Verb Sense Disambiguation in STRING

Ricardo Filipe Mendes Pires

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Supervisors: Prof. Nuno João Neves Mamede
Prof. Jorge Manuel Evangelista Baptista

Examination Committee

Chairperson: Prof. Miguel Nuno Dias Alves Pupo Correia
Supervisor: Prof. Nuno João Neves Mamede
Member of the committee: Prof. Helena Gorete Silva Moniz

Junho 2017

Resumo

A desambiguação dos sentidos das palavras tem uma grande influência sobre o desempenho de várias tarefas de Processamento de Língua Natural, uma vez que o significado exacto de uma palavra ambígua num dado contexto pode influenciar a análise sintática da frase em que esta se insere, por exemplo nas relações de dependência entre os seus constituintes. Esta dissertação foca-se no problema da Desambiguação dos Sentidos dos Verbos, um subproblema da Desambiguação dos Sentidos das Palavras, no contexto do sistema STRING, uma cadeia de Processamento de Língua Natural desenvolvida para o Português.

Esta dissertação propõe uma nova abordagem à geração de regras de desambiguação para uso na STRING. Esta abordagem utiliza um conjunto de regras declarativas e o conhecimento presente no ViPEr, um Léxico-Gramática para o Português Europeu, para gerar regras de desambiguação. É ainda proposta uma implementação do algoritmo de aprendizagem automática *Transformation-based Learning*, como meio de optimizar a ordem das regras de desambiguação geradas.

Abstract

Word Sense Disambiguation has a great influence over the performance of several Natural Language Processing tasks, since the exact meaning of an ambiguous word in a given context may have an impact on the syntactic parsing of the sentence, for example in the parsing of the dependency relations among its constituents. This dissertation focuses on Verb Sense Disambiguation, a subproblem of Word Sense Disambiguation, in the context of STRING, a NLP chain developed for Portuguese.

This dissertation proposes a new approach for the generation of disambiguation rules to be used in STRING. This approach uses a set of declarative rules and the knowledge present in ViPEr, a Lexicon-Grammar for the European Portuguese verbs, to generate disambiguation rules. An implementation of the Transformation-based Learning machine-learning algorithm is also proposed, as a way to optimize the order of the generated disambiguation rules.

Palavras Chave
Keywords

Palavras Chave

Processamento de Língua Natural

Desambiguação de Sentido de Verbos

Desambiguação baseada em regras

Aprendizagem Automática

Keywords

Natural Language Processing

Verb Sense Disambiguation

Rule-based disambiguation

Machine Learning

Contents

List of Figures

List of Tables

Acronyms

NLP Natural Language Processing

WSD Word Sense Disambiguation

VSD Verb Sense Disambiguation

POS Part of Speech

XIP Xerox Incremental Parser

ViPEr Verb for European Portuguese

MFS Most Frequent Sense

ML Machine Learning

SR Semantic Roles

1 Introduction

1.1 Motivation

Nowadays, Natural Language Processing (NLP) is a field of research with a wide array of applications. However, it still faces several unsolved challenges, posed by the very nature of language, among which *ambiguity* is paramount. Ambiguity is present at several levels of analysis and in many forms. One of the most difficult aspects of this complex phenomena is word sense ambiguity, that is, the possibility of the same word being assigned different semantic values depending on its given context. This has a large impact on the results of NLP, since a mistake in disambiguation can change the meaning of the text being processed. This is a particularly relevant matter when it comes to verbs, which, for the most part express semantic predicates, since these are some of the most important components for determining the meaning of sentences and assigning them an adequate syntactic parse.

1.2 Goals

This project focuses on trying to improve the Verb Sense Disambiguation (VSD) mechanisms currently in place on the STatistical and Rule-based Natural lanGuage processing chain (STRING) (Mamede et al. 2012) system, using recent developments on ViPEr (Baptista 2012) - a database of the European Portuguese Verbs. For each verb, this database contains a set of semantic and syntactic features that describe its use.

Two approaches are used at the time of writing, a rule-based (Travanca 2013) and a machine-learning-based (Travanca 2013; Suíssas 2014) approach.

The rule-based approach is implemented through a module that uses the knowledge contained in ViPEr to automatically produce disambiguation rules, which are then applied within the STRING chain. However, this approach suffers from some shortcomings: it is not very flexible, which makes it hard to keep up with ViPEr's recent developments; and all the knowledge used in the rule generation process is embedded in the code of the module itself.

The machine learning approach uses *corpora* annotated according to the information provided by ViPEr and a machine learning algorithm to build models for each verb, which are then used to choose between a set of verb senses for that verb.

This project aims at refining verb sense disambiguation in STRING by improving upon the current rule-based approach shortcomings. In addition, this project will use the Transformation-Based Learning (TBL) machine-learning algorithm to help improve rule-based disambiguation.

1.3 Document Structure

This dissertation is structured as follows:

In Chapter 2, the resources upon which the project is based are presented, focusing first on the STRING chain, followed by ViPEr and, finally, the TBL algorithm.

Chapter 3 presents the approaches to VSD currently used in STRING.

Afterwards, in Chapter 4, the approaches implemented herein are presented, detailing several decisions taken during their development.

Throughout Chapter 5, the different evaluation scenarios used to test both approaches are presented, followed by a discussion on the results obtained in each experiment and the main conclusions that can be drawn from them.

Finally, in the last chapter (Chapter 6), an overview of the entire work is provided and the main conclusions are presented. The Chapter ends by pointing to future work, providing possible directions for expanding and improving the modules here developed.

2 Resources

This chapter describes the STRING NLP chain and the ViPEr database, two resources used by the Verb Sense Disambiguation approaches described in Chapter 3. STRING uses the results of the rule-based and the machine learning modules to perform disambiguation, while ViPEr contains the linguistic knowledge necessary to generate the rules and models produced. In this chapter, the TBL algorithm is also described in detail. TBL is a machine-learning algorithm based on learning through *corpus* transformation, that has been used to solve other NLP problems, such as POS tagging.

2.1 STRING

STRING is a hybrid, statistical and rule-based NLP chain for the Portuguese developed at the Spoken Language Systems Laboratory (L^2F). It has a modular structure, and performs basic text processing tasks, such as tokenization and text segmentation, part-of-speech tagging, morphosyntactic disambiguation, shallow parsing (chunking), and deep parsing (dependency extraction). It also performs additional NLP tasks, such as Named Entity Recognition, Anaphora Resolution and Time Normalization, among others.

2.1.1 Architecture

As previously indicated, STRING has a modular structure. It's comprised of 3 stages and an additional set of post-processing modules, as shown in Figure 2.1.

The first stage of the chain is responsible for performing segmentation, part-of-speech (POS) tagging, and sentence splitting. This is performed by the Lexical Morphological Analyser (LexMan) (Vicente 2013; Almeida 2016) module. The second stage is the disambiguation stage, which is comprised of two steps: rule-driven morphossyntactic disambiguation, performed by the Rule Driven Converter (RuDriCo2) (Diniz 2010; Diniz et al. 2010) module; and statistical disambiguation, performed by the Morphossyntactic Ambiguity Resolver (MARv4) (Ribeiro ; Rodrigues 2007) module. The final stage is the syntactic analysis stage (parsing). This stage is performed by the XIP parser (Aït-Mokhtar et al. 2002). Finally, a set of post-processing modules uses the output of STRING to perform specific

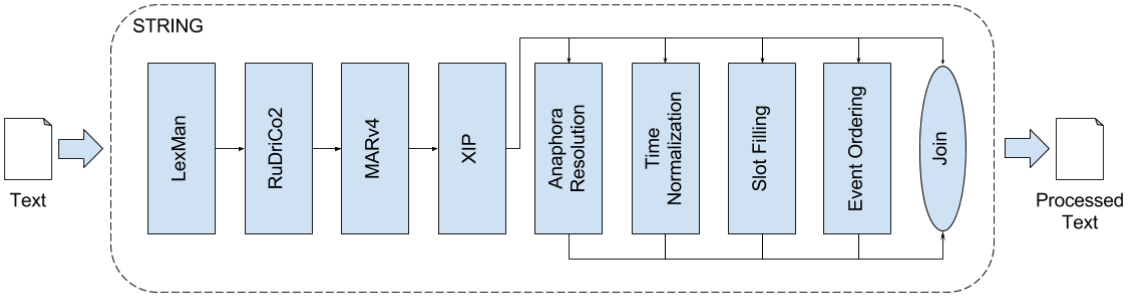

Figure 2.1: STRING architecture

NLP tasks. This is comprised of an anaphora resolution module, a slot filling module, a time expression normalization module, and an event ordering module.

The first three stages are always performed, whereas the fourth stage can be skipped if not necessary. In addition, not all of the modules of the fourth stage need to be used in every case. The communication between the different modules is done using eXtensible Markup Language (XML) files.

In the following lines, each module is described in further detail.

2.1.2 LexMan

LexMan (Vicente 2013; Almeida 2016) is a text segmenter and morphological analyser. It uses Finite State Transducers to segment both the text into sentences and these sentences into tokens (such as words, punctuation, numbers, abbreviations, e-mail addresses, etc.) Then, for each token, LexMan assigns its POS (out of twelve categories, such as noun, verb, adjective or pronoun), as well as other relevant linguistic features (tense, person, number, gender, etc.)

LexMan also functions as a tool for updating and managing the lexicons used by the system. From the verbs lemmas, it generates dictionaries of their corresponding inflected forms and builds the finite state transducers used in tokenization and POS-tagging. This module is also able to deal with productive word formation phenomena, like prefixation and suffixation (the latter being under development at the time of writing).

2.1.3 RuDriCo2

RuDriCo2 (Diniz 2010; Diniz et al. 2010) is the module responsible for refining the tokenization and for disambiguating certain POS tags. It takes the LexMan output and uses a set of declarative rules, based

on the concept of pattern matching, in order to modify the initial segmentation or to disambiguate some POS tags based on the context of the words.

When it comes to modifying segmentation, RuDriCo2 performs both expansion and contraction. Expansion is performed for example on the word *ao*, a contraction of the preposition *a* (to) and the article *o* (the), and results in a separate segment for each of the contracted elements. This module also changes the initial tokenization and groups together the elements of several compound words. This can be exemplified by the compound preposition *em cima de* ('on top of'). It results from the combination of *em cima* (a locative adverb, 'on top') with the preposition *de* 'of'. There are also context-based tokenization-changing rules. Take the ambiguous segments *cerca* 'fence' and *de* 'of'. If these two segments are followed by a number, then they will be contracted to form the adverb *cerca de* 'around', as in *Comprei cerca de 3kg de laranjas.* ('I bough around 3kg of oranges').

Finally, the context-based disambiguation can also be used for correctly identifying the class of a segment in the case of ambiguity. For example, *a* can be an article, a preposition or a pronoun. If it is preceded by a preposition, then RuDriCo2 will identify it as an article.

2.1.4 MARv4

MARv4 (Ribeiro ; Rodrigues 2007) is another module responsible for resolving morphossyntactic ambiguity. It analyses the labels assigned to each token by the previous module, and chooses the most likely one based on a Hidden Markov Model, using the Viterbi algorithm to compute it. The language model used is based on both trigrams for enconding context information regarding entities, and unigrams for lexical information.

2.1.5 XIP

XIP (Aït-Mokhtar et al. 2002) is the module responsible for performing syntactic analysis in the STRING chain. First, it allows the introduction of additional lexical, syntactic and semantic information to the result of the previous modules, and then it performs syntactic analysis through the sequential application of local grammars and morphossyntactic disambiguation rules, the calculation of chunks and the extraction of dependencies between chunks. XIP uses nodes as the basic data representation structure. Each node has a category (such as *noun*), a set of feature-value pairs (*plural*, `plural=+`), and a set of brother nodes.

XIP itself is composed of several modules:

- Lexicons: used to add information to tokens. XIP supports the definition of custom lexicons, which are used to add new features not yet present on the pre-existing lexicon;

- Local Grammars: consisting in pattern-matching rules that consider the context of a given pattern. These rules are used to identify entities composed by several different elements, and then group those elements into a single entity (e.g. *Ministério da Justiça*, Ministry of Justice);

- Chunking Module: This module is responsible for grouping sequences of nodes into structures called *chunks*. These chunks represent basic phrase types, such as noun phrase (NP), prepositional phrase (PP) or finite verb phrase (VF). This is done through elementary rule-based syntactic analysis. Figure 2.2 shows the output for the sentence *O Pedro foi ao Japão* (Pedro went to Japan). The NP, VP and PP chunks are examples of the XIP nodes, which are processed afterwards by the Dependency Extraction Module;

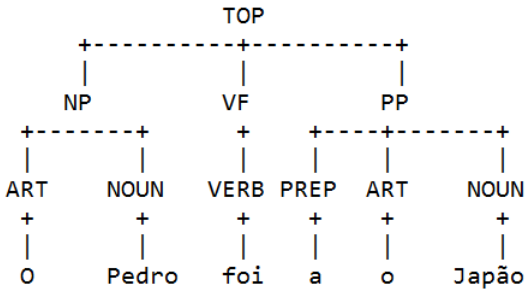

Figure 2.2: XIP Chunking Output Tree

- Dependency Extraction Module: This module extracts the syntactic-semantic relationships between nodes. To do so, it uses a set of dependency rules, which it applies using a logic programming paradigm. A pattern and a set of conditions are matched, and then the consequent of the rule is applied. These rules use the following syntax:

```
|pattern| if <condition> <dependency_terms>.
```

The rule has three components, a pattern, a condition, and the dependency terms. Consider now Example 1, which will be referred to throughout this section.

```
|#1{?*,num[quant,sports_results]}|
if (~NE[quant,sports_results](#1))
NE[quant=+,sports_results=+](#1)
```

Example 1. XIP Dependency Rule

The pattern is composed by a Tree Regular Expression (TRE), and it is used to match parts of the chunking tree in order to apply the dependency rule. A TRE is a special type of regular expression used to establish connections between nodes. In particular, a TRE explores the inner structure of the nodes through the use of curly brackets (`{}`). Consider Example 2:

```
NP{det,noun}.
```

Example 2. Describing the inner structure of a XIP node

This means that the inner structure of an NP node must be inspected in order to check if it is composed by a determiner followed by a noun.

TREs support the use of several operators:

- Semicolon (`;`) - used to represent *disjunction*;
- Asterisk (`*`) - used to represent *zero or more*;
- Question Mark (`?`) - used to represent *any*;
- Circumflex (∧) - used to explore subnodes for a given category.

A special notation is also used to refer to nodes in the pattern, composed by an hashtag (`#`), and a number. In the same way, there is a special notation to refer to features of a given category, which are represented inside square brackets (`[]`).

Consider the pattern in Example 1, `#1{?*,num[quant,sports_results]}`. There is only one node in the pattern, referred to as `#1` (should there be more, they would be referred to as `#2`, `#3`, etc.). This pattern is matched by nodes that have as children any node, followed by a number (also a child) with the features `quant` and `sports_results`, that is, number formats designating sports results (e.g. 7-3).

The condition can be any boolean expression that matches the XIP syntax. This condition must be met for the rule to be fired. Parenthesis are used to group statements and apply precedence, as is typical in programming languages. The usual negation (∼), conjunction (`&`) and disjunction (|) operators are supported. Conditions refer to the nodes presented in the pattern and represented by the variables `#1`, `#2` etc. They also consider dependencies and features. In the condition `if (∼NE[quant, sports_results](#1))` from Example 1, `NE` represents a Named Entity dependency. This is a unary dependency (with only one argument), though the arity of XIP dependencies can vary (usually two or more arguments). Features are represented in the same way as in the pattern. This condition means that the node referred to by `#1` only matches the rule if

it does not yet have a `NE` dependency with the features `quant` and `sports_results` marked positively.

Finally, the dependency terms are the consequent of the rule. Once again the following dependency terms are present in Example 1, `NE[quant=+, sports_results=+](#1)`. This consequent means that the dependency `NE` is added to the node `#1`, and it is assigned the features `quant` and `sports_results`.

So, the rule in Example 1 would mean that if a node has any children nodes, followed by a number with the features `quant` and `sports_results`, and if it has not yet been extracted as a `NE` with the features `quant` and `sports_results`, then the `NE` dependency with the features `quant` and `sports_results` is added to that node.

2.2 ViPEr

ViPEr is a Lexicon-Grammar for European Portuguese Verbs (Baptista 2012). It is a linguistic resource with the syntactic and semantic properties of the most frequent verbs of the Portuguese language. These fine-grained properties, along with the constraints the verbs impose on their arguments (such as the number of arguments, the prepositions each verb uses to introduce its complements or the main shape-changes these structures can undergo) are meant to facilitate the correct identification of verb meanings in texts. ViPEr focuses on distributional (or lexical) verbs, whose meaning allows for an intensional definition of their respective syntactic construction and the semantic constraints on their arguments.

Each verb is described by a syntactic frame composed of the basic constituents of the sentence. The frame takes the form: *N0, Prep1, N1, Prep2, N2, Prep3, N3*. *N0-N3* describe the verb's arguments. *N0* corresponds to the subject of the sentence, and *N1-N3* represent the verb's complements. *Prep1-Prep3* are the prepositions that introduce the respective complement. *Prep1* may be absent in the case of direct-transitive verbs (see below).

Each component of the frame has a set of boolean properties, associated with each verb class. The properties accept one of two values, '+' if the property must be positive, and '-' otherwise. For example, if a verb requires its subject to be a human noun, then it has the property *N0=Hum* marked with a '+'. Other properties include *Npl*, for plural nouns, *QueF* for completive subclauses, *vse* for intrinsically pronominal (reflex) constructions, among others.

There are some restrictions on which properties may be positive for each of the components of the verb. Since *N0* represents the subject of the sentence, it does not have preposition properties. In addition, only the *N1* component, which represents the first verb complement, may lack *Prep1* since in European Portuguese the direct complement does not allow for a preposition. As a result, *N1* is the

Number of classes per lemma	Lemmas (2013)	Lemmas (2016)	Difference
1	4,267	3,888	-379
2	515	586	+71
3	159	189	+30
4	62	76	+14
5	19	21	+2
6	8	11	+3
7	2	3	+1
8	3	5	+2
10	1	1	+0
11	1	1	+0
Total	5,037	4,781	-256

Table 2.1: ViPEr's statistics

only component that allows a positive *Prep1=0* property, meaning that a preposition introducing this complement is not necessary.

Table 2.1 shows a comparison of ViPEr's statistics in 2013 and at the time of writing. The left column indicates the number of classes per lemma, that is, lemmas belonging to 1, 2, or more different classes. These correspond to different word senses that the same lemma can express.

As shown, ViPEr has suffered important changes since its inception. In addition to more verbs being identified as having multiple senses, the properties used to describe each sense have also been further refined, from 114 in 2013 to 128 in 2016.

It should be noted that the difference in the total figures from 2013 to the present date has to do with the fact that an important number of constructions proved to be rarely used in *corpora*. Therefore, those constructions are not integrated into the system, and do not contribute to the ambiguity of the verbs.

2.3 Transformation-Based Learning

TBL is a machine-learning algorithm that has been successfully used in several NLP problems, such as POS tagging and syntactic parsing. It derives linguistic information from an unannotated *corpus*, and builds and improves its linguistic model by manipulating the *corpus* in order to find the closest result to the truth. The truth, in this context, is a manually annotated *corpus* that is used as reference. Figure 2.3 shows the architecture of the algorithm.

The algorithm takes unannotated text, and through an initial state annotator produces the annotated text to be used. Different initial state annotators have been used with this algorithm: For POS tagging, the output of a stochastic n-gram tagger, labelling all words with the most likely tag or labelling all the words as nouns have all been used; for syntactic parsing, initial state annotators have

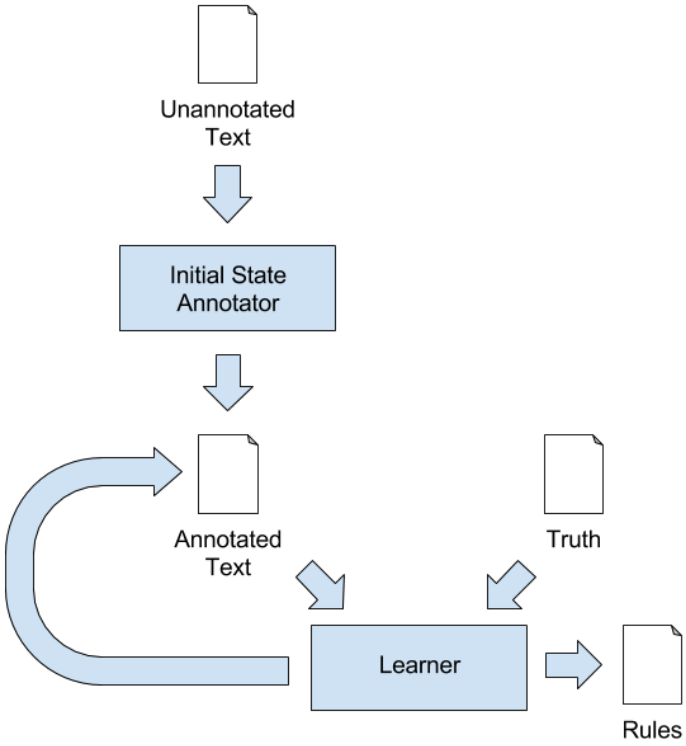

Figure 2.3: Transformation-Based Learning - Architecture

ranged from the output of sophisticated parsers to random tree structures with random non-terminal labels.

Following the initial state annotation, a learner compares the annotated text to the truth, in order to produce transformations that drive the annotated text closer to the reference.

These transformations are composed by a rewrite rule, and a triggering environment. The rewrite rule indicates how to transform the text, and the triggering environment determines the conditions in which this transformation should be applied.

The learner follows an iterative process, in which different transformations are applied individually to the annotated text, and the best one is chosen and added to an ordered list. To produce this list of transformations, several approaches may be used. In the original TBL implementation (Brill 1995), a greedy best-first approach was used. This means that at every iteration, the transformation that drives the annotated text closer to the truth is chosen and then added to the ordered list. However, greedy best-first approaches are usually not optimal, thus other approaches may be used to improve upon this.

Finally, when no more transformations can be reordered in such a way that brings the annotated text closer to the truth, the learner stops, and outputs the final ordered list of transformations.

In STRING, VSD is performed through XIP rules and machine-learning. XIP rules fit the definition of transformation perfectly, they have a triggering environment and a rewrite rule. TBL can then be used to improve the rule-based disambiguation by reordering the disambiguation rules.

3 Verb Sense Disambiguation in STRING

The previous chapter described STRING, a natural language processing chain, and ViPEr, a resource upon which the verb sense disambiguation approaches used in the STRING system are based. This chapter will now present the two approaches to Verb Sense Disambiguation that are currently in place in the STRING chain, along with the ways in which they use the information contained in those resources to perform this task.

3.1 Rule-based disambiguation

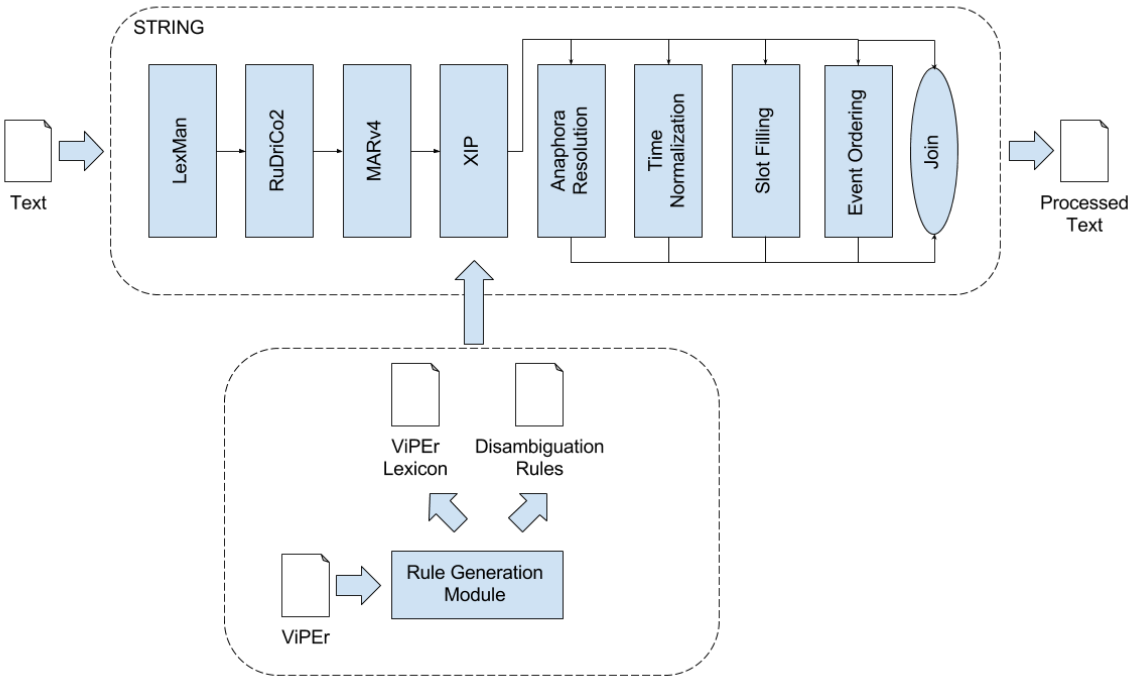

Figure 3.1: Rule-based VSD in STRING

This approach (Travanca 2013) consists in using the information present in ViPEr to generate a set of XIP disambiguation rules, and applying those rules to obtain disambiguated verb senses. The rule-generation is performed by a module that takes ViPEr as input and generates the set of XIP dis-

ViPEr property	XIP dependency	Description
n0	SUBJ	Subject
direct-transitive-n1	CDIR	Direct complement
indirect-transitive-n1	MOD	Modifier
indirect-transitive-n2	MOD	"
indirect-transitive-n3	MOD	"
prep1	PREPD	Preposition
prep2	PREPD	"
prep3	PREPD	"
clitic	CLITIC	Clitic dependency
auxiliary-verb	VLINK	Verbal link

Table 3.1: ViPEr-XIP Dependency Mappings

ambiguation rules. These are then added to the XIP module, which is responsible for applying them. Figure 3.1 shows how this module fits into the STRING chain.

First of all, ViPEr and XIP encode information differently, and as a result ViPEr properties rarely translate directly into XIP features. To address this issue, a mapping file is used to translate ViPEr properties to XIP features. There are three categories of mappings: dependency, node-feature and dependency-feature.

Dependency mappings convert ViPEr properties into a dependency in a XIP rule. If we refer back to Example 1 in Section 2.1.5, the NE category can be referred to as a dependency. Table 3.1 contains the dependency mappings.

In Table 3.1, SUBJ and CDIR dependencies are straightforward, representing the syntactic relation between the verb and its subject and direct complement, respectively. The MOD dependency is an umbrella relation holding between the verb and a prepositional phrase, irrespective of its essential or circumstantial status (which is only solved at a later stage). The PREPD dependency links a preposition to the head noun of the PP chunk it introduces. CLITIC dependency relates a verb and a clitic personal pronoun, irrespective of its enclitic, mesoclitic or proclitic use (at the end, the middle or before the verb, respectively). Finally, VLINK corresponds to the relation between either an auxiliary verb and the main verb (e.g. *tinha* -> *escrito* 'had written'), or between two auxiliary verbs (e.g. *poderá* -> *ter escrito* 'may have written').

Node-feature mappings convert ViPEr properties into XIP node features. A node-feature is a feature attached to a given node in the XIP representation of the text. Consider Example 3.

```
|#1[markviper, lemma:lembrar,13,09I,09I=~]|

if ( (MOD(#1,#2) & PREPD(#2, #3[lemma:"de"])) )

~
```

Example 3. XIP Dependency Rule - Node feature

ViPEr property	XIP feature	Description
vse	`ref`	Reflex construction
dative	`dat`	Dative construction
vdic	`vdic`	Dicendi construction
copulative	`cop`	Copulative construction
Hum	`UMB-Human`	Human
nHum	`~Hum`	Non human
Nloc	`advloc, proxadv, UMB-location` [1]	Location
Npl	`pl, sem-group-of-things`	Plural
Loc	`preploc`	Locative preposition

Table 3.2: ViPEr-XIP Node-Feature Mappings

In this case, node `#3` is further specified with a node-feature, stating that the lemma of this node must be the preposition *de* (of). The meaning of XIP disambiguation rules will be presented later on in this Section. Table 3.2 contains the most relevant mappings required to understand the module. Some of these features are self-evident. The *vse* property refers to intrinsically reflexive constructions (e.g. *suicidar-se* 'suicide himself') so it must be mapped onto a reflex pronoun attached to the verb (through the `CLITIC` dependency; see above). The dative construction represents the indirect complement whose base form is a *PP* that can be reduced to a dative personal pronoun, hence the XIP feature `dat`. The *vdic* property indicates that the verb can be used as a *verbum dicendi*, that is, to introduce direct speech, which allows for a subject inversion transformation (e.g. *"Não sei", disse ele.* '"I do not know", said he'). This is mapped onto a XIP feature introduced by the parser once the *verbum dicendi* construction has been detected. *Copulative* verbs are not included in ViPEr, though it is necessary to take them into account to adequately parse passive sentences with auxiliary (copulative) verbs *ser* 'be' or *estar* 'be' and its variants. The corresponding XIP feature is `cop`. XIP also uses a generic feature `UMB-Human` to encapsulate all nouns referring to human entities, which is thus mapped onto the distributional constraint *Hum* represented in ViPEr. The distributional constraint *Nloc* corresponds to locative complements. These are mapped onto locative adverbs (`advloc` and `proxadv`) as well as the generic XIP feature for locative nouns (and named entities), which encapsulates several other XIP features. The distributional constraint *Npl* (plural noun) distinguishes several verb classes (e.g. *dispersar* 'disperse') that impose a given syntactic slot to be filled by a noun in the plural (XIP feature `pl`, e.g. *Os alunos dispersaram.* 'The students dispersed.') or by a collective noun (XIP feature `sem-group of things`, e.g. *A multidão dispersou.* 'The crowd dispersed.'). Finally, the *Loc* distributional feature is used for locative prepositions, represented in XIP by `preploc`.

Finally, dependency-feature mappings convert ViPEr properties into XIP dependency features. In Example 1, the `[quant,sports_results]` that follow the category `NE` are dependency features. So far, only the ViPEr property `QueF` indicating the sentential nature of a verbal argument (subject or com-

[1] The `Nloc` property maps into several more XIP features, though they were omitted here for the sake of simplicity.

plement) is mapped onto dependency-features `sentential`, `completive` and `inf` attached to the main clause constituent dependencies (`SUBJ`, `CDIR` and `MOD`).

As for the rule-generation process, this is composed of several steps. The first step is difference-finding. Only the subset of ambiguous verbs from ViPEr is then processed. These are the verbs that have two or more constructions associated with the same lemma. Each construction is represented by its ViPEr class, and corresponds basically to a different word sense [2].

For each ambiguous verb, each construction is compared pairwise against the other constructions it may have, generating a difference for each pair of constructions. A difference, in this context, is the set of differences in the properties specified for one construction whose values are distinct from the values of the same properties in the construction it is being compared to.

After all the differences are found, the rules are built. A XIP condition is generated based on the type of difference, and on the presence (or absence) of certain ViPEr features on one or both verb classes of the difference. The information is added to a XIP dependency rule template. This template has the following format:

```
|#1[markviper, lemma:<verb lemma>,<class1>,class2>,<class2>=~]|
if ( <XIP condition> )
~
```

This results in a XIP dependency rule that is applied to any node with the XIP feature `markviper`, with the lemma present in `<verb lemma>`, and classes `<class1>` and `<class2>`. The rule removes `<class2>` from the node if the `<XIP condition>` is verified. The feature `markviper` in the XIP output is added to all verbs represented in ViPEr.

As previously mentioned, the knowledge used to generate the rules is embedded in the code of the rule-generation module code. This has obvious disadvantages: it is very difficult to change the rule-generation process, since it requires understanding and changing the code itself; the maintainability of the module is low, meaning some changes in ViPEr can break its functionality. In order to cope with this design, we undertook a comprehensive and detailed analysis of the module's code in order to extract the knowledge embedded in it. Since the whole process is complex, an abstraction is used to represent this knowledge, using a special syntax. This is illustrated in Rule 1.

[2] Rarely, the same verb sense can be associated to different syntactical structures, as a result, for example, of complement preposition. This entails splitting the verb in two distinct classes. *O Pedro namora (com) a Ana* 32H/35S

Rule 1

```
ViPEr class 1:   {<positive ViPEr features>}
                 {<negative ViPEr features>}
ViPEr class 2:   {<positive ViPEr features>}
                 {<negative ViPEr features>}
XIP conditions:  <conditions>
```

In Rule 1, there are three segments: `ViPEr class 1`, `ViPEr class 2` and `XIP conditions`. The first two segments contain two sets of ViPEr features each, one of positive features, and one of negative features. The condition segment is only generated if all four sets of features are verified.

There are three categories of differences: *Structural Differences*, *Preposition Differences* and *Complement Differences*. Throughout this section, one rule of each category will be presented and explained.

Structural differences are differences in the type of structural construction the verb class allows. These structural constructions are: Dicendi, Dative, Reflex and Passive (either with the auxiliary verb *ser* 'be' or *estar* 'be' and its variants).

Consider Example 4, which represents the structural and semantic difference between two distinct constructions of the verb *acrescentar* 'add', corresponding to two classes in ViPEr. The construction of class 06 allows for a *verbum dicendi* construction, whereas that of class 36R does not.

`06` - *O Pedro acrescentou que a Ana é uma pessoa competente.* (Pedro added Ana is a competent person)

`36R` - *O Pedro acrescentou um pacote de açúcar raro à sua colecção.* (Pedro added a rare sugar packet to his collection)

Example 4. Two structurally different verb constructions of verb *acrescentar* 'add'

As mentioned above, the *verbum dicendi* property indicates that the verb is used to introduce direct speech, with subject-verb inversion (Baptista 2010), as in *"A Ana é uma pessoa competente", acrescentou o Pedro* '"Ana is a competent person", Pedro added'. This property is mapped onto a XIP feature, `vdic`, which is added to the verb by XIP once the subject-verb inversion pattern, along with the direct speech clause, is detected. Rule 2, will then be used to generate the XIP rule responsible for disambiguating these two verb constructions. This takes the form of the rule in Example 5.

Rule 2

```
06:              {Vdic} {}
36R:             {} {Vdic}
XIP conditions:  HEAD(#1,?[vdic])
```

```
|#1[markviper, lemma:acrescentar,06,36R,36R=~]|
if ( HEAD(#1,?[vdic] )
~
```

Example 5. Generated Rule - Verbum Dicendi Difference

The rule in Example 5 reads as follows: the ViPEr class feature 36R on the verb *acrescentar* is discarded if the verb is marked with the feature `vdic`, that is, if the parsing detected that verb in a pattern corresponding to a *verbum dicendi* construction.

Next, *preposition differences* are differences in the *Prep* fields that introduce each complement of a verb. These differences restrict the type of XIP dependency that can be generated.

`35R` - *O Pedro confia no João.* (Pedro trusts João)

`36DT` - *O Pedro confiou um livro ao João.* (Pedro entrusted a book to João)

Example 6. Preposition difference between two verb constructions of verb *confiar* 'trust/entrust'

Consider Example 6. Construction 35R has only one complement, *N1*, a prepositional (oblique) complement. Construction 36DT however, has two complements: *N1*, *um livro* 'a book' is the direct complement, and *N2*, *ao João* 'to João' is the indirect (dative) complement. The prepositions that introduce *N1* in both constructions are different. Construction 35R has the preposition *em* (*no* is the contraction of the preposition *em* with the article *o*) introducing its *N1*, while construction 36DT has no preposition introducing its *N1*. As a result, the property `Prep1=0` is marked with a '-' in construction 35R and with a '+' in construction 36DT. This will generate a preposition difference, shown in Rule 3, which will then be used to generate the XIP disambiguation rule in Example 7.

Rule 3

```
36DT:             {Prep1=0} {}
35R:              {} {Prep1=0}
XIP conditions:   CDIR(#1,?)
```

```
|#1[markviper, lemma:confiar,36DT,35R,35R=~]|
if ( CDIR(#1, ?)  )
~
```

Example 7. Generated Rule - Prep1 Preposition Difference

The rule in Example 7 reads as follows: for the verb *confiar*, the ViPEr class 35R is discarded as this verb construction is marked with the feature `Prep1=0`, that is, if the parsing detected that verb in a construction with a direct complement. This example has a particularity: a direct complement may only be present in the *N1* argument in ViPEr, thus this rule can only be applied to *Prep1*.

Finally, *complement differences* describe differences in the complements themselves. These differences determine additional node-feature restrictions to be applied to the XIP dependency determined by the prepositions. Consider Example 8.

02 - A atitude do Pedro conta imenso para o seu sucesso. (Pedro's attitude really matters for his success)

32PL - O Pedro contou os alunos. (Pedro counted the students)

Example 8. Complement difference between two verb constructions of verb *contar* 'matter / count'

Construction 02 has a prepositional complement introduced by *para* 'to'. Construction 32PL also has one complement, *N1 os alunos*, which is a direct complement, meaning it is marked with a '+' in the `Prep1=0` property, and it refers to a human and plural noun, which means it is marked with a '+' on the properties *Hum* and *Npl*. As mentioned above, the difference in the *Prep1* results in a `CDIR` XIP dependency. The difference in the *N1* properties produces the node-features `UMB-Human` (from the mapping of the `Hum` property), and `pl` and `sem-group-of-things` (from the mapping of the `Npl` property). Thus, rule 4 is used to generate the XIP disambiguation rule present in Example 9. Note that, since `Npl` has two mapping values, the condition is folded for each of them, and the resulting conditions are grouped using logic disjunction.

Rule 4

```
32PL:             {Npl, Hum, Prep1=0} {}
02:               {} {Npl, Hum}
XIP conditions:   CDIR(#1,?[pl,UMB-Human]) |
                  CDIR(#1,?[sem-group-of-things,
                  UMB-Human])
```

```
|#1[markviper, lemma:contar,32PL,02,02=~]|
if ( CDIR(#1, ?[pl,UMB-Human]) | CDIR(#1, ?[sem-group-of-things,
UMB-Human]) )
~
```

Example 9. Generated Rule - N1 Complement Difference

This rule reads as follows: the ViPEr class feature 02 on the verb *contar* is discarded if the verb construction is marked with the features `Prep1=0`, `Npl` and `Hum`, that is, if the parsing detected the verb in a construction with a direct complement, and the noun in the complement is a plural noun (Npl) and it refers to a human entity.

The examples above illustrate the rule generation process. The remaining rules, not presented in this section, also follow this general process.

This sums up the process of generating XIP Dependency rules. The generated rules are then added to the XIP module, and used for disambiguation.

These rules were used together with the Most Frequent Sense (MFS) to determine the correct sense of a verb. According to (Travanca 2013), the MFS yields better results for most verbs, while rules yield much better results for a small number of highly ambiguous verbs. Thus, the best results are produced by combining the rules and the MFS. This is done by using a MFS classifier to decide the verb sense of the instances that are still ambiguous after rule-based disambiguation.

3.2 Machine-learning-based disambiguation

This approach follows a typical supervised classification procedure. It is divided in two steps, training and prediction, and it involves three modules: a feature extraction module, a machine learning algorithm and a classification module. Figure 3.2 shows the general architecture of the machine-learning architecture currently in place in STRING.

In the training step, a feature extraction module within STRING is responsible for extracting the features that are used to describe the verb senses.

To build the training *corpus*, a set of ambiguous verbs was selected according to the number of instances left to disambiguate after the rule-based step. This set was collected from the CETEMPúblico (Santos and Rocha 2001) *corpus*. For each verb, about 500 sentences were collected. These sentences were then manually annotated, and reviewed by linguists. The annotation process consisted in assigning a ViPEr class to the target verb of each sentence.

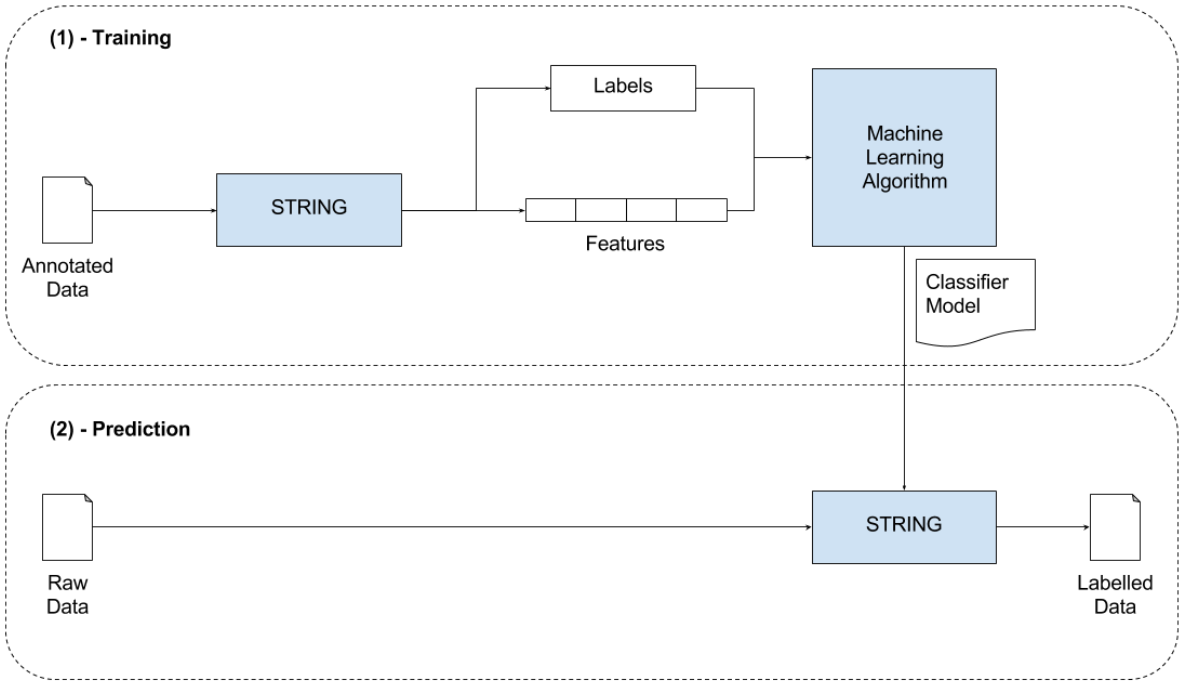

Figure 3.2: Machine-Learning-basedVSD in STRING

The features used can be separated into three groups:

1. *Local* (or contextual) *features*, which describe the information surrounding the target word. A window of size 3 is considered around the target verb. For each of these tokens, its Part of Speech (POS) tag and lemma are collected;

2. *Syntactic features*, which describe the sentence nodes with a XIP dependency relation with the target verb. For each of these nodes, the POS tag, the lemma, and the name of the dependency relation were extracted. Despite XIP containing several types of dependencies, only `SUBJ` (subject), `CDIR` (direct object) and `MOD` (modifier) are considered; and

3. *Semantic features*, which describe the head word of nodes that have a XIP dependency relation with the target verb. Each semantic trait is accompanied by the name of the dependency. These features are also present in ViPEr, such as *human*, *location*, and *animal*.

These features, along with their reference class, are then passed to the machine learning algorithm, which then builds a classifier model to be used in the prediction step. There are two algorithms currently implemented: MegaM (Daumé 2004), an implementation based on Maximum Entropy Models, and an implementation based on the Naive Bayes algorithm (Suíssas 2014). Other algorithms were tested, namely Support Vector Machine (SVM)s, Decision Trees, ID3 algorithm, CART method and Conditional Random Fields (CRF)), however, they have been discarded for they could not produce good results.

In the prediction step, the feature extraction module once again extracts the features associated with the verb to classify. Then, the classification module loads the correct model for the verb lemma of the instance being classified.

The major limitation of this approach is the need for a manually annotated *corpus* for each lemma, in order to generate the corresponding model. Manually annotating *corpora* is a non-trivial task, that must be supervised and reviewed by linguists. At the time of writing, this approach supports twenty four verb lemmas.

4 Proposal

The previous chapter introduced the current approaches to Verb Sense Disambiguation used in the STRING chain. Their inner workings were explained in detail, along with the way they use the information contained in ViPEr to produce the rules, and the models implemented in STRING to perform disambiguation. However, these approaches have some shortcomings, and the main goal of this project is to attempt to address these issues in order to improve the quality of VSD in STRING. Section 4.1 presents a proposal for a new approach to rule-based disambiguation in STRING.

The previous chapter also introduced the TBL algorithm. Since the order of the disambiguation rules has an impact on the results of disambiguation, TBL can be used as a means of reordering the disambiguation rules. This was made possible by the simplicity of the rule-based proposal presented in this chapter. Section 4.2 presents a proposal for a solution based on TBL.

4.1 Rule-generation Module

Figure 4.1 shows how the rule-generation module here developed fits in the STRING chain. This module takes ViPEr and a set of declarative rules as input, and produces a set of XIP dependency rules as output. This set of rules is then used by XIP to perform the actual verb sense disambiguation.

Figure 4.2 shows the architecture of the new rule-generation module (cp. 3.1). The module's processing is carried out in two stages. In the first stage, both ViPEr and the set of declarative rules are parsed, and the information contained therein is stored. In the second stage, the module uses the stored information and applies the declarative rules to the verbs present in ViPEr. Section 4.1.1 will describe the declarative rules used in this process. Section 4.1.2 will present the structure and the inner workings of the module itself.

4.1.1 Declarative Rules

As mentioned in Section 3.1, the previous approach to rule-generation had a major disadvantage, in that most of the knowledge required for this process was embedded in the code of the module. In order to avoid this, a new way of representing this knowledge was devised. Instead of being embedded in the code, this knowledge is now represented under the guise of declarative rules.

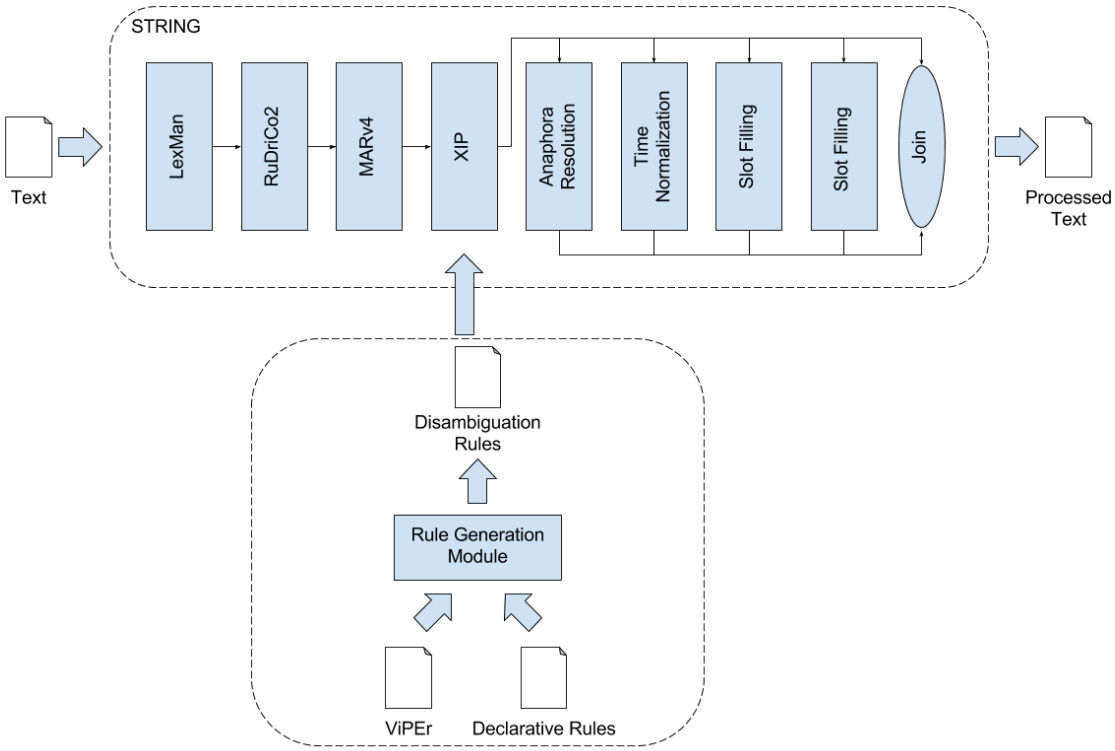

Figure 4.1: Rule-generation Module in STRING

These rules follow the format:

```
VIPER  v1{<props +>}{<props ->}{<pairs +>}{<pairs ->}
       v2{<props +>}{<props ->}{<pairs +>}{<pairs ->}
       {{<XIP condition>}}
```

This means:

- `v1` refers the first ViPEr class of the pair to disambiguate
- `v2` refers the second ViPEr class of the pair to disambiguate
- `<props +>` is the set of ViPEr properties that the ViPEr class must have in order for the XIP rule to be generated;
- `<props ->` is the set of ViPEr properties that the ViPEr class must not have in order for the XIP rule to be generated;
- `<pairs +>` is the set of pairs[1]of ViPEr properties that the ViPEr class must have in order for the XIP rule to be generated (Note that the class is considered as not having a given pair if it does not have at least one of its properties);

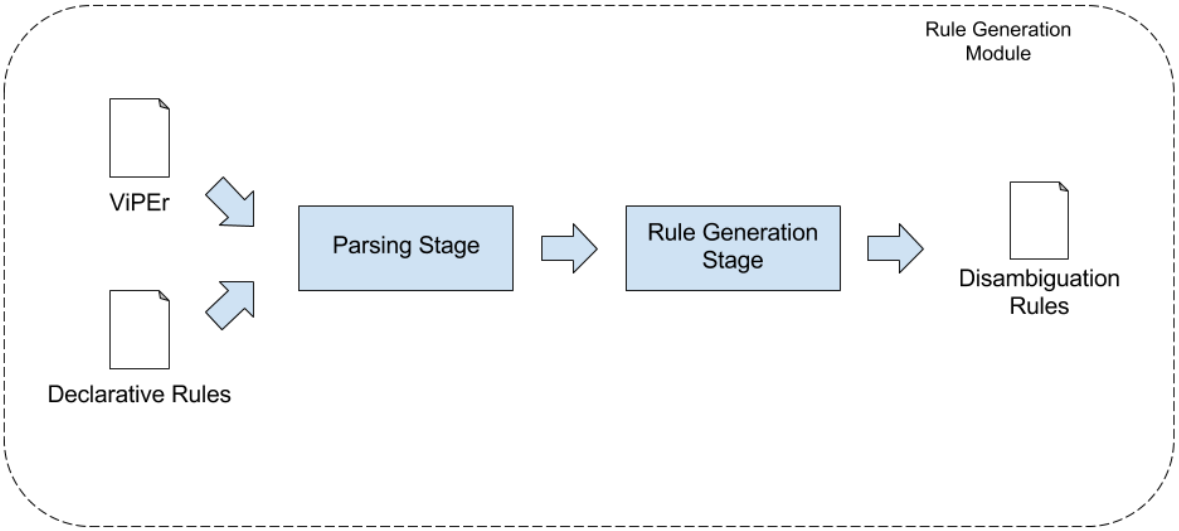

Figure 4.2: Rule Generation Module Architecture

- `<pairs ->` is the set of pairs of ViPEr properties that the ViPEr class cannot have in order for the XIP rule to be generated;
- `XIP Condition` is the XIP condition of the generated rule (Note that the argument `#1` refers to the verb being disambiguated);

The following examples will explain in further detail the semantics of the rules:

Consider Declarative Rule 1.

```
VIPER  v1{vse}{}{}{}
       v2{}{vse}{}{}
       {{CLITIC(#1,?[ref])}}
```

Declarative Rule 1. Single Property Rule

This rule means that for a given pair of ViPEr classes `v1` and `v2`, class `v2` is removed from the verb if the following conditions are true:

- If `v1`, in ViPEr, has a '+' in the ViPEr property `vse`;
- If `v2`, in ViPEr, has a '-' in the ViPEr property `vse`;

[1] A pair of features is identified by the ampersand (`&`) that join them. A ViPEr class is considered to have a pair of features if both features are marked with a '+'

- If the verb is the first argument of a `CLITIC` XIP dependency, and the second argument of the `CLITIC` XIP dependency has the `ref` XIP feature. In plain English, an intrinsically pronominal verb construction (`vse`) must show a reflex clitic pronoun.

Consider Declarative Rule 2.

```
VIPER  v1{N1=Hum,N1:cdir}{}{}{}
       v2{}{N1=Hum,N1:cdir}{}{}
       {{CDIR(#1[transf-passiva:~],#2[UMB-Human])}}
```

Declarative Rule 2. Multiple Property Rule

This rule means that for a given pair of ViPEr classes `v1` and `v2`, class `v2` is removed from the verb if the following conditions are true:

- If `v1`, in ViPEr, has a '+' in both ViPEr properties `N1=Hum` and `N1:cdir`;
- If `v2`, in ViPEr, has a '-' in the `N1=Hum` and the `N1:cdir` ViPEr properties;
- If the verb is the first argument of a `CDIR` XIP dependency and it does not have the `transf-passiva` XIP feature (that is, the verb is in an active sentence form), and the second argument of the `CDIR` XIP dependency has the `UMB-Human` XIP feature.

In other words, a direct-transitive verb construction with a human direct (object) complement is selected if such pattern is found in an entire sentence, while other constructions of the same verb are discarded if they do not have such properties.

Consider now Declarative Rule 3.

```
VIPER  v1{}{}{N123=Hum&Prep123=a}{}
       v2{}{N123=Hum,Prep123=a}{}{}
       {{MOD(#1,#2[UMB-Human]) & PREPD(#2,?[surface:"a"])}}
```

Declarative Rule 3. Positive Pairs and Negative Properties Rule

This rule means that, for a given pair of ViPEr classes `v1` and `v2`, class `v2` is removed from the verb if the following conditions are true:

- If `v1`, in ViPEr, has a '+' in any of the pairs `N1=Hum&Prep1=a`, `N2=Hum&Prep2=a` or `N3=Hum&Prep3=a`;
- If `v2`, in ViPEr, has a '-' in all the properties `N1=Hum`, `Prep1=a`, `N2=Hum`, `Prep2=a`, `N3=Hum` and `Prep3=a`;

- If the verb is the first argument of a `MOD` XIP dependency, and the second argument of the `MOD` XIP dependency has the `UMB-Human` XIP feature;
- If the second argument of the `MOD` XIP dependency is the first argument of the `PREPD` XIP dependency, and the second argument of the `PREPD` XIP dependency has the XIP feature `surface:"a"`;

In plain words, given two verbal constructions, one is discarded if it does not allow for a prepositional complement filed with a human noun and introduced by the preposition *a* if such patter is found for the target verb, and the other construction does.

Finally, consider Declarative Rule 4.

```
VIPER  v1{}{}{N123=o-facto-de-Vinf&Prep123=a}{}
       v2{}{}{}{N123=o-facto-de-Vinf&Prep123=a}
       {{MOD[sentential](#1[transf-passiva:~],#3) &
       VDOMAIN(#2,#3) & INTROD(#4,#3) & QBOUNDARY(?[surface:"a"],#4,?)}}
```

Declarative Rule 4. Multiple Pairs Rule

This rule means that for a given pair of ViPEr classes `v1` and `v2`, class `v2` is removed from the verb if the following conditions are true:

- If `v1`, in ViPEr, has a '+' in any of the pairs `N1=o-facto-de-Vinf & Prep1=a`, `N2=o-facto-de-Vinf & Prep2=a`, or `N3=o-facto-de-Vinf & Prep3=a`;
- If `v2`, in ViPEr, has a '-' in at least one of the features of the pairs `N1=o-facto-de-Vinf&Prep1=a`,

  `N2=o-facto-de-Vinf&Prep2=a`

  or `N3=o-facto-de-Vinf&Prep3=a`;
- If there is a `MOD` XIP dependency with the feature `sentential`, of which the verb is the first argument, and the verb does not have the feature `transf-passiva`;
- If the second argument of the `MOD` XIP dependency is the second argument of the `PREPD` XIP dependencies `VDOMAIN` and `INTROD`;
- If the first argument of the XIP dependency `INTROD` is the second argument of the XIP dependency `QBOUNDARY`;
- If the first argument of the XIP dependency `QBOUNDARY` has the XIP feature `surface:"a"`;

In other words, a verb construction is discarded if the target verb shows a sentential complement introduced by preposition *a* and the operator noun *facto* in an active sentence form, if that verb construction does not allow for those properties and there is another verb construction that does.

These examples cover most of the declarative rules used to generate XIP disambiguation rules. Note that, while not described in any of the examples, the properties of the format `N123=X` should be interpreted as `N1=X & N2=X & N3=X` when used in the context of the `<props +>` field. The same applies to properties of the format `Prep123=Y`.

The next section focuses on the rule-generation process, and how the rules described in this section are used to produce the XIP disambiguation rules.

4.1.2 Rule-generation Module

In this section, the rule-generation module is explained in detail, with Section 4.1.2.1 detailing the Parsing stage, and Section 4.1.2.2 describing the Generation process.

4.1.2.1 Parsing Stage

The first step in the rule-generation process, is the parsing of the declarative rules. The information present in these rules is then stored in a `Rule` object. Figure 4.3 shows the UML for the `Rule` class.

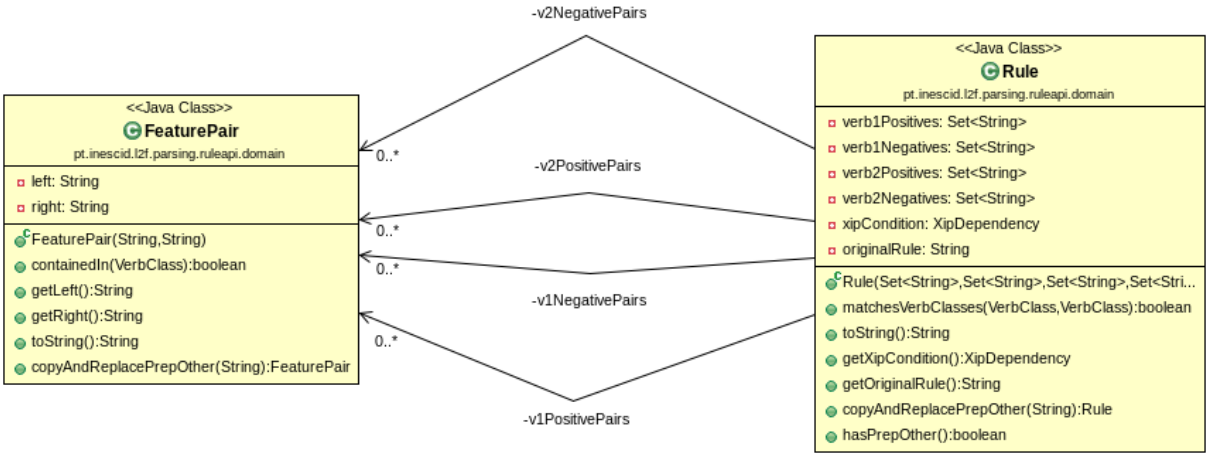

Figure 4.3: Rule class diagram

The attributes of the `Rule` class directly match those of the syntax of the declarative rules previously introduced in Section 4.1.1. The attributes `verb1Positives`, `verb1Negatives`, `verb2Positives` and `verb1Negatives` represent respectively the `<props +>` and `<props ->` fields of verb construction `v1`, and `<props +>` and `<props ->` fields of verb construction `v2`. These attributes are `Sets` of strings, meaning that they do not contain duplicates.

The `FeaturePair` class is used to represent pairs of ViPEr features such as `N1=Hum&Prep1=a`. It is composed of two strings, `left` and `right`. Each string represent a ViPEr property. In the case of the `N1=Hum&Prep1=a`, `N1=Hum` would be stored in the `left` attribute, while `Prep1=a` would be stored in the `right` attribute. The attributes `v1PositivePairs`, `v1NegativePairs`, `v2PositivePairs` and `v2NegativePairs` are thus `Sets` of `FeaturePair` objects, and represent respectively the `<pairs +>` and `<pairs ->` fields of `v1`, and `<pairs +>` and `<pairs ->` fields of `v2`.

Afterwards, the ViPEr file is parsed. For each verb therein, the information regarding the ViPEr properties is stored in a `Verb` Java object. Note that verb senses marked with either the `Filter Exotic` or the `Filter XIP (not4xip)` are ignored, since they're not used in XIP. Figure 4.4 shows the UML diagram for the `Verb` class.

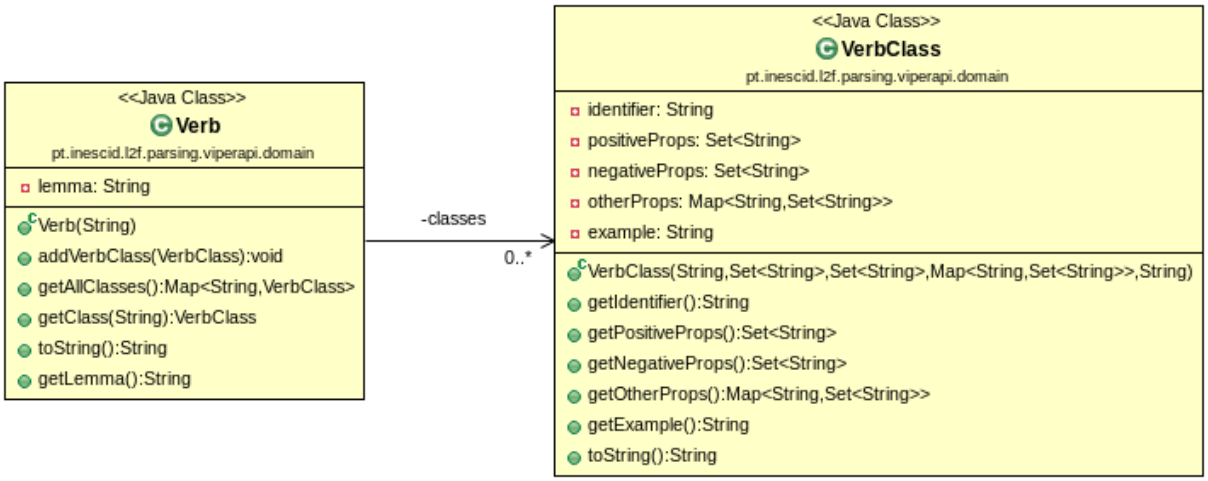

Figure 4.4: Verb class diagram

Each `Verb` object represent a verb present in ViPEr. It contains a dictionary of `VerbClass` objects, the `classes` attribute. This dictionary uses the ViPEr identifier of the verb sense as the key to the corresponding `VerbClass` object.

The `VerbClass` class represents a ViPEr verb sense of a given verb. The `positiveProps` and `negativeProps` attributes are `Sets` of strings containing respectively the ViPEr properties marked with a `'+'` and the ViPEr properties marked with a `'-'`. The `otherProps` attribute is used to represent ViPEr properties that have values that are neither `'+'` nor `'-'`. An example would be the `Prep1=other` property, which may have different sets of values for different ViPEr classes. For example, indicative infrequent prepositions selected by the verb, besides those basic prepositions indicated in a binary fashion (*a*, *com*, *de*, etc.). For the verb *abonar* of the ViPEr class `05`, the `Prep1=other` property has the value `a favor de`, as in *O facto de o Pedro fazer isso abona a favor dele* ('The fact that Peter does that favours him'). This is represented in the `otherProps` attribute as a dictionary entry, where the

key is the property name (`Prep1=other` in this example), and the value is a `Set` of strings with all the possible values (a `Set` with the single string `a favor de` in this example).

4.1.2.2 Generation Stage

In this step, the `Rule` objects created during the parsing stages are used to determine for which pairs of verb senses the corresponding XIP rule should be generated. The following pseudo-code illustrates this process:

```
for each (Verb v)
 for each (Rule r)
  for each (VerbClass vc1 of Verb v)
   for each (VerbClass vc2 of Verb v)
    if (r.matchesVerbClasses(vc1, vc2))
       {
        generate xip rule for vc1 and vc2
          with the xip condition in r
       }
```

The `matchesVerbClasses` method of the `Rule` objects is responsible for identifying whether a `Rule` object matches a pair of `VerbClass` objects. This is done by checking if:

- the `VerbClass` object `vc1` contains all the properties in `verb1Positives`;
- the `VerbClass` object `vc1` contains all the property pairs in `verb1PositivePairs`;
- the `VerbClass` object `vc1` does not contain any of the properties in `verb1Negatives`;
- the `VerbClass` object `vc1` does not contain any of the property pairs in `verb1NegativePairs`;
- the `VerbClass` object `vc2` contains all the properties in `verb2Positives`;
- the `VerbClass` object `vc2` contains all the property pairs in `verb2PositivePairs`;
- the `VerbClass` object `vc2` does not contain any of the properties in `verb2Negatives`;
- the `VerbClass` object `vc2` does not contain any of the property pairs in `verb2NegativePairs`.

If all of these conditions are true, then the a `XipRule` is created. Figure 4.5 shows the class diagram of the `XipRule` class.

The `lemma` attribute corresponds to the lemma of the verb that contains the matching verb senses. The `classToKeep` corresponds to the identifier of the `vc1` object, and the `classToRemove` corresponds to the identifier of the `vc2` object. `cTKExample` is a string that contains a text example of the

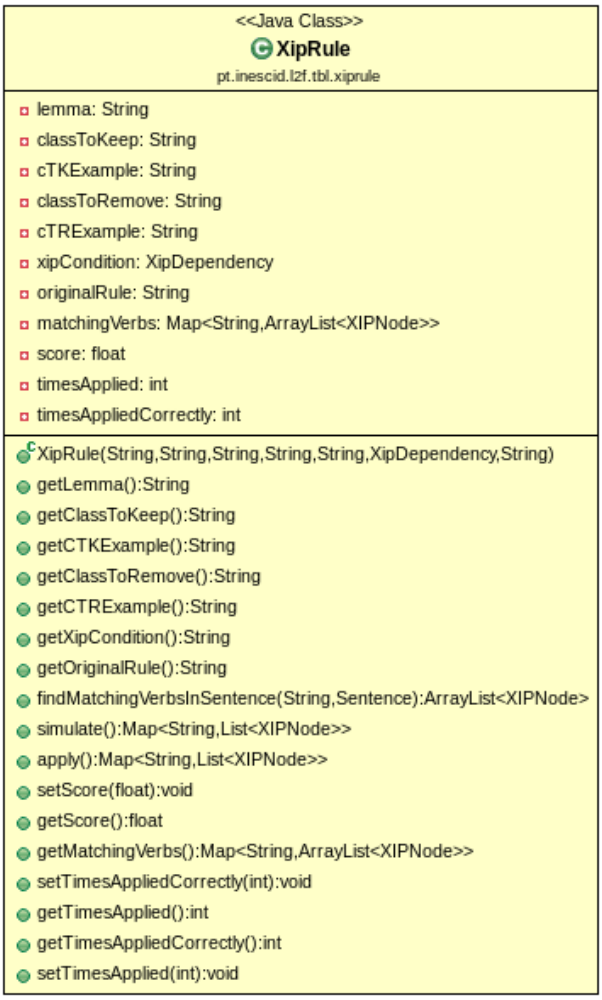

Figure 4.5: XipRule class diagram

verb sense in the `classToKeep` attribute, while `cTRExample` is a string that contains a text example of the verb sense in the `classToRemove` attribute. The `xipCondition` attribute is a `XipDependency` object that represents the XIP condition that must be matched in XIP for the rule to be triggered (the `XipDependency` class will be explained in further detail in the following sections). Finally, the `originalRule` attribute is a string containing the declarative rule that resulted in the creation of the `XipRule` object.

After all the verbs and rules have been processed, then a text file is created. Each `XipRule` object produces a XIP disambiguation rule, that follows the format:

[1]These examples also come from ViPEr

```
//Rule no <rule number>
//<classToKeep> - <cTKExample>
//<classToRemove> - <cTRExample>
//Generated by rule: <originalRule>
|#1[markviper,lemma:<lemma>,<classToKeep>,<classToRemove>,<classToRemove>=~]|
if( <xipCondition> )
~
```

At the time of writing, there are 79 declarative rules, that result in a total of 10,691 XIP disambiguation rules after the generation process.

The rule-generation process has now become much simpler than in the solution currently implemented in STRING. Since the declarative rules contain all the knowledge necessary for disambiguation, the module only checks which pairs of ViPEr classes should be disambiguated according to these rules and generate the respective XIP dependency rules. This makes this approach much easier to maintain and evolve, since the module itself will need at most minor changes on the parsing stage, and all the knowledge-based changes can be applied only to the rules with virtually no impact on the module itself.

4.2 Transformation-based Learning Module

This section describes the implementation of the TBL system. This system does not perform verb sense disambiguation, but instead attempts to improve the basis of the actual disambiguation performed by XIP. This is done by using the TBL algorithm to order the disambiguation rules produced by the module presented in the previous section. As mentioned before, rule-order is critical to the VSD task. Figure 4.6 shows the architecture of the system.

The system is composed by six modules:

1. **Rule Generator** - This is the same module described in the previous section. The TBL module operates upon the rules generated by this module. Thus, including the module in the system prevents having to maintain two code bases that perform the same exact functions;

2. **Processed Corpus Parser** - This module is responsible for parsing a *corpus* that has been previously processed by XIP. The result of this module will be used as the learning *corpus* used by the TBL algorithm implementation;

3. **Reference Corpus Parser** - This module, like the previous one, is responsible for parsing the reference *corpus* into the system. This *corpus* is then used by the Evaluator module in the learning process;

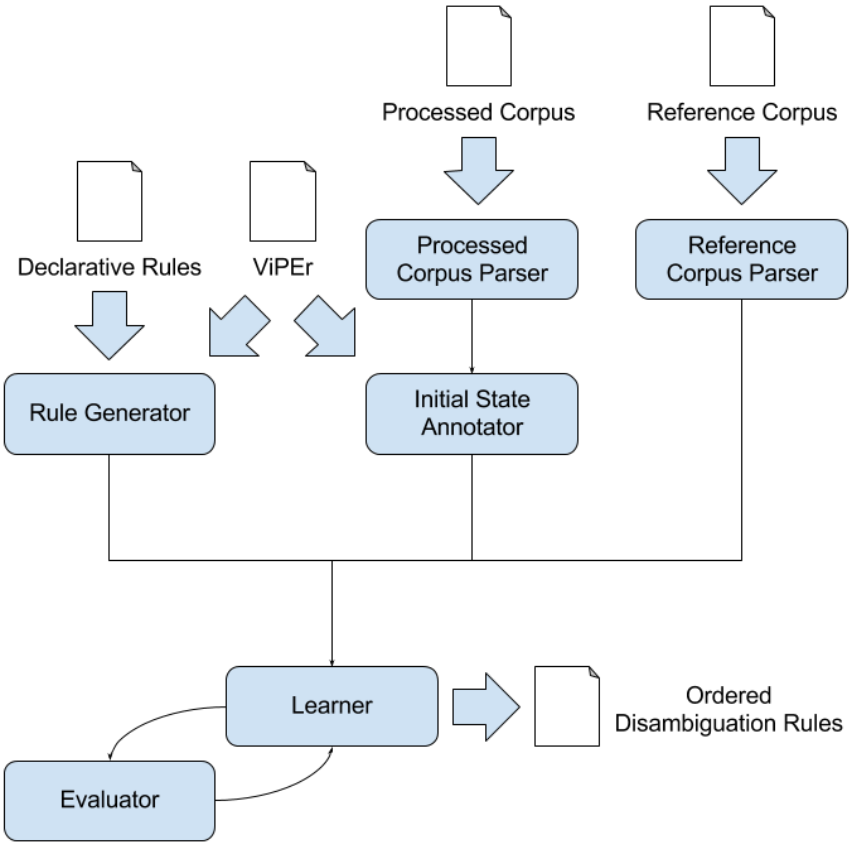

Figure 4.6: TBL Module - Architecture

4. **Initial State Annotator** - This module takes as input both ViPEr and the result of the parsing performed by the Processed Corpus Parser module, and adds all appropriate ViPEr classes to every verb in the *corpus*;

5. **Learner** - This module uses the result of the Processed Corpus Parser module, along with the generated XIP disambiguation rules, to perform the learning process. This process then produces a set of ordered XIP disambiguation rules;

6. **Evaluator** - This module evaluates the results of applying a given XIP disambiguation rule to the working *corpus*. These results are used in the learning process.

These modules (excluding the Rule Generator already described above) will be presented in further detail in the following subsections.

4.2.1 Processed Corpus Parser

This module is responsible for parsing the XML resulting from processing a given *corpus* in XIP. In order to understand the parsing process, it is important for the structure of the XML to be explained. There are seven types of nodes that compose the XML. These types of nodes will be explained, and exemplified with the actual XML resulting from processing the sentence *Simeão II regressa à Bulgária 50 anos depois.* (Simeão II returns to Bulgaria 50 years later.)

- FEATURE - The FEATURE nodes represent a node feature (when the parent is a NODE node) or a dependency feature (when the parent is a DEPENDENCY node). These nodes are composed by an attribute tag that contains the name of the feature, and a value tag that contains the value of the feature. Example 10 presents the XML of a FEATURE node representing the MARKVIPER feature (this feature indicates whether a verb is in ViPEr or not).

  ```
  <FEATURE attribute="MARKVIPER" value="+"></FEATURE>
  ```

  Example 10. FEATURE node

- READING - The READING nodes represent the lemma of a given lexical unit. These nodes have the lemma attribute, which identifies the lemma of that unit, and the pos attribute, which identifies the type of part-of-speech the node represents. These nodes may also have FEATURE nodes as children.

- TOKEN - TOKEN nodes represent tokens. They have three attributes: the pos attribute (see above); the start attribute, which indicates the index of the first character of the token; and the end attribute, which indicates the index of the last character of the token. These nodes also have a surface, which is the token as it appears in the *corpus*, and a child READING node. Example 11 presents the XML of a TOKEN node representing the token *regressar*:

```
<TOKEN pos="VERB" start="10" end="17">
    regressa
    <READING lemma="regressar" pos="VERB">
      <FEATURE attribute="SEM-CHANGE-PLACE" value="+"></FEATURE>
      <FEATURE attribute="VAI" value="+"></FEATURE>
      <FEATURE attribute="37LD" value="+"></FEATURE>
      <FEATURE attribute="PRES" value="+"></FEATURE>
      <FEATURE attribute="IND" value="+"></FEATURE>
      <FEATURE attribute="3P" value="+"></FEATURE>
      <FEATURE attribute="SG" value="+"></FEATURE>
      <FEATURE attribute="VERB" value="+"></FEATURE>
      <FEATURE attribute="HMMSELECTION" value="+"></FEATURE>
   </READING>
</TOKEN>
```

Example 11. `TOKEN` node

- `NODE` - The `NODE` nodes represent a chunk (refer back to Figure 2.2, p. 6). These types of nodes may have other `NODE` nodes as children, as well as `FEATURE` nodes and `TOKEN` nodes. In addition, these nodes have the `num` tag, that indicate the number of the node (this is used as a unique identifier within an `XML` file), the `tag` attribute, which contains the tag of the chunk, and the `start` and `end` attributes (these represent the same thing as in the `TOKEN` nodes). Example 12 represents the XML of a `VERB` chunk that contains the token *regressa*.

```
<NODE num="4" tag="VERB" start="10" end="17">
    <TOKEN pos="VERB" start="10" end="17">
      regressa
      ...
   </TOKEN>
</NODE>
```

Example 12. `NODE` node

- `PARAMETER` - The `PARAMETER` nodes are used to indicate which nodes are parameters to a given XIP dependency. They contain three attributes: the `ind` attribute, which indicates the index of the parameter; the `num` attribute, which indicates the number of the parameter node; and finally the `word` attribute, which contains the lemma of the token contained by the node.

- `DEPENDENCY` - The `DEPENDENCY` nodes represent a XIP dependency. These nodes may contain `FEATURE` nodes, representing dependency features, and they have two or more `PARAMETER`

nodes, representing the dependency's parameters. In addition, they have the `name` attribute, indicating the name of the dependency. Example 13 presents the `XML` corresponding to the subject dependency of the example sentence.

```
<DEPENDENCY name="SUBJ">
    <FEATURE attribute="PRE" value="+"></FEATURE>
    <PARAMETER ind="0" num="4" word="regressar"></PARAMETER>
    <PARAMETER ind="1" num="21" word="Simeão II"></PARAMETER>
</DEPENDENCY>
```

Example 13. `DEPENDENCY` node

- `LUNIT` - The `LUNIT` nodes represent a sentence. They have a single attribute, `language`, that indicates the language of the sentence. In addition, they contain a single `NODE` child, with the value `TOP` on its `tag` attribute. This represents the `TOP` chunk of the sentence, and contains all the other nodes. Finally, they also contain several `DEPENDENCY` children, representing the dependencies within a given sentence. Example 14 presents the XML of the example sentence.

```
<LUNIT language="Portuguese">
    <NODE num="20" tag="TOP" start="0" end="44">
      ...
   </NODE>
    <DEPENDENCY name="SUBJ">
       <FEATURE attribute="PRE" value="+"></FEATURE>
       <PARAMETER ind="0" num="4" word="regressar"></PARAMETER>
       <PARAMETER ind="1" num="21" word="Simeão II"></PARAMETER>
   </DEPENDENCY>
   ...
</LUNIT>
```

Example 14. `LUNIT` node

The `DEPENDENCY` nodes are parsed, and their parameters are used to build `XipNode` Java objects. This process is recursive, and a node tree is built, with the `NODE` node referred by the `PARAMETER` node as the root of the tree. Figure 4.7 presents the `UML` diagram of the `Dependency` and `XipNode` classes, as well as the necessary related classes. Finally, all the dependencies of a given sentence are added to a `Sentence` object. This is merely a container for a set of `Dependency` objects.

Note that there is no object that corresponds to `READING` nodes. This is due to `Token` Java objects containing all relevant information from both `TOKEN` and `READING` nodes. Aside from this, the system contains a close to 1:1 representation of the `XML` node structure in the form of Java objects.

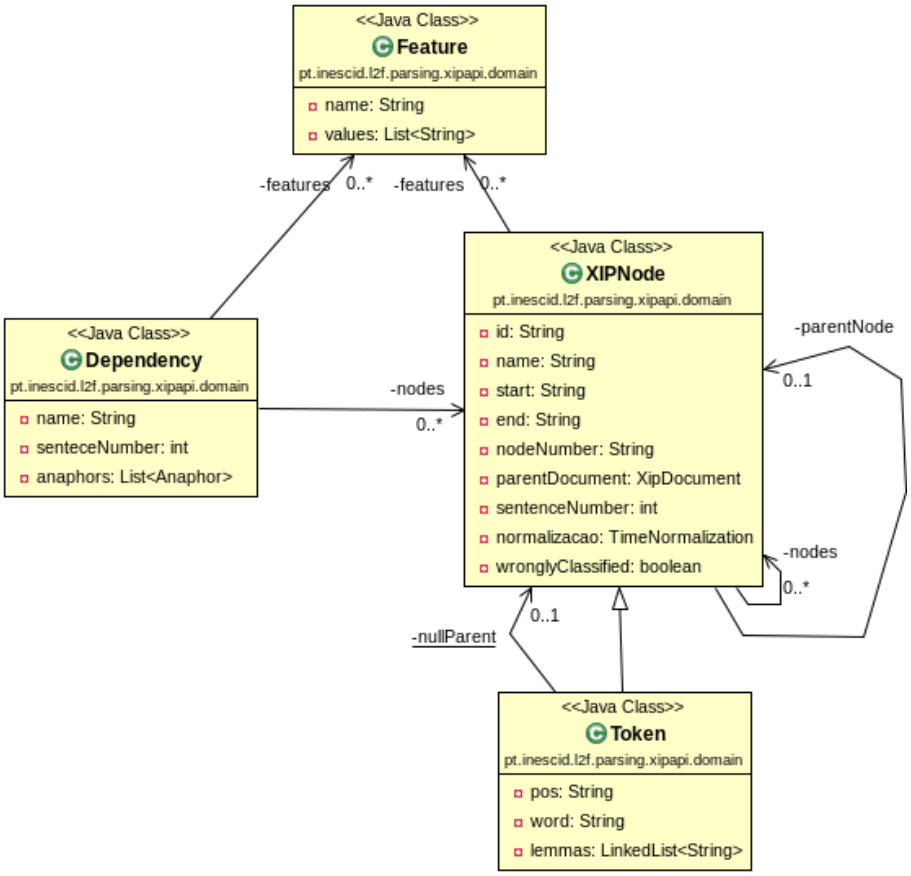

Figure 4.7: Dependency, XipNode and related classes class diagrams

Alongside the parsing process, the system stores which verb lemmas occur in the *corpus*. This is relevant in the last step performed by this module, which is the selection of the relevant verbs from the result of the parsing process. This is done by checking, for each sentence, which verbs match each XIP disambiguation rule.

In Section 4.1.2.2, and for clarity purposes, the `XipRule` class has not been not fully explained. Specifically, the details of the `XipDependency` class were not provided. This class is not very important in the context of generating the rules, but it becomes crucial to the TBL implementation.

XIP is a complex system that performs more tasks than just verb sense disambiguation. As a result, it would not be very efficient to use it in TBL. Thus, a simulator has been implemented. The `XipDependency` class, along with the `XipRule` class, perform this simulation. In the Processed Corpus Parser, each `XipRule` object finds which verbs it may be applied to. This is done through matching its `XipDependency` object with the `Dependency` objects that resulted from the parsing stage. Figure 4.8 shows the class diagram of the former, while Figure 4.9 shows the class diagrams of the latter.

The `XipDependency` abstract class contains the method `findMatchesInSentence`. This method receives a `Sentence` object, and returns all `XipNode` objects containing verbs from the `Dependency`

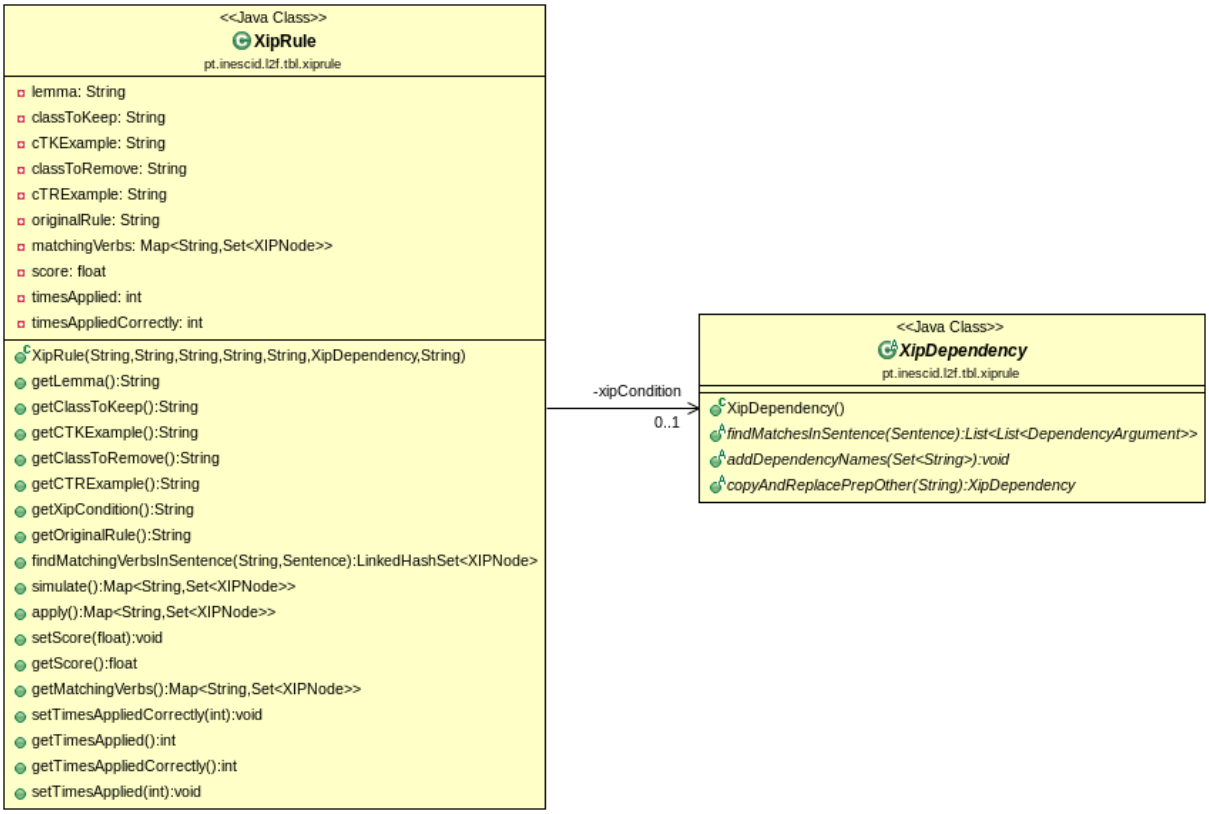

Figure 4.8: XipRule class diagrams

objects that match the dependency. The matching process simulates XIP's. `XipDependency` has four subclasses: `SingleDependency`, `NegatedDependency`, `AndDependency` and `OrDependency`.

`SingleDependency` objects represent XIP conditions with a single dependency, for example `CLITIC(#1,?[ref])`.

`NegatedDependency` objects contain any `XipDependency`, and represent a negative dependency, such as `~CLITIC(#1,?[ref])`. For this example a `NegatedDependency` object containing a `SingleDependency` object for the `CLITIC(#1,?[ref])` dependency would be created.

`AndDependency` objects represent XIP conditions that contain several individual dependencies joined by an `AND` condition, such as in `MOD(#1,#2[UMB-Human]) & PREPD(#2,?[surface:"a"])`. This example is composed of an `AndDependency` object, containing two `SingleDependency` objects, one for `MOD(#1,#2[UMB-Human])` and another for `PREPD(#2,?[surface:"a"])`. In turn, the `OrDependency` objects as with the `AndDependency` are used to represent several individual dependencies joined by an `OR` condition in the same way.

This structure makes use of the *Composite* design pattern (all subclasses of XipDependency are treated in the same way) to create a logical tree, in the same fashion as a traditional compiler would

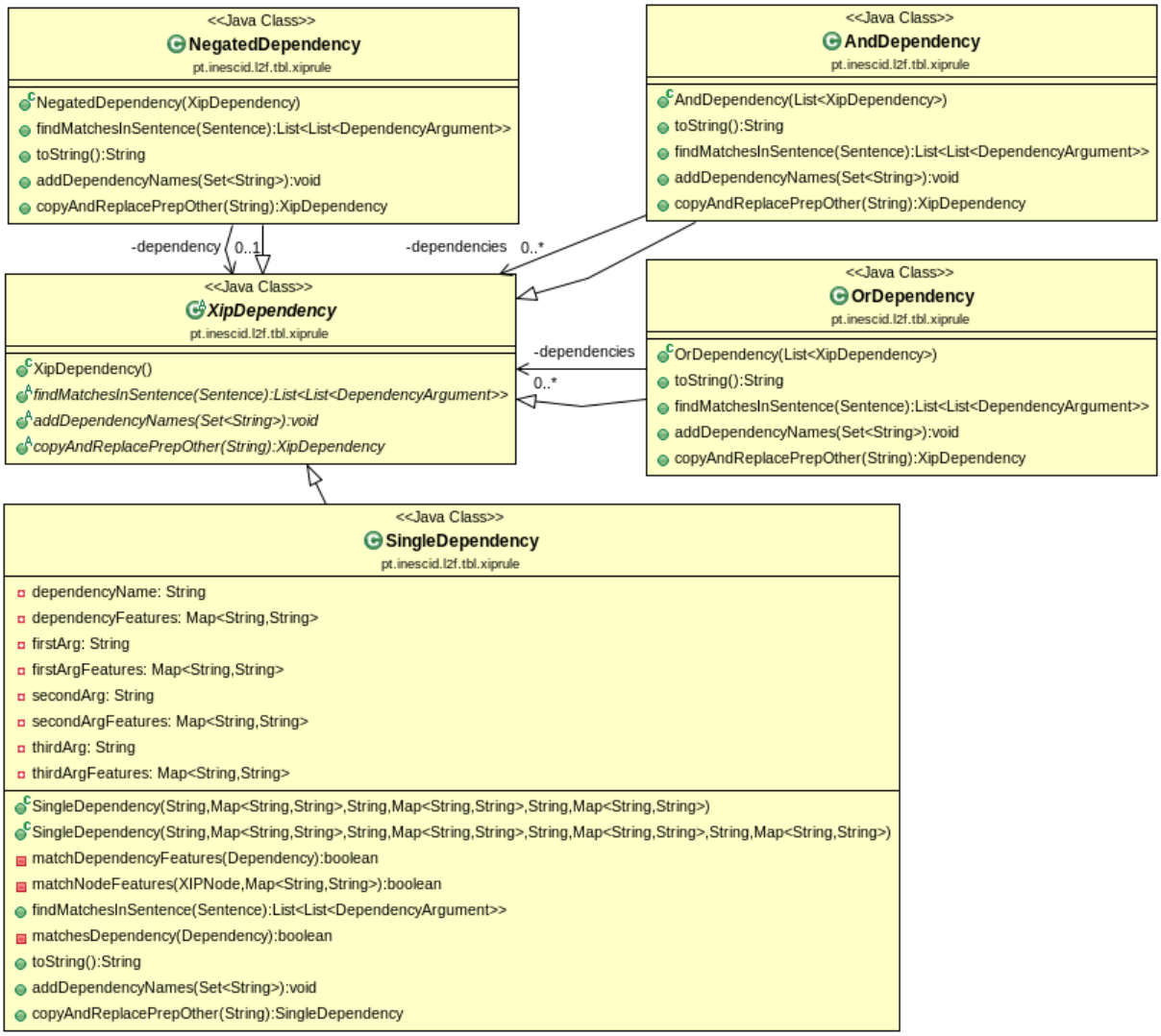

Figure 4.9: XipDependency class diagrams

do for logical expressions. `SingleDependency` objects are the leaf nodes, and contain the basic behaviour of a dependency, while the other objects are the branch nodes that add behaviour on top of that of the leaf nodes.

Afterwards, the rule goes through all the resulting `XipNode` objects to check if the node has the `MARKVIPER` feature (meaning it is an ambiguous verb that is present in ViPEr and should be disambiguated by the XIP disambiguation rules), if the lemma of the node matches that of the rule, and if the node contains a feature for the verb class to be kept and another for the verb class to be removed. All `XipNode` objects that match these conditions are stored in the `matchingVerbs` attribute.

This step means that the *corpus* only needs to be iterated through once. Considering that the Parole *corpus* has around 250,000 words, of which only 21,444 are full verbs (less than 10%); and that of these,

only 12,407 are fit for disambiguation, thus are matched by the XIP rules; this means that the execution time of the system is reduced.

4.2.2 Reference Corpus Parser

This module is responsible for parsing the reference *corpus*, and for storing the information for later use in the Evaluator module.

The reference *corpus*, like the processed *corpus*, is in the `XML` format. However, its structure is significantly different, and much simpler. In the context of this project, only two node types are relevant:

- `w` - `w` nodes represent a token. Among its attributes, only a few are relevant: the attribute `id` indicates the number of the node, and is used as a unique identifier; the attribute `low` indicates the index of the first character of the token; in the same way, the attribute `hig` indicates the index of the last character of the token; the `lemma` attribute indicates the lemma of the token; and finally, the `viper` attribute, present only in verbs, indicates the ViPEr class of the verb. In addition, these nodes also have a surface, which is the token as it appears in the *corpus*.
- `s` - `s` nodes are nodes that represent a sentence. These nodes contain `w` nodes as children.

These nodes are parsed into `Word` and `Sentence` objects respectively. The attributes of the `w` nodes are parsed into `Attribute` objects. Finally, when all sentences are processed, they are stored in a `CorpusDocument` object. Figure 4.10 presents the class diagram for these classes.

4.2.3 Initial State Annotator

This module is executed after both the Processed Corpus Parser and the Reference Corpus Parser modules. It is responsible for adding all possible ViPEr classes of a given verb to every `XipNode` object containing that verb. This is done by adding `Feature` objects for each possible ViPEr class of a given verb to the corresponding `XipNode` object. As an example, consider verb *abanar* (to shake). These verb has two ViPEr classes, `32C` and `32CL`. The Annotator checks if the corresponding `XipNode` object has both classes, and if any of them are missing, it adds them to the object.

4.2.4 Learner

This is the module where the learning algorithm takes place. This algorithm uses a heuristic search algorithm to reorder the XIP disambiguation rules, according to a given heuristic.

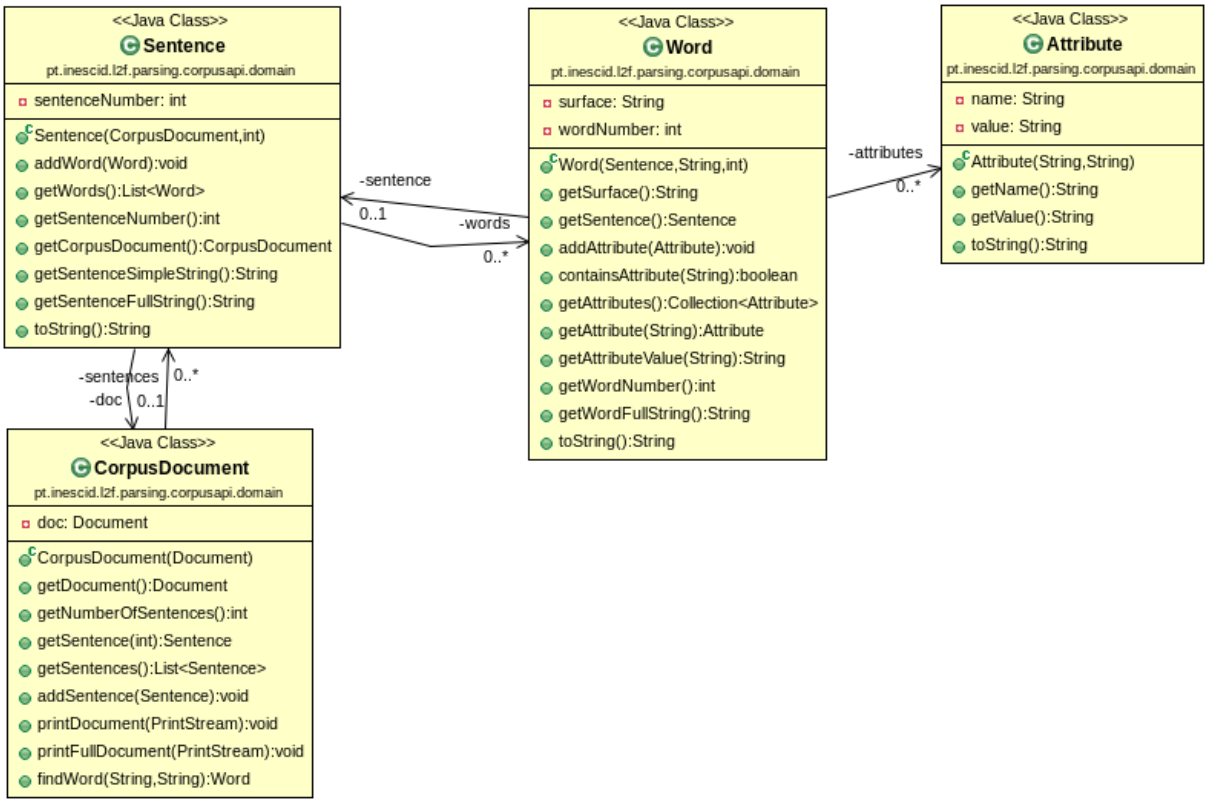

Figure 4.10: Word, Attribute, Sentence and CorpusDocument class diagrams

As mentioned in Section 2.3, TBL may use different tree search algorithms for the learning step. In the context of this project, a Greedy Best-first Search (GBFS) (Russell and Norvig 2002) search algorithm was chosen. It is a simple algorithm, and thus an interesting option as the first algorithm to try TBL with. The heuristic used for this implementation is the accuracy of a given XIP dependency rule.

The implementation of the GBFS algorithm works as follows:

1. Every XIP disambiguation rule is simulated. This is done through the `simulate` method of the `XipRule` class. The `XipRule` object in question goes through all the `XipNode` objects it matches, makes a copy of that `XipNode` object, and removes the appropriate ViPEr class. The copied objects are then evaluated to determine the heuristic value of the `XipRule` object;

2. The XIP rule with the highest heuristic score is then placed in a list and applied to the actual verbs it matches (as opposed to the copies). This is done through the `apply` method of the `XipRule` class. This method does the same thing as the `simulate` method, but on the actual `XipNode` objects instead of copies;

3. If there are still XIP rules that were not placed in the list, then the process is repeated from step 1, otherwise the process ends.

In the end, an ordered list is produced, where the rules are ordered by their accuracy.

Note that it is in the context of this module that the second part of the XIP simulation occurs, both in the `simulate` and the `apply` methods of `XipRule`. This is the simulation of the actual application of the rules, as opposed to the simulation of the matching performed in the Processed Corpus Parser.

Despite the fact that only a single search algorithm was implemented, the module was built in a way that allows for its easy extension. This was done through a *Strategy* design pattern. Figure 4.11 shows the class diagram for this pattern.

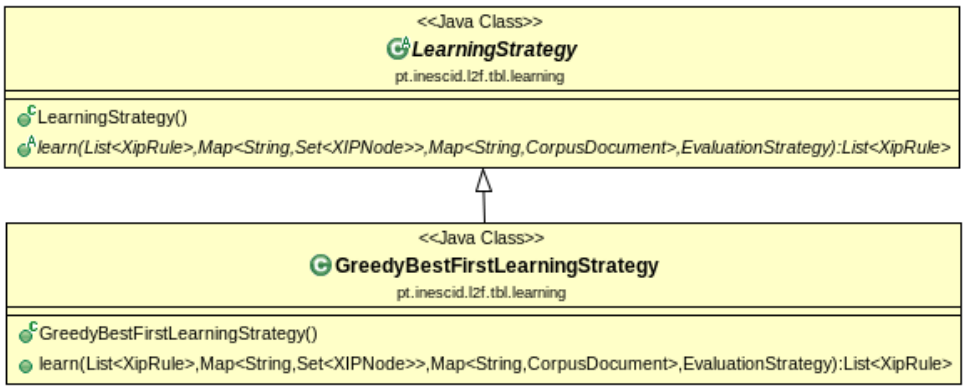

Figure 4.11: Learner Strategy Pattern class diagram

Class `LearningStrategy` is an abstract class that defines the interface that a new algorithm implementation must follow. Class `GreedyBestFirstStrategy` is an implementation of `LearningStrategy`, that implements a GBFS algorithm in its `learn` method. Thus, to implement a new algorithm, one needs only to create a concrete subclass of `LearningStrategy`, and write the specific `learn` method.

4.2.5 Evaluator

The last module in the TBL system is the Evaluator. This module is responsible for evaluating the results of applying a given XIP rule to the *corpus*.

Two different ways of evaluating the rules were implemented in this context:

- Word-by-Word - In this method, for each `XipNode` object that the rule being evaluated changed, the system goes through the reference word by word. This is done by iterating through the `Word` nodes that resulted from parsing the reference. The value of the `low` and `hig` attributes of each `Word` object are compared with the values of the `start` and `end` attributes of the `XipNode`. If they match, the verb was found in the reference. Then, if the `XipNode` still has the feature corresponding to the ViPEr class in the `viper` attribute of the `Word`, then the rule is considered as having

been applied correctly to that verb. Otherwise, that rule is considered as being wrongly applied to that verb. If a verb is wrongly classified, it is marked as such, and it is no longer considered in evaluating the following rules. This is done to prevent an erroneous rule to introduce error in the evaluation of following rules.

- Word-by-Word in Sentence - This method was implemented mostly for performance reasons. Instead of going through every word of the reference to search for a verb, it considers the number of the sentence in which the verb occurs, and in that sentence, searches word by word until it finds the verb in question. This is naturally faster than always searching word by word from the start of the reference for every verb. This method requires that the number of sentences in the *corpus* and the reference match. If they do not, then the method is unable to evaluate any verb in mismatched sentences. A file containing a list of every verb that has not been found in a given sentence is then created, in case debugging the sentence splitting process proves to be necessary. This method was, in fact, invaluable in the development of the project, by helping to solve some problems with the sentence splitting task in the STRING chain.

Like the Learner module, the Evaluator was also implemented through the use of a *Strategy* design pattern. Figure 4.12 presents the class diagram for this.

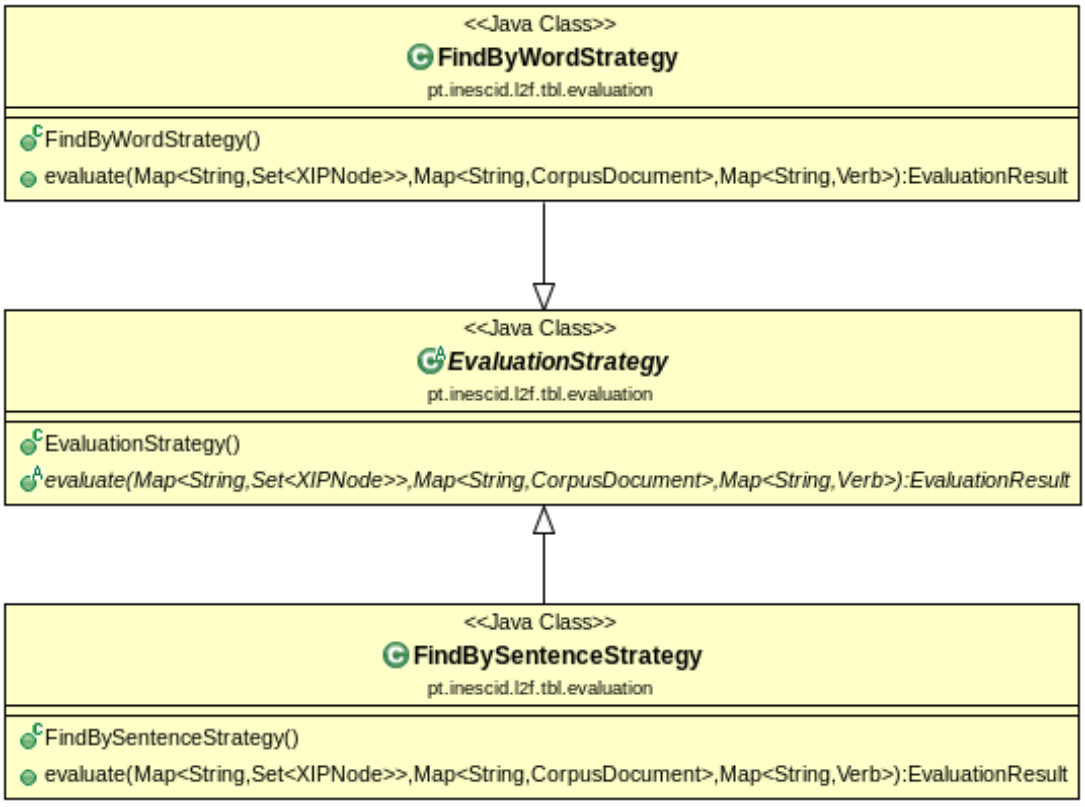

Figure 4.12: Evaluator Strategy Pattern class diagram

The `FindByWordStrategy` object implements the Word-by-Word method, while the `FindBySentenceStrategy` object implements the Word-by-Word in Sentence method. They are both subclasses of the `EvaluationStrategy` abstract class, which defines the interface that an implementation must follow. Once again, this was done to allow for a simple implementation of new ways of performing evaluation.

This concludes the description of the TBL module. At the time of writing, the behaviour of the XIP simulator has a very high degree of accuracy. There are however a few cases where it is less than accurate. This is not due to implementation errors, but due to the way STRING works and how the *corpus* is used after being processed by XIP.

Consider the sentence "- Olha, minha menina, não me interessa nada saber há quanto tempo não tinhas uma miséria de uma nota." (Look, missie, I really don't care how long ago you hadn't had a miserable grade). When this sentence is first processed by XIP, the verb *interessar* ('to care') is part of a CDIR dependency. However, XIP has a set of post-processing rules that apply after the verb sense disambiguation step. Refer to Example 15.

```
// Example (05):  Agrada-me que o Pedro faça isso.

// Example (05):  Urge que o Pedro faça isso.

if ( ^CDIR[post,sentential,interrogat:~](#1[markviper,01I],#2) ||

^CDIR[post,sentential,interrogat:~](#1[markviper,04],#2) ||

^CDIR[post,sentential,interrogat:~]#1[markviper,05],#2) ||

^CDIR[post,sentential,interrogat:~](#1[markviper,05H],#2) )

SUBJ[completiv=+](#1,#2)
```

Example 15. Post-processing XIP rule

This rule was built for verb constructions that require a sentential subject, which is placed after the verb. These verb constructions are incorrectly parsed as CDIR, and these rules are used to correct this issue. This rule is applied to the example sentence, as the verb *interessar* is part of a CDIR dependency, and it is has been classified with the `05H` ViPEr class. This rule then replaces the CDIR dependency with a SUBJ dependency. This means that, when the simulator analyses this sentence, the verb is already part of the SUBJ dependency, and therefore it does not match any of the CDIR rules that it would have matched in XIP.

Unfortunately, these cases are not easily measurable. The TBL module, before reordering the rules, correctly disambiguates a total of 4,571 verbs, while STRING using the same unordered rules correctly disambiguates 4,556 verbs. While it was expected that STRING would disambiguate more verbs than

TBL, given the previous example, what happens is the opposite, with TBL correctly disambiguating 15 more verbs than STRING. This means that other issues also cause some verbs to be disambiguated by the simulator when it should not do so. The process of identifying these differences is very time-consuming, requiring manual analysis of the *corpus* while also running the system in debug mode. Since the difference is so small (15 verbs out of a total of 12,494 ambiguous verbs, or about 0.12%), identifying all those issues was not considered an high priority task.

5 Evaluation

In the previous chapter, the proposal for a new approach to verb sense disambiguation rule generation and for a machine-learning-based way to improve these rules were presented. In this chapter, the evaluation scenarios considered for both proposals, followed by the results obtained in each case, are presented. At the end of each section, an analysis of the results achieved with these approaches is made, and some conclusions that can be drawn from these experiments are presented.

5.1 Evaluation Methodology

5.1.1 Evaluation Corpus

To evaluate the performance of the proposals presented in the previous chapter, the PAROLE *corpus* (Nascimento et al. 1998) was used. This *corpus* contains around 250,000 words, and each verb has been manually annotated with its ViPEr class and reviewed by linguists. It is composed by a collection of full texts from a variety of different sources, such as journalistic text from newspapers, or prose from novels.

At the time of writing, the *corpus* contains a total of 38,927 verbs, of which 21,444 (55.10%) are full verbs. Table 5.1 presents some figures regarding the verbs of this *corpus*. Note that the *pro-verbo* column indicates the number of verbs that fill the *pro-verbo* role (or *verbo-vicário* in Brazilian Portuguese grammatical terminology). These verbs (typically, the verb *fazer* 'do') are used to replace the predicate of a sentence, sometimes even including the complements. In a way similar to that of pronouns replacing nouns, e.g. *O Pedro leu o jornal mas o João não fez isso* ('Pedro read the newspaper, but João didn't do that').

Not all of the 21,444 verbs are ambiguous. Table 5.2 presents the total number of ambiguous verbs present in the *corpus*, as well as the number of verbs that are considered for the results of the disambiguation step in STRING. To be considered for the results of disambiguation, a verb has to match three criteria: the verb must not be tagged with the wrong POS by a previous module of the chain; the verb must not have been a different lexical unit (such as a name) wrongly identified as a verb by a previous module; and the verb must be present in ViPEr. If they do not match these criteria they are ignored, since, otherwise, previous errors on the chain and the absence of the verbs in ViPEr would be reflected in the results.

Finally, Table 5.3 presents the verb distribution by number of meanings. Verbs that are not considered for disambiguation results were ignored in this context.

	Total	**Full Verbs**	**Auxiliary Verbs**	**Fixed Expressions**	***Pro-verbo***
Count	38,920	21,444	16,780	485	211
%	100	55.09	43.11	1.25	0.54

Table 5.1: Verb occurrences in the PAROLE Corpus

	Total	**Processed**	**Wrong POS**	**Wrong Lemma**	**Not present in ViPEr**
Count	13,954	12,494	171	17	1272
%	100	89.54	1.23	0.12	9.12

Table 5.2: Verbs considered for disambiguation results in the PAROLE Corpus

Meanings	**Count**	**%**
1	8,950	41.74
2	6,635	30.94
3	2,772	12.93
4	1,689	7.88
5	494	2.3
6	527	2.46
7	173	0.81
8	204	0.95
Total	21,444	100
Total Ambiguous	12,494	58.26

Table 5.3: Full Verbs distribution by number of potential meanings in the PAROLE Corpus

Since there is no other manually annotated *corpus* that contains ViPEr data that is large enough to provide a significant sample size, the training *corpus* must be the same used to evaluate the results of the TBL system. However, since this is a machine-learning-based system, the results are adapted to the training *corpus*. Thus, to avoid bias as a consequence of this, a 10-fold cross-validation process was used to evaluate the results of the TBL system. Table 5.4 presents the average number of verbs and standard deviation for each of the ten folds of the partitioned *corpus*.

The training *corpus* displays a fairly even distribution of verbs, presenting a 6.39% standard deviation. However, this is not enough to guarantee that the learning process is evenly weighted across all the partitions of the *corpus*, since the distribution of verbs by their ViPEr class varies a great deal across the folds.

	Average Instances	Standard Deviation
Full Verbs	2,144.4	135.08
Total Verbs	3,892.0	248.63

Table 5.4: Verb Statistics from the partitioned PAROLE Corpus

5.1.2 Measures

The primary goal of the evaluation process is to measure the correctness of the results produced by the system against a reference value, that is, how many verbs were correctly classified by the system, among all the verbs that had been hand-tagged in the *corpus*. This fits the definition of *accuracy*. The formula used for accuracy is presented below, with n_c being the number of correctly disambiguated instances and N the total number of ambiguous verb instances present in the evaluation *corpus*.

$$Accuracy = \frac{n_c}{N}$$

The secondary goal of the evaluation process is to measure how the new rule generator performs in comparison to the one currently in place in the STRING chain, and how the rules generated affect the performance of STRING. This will be measured in terms of the execution time of the rule generators, and in terms of the execution time of STRING using a 5,000 word *corpus* with both sets of rules.

5.2 Baseline

Generally, a baseline is a starting point for comparison of a system's performance. In the context of this project, it is necessary to evaluate two different scenarios, thus requiring a baseline for each scenario:

- The performance of the STRING chain with the rules generated by the new rule-generation module as opposed to the rules generated by the module currently in place. For this scenario, the baseline considered is the result of processing the evaluation *corpus* with the STRING chain using the set of rules generated by the module currently in place. This should provide a direct comparison measure of the performance of verb sense disambiguation in STRING before and after the implementation of the new rule-generation module.
- The performance of the rules themselves. For this scenario, the baseline used is the result of processing the evaluation *corpus* with the STRING chain, using the rules generated by the module currently in place, without the use of the MFS or the Naive-Bayes classifiers used by STRING after

XIP. This should provide a comparison on how the former and new rules perform by themselves, without considering the influence of these classifiers.

For evaluating the TBL algorithm, the baseline considered will be the best result for each of the previously mentioned scenarios.

5.3 Rule Generation Evaluation

In this section, the results of evaluating the system using the rules produced by the rule generator currently in place will be compared to the results of evaluating the system with the rules produced by the new rule generator in three different scenarios:

- Without the use of any of the classifiers used by the STRING chain - This should provide a direct comparison of the rules;
- With the MFS classifier used by STRING - Since MFS produces highly accurate results, this scenario is used to evaluate how the rules interact with this classifier;
- With both the MFS and the Naive-Bayes classifiers used by STRING - This presents the results of STRING in a live environment.

5.3.1 No classifiers

This subsection presents the results of evaluating the rules without the use of later classifiers.

At the time of writing, the former rule generator produces a total of 1,487 XIP disambiguation rules. Table 5.5 presents the results obtained using the original set of rules.

Meanings	Instances	Correctly Disamb.	Fully Disamb.	Wrongly Disamb.	Not Disamb.	Ambiguous Left
8	204	51	30	99	54	75
7	173	92	61	45	36	67
6	527	115	77	12	400	438
5	494	141	91	39	314	364
4	1,689	238	109	55	1,396	1,525
3	2,772	681	449	177	1,914	2,146
2	6,635	641	641	63	5,931	5,931
Total	12,494	1,959	1,458	490	10,045	10,546
%	100	15.68	11.67	3.92	80.4	84.41

Table 5.5: Results when using the former rules (no post-XIP classifiers)

The results in Table 5.5 show that the former set of rules only disambiguated a small subset (2,449 verbs) of the ambiguous verbs in the *corpus*. Out of these, 1,959 verbs were correctly disambiguated, meaning that these rules have a 15.68% accuracy. In addition, 1,458 out of the 1,959 are fully disambiguated, that is, just a single ViPEr class remained after the disambiguation process, while 490 do not contain the correct ViPEr class.

The new rule generator produces a total of 10,691 rules. Table 5.6 presents the results obtained using this new set of rules. These results show an increase in correctly disambiguated instances from 1,959 (15.68%) to 4,556 (36.47%), meaning the accuracy improved by 20.79%. In addition, the number of fully disambiguated verbs increased from 1,458 (11.67%) to 2,541 (20.34%), a difference of 8.67% . Unfortunately, the number of wrongly disambiguated verbs also increased, from 490 (3.92%) to 1,125 (9%), or 5.08%.

Meanings	Instances	Correctly Disamb.	Fully Disamb.	Wrongly Disamb.	Not Disamb.	Ambiguous Left
8	204	68	8	72	64	124
7	173	154	34	19	0	120
6	527	336	72	111	80	344
5	494	275	95	71	148	328
4	1,689	899	208	219	571	1,262
3	2,772	1,243	543	314	1,215	1,915
2	6,635	1,581	1,581	319	4,735	4,735
Total	12,494	4,556	2,541	1,125	6,813	8,828
%	100	36.47	20.34	9	54.53	70.66

Table 5.6: Results when using the new rules (no post-XIP classifiers)

Figure 5.1 presents a chart with the comparison of these results.

The new rules show a clear improvement over the previous rules, increasing both the number of partially and fully disambiguated verbs. In addition, while the new rules show an higher number of wrongly disambiguated verbs, the ratio is actually lower (1,125 out of a total of 5,681 disambiguated verbs, hence 19.8%) when compared to the previous rules (490 out of a total of 2,449 disambiguated verbs, that is 20.01%).

5.3.2 MFS classifier

In the previous subsection, the results of evaluating both sets of rules without the use of classifiers were presented and discussed. The new set of rules is promising, showing an overall improvement in accuracy of 20.07%. In this section, the rules are evaluated with the addition of the MFS classifier used

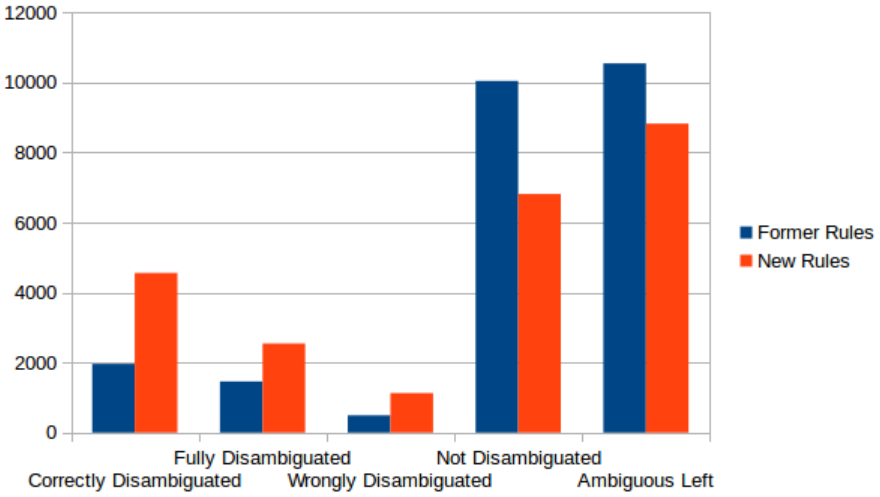

Figure 5.1: Rule results comparison (no post-XIP classifiers)

by the STRING chain. This should allow for an overview of how both sets of rules interact with this classifier.

Table 5.7 presents the results of processing the evaluation *corpus* with the former set of rules and the MFS classifier. Note that the **Fully Disambiguated**, **Not Disambiguated** and **Ambiguous Left** columns were omitted, since, as mentioned in the previous subsection, the MFS classifier fully disambiguates every verb, thus these data are not relevant. As expected, all ambiguous verbs are fully disambiguated, since the MFS classifier simply assigns to each ambiguous verb its most frequent sense. There is a 65.33% increase in accuracy, going from 15.68% to 81.01%. Note that the MFS is computed using the full *corpus*, meaning the results are biased. Unfortunately, there is no other annotated *corpus* large enough to compute the MFS.

Meanings	**Instances**	**Correctly Disamb.**	**Wrongly Disamb.**
8	204	66	138
7	173	96	77
6	527	362	165
5	494	336	158
4	1,689	1,336	353
3	2,772	2,172	600
2	6,635	5,762	873
Total	1,2494	10,130	2,364
%	100	81.08	18.92

Table 5.7: Results when using the former rules (with the MFS classifier)

Table 5.8 shows the results of processing the evaluation *corpus* with the new set of rules and the MFS classifier. As before, the accuracy increased by a large margin, going from 36.47% to 77.64%, a 41.17% increase.

Meanings	Instances	Correctly Disamb.	Wrongly Disamb.
8	204	70	134
7	173	94	79
6	527	276	251
5	494	328	166
4	1,689	1,220	469
3	2,771	2,080	691
2	6,635	5,632	1,004
Total	12,494	9,700	2,794
%	100	77.64	22.36

Table 5.8: Results when using the new rules (with the MFS classifier)

Figure 5.2 presents a chart comparing both sets of results. The former rules alongside the MFS classifier perform 3.37% better than the new rules, disambiguating 10,130 verbs correctly, as opposed to 9,700 verbs for the new set of rules.

An additional 430 verbs were wrongly disambiguated by the new set of rules. This is due to a subset of the verbs that were wrongly classified by the new rules also being reduced to a single ViPEr class, meaning they are no longer considered ambiguous. As a consequence, they are not considered by the MFS classifier. In addition, despite the new rules fully disambiguating more verbs, they do not overlap completely with those fully disambiguated by the former rules. This causes the MFS classifier to apply to verbs it would not apply to with the former rules, where unfortunately the MFS is not the correct choice.

These results indicate that, since the MFS classifier has such a high accuracy, it is more important to have a low error count than a high success count.

5.3.3 MFS and Naive-Bayes classifiers

The previous subsection presented the results of evaluating both sets of rules with the additional use of the MFS classifier used by the STRING chain. In this subsection, the rules will be evaluated with the addition of the Naive-Bayes classifier used by the STRING chain, aiming at presenting the results produced by STRING in a live environment.

Table 5.9 presents the results of processing the evaluation *corpus* with the former set of rules, and both the MFS and the Naive-Bayes classifiers. Note that the **Fully Disambiguated**, **Not Disambiguated**

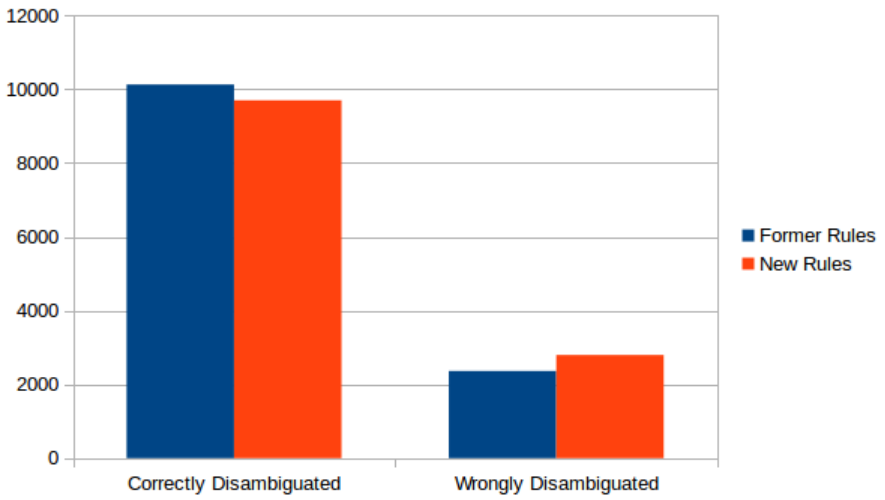

Figure 5.2: Rule results comparison (with the MFS classifier)

and **Ambiguous Left** columns were omitted, since, as mentioned in the previous subsection, the MFS classifier fully disambiguates every verb, thus these data are not relevant.

The Naive-Bayes classifier, when combined with the former rules and the MFS classifier, improves the results. An additional 103 verb instances are correctly classified, which corresponds to a 0.82% improvement.

Meanings	Instances	Correctly Disamb.	Wrongly Disamb.
8	204	66	138
7	173	96	77
6	527	301	226
5	494	336	158
4	1,689	1,327	362
3	2,772	2,237	535
2	6,635	5,870	765
Total	12,494	10,233	2,261
%	100	81.9	18.1

Table 5.9: Results when using the former rules (with the MFS and the Naive-Bayes classifiers)

Table 5.9 presents the results of processing the evaluation *corpus* with the new set of rules, and both the MFS and the Naive-Bayes classifiers. As in the previous case, the results slightly improved with the addition of the Naive-Bayes classifier. In this case, an additional 164 verb instances were correctly disambiguated (1.31%). The additional improvement is caused by the fact that the new rules together with the MFS classifier wrongly classify more instances of the verbs that are disambiguated with the Naive-Bayes classifier.

Meanings	Instances	Correctly Disamb.	Wrongly Disamb.
8	204	70	134
7	173	93	80
6	527	248	279
5	494	328	166
4	1,689	1,282	407
3	2,772	2,098	673
2	6,635	5,745	891
Total	12,494	9,864	2,630
%	100	78.95	21.05

Table 5.10: Results when using the new rules (with the MFS and the Naive-Bayes classifiers)

Finally, Figure 5.3 presents a comparison of the results. At the time of writing, STRING performs better in a live environment with the former set of rules, rather than the rules produced by the new rule generation module.

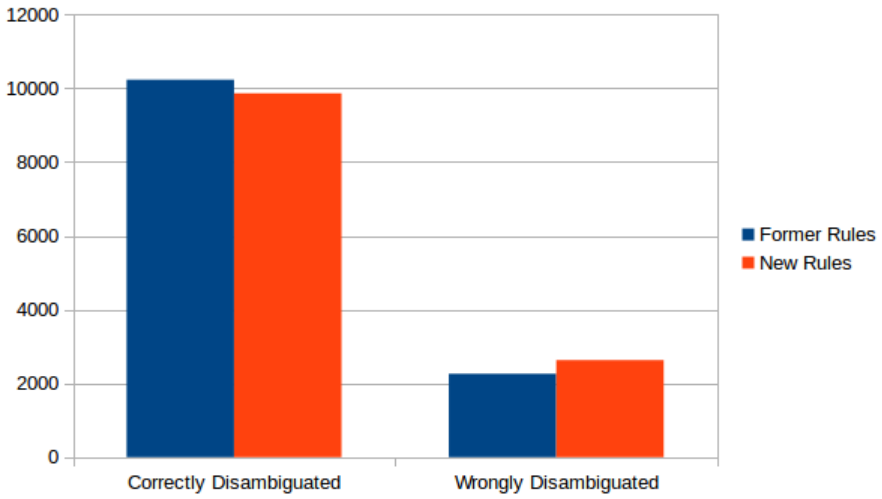

Figure 5.3: Rule results comparison (MFS+NB)

5.3.4 Performance

In addition to the impact of the rules in the results of verb sense disambiguation, it is also necessary to measure the impact of the rules in the runtime of the STRING chain.

To perform this measurement, a different *corpus* was used. This is a *corpus* containing 5,000 sentences collected from the CETEMPúblico *corpus*. Table 5.11 shows the runtime of STRING with each set of rules, along with a comparison. The new rules have a significant negative impact on the overall

performance of STRING when compared to the rules currently used, increasing the runtime with the profiling *corpus* by 18.07%. This is to be expected, since the number of rules increased about 7.2 times, from 1,487 to 10,691.

	Former rules	New rules	Difference
Runtime (s)	548	647	+99 (18.07%)

Table 5.11: Performance comparison

5.4 TBL Evaluation

While the previous sections focused on evaluating the performance of the disambiguation rule generators, this section will now present the evaluation of the STRING chain using the rules reordered by the TBL system.

As in the previous section, two evaluation scenarios will be described. The first scenario focuses only on evaluating the impact of the reordering of the disambiguation rules performed by the TBL algorithm. The second scenario evaluates the results of TBL on the performance of the live STRING chain, by including both classifiers.

5.4.1 No classifiers

Table 5.12 presents the results of evaluating the performance of STRING with the rules reordered by TBL, without considering the MFS or Naive-Bayes classifiers. The results presented are the average of the results of evaluating each of the ten folds of the *corpus*. In addition, it presents a comparison of these results with the baseline considered for the evaluation of the algorithm, which in this case, are the results of the unordered rules produced by the new rule generation module.

	Instances	Correctly Disamb.	Fully Disamb.	Wrongly Disamb.	Not Disamb.	Ambiguous Left
Average	1,249.4	468.6	269.7	99.7	681.1	880
Total	12,494	4,686	2,697	997	6,811	8,800
Baseline	12,494	4,556	2,541	1,125	6,813	8,828
		+130 (1.04%)	+156 (1.25%)	-128 (1.03%)	-2 (0.02%)	-28 (0.22%)

Table 5.12: Results obtained using TBL for rule reordering (no post-XIP classifiers)

TBL shows a direct improvement over the unordered rules in every measure. It correctly disambiguates an additional 130 verbs, while fully disambiguating 156 more and reducing the wrongly dis-

ambiguated verbs by 128. This shows that, while it only disambiguates 2 verbs that previously were not, it improves the overall quality of the disambiguation.

5.4.2 MFS and Naive-Bayes classifiers

Table 5.13 presents the results of evaluating the performance of the rules as reordered by TBL, while considering the MFS and Naive-Bayes classifiers for an overview of its impact on the performance of the live STRING chain. It also presents a comparison with the results of both the sets of rules considered in Section 5.3. Once more, since the MFS classifier is used, the “Fully Disambiguated", “Not Disambiguated" and “Ambiguous Left" data were omitted.

	Instances	**Correctly Disamb.**	**Wrongly Disamb.**
Average	1,249.4	996.7	252.7
Total	12,494	9967	2527
New Rules	12,494	9,864 +103 (0.82%)	2,630 -103 (0.82%)
Baseline (Former Rules)	12,494	10,233 -266 (2.13%)	2,261 +266 (2.13%)

Table 5.13: TBL (MFS + NB) results

As expected, based on the previous results, TBL shows an improvement over the unordered set of rules produced by the new rule generator, improving the accuracy of the system by 0.82%, from 78.95% to 79.77%. However, it still comes up 2.13% short of the baseline.

5.4.3 Performance

There are two important points to evaluate when it comes to the performance of TBL: the runtime the algorithm takes to reorder the rules, and the runtime of STRING with the set of rules ordered by TBL.

To obtain the runtime of the algorithm, the average of each cross-validation iteration runtime was taken. On average, TBL took 7 hours, 24 minutes and 47 seconds to finish. This may seem like a long time, but the learning process is performed offline, not bearing any direct impact on the runtime of STRING.

To measure the actual impact of the performance of STRING, the 5,000 word profiling *corpus* was processed with each of the sets of rules produced by TBL in each iteration of the cross-validation process, and the runtime of each was measured. Table 5.14 shows the average runtime and compares it to the average runtime of the rule sets evaluated in Section 5.3.

	Runtime (s)	
Baseline (Former Rules)	548	1,487
New Rules	647	10,691
TBL Rules	647	10,691

Table 5.14: Performance comparison

The rules reordered by TBL have the same performance impact as the unordered rules in relation to the former rules, increasing the runtime of STRING by around 99 seconds.

5.5 Discussion

In the previous sections, the scenarios used to evaluate the modules developed throughout this project were presented.

In a vacuum, the new rules proved to be more accurate than the former rules, showing an improvement of 20.07%. However, when used alongside the MFS classifier, the new rules performed worse than the former rules, attaining an accuracy of just 78.95% as opposed to the 81.9% accuracy obtained with the former rules. This is due to the fact that the new rules have a higher number of wrongly disambiguated verbs (even if the ratio is lower when compared to the former rules). The MFS classifier only considers verbs that are still ambiguous after the rule-based disambiguation, and the higher number of fully by wrongly disambiguated instances prevents the MFS to be applied to some of these cases.

The Naive-Bayes classifier proved to remain relevant, increasing the number of correctly disambiguated instances in both cases.

In addition, the new rules increased the runtime of STRING by 99 seconds, meaning that, at the time of writing, additional work is needed, in order to improve the rules enough to justify the decrease in performance they cause.

The TBL algorithm has improved the results of the new rules by 0.82%, although this is not enough to overtake the performance of the former rules. The learning process is long, but since it is meant to be performed only when new rules are generated, and it is an offline process, its runtime does not have any impact on the performance of the STRING chain.

An additional experiment was made to follow up on these results. With the new debugging tools, the accuracy of each declarative rule was checked, and the rule was removed if its accuracy was below 60%. A total of 9 rules were thus removed. Afterwards, both scenarios used to evaluate the TBL module

were repeated, but with this new set of declarative rules. Table 5.15 presents the results of evaluating the module excluding the MFS and Naive-Bayes classifiers.

	Instances	Correctly Disamb.	Fully Disamb.	Wrongly Disamb.	Not Disamb.	Ambiguous Left
All Declarative Rules	12,494	4,686	2,697	997	6,811	8,800
Excluded Low Performing Rules	12,494	4,679	2,699	981	6,834	8,813
		-7 (0.06%)	+2 (0.02%)	-16 (0.13%)	+23 (0.18%)	+13 (0.1%)

Table 5.15: TBL results - Excluded poorly performing declarative rules

Removing the poorly performing rules results in a slightly lower number of correctly disambiguated verbs (-0.18%), but the number of fully disambiguated verbs increased by 0.02% and the number of wrongly disambiguated verbs decreased by 0.13%. This shows that, while removing the poorly performing rules does increase the quality of the disambiguation, their impact is actually fairly low. Table 5.16 shows how this reflects on the quality of the disambiguation in STRING.

	Instances	Correctly Disamb.	Wrongly Disamb.
All Declarative Rules	12,494	9,967	2,527
Excluded Low Performing Rules	12,494	9,979	2,515
		+12 (0.1%)	-12 (0.1%)

Table 5.16: TBL (MFS + NB) results - Excluded poorly performing declarative rules

Once again, the impact of removing these declarative rules was fairly low, with only a 0.1% increase. This means that, to improve the results of the combination of rules and MFS, it is not enough to exclude low accuracy rules. Instead, this result indicates that taking a look at rules that apply to a large number of verbs may be more important than solely looking at poorly performing rules.

Despite the overall results of the system not improving, the new rule generation module is much more flexible than the one currently in place. In addition, TBL provides extra tools that help debugging the declarative rules in order to make it easier to improve the overall accuracy of the disambiguation rules. This means that the tools are now in place to improve the rules, hopefully leading to an eventual increase in the accuracy of the VSD process.

6 Conclusions

6.1 Conclusions

This dissertation addressed the task of Verb Sense Disambiguation in the context of STRING, a NLP chain. This is the task of, given an ambiguous verb, selecting the correct verb sense according to its context. The STRING chain was described, along with ViPEr, the European Portuguese Lexicon-Grammar of Verbs. In addition, the current approaches to VSD used in STRING were described.

A new solution for generating rules to perform rule-based disambiguation was developed. This solution uses the information provided by ViPEr, along with a set of declarative rules developed alongside two experts to generate a set of XIP disambiguation rules. All the knowledge required for the generation process is contained in these rules, making this approach highly flexible, as opposed to the rule generation solution previously used, where all the knowledge was built into the code of the generator.

Additionally, a new machine-learning-based solution was implemented, specifically the TBL algorithm. Unlike the previous machine-learning work that was developed for STRING, this one does not use machine-learning to perform disambiguation, but rather to improve the rule-based disambiguation. This algorithm considers the context (the ViPEr classes of a given verb and the XIP dependencies associated to it), to perform transformations (removing a ViPEr class by applying a XIP disambiguation rule). Through this, the algorithm evaluates each transformation, and reorders them according to their results.

To evaluate the performance of the new rule generation module, three scenarios were considered:

- The first scenario used only the rules for disambiguation, with the results of the rules generated by the previous module used as the baseline. This provided a direct comparison between both sets of rules, with the new rules improving the system on every aspect.
- The second scenario added the MFS classifier. With this addition however, the accuracy of the new rules was overtaken by the accuracy of the previous rules. This was due to the fact that the new rules wrongly disambiguate more verbs, despite the wrong-to-correct ratio being smaller. This means that, when using the MFS classifier, it is more important for the rules to avoid errors than to increase the number of successes.

- The third and final scenario added the Naive-Bayes classifier, to evaluate the accuracy of the live STRING chain. The classifier increased the accuracy of the system in both cases by similar amounts. As it stands, the previous rules proved to be the best solution for disambiguation in STRING (by a margin of 2.13%). However, it is important to note that they are static, while the new rules still have room for improvement.

The TBL-based rule reordering system was also evaluated against the first and third scenarios. The first was used to determine if the reordering of the disambiguation rules performed by TBL improved the results, and the second one to measure its impact on the overall performance of STRING. TBL did improve the results of rule-based disambiguation, although not enough to surpass the previous rules. However, once the new set of rules is further improved, TBL is a promising alternative to provide an additional increase in disambiguation accuracy.

6.2 Contributions

While the work described in this thesis did not directly improve the results of the verb sense disambiguation task, it made some significant contributions to the STRING system as a whole.

First of all, it contributed with a highly flexible solution to disambiguation rule generation which is now simpler to maintain. This should allow for much easier tweaking and improving of the rules in the future.

A TBL implementation was developed for use with XIP and the declarative rules previously mentioned. Alongside this, a new set of debugging tools was implemented which helps in identifying declarative rules with poor performance. In addition, these tools also allow for the identification of sentence segmentation problems in the STRING chain. Indeed, a significant number of segmentation problems was identified and corrected during this project, which also contributed to improve the system as a whole.

Furthermore, during the course of this project, *corpora* for three verbs, v.g. *esconder* (to hide), *esperar* (to wait / to hope) and *interessar* (to take an interest in / to be interesting to), were manually annotated for future use.

6.3 Future Work

Much work remains to be done when it comes to VSD in STRING. Though the solutions implemented during the course of this project did not directly improve the results of STRING, a new framework has

been produced, which if explored to the fullest, by integrating new linguistic knowledge, will eventually enable the system to achieve better results.

The declarative rules can now be reviewed and tweaked, for there is a set of tools that provide clear metrics on how each of them performs.

The TBL implementation currently uses a GBFS search algorithm for the learning process. However GBFS is not an optimal search algorithm. This means that there is still room to experiment with heuristic search algorithms, which are optimal, and thus are expected to produce better results. Indeed, the module was implemented with extensibility in mind, meaning only the algorithm itself and the required data structures would need to be implemented, with barely any change in the remaining module.

When it comes to machine-learning, there are still some experiments to be made. First of all, it would be interesting to take into account the current results of VSD in STRING to decide which additional verbs may be disambiguated through machine-learning, thus, which verbs should have *corpora* annotated. Machine-learning can also be used to perform pre-classification on *corpora* to annotate, helping reduce the potential workload for the annotator. Another interesting experiment would be to use machine-learning to disambiguate between some verb senses, and leaving the rest to be disambiguated by rules. Finally, a meta-classifier can be used to determine which verbs should be disambiguated by rules and which should be disambiguated by machine-learning.

One of the main problems the STRING team currently faces is the lack of annotated *corpora*. As already mentioned, the pre-classification step could be used to aid annotators in this work. In addition, for well-performing verbs, the systems currently in place could be used to directly annotate *corpora*. While these *corpora* would still need revision afterwards, the workload should be substantially reduced.

Currently, STRING performs its NLP tasks sequentially, meaning it performs lexical analysis, followed by syntactical and finally semantic analysis. However, human beings do not always process natural language in this way. It would be interesting to perform processing in several iterations, instead of doing so in a single run.

To conclude, the contributions of this project presented above helped improve the STRING chain as a whole. New tools are now in place to improve the quality of this task, both through improving the declarative rules upon which this work was based, as well as through trying some of the methods suggested above.

Bibliography

Aït-Mokhtar, S., J. P. Chanod, and C. Roux (2002, June). Robustness beyond shallowness: Incremental deep parsing. *Nat. Lang. Eng. 8*(3), 121–144.

Almeida, H. (2016, November). Suffix Identification in Portuguese using Transducers. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa.

Baptista, J. (2010). *Les Tables. La grammaire du français par le menu. Mélanges en hommage à Christian Leclère*, Chapter Verba dicendi: a structure looking for verbs, pp. 11–20. Number 6. Louvain-la-Neuve: CENTAL/Presses Universitaires de Louvain.

Baptista, J. (2012, September). ViPEr: A Lexicon-Grammar of European Portuguese Verbs. In *Proceedings of the 31st International Conference on Lexis and Grammar*, pp. 10–16.

Brill, E. (1995, December). Transformation-Based Error-Driven Learning and Natural Language Processing: A Case Study in Part-of-Speech Tagging. *Computational Linguistics 21*(4), 543–566.

Daumé, H. (2004, August). Notes on CG and LM-BFGS Optimization of Logistic Regression. Paper available at `http://pub.hal3.name#daume04cg-bfgs`, implementation available at `http://hal3.name/megam/`.

Diniz, C. (2010, October). Um conversor baseado em regras de transformação declarativas. Master's thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa.

Diniz, C., N. Mamede, and J. Pereira (2010, September). RuDriCo2 - a faster disambiguator and segmentation modifier. In *II Simpósio de Informática (INForum)*, pp. 573–584.

Mamede, N., J. Baptista, C. Diniz, and V. Cabarrão (2012). STRING: A Hybrid Statistical and Rule-Based Natural Language Processing Chain for Portuguese. In *Proceedings of the 12th International Conference on Computational Processing of the Portuguese Language - Demo sessions* `http://www.propor2012.org/demos/DemoSTRING.pdf`, PROPOR 2012.

Nascimento, M., P. Marrafa, L. Pereira, R. Ribeiro, R. Veloso, and L. Wittmann (1998). LE-PAROLE - Do corpus à modelização da informação lexical num sistema-multifunção. In *XIII Encontro Nacional da Associação Portuguesa de Linguística*, pp. 115–134. Lisboa: APL/Colibri.

Ribeiro, R. Anotação Morfossintáctica Desambiguada do Português. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa.

Rodrigues, D. (2007, November). Uma evolução do sistema ShRep. Optimização, interface gráfica e integração de mais duas ferramentas. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa.

Russell, S. J. and P. Norvig (2002, December). *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall.

Santos, D. and P. Rocha (2001). Evaluating CETEMPúblico, a Free Resource for Portuguese. In *Proceedings of the 39th Annual Meeting of ACL*, ACL '01, Stroudsburg, PA, USA, pp. 450–457. Association for Computational Linguistics.

Suíssas, G. (2014, November). Verb Sense Classification. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa.

Travanca, T. (2013, June). Verb Sense Disambiguation. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa.

Vicente, A. (2013, June). LexMan: um Segmentador e Analisador Morfológico com Transdutores. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa.