 Description of ADIOS

The ADIOS (Automatic Distillation of Structure) model was proposed as a statistical method of grammar induction that yields symbolic results in the form of a context-free grammar (Solan et al. 2002). ADIOS thus spans the distinction between statistical and symbolic methods and representations. ADIOS was designed to handle raw text corpora, but can take any sequential, sentential data as input. As the name of the algorithm implies, the ADIOS model distills structure, automatically, from a corpus. The result is a context-free grammar that represents the structure of sentences at various hierarchical levels, the specifics of which will be discussed shortly. ADIOS is a data-driven model; that is, no strict assumptions are made about what an induced grammar should look like prior to the application of the algorithm to a corpus. It is also a probabilistic model, in that it searches for redundancy in a given corpus, detecting patterns probabilistically in order to reduce redundancy. As such, ADIOS should be applicable to any set of sentential data with a finite lexicon. The authors of the ADIOS algorithm have applied it to a variety of such problems. In addition to natural language corpora, ADIOS has been applied to corpora generated from artificial

[1]The ADIOS algorithm is notoriously difficult to fully comprehend. While the algorithm was used in the experiments described in this thesis, the description here is not an adequate substitution for the multiple papers written by the algorithm's authors themselves. Indeed, many find even the original ADIOS papers a bit opaque and difficult to grasp.

grammars (Solan 2006), as well as to enzyme protein sequences (Kunik et al. 2005), with results comparable to (and often better than) other established methods.

ADIOS was largely inspired, like the EMILE and ABL algorithms, by Zellig Harris' (1954) notion of *distributional structure*. Harris, in fact, was Chomsky's teacher, and was considered a major proponent of the Post-Bloomfieldian empiricist school of linguistics, against which Chomsky is seen to have rebelled (Newmeyer 1986). While Chomsky may have borrowed from Harris in many aspects of his linguistic theory, a major difference in their conceptions of language relate to acquisition; while Harris advocates an empirical, distributional model of discovery, Chomsky is known for his advocacy of an innate language acquisition device, with significant grammatical knowledge pre-programmed in the individual prior to language acquisition. ADIOS was thus formulated in line with Harris' distributional model, which suggests:

- "First, the parts of a language do not occur arbitrarily relative to each other: each element occurs in certain positions relative to certain other elements." (1954:146)
- "Second, the restricted distribution of classes persists for all their occurrences; the restrictions are not discarded arbitrarily, e.g. for semantic needs...All elements in a language can be grouped into classes whose relative occurrence can be stated exactly." (1954:146)
- "Third, it is possible to state the occurrence of any element relative to any other element, to the degree of exactness indicated above, so that distributional statements can cover all the material of a language, without requiring support from other types of information." (1954:146-147)
- "Fourth, the restrictions on relative occurrence of each element are described most simply by a network of interrelated statements, certain of them being put in terms of the results of certain others, rather than by a simple measurement of the total restriction on each element separately." (1954:147)

The influence of Harris' distributional structure on the formulation of the ADIOS algorithm becomes clear with a review of the algorithm itself.

3.2 Elements of the ADIOS algorithm

ADIOS depends on a set of hierarchical elements, each of which are discussed below.

3.2.1 Graph

The ADIOS algorithm initially represents a corpus as a directed multigraph, with the terminal nodes of the graph initially representing the smallest salient unit in the corpus, i.e., words, phonemes, morphemes, part-of-speech tags, proteins, etc., depending on the problem. Individual nodes are joined by edges; the graph thus becomes a set of "bundles" of nodes and edges (Edelman et al. 2003). The graph contains a BEGIN node and an END node, between which all paths in the graph are connected with edges.

3.2.2 Paths

A sentence in the graph is represented by a *path*. A path is a sequence of nodes, beginning with the BEGIN node and ending with the END node. As the algorithm progresses, nodes in the graph, and thus in the paths of the graph, are replaced by *patterns* and *equivalence classes* (i.e., with non-terminal nodes), compressing the paths until a stopping criterion is reached.

3.2.3 Patterns

Patterns are sequences of nodes. The nodes can be terminal nodes, equivalence classes or other patterns. ADIOS distills patterns probabilistically from the graph, reducing redundancy in paths by replacing terminal nodes with non-terminal patterns.

```
1: Initialization (load all sentences)
2: repeat
3:    for all m = 1:N do {N is the number of paths in the graph}
4:       Pattern Distillation(m) {Identifies new significant patterns
         in search path m using MEX criterion}
5:       Generalization(m) {Generate new pattern candidates for search path m}
6:    end for
7: until no further significant patterns are found
```

Figure 3.1: The ADIOS algorithm. See Figures 3.2, 3.3 and 3.4 for pseudocode detailing Initialization, Pattern distillation and Generalization. Adapted from Solan (2006:34)

```
1: for all sentences m = 1:N do {N is the number of sentences in the corpus}
2:    load sentence m as a path onto a pseudograph whose vertices
      are the unique words in the corpus
3: end for
```

Figure 3.2: The Initialization algorithm. Adapted from Solan (2006:35)

3.2.4 Equivalence classes

ADIOS distills sets of terminal and non-terminal nodes that occur in the same contexts, and considers each set as a class of equivalent nodes. Terminal nodes in the graph are replaced by equivalence class nodes as the algorithm proceeds, further compressing the graph across paths.

3.3 The algorithm

The algorithm has three phases: *Initialization*, *Pattern Distillation* and *Generalization*. Figure 3.1 shows pseudocode for the general algorithm.

```
1: for all m = 1:N {N is the number of paths in the graph}
2:    for all i = 1:LENGTH(m) do
3:       for j = i+1:LENGTH(m) do
4:          Find the leading significant pattern by performing
            MEX on the search segments(i,j).  Choose the leading
            pattern P for the seach path
5:          Rewire the graph, creating a new node for P
6:          Mode A: replace the string of nodes comprising P with the
            new node P
7:          Mode B: replace the string of nodes comprising P only
            in paths in which P is significant according to MEX criterion
8:       end for
9:    end for
10:end for
```

Figure 3.3: The Pattern Distillation procedure. Adapted from Solan (2006:36)

3.3.1 Phase I: Initialization

Initialization (Figure 3.2) involves loading all sentences (*paths*) into the graph. Nodes are catalogued and edges are created between nodes to create a graph that represents all sentences in the corpus. The next two phases are repeated until no new significant patterns are found in the graph.

3.3.2 Phase II: Pattern Distillation

After all paths have been loaded onto the graph, the algorithm searches for "sub-paths that are shared by a significant number of partially-aligned paths" using a motif-extraction (MEX) algorithm(Figure 3.3) (Solan 2006:34). Patterns are extracted by finding sub-paths of high probability, using the number of edges entering in and exiting from a sub-path as factors in the calculation of this probabilty. Thus, significant "bundles" of node sequences are extracted as candidate patterns.

```
1: For a given search path m
2: for i = 1:(LENGTH(m) - L - 1) do {L is the size of the context window}
3:    slide a context window of size L along the length of the search path
4:    examine the generalized search paths
5:    for j = i + 1:(i + L - 2) do
6:       define a slot at j
7:       define the generalized path consisting of all paths with same prefix
         and same suffix
8:       perform MEX on the generalized path
9:    end for
10:end for
11:Choose the leading P for all searches performed on each generalized path
12:For the leading P define an equivalence class E consisting of all the vertices
   that appeared in the relevant slot at location j of the generalized path
13:rewire the graph
```

Figure 3.4: The Generalization procedure. Adapted from Solan (2006:37)

3.3.3 Phase III: Generalization

The generalization phase (Figure 3.4) finds the most significant pattern, then generalizes over the graph by creating equivalence classes from all of the variable node elements in the pattern. For instance, if a pattern [the x man] is discovered as significant, all of the nodes that can fill the x slot are assigned to an equivalence class. The graph is then rewired by replacing sequences of nodes with their equivalent patterns.

3.3.4 Parameters

The ADIOS algorithm depends on multiple parameters. Zolan (2006) references the two most important parameters as η and α, described below.

Eta (η)

Solan writes that the motif extraction algorithm (MEX) "is designed to find patterns such that different paths converge onto them (at the beginning of the pattern) and diverge from them (at the end)" (2006:65). While looking for candidate patterns as sub-paths, the motif extraction algorithm looks for a *decrease ratio* of sub-sequences of the given subpath. That is, the outside nodes of a candidate pattern have certain probabilities associated with incoming and outgoing edges. The ratio of these probabilities decreases as the number of edges going into or out of the given node increase, thus indicating the boundary of a potentially significant pattern. The η value represents a "cutoff parameter" for the decrease ratio; if the decrease ratio for a given sub-path is less than η, the sub-path is considered a candidate pattern.

Alpha (α)

The α value represents a significance level for the decrease ratio. A significance test for the decrease ratio is performed based on binomial probabilities. If the significance is not less than α, the pattern is rejected as a candidate pattern. The candidate pattern with the most significant decrease ratio thus becomes the leading pattern, which is ultimately rewired into the graph.

3.4 Case studies

While ADIOS has been applied to non-linguistic data with some apparent success (see, e.g. Kunik et al. 2005 for ADIOS applied to protein classification), this section will focus on its application to both artificial and natural language corpora.

3.4.1 Artificial grammars

Artificial grammar experiments are typically a good initial test for the adequacy of a grammar induction algorithm. This is due to the fact that a target grammar can be explicitly specified, and both precision and recall can be tested without difficulty; new sentences to test recall

are easily generated by the target grammar, and the target grammar can be used to test precision on sentences generated by the candidate grammar. Solan (2006) tested the ADIOS algorithm on 2000 sentences generated from a context-free grammar which used 29 terminal nodes and 7 rules. He found that algorithm produced 28 patterns and 9 equivalence classes, achieving 99% recall and 100% precision upon testing.

3.4.2 The CHILDES corpus

ADIOS was trained on the CHILDES corpus (MacWhinney & Snow 1985), a corpus of conversations between parents and their young children (Solan 2006). The resultant grammar was then used in several tests of English as a Second Language (ESL) grammaticality judgements. In one of these tests, the Göteborg multiple-choice ESL test, presents 100 sentences to the test-taker, each with one blank slot in the sentence, requiring him or her (or it, as it were) to choose from three choices to fill the blank. On this test, ADIOS scored 60%, which is the average for 9th grade ESL students (Solan 2006:60).

3.4.3 Cross-language syntactic comparison

ADIOS has also been used for cross-linguistic syntactic comparison. Solan (2006) applied the algorithm to six different, parallel translations of 31,100 verses of the Bible, in English, French, Spanish, Danish, Swedish and Chinese. The resultant grammars were compared by counting the combinations of elements making up resultant patterns. A pattern can consist of any combination of terminal nodes, equivalence classes or other patterns. By comparing resultant grammars from the different languages in terms of the combinations of elements in patterns, it was found that judgements could be made regarding the similarity of the languages tested. So, it was found in this study that Chinese was the most different from the other five (European) languages, and that Spanish was closest to French, and Danish closest to Swedish, as would be expected from any method of comparative syntax (Solan 2006; Horn et al. 2004).

3.5 Conclusion

Given the apparently promising results of the ADIOS algorithm, the ADIOS algorithm was used in the subsequent experiments in grammatical comparison, as documented in the following chapter.

Chapter 4

Experiments

4.1 Overview

This chapter will describe the experiments implemented in this study, as well as their results and interpretations. First, the data, including the results of previous studies on the same data, will be described and discussed. Then, the experimental methods will be outlined in detail, followed by a discussion of the results obtained in the experiments.

4.2 The data

4.2.1 Description of data

The corpus used was gathered by the Georgia (UGA) Regents' Center for Learning Disorders from a sample of college writers. Four groups were identified from the samples: one group was evaluated with learning disability (LD, n=87), one with attention-deficit hyperactive disorder (ADHD, n=50), one group with both LD and ADHD (Comorbid, n=58) and one normal control group (Normal, n=92).

The corpus consisted of essays written by the participants about one of four topics. The participants were given 30 minutes each to write the essay.

4.2.2 Results from Gregg, et al. (2002)

The Learning Disorders corpus was originally analyzed by Gregg et al. (2002). Gregg et al. analyzed the discourse complexity of the essays, finding that there were significant differences between the learning disabled groups (LD, ADHD and Comorbid) and the Normal group

in terms of verbosity and type-token ratio. The differences in verbosity (i.e., token counts) and type-token ratio are to be expected; type-token ratio is related to token counts, so these measurements would certainly be expected to correlate. Furthermore, in an analysis of five factors contributing to the quality and complexity of discourse from each group, it was found that "all the groups of writers tended to use predominantly the same dimensions (factors) in their writing; the difference was in the degree to which specific features contributed to the factors" (Gregg et al. 2002:35).

4.3 Setup

To prepare the data for running the ADIOS algorithm, each essay was first tagged for parts of speech using Hugo Liu's (2004) MontyLingua software. While ADIOS does not require the use of part-of-speech (POS) tags, it does require samples of sufficient size to achieve even minimal results. Thus, only the part of speech strings were used in the grammar induction phase of the experiments due to the variability in lexical types used across the essays. Each file was converted to sentences of POS tags only, and prepared appropriately for the ADIOS software.

The software used was a custom built, C# port of a Java version of ADIOS found on the internet (Ruppin 2006).[1] The Java version of ADIOS was ported to a C# class library, and enabled to run batches of files with multiple parameters.

ADIOS was then applied to each participant essay, using six values each for α and η (0.05,0.25,0.45,0.65,0.85,0.95). Each file was thus processed by the ADIOS algorithm under 36 different parameter settings, and produced files containing resultant grammars, as a set of patterns and equivalence classes, for each run of the algorithm. Files were also generated with the original sentences reduced to their minimal paths, as generated by the algorithm. In other words, the terminal nodes of each original sentence were replaced with their corresponding patterns and equivalence classes, and these paths were printed to a separate file.

[1] See Appendix for selected code examples.

4.4 Method

4.4.1 Measurements

After running the ADIOS algorithm on each file as described above, the resulting files were then processed in order to take the measurements described here.

Compression

One way to test the performance of a grammar is to measure how well it is able to compress terminal nodes in a sentence and represent each sentence as a string of non-terminal nodes. This was measured by dividing the number of nodes in the ADIOS paths files by the original number of words. If any of the original terminal nodes, or POS tags in this case, are replaced by patterns, then the total number of nodes will be less than the original number of nodes. Compression is thus understood as a percentage, and, furthermore, a lower score means a greater compression. So, for instance, a compression score of 0.75 means that the resulting paths are 75% of their original lengths after compressing the original paths using the grammar generated by the algorithm.

Number of patterns and equivalence classes

The number of patterns and equivalence classes generated was also counted for each output file. The pattern/equivalence class counts were seen as a measure of grammatical consistency; if ADIOS was able to distill any structure at all from a given essay, then there must have been patterns that have at least met the criteria of the α and η parameters, as described in the previous chapter. If patterns and equivalence classes were induced, then there was some consistency in the use of grammar in the sample.

α	Compression	Equivalence Classes	Patterns
0.05	0.9071	1.1456	2.6539
0.25	0.8541	2.3088	4.8344
0.45	0.8538	2.2557	4.8627
0.65	0.8545	2.2742	4.8540
0.85	0.8541	2.2846	4.8546
0.95	0.8520	2.3538	4.9198

Table 4.1: Effects of the α parameter on the measurements performed here. The least-squares mean is shown, representing the mean by α after other effects are controlled

Pattern types

The types of patterns, in terms of terminal/non-terminal node sequences, were also counted and compared, following Solan's (2006) method of comparing grammars from different languages.

4.5 Results

[2]

4.5.1 Parameter effects

The first analysis perfomed on the resulting data was an analysis of the effects of α and η on the compression, number of patterns and number of equivalence classes generated from the data. Table 4.1 shows the least squares means of these variables by the α parameter. As α increases, compression increases (i.e., the compression score decreases), and the numbers of patterns and equivalences classes increase.

[2]The data analysis and all graphs for this paper were generated using SAS software. Copyright, SAS Institute Inc. SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc., Cary, NC, USA.

η	Compression	Equivalence Classes	Patterns
0.05	0.9975	0.0389	0.0426
0.25	0.9368	0.2067	1.4959
0.45	0.8700	0.7840	3.9503
0.65	0.8006	3.1127	6.8840
0.85	0.7929	3.5920	6.9578
0.95	0.7778	4.8884	7.6487

Table 4.2: Effects of the η parameter on the measurements performed here. The least-squares mean is shown, representing the mean by η after other effects are controlled

Group	Mean Words	Mean Sentences
ADHD	322.7600	17.1000
LD	327.2697	16.955
Comorbid	316.1552	16.2414
Normal	386.4022	20.1522

Table 4.3: Word and sentence count; means by group

Table 4.2 show similar increases as η increases. However, there seems to be a significant increase when η=0.65 in all of the dependent variables.

4.5.2 Group effects on word and sentence count

There were also group effects found on the number of words in the original sample essays. These analyses were performed by Gregg et al. (2002), and the analysis here agrees with the results of that paper. Table 4.3 shows the means of word and sentence counts by group. Figure 4.1 shows a boxplot of the number of words by group, and Figure 4.2 shows a boxplot graph of the number of sentences by group. Table 4.4 shows the significance levels of multiple t-tests perfomed, comparing the number of words by group. These results show, in agreement

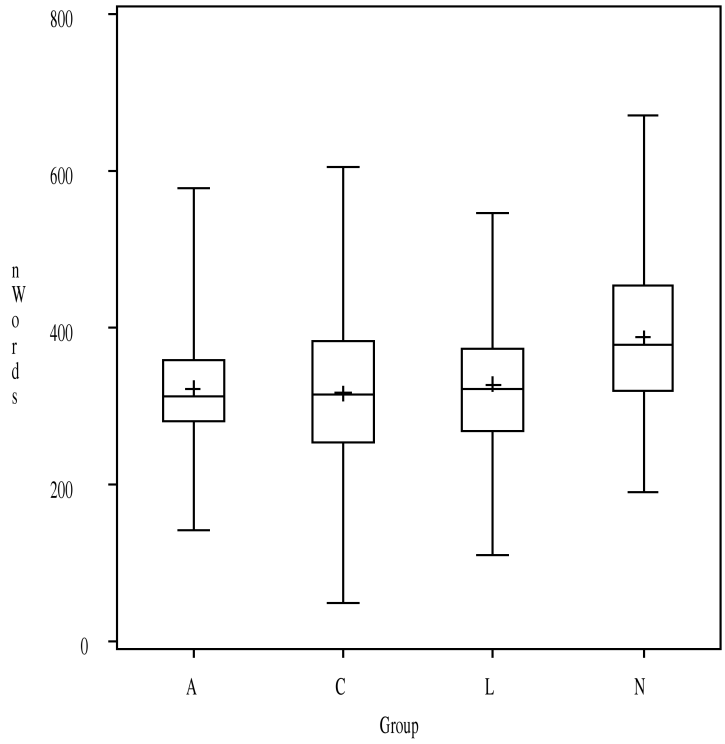

Figure 4.1: Boxplot depicting the number of words by group

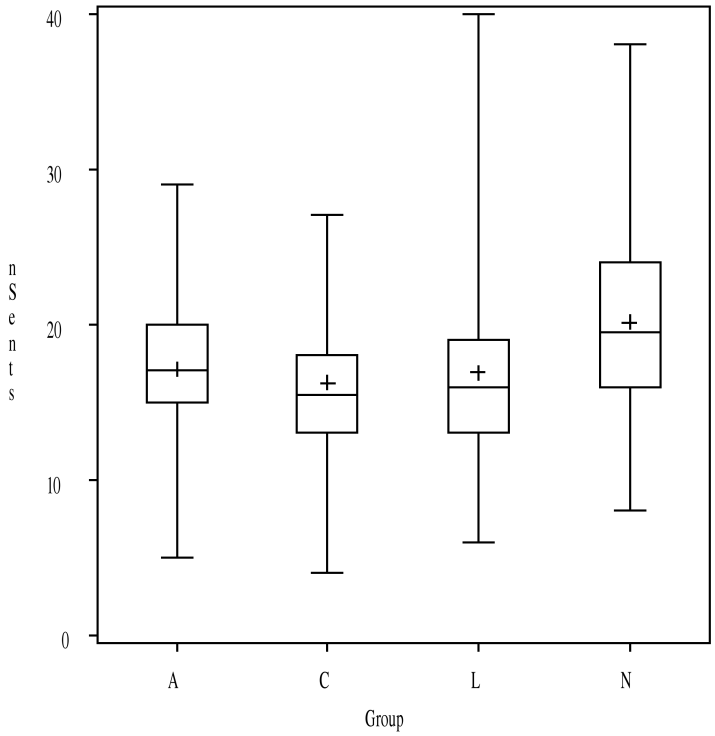

Figure 4.2: Boxplot depicting the number of sentences by group

Group	LD	Comorbid	Normal
ADHD	1.0000	1.0000	**0.0051**
LD		1.0000	**0.0012**
Comorbid			**0.0004**

Table 4.4: Multiple two-tailed t-tests comparing word counts by group, showing p-values (probability that groups are not different), using Bonferroni correction to control for multiple tests. $p<0.05$ bolded

	Compression	Eq.Classes	Patterns
p-value	<0.0001	0.0083	0.0058

Table 4.5: ANCOVA p-Values for group effects on the measurements after effects of covariates removed

with Gregg et al., that the three learning-disabled groups are not different from each other, but all three are significantly different from the normal group in terms of the number of words (cf. *verbosity* in Gregg et al.).

4.5.3 ANCOVA: Group effects

Further exploration of the data showed that there were general group effects on each of the three measurements. An analysis of covariance (ANCOVA) was performed to test for group effects on each of the measurements after removing the variance contributed by the number of words, number of sentences, and the α and η variables. Table 4.5 shows the p-values of group effects for each of the measurements after performing an ANCOVA.

Table 4.6 shows the least squares means of each of the three measurements by group. While significance is not measured in this statistic, the normal group shows a greater com-

Group	Compression	Eq.Classes	Patterns
ADHD	0.8666	2.0594	4.4014
Comorbid	0.8618	2.0482	4.5089
LD	0.8638	2.1032	4.4599
Normal	0.8582	2.2044	4.6161

Table 4.6: Least squares means of each of the measurements by group after effects of covariates removed

pression, a greater number of equivalence classes and a greater number of patterns than the other three groups. These results seem promising given that group was shown to have a significant effect on all of the dependent variables in the analysis of covariance. However, given that there were group effects on word count as well, it was deemed necessary to explore the effects of word count on the dependent variables.

4.5.4 Word count effect on measurements

Word counts were found to be correlated with each of the dependent variables. Figures 4.3, 4.4 and 4.5 show scatterplots of compression, number of equivalence classes and number of patterns, respectively, by number of words. Regression lines are shown in each of the figures, depicting the relationships between word counts and the dependent variables. Compression, number of equivalence classes and number of patterns all increase as the number of words in a sample increase.

This finding is not surprising, since the ADIOS algorithm is a data-driven, statistical method; more data should improve the results significantly regardless of the source. This does present a few problems, however: the dependent variables are influenced by the number of words, group is a predictor of number of words, and the number of participants in each group is not matched across groups. It was deemed necessary to reduce the effects of these potential confounding factors by matching the groups for word counts and group sizes.

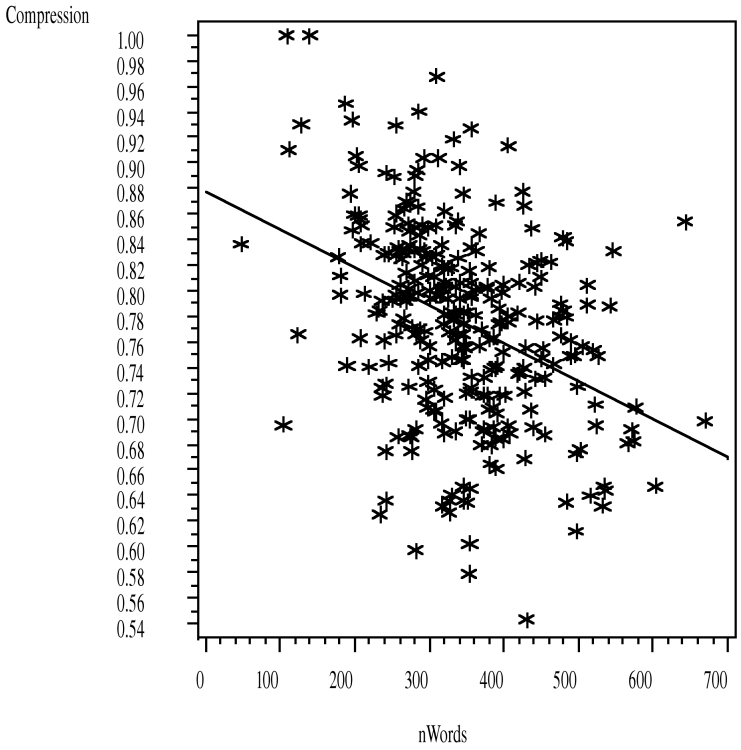

Figure 4.3: Compression by number of words, with regression line

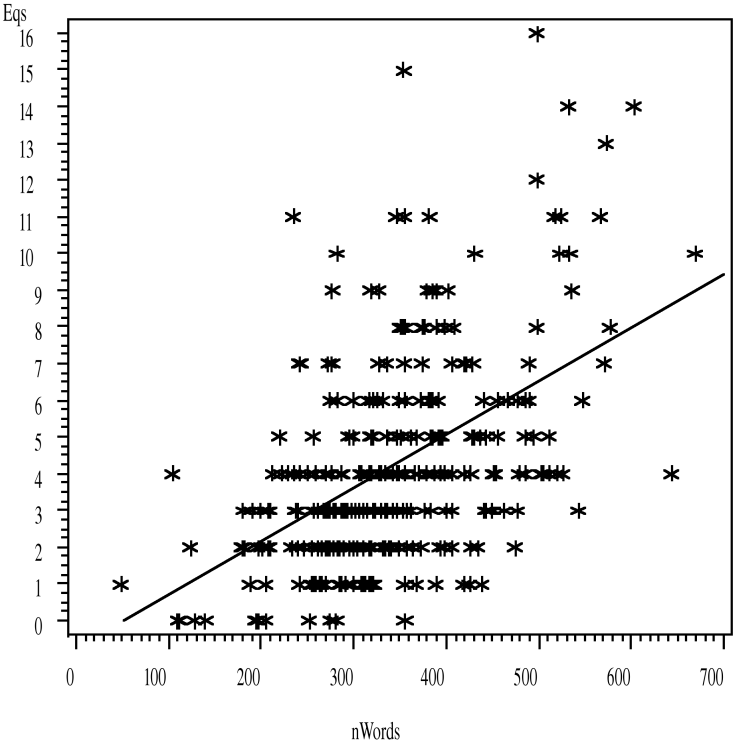

Figure 4.4: Number of equivalence classes by number of words, with regression line

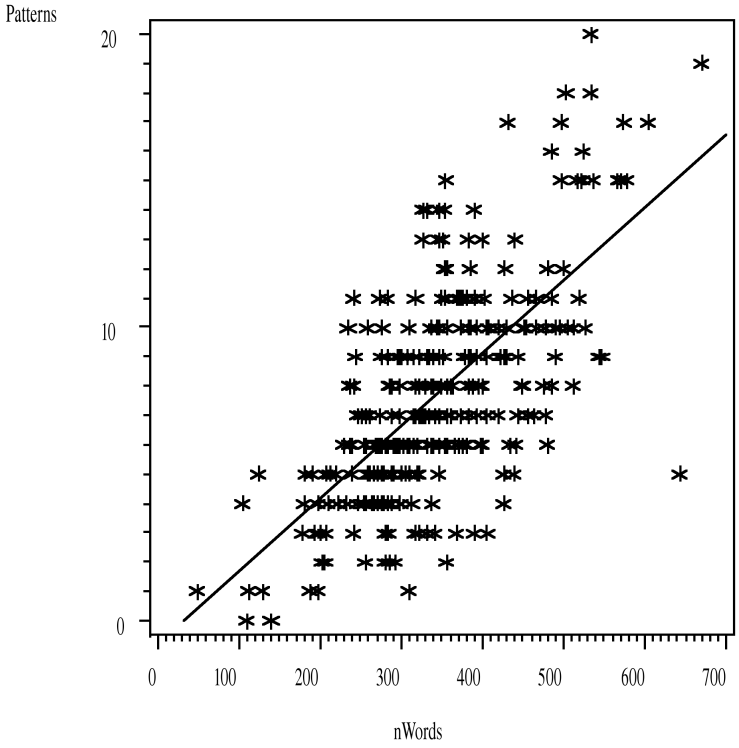

Figure 4.5: Number of patterns by number of words, with regression line

Group	LD	Comorbid	Normal
ADHD	1.0000	1.0000	1.0000
LD		1.0000	1.0000
Comorbid			1.0000

Table 4.7: Multiple two-tailed t-tests comparing word counts by group after matching participants by word count, showing p-values (probability that groups are not different), using Bonferroni correction to control for multiple tests. $p<0.05$ bolded

	Compression	Eq.Classes	Patterns
p-value	<0.0001	0.0002	0.00007

Table 4.8: ANCOVA p-Values for group effects on the measurements after effects of covariates removed

Group	Compression	Eq.Classes	Patterns
ADHD	0.8708	1.9275	4.1135
Comorbid	0.8670	1.8686	4.1666
LD	0.8706	1.9914	4.1806
Normal	0.8611	2.1092	4.3809

Table 4.9: Least squares means of each of the measurements by (matched) group after effects of covariates removed

4.5.5 Matched vs. Unmatched data

Matching was performed by iterating through the group with the fewest participants (ADHD), and selecting the participant from each group with the closest number of words to the selected participant from the ADHD group that was not already selected for inclusion. Whereas the unmatched groups showed a strong effect of group on word counts (see Table 4.4), the matched groups showed no such effect, as shown in Table 4.7.

4.5.6 General group effects on matched data

An ANCOVA was re-run on matched data, which showed general group effects on all of the dependent variables. Compare Tables 4.8 and 4.5, which show the group effects, from the matched and unmatched data, respectively, on the dependent variables after the effects of covariates have been controlled. Table 4.9 shows the least squares means of the dependent

DV	α	η
Compression	0.95	0.85
Eq.Classes	0.25	0.95
Patterns	0.65	0.95

Table 4.10: Alpha and eta values used for each dependent variable

variables by group. The normal group still shows greater compression, a greater number of equivalence classes and a greater number of patterns generated than the other three groups, although, again, significance was not tested for these effects.

4.5.7 Choosing α and η values

The data were further divided for statistical tests on particular measurements. To control for the effect of α and η on the dependent variables, particular α and η values were chosen for each dependent variable based on the variance of the variable. It was supposed that measurements with greater variances under particular α and η settings would more prominently show group effects; that is, with matched data, and with the α and η variables uniform across the data for a particular dependent variable, variance would be more likely due to group effect rather than other effects. Table 4.10 shows the values used for each dependent variable.

4.5.8 Group effects on measurements

Since the matched dataset showed general group effects, the divided data sets for each dependent variable were analyzed for individual group effects. Multiple two-tailed t-tests were run, comparing groups for each dependent variable. Both Bonferroni and bootstrapping corrections for multiple tests were performed on the outcomes.

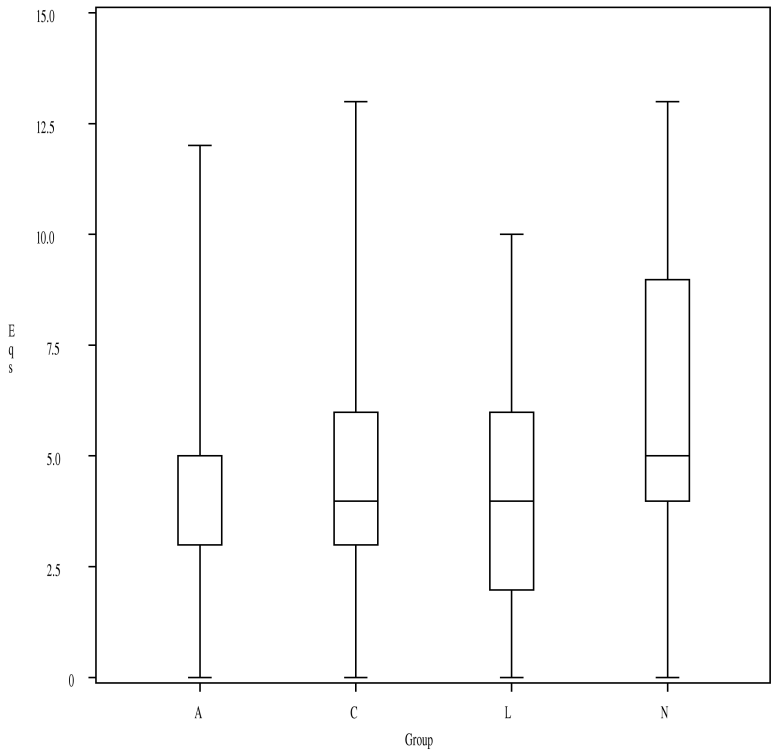

Figure 4.6: Boxplot depicting number of equivalence classes generated by group

Compression

An ANCOVA on the divided data showed no group effect on compression when α=0.95 and η=0.85. While the raw p-values showed significant differences between the normal and ADHD groups (p=0.0457), these differences disappeared with both Bonferroni and bootstrapping corrections.

Number of Patterns

An ANCOVA on the divided data showed no significant group effect on the number of patterns generated when α=0.65 and η=0.95. No significant differences were found between the groups, even before correction for multiple tests.

Group	LD	Comorbid	Normal
ADHD	1.0000	1.0000	**0.0285**
LD		1.0000	**0.0059**
Comorbid			0.1179

Table 4.11: Multiple two-tailed t-tests comparing equivalence classes by group, showing p-values (probability that groups are not different), using Bonferroni correction to control for multiple tests. $p<0.05$ bolded

Number of Equivalence classes

An ANCOVA on the divided data when α=0.25 and η=0.95 showed a significant group effect on the number of equivalence classes generated. Figure 4.6 depicts a boxplot of number of equivalence classes generated by group. Table 4.11 shows that the normal group is significantly different from the ADHD and LD groups. While the raw p-values show that the normal group is also significantly different from the comorbid group (raw p=0.0042), this significance weakened upon correction for multiple tests (corrected p=0.1179). There were no differences between the other three groups.

Pattern Types

An analysis similar to the cross-linguistic comparison of within-pattern sequence types in Solan (2006) was performed on the data. No significant differences were found.

4.6 Discussion

While the ADIOS algorithm was able to distill patterns from the writing samples, no complete grammars were induced from the data. This is likely due to the sample sizes, especially in terms of the number of words and the number of sentences in a given sample. While altering the α and η parameters increased the compression, number of equivalence classes

and the number of patterns generated from the data, the ADIOS algorithm itself is sensitive to corpus size, and thus was unable to completely induce a full grammar from such sparse samples.

Taking this into consideration, the comparison of the resultant grammars did seem to show some interesting results. Following Gregg at al. (2002), there were some significant differences found between the three learning-disabled groups and the normal group. Notably, after matching the data for word counts, there were effects on the number of equivalence classes generated that are less likely to be influenced by word counts than in the unmatched data. The counts of patterns and equivalence classes were intended to be measures of grammatical consistency. If this interpretation of this variable holds, then it can be concluded that the three learning disabled groups use structure less consistently than the normal group. While the comorbid group did not meet the level of significance (p=0.05) upon Bonferroni correction, it is understood that this kind of correction is typically very conservative, and it should not be dismissed outright that the comorbid group shows no differences from the normal group; indeed, there were virtually no differences between the three learning-disabled groups in terms of this measurement.

Chapter 5

Conclusion

5.1 Discussion

ADIOS as a Grammar Induction algorithm

Given the limited work done in natural language grammar induction, the ADIOS algorithm seems promising. Since much of the research in grammar induction has been in artificial and formal langauges, and much of it does not incorporate cognitive and psycholinguistic models, the ADIOS algorithm presents a method that meets the cognitive plausibility criterion suggested in the introduction to this paper.

The ADIOS algorithm induces grammars from raw corpora, using only positive examples in an unsupervised fashion. This is clearly desirable in an algorithm that should not only infer grammars but also meet the requirement of cognitive plausibility. The ADIOS algorithm, as a statistical method that yields rule-based results, meets this criterion.

The algorithm itself has a broad range of potential applications, natural language grammar induction being but one; its application to natural lanaguge, however, seems to surpass other methods both in terms of its effectiveness and its its accuracy. A good grammar induction algorithm has many potential uses in machine translation, information extraction, and other areas of natural language processing and computational linguistics.

Comparing grammars for psycholinguistics

While results here show some promise, more effective ways of comparing grammars need to be developed. One shortcoming of the experiments above was the limitation placed by

the data sets. Short essays are clearly not enough for the induction of robust, generalizable grammars. Furthermore, without such generalizable grammars, comparison of candidate grammars seems almost fruitless. Unfortunately, limited data sets limit the results, and this may have been the case with the results from the experiments outlined in the previous chapter.

5.2 Future work

There are many directions that could be taken for future work. Inducing a more generalizable grammar could be acheived with a larger data set. Furthermore, with a larger corpus, a grammar could be induced from the raw corpus, as opposed to sequences of part-of-speech tags. Such a grammar could have uses in psycholinguistic comparisons, as has been attempted in this paper.

The ADIOS algorithm could also be applied to machine translation and information extraction. A grammar represents the logical relations and dependencies from natural language, and, with a sufficiently descriptive grammar, these relationships could be extracted from raw text and represented in logical form. This application of the algorithm could very well be applied to machine translation, and to the creation of knowledge bases from large corpora.

For linguists, the automatic induction, or "discovery," as the Post-Bloomfieldian linguists would put it, of a natural language grammar, is an end in itself. If the ADIOS algorithm, or any other grammar induction method, lives up to these expectations, then its uses are potentially inexhaustible.

Bibliography

Adriaans, P. W. (1992). *Language Learning from a Categorial Perspective*. PhD thesis, Universiteit van Amsterdam.

Adriaans, P. W. (1999). Learning shallow context-free languages under simple distributions. *ILLC Report PP-1999-13*. Institute for Logic, Language and Computation, Amsterdam.

Chomsky, N. (1965). *Aspects of the theory of syntax.* Cambridge, MA: The MIT Press.

Cicchello, O. & Kremer, S. C. (2003). Inducing grammars from sparse data sets: a survey of algorithms and results. *Journal of Machine Learning Research*. 4:603-632.

Edelman, S., Solan, Z., Horn, D. & Ruppin, E. (2003). Rich Syntax from a raw corpus: unsupervised does it; a position paper presented at Syntax, Semantics and Statistics; a NIPS-2003 workshop, Whistler, BC, Dec. 2003.

Gregg, N., Coleman, C., Stennett, R. B., & Davis, M. (2002). Discourse complexity of college writers with and without disabilities. *Journal of Learning Disabilities*. 35(1):23-38,56.

Harris, Z. S. (1954). Distributional structure. *Word* 10: 146-162.

de la Higuera, C. (2005). A bibliographical study of grammtical inference. *Pattern Recognition*. 38:1332-1348.

Hopcroft, J. E. & Ullman, J. D. (1969). *Formal Languages and their relation to automata.* Reading, MA: Addison-Wesley.

Horn, D., Solan, Z., Ruppin, E., & Edelman, S. (2004). Unsupervised language acquisition: syntax from plain corpus. presented at the *Newcastle Workshop on Human Language*, Feb. 2004.

Koza, J. R. (1992) *Genetic Programming: On the Programming of Computers by the Means of Natural Selection*. Cambridge, MA: The MIT Press.

Kunik, V., Solan, Z., Edelman, S., Ruppin, E. & Horn, D. (2005) Motif extraction and protein classification. CSB, Aug. 2005.

Lappin, S. & Shieber, S. M. (2007). Machine learning theory and practice as a source of insight into Universal Grammar. *Journal of Linguistics*. 43:393-427.

Liu, H. (2004). *MontyLingua*. Web: http://web.media.mit.edu/∼hugo/montylingua/.

MacWhinney, B. & Snow, C. (1985). The Child Language Exchange System. *Journal of Computational Linguistics*. 12:271-296.

Manning, C. D. & Schütze, H. (1999). *Foundations of statistical natural language processing.* Cambridge, MA: The MIT Press.

McFall, J. D. (2004). *Semantic structure and the consequences of complexity: the evolutionary emergence of grammar.* Unpublished masters thesis, The University of Georgia, Athens, GA.

Newmeyer, F. J. (1986). *Linguistic theory in America.* Orlando, FL: Academic Press.

Ruppin, E. (2006). *JAdios*. NLP Spring Workshop 2006. Tel Aviv University. Web: http://www.cs.tau.ac.il/∼sandban/nlp_workshop_spring_2006.htm

Sampson, G. (1997). *Educating Eve: the 'language instinct' debate.* London: Cassell.

Solan, Z. (2006). *Unsupervised learning of natural languages.* Unpublished doctoral thesis, Tel Aviv University, Tel Aviv, Israel.

Solan, Z., Ruppin, E., Horn, D., Edelman, S. (2002). Automatic acquisition and efficient representation of syntactic structures. NIPS-2002.

Solan, Z., Horn, D., Ruppin, E., & Edelman, S. (2003a) Unsupervised efficient learning and representation of language structure. in *Proc. CogSci-2003*. Boston, MA.

Solan, Z., Horn, D., Ruppin, E., & Edelman, S. (2003b) Unsupervised context sensitive language acquisition from a large corpus. in *Proc. NIPS-2003*.

Tomasello, M. (1990). Grammar yes, generative grammar no. Peer commentary on Pinker and Bloom (1990), in *Behavioral and Brain Sciences* 13: 759-760.

Tomasello, M. (1995). Language is not an instinct. *Cognitive Development* 10: 131-156.

van Zaanen, M. (2000). ABL: Alignment-based learning. *Proceedings of the 18th International Conference on Compuational Linguistics COLING*. 961-967.

van Zaanen, M. & Adriaans, P.W. (2001). Comparing two unsupervised grammar induction systems: Alignment-based learning vs. EMILE. *University of Leeds, School of Computing Research Report Series, Report 2001.05*.

Vervoort, M. R. (2000). *Games, Walks and Grammars*. PhD thesis, Universiteit van Amsterdam.

Appendix A

Appendix A: Selected code

This appendix contains selected code from the C# port of JAdios, a Java version of ADIOS found on the website for the NLP Spring Workshop 2006 at Tel Aviv University. http://www.cs.tau.ac.il/∼sandban/nlp_workshop_spring_2006.htm

The code is largely adapted, including the code comments, from JAdios, and was changed to make use of features of C# that are not found in Java (e.g., generic collections).

A.1 Graph class

```
using System;
using System.Collections.Generic;
using System.Text;
using JAdiosPort.Helpers;
using System.IO;
using Path = JAdiosPort.Model.Path;

namespace JAdiosPort.Model
{
    public class Graph
    {
        private static Graph instance = null;

        public INode BeginNode = new Node("**");

        public INode EndNode = new Node("##");

        public Dictionary<string, INode> nodes;

        private SortedVector<Pattern> patterns;

        private SortedVector<EquivalenceClass> equivalenceClasses;

        private List<Path> paths;
```

```
private List<Edge> edges;

private int initialSize;

private Graph()
{
    this.nodes = new Dictionary<string, INode>();
    this.patterns = new SortedVector<Pattern>
        (new SortedVectorComparer<Pattern>());
    this.equivalenceClasses = new SortedVector<EquivalenceClass>
        (new SortedVectorComparer<EquivalenceClass>());
    this.paths = new List<Path>();
    this.edges = new List<Edge>();
    nodes.Add(BeginNode.Label, BeginNode);
    nodes.Add(EndNode.Label, EndNode);
    this.initialSize = -1;

}

public static Graph GetSingleInstance()
{
    if (instance == null)
    {
        instance = new Graph();
    }
    return instance;
}

public Graph Reset()
{
    instance = new Graph();
    return instance;
}

public INode GetBeginNode()
{
    return BeginNode;
}


public INode GetEndNode()
{
    return EndNode;
}

public int CountEdges()
```

```
{
    int count = edges.Count - paths.Count;
    return count;
}

public int CountPaths()
{
    return paths.Count;

}

public int CountNodes()
{
    return nodes.Count;
}

public int CountPatterns()
{
    return patterns.Count;
}

public int CountEquivalenceClasses()
{
    return equivalenceClasses.Count;
}


public void RegisterPattern(Pattern p)
{
    this.patterns.Add(p);
}

public void RegisterEquivalenceClass(EquivalenceClass ec)
{
    this.equivalenceClasses.Add(ec);
}

public INode GetNode(string label, bool createIfNew)
{
    INode tmp;
    if ((createIfNew) && !nodes.ContainsKey(label))
    {
        tmp = new Node(label);
        nodes.Add(label, tmp);
    }
    else
    {
```

```
                tmp = nodes[label];
            }

            return tmp;
        }

//    //used at training - shuffles order of paths before training, essential
//    //for using multiple learners - each learns in a different order

        public void ScramblePaths()
        {
            Random rand = new Random();
            List<Path> tmpPaths = new List<Path>();
            while (paths.Count > 0)
            {
                int idx = rand.Next(paths.Count);
                Path p = paths[idx];
                tmpPaths.Add(p);
                paths.Remove(p);
            }

            paths = new List<Path>(tmpPaths);
        }

        public Path GetPath(int index)
        {
            return paths[index];
        }

        public List<Path> GetPaths()
        {
            return paths;
        }

        public Pattern GetPattern(int id)
        {
            return patterns.GetId(id);
        }

        public List<Pattern> GetPatterns()
        {
            return (List<Pattern>)patterns;
        }

        public EquivalenceClass GetEquivalenceClass(int id)
        {
            return equivalenceClasses[id];
```

```
}

public List<EquivalenceClass> GetEquivalenceClasses()
{
    return (List<EquivalenceClass>)equivalenceClasses;
}

public void PrintPath(int id)
{

    foreach(Edge ed in BeginNode.GetOutEdges())
    {
    //Edge ed = (Edge)e.next();
        Edge tmp = ed;
        if (tmp.Path.GetId() == id)
        {
            while (tmp.NextEdge!=null)
            {
                Console.Write(tmp.ToNode.Label+"->");
                tmp=tmp.NextEdge;
            }
            Console.WriteLine();
            return;
        }
    }
}

public bool AddPattern(Pattern p)
{
    Path patternPath = p.GetPath();
    bool result = false;

    List<Edge> candidateEdges = null;
    //get all the edges coming out of the first node of the pattern
    candidateEdges = patternPath.GetNode(0).GetOutEdges();

    //get the set of edges that are outEdges
    //of the last node in the pattern
    //and are a continuation of a path that
    //went throgh the entire pattern:
    for (int i=1;i<patternPath.GetLength();i++)
    {
        INode node = patternPath.GetNode(i);
        //get the edges that continue the last
        //candidateEdges and pass through the pattern
        candidateEdges = node.GetOutEdges(candidateEdges);
```

```
}
candidateEdges.Sort();
if (candidateEdges.Count > 0)
{
    //Enumeration e = candidateEdges.elements();
    List<Path> pathsToSqueeze = new List<Path>();

    //loop throgh the candidateEdges:
    foreach(Edge outOfPattern in candidateEdges)
    {

        // do one more test to be sure that the sub-path
        // candidate is still
        // available:
        Edge tmp = outOfPattern;
        bool validSqueeze = true;
        for (int i=patternPath.GetLength()-1;i>=0;i--)
        {
            INode patNode = patternPath.GetNode(i);
            //if the edge no longer exists:
            if (tmp == null)
            {
                validSqueeze = false;
                break;
            }
            else if (patNode is EquivalenceClass)
            {
                if (!((EquivalenceClass)patNode).Contains(tmp.FromNode))
                {
                    validSqueeze = false;
                    break;
                }
            }
            else if (tmp.FromNode != patNode)
            {
                validSqueeze = false;
                break;
            }
            //go one edge backwards:
            tmp = tmp.PreviousEdge;
        }

        //if we can't squeeze the graph,
        //move to next edge in candidateEdges
        if (!validSqueeze)
        {
            continue;
```

```
    }

    //remove the edge coming out of the
    //pattern from the last node of the pattern
    outOfPattern.FromNode.RemoveOutEdge(outOfPattern);

    Edge intoPattern=null;
    tmp = outOfPattern.PreviousEdge;
    //remove all the edges of the pattern
    for (int i=1;i<patternPath.GetLength();i++)
    {
        intoPattern = tmp.PreviousEdge;
        this.edges.Remove(tmp);
        tmp.FromNode.RemoveOutEdge(tmp);
        tmp.ToNode.RemoveInEdge(tmp);
        tmp.PreviousEdge = null;
        tmp.NextEdge = null;
        tmp=intoPattern;
    }

    //remove the edge going into the pattern
    //from the first node of the pattern
    intoPattern.ToNode.RemoveInEdge(intoPattern);
    //replace the path with a single pattern node
    intoPattern.NextEdge = outOfPattern;
    intoPattern.ToNode = p;
    outOfPattern.FromNode = p;
    outOfPattern.PreviousEdge = intoPattern;
    p.AddInEdge(intoPattern);
    p.AddOutEdge(outOfPattern);
    pathsToSqueeze.Add(outOfPattern.Path);

    p.AddSqueezeInstances();

    result = true;
}

// take all the paths we squeezed in the
//graph and squeeze the path representation
foreach(Path tmpPath in pathsToSqueeze)
{
    tmpPath.Squeeze(p,-1,-1);
}

if (result)
{
    //add the pattern to the list of patterns
```

```
            this.patterns.Add(p);

            foreach(INode tmp in p.GetPath().GetNodes())
            {
                //if one of the nodes of the pattern is
                //an ec that is not regirstered yet:
                if (tmp is EquivalenceClass)
                {
                     if(!this.equivalenceClasses.Contains
                         ((EquivalenceClass)tmp))
                    {
                        //register the ec in our graph
                        this.equivalenceClasses.Add
                        ((EquivalenceClass)tmp);
                    }
                }
            }
        }
        Console.WriteLine("Adding pattern "+p.GetId()+" to Graph:"+p);
        Console.WriteLine("Squeezed "+p.GetSqueezeInstances()+
            " times --> compression: "+GetCompression());

        return result;
    }
    else
    {
        Console.WriteLine("CANNOT SQUEEZE:" +
            patternPath.ToFormattedString(false));
        return false;
    }
}

//adds a path to the graph
public void AddPath(Path path)
{
    path.SetId(paths.Count);
    paths.Add(path);
    INode fromNode = path.GetNode(0);
    INode toNode = fromNode;

    //add an edge from the begin node to the first node of the path
    Edge edge = new Edge(path, -1, BeginNode, fromNode);
    this.edges.Add(edge);
    BeginNode.AddOutEdge(edge);
    fromNode.AddInEdge(edge);

    //add the rest of the path to the graph
```

```
    Edge lastEdge = edge;
    for (int i = 0; i < path.GetLength() - 1; i++)
    {
        toNode = path.GetNode(i + 1);
        edge = new Edge(path, i, fromNode, toNode);
        this.edges.Add(edge);
        fromNode.AddOutEdge(edge);
        toNode.AddInEdge(edge);
        lastEdge.NextEdge = edge;
        edge.PreviousEdge = lastEdge;
        fromNode = toNode;
        lastEdge = edge;
    }

    //add the edge between the last node of the path and the end node
    edge = new Edge(path, -1, toNode, EndNode);
    this.edges.Add(edge);
    toNode.AddOutEdge(edge);
    EndNode.AddInEdge(edge);
    lastEdge.NextEdge = edge;
    edge.PreviousEdge = lastEdge;
}


private void SqueezeIds()
{
    int id = 0;
    foreach(Pattern p in patterns)
    {
        p.SetId(id++);
    }
    id = 0;

    foreach(EquivalenceClass ec in equivalenceClasses)
    {
        ec.SetId(id++);
    }

}

public double GetAvgPatternLength()
{
    double size = 0.0;
    foreach (Pattern p in this.patterns)
    {
        size += p.GetPath().GetLength();
    }
```

```
    size /= patterns.Count;
    return size;
}


public void StampSize()
{
    this.initialSize = CountEdges();
}

public double GetCompression()
{
    return ((double)CountEdges()) / initialSize;
}

public void PrintPatternsFile(string filename)
{
    SqueezeIds();

    string patternsFile = Finals.WORKING_DIR + filename +
        Finals.PATTERNS_FILE_APPENDIX + Finals.SYSTEM_FILE_TYPES;

    try
    {
        StreamWriter fos = new StreamWriter(patternsFile);

        foreach(EquivalenceClass ec in equivalenceClasses)
        {
            fos.WriteLine(ec.ToString());
        }

        foreach(Pattern pattern in patterns)
        {
            fos.WriteLine(pattern.ToString());
        }
        fos.Flush();
        fos.Dispose();
    }
    catch (IOException ioe)
    {
        Console.WriteLine(ioe.Message);
        Console.WriteLine(ioe.StackTrace);
    }
}

public void PrintPathsFile(string filename)
```

```
{
    SqueezeIds();

    string pathsFile = Finals.WORKING_DIR + filename +
        Finals.PATHS_FILE_APPENDIX + Finals.SYSTEM_FILE_TYPES;
    try
    {
        StreamWriter fos = new StreamWriter(pathsFile);

        foreach(Path path in paths)
        {
            fos.WriteLine(path.ToString());
        }
        fos.Flush();
        fos.Dispose();

    }
    catch (IOException ioe)
    {
        Console.WriteLine(ioe.Message);
        Console.WriteLine(ioe.StackTrace);
    }

}

public void PrintFormattedPaths(string filename)
{
    string formattedFile = Finals.WORKING_DIR + filename +
        Finals.FORMATTED_PRINT_FILE_APPENDIX + Finals.SYSTEM_FILE_TYPES;
    try
    {
        StreamWriter fos = new StreamWriter(formattedFile);

        foreach(Path path in paths)
        {
            fos.WriteLine(path.ToFormattedString(true));
        }

        fos.Flush();
        fos.Dispose();

    }
    catch (IOException ioe)
    {
        Console.WriteLine(ioe.Message);
        Console.WriteLine(ioe.StackTrace);
    }
```

```
}

public void PrintPatternTrees(string filename)
{
    string treeFile = Finals.WORKING_DIR +
        filename + ".tree_vector" + Finals.SYSTEM_FILE_TYPES;
    string labelsFile = Finals.WORKING_DIR +
        filename + ".patterns_labels" + Finals.SYSTEM_FILE_TYPES;
    string colorsFile = Finals.WORKING_DIR +
        filename + ".label_colors" + Finals.SYSTEM_FILE_TYPES;

    try
    {
        StreamWriter fost = new StreamWriter(treeFile);
        StreamWriter fosl = new StreamWriter(labelsFile);
        StreamWriter fosc = new StreamWriter(colorsFile);


        foreach(Pattern patt in patterns)
        {
            INode n = (INode)patt;
            PatternTree p = new PatternTree(n);
            fost.WriteLine(p.GetTree());
            fosl.WriteLine(p.GetLabels());
            fosc.WriteLine(p.GetColors());
        }
        fost.Flush();
        fost.Dispose();
        fosl.Flush();
        fosl.Dispose();
        fosc.Flush();
        fosc.Dispose();
    }
    catch (IOException ioe)
    {
        Console.WriteLine(ioe.Message);
        Console.WriteLine(ioe.StackTrace);
    }

}

public bool CheckPath(int id)
{
    Path path = this.GetPath(id);
    Edge edge = null,edge2=null;
    bool result = false;
```

```
            foreach(Edge e in this.BeginNode.GetOutEdges())
            {
                edge = e;
                if (edge.Path == path){
                    result = true;
                    edge2=edge;
                    break;
                }
            }
            for (int j=0;j<path.GetLength();j++)
            {
                INode node = path.GetNode(j);
                if (node != edge.ToNode)
                {
                    result = false;
                    break;
                }
                edge = edge.NextEdge;
                if (edge==null){
                    result = false;
                    break;
                }
            }
            if (result==false)
            {
                Console.WriteLine("Path:"+path.ToString());
                while (edge2!=null)
                {
                    Console.WriteLine(edge2.ToNode.Label+"->");
                    edge2 = edge2.NextEdge;
                }
            }
            return result;
        }
    }
}
```

A.2 INode interface

```
using System;
using System.Collections.Generic;
using System.Text;
using JAdiosPort.Model;

namespace JAdiosPort
```

```
{
    public interface INode
    {
        int CountEdges();
        void AddInEdge(Edge edge);
        void AddOutEdge(Edge edge);

        void RemoveInEdge(Edge edge);
        void RemoveOutEdge(Edge edge);

        List<Edge> GetOutEdges();
        List<Edge> GetOutEdges(List<Edge> inEdgesCandidates);

        List<Edge> GetInEdges();
        List<Edge> GetInEdges(List<Edge> outEdgesCandidates);


        string Label { get;}


    }
}
```

A.3 Pattern class

```
using System;
using System.Collections.Generic;
using System.Text;
using JAdiosPort.Helpers;

namespace JAdiosPort.Model
{
    public class Pattern : Node,ISortedVectorElement,IComparable,INode
    {
        private static int LAST_ID = 1000;

        private double significance;
        private Path path; //the nodes that comprise the pattern
        private int id;

        private int squeezeInstances;

        public Pattern(Path path, double significance, int id)
            : base("_P" + id.ToString())
        {
```

```
        this.id = id;
        LAST_ID = (LAST_ID <= id) ? id + 1 : LAST_ID;
        this.significance = significance;
        this.squeezeInstances = 0;
        this.path = path;
    }

    public Pattern(Path path, double significance)
        : base("_P" + LAST_ID.ToString())
    {

        this.id = LAST_ID++;
        this.significance = significance;
        this.path = path;
    }

    public void UpdatePattern(Path path, double significance)
    {
        this.significance = significance;
        this.path = path;
    }

    /* compare by significance of pattern and then
    //by productivity, i.e. how many
* different 'strings' can be generated by this pattern
*/
    public int CompareTo(Object o)
    {
        if (((Pattern)o).significance == this.significance)
        {
            if (this.CalcProductivity() > ((Pattern)o).CalcProductivity())
                return -1;
            else if (this.CalcProductivity() < ((Pattern)o).CalcProductivity())
                return 1;
            else
                return 0;
        }
        else if (significance < ((Pattern)o).significance)
            return 1; // put lower significance at the end (reverse ordering)
        else return -1;
    }

    #region ISortedVectorElement Members

    public int GetId()
    {
        return id;
```

```
}

#endregion

public void SetId(int id)
{
    this.id = id;
    this.label = "__P" + id;
}

public void AddSqueezeInstances()
{
    squeezeInstances++;
}

public void SetSqueezeInstances(int squeezeInstances)
{
    this.squeezeInstances = squeezeInstances;
}

public int GetSqueezeInstances()
{
    return squeezeInstances;
}

public static void SetLastId(int id)
{
    LAST_ID = id;
}

public Path GetPath()
{
    return path;
}

public double GetSignificance()
{
    return significance;
}

public override string  ToString()
{
    return "P:" + id + ":" + significance + ":" +
        squeezeInstances + path.ToString();
}

public String ToFormattedString()
```

```
    {
        if (Finals.EC_WINDOW_SIZE == -1)
        {
            return (path.ToFormattedString(false));
        }
        else
        {
            return ("[P" + id + ":" + path.ToFormattedString(false) + "]");
        }
    }

    /* static method - gets a pattern from the graph according to ID
*/
    public static Pattern GetPattern(string label)
    {
        int patIndex = int.Parse(label.Substring(3));
        return Graph.GetSingleInstance().GetPattern(patIndex);

    }

    public bool Equals(Pattern p)
    {
        return this.path.Equals(p.path);
   }

    public static List<Pattern> RemoveEquals(List<Pattern> patterns)
    {
        List<Pattern> result = new List<Pattern>();
        while (patterns.Count > 0)
        {
            IEnumerator<Pattern> i = patterns.GetEnumerator();
            i.MoveNext();

            Pattern bestPattern = i.Current;

            patterns.Remove(bestPattern);

            while (i.MoveNext())
            {
                Pattern p = i.Current;

                if (p.Equals(bestPattern))
                {
                    if (p.significance > bestPattern.significance)
                        bestPattern = p;
                    patterns.Remove(p);
```

```
                }

            }

            result.Add(bestPattern);

        }
        return result;
    }

    /* this mehtod calculates how many different sentences can be generated
* from this pattern recursively
*/
    public int CalcProductivity()
   {
      int i, j, productivity = 1;

      //loop through the nodes in the path
      for (i = 0 ; i < path.GetLength() ; ++i)
      {
         //if we reached an ec
         if (path.GetNode(i) is EquivalenceClass)
         {
            EquivalenceClass ec = (EquivalenceClass) path.GetNode(i);
            int sumOfElements = 0;

            //calc number of possible elements generated by ec
            for (j = 0 ; j < ec.GetSize() ; j++)
            {
               if (ec.GetNode(j) is Pattern)
                    {
                        Pattern p = (Pattern)ec.GetNode(j);
                  sumOfElements += p.CalcProductivity();
                    }
               else
                  sumOfElements += 1;
            }
            //multiply the productivity with num of options for the ec
            productivity *= sumOfElements;
         }
         //if it's a pattern calc recursively
         else if (path.GetNode(i) is Pattern)
         {
            Pattern pattern = (Pattern) path.GetNode(i);
            productivity *= pattern.CalcProductivity();
         }
      }
```

```
            return productivity;
        }


    }
}
```

A.4 EquivalenceClass class

```
using System;
using System.Collections.Generic;
using System.Text;
using JAdiosPort.Helpers;

namespace JAdiosPort.Model
{
    public class EquivalenceClass : Node,ISortedVectorElement,INode
    {
        private static int LAST_ID = 1000;

        private int id;
        private List<INode> nodes;
        private List<int> frequencies;

        public EquivalenceClass(int id) : base("_E" + id.ToString())
        {

            this.id = id;
            LAST_ID = (LAST_ID <= id) ? id + 1 : LAST_ID;
            this.nodes = new List<INode>();
            this.frequencies = new List<int>();
        }

        public EquivalenceClass() : base("_E" + LAST_ID)
        {

            this.id = LAST_ID++;
            this.nodes = new List<INode>();
            this.frequencies = new List<int>();
        }

        //this constructor adds the elements in the vector to the ec
        public EquivalenceClass(List<INode> elements)
            : base("_E" + LAST_ID.ToString())
        {
```

```
        this.id = LAST_ID++;
        foreach(INode node in elements)
        {

            AddNode(node);
        }
    }

    public int GetFrequency(int index)
    {
        return frequencies[index];
    }

    private void AddFrequency(int index, int amount)
    {
        frequencies[index] += amount;

    }

//    /* the method adds a node to the ec or
    //increments the frequencies by 1
//     * if it already exists
//     */
    public void AddNode(INode node)
    {
        int nodeIdx = nodes.IndexOf(node);
        if (nodeIdx == -1)
        {
            nodes.Add(node);
            frequencies.Add(1);
        }
        else
        {
            AddFrequency(nodeIdx, 1);
        }
    }

//    // adds a vector of nodes to the ec
    public void AddNodes(List<INode> nodes)
    {
        foreach (INode n in nodes)
        {
            AddNode(n);
        }
    }

    public INode GetNode(int index)
```

```
{
    return nodes[index];
}

public List<INode> GetNodes()
{
    return nodes;
}

public void SetId(int id)
{
    this.id = id;
    this.label = "_E" + id.ToString();
}

public static void SetLastId(int id)
{
    LAST_ID = id;
}

public int GetSize()
{
    return this.nodes.Count;
}

public bool Contains(INode node)
{
    return nodes.Contains(node);
}

// true iff all elements of this appear in ec and
//cover at least Finals.EC_OVERLAP_COVERAGE of it's elements
public double Overlap(EquivalenceClass ec)
{
    foreach (INode n in nodes)
    {
        if (!ec.GetNodes().Contains(n))
        {
            return 0.0;
        }
    }

    int a = 0;
    int b = 0;

    foreach (INode n in ec.GetNodes())
    {
```

```
            if (nodes.Contains(n))
            {
                a++;
            }
            else
            {
                b++;
            }
        }

        //calc percentage of overlap
        double result = ((double)a) / (a + b);
        //if the overlap is less than threshold return '0'
        if (result > Finals.EC_OVERLAP_COVERAGE)
        {
            return result;
        }
        else
        {
            return 0.0;
        }
    }

    public override string ToString()
    {
        StringBuilder sb = new StringBuilder("E:");
        sb.Append(id);
        sb.Append(":");
        for (int i = 0; i < nodes.Count; i++)
        {
            sb.Append(nodes[i].Label);
            sb.Append(":");
            sb.Append(frequencies[i]);
            sb.Append(":");
        }
        return sb.ToString();
    }

    public string ToFormattedString()
    {
        StringBuilder result = new StringBuilder();
        result.Append("{E" + id + ":");
        foreach (INode n in nodes)
        {
            if (n is Pattern)
            {
                Pattern p = (Pattern)n;
```

```
            result.Append(p.ToFormattedString());
        }
        else if (n is EquivalenceClass)
        {
            EquivalenceClass ec = (EquivalenceClass)n;
            result.Append(ec.ToFormattedString());
        }
        else
        {
            result.Append(n.ToString());
        }
    }
    result.Append("}");
    return result.ToString();
}

//    //static method: get an ec in the graph
public static EquivalenceClass GetEquivalenceClass(string label)
{
    int ecIndex = int.Parse(label.Substring(2));
    return Graph.GetSingleInstance().GetEquivalenceClass(ecIndex);
}

#region ISortedVectorElement Members

public int GetId()
{
    return this.id;
}

#endregion

/////
// METHODS OVERRIDING NODE:
/////

public new int CountEdges()
{
    int result = 0;
    foreach(INode n in nodes)
    {
        result += n.CountEdges();
    }
    return result;
}

public new List<Edge> GetOutEdges()
```

```
{
    List<Edge> result = new List<Edge>();
    foreach (INode n in nodes)
    {
        result.AddRange(n.GetOutEdges());
    }
    return result;
}


// returns a vector of all edges going out of this node, that
//are continuing edges contained in the inEdgesCandidates group.
public new List<Edge> GetOutEdges(List<Edge> inEdgesCandidates)
{
    List<Edge> result = new List<Edge>();
    foreach(INode n in nodes)
    {
        result.AddRange(n.GetOutEdges(inEdgesCandidates));
    }
    return result;
}

public new List<Edge> GetInEdges()
{
    List<Edge> result = new List<Edge>();
    foreach(INode n in nodes)
    {
        result.AddRange(n.GetInEdges());
    }
    return result;
}

public new List<Edge> GetInEdges(List<Edge> outEdgesCandidates)
{
    List<Edge> result = new List<Edge>();
    foreach(INode n in nodes)
    {
        result.AddRange(n.GetInEdges(outEdgesCandidates));
    }
    return result;
}

public override bool Equals(Object o)
{
  if (!(o is EquivalenceClass))
        return false;
  return this.Overlap((EquivalenceClass)o)==1.0;
```

```
       }

        public override int GetHashCode()
        {
            return base.GetHashCode();
        }
    }
}
```

A.5 Edge class

```
using System;
using System.Collections.Generic;
using System.Text;

namespace JAdiosPort.Model
{
    public class Edge : IComparable<Edge>
    {
        private Path path;
        private INode fromNode, toNode;
        private Edge nextEdge, prevEdge;

        public Edge(Path parentPath, int index,
        INode fromNode, INode toNode)
        {
            this.path = parentPath;
            this.fromNode = fromNode;
            this.toNode = toNode;
            this.nextEdge = null;
            this.prevEdge = null;
        }

        public INode FromNode
        {
            get
            {
                return fromNode;
            }
            set
            {
                fromNode = value;
            }
        }
```

```
public INode ToNode
{
    get
    {
        return toNode;
    }
    set
    {
        toNode = value;
    }
}

public Path Path
{
    get
    {
        return path;
    }
}

public Edge NextEdge
{
    get
    {
        return nextEdge;
    }
    set
    {
        nextEdge = value;
    }
}

public Edge PreviousEdge
{
    get
    {
        return prevEdge;
    }
    set
    {
        prevEdge = value;
    }
}

public override string ToString()
{
    return "[" + fromNode + "->" + toNode + "]";
```

```
        }


        #region IComparable<Edge> Members

        public int CompareTo(Edge other)
        {
            // Compare by path id, and then by order in path

            if (this == other)
            {
                return 0;
            }

          Edge e = other;

          if (this.path.GetId() != e.path.GetId())
          {
             return this.path.GetId() - e.path.GetId();
          }
          // assuming were on the same path id:
          Edge tmp = this.PreviousEdge;
          while (tmp!=null)
            {
             if (tmp == e)
                {
                return 1; // e precedes this
             }
             tmp = tmp.PreviousEdge;
          }
          return -1; // we scanned until beginning of path ==>
          this precedes e
        }

        #endregion
    }
}
```

A.6 Path class

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace JAdiosPort.Model
```

```
{
    public class Path
    {
        private int id;
        private List<INode> nodes;


        public Path(string line, char splitChar)
        {
            Initialize(line, splitChar);
        }

        public Path()
        {
            nodes = new List<INode>();
            id = -1;
        }

        public void Initialize(string line,char splitChar)
        {
            Graph graph = Graph.GetSingleInstance();

            nodes = new List<INode>();

            //tokenize input
            string[] splitLine = line.Split(new char[] { splitChar }
                , StringSplitOptions.RemoveEmptyEntries);

            //loop through all the tokens of the string

            foreach (string label in splitLine)
            {
                //if it's not a 'begin' node or an 'end' node:
                if (label.CompareTo(graph.BeginNode.Label) != 0
                    && label.CompareTo(graph.EndNode.Label) != 0)
                {
                    INode tmp;
                    //if it's the token of a pattern

                    if (label.StartsWith("_P"))
                    {
                        tmp = Pattern.GetPattern(label);
                        //if this pattern does not exist, create a new one
                        if (tmp == null)
                        {
                            tmp = new Pattern(new Path(), 0,
                                int.Parse(label.Substring(3)));
```

```
                    graph.RegisterPattern((Pattern)tmp);
                }

            }
            //if it's the token of an ec
            else if (label.StartsWith("_E") ||
            label.StartsWith("__E"))
            {
                tmp = EquivalenceClass.GetEquivalenceClass(label);
            }
            //if it's a regular node
            else
            {
                //this creates the node if it doesn't exist in the graph
                tmp = graph.GetNode(label, true);
            }
            //add the new node to the path's vector of node
            nodes.Add(tmp);
        }
    }

    id = -1;
}

public Path Duplicate()
{
    Path result = new Path();
    foreach (INode n in nodes)
    {
        result.nodes.Add(n);
    }

    return result;
}

public void AddNode(INode node)
{
    nodes.Add(node);
}

public void SetECNode(EquivalenceClass ec, int index)
{
    nodes.RemoveAt(index);
    nodes.Insert(index, ec);
}

public void SetNode(INode node, int index)
```

```
{
    nodes.RemoveAt(index);
    nodes.Insert(index, node);
}

public void SetId(int id)
{
    this.id = id;
}

public int GetId()
{
    return id;
}

public int GetLength()
{
    return nodes.Count;
}

public INode GetNode(int index)
{
    try
    {
        return nodes[index];
    }
    catch (ArgumentOutOfRangeException ioore)
    {

        return null;
    }
}

public List<INode> GetNodes()
{
    return nodes;
}

public Path GetSubPath(int start, int end)
{
    Path result = new Path();
    for(int i = start; i <=end; i++)
    {
        result.AddNode(nodes[i]);
    }

    return result;
```

```
}


// takes all the patterns in the graph and
// squeezes the occurrences of them
// in the graph
public bool SqueezeToModel()
{
    Graph g = Graph.GetSingleInstance();
    int result = 0;

    foreach (Pattern p in g.GetPatterns())
    {
        result += Squeeze(p, -1, -1);
    }

    return (result > 0);
}

//replaces occurencrs of patterns in our path with a pattern node
public int Squeeze(Pattern pattern, int beginIdx, int endIdx)
{
    Path tmp = new Path();
    tmp.AddNode(pattern);
    int replaced = ReplacePath(pattern.GetPath(),
        tmp, beginIdx, endIdx);
    return replaced;
}


// replaces occurences of the old path with new
//path between beginIndex and EndIndex
public int ReplacePath(Path oldPath, Path newPath,
     int beginIdx, int endIdx)
{
    int result = 0;
    int index = 0;
    do
    {
        index = this.FindPath(oldPath, beginIdx, endIdx);
        if (index != -1)
        {
            for (int j = 0; j < oldPath.GetLength(); j++)
                nodes.RemoveAt(index);
            for (int j = 0; j < newPath.GetLength(); j++)
                nodes.Insert(index + j, newPath.GetNode(j));
            result++;
```

```
        }
    } while (index != -1);
    return result;
}

//replace the nodes from index i with subpath
public void ReplacePath(int index, Path subPath)
{
    for (int i = 0; i < subPath.GetLength(); i++)
    {
        this.nodes.RemoveAt(i + index);
        this.nodes.Insert(i + index, subPath.GetNode(i));
    }
}

// return - the index of the first occurance of
// target in this path or -1 if none
public int FindPath(Path subPath, int beginIndex, int endIndex)
{
    int j;

    int subLength = subPath.GetLength();

    // reset endIndex if necessary
    if ((endIndex == -1) || (endIndex>nodes.Count-subLength))
    {
         endIndex = (nodes.Count-subLength);
    }
    //set beginIndex if necessary
    beginIndex = beginIndex<0? 0:beginIndex;
    for (int i=beginIndex;i<=endIndex;i++)
    {
         j=0;
         while (subLength>j)
         {

            INode node = subPath.GetNode(j);
            if (node == this.GetNode(i+j))
            {
                j++;
            }
            else if ((node is EquivalenceClass) &&
                      (((EquivalenceClass)node).Contains
                      (this.GetNode(i+j))))
            {
                j++;
            }
```

```
            else
            {
                break;
            }
        }
        if (j==subLength) return i;
    }
    return -1;
}


//count how many times a subpath occurs in our path
public int CountSubPaths(Path subPath)
{
    int result = 0;
    int j = this.FindPath(subPath, 0, -1);
    while (j != -1)
    {
        result++;
        j = this.FindPath(subPath, j + subPath.GetLength(), -1);
    }
    return result;
}

public override bool Equals(object obj)
{
    if (!(obj is Path))
        return false;
    Path p = (Path)obj;
    if (p.GetLength() != this.GetLength())
        return false;
    List<INode> pathNodes = p.GetNodes();
    for (int i = 0; i < p.GetLength(); i++)
    {
        if (pathNodes[i] != nodes[i])
        {
            return false;
        }
    }
    return true;
}

public override string ToString()
{
    StringBuilder result = new StringBuilder();
    for (int i = 0; i < nodes.Count; i++)
    {
```

```
                result.Append(":");
                result.Append(this.GetNode(i).Label);
            }
            return result.ToString();
        }

        public override int GetHashCode()
        {
            return base.GetHashCode();
        }

        public string ToFormattedString(bool whiteSpace)
        {
            StringBuilder result = new StringBuilder();
            foreach (INode node in this.GetNodes())
            {
                if (node is Pattern)
                {
                    Pattern p = (Pattern)node;
                    result.Append(p.ToFormattedString());
                }
                else if (node is EquivalenceClass)
                {
                    EquivalenceClass ec = (EquivalenceClass)node;
                    result.Append(ec.ToFormattedString());
                }
                else
                {
                    result.Append(node);
                }
            }
            return result.ToString();
        }
    }
}
```

Appendix B

Appendix B: Sample ADIOS input and output

This appendix contains sample input and output from the ADIOS algorithm. The first sample shows the text of sentences prepared for the algorithm (i.e., POS tagged and formatted for the software). The second sample shows a set of patterns and equivalence classes ("P" and "E", respectively) induced from the sentences. The last sample shows the compressed paths of the original sentences, i.e., the original sentences with elements replaced by the patterns induced by the algorithm. Note that while each original sentence is represented in the paths, they are not ordered in the same way; the version of ADIOS used here scrambles the sentences in order to accommodate for multiple learners, since the order of presentation does matter to the algorithm. Also note that a fuller corpus produces sentences that are often maximally compressed into a single pattern, or a sequence of two or three patterns. Since sparse data sets were used here, compression is appropriately less.

B.1 Sample Input

```
** EX VBP JJ NNS IN JJ NNS NN . ##
** DT IN DT NNS MD VB VBN TO VB IN NN CC NN NNS IN RB IN JJ JJ NNS . ##
** IN PRP$ NN , DT JJ NN IN JJ NNS IN DT NNP NNPS VBZ NN . ##
** DT IN DT NN IN NNS NN VBZ IN NNS , NNS , CC NNS . ##
** DT NNS VBZ DT JJ NN IN WP NNS VBP CC VB IN DT NN IN DT NN . ##
** DT NN , DT NN VBZ RB JJ NNS IN NNS . ##
** IN NN , PRP VBZ JJ NN IN DT NN VBN DT NN TO VB CC VBD VBG IN PRP$ NNS . ##
** CC , PRP RB RB VB IN DT JJ NNS NNS VBP IN PRP$ NN . ##
** IN NNS RB VB DT JJ NNS PRP$ NNS VBP VBG , PRP MD VB DT NN IN IN
     PRP VBP JJ PRP MD VB VBN CC VBN . ##
** DT NN IN DT NNS MD RB VB VBN IN JJR NNS IN JJ IN WP NNS VBP NN . ##
** DT NN IN NN MD VB IN DT NN , CC RB JJR NNS IN NNS MD VB VBN IN RB . ##
** IN NN TO DT NNS , NNS CC NNS VBP VBN DT JJ NN IN DT NN IN NNS NN . ##
** NNP VB VBN TO VB IN IN JJ NN IN . ##
** IN NNS VBP WP PRP$ NNS VBP VBG , JJ NNS MD VB IN . ##
** DT MD RB VB TO NNS . ##
```

```
** PRP MD RB VB JJ TO VB IN DT DT NNS IN DT NNP NNPS , CC PRP MD VB RBR JJ
     IN NNS TO VB JJ . ##
** NNP MD VB DT VBN NN IN DT NN POS NN . ##
** RB , NNS MD VB DT JJ NN IN VBG NN . ##
** JJ NNS NN NN NN IN PRP VBZ JJ IN DT NNS TO VB . ##
** IN NN , EX VBZ DT VBG NN IN JJ NNS . ##
** TO VB DT NN , NNS MD VB CC VB JJR NN TO VB IN PRP$ NNS . ##
** RB , IN NNS VBP VBG PRP$ NNS NN IN JJ CC NN NNS IN DT JJ NN , JJR NNS
     NN MD RB VB TO VB TO NN TO VB WRB PRP VBP . ##
** IN NN , NN VBZ DT JJ NN IN JJ NNS NN . ##
** NN VBZ VBG NNS IN WP NNS NNS TO VB JJ . ##
** IN EX VBP VBG NNS , EX VBP RB VBG NNS . ##
** NNP DT JJ , VBG NN , CC NN VBP RB CD NNS TO DT JJ NN IN NN NNS VBP
     IN IN DT JJ NN . ##
```

B.2 Patterns and Equivalence Classes

```
E:0:VBN:1:JJ:1:
E:1:WP:1:VBG:1:
E:2:MD:1:TO:1:
E:3:DT:1:NNS:1:
E:4:PRP:1:EX:1:NN:1:
E:5:__P4:1:IN:1:
E:6:NNS:1:NN:1:
E:7:__P7:1:NNS:1:
E:8:VBZ:1:__P2:1:
E:9:JJR:1:__P1:1:
E:10:__P0:1:JJ:1:
E:11:NN:1:__P3:1:
E:12:POS:1:IN:1:
E:13:VBN:1:,:1:
E:14:NNS:1:PRP:1:
E:15:PRP:1:EX:1:
E:16:JJ:1:VBG:1:
E:17:EX:1:NN:1:
E:18:VBG:1:CD:1:
P:0:0.989259900960819:7:DT:_E0:NN:IN
P:1:0.967075508923403:2:NNS:VBP:_E1:PRP$:NNS
P:2:0.994242400656564:21:_E2:VB
P:3:0.967075508923403:3:DT:NN:IN:_E3:NN
P:4:0.997390795814413:3:NN:,:_E4:VBZ
P:5:0.967075508923403:2:CC:NN:NNS:IN
P:6:0.967075508923403:2:_E5:JJ:NN:IN
P:7:0.963403286130369:22:_E3:_E6
P:8:0.967075508923403:2:_E6:IN:JJ:_E7
```

```
P:9:0.967075508923403:2:_E8:__P0:_E1:_E6
P:10:0.967075508923403:2:IN:_E9:_E6:IN:JJ
P:11:1:2:_E5:_E10:JJ:_E7
P:12:0.967075508923403:2:IN:_E11:_E8:IN:_E7:,
P:13:0.967075508923403:2:_E10:_E7:_E12:_E6
P:14:0.995792720242171:4:__P7:_E13:_E7
P:15:0.967075508923403:2:_E6:IN:_E14:_E8:_E0:IN
P:16:0.967075508923403:4:MD:RB:VB
P:17:0.967075508923403:2:IN:_E15:VBP:_E16:_E14
P:18:0.967075508923403:2:_E17:VBP:RB:_E18:NNS
```

B.3 Compressed Paths

```
:NN:VBZ:VBG:NNS:IN:WP:__P7:__P2:JJ:.
:NNP:__P2:__P13:.
:__P7:IN:__P7:__P16:VBN:__P10:IN:WP:NNS:VBP:NN:.
:IN:__P1:VBP:VBG:,:JJ:NNS:__P2:IN:.
:IN:NNS:RB:VB:DT:JJ:NNS:PRP$:NNS:VBP:VBG:,:PRP:__P2:__P7:IN:__P17:__P2:
     VBN:CC:VBN:.
:DT:IN:__P7:__P2:VBN:__P2:IN:NN:__P5:RB:__P11:.
:IN:NN:TO:__P14:CC:NNS:VBP:VBN:__P0:__P3:.
:__P17:,:__P18:.
:IN:__P4:DT:VBG:__P8:.
:IN:PRP$:NN:,:__P0:JJ:NNS:IN:DT:NNP:NNPS:VBZ:NN:.
:__P7:__P9:VBP:CC:VB:IN:__P3:.
:PRP:__P16:JJ:__P2:IN:DT:__P7:IN:DT:NNP:NNPS:,:CC:PRP:__P2:RBR:JJ:IN:
     NNS:__P2:JJ:.
:CC:,:PRP:RB:RB:VB:IN:DT:JJ:__P7:VBP:IN:PRP$:NN:.
:RB:,:__P10:__P5:DT:JJ:NN:,:JJR:__P7:__P16:__P2:TO:NN:__P2:WRB:PRP:VBP:.
:IN:__P6:__P14:__P2:CC:VBD:VBG:IN:PRP$:NNS:.
:IN:__P11:.
:__P7:__P12:CC:RB:JJR:__P15:RB:.
:EX:VBP:JJ:__P8:.
:DT:__P12:NNS:,:CC:NNS:.
:__P14:VBZ:RB:__P13:.
:__P2:__P14:__P2:CC:VB:JJR:NN:__P2:IN:PRP$:NNS:.
:NNP:VB:VBN:__P2:IN:__P6:.
:DT:__P16:TO:NNS:.
:RB:,:NNS:__P9:.
:JJ:__P7:NN:__P15:__P7:__P2:.
:NNP:DT:JJ:,:VBG:NN:,:CC:__P18:TO:__P0:NN:NNS:VBP:IN:IN:DT:JJ:NN:.
```