Imperial College London

Department of Computing

Machine Learning of Rules for Part of Speech Tagging

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing Science / Machine Learning of Imperial College London

September 2014

Abstract

In this thesis we approach the problem of Part-of-Speech tagging which is one of the most important problems in the field of Machine Translation. Some research has been invested lately in machine learning of rules for PoS taggers, mainly using Inductive Logic Programming systems such as Progol. A newer ILP framework, Meta Interpretive Learning arose from the need to tackle some underexplored areas of logic learning, namely predicate invention and learning of recursion. Throughout the thesis we detail the implementation of two PoS disambiguators that use different approaches to machine learning of rules using the Metagol system, one of which achieves state of the art level accuracy. With MIL being introduced in 2013, the project has a highly exploratory nature: our end goals were to provide a Proof of Concept, suggest solutions for some of the issues we encountered and discuss tradeoffs between producing good rules and execution time. Even though we did not primarily aim for high accuracy, our final system is able to compete with other recent implementations of rule-based PoS taggers.

Contents

Chapter 1

Introduction

1.1 Problem description / Motivation

Part of speech tagging is an important problem in the field of NLP (Natural Language Processing) and a key part in most of the machine translation systems. Part of speech tagging deals with the problem of identifying the part of speech of each word in a corpus. Most of the time the tagging process includes deduction of other detailed attributes such as gender (nouns), tense (verbs), number (pronouns). The result of this process is often fed into machine translation systems that use algorithms which make translation decisions based on the tags of the words (part of speech plus attributes).

A part of speech tagger can be broken down into two different pipelined components, namely lexical analysis and disambiguation. Lexical analysis is the first stage of the PoS tagging in which words are looked up in dictionaries and assigned all possible interpretations (readings) disregarding the context around them. This leads to a problem that needs to be solved by the second component in the pipeline: some of the words are ambiguous with respect to their part of speech (one example in English is the word *fall* which could be either a noun or a verb) or with respect to their attributes (*read* can be either present or past tense of the verb *to read*). The percentage of ambiguous tokens in the English language is around 33% (obtained by applying the Apertium [1] lexical analysis on a corpus of almost ten thousand words from different sources). The goal of the disambiguation process is to assign the correct reading to each one of the ambiguously tagged words. While the first part may seem trivial, the disambiguation is much more complex. Modern PoS taggers use machine learning algorithms (such as Hidden Markov Models or Maximum Entropy models) in order to score readings and decide which is the most probable one. These statistical approaches have high accuracy, over 90% of the words have correct readings after the PoS tagging. Almost perfect ($> 99\%$) disambiguation has been demonstrated using disambiguation rules written by linguists. The major problem here is that rules require human knowledge and are hard to maintain as their number increases.

There is an area where statistical and rule based approaches overlap and it usually

[1] http://www.apertium.org/

involves automatically generating and selecting rules, although currently no system can match rules produced by linguists. Throughout the thesis we will deal with the problem of learning such rules making use of a framework inside ILP (Inductive Logic Programming), namely Meta Interpretive Learning. Our goal is to demonstrate the possibility of learning disambiguation rules as Prolog programs.

The importance of PoS tagging is immediate from the MT (machine translation) point of view: better tagging means more information for the rest of the translation pipeline and usually leads to higher quality translations. Improving translation techniques results in better access to information across different languages and cultures. The Apertium project (which we will use as a reference throughout this thesis) is, for instance, focused on language pairs for which major machine translation systems have limited or no coverage so its performance directly affects speakers of the respective languages, lowering the language barrier between them and information available in foreign languages.

The part of speech tagger problem has been tackled multiple times in the literature [1, 2, 3, 4, 5], most the applications being in the field of Machine Translation.

1.2 State of the Art

There are two fundamentally different approaches to the Part-of-Speech tagging problem.

On one side we have statistical approaches. The main idea behind them is to design probabilistic models (Hidden Markov Models, Artificial Neural Networks), train and use them to classify ambiguously tagged words. State of the art PoS tagging systems use 3-gram or, sometimes, even 4-gram models (the features are extracted from 3 and 4 tokens around a certain word) [6].

On the other side we have rule-based approaches, where the idea is to gather a set of good rules that can be used for Part-of-Speech disambiguation. While the Brill tagger [7] is a good example of a rule based system where rules are machine learned, there are also frameworks such as Constraint Grammar (introduced in [8] and later refined in [9]) which allows humans to manually write rules for disambiguation.

While human produced rules usually achieve $\sim 99\%$ accuracy under CG-3, machine learned statistical models perform better in practice ($\sim 96\%$) than state of the art rule-based methods ($\sim 94\%$).

Researchers have expressed interest in logic learning approaches to learning rules, most notably using the Progol ILP system [10, 11].

1.3 Thesis structure

In Chapter 2 we will discuss the key differences between statistical and rule-based machine translation systems. We will focus on the latter while studying implementation details of the Apertium system. The reader will be presented with examples of how rules can be used in machine translation at some of the different stages of the Apertium pipeline. Towards the end of the chapter we will discuss the concept of Constraint Grammar and how hand written rules can lead to good part of speech disambiguation.

Chapter 3 is an overview of the Inductive Logic Programming field. We will focus on the Meta Interpretive Learning framework and, more specifically, on the *Metagol_D* system, discussing key concepts such as concept learning, action learning and episodic learning. Each of the sections contain brief examples of what is achievable and how the learning works in general.

The relation between MIL and learning of disambiguation rules will be presented in chapter 4. We will discuss the use of concept / action learning in the context of PoS disambiguation together with implementation details of the learners in each case. This chapter also includes implementation of a trivial statistical approach to PoS disambiguation which will be used as a baseline.

In chapter 5 we will deal with the problem of selecting a good set of rules for the disambiguation: we employ a genetic algorithm that takes all the rules learned by the *Metagol_D* system and try to find the best combination. This step is necessary because at previous stages we generate rules from random examples with different generalization and usefulness degrees.

Detailed results of all experiments are presented and discussed in Chapter 6, while Chapter 7 is reserved for conclusion and future work.

Chapter 2

Rule based translation systems

2.1 Statistical vs Rule based

The main two trends in Part of Speech tagging and, more broadly, Machine Translation systems are statistical and rule based approaches. The fundamental difference between the two types of systems is the that decision making relies either on statistical measures or rules.

A notable SMT (Statistical Machine Translation) system example is Google Translate [1], while RBMT (Rule Based Machine Translation) has been implemented by projects like Apertium [2].

The choice of the used approach is usually dependent on the available resources. In the case of SMT, the system uses aligned corpuses of translated texts (also called parallel translations) in order to be able to build a good statistical model. Depending on data availability, simpler or more complex models can be used. In any case, this type of system usually requires large amounts of data, but less human intervention. On the other hand, RBMT are employed whenever the data is scarce. This scarcity of data needs to be replaced by human knowledge in the form of rules written by linguists. The main issue with this approach is that it needs linguists trained in each one of the source and target language pairs. The rules are usually hard to maintain as their number increases, but they can achieve much higher accuracies in specific situations where only a few examples would be available for the SMT systems.

On most occasions the two approaches overlap and we face hybrid translation systems which use a different approach for different stages in the pipeline. For instance, the Apertium project is traditionally a RBMT system, but it also uses statistical modeling for parts of its pipeline (it can make use of statistical models for PoS disambiguation, while using rules for grammar transfer). The translation system could also use different combinations of algorithms in the pipeline for each language pair in particular; again, the choice is made based on availability of data and / or human knowledge). A survey on statistical machine translation systems can be found in [12].

[1]http://translate.google.com

[2]http://www.apertium.org

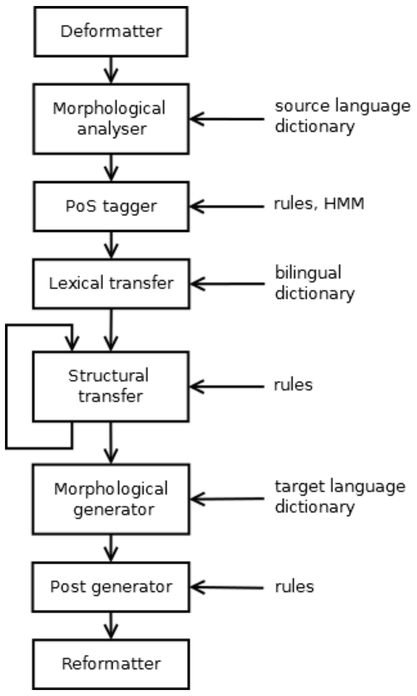

Figure 2.1: Apertium translation pipeline.

2.2 Apertium project

The information in this section is mainly based on the introductory paper of the Apertium project. The paper [13] presents the main idea behind the Apertium translation system along with details of the translation pipeline design choices.

The translation platform was initially aimed at translating related languages (the two systems that formed the basis of the project being a Spanish-Catalan MT system [14] and a Spanish-Portuguese MT system [15]), but has been extended later to cover more distant language pairs, such as Spanish-English [13]. Apertium is the engine behind the web based translation interface at *http://www.apertium.org*, as one of the goals of the project is to provide tools and a sufficiently modular design for building MT related projects on top of it; more details can be found in its introductory paper [16].

Before advancing to the details of the Apertium pipeline we need to discuss the Apertium *stream format*. The stream format is a standard way of encoding needed linguistic information from one stage to the other. Examples of input / output will be provided for each of the modules in the Apertium engine, while trying to translate the HTML formatted sentence "`<p> Yesterday I read a book </p>.`" using the English - Spanish language pair. Figure 2.1 illustrates the interaction between the main components of the Apertium translation engine.

Deformatter

Since the translation is supposed to only transform the contents of a document and not its format, the deformatter module marks the format of the input document with `[]` so that later stages will be able to distinguish the content that is to be analyzed. After deformatting has been applied, our sentence looks like this:

```
.[][<p>]Yesterday I read a book.[][<\/p>
]
```

We can see in the previous example that the HTML tags have been isolated by the deformatter. It also marks the sentence boundaries with `.[]` and uses
as an escape character (we will see that the slash character (/) has a special meaning in the Apertium stream format, so it needs to be escaped). Note that the newline at the end will also be ignored at later stages, being thought as part of the formatting of the document.

Morphological analyzer

The morphological analyzer segments the input in tokens (surface forms or, where detected, multi-word lexical units) and performs a lexical analysis on each one of them [13]. In order to do so it uses a compiled dictionary of the source language (English, in our case). This stage can be seen as the first step of the PoS tagging: each token is augmented with one or more readings which contain information regarding a morphological part (*noun*, *verb*, *pronoun*, etc.) together with possible attributes (such as number, gender, tense, etc.). Take the following example (our initial sentence after this stage):

```
^./.<sent>$
[][<p>]
^Yesterday/Yesterday<adv>$
^I/I<num><mf><sg>/prpers<prn><subj><p1><mf><sg>$
^read/read<vblex><inf>/read<vblex><pres>/
    read<vblex><past>/read<vblex><pp>$
^a/a<det><ind><sg>$
^book/book<n><sg>/book<vblex><inf>/book<vblex><pres>$
^./.<sent>$
[][<\/p>
]
```

Regarding the stream format, we can see that format related input has been ignored; the analysis did transform words into tokens and all of them have some associated readings, separated by slashes (/). A token with more than one reading is said to be ambiguous (examples include `read` and `book`). The ambiguity comes in two different forms. In some cases the part of speech is unambiguous, but its attributes are: take, for instance `read/read<vblex><inf>/read<vblex><pres>/...` – the analysis tells us that this is a verb for sure, but its tense is still ambiguous (a choice of infinitive, present, past and present perfect). On the other hand in the case of `book` the analysis reveals that it could be both a noun and a verb (so the morphological class could not be identified from the dictionary alone). The ambiguity will be solved by the next stage of the translation engine. Some of the words do not experience any kind of ambiguity: the word `a` is, for sure, a determiner and has only one reading associated (`a<det><ind><sg>`).

If the dictionary lookup fails, the tokens are marked using an $\star$ sign.

Part of Speech tagger

The name of this stage in the Apertium translation engine as described in [13] is somewhat misleading. The part of speech tagger deals, in the context of the Apertium system, with disambiguation - its goal is to select one reading per token. The implementation used for this example is based on Hidden Markov Models trained using information from a hand tagged corpus. An alternative to this approach is to have manually produced rules for a Constraint Grammar style disambiguator (this technique is also implemented by the Apertium project, CG-3). The morphologically analyzed and disambiguated sentence becomes:

```
^.<sent>$
[][<p>]
^Yesterday<adv>$
^prpers<prn><subj><p1><mf><sg>$
^read<vblex><past>$
^a<det><ind><sg>$
^book<n><sg>$
^.<sent>$
[][<\/p>
]
```

Notice that after this stage each token has only one reading associated.

Lexical transfer

At the lexical transfer stage, tokens are looked up in a bilingual dictionary and augmented with their corresponding translation in the target language. The following example is self explanatory:

```
^.<sent>/.<sent>$
[][<p>]
^Yesterday<adv>/Ayer<adv>$
^prpers<prn><subj><p1><mf><sg>/prpers<prn><tn><p1><mf><sg>$
^read<vblex><past>/leer<vblex><past>$
```

```
^a<det><ind><sg>/uno<det><ind><GD><sg>$
^book<n><sg>/libro<n><m><sg>$
^.<sent>/.<sent>$
[][<\/p>
]
```

Note that all words have been successfully translated into their Spanish counterpart; in the event of a lookup fail, the token is not translated and will be marked by prepending an `@` sign.

Structural transfer

The structural transfer stage applies grammar transfer rules from the source language into the target language. Notice, in the following example, the removal of the pronoun "`I`" because in Spanish it can be inferred from the verb form. The grammar transfer rules are encoded in multiple manually produced XML files (the process actually consists of three different stages: *transfer*, *interchunk* and *postchunk*, each one of them with focus on different types of grammar transformations).

```
^.<sent>$
[][<p>]
^Ayer<adv>$
^leer<vblex><ifi><p1><sg>$
^uno<det><ind><m><sg>$
^libro<n><m><sg>$
^.<sent>$
[][<\/p>
]
```

Also, note the disappearance of the source language forms - from this stage on, we only deal with the target language.

Morphological generator

The morphological generation stage transforms each token according to its associated tags into the correct form of the word in the target language. Consider the output of this stage for our example:

```
.[][<p>]Ayer leí un libro.[][<\/p>
]
```

The infinitive form of "`leer`" (*to read*) has been transformed into its past tense, first person, singular form "`leí`" (inferred from the tags "`<ifi><p1><sg>`"), in accordance to the Spanish morphology.

Post generator

The post generator performs different language specific orthographic operations on the translated text based on rules from a manually produced XML file. The operations include contractions, apostrophations (French *j'adore*) / hyphenation (Romanian *s-au*) or epenthesis (English *a apple* → *an apple*) [13]. There are no such operations to be performed on our example, so the output of this stage is the same as the previous one:

```
.[][<p>]Ayer leí un libro.[][<\/p>
]
```

Reformatter

The reformatter reverts the changes made to the document at the deformatting stage. Using the HTML reformatter, we get the following final result (correctly translated from English to Spanish, with HTML format preservation):

```
<p> Ayer leí un libro </p>
```

2.3 Rules in Machine Translation

In this section we are going to detail the use of rules in a RBMT system, with examples from the Apertium translation engine. Although many stages of the pipeline make use of rules, we will briefly discuss lexical / structural transfer and then focus on PoS disambiguation. By studying the transfer we can get an idea of the expressiveness of the rules, while the constraint grammar style PoS disambiguation rules will serve as a starting point for our own disambiguator design.

The lexical transfer rules analyze each token in the stream and provide translations for each word / expression (usually looking them up in a bilingual dictionary). Because they are applied on a single token at a time they cannot do word reordering, but are able to add, remove or change tags (although the decisions will not have any information from the the surrounding tokens or their properties). This restriction is in place in order to create a clear separation between lexical transfer and structural transfer, leading to higher modularity.

Structural transfer rules are able to process multiple tokens at once and have access to the full stream of tokens when performing transformations (the previously mentioned restriction is not in place anymore). Because the rules can also do word reordering, they are used for grammar transfer and rarely provide word to word translations (this is nevertheless possible, but all such rules should be pushed into the lexical transfer stage). A simple example where word reordering is needed is the following English to German translation: "Today I *(subj)* went *(pred)* home" → "Heute ging *(pred)* ich *(subj)* nach Hause."; this is because in German, for stylistic reasons, the relative order or the subject and predicate in a sentence differs from the English one (**PS** compared to **SP**). Thus, a simple structural transfer rule could be "Identify the subject and predicate and swap them".

There are different algorithms that can be used to apply rules for part of speech disambiguation. All of the algorithms make decisions of tag addition / removal based on the context of a token, namely its surrounding tokens and their respective tags (if the tags could have been unambiguously determined only by looking at isolated tokens, it would have been done in the morphological analysis step).

The Brill tagger has been introduced in Eric Brill's PhD thesis [7] in 1992 and is one notable example for rule based PoS disambiguation. It is able to achieve good disambiguation accuracies by following an interesting rule application policy: it tries to identify and change incorrect tags according to their context and some predefined rules until convergence. In the context of the Brill tagger, rules are also called patches, as they "patch" (make corrections) on tokens.

Some of the following rules have been included in the original paper introducing the Brill tagger [7] (rules 4-9 were skipped, but the original numbering has been kept):

```
(1)  TO IN NEXT-TAG AT
(2)  VBN VBD PREV-WORD-IS-CAP YES
(3)  VBD VBN PREV-1-OR-2-OR-3-TAG HVD
(10) NP NN CURRENT-WORD-IS-CAP NO
```

A natural language rewriting of the first rule would be "replace `TO` with `IN` if next token has the tag `AT`". The general form of the patches in natural language is "replace `TAG` with `OTHER_TAG` if `SOME_TEST`". While the first patch tests for a tag, in the second rule we see that the test is `PREV-WORD-IS-CAP YES`, which translates to "the previous word starts with a capital letter". As opposed to `(2)`, rule `(10)` will be applied whenever the test is false (note the `NO` at the end which implies that `CURRENT-WORD-IS-CAP` is false). Lastly, rule `(3)` looks for the `HVD` tag in any of the previous three tokens. [7]

This selection of rules shows us some of the key aspects of Brill style rules: tag replacing, tests based on the context and negation. The Brill tagger needs in practice a few hundreds of rules either written by linguists or mined from a tagged corpus using machine learning (for comparison, the original paper states an error rate of less than 5% for the English language with only 71 rules).

Another example is Constraint Grammar, which will be discussed in more detail in the next section.

2.4 Constraint Grammar

The concept of Constraint Grammar was first introduced by Fred Karlsson in 1990 [8]. Pasi Tapanainen designed an improved version of CG-1 for his book in 1996 (CG-2) [9]. Apertium uses a newer version of constraint grammar, namely CG-3, provided by an open source implementation of CG-2 with further improvements. The use of CG-3 has been demonstrated in [17], where the author performed grammatical analysis on

Portuguese text. This paper [18] shows great improvements over a statistical disambiguation approach by including CG-3 rules (the article states that rules were written in about 20 hours by linguists).

We will briefly discuss the following rules that are implemented in Apertium for the language pair `hbs-slv` (Serbo-Croatian → Slovenian):

```
# Rule 1: Proper noun / Adjective is Proper noun.
SELECT:NP NProp
    IF (0C NProp OR A) ;

# Rule 2: Noun after noun is genitive.
SELECT:Noun_Noun<gen> N + Gen
    IF (-1 N) ;

# Rule 3: Modified word gender/number/case cleaning.
REMOVE:ModifiedCase Nomen + $$CASE
    IF (-1 Modifier) (NOT -1 Modifier + $$CASE) ;
```

First rule instructs the parser to always select a `NProp` tag (proper noun) over an `A` (adjective) tag. The condition following the `IF` requires that the current token (relative position 0) is ambiguously tagged with `NProp` and `A`. The second rule selects a reading that contain both `N` and `Gen` tags whenever the previous token (the one in position -1 relative to the current token) is a noun. The third rule is more complex and has the effect of removing the `$$CASE` value if the previous token is a modifier which does not have the respective `$$CASE`. This heuristic is based on the fact that the `Nomen` and its modifier should be in accordance with respect to gender, number and case.

These three rules contain examples of the two basic actions in CG (reading selection and removal), multiple tests, references to tokens from the context (relative positions to the current token), logical negation and the use of tag variables.

In order to produce more compact and readable rules the CG framework allows the definition of lists and sets of tags. Note that these definitions are not necessary, they only provide a more structured way of writing rules:

```
# Words that have gender, number and case:
LIST Nomen = top al ant cog n np prn adj;

# Words that agree in case, number, gender with
# the following word.
SET Modifier = A | Num | DemPron | PosPron | IndPron;
```

Hand written rules used in combination with the Constraint Grammar framework can achieve high accuracies, but there are still a few issues to deal with. One problem is the need of linguists who have a good understanding of natural language grammars and are trained in both the source and target languages. Another issue is the maintainability of the rules; as the set incorporates more and more rules, overlaps and clashes between them becomes a real problem (this is partly solved by prioritizing rules with respect to the time when they were added). In order to maintain the high accuracy and make the process fully automatic, research has been invested lately in machine learning of Constraint Grammar style rules.

The research done in this thesis shows that there is potential for automatically learning rules using Meta Interpretive Learning, an Inductive Logic Programming framework. Our disambiguators will use two fundamentally different rule designs which are less expressive versions of CG style rules (our rules can be translated into CG rules, but the other way around is not possible).

Chapter 3

ILP. Meta Interpretive Learning

Inductive Logic Programming (ILP) has been introduced in a research paper by S. Muggleton in 1991 [19]. The field of ILP finds itself at the intersection between machine learning and logic programming [20], by providing us with methods to learn theories in the form of logic programs. In the following definition of Machine Learning "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" [21] it is not clearly specified what T and E might be. The most important thing that logic learning achieves is the unification (in terms of representation) of algorithm, data and hypothesis: all of them can be described as logic programs. Inductive Logic Programming builds up on this and explores ways to compute a suitable hypothesis from examples. This paper [22] brings to the surface further details and provides the reader with applications where the unified representation is important.

A typical ILP setting uses the following three concepts: *background knowledge* (B), *examples* (E) and *hypothesis* (H). Background knowledge can be seen as a collection of facts (logic statements) known in advance. With the use of B, the hypothesis needs to unify with all positive examples (E^+) and fail for all negative examples (E^-). The formal way of expressing this is described by the following equation (we say that B and H entail E) [23]:

$$B \land H \models E \tag{3.1}$$

One way to approach the problem would be to formulate and test all possible hypotheses H. We can avoid doing so by performing a directed search from the (B, E) tuple, since the previous equation is equivalent (as we also learn from [23]) to

$$B \land \overline{E} \models \overline{H} \tag{3.2}$$

One notable example of ILP system that implements this search is the Progol system, also being described in [23]. These systems can be used to learn almost any logic program that a human engineer could write. By using suitable background knowledge and no more than 10 examples, the following hypothesis has been derived [20]:

```
qsort([], []).
qsort([X|T], S) :- part(X, T, L1, L2),
                   qsort(L1, S1),
                   qsort(L2, S2),
                   append(S1, [X|S2], S).
```

In this example the background knowledge implemented the definitions of the two predicates used in the body of `qsort`, namely `part` (which partitions the elements of T in lists L_1 and L_2 with respect to the chosen pivot X) and `append` (which produces the result S as the concatenation of S_1, the pivot X and S_2).

Inductive Logic Programming could prove a great tool for Machine Translation from many perspectives and there have been multiple attempts at learning rules for a Part-of-Speech tagger. In [10], for instance, the author provides a simple language grammar for the disambiguator and uses ILP to identify patterns in the data and build a logic program that will decide which tags to remove from each word. Note that in this case the task of a human operator is not completely eliminated. The major issue with this approach is that it is not language agnostic: the grammar encoded as background knowledge is language specific and needs to be produced by a human operator. Nevertheless, this approach yielded good results in terms of accuracy, performing almost as good as other state of the art Part-of-Speech taggers. Another interesting example and more relevant for our case is [11]. Here, the target was to demonstrate the possibility of learning Constraint Grammar style disambiguation rules using Inductive Logic Programming. In contrast with [10], the authors did not introduce any grammar rules in the background knowledge. The reported accuracy on a large training set was 97% which, by their definition, does not mean complete disambiguation, but careful removal of wrong readings (i.e. some of the words were still ambiguously tagged, but they were counted as accurate whenever they still retained the correct reading). The final logic program was also not very compact, consisting of about 7000 rules.

As it also results from the previously cited experiments, rule-based PoS tagging can be easily represented in terms of logic programs, mainly because rules can be encoded as logic statements (in the Prolog programming language, for instance, each rule could be represented by a different predicate).

3.1 MIL as an ILP framework

Meta Interpretive Learning is a framework for ILP that emerged in the recent years. It arose from research in the field of ILP focused on two key concepts: learning of recursion and predicate invention, both being previously underexplored despite initial interest [24].

While ILP systems such as Progol use inverse entailment to produce a single clause from a single example, Meta Interpretive Learning uses logical abduction to build H as a solution to the set of positive examples E^+ (while checking the integrity of the theory against the set of negative examples) [24].

A subtle refinement is that B (the background knowledge) can be written, in the case of MIL, as the tuple $B = (B_M, B_A)$, where B_A is represented by the building blocks of the hypothesis (regular predicates, same as before), while B_M defines the metarules (rules for building the hypothesis). This further refinement gives us control over how the hypothesis might look like, while also providing more structure to the learner in general by making a clear distinction between the "building blocks" (B_A) and the "binding material" (B_M).

In [24] we find an interesting example of learning a FSM (Finite State Machine) only from positive examples. It is possible to only supply positive examples to any machine learning algorithm at the expense of over-generalization. Although it depends on the background knowledge, this is less likely in the case of MIL because of its inductive bias. The inductive bias of a machine learning algorithm is represented by the constraints used to build the hypothesis (which, in turn, is used to output prediction for previously unseen examples) [25]. One of the simplest examples of inductive bias is known as "Occam's Razor" which is equivalent to the popular "entities should not be multiplied beyond necessity" phrase (John Punch, 1639). In our case, the reduction of the hypothesis is achieved through iterative deepening (the size of the theory is increased by one each time the learner fails to find a hypothesis of the current size [1]). Metarules and background knowledge also introduce inductive bias and keep the hypothesis from being over-general. It is nevertheless still possible to produce overgeneralization: for instance we could define one predicate that states "Everything is true."; then indeed, with a suitable metarule and no negative examples, this predicate could be used by the MIL learner to build a theory with one clause that is always true.

3.2 Metagol_D - Dyadic Metagol

While the hypothesis space searched by MIL has many interesting properties, a more recent paper [26] introduces the Metagol$_\mathrm{D}$ system (Dyadic Metagol) and demonstrates two of its key properties. First, recursive logic programs learned using the Metagol$_\mathrm{D}$ will always terminate (this is achieved by ensuring certain ordering constraints of the predicates in the body of a clause). Secondly, a logarithmic bound in the number of examples of the hypothesis size is enough to PAC-learn [2] the H_2^2 space (H_j^i is defined in the paper as the space of all hypotheses that consist of predicates of arity at most i, while having at most j atoms in their bodies).

The name of the system, Dyadic Metagol, comes from the fact that it explores the H_2^2 space mentioned above: its metarules are based solely on predicates of arity one (we call them monadic predicates) and two (dyadic predicates). In [26] the authors designed a simulated experiment to prove that a robot would be able to learn how to build a stable wall using Metagol$_\mathrm{D}$. The final hypothesis is presented below (has been taken from the cited paper):

[1]By size we refer to the number of clauses in the theory.

[2]PAC - Probably Approximately Correct

```
buildWall(X, Y) :- a2(X, Y), f1(Y).
buildWall(X, Y) :- a2(X, Z), buildWall(Z, Y).
a2(X, Y)        :- a1(X, Y), f1(Y).
a1(X, Y)        :- fetch(X, Z), putOnTopOf(Z, Y).
f1(X)           :- offset(X), continuous(X).
```

Notice predicate invention (`a2`, `a1` and `f1`) and the use of recursion in the second clause (`buildWall`). Predicates `fetch`, `putOnTop`, `offset` and `continuous` form the background knowledge and act as fluents (sensors) and actions that can be carried out by the robot. This logic program enables a robot to make a plan for building a stable wall from a heap of bricks.

We are now going to discuss metarules in more detail with respect to Metagol$_D$'s implementation in Prolog. A metarule is a Prolog fact (clause with no body) of the following form:

```
metarule(MetaruleName, MetaSubst, Rule, PreCond, Prog).
```

The following is a code example of an actual metarule that implements a post checked action (this metarule fits, for instance, the definition of `a2` predicate in the stable wall building example):

Listing 3.1: Metarule example

```
metarule(
  post_checked_action,                             % Metarule name.
  [P/2, Q/2, R/1],                                 % Metasubstitutions
  ( [P,X,Y] :- [[Q,X,Y]-true, [R,Y]-true] ),       % Rule.
  (
    pred_above(P/2, Q/2, Prog),                    % Preconditions.
    pred_above(P/2, R/1, Prog)
  ),
  Prog).                                           % Program.
```

The `MetaruleName` is only used to specify to a meta-interpreter which are the metarules that it can use to build the hypothesis. `MetaSubst` contains a list of variables that are going to be bound by the meta-interpreter to actual predicates from the background knowledge; each of the variables has an arity associated and it will be bound only to those predicates from B_A that match the respective arity. The `Rule` part contains the structure of the clause that will be abduced by Metagol$_D$. In the example above, it states that the rule should be of the form `P(X, Y) :- Q(X, Y), R(Y).` with postconditions `true` for `Q` and `R`. The postconditions can perform complex tests to ensure some needed external consistencies (i.e. not derived from examples); note that these postconditions can make use of everything present in the clause. `Preconditions` are implemented here as a quick test on the current metasubstitutions. They are used in the example above for a precedence test that ensures that P will be always above Q and R, thus not producing recursion (demonstration is presented in [26]). Finally, `Prog` serves as an internal state of the meta-interpreter and will not be given by the programmer.

Metagol$_D$ can be used for robot planning, as we saw before, if we think of monadic predicates as *fluents* and dyadic predicates as *actions*. All these predicates will receive a *state of the world* and will either test or change it. The concept of background knowledge becomes more clear in this context: the robot has to implement all the actions it can carry on and all sensors. All we need is an encoding of the state of the world, same for all predicates and some metarules. The rule structures we present below are adapted from an older version of Metagol$_D$ written by Stephen Muggleton in Prolog and also referenced in the paper [26].

```
SimpleAction(S1, S2)   <-
CheckedAction(S1,S2)   <- Action(S1,S2), PostCondition(S2)
GuardedAction(S1,S2)   <- PreCondition(S1), Action(S1,S2)
ComplexAction(S1,S2)   <- SubAction1(S1,S3), SubAction2(S3,S2)

SimpleCondition(S1)    <-
ComplexCondition(S1)   <- SubCondition1(S1), SubCondition2(S1)
PostCondition(S1)      <- Action(S1, S2), Condition(S2)
```

This example has a good expressive value because it shows us how actions and fluents can be put together in meaningful ways. Also, this set of metarules is enough to learn all the H_2^2 space since it contains all the possible *useful* combinations of predicates of arity at most two with at most two clauses in the body. By *useful* we mean that some combinations are missing: predicates with one clause in the body are not present. This was intended, take for instance `Cond(X) :- Cond1(X).`: `Cond` is never useful since we could just directly use `Cond1` instead every time we needed, and the same goes with others.

3.2.1 Concept learning

We will call the task of learning monadic predicates *Concept learning*. This is because the only argument of the monadic predicate can be a concept on which it will decide, returning true or false depending on whether the unification has succeeded or failed, respectively.

The hypothesis, however, does not need to consist only of monadic predicates (simple and complex conditions): we can imagine a scenario in which we want to learn whether a car looks good in the color red or not. A simple way of doing so is to encode the state of the car, have a dyadic predicate that paints it and then a monadic one that checks whether it looks good or not:

```
looks_good_red(Car) :-
    paint_red(Car, RedCar), looks_good(RedCar).
```

The metarule we need to use is listed in the previous subsection as *PostCondition* and proved to be particularly useful for the first part of our project. We will implement our first part of speech disambiguator around this idea of concept learning: given a context (a list of words) the action will extract one word and the condition will check the existence of a certain tag.

The same approach has been used in the two previously mentioned research papers that dealt with PoS tagging [10, 11]: the learned rules were, in fact, predicates for identification of concepts (such as nouns, adjectives) or for making the decision of reading removal.

3.2.2 Action learning

When we say *action learning* we mean learning of dyadic predicates. Again, this does not mean that the theory should consist solely of dyadic predicates, but merely that the specific predicate that we want to learn has an arity of 2, as in the case of the *wall building robot* example.

While monadic predicates can model concept identification as we illustrated before, dyadics can be thought as actions that change the state of an object. While there are no research papers (to the best of our knowledge) that target directly at learning dyadic predicates for part of speech tagging disambiguation, we found a way to make use of them: instead of learning rules for identifying concepts based on their properties and the context around them and pairing these rules with only one, well defined action (such as the reading removal in [11]), we let the MIL learner to also learn the transformation that should be applied on the words (selection / removal of readings). This way, part of the disambiguator is in the rules themselves – they are the ones that directly transform the stream of words, while the rest of the algorithm only selects and applies rules until no further change is possible. We think this is a better approach and the ultimate goal for MIL would be to learn the whole disambiguator implementation from examples, not just rules. This is not possible at the moment due to extremely poor performance when dealing with large hypotheses.

3.2.3 Breadth First Search example

While experimenting with Metagol$_{\mathrm{D}}$ we decided to test it on a simple example: learning of the BFS algorithm.

First of all, we need to define the state of the BFS algorithm. We encoded this state as a list of three elements, $[Q, V, G]$. Q will act as a queue, V is the list of nodes visited up to a point and G contains the graph as a list of $[U, [V_1, V_2, ...]]$ (U is a node in the graph, while $V_1, V_2, ...$ its neighbors). One example is the following list which encodes the graph from figure 3.1 with $Q = \{3, 4\}, V = \{1, 2, 3, 4\}$:

```
[[3, 4],              % Queue.
 [1, 2, 3, 4],        % Visited nodes.
 [[1, [2, 3]],        % Graph adjacency lists.
  [2, [3, 4]],
  [3, [4]],
  [4, []],
  [5, []]]]
```

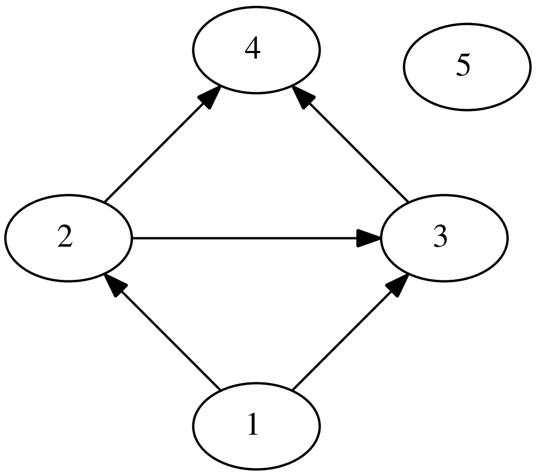

Figure 3.1: Example graph.

Now for the predicates that operate on this encoding of the BFS state, making up the background knowledge:

- `queue_empty` – extracts Q from the state and succeeds if it is an empty list.
- `queue_not_empty` – extracts Q from the state and succeeds if it is not an empty list.
- `add_neighbor_nodes_to_queue` – extracts the first node in Q and pushes all its neighbors at the end of Q.
- `remove_queue_duplicates` – removes any duplicate nodes from the queue (keeping only the first occurrence).
- `pop_queue` – removes the first element from Q.
- `mark_queue_visited` – adds all elements in Q to V and removes the duplicates from V (set reunion of Q and V in V).

The only monadic predicates are `queue_empty` and `queue_not_empty`, with the rest of them being dyadic.

Metagol$_{\mathrm{D}}$ was able to learn the following hypothesis from only three positive examples (pairs of initial and final states of the BFS algorithm) and no negative examples:

Listing 3.2: Breadth First Search – Generalised Metagol

```
% Episode 1
bfs_step(A,B) :- bfs_step_1(A,C), bfs_step_2(C,B).
bfs_step_1(A,B) :-
  add_neighbor_nodes_to_queue(A,C), remove_queue_duplicates(C,B).
bfs_step_2(A,B) :-
  mark_queue_visited(A,C), pop_queue(C,B).
```

```
% Episode 2.
bfs(A,A) :- qempty(A).
bfs(A,B) :- not_qempty(A), bfs_step(A,C), bfs(C,B).
```

Please note the `Episode x` comments in listing 3.2. This matter will be treated in the next subsection.

3.2.4 Episodic learning

The idea of episodic learning in MIL is useful whenever we want to learn a chain of clauses that depend one on the other. Take the BFS example in 3.2: in the first episode we provide examples for one step of the Breadth First Search algorithm while in the second episode we provide examples for the whole algorithm. The MIL learner will link the two episodes and try to reuse all the predicates it created in previous episodes. This is why it is able to use `bfs_step` in the definition of `bfs` – we did not explicitly include it in the background knowledge, but it was created in a previous episode.

In the case of a robot learning its way in on a 2D grid the episodic learning can be used to build an exponential ladder of movements:

- Episode 1 – the learner gets examples for moving one position to the right and learns the `move_right_1` predicate.
- Episode 2 – the learner gets examples for moving two positions to the right, reuses `move_right_1` and learns the `move_right_2` predicate.
- Episode 3 – the learner gets examples for moving four positions to the right, reuses `move_right_2` and learns the `move_right_4` predicate.
- Episode N – same process is applied, the learner will output a hypothesis for moving 2^{N-1} cells to the right.

Using the predicates learned in previous episodes, the robot can now combine, for instance `move_right_2` and `move_right_4` to move 6 positions to the right. In this way, any movement up to $2^N - 1$ is possible in just $N + 1$ episodes (N for learning the exponential ladder and one for the target movement, if the number of cells is not a power of two).

Chapter 4

Learning Prolog disambiguation rules

4.1 General description

Throughout this chapter we are going to discuss two different PoS disambiguators written in Prolog, along with their respective rule designs and learning processes. Although the two disambiguators are different in some respects, both follow the same general idea of identifying suitable contexts in which some rules can be applied. Rules are mined using random examples fed into a MIL learner and then selected by two different algorithms, the most notable one being a genetic algorithm.

The PoS disambiguation stage comes immediately after morphological analysis in the translation pipeline. Let's consider the following example which has been ambiguously tagged in the previous stage ("Yesterday I read a book"):

```
^Yesterday/Yesterday<adv>$
^I/I<num><mf><sg>/prpers<prn><subj><p1><mf><sg>$
^read/read<vblex><inf>/read<vblex><pres>/
    read<vblex><past>/read<vblex><pp>$
^a/a<det><ind><sg>$
^book/book<n><sg>/book<vblex><inf>/book<vblex><pres>$
^./.<sent>$
```

This constitutes input for our disambiguator which should remove wrong readings and output the unambiguous version. First, we use Prolog's DCGs (Definite Clause Grammars) to parse the raw text and get lists of split tokens as follows:

- isolate **tokens**:
  `["Yesterday/Yesterday<adv>", ...]`
- split tokens into **readings**:
  `["I", "I<num><mf><sg>", "prpers<prn><subj><p1><mf><sg>"]`

- augment each reading with **metadata**:
  `["I<num><mf><sg>", ["num", "mf", "sg"]]`

The last step is necessary for performance reasons: most of the time rules will lookup tags inside tokens, so preprocessing and adding them as metadata will save computing time later in the disambiguation process.

The input reader module will present us with a list of tokens on which rules can now be applied. A context is represented by a list which contains a central token (to be disambiguated) and some of the tokens around it. In our case we will always take into consideration the same amount of tokens to the left and to the right of the word that needs disambiguation, which we call context radius. More formally, a context of radius r around token T_i is:

$$[T_{i-r}, ..., T_{i-1}, T_i, T_{i+1}, ..., T_{i+r}] \tag{4.1}$$

Disambiguation rules are Prolog predicates that operate on contexts. They are produced by an MIL learner and, in our implementation, can be monadic or dyadic. A monadic rule is actually a test that is performed by the disambiguator, while the dyadic rules perform context transformations (note that in order to produce disambiguation some readings need to be removed; in the case of monadic rules the action of selecting some reading is performed whenever the predicate is true for a certain context). Example rules:

```
(1) concept(concept_2867, "n").
    concept_2867(C) :- post_token(C, T), is_sent(T).
(2) rule_3(A, B) :- one_part_b_n(A), cg_remove_p1(A, B).
    token_b([_, B, _, _, _], B).
```

The first rule is specific to the first disambiguator that we implemented as a proof of concept. The predicate `concept` links the test `concept_2867` to the tag `n` - this way the disambiguator knows that it has to select readings that contain this specific tag whenever `concept_2867` unifies. Here we worked on contexts of radius 1 (lists of three elements, $[T_{pre}, T_{central}, T_{post}]$) and the predicate `post_token` extracts T_{post} from the context, while `is_sent` checks whether the extracted token contains the tag `sent`. Overall the rule can be translated as *"select noun (remove every other reading) whenever the central token is the last word in a sentence"* (`sent` tags mark sentence boundaries). An intuition of this heuristic rule is that sentences often end with a noun ("Yesterday I read a book.").

The second rule (corresponding to the second disambiguator that we wrote) contains one test (`one_part_b_n`) and one action (`cg_remove_p1`). It works on contexts of radius 2 encoded as $[T_a, T_b, T_c, T_d, T_e]$, where T_c is the central token. The example also includes the implementation of the helper predicate `token_b` which extracts the token in position b (can be read as *the previous word*). So, overall, the rule says *"if the previous word can be a noun (has at least one reading containing* `n`*) remove readings that imply first person from the current token"*. The intuition behind the rule might be that the noun preceding a verb is often the subject, so the number of the verb can't be first person ("The horse fell.").

After the disambiguators repeatedly select and apply these rules we get less and less ambiguity (readings get removed). The algorithm stops when no further changes can be made to the stream of tokens.

In order to produce candidate rules we use the MIL (Meta Interpretive Learning) framework. The process can be summarized as follows:

- **Rule design.** We can see from the example above that the design of rules is specific to each disambiguator. The idea we had in mind in the first case was a monadic predicate that would be able to decide the part of speech (and its attributes) based on a 1-radius context around the central token. In the second case we wanted to learn Prolog programs that performed Constraint Grammar style transformation (reading selection / removal) by analyzing a 2-radius context around the central token (leading us to a rule represented as a dyadic predicate with a few tests in its body followed by a `cg_select` / `cg_remove` action).

- **Background knowledge.** As background knowledge for the MIL learner we had to implement helper predicates (building blocks for the actual rules). We can distinguish between two types of background knowledge: hand written and automatically generated. The hand written predicates are mainly used for the extraction of tokens from contexts (such as "`pre_token([C, T]) :- C = [T, _, _].`"), while the automatically generated background knowledge provides wrappers like "`is_pr(Token) :- token_has_tag(Token, "pr").`", for all possible tags.

- **Metarules implementation.** For each type of disambiguation rule we need to implement a metarule that tells the MIL learner what is the structure of the rule we wanted to produce. While the metarules for the first disambiguator are simpler, in the second case they also include post tests that ensure certain consistency properties.

- **Example acquisition.** At this point we have everything we need to start learning rules from examples. For this step we have implemented example generators that use some filtering and randomly draw instances of contexts that are fed into the MIL learner. The training corpus consists of aligned files of ambiguously tagged and handtagged texts.

- **Learning of rules.** The actual rules are learned by feeding the randomly generated examples to the learner and checking whether it was able or not to find a suitable hypothesis. This is the most time consuming step because it involves the probability that a randomly generated example will actually produce any rule at all. In order to lower the time we introduced some simple filters, such as isolating examples by their ambiguity class (for instance, the example generators are able to select only those contexts where we need to decide between a noun and a verb).

Because of the randomly selected contexts that make up the examples we can end up with some less accurate rules for the disambiguators. The extra step that we took to take care of this is **rule selection**. The rule selection receives all the rules that have been mined by the MIL learner and selects a subset of them that provides the best accuracy on the training data. This enables us to remove unnecessary rules (thus speeding up the disambiguation process by having to check less rules) and also assigns priorities in the case of the second disambiguator.

4.2 Machine learning framework setting

The data we used in our experiments for training and evaluation has been made available by the Apertium open source organization under the GNU GPL v2 license. It consists of a series of texts ambiguously tagged by the morphological analyzer which are aligned with unambiguously hand tagged counterparts.

At training time, we present the MIL learner with pairs of contexts (ambiguous and hand tagged) and expect it to form a hypothesis (rule) that fits the supplied pairs.

At evaluation time, we supply the ambiguously tagged testing data to our disambiguators (along with learned rules) and store their prediction. An evaluation script compares our prediction with the hand tagged file and counts the errors. The script computes two different metrics:

- **Overall accuracy.**

  $$A_{overall} = 1 - \frac{E_{count}}{N_{total}} \tag{4.2}$$

  where E_{count} is the number of errors the disambiguator made (differences from the handtagged files) and N_{total} is the total number of tokens in the input file. It can be thought as "the fraction of correct, unambiguous tokens from all the tokens".

- **Disambiguation accuracy.**

  $$A_{disambiguation} = 1 - \frac{E_{count}}{N_{ambiguous}} \tag{4.3}$$

  where E_{count} is the same as above (error count) and $N_{ambiguous}$ is the number of ambiguous tokens in the input file. This accuracy is strictly related to disambiguation: it reveals what fraction of the ambiguous tokens has been correctly disambiguated.

The first metric is always higher than the second one (since the disambiguators do not change already unambiguous tokens in the input) and is the one usually reported in research papers. The second metric allows us to take a better look at the disambiguation process alone, eliminating the already unambiguous tokens from the equation.

In our experiments, we employed a 3-fold cross validation. We split the data in three equal parts and we always use two of them for training and one of them for testing.

We repeat the process three times with a different fold left out for testing. This way we ensure that we use all examples for testing at least once.

In order to be able to produce background knowledge for the MIL learner we needed to implement tools for extracting relevant information from the data, such as ambiguity classes, most frequent lemmas and tags.

We found out that we have to deal with total number of 60 tags: parts of speech (`adj` - adjective, `n` - noun, `vblex` - lexical verb, `det` - determiner) and their attributes (`past` - past tense, `sg` - singular, `f` - feminine, `gen` - genitive case).

Some of the rules run tests against the lemma part of the token (the word in the original text without tags: `^I/prpers<prn><subj><p1><mf><sg>$` → `I`). These rules tend to overfit the data, but they might as well prove useful when perform tests on frequent words (articles, determiners). The following table illustrates the 5 most frequent lemmas in our corpus (along with their respective counts):

Lemma	Occurrences
`the`	986
`of`	644
`and`	550
`to`	456
`in`	430

Table 4.1: Top lemmas from the dataset.

An ambiguity class is a collection of tags that always appear together in the same token after the morphological analysis. Ambiguity classes prove to be useful at the example generation step - filtering by ambiguity class reduces the probability of getting examples that contain completely unrelated contexts, thus increasing the probability that the learner will produce a valid (and useful) hypothesis. Note that this kind of filtering is only used for the second disambiguator that uses dyadic predicates as rules. The following table contains the top 5 ambiguity classes:

Ambiguity class	Occurrences
`<vblex><past>/<vblex><pp>`	1335
`<vblex><inf>/<n><sg>/<vblex><pres>`	1005
`<adj>/<n><sg>`	813
`<adv>/<pr>`	556
`<vblex><inf>/<vblex><pres>`	423

Table 4.2: Top ambiguity classes from the dataset.

4.3 Baseline disambiguator implementation

The accuracy of our disambiguators will be compared to a baseline implementation (discussed in detail in this section) and the HMM (Hidden Markov Models) disambiguator described in this paper [6] and currently implemented as part of the Apertium translation engine.

The baseline implementation is based on a probabilistic unigram language model: a token is disambiguated to the reading that provides the highest probability of the tags. Let's consider the following token: $X = \{R_1, R_2, ..., R_r\}$. $R_i \in X$ is a reading and can be viewed as a set of tags: $R_i = \{T_1, T_2, ..., T_t\}$; we want to choose the reading that has the maximum probability with respect to its tags.

Per tag probabilities are computed based on their frequencies in the hand tagged files as follows:

$$p(T) = \frac{N(T)}{N} \tag{4.4}$$

where $N(T)$ is the number of times tag T appears in the hand tagged tokens and N is the total number of tokens.

We can view a reading as a set of tags $R = \{T_1, T_2, ..., T_k\}, |R| = k$. The probability of a reading R is defined as:

$$p(R) = \prod_{T \in R} p(T) \tag{4.5}$$

The baseline disambiguator will then choose a reading $R \in X$ such that $p(R) \geq p(R'), \forall R' \in X$.

In order to avoid rounding errors we can apply ln in equation 4.5 and use the additiveness property of logarithms to transform the product into a sum as follows:

$$ln(p(R)) = ln(\prod_{T \in R} p(T)) \tag{4.6}$$

$$ln(p(R)) = \sum_{T \in R} ln(p(T)) \tag{4.7}$$

And since the natural logarithm is a strictly increasing function we continue to select the reading R with the highest log probability. More formally, we now choose $R \in X$ such that $ln(p(R)) \geq log(p(R')), \forall R' \in X$.

The effect of this probabilistic model on the data is that it will always return the same, most probable reading from each ambiguity class. The following table illustrates this on the most frequent five ambiguity classes:

The baseline has a disambiguation accuracy of 60.60% and an overall accuracy of 89.05%. When compared against the hand tagged data, the raw output from the morphological analyzer has an overall accuracy of 72.30% (which means that this percentage of tokens came out unambiguous after the dictionary lookup in morphological analysis stage of the translation pipeline).

Ambiguity class	Selected reading
`<vblex><past>/<vblex><pp>`	`<vblex><past>`
`<vblex><inf>/<n><sg>/<vblex><pres>`	`<n><sg>`
`<adj>/<n><sg>`	`adj`
`<adv>/<pr>`	`<adv>`
`<vblex><inf>/<vblex><pres>`	`<vblex><pres>`

Table 4.3: Reading selection by the baseline algorithm for top ambiguity classes.

4.4 Using concept learning

General description and rule design

In this section we are going to detail the idea, implementation and results of a disambiguator based on concept learning. This was a proof of concept disambiguator that was implemented as the first part of the project.

This disambiguator works by concept identification - the disambiguator tries to recognize a certain concept in the stream of tokens (a *noun*, for instance) and then removes every reading that does not match the specified concept. Specifically, if the disambiguator recognizes the *noun* concept in a token, then all readings that do not contain the tag `<n>` will be removed:

```
book/book<n><sg>/book<vblex><inf>/book<vblex><pres>
                          →
                   book/book<n><sg>
```

Having in mind that the decision (concept identification) should be made by looking at the context, we decided to implement the rules as monadic Prolog predicates that receive a context as a parameter and unify iff the central token matches the concept. We came up with the following simple rule design:

```
concept(test_pred, "tag1").
test_pred(Ctx) :- extract_token(Ctx, Token), is_tag2(Token).
```

The first part of the rule is a Prolog *fact* that links the predicate `test_pred` to the tag `tag1`. This way we know that the `test_pred` predicate will be used to identify the concept of `tag1`. The disambiguator will try the `test_pred` on each context in the ambiguously tagged stream and whenever it unifies, it will remove all readings from the central token that do not contain the tag `tag1`.

Let's take a look at the body of the `test_pred` predicate. First, it calls an `extract_token` predicate (a choice of `pre_token` / `post_token`), which binds the `Token` atom to either the previous or the next token with respect to the central one from the context. The whole test unifies only if `is_tag2(Token)` is true (actual examples of `is_tag2(Token)` include `is_adj(Token)`, `is_vblex(Token)`, etc.).

A natural language formulation of a rule can be "If the previous token is a noun, the current one must be an adjective". Its Prolog representation would then be:

```
concept(sample_rule, "adj").
sample_rule(Ctx) :- pre_token(Ctx, Token), is_n(Token).
```

It is worth noting that for concept learning disambiguator we are using contexts of radius 1. That is, the disambiguator (as well as the learner) have access to only three tokens from the stream at a time:

$$[T_{pre}, T_{central}, T_{post}]$$

where $T_{central}$ is the token for which we decide which readings to keep and which to remove (this is, actually, the reason for which we have only two predicates for token extraction, `pre_token` $\rightarrow T_{pre}$ and `post_token` $\rightarrow T_{post}$).

Metarules definition

In order to be able to use MIL to learn such rules, we need to specify only one metarule for the learner:

```
metarule1(
  dis1rule,
  [P/1,Q/2,R/1], ([P,X] :- [[Q,X,Y]-true,[R,Y]-true]),
  (pred_above(P/1,Q/2,Prog),
   pred_above(P/1,R/1,Prog)), Prog).
```

In this metarule definition, apart from name, preconditions and program, we specify the metasubstitutions and the rule structure. The metasubstitutions list tells MIL that we are going to use three predicates in the clause we are trying to build: `P` and `R` of arity 1 and `Q` of arity 2. The rule structure informs the learning framework that in order for `P(X)` to be true, `Q(X, Y)` and `R(Y)` need to be both true. In our case, `Q` will be used for token extraction (so `X` is will be the context, while `Y` will be a Token) and `R` will be used for testing the token (replaced by an `is_tag` predicate). Finally `P(X)` is the disambiguation rule we are building (and will be used to identify a certain concept).

Background knowledge

We need to provide the MIL learner with background knowledge in order to be able to substitute predicate variables `Q` and `R` and form theories. For the substitution of `Q` we provide the following hand written dyadic predicates:

```
pre_token (Ctx, Token) :- Ctx = [Token, _, _].
post_token(Ctx, Token) :- Ctx = [_, _, Token].
```

The substitution of `R` requires us to write one predicate per part of speech, so we implemented a script that automatically generates the necessary monadics:

```
is_pr(Token)        :- token_has_tag(Token, "pr").
is_past(Token)      :- token_has_tag(Token, "past").
is_pri(Token)       :- token_has_tag(Token, "pri").
...
```

Note that it is also possible to just generate a list of constants of the form `["pr", "past", "pri", ...]` and then use them in combination with `token_has_tag` directly in the learner (with a suitable metarule). While the two approaches of tag specification are perfectly equivalent we chose the first one because in the second disambiguator the latter would be less efficient.

The background knowledge for this disambiguation approach counts 62 predicates (2 for token extraction and 60 for checking tags). At this point we can feed examples into the MIL learner and record the generated hypothesis (if there is any, unrelated examples might not be explainable in the bounds of the theory size that we impose).

Manual example

In order to get a better understanding of how examples should be built let's take a look at the learning episode in listing 4.1.

Listing 4.1: Manual example for disambiguator 1

```
episode(
  % Concept name.
  concept_n,
  % One positive example.
  [[concept_n,
    [["a", ["a<det><ind><sg>", ["det", "ind", "sg"]]],      % Pre token.
     ["book", ["book<n><sg>", ["n", "sg"]]],                % Central token.
     [".", [".<sent>", ["sent"]]]]]]                        % Post token.
  ],
  % No negative examples.
  []
).
```

From the listing above (and from A.1) we can see that one learning episode consists of multiple positive and negative examples. Each one of these examples represents a different 1-radius context encoded in Prolog lists, as described in the beginning of this chapter.

Note that the MIL learner was intentionally only allowed to use the `post_token` and `is_sent` predicates for this demonstration. Keeping this in mind and looking at the code above we could easily formulate the following heuristic rule in natural language: “A noun reading should be selected whenever it is the last word in a sentence”. In other words, if the next token marks a sentence boundary then the current one is a noun. As expected, Metagol comes up with the same hypothesis:

```
EXAMPLE EPISODE: concept_n
```

```
TRY CLAUSE BOUND: 1
TRY NEW PREDICATE BOUND: 0
TRY METARULE SET: [dis1rule]
FINAL HYPOTHESIS FOR EPISODE: concept_n, BOUND: 1

concept_n(A) :- post_token(A,B), is_sent(B).
```

The task of hand picking examples for Metagol is very tedious and error prone. It is much more easier for a human to check the data and come up with good rules than to manually select and provide examples to a machine learning algorithm. Nevertheless, this provided us with a better understanding of the data and led to improvements of the automated example generators we implemented.

Generating examples and learning

The purpose of any machine learning algorithm is to provide us with parameters such that our model explains the data as accurately as possible. In our case these parameters are represented by the rules we want to learn. Because our model does not take noise into account we can't just feed in all possible examples that fit a certain concept (a noun, let's say) and expect good rules because the algorithm will tend to overfit and it will also take a large amount of time for Metagol to find a suitable hypothesis. Instead, we rely on sampling a few examples each time, feeding them to the learning and recording the theories. This does not eliminate the problem of overfitting, but significantly lowers the execution time. We are going to deal with the problem of overfitting at a later point by pruning the list of learned rules.

Because this disambiguator relies on the idea of concept identification it makes little sense to provide cross-concept examples. That is, we will always try to find sets of examples that will fit the same concept. Having this in mind, we have implemented simple filters for our data that enable us to select only those 3-token contexts in which the central token (in its disambiguated form) fits the concept we want to learn.

We might, for instance, want to learn the concept of *singular* (represented by the `sg` tag) in our sample sentence "Yesterday I read a book.". Since we set the goal ourselves we will only provide examples in which the `sg` tag is present, such as "Yesterday, *I*, read" or "a, *book*, .". We can also learn rules without doing so, we might provide two examples of different concepts and, if our theory bound permits, we will get rules for explaining both; but in this situation we then need to trace back which rule applies to which concepts because in the disambiguator we need to link concept tags to rules through the `concept` fact (above all, as mentioned previously, the less related the examples are, the more time it takes to learn a hypothesis).

Our example generator applies a by-tag filtering first and then randomly samples the pool of contexts. We typically use 8 examples with a theory size bound (number of Prolog clauses in the hypothesis) of 2-3. This is one simple way to avoid (or, at least, to lower the probability of) overfitting: we ask that a small number of rules for concept identification will cover a relatively large number of examples.

For a small automatically generated example, please refer to code listing A.1 in Appendix A. Note that we are also using one negative example because we are trying to avoid theories that fit other concepts as well - the negative example contains, as its central token, one word that could not be found in the dictionary by the morphological analyzer. Metagol will output a rule for concept identification that holds for all three positive examples, but fails to unify when the negative example is supplied. We typically use 3-4 negative examples each time (this has been found to be a good value by trial and error, taking into account execution time and the frequency of failures to find rules that would cover both positives and negatives).

The final hypothesis for the learning episode in A.1 is presented below:

```
FINAL HYPOTHESIS FOR EPISODE: concept_x, BOUND: 1
concept_x(A) :- pre_token(A,B), is_adj(B).
```

This is a good heuristic which says that a noun can be identified if there is an adjective before (obviously, the ambiguously tagged token must have the `<n>` tag associated with at least one of its readings).

Using this method, we were able to mine about 900 rules. As we fed random examples to the learner, some of the rules were low quality, so we had to prune the initial set (this matter is discussed over the next subsection).

Rule pruning

The main idea behind rule pruning was to avoid overfitting and, to some extent, rule overlapping. Since this disambiguator (implemented as the first step of the project) was initially designed to serve as a Proof of Concept, we implemented a simple algorithm for rule pruning just to get an idea of what can be achieved by a rule learning approach to the task of PoS disambiguation.

First, we isolated all the rules that we were able to learn in the previous step and fed them to the disambiguator one by one. This way we could measure the influence that a single rule has on the accuracy. Then, we listed the rules in decreasing order with respect to their accuracy gains and only kept the top K_1 ones.

In order to make sure that each concept has the chance to be identified, we augmented our list with the best K_2 rules for each concept class (again, with respect to the previously computed accuracy gain). Note that concept class actually means the tag that was linked to the rule's identification predicate (such as `<n>`, `<sg>`, etc.).

Finally, we eliminated duplicates from the list in order to make sure that no predicate appears twice in the Prolog code file that was going to be supplied to the disambiguator. We repeated the experiments with different values for K_1 and K_2 and found that good values for our conditions were $K_1 = 60$ and $K_2 = 1$.

Since the total number of rules is

$$N_{rules} = K_1 + N_{concepts} K_2 \tag{4.8}$$

and we had about 60 different tags, we ended up with a total of about 120 rules, not too far from the usual hand written CG-3 rule files.

Results

The disambiguator that we implemented in the first half of the project was designed to be more exploratory than complex. The idea behind it was to prove that it is possible to write a rule based disambiguator in Prolog with machine learned rules. In this subsection we present results solely for the first part of the project (for an overview, comparisons and further discussion please refer to Chapter 6).

The total number of rules we managed to learn using Metagol was about 900 (approximately 15 rules / concept). As mentioned before, in the pruning step we got rid of most of them, keeping only about 120 rules (approximately 13.33% of them).

Table 4.4 shows the accuracy achieved by the set of rules before and after pruning (baseline has been also added for comparison).

	Before pruning	After pruning	Baseline
$A_{disambiguation}$	44.73%	70.66%	60.60%
$A_{overall}$	84.60%	91.83%	89.05%

Table 4.4: Comparative results for PoS tagger 1

While high accuracy was not the main goal, we had better results than initially expected. We can see that our disambiguator produced after rule pruning an improvement of almost 10% over the baseline. The plot in figure 4.1 shows the evolution of the accuracy after adding the top 60 rules by accuracy gain. The blue curve is a 3^{rd} degree polynomial interpolation of the actual data points and helps us to see the general trend.

As we can see in this figure, the first 60 rules are enough to produce an improvement over the baseline by almost over 7% (blue curve above). The rest of improvement comes from the second part of the pruning process in which we have kept the best rule for each concept (green level). It is interesting to note that the first 5 rules seem to have a huge influence on the disambiguation accuracy.

As a short conclusion, we managed to implement a PoS disambiguator in Prolog with a simple rule design. While rules are traditionally manually written by linguists, in our case we proved that they can be machine learned. By using the Metagol system for learning (and with little post processing) we were able to substantially improve the accuracy of our disambiguator over the baseline.

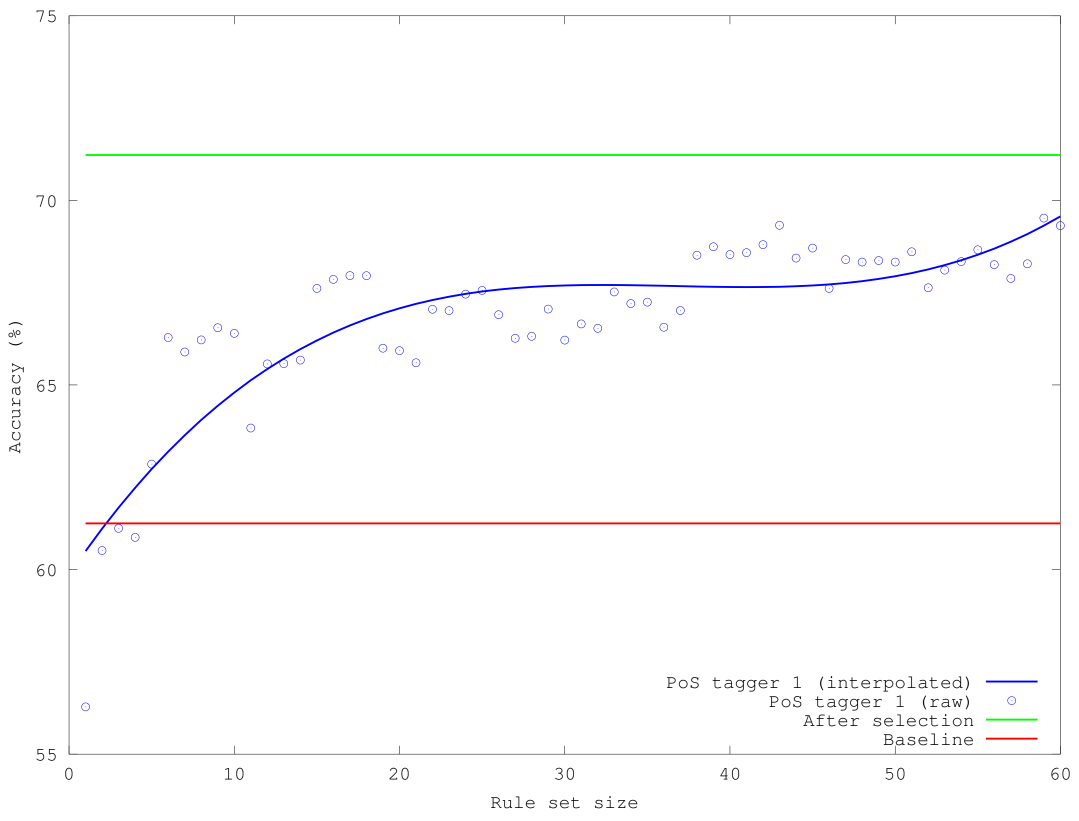

Figure 4.1: Accuracy comparison for PoS disambiguator 1

4.5 Using action learning

General description and rule design

In this section we are going to discuss implementation details and results of the second disambiguator we implemented as part of this project. There are major improvements over the previous PoS disambiguator with respect to metarules (with influence on the rule types), example generator and selection process (equivalent to rule pruning in the other case).

The main idea behind this disambiguator is that its rules also do stream transformations (remember that in the previous case the disambiguator used rules merely to identify concepts and it was always applying the same action). We decided to mimic the Constraint Grammar framework, so the actions that our disambiguator can take are `cg_remove` and `cg_select`.

It is obvious that in this case we can not use monadic predicates and we will need to learn dyadic predicates, as we basically want them to map from an input context to a (at least partly) disambiguated context. As in the previous case we need some tests that guard the decisions of applying `cg_remove` / `cg_select` so the rules will look like this:

```
rule(In, Out) :- test1(In), test2(In), ..., testN(In),
                    action(In, Out).
```

First, how many tests we need in order to produce a good disambiguation? Surely, less tests will produce more generalization, while more tests might be more accurate in some situations (while being prone to overfitting). By looking at human produced rules in Apertium we can see that linguists are able most of the time to achieve very high accuracies (approx. 99%) with no more than 4 tests per rule (usually, one of these tests is reserved to check the central token). We decided, then, to restrict our rules at four context tests, while the first one will always check for a tag in the central token.

At this point, we wanted to take advantage of the two strong points of MIL, namely predicate invention and learning of recursion. As it turns out, in our design of rules, there aren't good reasons for recursion. We could think of some scenarios where recursion could be involved, but as it turns out, because of just a few number of tests suffice, there is no real need for it. Nevertheless, the last chapter of the thesis presents one scenario in which PoS disambiguators could benefit from learning of recursion.

Now let's turn and investigate predicate invention within the MIL framework. As we left recursion outside of this discussion, predicate invention occurs whenever it is not possible for the learner to produce a hypothesis without any new predicates, while obeying the metarules. With no recursion, it is always possible to produce a single Prolog clause that fits all positive examples and does not fit any negative examples, because we can always replace the call to a predicate with its body. Because of the way Metagol works at the moment, the more metarules we need and the more invented predicates we need to learn, the more time it takes. Initially, we tried to use the metarules that would have produced the following types of predicates (dyadics with at most two clauses in the body / monadics for tests):

```
disambiguate(C1, C2) :- test(C1), remove_part(C1, C2)
disambiguate(C1, C2) :- test(C1), select_part(C1, C2)
disambiguate(C1, C2) :- test(C1), disambiguate1(C1, C2)
test(C)              :- test1(C), test2(C).
```

This way, we could link multiple tests and end with one of the two predicates at the top, applying one of the possible actions. The problem is that in this case a rule consisting of four tests and one action would look like this:

```
disambiguate(C1, C3) :- p1(C1), remove_part(C2, C3).
p1(C) :- p2(C), p3(C).
p2(C) :- some_test1(C), some_test2(C).
p3(C) :- some_test3(C), some_test4(C).
```

Notice the invented predicates p1, p2, p3: the only case in which they are truly useful is whenever MIL can reuse them. In our experiments we found out that invented predicates are rarely used in multiple rules. Moreover, due to the performance of Metagol and the increased size of the resulting hypotheses, they become infeasible. We then decided to create three metarules that would result in clauses with 1, 2 and 3 tests (one extra test will always be added by us at a later point, at filtering and example generation time).

Unlike the previous disambiguator, we will use here contexts of radius 2:

$$[T_a, T_b, T_c, T_d, T_e] \tag{4.9}$$

We can see in 4.9 that each of the tokens has assigned a different letter: this letter is paired with context extraction predicates that are used in the background knowledge as helper predicates. The predicate `token_b(Context, T)` will bind `T` to the second token in the `Context`. In the Constraint Grammar introductory paper [8] it is suggested that a window of 5 tokens should be sufficient to produce good disambiguation.

Another addition to this disambiguator consists of rule priorities. Each one of the rule predicates is linked to a certain priority through a Prolog fact. One example is `disambiguator_rule(rule_491, 2).`: the fact states that `rule_491` has a priority of `2`. Note that in our case the lower the number, the higher the priority – in other words, the disambiguator will exhibit a preference for rules that have assigned lower numbers. From this point on we will refer to the *minimum priority rule* as the rule with the lowest number assigned.

The workflow of the disambiguator for a single context is presented in figure 4.2.

The step for finding candidate rules imposes three restrictions on the rules:

- $R(C, C')$ – the tests in the rule must unify with C and produce a (possibly) disambiguated context, C'.
- $C' \subset C$ – the readings of the central token in C' need to be a subset of those in C. That is, the rule must produce some disambiguation (remove at least one reading).
- C' has at least one reading – there must be at least one reading in the central token of the disambiguated context. Indeed, the Constraint Grammar framework uses this constraint: rules cannot remove the last reading of a token.

Using this workflow, the disambiguator iterates over all tokens, extracts their contexts and produces disambiguation. The process is over when no further disambiguation can be done using the rules – in this case, the disambiguator simply chooses one random reading for the still ambiguous tokens and writes the stream.

Metarules definition

There are two important matters to be discussed in this subsection: rule types and post conditions.

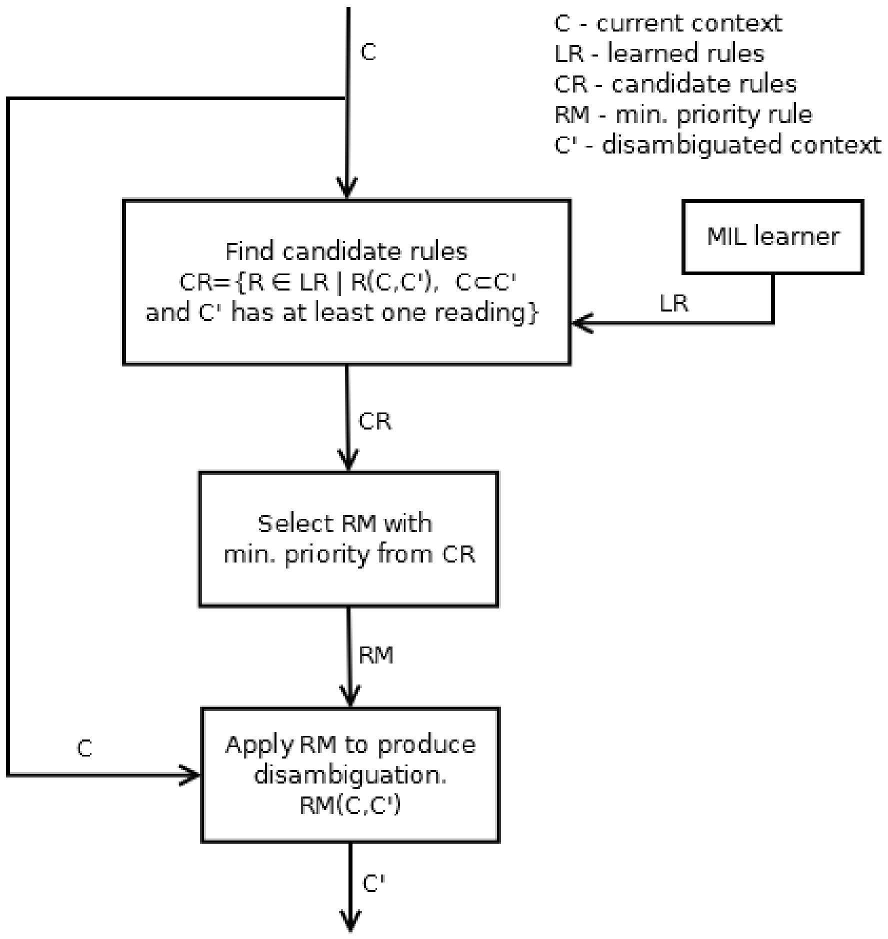

Figure 4.2: Per context disambiguation.

We argued that in our case, using predicate invention does not make much sense taking into account the performance of the learning system, the size of the theory and the fact that these invented predicates will probably not be reused in multiple clauses. This affected the way we designed our rules. Specifically, we know that we always want only one action per rule and at most 3 context tests, so we decided to write three metarules that will produce predicates with 1, 2 or 3 context tests and not to make use of predicate invention (since we would end up with the same programs, but expressed in a different way). Let's take a look at the metarule that could lead to the abduction of a clause with two tests (`T1` and `T2`) and one action (`A`):

Listing 4.2: Metarule 2 example

```
metarule1(ruletype2,
  [P/2, T1/1, T2/1, A/2],
  (
    [P,CtxAmb,CtxHt] :-
      [
        [T1,CtxAmb]-true,
        [T2,CtxAmb]-true,
        [A,CtxAmb,Result]-learned_rule_sanity_check(CtxAmb, CtxHt, Result)
      ]
  ),
        (
    pred_above(P/2, T1/1, Prog),
    pred_above(P/2, T2/1, Prog),
          pred_above(P/2, A/2, Prog),
    T1 \=@= T2
  ),
  Prog).
```

First, notice the parameters of `P`: `CtxAmb` – the ambiguous context, `CtxHt` – the handtagged context. Sure, we are in the learning part and, since we want to learn a function P, we need to provide pairs of input-output as examples. Now, we don't want to force the learner to remove or select readings so that it produces full disambiguation – it is better to be careful when removing or selecting readings and we don't have to do it at once, with only one rule. But in this case, we need some other way to check whether the rule is good or not.

First of all, note that the action `A` in the metarule does not need to unify with the handtagged context in its second parameter and its result is bound to the variable `Result`. Indeed, unification would not work in those partial disambiguation cases. The hypothesis that MIL will learn will look like `p(A, B) :- test1(A), test2(A), action(C).` and we will have to process them and replace `C` with `B`.

Checking if a rule works on a certain example (`CtxAmb / CtxHt` pair) at learning time is done through the `learned_rule_sanity_check` predicate that establishes the following post constraint between `CtxAmb`, `CtxHt` and `Result`:

$$C_{handtagged} \subset C_{result} \subseteq C_{ambiguous} \tag{4.10}$$

where $C_1 \subset C_2$ means that the readings of the central token in C_1 are a subset of the readings of the central token in C_2. The rationale behind this constraint is that a good rule needs to produce some disambiguation ($C_{result} \subset C_{ambiguous}$) and also needs to still keep the only reading in the handtagged version of the context ($C_{result} \subseteq C_{ambiguous}$). Note the *subset or equal* in the second relation: the rule is not restricted from producing full disambiguation.

Background knowledge

We need to provide Metagol with building blocks for the hypotheses that we want it to be able to learn.

First, let's focus on tests. We introduce two types of tests:

- `one_part_pos_tag`
  At least one of the readings of the token in position `pos` has the tag `tag`. Example:

```
one_part_e_vaux(Ctx) :- token_e(Ctx, T),
                        token_has_tag(T, "vaux").
```

- `all_part_pos_tag`
  All of the readings of the token in position `pos` have the tag `tag`. Example:

```
all_part_e_vaux(Ctx) :- token_e(Ctx, T),
                        token_all_tag(T, "vaux").
```

We have generated all the possible combinations of `one` / `all`, positions and tags and included them in the `monadics` set of predicates for Metagol.

On the other hand, the `dyadics` set of predicates contains all the possible actions that can be carried on a context. The general form of such a predicate is

- `cg_select_tag`
  Transforms the central token of the input context by removing all readings that do not contain the tag `tag`. Example:

```
cg_select_n(CtxIn, CtxOut) :- cg_select(CtxIn, CtxOut, "n").
```

- `cg_remove_tag`
  Transforms the central token of the input context by removing all readings that contain the tag `tag`. Example:

```
cg_remove_n(CtxIn, CtxOut) :- cg_remove(CtxIn, CtxOut, "n").
```

Predicates that are present in the definitions of the above (such as `cg_remove` or `token_all_tag`) are not part of the background knowledge – they are helper predicates implemented by hand.

Manual example

In listing 4.3 we provide a manual example to the MIL learner. Note the context of radius 2 (5 tokens) and the pair of ambiguous / handtagged instances. We did not provide any negative examples and, in the case of this second disambiguator, we never will (details of why we chose to do so are explained in the next subsection). The last line in the code listing tells Metagol which metarules should use in order to build the hypothesis (in this specific case, `ruletype1`, which means one test followed by a select / remove action).

Listing 4.3: Manual example for disambiguator 2

```
episode(
  % Concept name.
  rule_x,
  % One positive example.
  [
    [rule_x,
      [
        ["Yesterday", ["Yesterday<adv>", ["adv"]]],
        ["I", ["prpers<prn><p1><sg>", ["prn", "p1", "sg"]]],
        ["read",
            ["read<vblex><inf>", ["vblex", "inf"]],
            ["read<vblex><pres>", ["vblex", "pres"]],
            ["read<vblex><past>", ["vblex", "past"]],
            ["read<vblex><pp>", ["vblex", "pp"]]],
        ["a", ["a<det><ind><sg>", ["det", "ind", "sg"]]],
        ["book", ["book<n><sg>", ["n", "sg"]]]
      ],
      [
        ["Yesterday", ["Yesterday<adv>", ["adv"]]],
        ["I", ["prpers<prn><p1><sg>", ["prn", "p1", "sg"]]],
        ["read", ["read<vblex><past>", ["vblex", "past"]]],
        ["a", ["a<det><ind><sg>", ["det", "ind", "sg"]]],
        ["book", ["book<n><sg>", ["n", "sg"]]]
      ]
    ]
  ],
  % No negative examples.
  []
).

metaruless([[ruletype1]]).
```

The output of the learner can be seen below:

```
EXAMPLE EPISODE: rule_x

TRY CLAUSE BOUND: 1
TRY NEW PREDICATE BOUND: 0
TRY METARULE SET: [ruletype1]
FINAL HYPOTHESIS FOR EPISODE: rule_x, BOUND: 1

rule_x(A,B) :- all_part_a_adv(A), cg_remove_inf(A,C).
```

This is not a particularly good rule (*If all readings from token in position -2 contain "adverb" then remove "infinitive" from the central token*), but if fits the example and does not break our consistency rules: by removing the infinitive of *read* it reduces the ambiguity of the central token without removing the correct reading.

Generating examples. Three stage learning

In order to generate examples for the learner we first produce a set of *(ambiguous, handtagged)* tuples from our training set. Samples from this set will be our examples. Remember that in the case of the first disambiguator we found out that if we feed too distant examples Metagol will probably not be able to produce a hypothesis for us. One way of ensuring we do not get too distant examples is to filter them by ambiguity class or tags of the central token.

We chose to filter contexts by the presence of a certain tag in the central token because then we can use this information after learning and augment the rule with it.

This is why, unlike learning rules for the first disambiguator, we need to process the rules we get from Metagol in two ways. First, we are going to replace the variable `C` with `B` (it is different only so we are able, at learning time, to check our constraints). Then we are going to add another test to the clause. This test checks if the central token has the tag we were targeting when we did the filtering of examples. The pseudocode 4.1 illustrates the complete process.

Algorithm 4.1 Rule learning algorithm.

```
1: $A$ = ambiguous contexts, $H$ = corresponding handtagged contexts.
2: $Tags$ = all tags present in $A \cup H$.
3: $C_p \leftarrow \{(c_a, c_h) | c_a \in A, c_h \in H\}$, all context pairs.
4: for $tag \in Tags$ do
5:     $C_{tag} = \{(c_a, c_h) \in C_p |$ central token of $c_a$ has tag $tag\}$
6:     for $i = 1$ to $N_{examples}$ do
7:         Sample $C_{tag}$, build the episode file for Metagol and get a rule $R$.
8:         Fix rule $R$: replace C with B in the body.
9:         Augment rule $R$: prepend one_part_c_tag(A).
10:        Save rule.
11:    end for
12: end for
```

We implemented three metarules that Metagol can use to produce predicates with 1, 2 or 3 tests. We do a three stage learning and in each of the stages we use one of the metarules. In the first stage we use a large number of examples (about 12) each time we call Metagol and we ask for only one test – the idea behind this is that we want to get good generalization with our rules (less constraints means more generalization because it increases the chance that the rule will fit many examples). After we have learned a set of predicates we use them to produce disambiguation and save the partially disambiguated file. This file will be used as the "ambiguously tagged" input for the second stage of learning.

In the second stage of learning we learn predicates with 2 tests (using the second metarule), but supply less examples (about 6). This is partly because now we have less contexts to sample from (they come from the result of disambiguation using rules learned in the first stage) and partly because it is a lower chance that a predicate with two constraints will fit many examples. We gather all the rules that we are able to learn this way, use them to disambiguate the result of stage 1 and we get an even less ambiguous result. Again, the result of this stage will make for the input to our third (and final) stage.

The third stage of learning makes use of the third metarule. We use as input the result of stage 2, we feed even less examples (2-3) to Metagol and ask for rules with three tests.

At this point we have three sets of rules, one set for each of the learning stages. We are going to set priorities to them so that rules with less tests are applied first, exactly in the order of learning. Although we are also going to use a genetic algorithm to assign different priorities to rules with the same number of tests, all the rules from stage 1 will be preferred to those from stages 2 and 3, while all the rules from stage 2 will be preferred to those from stage 3, so the genetic algorithm will only optimize one stage at a time.

Rule selection

In the learning stage we were able to learn about 1000 rules. We know that some of them have a low quality, while others are good and the goal of the *rule selection* step is to select a subset of good rules that can achieve the highest accuracy. In the case of this disambiguator we replaced the naive pruning of rules (which was implemented for the proof-of-concept disambiguator) with a genetic algorithm that assigns priorities.

At learning time, all the rules that come from Metagol in one stage have the same priority. In Chapter 5 we show and discuss the implementation of a genetic algorithm that assigns a different priority to each rule from a certain stage.

After each rule has a priority assigned it is just a matter of removing all rules with low priorities which would be, anyway, probably covered by higher priority rules. So in this case the pruning is not done by accuracy gain (as in the case of the first disambiguator), but based on a priority that is optimized by a genetic algorithm. Figure 4.3 illustrates the evolution of the accuracy by the number of rules. As we can see from the graph, the best size of the rule set is about 400, which is twice as much as linguists usually write for a CG-3 parser (although our rules have less expressive power).

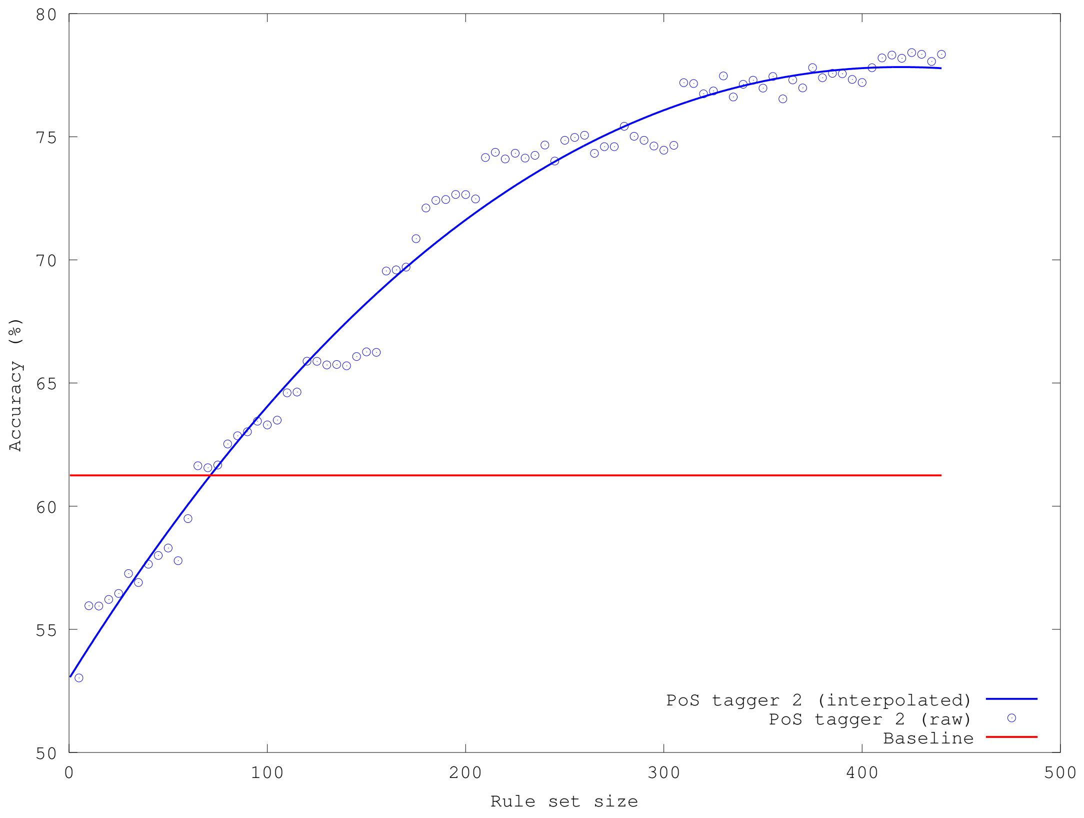

Figure 4.3: Accuracy comparison of PoS disambiguator 2

Results

Table 4.5 presents the accuracy comparison between our probabilistic baseline implementation and the second PoS disambiguator.

	PoS tagger 2	Baseline
$A_{disambiguation}$	78.35%	60.60%
$A_{overall}$	93.97%	89.05%

Table 4.5: Accuracy comparison for PoS tagger 2.

As we can see, we have an 18% improvement over the baseline (and nearly 8% from the proof-of-concept implementation). The overall accuracy is almost 94% and this is in line with other rule learning based implementations, such as [10]. While [11] reports a higher accuracy, it does not fully disambiguate the stream and the final set of rules consists of about 7000 rules (we managed to select a much more compact set).

Note that, unlike the case of the disambiguator presented in [10], we did not implement the grammar of the English language in the background knowledge, but merely provided the MIL learner with simple tests and actions.

Chapter 5

Selecting a good set of rules

Genetic Algorithms have been around for decades now and are used in many optimization problems. A genetic algorithm employs a heuristic search in the solution space of optimization problems by mimicking the process of natural selection [21].

Genetic algorithms are used on a large scale both for theoretical problems (such as TSP [1] [27], shortest path in graphs [28] or the Knapsack problem [29]) and real world problems (traffic routing [30], helicopter flight [31] or Quality-of-Service management in Internet routing [32]).

A possible workflow of a genetic algorithm is illustrated in pseudocode 5.1.

Algorithm 5.1 Genetic algorithm workflow.

```
1: Input: $I$ - iteration count.
2: Input: $N_1$ - population size.
3: Input: $N_2$ - offspring count.
4: Generate an initial population $P$ of size $N_1$.
5: for $i = 1$ to $I$ do
6:     $O = \emptyset$
7:     for $j = 1$ to $N_2$ do
8:         $(C_1, C_2) \leftarrow$ select_candidates($P$)
9:         $O \leftarrow O \cup$ crossover($C_1, C_2$)
10:    end for
11:    $P' \leftarrow P \cup O$
12:    $P \leftarrow$ select best $N_1$ individuals from $P'$.
13: end for
14: Output: best individual in $P$.
```

As a summary, the main idea is to start with an initial population (if we have no information about a good solution, then we will rely on randomly generated individuals). Each generation we will produce a number of offspring which will be included in the population. At the end of each generation we have the "survival of the fittest" step,

[1]Traveling Salesman Problem

which selects only the best individuals from our current population. The key aspects in any genetic algorithm implementation are the fitness function, crossover function and an algorithm which selects candidates for breeding. In our pseudocode, the genetic recombination happens in line 9, while the candidates C_1 and C_2 that will produce the new individual are selected in line 8.

As a variation, instead of a fixed iteration count we can check how much improvement we have over each generation and decide to stop the algorithm when no further improvement is detected. Also, moving closer to the natural processes we can include, with small probability, random mutations in offspring – this will bring a little bit of diversity but we need to be careful as it could be detrimental to the convergence of the process (sometimes the probability of random mutations is decreased over time, as we want to move from exploration to exploitation).

We use a genetic algorithm to optimize the priority assignment for the rules that we learned in the case of the second disambiguator.

First, we need to discuss the genetic encoding of our problem. Because we want to optimize priorities for rules and we also want them to be different we can see an individual as a permutation of priorities. Let's take the following example (the link between a rule predicate and its priority is given by the `disambiguator_rule` Prolog fact):

```
disambiguator_rule(rule_1, 1).
disambiguator_rule(rule_2, 2).
disambiguator_rule(rule_3, 3).
disambiguator_rule(rule_4, 4).
disambiguator_rule(rule_5, 5).
```

In this arrangement each rule has a different priority and there are 5 rules in total. If we have a fixed ordering of these predicates (`rule_1`, `rule_2`, etc.) we can represent priorities in a simple list of 5 elements where the number in position i is the priority of the i-th rule. The individual in our example will be encoded as `[1, 2, 3, 4, 5]`. Now, finding the best priority assignment is reduced to finding the best permutation of numbers from 1 to 5. For instance, the permutation `[5, 2, 3, 1, 4]` will produce the following individual:

```
disambiguator_rule(rule_1, 5).
disambiguator_rule(rule_2, 2).
disambiguator_rule(rule_3, 3).
disambiguator_rule(rule_4, 1).
disambiguator_rule(rule_5, 4).
```

Each one of the numbers in the permutation represents one *gene* and genes in the same positions from two different individuals are called *alleles*. Notice that we are dealing with a fixed length genetic code. Sometimes in practice we might need to work with variable length code.

The next important aspect is the *fitness function*. In other words, we need a measure to evaluate a certain individual. In our case this is done by writing the `disambiguator_rule` facts to a Prolog file (this way establishing priorities for rules), run our PoS tagger on the training data and then evaluate its accuracy. That is, our fitness function will output a floating point value between 0 and 100, depending on the ratio of correctly disambiguated tokens.

The selection of candidates for breeding can be done in multiple ways, the most commonly used being *roulette wheel selection*. This technique picks each individual with a probability of

$$p_i = \frac{w_i}{\sum_{i=1}^{N} w_i} \tag{5.1}$$

where the weight w_i is the fitness value of the i-th individual and N is the population size.

Roulette wheel selection did not behave well in our case due to the fact that the values of the fitness function were too close to each other. In turn, we used the *rank based selection*, which replaces the weights w_i with r_i, where r_i is the position of the individual in the population sorted in descending order by fitness.

The last detail of our genetic algorithm implementation is the *genetic recombination* function (also called *crossover*). Because we are working on permutations, it is compulsory that the offspring will always be a permutation. Out of the many techniques that are consistent with this constraint we implemented *order 1 crossover*, detailed in the rest of this chapter.

We will take the following two individuals and do a step by step demonstration of the *order 1 crossover*:

Parent 1	1	8	3	4	7	6	5	2
Parent 2	4	2	3	7	8	1	6	5
Offspring								

First, we select a contiguous chain of alleles from the first parent and we drop them to the offspring:

Parent 1	1	8	3	4	7	6	5	2
Parent 2	4	2	3	7	8	1	6	5
Offspring				4	7	6		

Because we don't want duplicates in our offspring, we cross out all the genes from this chain in the second parent:

Parent 1	1	8	3	4	7	6	5	2
Parent 2	~~4~~	2	3	~~7~~	8	1	~~6~~	5
Offspring				4	7	6		

We proceed by selecting all remaining genes in the second parent:

Parent 1	1	8	3	4	7	6	5	2
Parent 2	~~4~~	2	3	~~7~~	8	1	~~6~~	5
Offspring				4	7	6		

And complete the gaps in the offspring with them, preserving their original order:

Parent 1	1	8	3	4	7	6	5	2
Parent 2	~~4~~	2	3	~~7~~	8	1	~~6~~	5
Offspring	2	3	8	4	7	6	1	5

The genetic algorithm produced in the end a very good set of rules. Unfortunately, due to the size of the rule set, the fitness function was slow to evaluate (around 0.2 seconds). We ran multiple experiments with this genetic algorithm and the longest one took about 10 hours to complete.

Chapter 6

Results

In this chapter we are going to discuss the results of our work.

Most importantly, we achieved the goal of our project: we managed to make use of the newest ILP framework (Meta Interpretive Learning) and showed that it is feasible to use it to machine learn rules for a Part-of-Speech disambiguator. Another achievement of this project is that all of our code has been uploaded to the Apertium repository. This means that there is an active interest in machine learning of rules for PoS tagging and it is a starting point for exploring other possibilities. Because the project was designed to be more exploratory we encountered some issues in our experiments and provided solutions and workarounds that led to state of the art level disambiguation accuracy. Nevertheless, we have learned some important lessons, summarized below:

- **Rule pruning** – Looking at CG-3 files produced by linguists we can see that the average rule set size is about a few hundreds. While some researchers produced huge sets of rules [10], we managed to keep our set robust (approx. 420 rules) with almost the same level of accuracy. Rule pruning also prevents overfitting and produces a better generalization on unseen examples.

- **Selection** – The idea of assigning priorities and optimize them with a genetic algorithm proved to be good judging by the increase in accuracy. At the stage of priority assignment there is a tradeoff between execution time and getting a better set of rules. Rule prioritization also makes it easier to prune the rules with low priority and produce a compact final set.

- **Execution time** – The somewhat low performance of the Metagol system forced us to look for workarounds when it came to large theories, large number of examples, predicate invention and avoidance of overfitting. After carefully analyzing every one of these issues we came up with ideas that provided the needed balance in order to lower the execution time.

- **Learning by tag classes** – Example selection is directly related to execution time: the more distant the examples we provide to Metagol are, the less likely the system is to explain them with a short theory (1-2 rules). We found out that dividing the examples into tag classes is enough to ensure a smaller distance between examples (as measured in the number of rules needed to *explain* them). Furthermore, we can use this information after the learning process and link each rule to its tag class. This resulted in the most important accuracy gain in the case of the second disambiguator.

- **Predicate invention** – While predicate invention proves to be useful in other problems, it has only been tested on small, toy examples. We tried to let Metagol introduce new predicates in order to find rules but we arrived at the conclusion that is better to design our rules from the beginning in a way such that new predicates will not be needed (we made this decision mostly due to performance issues). Nevertheless, as showed in the BFS example, it is possible to use it and may become available to real world problems with time (as researchers will find ways to improve the speed of the MIL systems).
- **Overfitting** – We managed to avoid some overfitting at learning time by forcing the Metagol system to output single-test rules, while supplying large amounts of examples. In the case of our second PoS tagger we moved forward with this idea and employed the *three stage learning* technique. We also dealt with this problem at later stages, namely selection (priority assignments) and pruning.
- **Three stage learning** – The idea that we should learn rules in three distinct stages and then set different levels of priority based on the stage in which each rule has been produced yielded good results, most importantly improvements in terms of accuracy. The goal of this technique was to find out where more general rules cannot be applied and patch these with more specific rules.
- **Negative examples** – Lastly, we found out that due to the inductive bias of Metagol (induced by metarules and background knowledge) we do not always need negative examples. Because our tests were specific enough, overgeneralization was less likely. Indeed, in the case of the second PoS tagger we did not supply any negative examples to the learner.

From the accuracy point of view, table 6.1 presents a summary of the results we were able to achieve. It also includes results from [10, 11] and the current Hidden Markov Models tagger implemented in Apertium [6]. Note that accuracy reports were taken from the papers and were trained and evaluated on different datasets.

Apart from [11] that did not produce full disambiguation of the stream (thus leading to a very high accuracy in the sense that the tokens preserved the correct reading), we can see that our second disambiguator competes with other state of the art PoS taggers. The HMM tagger is still performing better, but it uses a statistical approach (as opposed to a rule learning based one). It is notable that our disambiguator produced less rules than [11], while having a comparable accuracy to [10] even though it did not have any access to knowledge about the English language grammar.

Method	$A_{overall}$	$A_{disambiguation}$	Notes
Baseline	61.25%	89.21%	Simple probabilistic model.
PoS D1	71.23%	91.98%	Concept identification.
PoS D2	78.35%	93.97%	Dyadic predicates, CG-style actions.
HMM	~80-85%	~95-96%	Statistical approach.
Progol Gram.	–	94%	Grammar in background knowledge.
Progol CG	–	97%	Flawed accuracy formula.

Table 6.1: Method comparison.

Figure 6.1 illustrates a comparison between our two disambiguators from the point of view of rule set size influence on accuracy. It is interesting that the *concept identification* tagger achieves good accuracies faster. It is nevertheless surpassed by the second implementation that used the three stage learning technique to patch more subtle situations in the grammar (note that the best rule set size was 120 and 440 for the first and second disambiguator, respectively).

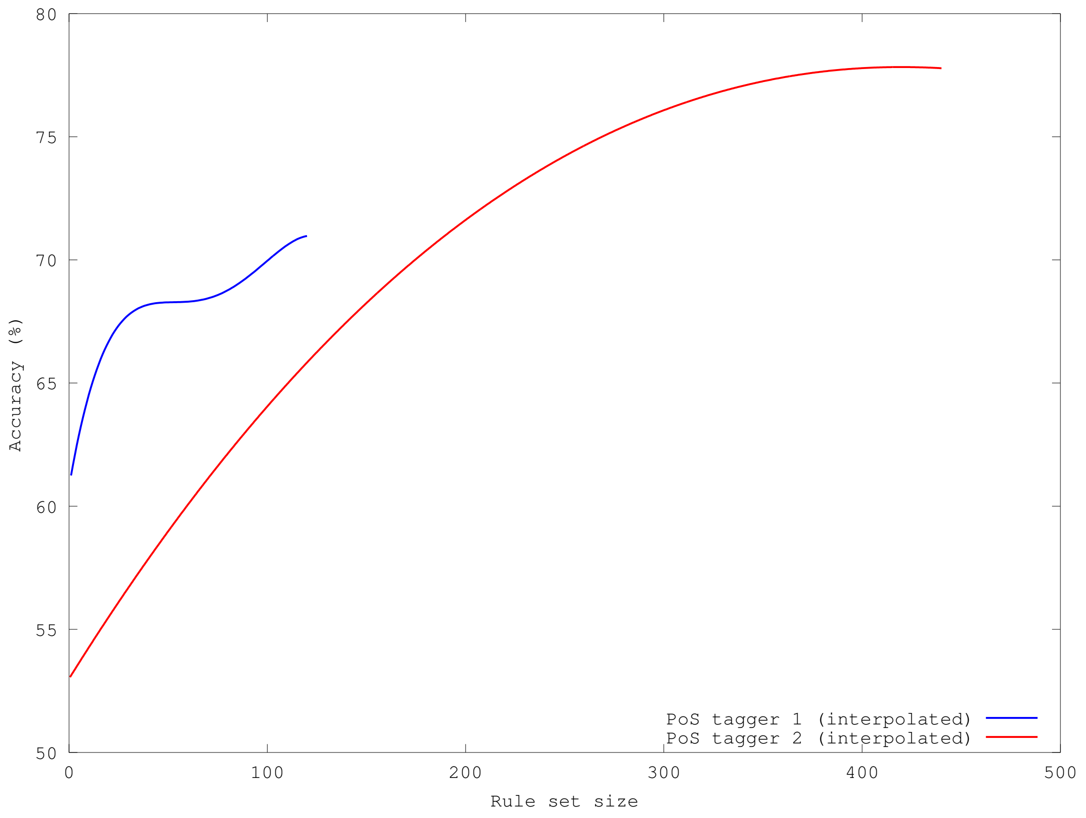

Figure 6.1: Accuracy evolution by rule set size.

Chapter 7

Conclusion and future work

As a conclusion, we achieved the main goals of the project. This work serves as a Proof-of-Concept for using the MIL framework for machine learning of rules to use with a PoS tagger. We provided workarounds for performance issues and proved that we can produce good disambiguation. The limitations of Metagol that we encountered may disappear in the future and, because it is able to learn complex Prolog programs as its hypothesis, it is a perfect fit especially for rule learning tasks.

Leaving aside performance limitations, it is theoretically possible to learn a full disambiguator using the Meta Interpretive Learning framework. This was also our initial target, but we moved to rule learning because the Metagol system is still unable to cope with large amounts of data or long hypotheses. Because it is possible to express our disambiguator in the H_2^2 space (which makes it definitely learnable), a challenge for the future would be to learn the complete disambiguator from examples (not only rules, but an actual Prolog program that would map the stream of ambiguous tokens to the stream of correct, non-ambiguous tokens).

Another approach would be to learn the grammar of a language as a set of predicates. This set of predicates could be later used to produce a very compact set of rules, as described in [10].

A more interesting approach that would have less performance requirements could be analyzing the text at sentence level and learning multiple per sentence grammars. At disambiguation time we could try to fit one of these grammars on the sentence we are analyzing and decide on tags (maybe also in combination with some rules).

Research on MIL and improvements could simplify the process of learning – with less overfitting we don't need pruning, selection or three stage learning and we could theoretically learn the complete set of rules that covers all of our examples in one step.

At the moment, selection works by assigning different priorities to different predicates. Note that the genetic algorithm does not change the rules in any way. Another idea would be to use genetic programming in order to get better rules from the ones already learned by the Metagol system. This would defeat, to some extent, the purpose of learning with Metagol, but it could lead to a more robust final set.

Appendices

Appendix A

Auto-generated examples

Listing A.1: Auto-generated example for disambiguator 1

```
episode(
  % Concept name.
  concept_x,
  % Positive examples.
  [[concept_x,
     [["elliptical", ["elliptical<adj>", ["adj"]]],
      ["orbits", ["orbit<n><pl>", ["n", "pl"]]],
      ["to", ["to<pr>", ["pr"]]]]],
   [concept_x,
     [["cantonal", ["cantonal<adj>", ["adj"]]],
      ["structures", ["structure<n><pl>", ["n", "pl"]]],
      ["remained", ["remain<vblex><past>", ["vblex", "past"]]]]],
   [concept_x,
     [["Middle", ["Middle<adj>", ["adj"]]],
      ["Ages", ["Age<n><pl>", ["n", "pl"]]],
      ["in", ["in<pr>", ["pr"]]]]]
  ],
  % Negative examples.
  [[concept_x,
     [["if", ["if<cnjadv>", ["cnjadv"]]],
      ["unmarried", ["*unmarried", []]],
      [",", [",<cm>", ["cm"]]]]]
  ]
).
```

Listing A.2: Auto-generated example for disambiguator 2

```
episode(rule_x, % Rule name.
[ % Positive examples.
[rule_x,
  [
    ["minimum",
      ["minimum<adj>", ["adj"]],
      ["minimum<n><sg>", ["n", "sg"]]],
    ["guaranteed",
      ["*guaranteed", []]],
    ["under",
      ["under<adv>", ["adv"]],
      ["under<pr>", ["pr"]]],
    ["the",
      ["the<det><def><sp>", ["det", "def", "sp"]]],
    ["constitution",
      ["constitution<n><sg>", ["n", "sg"]]]],
    [
      ["minimum", ["minimum<n><sg>", ["n", "sg"]]],
      ["guaranteed", ["*guaranteed", []]],
      ["under", ["under<pr>", ["pr"]]],
      ["the", ["the<det><def><sp>", ["det", "def", "sp"]]],
      ["constitution", ["constitution<n><sg>", ["n", "sg"]]]
    ]
  ],
[rule_x,
  [
    ["educational",
      ["educational<adj>", ["adj"]]],
    ["pillars",
      ["*pillars", []]],
    ["in",
      ["in<adv>", ["adv"]],
      ["in<pr>", ["pr"]]],
    ["Iran",
      ["Iran<np><top><sg>", ["np", "top", "sg"]],
      ["Iran<np><top><pl>", ["np", "top", "pl"]]],
    ["and",
      ["and<cnjcoo>", ["cnjcoo"]]]],
    [
      ["educational", ["educational<adj>", ["adj"]]],
      ["pillars", ["*pillars", []]],
      ["in", ["in<pr>", ["pr"]]],
      ["Iran", ["Iran<np><top><sg>", ["np", "top", "sg"]]],
      ["and", ["and<cnjcoo>", ["cnjcoo"]]]]]
],
% No negative examples.
[]
).

% Define the sets of metarules to use.
metaruless([[ruletype3]]).
```

Bibliography

[1] Helmut Schmid. Probabilistic part-of-speech tagging using decision trees. In *Proceedings of the international conference on new methods in language processing*, volume 12, pages 44–49. Manchester, UK, 1994.

[2] Adwait Ratnaparkhi et al. A maximum entropy model for part-of-speech tagging. In *Proceedings of the conference on empirical methods in natural language processing*, volume 1, pages 133–142. Philadelphia, PA, 1996.

[3] John C Henderson and Eric Brill. Exploiting diversity for natural language processing. In *AAAI/IAAI*, page 1174, 1998.

[4] Jesús Giménez and Lluis Marquez. Svmtool: A general pos tagger generator based on support vector machines. In *In Proceedings of the 4th International Conference on Language Resources and Evaluation*. Citeseer, 2004.

[5] Helmut Schmid and Florian Laws. Estimation of conditional probabilities with decision trees and an application to fine-grained pos tagging. In *Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1*, pages 777–784. Association for Computational Linguistics, 2008.

[6] Zaid Md Abdul Wahab Sheikh, Felipe Sánchez Martínez, et al. A trigram part-of-speech tagger for the apertium free/open-source machine translation platform. 2009.

[7] Eric Brill. A simple rule-based part of speech tagger. In *Proceedings of the Workshop on Speech and Natural Language*, HLT '91, pages 112–116, Stroudsburg, PA, USA, 1992. Association for Computational Linguistics.

[8] Fred Karlsson. Constraint grammar as a framework for parsing running text. In *Proceedings of the 13th conference on Computational linguistics-Volume 3*, pages 168–173. Association for Computational Linguistics, 1990.

[9] Pasi Tapanainen. *The constraint grammar parser CG-2*. University. Department of General Linguistics, 1996.

[10] James Cussens. Part-of-speech tagging using progol. In *Inductive Logic Programming*, pages 93–108. Springer, 1997.

[11] Nikolaj Lindberg and Martin Eineborg. Learning constraint grammar-style disambiguation rules using inductive logic programming. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics-Volume 2*, pages 775–779. Association for Computational Linguistics, 1998.

[12] Adam Lopez. Statistical machine translation. *ACM Computing Surveys (CSUR)*, 40(3):8, 2008.

[13] MikelL. Forcada, Mireia Ginestí-Rosell, Jacob Nordfalk, Jim O'Regan, Sergio Ortiz-Rojas, JuanAntonio Pérez-Ortiz, Felipe Sánchez-Martínez, Gema Ramírez-Sánchez, and FrancisM. Tyers. Apertium: a free/open-source platform for rule-based machine translation. *Machine Translation*, 25(2):127–144, 2011.

[14] Raül Canals-Marote, Anna Esteve-Guillén, Alicia Garrido-Alenda, M Guardiola-Savall, Amaia Iturraspe-Bellver, Sandra Montserrat-Buendia, Sergio Ortiz-Rojas, Hermınia Pastor-Pina, Pedro M Pérez-Antón, and Mikel L Forcada. The spanish-catalan machine translation system internostrum. In *Proceedings of MT Summit VIII: Machine Translation in the Information Age*, volume 73, page 76, 2001.

[15] Carme Armentano-Oller, Rafael C Carrasco, Antonio M Corbí-Bellot, Mikel L Forcada, Mireia Ginestí-Rosell, Sergio Ortiz-Rojas, Juan Antonio Pérez-Ortiz, Gema Ramírez-Sánchez, Felipe Sánchez-Martínez, and Miriam A Scalco. Open-source portuguese–spanish machine translation. In *Computational Processing of the Portuguese Language*, pages 50–59. Springer, 2006.

[16] Luis Villarejo, Mireia Farrús, Sergio Ortiz, and G Ramírez. A web-based translation service at the uoc based on apertium. In *Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on*, pages 525–530. IEEE, 2010.

[17] Eckhard Bick. The parsing system palavras. *Automatic Grammatical Analysis of Portuguese in a Constraint Grammar Framework*, 2000.

[18] Mans Hulden and Jerid Francom. Boosting statistical tagger accuracy with simple rule-based grammars. In *LREC*, pages 2114–2117, 2012.

[19] Stephen Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[20] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19(20):629–679, 1994.

[21] Tom M Mitchell. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45, 1997.

[22] Ivan Bratko and Stephen Muggleton. Applications of inductive logic programming. *Commun. ACM*, 38(11):65–70, November 1995.

[23] Stephen Muggleton. Inverse entailment and progol. *New Generation Computing*, 13(3-4):245–286, 1995.

[24] Stephen H Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine Learning*, 94(1):25–49, 2014.

[25] Diana F Gordon and Marie Desjardins. Evaluation and selection of biases in machine learning. *Machine Learning*, 20(1-2):5–22, 1995.

[26] Stephen Muggleton and Dianhuan Lin. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 1551–1557. AAAI Press, 2013.

[27] John Grefenstette, Rajeev Gopal, Brian Rosmaita, and Dirk Van Gucht. Genetic algorithms for the traveling salesman problem. In *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pages 160–168. Lawrence Erlbaum, New Jersey (160-168), 1985.

[28] Mitsuo Gen, Runwei Cheng, and Qing Wang. Genetic algorithms for solving shortest path problems. In *Evolutionary Computation, 1997., IEEE International Conference on*, pages 401–406. IEEE, 1997.

[29] Paul C Chu and John E Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of heuristics*, 4(1):63–86, 1998.

[30] Barrie M Baker and MA Ayechew. A genetic algorithm for the vehicle routing problem. *Computers & Operations Research*, 30(5):787–800, 2003.

[31] Chad Phillips, Charles L Karr, and Greg Walker. Helicopter flight control with fuzzy logic and genetic algorithms. *Engineering Applications of Artificial Intelligence*, 9(2):175–184, 1996.

[32] Fei Xiang, Luo Junzhou, Wang Jieyi, and Gu Guanqun. Qos routing based on genetic algorithm. *Computer communications*, 22(15):1392–1399, 1999.