Proyecto de Grado

Integración de Herramientas para Procesamiento de Lenguaje Natural

Matías Laíno

3 de septiembre de 2015

Instituto de Computación - Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tutores:
Guillermo Moncecchi
Juan José Prada

Resumen

El Grupo de Procesamiento del Lenguaje Natural del Instituto de Computación de la Universidad de la República utiliza varias herramientas para realizar tareas frecuentes como etiquetado gramatical y análisis sintáctico. Las herramientas son en su mayoría desarrolladas por terceros, esto tiene como consecuencia que dos programas de la misma familia de herramientas (por ejemplo, dos analizadores sintácticos) aplican diferentes enfoques en su operativa. Difieren en aspectos como los parámetros de entrada (cantidad, formato, tipos), hasta a como es la ejecución del software en sí (nativa, sobre máquina virtual, sobre intérprete, etc.). Estas diferencias hacen que su empleo y explotación resulten más dificultosas de lo que deberían, no permitiendo a los usuarios enfocarse completamente en la tarea a resolver.

En este proyecto se propone el diseño e implementación de una plataforma que solucione esta problemática.

El presente informe comenzará por introducir al lector en la temática del PLN, proveyendo una reseña histórica de la disciplina, continuará relevando el estado del arte del área, detallando las herramientas más utilizadas y, finalmente, concluirá con la descripción del diseño e implementación de la biblioteca.

Índice

1. Introducción

El Procesamiento del Lenguaje Natural (PLN) es el área relacionada con la computación y la lingüística que se encarga de, básicamente, crear y utilizar técnicas que proveen mecanismos para que las computadoras sean capaces de entender el lenguaje natural. Esto puede ser utilizado para fines muy variados, desde la traducción automática sin intervención humana, hasta las asistentes virtuales de los teléfonos celulares. El Grupo de PLN del Instituto de Computación de la Universidad de la República (InCo) emplea para sus trabajos en el área diversas herramientas, algunas propias, otras hechas por terceros. Como se percibe que en el día a día el uso de las herramientas resulta en ocasiones más complicado de lo que debería ser, se propone el proyecto actual consistente en el diseño e implementación de una biblioteca que facilite su empleo por parte de los miembros del Grupo.

En este capítulo se realiza una presentación a la disciplina del PLN, dando un breve repaso a su historia, haciendo un listado de las tareas relacionadas más frecuentes, y finalizando con el planteo del problema y motivación del proyecto.

1.1. Un poco de historia

En 1949, Warren Weaver, coautor de ≪La Teoría Matemática de la Comunicación≫ ("*The Mathematical Theory of Communication*")[1] (libro seminal del campo de la teoría de la información), tras haber sido expuesto a la potencia y velocidad de las computadoras al haber presidido el Panel de Matemáticas Aplicadas en la Oficina De Investigación y Desarrollo Científico de los Estados Unidos durante la Segunda Guerra Mundial, escribió el memorando titulado ≪Traducción≫ ("*Translation*")[2]. En dicho documento, Weaver relata una historia en la cual un matemático, que había diseñado una técnica de descifrado, solicita a un colega que le facilite un mensaje codificado para ponerlo a prueba. Su colega le entrega un mensaje en turco encriptado en una columna de números de cinco dígitos. Al siguiente día el matemático retorna con el mensaje descifrado, pero confundido acerca del contenido: el matemático no estaba familiarizado con el idioma Turco y creía que su técnica había fallado, sin embargo el mensaje, notó el colega, estaba correctamente desencriptado.

Weaver propone en su memorando que un documento escrito en un determinado idioma podría ser considerado como si estuviera escrito en código. Visto de esta forma, el Alemán era inglés encriptado, y la traducción era desencriptación. El memorando de Warren Weaver se considera altamente influyente y estimuló el comienzo de la investigación en el área de traducción automática.

La siguiente breve historia del PLN está tomada de un estudio realizado por la Universidad de Birmingham[3] y de [4].

Los primeros esfuerzos se realizaron en la traducción del Alemán al inglés, dado que habían aún documentos de la guerra pendientes de traducción. Sin embargo, con el advenimiento de la Guerra Fría, esta tarea se vio obsoleta, y se pasó a la traducción entre ruso e inglés y francés.

Los primeros sistemas no tuvieron éxito, esto significó la hostilidad de las agencias de financiación. A pesar del fracaso, se realizaron avances, no había conocimiento heredado ni técnicas existentes en PLN, los primeros investigadores eran frecuentemente matemáticos que luchaban contra la maquinaria computacional primitiva, algunos eran bilingües convencidos de que su conocimiento en ambas lenguas les permitiría construir sistemas capaces de traducir textos técnicos. Sin embargo, la tarea resultó más compleja de lo que habían imaginado. Peor aún, aunque hablaban el lenguaje nativamente, les fue muy difícil codificar su conocimiento del lenguaje en un programa de computadora.

Se buscó ayuda en el campo de la Lingüística. Se volvió una tendencia para jóvenes investigadores de Lingüística el unirse a equipos de Traducción Automática. Sin embargo, no existían teorías lingüísticas apropiadas por ese entonces. Esto cambió en 1957 con la publicación de Syntactic Structures[5], por Noam Chomsky, quien revolucionó el campo de la lingüística e influenció casi todo el trabajo en PLN desde ese momento en adelante.

Pronto resultó obvio que era necesario preeditar los textos de entrada para resolver palabras ambiguas, las máquinas no podían hacerlo por ellas mismas. En 1958 el matemático y lingüista Israelí Bar-Hillel concluyó que la traducción completamente automática de alta calidad era imposible sin el conocimiento que aportaban los traductores humanos, y determinó que los métodos que utilizaban estaban condenados al fracaso[6].

En 1966, el Automatic Language Processing Advisory Committee (ALPAC) realizó una evaluación del avance de la investigación en PLN, determinando que la traducción automática se había vuelto más cara que la manual[7]. El resultado de este reporte trajo como consecuencia un cese mundial en la financiación a la investigación y una suerte de edad oscura para el área que duraría unos 14 años.

En los 80s hubo una revolución en PLN con la introducción de algoritmos de aprendizaje automático. A pesar de que los primeros algoritmos usados eran similares a los conjuntos de reglas gramaticales utilizados hasta el momento, se colocó énfasis en la investigación de modelos estadísticos. Estos modelos son, en general, mucho más robustos cuando son enfrentados con entradas no familiares, en especial entrada con errores.

1.2. ¿Qué es el Procesamiento del Lenguaje Natural?

> *Por Procesamiento del Lenguaje Natural, estamos hablando de técnicas computacionales que procesan lenguaje humano hablado y escrito. Esto es una definición inclusiva que abarca todo desde aplicaciones mundanas como conteo de palabras, hasta aplicaciones de punta como sistemas de respuesta de preguntas en la Web, o traducción automática en tiempo real. Lo que distingue a estas aplicaciones de procesamiento del lenguaje de otros sistemas de procesamiento de datos es el uso del conocimiento del lenguaje.*[8]

Considérese un sistema que debe entender y producir lenguaje natural para comunicarse con un ser humano. En primer lugar, el sistema debe poder ser

capaz de analizar una señal de audio, y recuperar la secuencia exacta de palabras que produjeron esa señal. A su vez, debe poder tomar una secuencia de palabras y poder generar una señal de audio que la persona pueda reconocer. Ambas tareas requieren conocimiento de *fonética* y *fonología*.

El sistema debe ser capaz de poder procesar variaciones de palabras individuales al resultar flexionadas, por ejemplo al ser conjugadas. Esto requiere conocimiento de *morfología*, el estudio de la estructura interna de las palabras.

Se requiere la capacidad de comprender que "¿Está abierta la puerta?" es una pregunta y "La puerta está abierta" es una afirmación. A su vez, debe poder ser capaz de producir una secuencia de palabras que tenga sentido y no responder, por ejemplo (*)"Abierta si puerta, está la". Definimos el conocimiento de como ordenar y agrupar las palabras como *sintaxis*.

Sin embargo, de nada sirve poder entender las palabras, cómo ordenarlas y agruparlas si el sistema no comprende la naturaleza de lo procesado, no podrá emitir una respuesta o realizar una acción solicitada. El conocimiento del significado de las palabras y su composición se conoce por *semántica*.

Sin resultar vital, puede ser deseable que, por ejemplo, el sistema sea capaz de entender y producir el uso de la amabilidad. Ante un pedido, podría responder sencillamente "no", o devolver algo mas amable como "me temo que no puedo hacer eso". Este conocimiento se llama *pragmática*, y se define como el estudio de algunos componentes de la relación entre el lenguaje y el contexto de uso[8].

Por lo antes expuesto, es posible dividir el estudio del lenguaje natural en cuatro niveles de análisis: morfológico, sintáctico, semántico y pragmático, pudiendo ser incluido el análisis de la fonética y fonología. Los cuatro niveles están ordenados de la forma descrita, por lo que comprenden una secuencia en la cual cada etapa toma como información de entrada la salida de la etapa anterior (con la obvia excepción de la primera etapa). Las definiciones dadas a continuación están basadas en las provistas en [8].

- **Análisis morfológico**: Consiste en detectar la relación entre las unidades mínimas que forman una palabra (morfemas), reconocerla y construir una representación estructurada. Una palabra es la combinación de una raíz y un conjunto de morfemas, los cuales son aplicados mediante flexión, derivación y composición.

  La flexión es un mecanismo de producción de palabras dentro de una misma clase. Por ejemplo, al flexionar la raíz "maestr-" con el morfema flexivo de género masculino, se obtiene la palabra "maestro", al flexionar con el morfema flexivo de género femenino y el morfema flexivo de número plural, se obtiene la palabra "maestras".

  La derivación es la combinación de una raíz con un afijo para generar una palabra de otra clase, o con otro significado. Ejemplo: estable (adjetivo, sufijo "-able") → estabilizar (verbo, sufijo "-izar") → estabilización (sustantivo, sufijo "-ación") → desestabilización (sustantivo, prefijo "des-", sufijo "-ación").

La composición es la unión de dos raíces. Ejemplo: medio + día = mediodía.

- **Análisis sintáctico** El análisis sintáctico (*parsing*) tiene como cometido el etiquetado de cada componente sintáctico (sintagma) que aparece en una oración, construyendo en el proceso un árbol de análisis sintáctico, el cual tiene como nodo raíz el símbolo de oración, y tiene como hojas a las palabras.

  Un analizador sintáctico (*parser*), para la construcción de dicho árbol, toma como entrada un texto y una determinada gramática, y devuelve el árbol, o falla en caso de no poder hacerlo con las reglas de producción gramaticales especificadas.

  Los algoritmos de análisis sintáctico pueden ser agrupados de acuerdo a su estrategia de análisis: descendente (*top-down*) y ascendente (*bottom-up*). En el tipo descendente, se comienza a partir de la raíz del árbol (el símbolo de oración) y se trabaja hasta llegar a las hojas (las palabras).

  En la figura 1 se puede apreciar un ejemplo de *parsing* descendente realizado sobre el texto "tomo jugo de manzana" y la siguiente gramática libre de contexto:

```
O ->GV
GV ->V GN
GN ->N GP
GN ->N
GP ->P GN
P ->de
N ->manzana
N ->tomo
V ->tomo
```

  Los primeros dos árboles corresponden a fallos en la generación del árbol sintáctico, son árboles que no se corresponden con el texto de entrada, mientras que el árbol final es un árbol sintáctico válido para la entrada. En el primer caso, se genera un árbol que no representa todo el texto de entrada, llegando sólo hasta "tomo jugo". El segundo árbol falla debido a que utiliza una regla de producción que genera un símbolo no terminal GP (Grupo Preposicional) excedente, pero no hay más palabras en la entrada sin procesar.

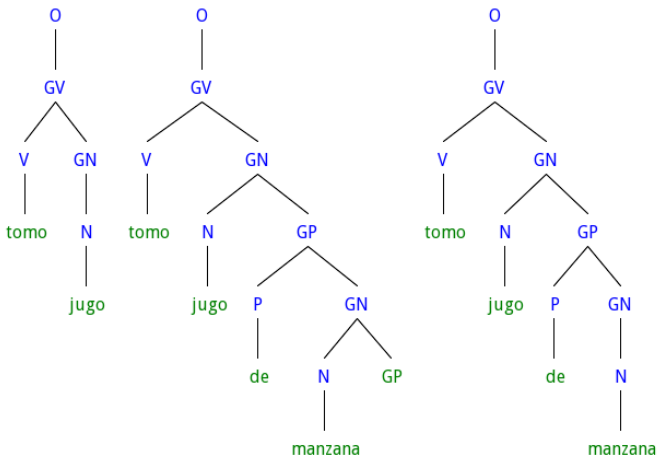

Figura 1: Análisis sintáctico descendente

El tipo ascendente comienza desde las hojas (las palabras de la entrada) y se sube hasta llegar al símbolo de la oración. En estos tipos de análisis, se puede llegar a explorar árboles que no llegarán nunca al símbolo de la oración.

En la figura 2 se tiene un ejemplo de *parsing* ascendente utilizando nuevamente el texto "tomo jugo de manzana", comenzando desde las palabras e intentando llegar al símbolo de oración. El primer árbol corresponde a un fallo de generación del árbol, el *parser* elige considerar la palabra "tomo" como sustantivo e intenta armar el árbol a partir de las reglas de producción, sin poder llegar al símbolo de oración. El segundo árbol es correcto para la entrada, al considerar a "tomo" como verbo, puede construir el árbol exitosamente.

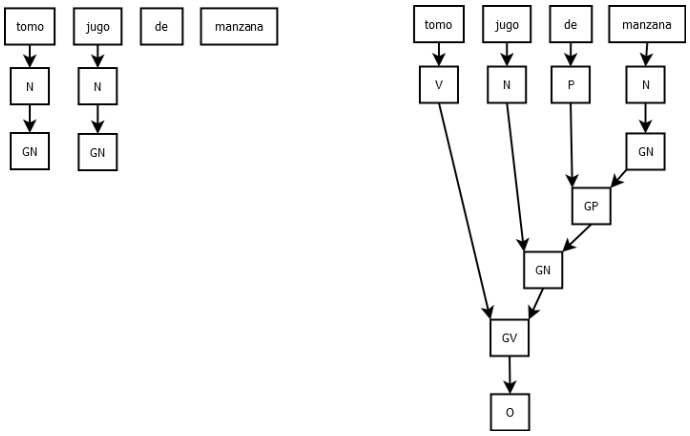

Figura 2: Análisis sintáctico ascendente

- **Análisis semántico y pragmático** El análisis semántico es la tarea de procesar el lenguaje natural y comprender su significado. Esto permite saber si una oración es una correcta representación de la realidad,

responder preguntas o saber que se requeriría para responderlas en caso de faltar información. Permitiría que una máquina pudiera entender que "Pablo le vendió un libro a Juan" y "Juan compró un libro a Pablo" son semánticamente equivalentes, y que si una es verdadera también lo es la otra.

La pragmática provee contexto: conocimiento del mundo, convenciones de uso del lenguaje, creencias e intenciones de los participantes en la comunicación.

1.3. Aplicaciones de PLN

A continuación se presenta una lista no exhaustiva de algunas de las aplicaciones más comunes del PLN, algunas son aplicaciones directas, mientras que otras sirven como subtareas para resolver problemas más grandes[8]. Cada una involucra un problema bien definido, una métrica estándar, un corpus estándar sobre el cual evaluar la tarea, y competencias dedicadas a la tarea específica.

- Traducción Automática: es la traducción automatizada de un idioma a otro. Es una de las tareas más difíciles y es considerado dentro de la categoría de problemas AI-completos.
- Reconocimiento de Entidades con Nombre (NER): es el reconocimiento y mapeo de elementos del texto a nombres propios, tales como nombres de personas, lugares, organizaciones, etc. Aunque las mayúsculas pueden ayudar en el reconocimiento en algunos lenguajes, esto por sí solo no es suficiente, entre otras razones porque frecuentemente las entidades con nombre abarcan múltiples palabras, las cuales no todas comienzan en mayúsculas[9].
- Resumen automático: es la extracción de contenido de una fuente de información, obteniendo lo más importante y devolviéndolo en un formato condensado para las necesidades del usuario[10].
- Extracción de Información: es la tarea de extraer información estructurada de documentos no estructurados.
- Categorización de documentos: es la tarea de asignar un documento a una o más categorías.

1.4. Planteo del Problema y Motivación

Se ha introducido al PLN como disciplina y se han reseñado sus cuatro niveles de análisis: morfológico, sintáctico, semántico y pragmático. El estudio comprendido en estas etapas es, como se explicó, secuencial, es decir que éstas se realizan una detrás de otra y en el orden especificado. Para cada etapa existe software especializado para su ejecución, es así que existen programas especialmente diseñados para análisis sintáctico, o para etiquetado gramatical. Estas herramientas son imprescindibles para el PLN dado que realizan las etapas necesarias para cualquier aplicación directa como la traducción automática, sin embargo el uso de cada herramienta tiene aparejado un conjunto de problemas propios:

- Conjunto heterogéneo de entornos de ejecución: cada herramienta ejecuta sobre un entorno diferente, siendo éste el sistema operativo directamente (ejecución nativa), la Java Virtual Machine (JVM), Python, etc. La variedad de entornos agrega una complejidad en su uso, algunas herramientas sólo pueden ser ejecutadas desde la línea de comandos del sistema operativo, otras sólo desde un programa Java o desde un intérprete de Python; cada entorno con sus peculiaridades inherentes. Esto le implica esfuerzo agregado al usuario de estudiar la documentación para inferir la forma de ejecución en adición al uso en sí.

  Ejemplo: MaltParser se ejecuta en la JVM, una máquina virtual; FreeLing se ejecuta nativamente contra el sistema operativo, sin intermediarios; NLTK es ejecutado en un interprete Python, el cual toma las instrucciones y las traduce a llamadas al sistema operativo a medida que las lee.

- Formatos de entrada y salida no estandarizados: diferentes paquetes de software desarrollados por diferentes personas operan de variadas formas. De esta manera se encuentran casos en los cuales programas del mismo tipo (por ejemplo, dos etiquetadores gramaticales) reciben los datos de entrada en formatos dispares, y devuelven sus resultados también en formatos incompatibles. Esto conlleva un nuevo esfuerzo agregado, el de la interpretación de los datos. En adición, los formatos dispares evitan que los resultados de variadas herramientas puedan ser explotados de forma común, es decir, si se fuera a procesar en lote una gran cantidad de datos de varios etiquetadores gramaticales de forma automática, el programa tendría que contemplar cada formato particular para poder procesarlo.

  Ejemplo: NLTK recibe como parámetro de entrada estructuras de datos de Python, mientras que FreeLing recibe los datos de entrada mediante un archivo de texto.

- *Tagsets* múltiples: como en el punto anterior, los variados programas cada uno puede utilizar un *tagset* diferente, esto hace que la interpretación de

los datos de salida presente una nueva dificultad, además de estar en un formato particular también puede estar utilizando un *tagset* diferente.

Ejemplo: FreeLing utiliza el *tagset* EAGLES, pero TreeTagger utiliza un *tagset* propio completamente diferente.

El objetivo de este proyecto es el desarrollo de una plataforma de software que provea soluciones a los problemas antes planteados (conjunto heterogéneo de entornos de ejecución, formatos de entrada y salida no estandarizados, múltiples *tagsets*), simplifique la utilización de las herramientas, y permita su interoperabilidad mediante la definición de mecanismos de comunicación. La plataforma unifica los entornos de ejecución, y es la responsable de homogeneizar los formatos de entrada y salida, así como los *tagsets* utilizados.

La plataforma está concebida primariamente para uso interno del Grupo de Procesamiento del Lenguaje Natural (PLN) del InCo, de esta forma se prestó especial atención a las herramientas más utilizadas, haciendo hincapié en el idioma español.

En el capítulo 2 de este informe se realiza un estudio del estado del arte, haciendo un relevamiento de las herramientas más utilizadas, en el capítulo 3 se detalla el diseño de la plataforma, en el capítulo 4 se detallan los detalles de su implementación, y finalmente en el capítulo 5 se establecen las conclusiones y se plantea el trabajo a futuro.

2. Estado del Arte

El desarrollo de una plataforma de software destinada a unificar herramientas requiere de un estudio previo de éstas. Es necesario un entendimiento acerca de su funcionamiento para poder realizar un diseño adecuado de la solución. Con esto en mente se realizó un estudio del estado del arte del área. Como se mencionó con anterioridad, el Grupo de Procesamiento del Lenguaje Natural utiliza con mayor frecuencia un subconjunto de las herramientas más populares en PLN, de modo que éste ha sido centro del estudio al momento de realizar el diseño.

2.1. Etiquetadores Gramaticales

Un etiquetador gramatical es un programa responsable de realizar POS-*tagging* sobre un texto tokenizado, en el cuadro 1 se listan los *taggers* incluidos en el estudio, el modelo empleado por cada uno para llevar a cabo su tarea, si proveen información morfológica, y si están disponibles para el español.

Nombre	Modelo	Español	Info. Morfo.
Morfette	Averaged perceptron	√	√
RDRPOSTagger	Single classification ripple down rules	√	√
TreeTagger	Modelo de Markov / decision trees	√	×
GENIA Tagger	Maximum Entropy	×	×
Stanford Log-linear Parser	Maximum Entropy	√	×
SVM Tool	Support Vector Machines	√	×
FreeLing	Modelo de Markov	√	√

Cuadro 1: POS-taggers

2.1.1. TreeTagger

TreeTagger es un programa que permite anotar texto con información acerca de las etiquetas POS y lemas, de forma independiente al lenguaje. Desarrollada en C por Helmut Schmid, la aplicación es un *tagger* sumamente rápido, capaz de procesar unos 8000 *tokens* por segundo. En adición la aplicación ejecuta bajo Windows, Linux y Mac.

El paquete de software está compuesto por dos aplicativos: uno de entrenamiento, el cual dado un lexicón y un corpus anotado a mano produce un archivo parámetro; y el *tagger*, que toma el parámetro y un archivo de entrada, y anota el texto con etiquetas POS y lemas.

Puede ser entrenado para cualquier lenguaje, dado un corpus anotado y lexicón, pero sin embargo cuenta *out-of-the-box* con archivos parámetros para una gran variedad de lenguajes, entre ellos el español, para el cual fue entrenado utilizando el corpus CRATER[11], y el lexicón del corpus CALLHOME[12].

En la tabla 2 se pueden ver los resultados de la evaluación de TreeTagger realizadas en [13] contra información del corpus Penn-Treebank, se utilizaron unas dos millones de palabras para entrenar, y 100.000 para evaluar. Para comparar, se utilizó un *tagger* trigrama (ver n-grama) estándar. Luego cuatro instancias de TreeTagger: la primera es TreeTagger utilizando bigramas, luego TreeTagger utilizando trigramas y reemplazando las frecuencias de palabras en 0 por 0.1, en tercer lugar TreeTagger en versión cuatrigrama, y finalmente TreeTagger trigrama reemplazando las frecuencias de palabras en 0 por el valor 10^{-10}.

Método	Contexto	Accuracy
Tagger trigrama	trigrama	0.9606
TreeTagger	bigrama	0.9578
TreeTagger (0.1)	trigrama	0.9634
TreeTagger	cuatrigrama	**0.9636**
TreeTagger (10^{-10})	trigrama	0.9632

Cuadro 2: Accuracy TreeTagger

Existen una multitud de *wrappers* del TreeTagger, incluyendo para Java y Python, e incluso existen algunas interfaces web.

El etiquetador se encuentra descripto en [13] y en [14] Al momento de escribir este documento, se encuentra en la versión 3.2[1].

2.1.2. RDRPOSTagger

RDRPOSTagger es un toolkit independiente del lenguaje para realizar POS-*tagging* basado en reglas, utilizando una metodología de Single Classification Ripple Down Rules.

Las técnicas utilizadas por el *tagger* están descriptas en [15] y [16] por Dat Quoc Nguyen, Dai Quoc Nguyen, Dang Duc Pham, y Son Bao Pham, los autores de este proyecto de código abierto construido en Python.

El *tagger* es independiente del lenguaje, sin embargo cuenta con una cantidad de modelos preentrenados para una variedad de idiomas, entre ellos uno para el español, entrenado sobre el corpus IULA LSP TreeBank. Se utiliza un *tagger* inicial simple para etiquetar cada palabra extraída del lexicón, sin embargo es posible reemplazar este *tagger* inicial por uno propio.

Se estima una velocidad de procesamiento de 2800 palabras por segundo para el inglés, sin embargo existe una implementación en Java capaz de procesar 92.000 palabras por segundo, esto se explica sencillamente por la diferente naturaleza de las plataformas: Python es un lenguaje dinámico e interpretado, mientras que Java corre sobre la JVM ejecutando *bytecode* compilado.

En [15] se dan los resultados de una evaluación hecha sobre el corpus Penn Treebank, obteniendo un *accuracy* de 0.97095.

[1] http://www.cis.uni-muenchen.de/ schmid/tools/TreeTagger/, último acceso: 14/09/14

Al momento de este artículo, el RDRPOSTagger se encuentra en su versión 1.1.3[2].

2.1.3. Morfette

Ésta es una herramienta de aprendizaje superficial de morfología inflexional. Dado un corpus de oraciones anotadas con etiquetas de lemas, etiquetas morfológicas, y opcionalmente un lexicón, morfette aprende a realizar análisis morfológico.

En la etapa de aprendizaje, se crean dos modelos diferentes: uno para *tagging* y otro para lematización. Las predicciones de ambos modelos luego se combinan dinámicamente.

La lematización se realiza como una tarea de clasificación, donde una clase de lematización se corresponde con la secuencia de operaciones de edición necesarias para transformar una palabra flexionada en su lema correspondiente.

Morfette utiliza algoritmos del tipo *Averaged Perceptron*, en lugar de Máxima Entropía tal cual usaba originalmente según se describe en [17] y [18]

Cuenta con varios modelos preentrenados, entre ellos uno para el español y más notablemente el francés (usado sobre un corpus de 160 millones de palabras), también es posible entrenarlo sobre corpus nuevos.

En [18] se realiza una evaluación de Morfette, obteniendo los resultados mostrados en la tabla 3 para las tareas de etiquetado morfológico, y lematización para el español, rumano y polaco.

Idioma	Morfo	Lematización
Español	0.9433	**0.9784**
Rumano	**0.9683**	0.9778
Polaco	0.8187	0.9329

Cuadro 3: *Accuracy* Morfette

Los autores son Grzegorz Chrupala, Djamé Seddah, Georgiana Dinu y Josef van Genabith, de la Universidad de Tilburg, Université Paris 4 La Sorbonne, University of Trento, y Dublin City University, respectivamente.

Morfette fue hecho sobre la plataforma funcional Haskell. Actualmente se encuentra en su versión 0.4.0[3].

2.1.4. GENIA Tagger

GENIA Tagger es un etiquetador gramatical que analiza oraciones en inglés y devuelve formas base, POS-*tags* y etiquetas de entidades nombradas. Está específicamente diseñado para textos de biología molecular.

Los *taggers* de propósito general usualmente no obtienen buenos resultados en textos biomédicos por sus particularidades léxicas, sin embargo este *tagger* es entrenado no sólo en el corpus Wall Street Journal, sino también en los

[2]http://rdrpostagger.sourceforge.net/, último acceso: 14/09/14
[3]https://sites.google.com/site/morfetteweb/home, último acceso: 14/09/14

corpus GENIA y PennBioIE, de modo que el *tagger* funciona bien en varios tipos de textos biomédicos.

	Wall Street Journal	Corpus GENIA
Tagger entrenado en corpus WSJ	**0.9705**	0.8519
Tagger entrenado en corpus GENIA	0.7857	**0.9849**
GENIA Tagger	0.9694	0.9826

Cuadro 4: Tabla comparativa de *accuracy*

A su vez, el GENIA Tagger es capaz de realizar NER, entrenado sobre el *dataset* NLPBA[19], se detalla en el cuadro 5 el desempeño del etiquetador en la tarea.

Entidad	Recall	Precision	F-score
Protein	**0.8141**	0.6582	0.7279
DNA	0.6676	0.6564	0.6620
RNA	0.6864	0.6045	0.6429
Cell Line	0.5960	0.5612	0.5781
Cell Type	0.7054	**0.7851**	**0.7431**
Promedio	0.7578	0.6745	0.7137

Cuadro 5: Desempeño de GENIA Tagger en tarea NER

Se encuentra descrito en [20], [21] y [22]

Está desarrollado en C, y actualmente se encuentra en su versión 3.0 desde el 2006[4].

[4]http://www.nactem.ac.uk/tsujii/GENIA/tagger/, último acceso: 14/09/14

2.1.5. Stanford Log-linear Part-Of-Speech Tagger

Stanford Log-linear POS Tagger es un etiquetador de Máxima Entropía de código abierto de Stanford. Es una implementación en Java de los etiquetadores log-lineal descritos en [23].

Puede ser utilizado como una aplicación independiente de línea de comandos, o como biblioteca Java. Adicionalmente, hay una multitud de *wrappers* escritos por terceros para plataformas como Ruby, Python (integrado con NLTK), .NET, GATE, o javascript, entre otros.

En [23] se realiza una evaluación sobre el corpus Penn Treebank, en la cual se obtiene un promedio *accuracy* de 0.9686.

Existen modelos pre entrenados para el español[5] a partir de la versión 3.4.1.

2.1.6. SVMTool

SVMTool[6] es un generador de código abierto de taggers secuenciales basados Support Vector Machines. Fue utilizado para POS-*tagging* y Base Phrase Chunking, obteniendo resultados competitivos con otros *taggers*. Para inglés se reporta un *accuracy* de 0.972, y para español de 0.9689. La aplicación fue desarrollada en Perl, sin embargo cuenta con un *port* en C++, disfrutando de un desempeño muy superior, gracias a la ejecución de código nativo.

SVMTool fue construido con seis propiedades cruciales en mente:

- Simplicidad: la herramienta es fácil de configurar y entrenar, contando con un simple archivo de configuración y pocos parámetros para tomar en cuenta. También cuenta con una API de acceso sencilla.
- Flexibilidad: El tamaño y forma del contexto de las características puede ser ajustado, y pueden ser tan ricos como se quiera.
- Robustez: Se puede manejar el problema de *overfitting* manipulando el parámetro C del algoritmo SVM. También cuenta con estrategias efectivas para manejar palabras desconocidas.
- Portabilidad: La herramienta es independiente del lenguaje, fue aplicada con éxito al español y al inglés sin más que un corpus supervisado. También puede aprender de información no supervisada sólo con la ayuda de un diccionario morfo-sintáctico.
- *Accuracy*: Resultados competitivos.
- Eficiencia: La versión de Perl exhibía una velocidad de *tagging* de 1.500 palabras por segundo, mientras que la versión de C++ alcanza una velocidad de 10.000 palabras por segundo.

[5] http://nlp.stanford.edu/software/tagger.shtml, último acceso: 14/09/14
[6] http://www.lsi.upc.edu/ nlp/SVMTool/

SVMTool se encuentra descrito en detalle en [24]. Cuenta con un portal *online* para probar la herramienta[7].

El paquete consiste de tres componentes principales, el aprendiz (SVMTlearn), el *tagger* (SVMTagger) y el evaluador (SVMTeval).

SVMTlearn hace uso de SVM-Light[8], implementación libre (para uso científico) de los SVMs de Vapnik, hecho en C.

A la fecha, se encuentra en su versión 1.3 (Perl), y 1.1.4 (C++).

[7] http://www.lsi.upc.edu/ nlp/SVMTool/demo.php

[8] http://svmlight.joachims.org

2.2. Analizadores Sintácticos

Un *parser* es una herramienta que procesa un texto, previamente etiquetado por un etiquetador gramatical, y construye un árbol de *parsing*. En el cuadro 6 se listan los *parsers* estudiados, se especifica si el árbol construido es un árbol de constituyentes o un árbol de dependencias.

Nombre	Árbol de constituyentes	Árbol de dependencias
MaltParser	×	√
DeSR	×	√
Stanford Parser	√	√
Spanish FrameNet	√	×
FreeLing	√	√

Cuadro 6: *Parsers*

2.2.1. MaltParser

El MaltParser es un sistema para realizar *data-driven dependency parsing*, generando un modelo sintáctico a partir de información en formato treebank, y obtienendo nueva a partir del modelo inducido. Los autores diferencian su sistema de los *parsers* basados en gramáticas al crear un modelo inductivo a partir del análisis sintáctico de una sentencia y el armado de una estructuras de dependencias, tras lo cual se utiliza aprendizaje automático para realizar elecciones en puntos no deterministas.

El *parser* implementa nueve algoritmos deterministas: Nivre arc-eager, Nivre arc-standard, Covington non-projective, Covington projective, Stack projective, Stack swap-eager, Stack swap-lazy, Planar y 2-planar. Se encuentra descrito en [25]

Permite a los usuarios definir modelos de características de complejidad arbitraria, e incluye dos paquetes de aprendizaje automático: LIBSVM (Support Vector Machines) y LIBLINEAR (Large Linear Classification).

MaltParser también permite a los usuarios recuperar frases continuas y discontinuas con etiquetas de frase, y funciones gramaticales.

Desarrollado en Java, MaltParser puede ser utilizado como aplicación independiente desde línea de comando, o puede ser integrado a una aplicación propia usando el JAR como biblioteca, se encuentra disponible en el repositorio oficial de Maven[9].

Es compatible con cualquier lenguaje considerando que se cuente con un treebank del lenguaje deseado en formato CoNLL[26], tras el entrenamiento con el treebank, se genera un archivo .mco (llamado archivo de configuración Single Malt), el cual es utilizado a partir de ese momento para parsear texto en el lenguaje entrenado.

Cuenta con una herramienta para optimización automática llamada MaltOptimizer[10]. La optimización de *parsers* puede ser una tarea no trivial, en

[9] http://search.maven.org/

[10] http://nil.fdi.ucm.es/maltoptimizer/, último acceso: 18/10/14

especial para desarrolladores de aplicaciones, esta herramienta ayuda a obtener resultados óptimos.

Para el modelo en español, se reportan los siguientes resultados de evaluación en dos corpus distintos[27]:

	LAS: Labeled Attachment Score	LCM: Labeled Complete Match
IULA Spanish LSP Treebank Test Set	**0.9314**	0.4760
Tibidabo Treebank	**0.8895**	0.3620

Cuadro 7: Evaluación MaltParser

Al momento de escribir este documento se encuentra en la versión 1.8, desde mayo del 2014[11].

2.2.2. DeSR

DeSR es un *parser* de dependencias de código abierto construido sobre C++ utilizando un enfoque *shift-reduce*, según descrito en [28].

El *parser* construye las estructuras de dependencias de forma voraz (*greedy*) escaneando las sentencias en una única barrida de izquierda a derecha o de derecha a izquierda, y eligiendo en cada paso si realizar un *shift* o crear una dependencia entre dos *tokens* adyacentes. La transición a realizar es aprendida de un corpus anotado, basado en las características del estado actual. El algoritmo construye árboles de dependencias completamente etiquetados.

DeSR puede ser configurado, seleccionando entre varios algoritmos de aprendizaje automático (Multi Layer Perceptron, Averaged Perceptron, Maximum Entropy, SVM), proveyendo modelos de características definidas por el usuario, etc. Cuenta con modelos preentrenados para una multitud de lenguajes, incluido el español, también se puede entrenarlo proveyendo un corpus de entrenamiento en formato CoNLL. Además de contar con una aplicación de consola, DeSR también provee una API, cuenta con un *wrapper* UIMA[12] y un simulador *online*[13].

Para el español se reporta un resultado de 0.6744 en Labeled Attachment Score (LAS) y 0.7033 en Unlabeled Attachment Score (UAS)[28] para la CoNLL-X shared task[26].

Actualmente se encuentra en su versión 1.4.3[14].

[11] http://www.maltparser.org/
[12] https://sites.google.com/site/desrparser/uima-wrapper
[13] http://medialab.di.unipi.it/Project/QA/Parser/sim.html
[14] https://sites.google.com/site/desrparser/Home, último acceso 18/10/14

2.2.3. Stanford Parser

El Stanford Parser es un paquete Java que provee una implementación de *parsers* probabilísticos, *parsers* de dependencias lexicalizadas y de gramáticas libres de contexto probabilísticas (PCFG), y un parser PCFG. Provee una interfaz de usuario que permite ver la estructura de una frase en forma de árbol.

Se proveen modelos para el inglés, Alemán, Árabe y español, entre otros.

Stanford Parser se encuentra descrito en los *papers* [29], [30] y [31].

La salida del *parser* incluye Standford Dependencies[15], el cual muestra una representación de relaciones gramáticas entre palabras de una sentencia, son fáciles de entender y son 3-uplas con la siguiente información: nombre de la relación, gobernador y dependiente.

A partir de la versión 3.4 se incluyó código para la ejecución de *parsing shift-reduce*. Versiones anteriores del Stanford Parser utilizaban algoritmos basados en programación dinámica para construir el árbol de análisis sintáctico, intentando buscar el mayor *accuracy* utilizando una gramática PCFG, el resultado era muy bueno, pero lento. Para *dependency-parsing*, los *parsers* basados en transiciones que utilizan operaciones *shift-reduce* para construir árboles de dependencias obtienen muy buenos resultados en una fracción del tiempo de algoritmos más complejos. Se afirma que algoritmos similares *shift-reduce* también pueden ser efectivos para la construcción de árboles de *parsing*. El *parser shift-reduce* es más rápido que versiones anteriores del Stanford Parser y tiene más *accuracy* que cualquier otro *parser* salvo el que utiliza Recursive Neural Networks.

En la tabla 2.2.3 se ven los resultados de una evaluación en la cual se contrasta el Stanford Parser con su versión Shift-Reduce[32] (ver F-Score).

Modelo	F1 de mejor modelo anterior	F1 de SR Parser
Árabe	**0.7815**	0.7966
Chino	0.7566	0.8023
Francés	0.7622	**0.8027**
Alemán	0.7219	0.7404

Cuadro 8: Evaluación Stanford Parser y Stanford Parser Shift-Reduce

Existen múltiples *wrappers* para otros lenguajes/plataformas, como Python, Jython, y .NET.

La versión actual es la 3.4, está disponible bajo la licencia GPL[16].

2.2.4. Spanish FrameNet

El proyecto FrameNet [17] es una base de datos léxica basada en anotar ejemplos de como las palabras son usadas en textos reales. Su conjunto de

[15] http://nlp.stanford.edu/software/stanford-dependencies.shtml
[16] http://nlp.stanford.edu/software/lex-parser.shtml
[17] http://sfn.uab.es

datos sirve para entrenamiento en etiquetado de roles semánticos y puede ser utilizado por ejemplo en extracción de información, traducción automática, reconocimiento de eventos, análisis de sentimientos, etc.

Está basado en una teoría llamada *Frame Semantics*, la cual estipula que la mayoría de las palabras pueden ser mejor entendidas en base a un *frame* semántico: una descripción de un tipo de evento, relación, o entidad y sus participantes[33]. Un *frame* semántico es una estructura de inferencias, conectado por convenciones lingüísticas a los significados de las unidades léxicas.

El proyecto Spanish FrameNet es desarrollado por la Universidad Autónoma de Barcelona y el International Computer Science Institute de Berkeley en cooperación con el proyecto FrameNet, y busca crear un recurso léxico online para el español basado en *Frame Semantics*, utilizando un corpus de 937 millones de palabras.

SFN provee tres herramientas *online*:

- Corpus del Español Actual: 540 millones de palabras lematizadas y etiquetadas con información de POS. Puede ser consultado a través de una interfaz web[18].
- Diccionario Español[19]
- Parser para el Español[20]: Construye árboles de *parsing* a partir de una frase en español, y devuelve la salida en formato SVG.

[18]http://spanishfn.org/tools/cea/english
[19]http://spanishfn.org/tools/dictionary
[20]http://spanishfn.org/tools/parser

2.3. Toolkits

En esta categoría se incluyen paquetes de *software* que no son fáciles de colocar en una sola de las anteriores categorías, o que abarcan múltiples categorías. En general son paquetes que agrupan diferentes funcionalidades en bibliotecas o entornos, por ejemplo un programa que realice tanto POS-*tagging*, como análisis sintáctico.

2.3.1. Apache OpenNLP

Apache OpenNLP es un *toolkit* de aprendizaje automático que soporta tareas comunes como tokenización, segmentación de oraciones, POS-*tagging*, NER, chunking, análisis sintáctico, etc. Incluye los algoritmos de máxima entropía y perceptron.

El *toolkit* es independiente del idioma, pudiendo ser entrenado manualmente para el que se desee, y provee modelos preentrenados para varios idiomas, lamentablemente en la actualidad para el español sólo provee modelos para la tarea de NER[21], sin embargo existen modelos provistos por terceros[22] para la tarea de POS-*tagging*. Cada una de las tareas cuenta con una API para su integración con otras herramientas, y una interfaz de línea de comandos. La biblioteca está hecha en Java y es accesible desde el repositorio público de Maven. Existen facilidades para utilizarlo desde .NET.

Sus componentes son:

- *Sentence detection*: Este componente es básicamente un desambiguador de los caracteres de puntuación, separando cada oración en su propia línea.
- *Tokenizer*: Segmentador de texto en *tokens*, son usualmente palabras, símbolos de puntuación, números, fechas, etc. Contiene múltiples implementaciones: *tokenizer* por espacios en blanco, el cual agrupa secuencias de caracteres que no son espacios; un *tokenizer* simple, que cataloga los *tokens* en secuencias de la misma clase de caracter; y un *tokenizer* de máxima entropía, que detecta bordes de *token* basado en un modelo probabilístico.
- *Detokenizer*: Realiza la operación inversa al *tokenizer*.
- NER: Reconocedor de entidades con nombre y números, por ejemplo detecta cuando se está hablando de una persona, de una organización, etc. Depende fuertemente del lenguaje y los modelos utilizados.
- *Document Categorizer*: Permite clasificar texto en categorías predefinidas. Basado en un framework de máxima entropía, depende del modelo utilizado. No contiene modelos por defecto, dado que las categoría dependen de los requisitos del usuario.

[21] http://opennlp.sourceforge.net/models-1.5/
[22] http://cavorite.com/labs/nlp/opennlp-models-es/

- POS-*tagger*: Etiquetador gramatical.
- *Chunker*: Divisor de texto en partes sintácticamente correlacionadas del texto, como grupo nominales, grupos verbales, etc., sin indicar su estructura interna como lo haría un *parser*, ni su rol en la oración.
- *Parser*: Analizador sintáctico.

La plataforma se encuentra descrita por múltiples *papers* en [34][35][36][37][38][39][40]. Se encuentra en su versión 1.5.3[23].

2.3.2. Natural Language Toolkit

El Natural Language Toolkit[24] (NLTK) es un conjunto de bibliotecas para PLN en Python. Provee interfaces simples de usar para más de 50 corpus y recursos léxicos como WordNet, y una suite de bibliotecas de procesamiento de texto para clasificación, *tokenization*, *stemming*, *tagging*, *parsing* y etiquetado semántico[41][42].

Algunas de las herramientas provistas por NLTK son:

- Tokenización de palabras.
- Tokenización de oraciones.
- Etiquetador gramatical.
- *Chunkers*.
- Reconocedor de entidades.
- Expresiones regulares.
- Gramáticas.
- *Parsers*.
- Stemmers/lematizadores.
- WordNet
- Algoritmos de aprendizaje automático.
- Corpus.

[23] http://opennlp.apache.org/, último acceso: 15/09/14
[24] http://www.nltk.org/

2.3.3. FreeLing

FreeLing es una suite de herramientas para análisis del lenguaje, de código abierto. Está diseñado para ser utilizado como una biblioteca externa, aunque provee un pequeño aplicativo de línea de comando para utilizarlo[43].

Algunas de las características de FreeLing:

- Tokenización
- Segmentación en oraciones.
- Análisis morfológico.
- Tratamiento de sufijos.
- POS-*tagging*.
- Shallow-parsing.
- NER.
- Predicción probabilística de categorías de palabras desconocidas.

Ente los idiomas soportados, se encuentran el inglés y español.

Está escrito en C++ usando STL, de modo que puede ser compilado en casi cualquier plataforma, aunque es preferible Linux.

La versión actual es la 3.1[25].

2.3.4. General Architecture for Text Engineering (GATE)

GATE es una suite de herramientas de código abierto en Java para tareas de PLN. Se compara usualmente con NLTK. Es una infraestructura para desarrollar y desplegar componentes que procesan lenguaje natural[44].

Contiene un entorno de desarrollo integrado (integrated development environment o IDE) propio, GATE Developer[26] para la construcción de componentes de software para la plataforma, ayuda en la construcción de máquinas de estados, support vector machines y otras estructuras de datos complejas y permite la visualización de los datos procesados, es conocido como "el Eclipse del Procesamiento del Lenguaje Natural", en referencia al IDE de programación Eclipse.

La aplicación web GATE Teamware[27] provee un ambiente colaborativo para realizar anotación semántica.

Entre las prestaciones, la más importante es la arquitectura, la cual permite una organización de alto nivel de composición de componentes de PLN.

Entre los lenguajes soportados se encuentra el español.

Su versión actual es 8.0[28].

[25] http://nlp.lsi.upc.edu/freeling/
[26] http://gate.ac.uk/family/developer.html
[27] http://gate.ac.uk/teamware/
[28] http://gate.ac.uk/

2.3.5. Stanford CoreNLP

El Stanford CoreNLP es una suite de herramientas de análisis del lenguaje natural que dado un texto crudo, puede extraer las formas base de las palabras, sus part-of-speech, obtener nombres de entidades, normalizar fechas, obtener cantidades, y determinar la estructura de oraciones en términos de frases y dependencias de palabras, indicando que grupos nominales se refieren a que entidades, etc. Es un *framework* integrado, lo que permite aplicar fácilmente las herramientas de manera secuencial en un *pipeline*, se pueden utilizar todas las herramientas con tan sólo un par de líneas de código, y provee bloques fundacionales para la construcción de aplicaciones de procesamiento de texto de más alto nivel para dominios específicos.

Integra muchas herramientas, como ser el Stanford Log-Linear POS Tagger y el Stanford Parser, ya antes mencionados, junto con herramientas de reconocimiento de entidades nombradas, resolución de coreferencia, y análisis de sentimientos.

Se halla descrito en [45].

La suite es de código abierto, publicada bajo GPL, y se encuentra en la versión 3.4.1[29].

2.4. Visualizadores y Anotadores

Se define visualizador a un programa que no realiza tareas de PLN, sino que su operativa es complementaria a la de las herramientas antes descritas. En general su operativa radica en tomar la salida de otro programa, y presentarlo de una forma visualmente más atractiva que la estándar exportando, por ejemplo, a un formato de imagen con colores que facilite la lectura del resultado. Un anotador es un software que permite realizar y visualizar anotaciones con información sintáctica sobre un texto determinado.

2.4.1. GrammarScope

GrammarScope es una herramienta construida en Java la cual toma un texto analizado con el *parser* de Stanford, y lo despliega de forma gráfica el árbol de análisis sintáctico, la estructura gramática, las dependencias tipadas y el grafo semántico.

Se encuentra en su versión 2.0[30]

[29] http://nlp.stanford.edu/software/corenlp.shtml, Último acceso: 18/10/14

[30] http://grammarscope.sourceforge.net/

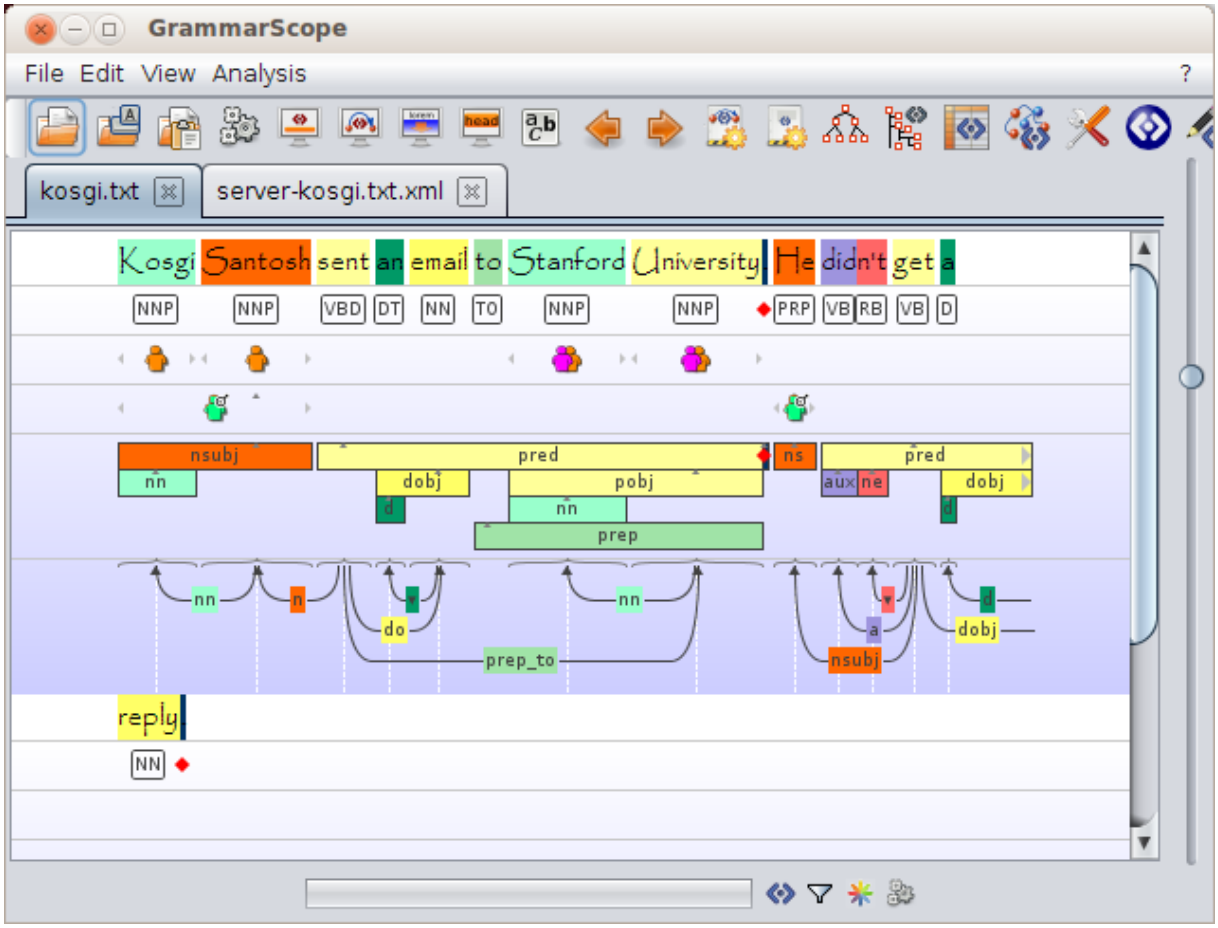

Figura 3: GrammarScope 2.0

2.4.2. Dependency Grammar Annotator (DgAnnotator)

DgAnnotator es un utilitario para visualizar grafos de dependencias[31], así como crearlos y manipularlos.

Cuenta con las siguientes características:

- Lee y produce archivos en formato CoNLL, así como en XML.
- Muestra diferencias entre dos árboles de dependencias sobre el mismo corpus.
- Exporta los árboles sintácticos en un formato gráfico.
- Realiza POS-*tagging* conectando a un POS-*tagger* en red.
- Tiene versiones para Windows y Linux.

[31] http://medialab.di.unipi.it/Project/QA/Parser/DgAnnotator/

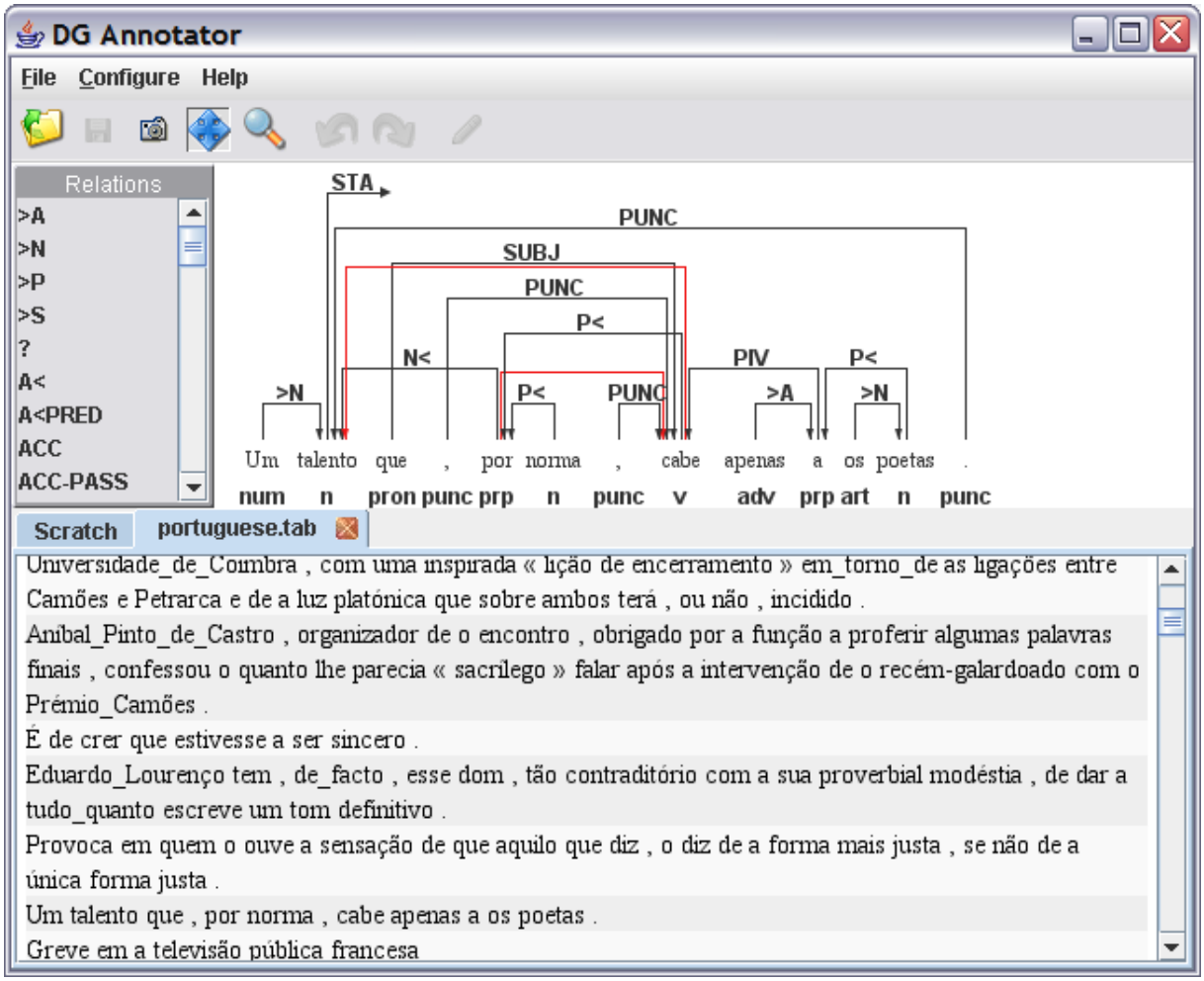

Figura 4: DgAnnotator

2.4.3. DependenSee

DependenSee es una herramienta desarrollada en Java que toma el grafo de dependencias de una frase generado por el *parser* de Stanford, y lo exporta en un gráfico en formato PNG.

Se encuentra en su versión 2.0.5[32].

[32]http://chaoticity.com/dependensee-a-dependency-parse-visualisation-tool/, Último acceso: 18/10/14

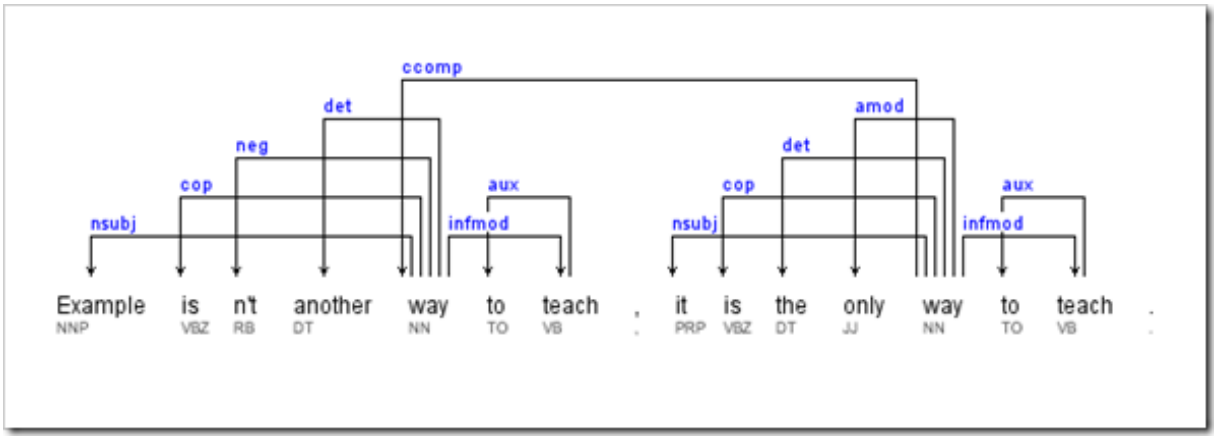

Figura 5: DgAnnotator

2.4.4. brat

brat es un ambiente *online* gratuito para anotado colaborativo que permite el trabajo concurrente de múltiples usuarios sobre un texto. Permite de manera sencilla anotar palabras o conjuntos de palabras y establecer relaciones entre las anotaciones. A su contiene otras funcionalidades secundarias como ser comparación de anotaciones entre diferentes versiones de un mismo texto, integración con herramientas externas automáticas, exportación a archivos de imagen, etc. El software está diseñado para ser desplegado en un web server local, de forma que una organización lo utilice según sus necesidades.[33]

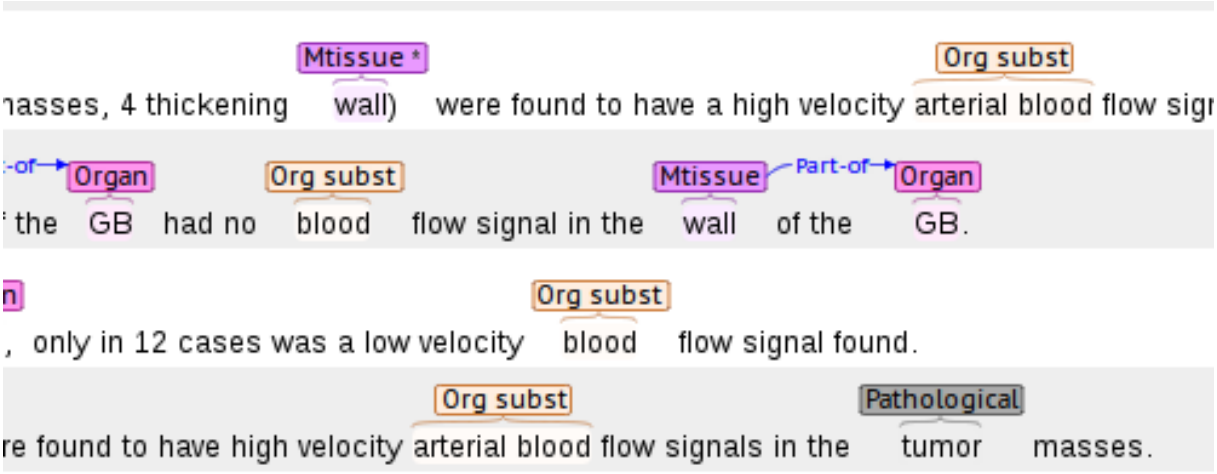

Figura 6: brat

[33] http://brat.nlplab.org/index.html

3. Diseño

Habiendo realizado un estudio del estado del arte, se cuenta con suficiente información para avanzar con la resolución de los problemas planteados en 1.4, siendo éstos:

- Fragmentación de ambientes de ejecución: cada herramienta se ejecuta de forma diferente, se desea un ambiente único que simplifique su uso.
- Formatos de entrada/salida heterogéneos: diferentes herramientas dentro de una misma familia (tokenizers, etiquetadores gramaticales, analizadores sintácticos) difieren en el formato que reciben los datos de entrada y el formato en el cual devuelven el resultado de su ejecución, se busca la estandarización de los formatos para cada familia de aplicaciones.
- *Tagsets* dispares: Los *tagsets* utilizados por cada herramienta pueden variar, dificultando la interoperabilidad entre las herramientas; se busca una estrategia a aplicar para facilitar la interoperabilidad.

El objetivo final es el diseño e implementación de una biblioteca que permita al programador utilizar las diversas herramientas sin necesidad de enfrentarse con los problemas planteados, esto se logrará mediante la integración de las herramientas en dicha biblioteca. Dado que el universo conformado por todos los programas de PLN es demasiado extenso, se ha reducido el conjunto de los programas a integrar a los más utilizados por el Grupo, estos son: TreeTagger, FreeLing, MaltParser y Stanford Shift-Reduce Parser (existen dos tipos de parsers de Stanford: el lexicalized parser y el shift-reduce parser, para el presente trabajo se enfocó el esfuerzo en el shift-reduce dado que éste cuenta con modelo para el idioma español). Cabe notar, sin embargo, que la solución planteada es genérica y puede ser aplicada a cualquier herramienta.

En este capítulo se expone el diseño de la solución al problema, se explican las decisiones tomadas, qué objetivos resuelven, y qué forma le dan a la solución final.

3.1. Uniformización de ambientes

Para comenzar con la descripción del diseño de la solución al primer objetivo, se debe documentar los contextos de las diferentes herramientas a integrar, de esta forma se explicará como se aplica la solución propuesta a cada caso.

TreeTagger y FreeLing son dos aplicativos construidos sobre C++, de modo que se ejecutan de forma nativa en el sistema operativo mediante invocaciones desde el *shell* (línea de comandos), mientras que MaltParser y Stanford Parser están desarrollados en Java, y su ambiente de ejecución es la JVM. En adición a los anteriores, se tiene por otro lado a NLTK, valiosa herramienta para PLN, el cual ejecuta sobre el intérprete de Python. Los entornos de ejecución, como se puede observar, varían.

Visto esto, parece natural que el primer objetivo a resolver sea la uniformización de ambientes de ejecución, y la estandarización del modo de invocación

de las distintas herramientas, dado que estas decisiones condicionan inevitablemente la implementación de todo el proyecto.

Para proveer un ambiente uniforme de ejecución se decidió la utilización del entorno Python para el desarrollo debido a lo imperante de éste dentro del grupo de PLN del InCo, su popularidad en las áreas de Aprendizaje Automático y PLN, y lo simple que resulta el desarrollo y prototipado rápido.

La propuesta es que la plataforma adquiriría la forma de una biblioteca de clases Python, la cual contendrá los variados módulos necesarios para el cumplimiento de los objetivos.

Luego de tomada la decisión de utilizar Python para el desarrollo de la plataforma, surge la interrogante de si es más conveniente crear una plataforma nueva o utilizar una existente. A partir del estudio del estado del arte, se ha tomado conocimiento de NLTK, un *toolkit* para PLN que cumple varios de los objetivos, algunos completamente y otros parcialmente. NLTK es un toolkit escrito en Python que aglomera una gran cantidad de utilitarios y algoritmos para PLN, es código abierto, goza de buena salud (a la fecha de la escritura de este informe, se reportan unos 59 *commits* al código fuente, de diez autores diferentes[34]), está extensamente documentado, y su uso liberal de interfaces para separar contrato de implementación abre las puertas a una fácil extensión. La facilidad de extensión de NLTK es particularmente ventajosa, dado que posee una carencia importante: no cuenta con varias de las herramientas más utilizadas por el Grupo de PLN del InCo. De esta forma se decide que lo más beneficioso es la extensión de NLTK mediante el uso de sus interfaces, esto permite basar el diseño en uno ya existente y aprobado por una comunidad extensa, y extiende de manera natural el alcance del proyecto al permitir la interoperabilidad entre la biblioteca de integración y todas las herramientas ya disponibles en NLTK.

Dicha biblioteca posee dos ventajas:

- Sencillez de uso de las herramientas integradas: cada programa tiene un modo de ejecución diferente, requiriendo la utilización de distintos comandos y configuraciones, a su vez cada uno toma los datos de entrada (por ejemplo, el texto a procesar) en un formato diferente. Para hacer uso de estas herramientas, es necesario contemplar las variaciones en las formas de uso de cada una. La biblioteca está diseñada para que toda una familia de herramientas pueda ser invocada de una forma simple, abstrayendo al usuario de los detalles más finos. Adicionalmente, la plataforma sobre la cual está construido el *toolkit* soporta introspección, esto permite poder acceder a la documentación del código desde el propio intérprete. Para ejemplificar, se contrastan a continuación los pasos para ejecutar el analizador sintáctico Stanford Parser directamente y por medio de NLTK.

  A través de NLTK:

```
1 parser = StanfordParser(model_path=
2     "edu/stanford/nlp/models/lexparser/
```

[34] https://github.com/nltk/nltk/pulse/monthly, último acceso: 29/5/2015

```
3       englishPCFG.ser.gz")
4   resultado = parser.raw_parse(
5       "the quick brown fox jumps over the lazy dog"))
```

Directamente:

Crear archivo de texto llamado “texto.txt” e ingresar el texto deseado, luego ejecutar desde la línea de comandos

```
java -mx200m edu.stanford.nlp.parser.lexparser
.LexicalizedParser
-retainTMPSubcategories -outputFormat
"wordsAndTags,penn,typedDependencies"
englishPCFG.ser.gz texto.txt
```

Es importante recordar que en el caso de NLTK se cuenta con un IDE que provee asistencia y documentación contextual al programador. Cuando se utiliza el *parser* directamente no hay alternativa salvo ingresar al sitio web y estudiar la documentación, especialmente los puntos acerca del modo de ejecución (la instrucción de línea de comandos), posibles banderas para el *runtime* Java (en el ejemplo, la bandera “mx200m” indica a la JVM la cantidad de memoria que debe utilizar), y los formatos de entrada aceptados (en el ejemplo de Stanford Parser se acepta texto plano como formato de entrada, por lo que no se puede apreciar la ventaja de NLTK en esta situación, sin embargo hay otras herramientas, como ser MaltParser, que no aceptan texto plano como entrada y requieren la conformación a un cierto formato).

- Uniformización de las salidas: el contar con un formato único de salida para una determinada familia de herramientas abre las puertas a su explotación de una forma más masiva, es posible, por ejemplo, construir analizadores que procesen las salidas de los POS-*tagger* FreeLing y TreeTagger de manera indistinta, sin que cada una requiera tratamiento especial, o la construcción de visualizadores que muestren de forma gráfica la salida de los analizadores gramaticales, de nuevo sin la preocupación del tratamiento especial para una u otra herramienta.

La decisión de utilizar Python para el desarrollo de la biblioteca resuelve entonces el objetivo de un ambiente uniforme. Se diseñan módulos que envuelvan a cada herramienta (*wrappers*), efectivamente ocultando la complejidad de las formas de invocación y requiriendo únicamente su uso desde Python. Los módulos, en adición, están documentados internamente a nivel de código, proveyendo al programador con información sobre su uso (mediante inspección del código o introspección de Python).

3.2. Homogeneización de formatos de entrada/salida

El segundo objetivo corresponde a la homogeneización de los formatos de entrada (parámetros de entrada) y salida (representación de datos), para este

propósito es necesario documentar los formatos para cada una de las herramientas a utilizar.

Es de interés mencionar que todos los aplicativos leen el texto a procesar desde un archivo de texto, y devuelven el resultado de igual forma en un archivo de texto destino.

3.2.1. FreeLing

FreeLing cumple, entre otras, las funciones de *tokenizer*, etiquetador gramatical, y analizador sintáctico, de modo que se deben analizar los formatos esperados para cada tipo de funcionalidad. En el caso de la tokenización, la herramienta espera recibir sencillamente el texto a tokenizar. El resultado devuelto es el texto tokenizado con exactamente un *token* por línea. Cuando actúa como etiquetador, se espera recibir los *tokens* del modo en que fueron devueltos por el *tokenizer*: uno por línea. La salida es de la siguiente forma:

```
<TOKEN> <LEMA> <ETIQUETA> <PROBABILIDAD>
```

Finalmente, en el modo de analizador sintáctico, FreeLing espera que la información de entrada se le presente de la forma:

```
<TOKEN> <LEMA> <ETIQUETA>
```

Su salida en este modo es un árbol de análisis sintáctico, el cual tiene una forma un poco más compleja que las vistas hasta ahora, se ejemplificará con un "Hola Mundo":

```
+interjeccio_[
  +(Hola hola I -)
  sn_[
    +grup-nom-ms_[
      +w-ms_[
        +(Mundo mundo NP00000 -)
      ]
    ]
  ]
]
```

Se puede ver que FreeLing serializa la estructura arborescente resultado de una forma bastante intuitiva, utilizando paréntesis rectos para indicar una relación de "descendiente-de", y ayudando a la visualización con indentación, los nodos que no tienen un símbolo de apertura de paréntesis rectos son las hojas del árbol de análisis, y contienen la información que se pasó como entrada

```
<TOKEN> <LEMA> <ETIQUETA>
```

3.2.2. TreeTagger

TreeTagger es un etiquetador gramatical sustancialmente más sencillo que FreeLing, ofrece muchas menos funcionalidades de análisis, sin embargo su salida es similar:

```
<TOKEN> <ETIQUETA> <LEMA>
```

3.2.3. MaltParser

En la categoría de analizadores sintácticos, MaltParser es un analizador que genera árboles de dependencias. Tanto su entrada como su salida pueden tomar uno de dos posibles formatos, según se lo configure: CoNLL y Malt-TAB. Se mostrará un ejemplo de entrada y salida utilizando el formato CoNLL, con el texto "En el tramo de Telefónica un toro descolgado ha creado peligro tras embestir contra un grupo de mozos.". El texto, en formato CoNLL, tokenizado y con sus correspondientes etiquetas POS, es de la forma:

```
1 En _ SPS00 SPS00 _ _ _ _ _
2 el _ DA0MS0 DA0MS0 _ _ _ _ _
3 tramo _ NCMS000 NCMS000 _ _ _ _ _
4 de _ SPS00 SPS00 _ _ _ _ _
5 Telefónica _ NP00000 NP00000 _ _ _ _ _
6 un _ Z Z _ _ _ _ _
7 toro _ NCMS000 NCMS000 _ _ _ _ _
8 descolgado _ VMP00SM VMP00SM _ _ _ _ _
9 ha _ VAIP3S0 VAIP3S0 _ _ _ _ _
10 creado _ VMP00SM VMP00SM _ _ _ _ _
11 peligro _ NCMS000 NCMS000 _ _ _ _ _
12 tras _ SPS00 SPS00 _ _ _ _ _
13 embestir _ VMN0000 VMN0000 _ _ _ _ _
14 contra _ SPS00 SPS00 _ _ _ _ _
15 un _ Z Z _ _ _ _ _
16 grupo _ NCMS000 NCMS000 _ _ _ _ _
17 de _ SPS00 SPS00 _ _ _ _ _
18 mozos _ NCMP000 NCMP000 _ _ _ _ _
19 . _ Fp Fp _ _ _ _ _
```

La salida de MaltParser es:

```
1 En _ SPS00 SPS00 _ 10 MOD _ _
2 el _ DA0MS0 DA0MS0 _ 3 SPEC _ _
3 tramo _ NCMS000 NCMS000 _ 1 COMP _ _
4 de _ SPS00 SPS00 _ 3 MOD _ _
5 Telefónica _ NP00000 NP00000 _ 4 COMP _ _
6 un _ Z Z _ 7 SPEC _ _
7 toro _ NCMS000 NCMS000 _ 10 SUBJ _ _
8 descolgado _ VMP00SM VMP00SM _ 7 MOD _ _
```

```
9 ha _ VAIP3S0 VAIP3S0 _ 10 AUX _ _
10 creado _ VMP00SM VMP00SM _ 0 ROOT _ _
11 peligro _ NCMS000 NCMS000 _ 10 SUBJ _ _
12 tras _ SPS00 SPS00 _ 10 MOD _ _
13 embestir _ VMN0000 VMN0000 _ 12 COMP _ _
14 contra _ SPS00 SPS00 _ 13 MOD _ _
15 un _ Z Z _ 16 SPEC _ _
16 grupo _ NCMS000 NCMS000 _ 14 COMP _ _
17 de _ SPS00 SPS00 _ 16 COMP _ _
18 mozos _ NCMP000 NCMP000 _ 17 COMP _ _
19 . _ Fp Fp _ 18 punct _ _
```

Para ilustrar las diferencias entre CoNLL y Malt-TAB, la salida en formato Malt-TAB es:

```
En SPS00 10 MOD
el DA0MS0 3 SPEC
tramo NCMS000 1 COMP
de SPS00 3 MOD
Telefónica NP00000 4 COMP
un Z 7 SPEC
toro NCMS000 10 SUBJ
descolgado VMP00SM 7 MOD
ha VAIP3S0 10 AUX
creado VMP00SM 0 ROOT
peligro NCMS000 10 SUBJ
tras SPS00 10 MOD
embestir VMN0000 12 COMP
contra SPS00 13 MOD
un Z 16 SPEC
grupo NCMS000 14 COMP
de SPS00 16 COMP
mozos NCMP000 17 COMP
. Fp 18 punct
```

Ambos formatos son intercambiables y muy similares, aunque visualmente son de interpretación mucho menos obvia que la provista por FreeLing. En primer lugar se imprimen el *token* y su etiqueta, eso es información que viene de la entrada. Luego va un número que representa el índice en la salida que le corresponde al *token* del cual el actual depende, y finalmente viene una etiqueta que representa el tipo de nodo. Así, en el ejemplo, se ve que el nodo raíz (etiquetado ROOT) es la palabra “creado”, y de ella dependen directamente las palabras “En”, “toro”, “ha”, “peligro”, y “tras”.

3.2.4. Stanford Shift-Reduce Parser

Finalmente, el Stanford Parser provee un analizador sintáctico *shift-reduce* de dependencias para el español, y emplea formatos de entrada y salida com-

pletamente diferentes a los vistos anteriormente. Para la entrada, el analizador requiere el siguiente formato:

```
<TOKEN> <ETIQUETA>
```

La salida es similar a la de FreeLing, se tiene el *token*, y mediante paréntesis se indica si tiene hijos, o no, por ejemplo:

```
{(ROOT (sentence (spec (SPS00 En) (DA0MS0 el)) (sn (spec (NCMS000
tramo) (SPS00 de)) (grup.nom (NP00000 Telefónica))) (Z un)
(NCMS000 toro) (sentence (S (sadv (grup.adv (VMP00SM
descolgado)))) (spec (grup.cc (S (sadv (grup.adv
(VAIP3S0 ha)))) (VMP00SM creado)) (NCMS000 peligro))
(SPS00 tras) (VMN0000 embestir) (S (sadv (grup.adv (SPS00
contra)))) (S (sadv (grup.adv (Z un)))) (sn (spec
(NCMS000 grupo) (SPS00 de)) (grup.nom (NCMP000 mozos)))
(Fp .))))}
```

De esta forma se ve que el formato es bastante similar, aunque de lectura más confusa al no contar con fines de línea.

3.2.5. Decisiones de diseño

Hasta aquí se ha visto ejemplificado el problema de múltiples formatos de entrada y salida, el uso de cada herramienta implica la utilización de un formato incompatible con los demás.

En 3.1 se toma la decisión de utilizar a NLTK como base para la biblioteca a desarrollar, citando como ventaja las APIs que establecen formatos de entrada y salida bien definidos para cada familia de herramientas. Las interfaces que provee NLTK describen familias de *tokenizers*, etiquetadores gramaticales, y analizadores sintácticos, estas interfaces establecen contratos que las clases que las implementan deben cumplir, viéndose forzadas a aceptar los mismos formatos de entrada y a utilizar los mismos formatos de salida. El formato utilizado por NLTK está enteramente compuesto por estructuras de datos de Python y es el siguiente:

- tokenizers: reciben una tira de caracteres de entrada (la oración), y de salida devuelven una lista de tiras de caracteres (los *tokens*).
- etiquetadores: reciben una lista de tiras de caracteres (los *tokens*), y devuelven una lista de tuplas `(TOKEN, ETIQUETA)`.
- analizadores sintácticos: reciben una tira de caracteres (la oración), y retornan una lista de estructuras arborescentes (árboles de NLTK), esta lista corresponde a los diferentes posibles análisis de la oración.

Más allá de lo que contempla NLTK, se decidió que la biblioteca contara con funcionalidades extras que proveyeran más información en un formato extensible, esto es en adición a las funcionalidades descritas por NLTK y no es su substitución.

Todas las herramientas, implementando estas interfaces, utilizarían los mismos formatos bien documentados e invariantes.

En resumen, NLTK provee interfaces que describen familias de herramientas, sus operaciones y sus formatos de entrada/salida, estos contratos garantizan que dos herramientas pertenecientes a la misma familia se comporten de forma similar. Al implementar estas interfaces se garantiza una conformación a los formatos bien definidos del *toolkit* y se alcanza el objetivo de la homogeneización de formatos de entrada/salida. Se agregan operaciones adicionales a las descritas por NLTK para facilitar aun más el uso de las herramientas integradas.

3.3. Formato extendido de salida para etiquetadores gramaticales

En adición a los formatos de salida de NLTK especificados en 3.2, se decidió crear un nuevo formato de salida para los etiquetadores gramaticales, extensible y capaz de expresar toda la información de la que fuera capaz el etiquetador. De esta forma, se elige como estructura de datos un diccionario de Python, con las claves siendo los diferentes datos de los cuales es capaz el etiquetador de obtener. Serializado a una tira de caracteres, así sería un ejemplo del formato extendido para FreeLing:

```
{word: <PALABRA>, lemma: <LEMA>, original_tag: <ETIQUETA NATIVA>,
coarse_tag: <ETIQUETA COARSE>, pos_tag: <ETIQUETA POS>,
features: [{probability: <PROBABILIDAD>}, {tagger: FreeLing}]}
```

Como se puede ver, este formato contiene más información que los formatos estándar de NLTK, en especial la clave `features`, la cual es el nodo que contiene las características extras que puede contener un etiquetador determinado, en este ejemplo se puede ver que FreeLing provee la probabilidad de una etiqueta, una característica inexistente en otros etiquetadores. El propósito del nodo `original_tag` se explica en 3.4.

A continuación se muestra un ejemplo de código utilizando los modos de etiquetado estándar y extendido:

```
1   import inco.pln.tokenize.freeling
2   import inco.pln.tag.freeling
3
4   entrada = u"En el tramo de Telefonica,
5   un toro descolgado ha creado peligro tras
6   embestir contra un grupo de mozos."
7
8   tokenizer =
9       inco.pln.tokenize.freeling
10          .FreeLing(ruta_a_freeling)
11  tagger =
12      inco.pln.tag.freeling.FreeLing(ruta_a_freeling)
13
```

```
14      # La variable tokens contiene el string
15      # de entrada tokenizado ,
16      # es una lista de strings .
17      tokens = tokenizer.tokenize(entrada)
18      texto_taggeado = tagger.tag(tokens)
19      texto_taggeado_extended =
20          tagger.tag_extended(tokens)
21
22      print "Salida estandar\n"
23      print texto_taggeado
24      print "\nSalida extendida\n"
25      print texto_taggeado_extended
```

Su salida (abreviada) es:

```
Salida estándar

[(u'En', u'SPS00'), (u'el', u'DA0MS0'), (u'tramo', u'NCMS000'),
...]

Salida extendida

[{'coarse_tag': u'S', 'lemma': u'en', 'pos_tag': u'SPS00',
  'word': u'En',
  'features': {'tagger': 'FreeLing', 'probability': u'1'}},
{'coarse_tag': u'D', 'lemma': u'el', 'pos_tag': u'DA0MS0',
  'word': u'el',
  'features': {'tagger': 'FreeLing', 'probability': u'1'}},
{'coarse_tag': u'N', 'lemma': u'tramo', 'pos_tag': u'NCMS000',
  'word': u'tramo',
  'features': {'tagger': 'FreeLing', 'probability': u'0.973363'}},
...]
```

3.4. Unificación de *tagsets*

Habiendo estudiado las diferencias entre las diferentes herramientas, se observa que varias de ellas utilizan distintos *tagsets* al momento de etiquetar gramaticalmente y de analizar sintácticamente, el objetivo perseguido aquí fue el de unificar los *tagsets* utilizados, de forma que el uso de una u otra herramienta no implicara un cambio tan radical en la información necesaria de entrada y la obtenida a la salida. Para comenzar se detallan las diferencias observadas.

FreeLing utiliza un *tagset* basado en el *tagset* propuesto por el grupo EAGLES para la anotación morfosintáctica de lenguas europeas[35], es un *tagset* sumamente descriptivo que permite expresar una multitud de datos morfológicos de las palabras. Cada etiqueta está compuesta por un conjunto de letras y dígitos que codifican las características morfológicas.

[35] http://nlp.lsi.upc.edu/freeling/doc/tagsets/tagset-es.html, último acceso: 29/5/2015

Por ejemplo: a la palabra "bonita" le corresponde la etiqueta "AQ0FS0" que representa un adjetivo (la "A" inicial), calificativo ("Q"), femenino (la "F") y singular (la "S"), los ceros siendo características que no aplican a la palabra o el lenguaje. Se observó que tanto MaltParser como el Stanford Parser utilizan ambos para el español el conjunto de etiquetas EAGLES para el español. Sin embargo TreeTagger no lo hace, utiliza un conjunto de etiquetas propio de la herramienta, y que es mucho menos descriptivo que el EAGLES[36]. Para ilustrar, la etiqueta que le corresponde a cualquier tipo de adjetivo es la etiqueta "ADJ", la cual no contiene ningún tipo de información morfológica, y se correspondería con la etiqueta EAGLES "A00000", o sea, una que indique un adjetivo pero no contenga más información.

Con el objetivo de la unificación, se tomó la decisión de utilizar el *tagset* EAGLES como *tagset* único, esto implica que las herramientas que no lo utilicen deben de ser capaces de convertir su *tagset* nativo al EAGLES. Esta decisión se vio justificada en el hecho de que es preferible no perder información morfológica, en caso de que el etiquetador la provea. Las etiquetas EAGLES son capaces de describir cualquier otro tipo de etiqueta (en particular las de TreeTagger), dado que lo único que cambia es que se colocan ceros en todos los campos faltantes.

En la tabla 9 se muestra un ejemplo de conversión de etiquetas de TreeTagger al tagset EAGLES.

[36]http://www.cis.uni-muenchen.de/ schmid/tools/TreeTagger/data/spanish-tagset.txt, último acceso: 29/5/2015

Palabra	Etiqueta TreeTagger	Etiqueta EAGLES
En	PREP	SP000
el	ART	DA0000
tramo	NC	NC00000
de	PREP	SP000
Telefónica	NC	NC00000
Fc		
un	ART	DA0000
toro	NC	NC00000
descolgado	VLadj	V0PS000
ha	VHfin	V000000
creado	VLadj	V0PS000
peligro	NC	NC00000
tras	PREP	SP000
embestir	VLinf	V0N0000
contra	PREP	SP000
un	ART	DA0000
grupo	NC	NC00000
de	PREP	SP000
mozos	NC	NC00000
.	FS	Fp

Cuadro 9: *Conversión de etiquetas de TreeTagger a etiquetas EAGLES*

3.5. Resumen

Se llega a la etapa de diseño con tres objetivos claros: la simplificación del uso de las herramientas mediante la homogeneización de tanto los ambientes de ejecución, como de los formatos de entrada y salida, y la simplificación de la interpretación del resultado de los etiquetadores gramaticales al unificar los *tagsets* utilizados.

Se decidió utilizar Python como entorno de ejecución, y en particular se optó por la integración con NLTK, un *toolkit* popular que cumple parcialmente con algunos de los objetivos y posee una API limpia y fácil de extender. La utilización de NLTK a su vez ayuda a cumplir el objetivo de simplificar los formatos de entrada y salida, al proveer interfaces bien definidas que describen contratos que deben cumplir las familias de herramientas, de esta forma, por ejemplo, todos los etiquetadores gramaticales reciben las mismas estructuras de datos como parámetros de entrada, y devuelven las mismas estructuras como resultado de su operativa, sin sorpresas entre herramientas. Adicionalmente, se diseñó un formato de salida extendido para los etiquetadores gramaticales, para proveer, si se desea, toda la información que puede retornar un etiquetador, en una estructura extensible.

Para el objetivo final, la unificación de *tagsets*, se eligió el *tagset* EAGLES como el *tagset* único para todas las herramientas de la biblioteca, se entiende que éste es suficientemente descriptivo como para poder expresar cualquier otro *tagset* con él.

4. Implementación

En el capítulo 3 se presentó el diseño de la biblioteca y como éste soluciona los problemas planteados en 1.4. El diseño propone su resolución de la siguiente manera:

- Utilización de la plataforma Python para unificar ambientes de ejecución, y tomar al *toolkit* NLTK como base.
- La utilización de interfaces y contratos de NLTK para homogeneizar los formatos de entrada/salida.
- La utilización del *tagset* EAGLES como *tagset* único, siendo éste un *tagset* sumamente descriptivo.

En este capítulo se ven detalles de implementación de la solución diseñada, se ve en particular como se relaciona la biblioteca con la plataforma elegida: Python/NLTK.

4.1. Integración con NLTK

Como se vio en 2.3.2, NLTK es un toolkit para Python (actualmente soporta Python 2.x y Python 3.x) que provee una múltitud de herramientas para PLN)

NLTK expone un conjunto de interfaces que permiten trabajar con variados componentes y algoritmos de forma casi transparente, permitiendo al programador enfocar su atención en la resolución del problema, en lugar de verse forzado a invertir excesivo tiempo en su aprendizaje.

A los efectos de los objetivos planteados, hay tres interfaces de NLTK que son relevantes: `TokenizerI`, `TaggerI`, y `ParserI`.

La interfaz `TokenizerI` define el contrato para los *tokenizers* dentro de NLTK, con la operación `tokenize`, la cual toma como parámetro el *string* a tokenizar, y devuelve una lista de *tokens*.

```
1 class TokenizerI(object):
2 ...
3     def tokenize(self, s):
4         """
5         Return a tokenized copy of *s*.
6         :rtype: list of str
7         """
```

La interfaz `TaggerI` corresponde a los etiquetadores gramaticales, y provee la operación `tag`, la cual recibe una lista de *tokens*, y devuelve una lista de tuplas, cada una conteniendo el *token* y el POS-*tag* asociado. Para mantener compatibilidad, se implementó esta operación. Sin embargo se proveyeron métodos adicionales que devuelven los *tokens*, sus POS-*tags* correspondientes y un conjunto de información adicional en un formato estándar que será descrito más adelante.

```
class TaggerI(object):
...
    def tag(self, tokens):
        """
        Determine the most appropriate tag sequence for
        the given token sequence, and return a
        corresponding list of tagged
        tokens.  A tagged token is encoded as a tuple
        ``(token, tag)''.
        :rtype: list(tuple(str, str))
        """
```

Finalmente, la interfaz `ParserI` describe a un analizador sintáctico en NLTK. Contiene la operación `parse`, ésta recibe como parámetro un *string*, y devuelve una estructura arborescente representando el árbol de análisis sintáctico. Análogamente al caso anterior, en adición a la operación requerida por la interfaz se agregaron algunas operaciones extra.

```
class ParserI(object):
...
    def parse(self, sent, *args, **kwargs):
        """
        :return: An iterator that generates parse trees
         for the sentence.
        When possible this list is sorted from most
        likely to least likely.

        :param sent: The sentence to be parsed
        :type sent: list(str)
        :rtype: iter(Tree)
        """
```

Al contar con estas interfaces, el programador se ve abstraído del problema de la forma de invocación de, por ejemplo, MaltParser. Cuenta con la garantía de que al cumplir con la interfaz `ParserI`, el *parser* se invoca de una forma estándar bien documentada por NLTK, al igual que cualquier otra herramienta escondida tras dicha interfaz.

Si los componentes a desarrollar implementan estas interfaces se comportarán como cualquier otro elemento de NLTK del mismo tipo, es decir que si para alguna operación provista por el *toolkit* se requiriese un `TaggerI` como parámetro, un `TaggerI` implementado por la biblioteca desarrollada será tan válido como uno incluido como parte de NLTK.

4.2. Organización de la biblioteca

Con conocimiento de las clases y componentes relevantes de NLTK necesarios para mantener compatibilidad, y habiendo diseñado la forma de alcanzar los objetivos planteados, se está en condiciones de describir los diferentes componentes que implementan la solución, como se organizan y como se utilizan.

4.2.1. Tokenizers

Como se mencionó anteriormente, la interfaz que define a los *tokenizers* es `TokenizerI`, ésta especifica una única operación `tokenize`, la cual toma una parámetro del tipo `string` (el texto de entrada a tokenizar), y devuelve una lista de *tokens*.

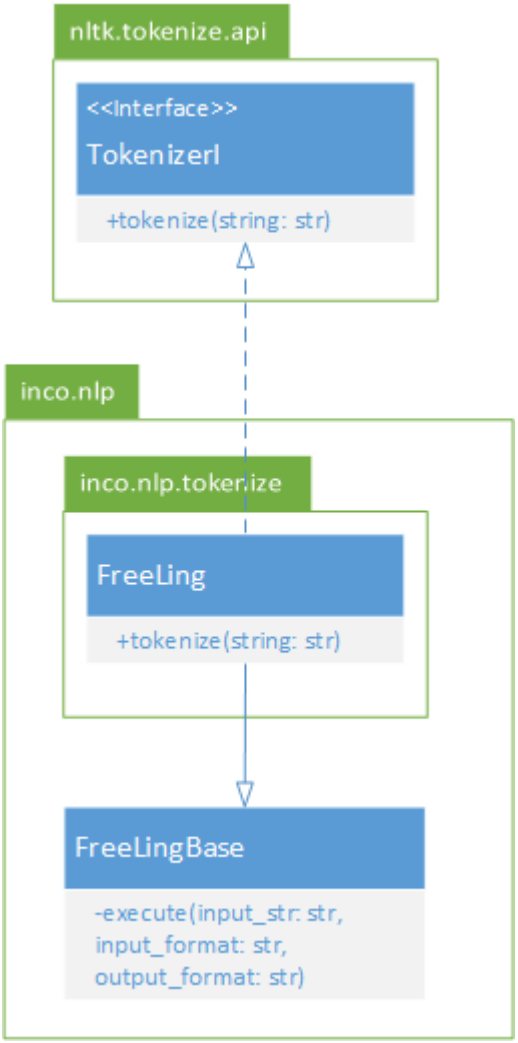

Figura 7: Diagrama de clases de *tokenizers*

Para los *tokenizers* de la biblioteca, se desarrolló únicamente un *wrapper* para FreeLing, cuando éste opera en esa modalidad. Dado que el código para la ejecución de FreeLing en los modos de *tokenizer*, etiquetador, y analizador sintáctico es el mismo, existe una clase base llamada `FreeLingBase` que encapsula toda la lógica relativa a la comunicación con su binario. En la figura 7 se representa la jerarquía de clases que implementan esta categoría de componentes.

Ejemplo de uso:

```
1    import inco.pln.tokenize.freeling
2    ...
```

```
3     entrada = u"En el tramo de Telefonica, un toro
4     descolgado ha creado peligro tras embestir
5     contra un grupo de mozos."
6
7     tokenizer = inco.pln.tokenize.freeling
8                     .FreeLing(ruta_a_binario)
9
10    # La variable tokens contiene el string
11    # de entrada tokenizado,
12    # es una lista de strings.
13    tokens = tokenizer.tokenize(entrada)
14
15    print tokens
```

La salida es:
`[u'En', u'el', u'tramo', u'de', u'Telefonica', u',', u'un', u'toro', u'descolgado', u'ha', u'creado', u'peligro', u'tras', u'embestir', u'contra', u'un', u'grupo', u'de', u'mozos', u'.']`

4.2.2. Etiquetadores gramaticales

Los etiquetadores gramaticales externos que se buscó integrar fueron FreeLing y TreeTagger, a continuación se describen las consideraciones tomadas durante el desarrollo de los *wrappers*.

En adición a la implementación de la interfaz `TaggerI` para los etiquetadores gramaticales, se agregaron varias operaciones de conveniencia para facilitar aún más su uso dentro de NLTK. Más específicamente se incluyó un modo de etiquetado crudo, y un modo de etiquetado extendido.

Como se vio en 4.1, la interfaz `TaggerI` provee una operación `tag(tokens: list(str))`, recibiendo por parámetro la lista de *tokens* a etiquetar. Sin embargo, resultaría útil una operación que, dada una tira de caracteres, la devolviera etiquetada, realizando la operación de *tokenization* ella misma, esto fue denominado etiquetado crudo (*raw*). De esta forma, los etiquetadores gramaticales de la biblioteca contienen la operación `raw_tag(string: str)`, la cual dada una tira de caracteres, devuelve una lista de *tokens* etiquetados gramaticalmente. Para realizar esto, es posible pasarle a los etiquetadores de la biblioteca por constructor una instancia de `TokenizerI`, la cual será utilizada para el proceso de *tokenization*. En caso de no pasar un `TokenizerI`, se utilizará como respaldo (*fallback*) el `word_tokenize` nativo de NLTK.

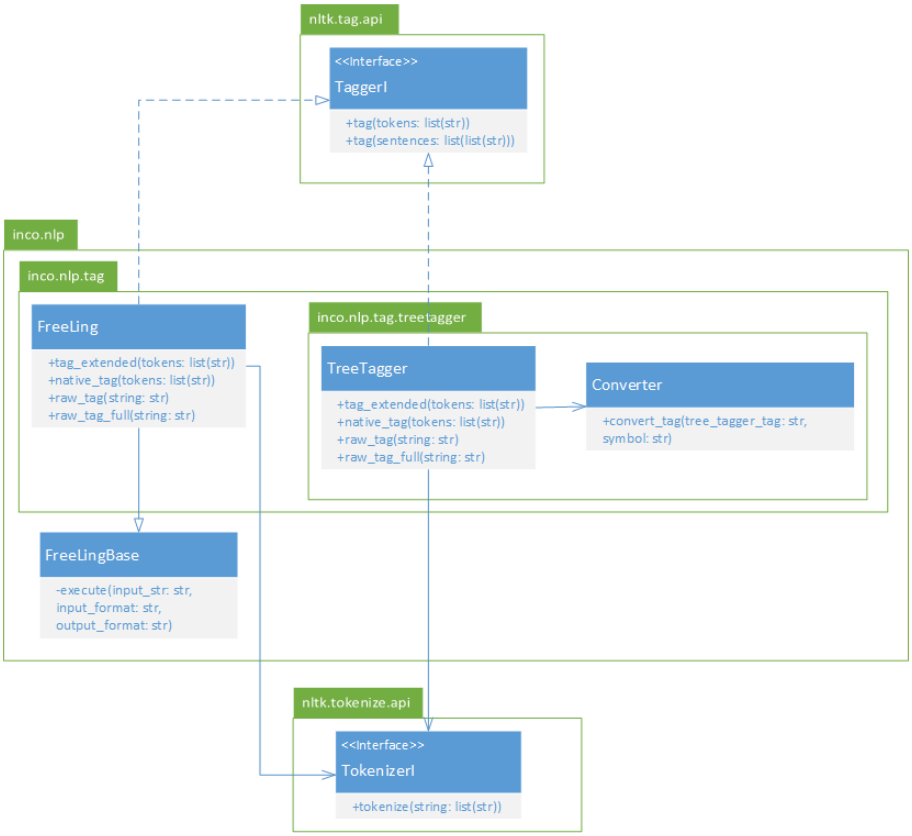

Figura 8: Diagrama de clases de etiquetadores

Como última adición se incluyó el modo de etiquetado extendido ya descrito, tiene como finalidad capitalizar en el hecho de que diferentes etiquetadores gramaticales proveen diferentes grados de información. En el modo extendido, la salida en lugar de ser una lista de tuplas *token*/etiqueta, es un diccionario el cual contiene toda la información que puede dar el etiquetador, tales como ser la probabilidad de una etiqueta, el lema, etc. Un diccionario es un formato de salida extensible que permite procesar la información de forma predecible, en lugar de tener que modificar la estructura de datos de salida para cada herramienta.

Para la unificación de *tagsets*, se decidió utilizar el *tagset* EAGLES, como se explicó en 3.4, para conseguir esto, en el caso de TreeTagger fue necesario transformar los *tags* nativos del etiquetador a los del *tagset* elegido, para esto se creó una clase `Converter` dentro del módulo de TreeTagger, responsable de la traducción al *tagset* EAGLES.

En la figura 8 se puede ver la jerarquía de clases correspondiente a los etiquetadores gramaticales.

```
1 class Converter:
2     @staticmethod
3     def convert_tag(tree_tagger_tag, symbol=None):
4         """
5         Translates a TreeTagger tag into a pair
6         (coarse-tag, pos-tag) using EAGLES tag set
```

```
7           for spanish.
8           :param tree_tagger_tag: TreeTagger tag
9           :type tree_tagger_tag: str
10          :param symbol: Symbol that corresponds to
11              the supplied tag.
12          :type symbol: str
13          :return: Pair (coarse-tag, pos-tag)
14              using EAGLES tag set.
15          :rtype: tuple(str, str)
16          """
```

Ejemplo de uso:

```
1   import inco.pln.tokenize.freeling
2   import inco.pln.tag.treetagger.treetagger
3   ...
4   entrada = u"En el tramo de Telefonica, un toro
5   descolgado ha creado peligro tras embestir
6   contra un grupo de mozos."
7
8   tokenizer = FreeLing(ruta_a_binario_freeling)
9   tagger = TreeTagger(ruta_a_binario_treetagger)
10
11  tokens = tokenizer.tokenize(entrada)
12
13  texto_taggeado = tagger.tag(tokens)
14
15  print texto_taggeado
```

La salida del ejemplo es:

```
[(u'En', 'SP000'), (u'el', 'DA0000'), (u'tramo', 'NC00000'), (u'de',
'SP000'), (u'Telefonica', 'NC00000'), (u',', 'Fc'), (u'un', 'DA0000'),
(u'toro', 'NC00000'), (u'descolgado', 'V0PS000'), (u'ha', 'V000000'),
(u'creado', 'V0PS000'), (u'peligro', 'NC00000'), (u'tras', 'SP000'),
(u'embestir', 'V0N0000'), (u'contra', 'SP000'), (u'un', 'DA0000'),
(u'grupo', 'NC00000'), (u'de', 'SP000'), (u'mozos', 'NC00000'), (u'.',
'Fp')]
```

4.2.3. Analizadores sintácticos

Se integraron los analizadores sintácticos FreeLing, MaltParser y el Stanford Shift-Reduce Parser.

Como se hizo para los etiquetadores, se agregaron algunas operaciones de conveniencia. La operación `raw_parse` permite el análisis sintáctico en modo "crudo", o sea, en lugar de recibir por parámetro una lista de tuplas *token*/etiqueta, se recibe una tira de caracteres y ésta es tokenizada, etiquetada, y analizada sintácticamente. Para realizar las operaciones de tokenización y etiquetado, se pasa por constructor una instancia de `TaggerI`, en caso de no pasar un `TokenizerI`, se utiliza el `word_tokenize` como *fallback*. Finalmente, la operación `tagged_parse` permite analizar sintácticamente una oración que

ya fue previamente etiquetada, recibiendo por parámetro una lista de tuplas *token*/etiqueta.

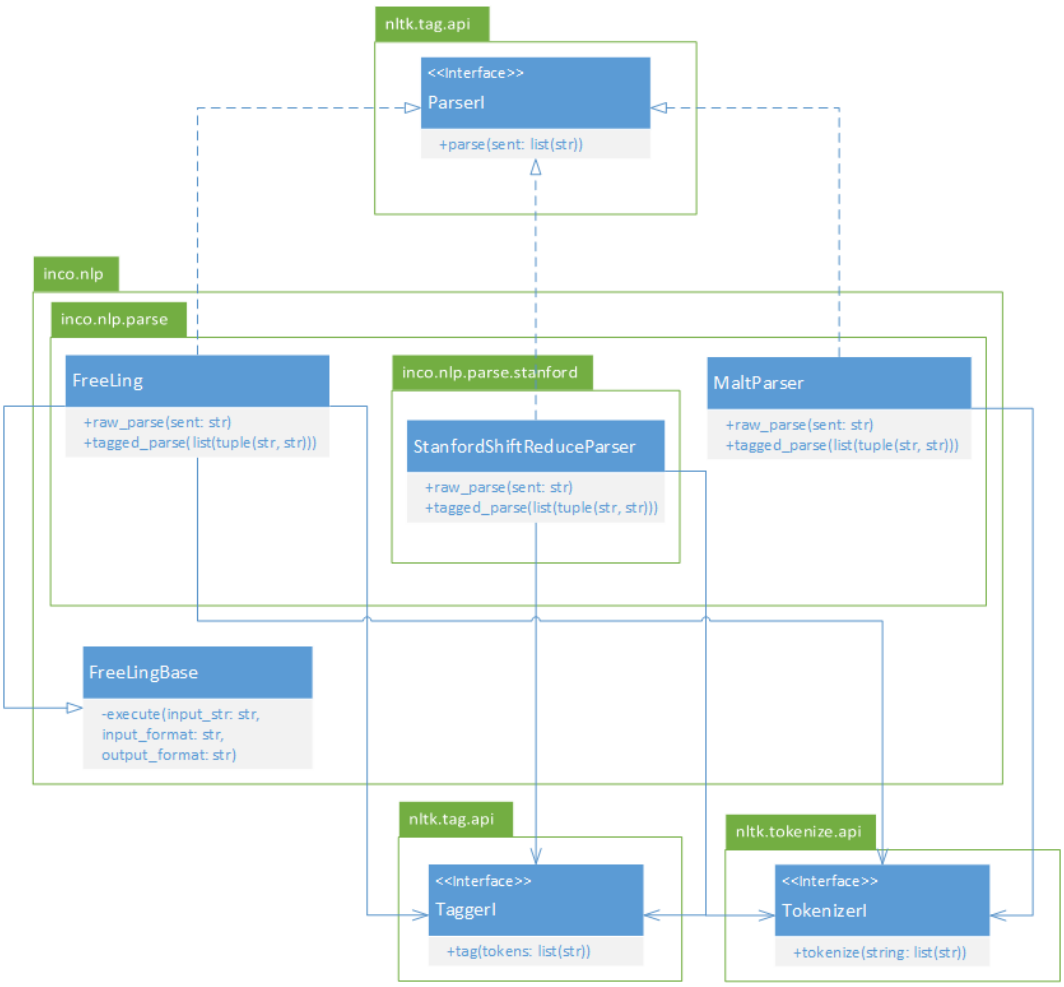

Figura 9: Diseño de analizadores sintácticos

Ejemplo de uso completo, usando en *pipeline* tres herramientas diferentes:

```
    import inco.pln.tokenize.treetagger
    import inco.pln.tag.freeling
    import inco.pln.parse.stanford
    ...
    entrada = u"En el tramo de Telefonica, un toro
    descolgado ha creado peligro tras embestir
    contra un grupo de mozos."

    tokenizer = FreeLing(ruta_a_binario_freeling)
    tagger = TreeTagger(ruta_a_binario_treetagger)
    parser = StanfordShiftReduceParser(ruta_a_jar,
        ruta_a_modelo_espanol, tagger)

    tokens = tokenizer.tokenize(entrada)
```

```
15 |     arbol_sintactico = parser.parse(tokens)
```

En la figura 9 se representa la estructura de clases relevante al componente de los analizadores sintácticos.

4.3. Consideraciones generales

Los objetivos buscados por la biblioteca pueden resumirse en el de integrar nuevas herramientas hasta el momento no disponibles a NLTK. Dado que el *toolkit* existe sobre Python, es necesario contar con una forma de invocar a éstas nuevas adiciones desde el ambiente de ejecución mencionado. En casi todos los casos, las herramientas presentaron ejecutables utilizables desde la línea de comandos, con la única salvedad del Stanford Shift-Reduce Parser. Para éste no fue posible contar con un ejecutable que permitiera su invocación mediante la línea de comandos del sistema operativo, de modo que fue necesario implementar una clase en Java que pudiera ser utilizada de esta forma, recibiendo en su operación `main` los parámetros necesarios y actuando de forma análoga a los demás casos encontrados.

Debido a que Python no provee la construcción de interfaz (ni de clase abstracta), éstas en realidad son clases "simples" que realizan un camino alternativo para simular estos mecanismos. Se definen las operaciones que comprenden la interfaz, y en el método se lanza una excepción `NotImplementedError`, forzando al desarrollador a sobreescribirlas.

4.4. Resumen

Se llega a la etapa de implementación con un diseño propuesto en el capítulo 3 que resuelve los problemas planteados en 1.4. Estos, se recuerda, son:

- Uniformización de ambientes de ejecución
- Homogeneización de formatos de entrada/salida
- Unificación de tagsets

Según se propone en el diseño, la utilización de la plataforma Python unifica efectivamente los entornos de ejecución, permitiendo al usuario utilizar un ambiente único que cuenta con mecanismos que habilitan la utilización de ambientes dispares, como ser la JVM o nativamente contra el propio sistema operativo, de forma transparente, simplificando significativamente el uso de las herramientas. Desde el punto de vista de la implementación, esto significa que la biblioteca desarrollada existe como un conjunto de módulos de Python los cuales contienen código que se comunica con los diferentes ambientes utilizados a través del sistema operativo, esto abstrae por completo al usuario de la existencia de dichos ambientes; dicho de otra forma, el usuario únicamente ve Python, nunca se entera siquiera que, por ejemplo, la JVM está involucrada siquiera en la ejecución de la herramienta.

La utilización de NLTK y sus interfaces como base para la biblioteca resuelven efectivamente el objetivo de la homogeneización de los formatos de

entrada/salida, dado que la conformación a los contratos del *toolkit* exige que se respeten formatos establecidos, de esta manera se obtienen familias de herramientas que se comportan de maneras similares (ejemplo, todos los etiquetadores gramaticales reciben por parámetro la lista de *tokens* a etiquetar en una estructura de lista). Esto, desde la perspectiva de la implementación, significa la extensión de las interfaces `TokenizerI`, `TaggerI` y `ParserI`, para la integración de las herramientas.

El objetivo de la unificación de los *tagsets* responde a la necesidad de aumentar la interoperabilidad entre las distintas herramientas. La situación encontrada es que distintas herramientas utilizan diferentes *tagsets* (ejemplo: FreeLing utiliza EAGLES, y TreeTagger utiliza un *tagset* propio), haciéndolas incompatibles entre sí. El diseño propuesto establece el empleo del *tagset* EAGLES como *tagset* único. Al momento de la implementación esto se traduce en la necesidad de convertir los *tags* a EAGLES según sea necesario, en el caso particular de las herramientas elegidas para integración se agregó una capa de conversión a la clase *wrapper* de TreeTagger, dado que es el único aplicativo que no utiliza EAGLES. De esta forma se obtiene la compatibilidad de salidas de las distintas herramientas, por ejemplo es posible utilizar TreeTagger para etiquetar y FreeLing para parsear.

5. Conclusiones y Trabajo Futuro

En este proyecto se plantea como objetivo la simplificación del uso de varias de las herramientas utilizadas por el Grupo de Procesamiento del Lenguaje Natural del InCo, éstas son todas invocadas mediante ambientes de ejecución diferentes (máquinas virtuales, nativas, intérpretes), reciben parámetros de entrada y devuelven la salida en formatos diferentes e incompatibles, y los *tagsets* utilizados difieren, dificultando aún más su uso y la explotación de los resultados.

Tras un análisis de las herramientas y en particular de NLTK, se encontró que el *toolkit* ya cumple parcialmente con algunos de los objetivos, por ejemplo abstrayendo las complejidades de las herramientas detrás de interfaces con contratos simples y bien definidos. De esta forma, se tomó la decisión de utilizar como entorno de ejecución a Python, capitalizando en la facilidad para el desarrollo rápido, y tomar como base los contratos de NLTK, efectivamente extendiendo el *toolkit*. Entre las carencias encontradas en el código existente está la falta de un tagset único y la ausencia de algunas herramientas muy utilizadas (por ejemplo Freeling).

La biblioteca desarrollada cumple con todos los objetivos propuestos, corrigiendo las faltas detectadas y utilizando a NLTK como base y plantilla sobre la cual se la construyó, obteniendo como beneficio adicional el mantener compatibilidad con el *toolkit*.

El diseño del proyecto resultó simple una vez se decidió tomar como fundación a NLTK, su extensiva documentación colaboró significativamente al desarrollo, acortando tiempos y reduciendo esfuerzos. De igual forma la implementación se vio beneficiada por el código del *toolkit*, pudiendo tomar como referencia algunas clases que responden a necesidades similares a las planteadas por el proyecto. La integración de las herramientas se consiguió sin mayores dificultades dada la disponibilidad de buena documentación en todos los casos salvo en uno: el Stanford Shift-Reduce Parser. En el caso mencionado, la documentación fue muy escasa, y se tuvo que recurrir a un código de ejemplo, pobremente documentado, e inferir a partir de allí el funcionamiento de la herramienta, forzando en una instancia a recurrir a los desarrolladores del *parser* para realizar una consulta[37]. Además, no es posible utilizar el *parser shift-reduce* desde la línea de comandos, a diferencia del *lexicalized parser* de Stanford, el cual es accesible desde un archivo de lotes (*batch*). En el caso de este *parser*, hubo que crear un *wrapper* adicional en Java, el cual a su vez fue utilizado desde el *wrapper* en Python.

5.1. Trabajo Futuro

El trabajo presentado admite mejoras, a continuación se detallan mejoras posibles:

- Invocación directa: Las APIs desarrolladas no permiten la ejecución de

[37] http://stackoverflow.com/questions/28548053/get-typeddependencies-using-stanfordparser-shift-reduce-parser, último acceso: 11/07/2015

las herramientas de forma completamente nativa, es decir, no es posible ejecutar FreeLing y obtener un archivo de texto resultado con la salida nativa de FreeLing, la biblioteca permite como mucho obtener las etiquetas nativas del etiquetador (en el caso de TreeTagger), pero no la salida con su formato original.

- Pasaje flexible de parámetros: Las herramientas integradas admiten una cantidad interesante de parámetros de entrada que permite la configuración a un nivel muy fino, usualmente estos parámetros son pasados mediante archivos de configuración, pero en ocasiones es posible utilizarlos mediante banderas desde la línea de comandos (notar que, sin embargo, esto no siempre es así, el analizador sintáctico Shift-Reduce de Stanford no es invocado mediante línea de comando, debiendo en su lugar ser ejecutado desde la JVM). Se podrían flexibilizar las APIs actuales para permitir la ejecución con una cantidad no determinada de parámetros de entrada, esto involucraría modificar los *wrappers*, en particular el caso del analizador Stanford Shift-Reduce sería, de las herramientas actuales, el que requeriría mayor esfuerzo y modificaciones.

- *Pull request* a NLTK: La biblioteca está diseñada e implementada extendiendo las interfaces de NLTK, de esta forma, sería algo muy positivo la posibilidad de integrarla al código fuente del *toolkit*, esto se logra haciendo un *pull request* al repositorio oficial de NLTK, previa una revisión del código por algún responsable de la integración, podría pasar a ser parte del *toolkit*, incrementando su utilidad y beneficiándose de más desarrolladores extendiendo el trabajo actual. Previo al *pull request*, es neceser realizar una inspección más minuciosa del código, asegurarse de que se conforma con los estándares de codificación de NLTK, y que se usan todas las funcionalidades internas que podrían haber sido, inadvertidamente, reimplementadas en el presente trabajo.

- Instalador: El trabajo actual funciona como una biblioteca de desarrollo, como tal es responsabilidad del desarrollador la obtención y configuración de las herramientas, para que puedan ser utilizadas por la biblioteca. Sería útil contar con un instalador, que sea responsable de acceder a los sitios de las herramientas, descargue, configure, y las deje disponibles para utilizar desde la biblioteca.

- Multiplataforma: A pesar de que el trabajo actual está desarrollado sobre Python, haciéndolo, en teoría, multiplataforma, no está probado en entornos similares a UNIX (en particular Linux y Mac OS X). Aunque la mayoría del código es independiente de la plataforma (todo el relacionado con el procesamiento de las entradas y salidas), las herramientas en sí no lo son, sus formas de configuración y ejecución pueden diferir entre una plataforma y otra. Por ejemplo, la interacción con el *shell* del sistema desde Python, utilizado para la invocación desde línea de comandos, puede tener sutilezas en otros sistemas, haciendo que la biblioteca

funcione incorrectamente. El desarrollo actual está probado en el sistema operativo Windows.

A. Uso de la biblioteca

Las herramientas integradas en la biblioteca se encuentran en los paquetes Python `inco.pln.tokenize` (para herramientas que realizan *tokenization*), `inco.pln.tag` (para herramientas que realizan POS-*Tagging*) e `inco.pln.parse` (para los analizadores sintácticos), dentro de estos paquetes residen los módulos de TreeTagger, FreeLing, MaltParser y Stanford SR. De esta forma, para utilizar por ejemplo el *tagger* de FreeLing hay que importar el paquete `inco.pln.tag.freeling`. Todas las clases del proyecto son esencialmente *wrappers* que abstraen detalles del uso de las herramientas, de modo que el uso de cualquier herramienta requiere su instalación previa, luego para utilizar el *wrapper* es necesario pasarle por parámetro durante su construcción la ruta en el *filesystem* donde se ubica. A modo de ejemplo, tómese el caso en el cual se desea utilizar TreeTagger para etiquetar un conjunto de *tokens*, su ejecutable (un archivo de lotes llamado `tag-spanish.bat`, en Windows) se encuentra en "`c:/users/pln/tools/treetagger/bin/tag-spanish.bat`", el código Python apropiado para utilizarlo es el siguiente:

```
1  from inco.pln.tag.treetagger import TreeTagger
2
3  ...
4
5  ruta_a_treetagger =
6     "c:/users/pln/tools/treetagger/bin/tag-spanish.bat"
7
8  tagger = TreeTagger(ruta_a_treetagger)
9
10 tagger.tag(tokens)
```

Supóngase ahora que no se dispone de los tokens, si no que se cuenta únicamente con el *string*, en ese caso se puede utilizar la operación `raw_tag`:

```
1  from inco.pln.tag.treetagger import TreeTagger
2
3  texto = "Esto es un texto de prueba, sin tokenizar"
4
5  ruta_a_treetagger =
6  "c:/users/pln/tools/treetagger/bin/tag-spanish.bat"
7
8  tagger = TreeTagger(ruta_a_treetagger)
9
10 tagger.raw_tag(texto)
```

Lo que hace el método `raw_tag` es realiza *tokenization* utilizando un *tokenizer* externo, ya sea uno pasado por el programador, o uno por defecto de NLTK. En caso de ejecutarlo como en el código de ejemplo, se está usando el `word_tokenize` de NLTK. Si se desea usar un *tokenizer* externo, hay que pasarlo por constructor al *tagger*:

```
1  from inco.pln.tokenize.freeling import FreeLing
2  from inco.pln.tag.treetagger import TreeTagger
```

```
3
4  texto = "Esto es un texto de prueba, sin tokenizar"
5
6  ...
7
8  ruta_a_treetagger =
9  "c:/users/pln/tools/treetagger/bin/tag-spanish.bat"
10
11 freeling_tokenizer = FreeLing(ruta_a_freeling)
12 tagger = TreeTagger(ruta_a_treetagger,
13                     freeling_tokenizer)
14
15 tagger.raw_tag(texto)
```

Imagínese que ahora se quiere agregar análisis sintáctico con Stanford SR al texto de ejemplo. Todos los parsers requieren que se le pase por constructor una instancia de un `TaggerI` para que realice su operativa, así como la ruta a su modelo para el español. Sea un caso donde se desea utilizar a FreeLing como *tokenizer*, a TreeTagger como *tagger*, y a Stanford SR como *parser*, el código sería muy similar a:

```
1  from inco.pln.tokenize.freeling import FreeLing
2  from inco.pln.tag.treetagger import TreeTagger
3  from inco.pln.parse.stanford_shift_reduce
4      import StanfordShiftReduceParser
5
6  texto = "Esto es un texto de prueba, sin tokenizar"
7
8  ...
9
10 ruta_a_treetagger =
11 "c:/users/pln/tools/treetagger/bin/tag-spanish.bat"
12 ruta_a_freeling =
13 "c:/users/pln/tools/freeling/bin/analyzer.exe"
14 ruta_a_stanford_sr =
15 "c:/users/pln/tools/stanfordsr/stanford-parser.jar"
16 ruta_a_modelo_stanford_sr =
17 "c:/users/pln/tools/stanfordsr/
18     stanford-srparser-2014-10-23-models.jar"
19
20 freeling_tokenizer = FreeLing(ruta_a_freeling)
21 tagger = TreeTagger(ruta_a_treetagger,
22                     freeling_tokenizer)
23 parser =
24     StanfordShiftReduceParser(ruta_a_stanford_sr,
25                               ruta_a_modelo_stanford_sr)
26
27 print parser.parse(texto)
```

Este código tiene la siguiente salida:

```
    (ROOT (sentence (spec (grup.cc (NCFSV00 E) (NCFSV00 s)) (NCFSV00 t)) (grup.cc (CC o) (NCFSV00
e)) (s.a (grup.a (NCFSV00 s))) (S (sadv (grup.adv (NCFSV00 u)))) (sn (spec (grup.cc (S (sadv (grup.adv
(NCFSV00 n)))) (NCFSV00 t)) (NCFSV00 e)) (grup.nom (S (NCFSV00 x) (s.a (grup.a (NCFSV00 t))) (S
(sadv (grup.adv (CC o)))) (sn (spec (grup.cc (S (sadv (grup.adv (NCFSV00 d)))) (NCFSV00 e)) (NCFSV00
p)) (grup.nom (NCFSV00 r))) (grup.verb (NCFSV00 u)) (sadv (grup.adv (NCFSV00 e) (NCFSV00 b))) (sn
(grup.nom (NCFSV00 a))) (sentence (Fc ,) (S (sadv (grup.adv (NCFSV00 s)))) (spec (grup.cc (S (sadv
(grup.adv (Fz i)))) (NCFSV00 n)) (NCFSV00 t)) (CC o) (S (sadv (grup.adv (NCFSV00 k)))) (sn (spec
(grup.cc (S (sadv (grup.adv (CC e)))) (NCFSV00 n)) (Fz i)) (grup.nom (NCFSV00 z))) (grup.verb (NCFSV00
a)) (sn (grup.nom (NCFSV00 r)))))))))
```

De esta forma se puede apreciar como se pueden combinar tres diferentes herramientas de forma completamente desatendida en lo que respecta a los tres objetivos planteados inicialmente: los ambientes de ejecución resultan transparentes al programador, ya que el sólo ve Python; los formatos de entrada y salida son a todos los efectos las descritas por los contratos de NLTK, no es necesario preocuparse que la salida de TreeTagger sea compatible con la entrada de Stanford SR; y finalmente los *tagsets* usados por las tres herramientas son para el usuario uno solo, el *tagset* EAGLES.

A.1. Frontend de pruebas

Además de la API, se desarrolló un *frontend* de pruebas a pedido de los tutores, a modo de demostrar el uso de la biblioteca, y de tener un punto de acceso para la rápida utilización de las herramientas, éste procura ser un ejemplo de utilización pero puede ser usado como producto final.

El *frontend* tiene tres pestañas: Tokenize, Tag y Parse, para realizar *tokenization*, POS-*Tagging* y análisis sintáctico, respectivamente. En todas ellas hay un campo de texto superior que permite ingresar el texto a procesar, en adición hay un botón "*Read from file*" que permite, como su título indica, abrir un archivo y leerlo por completo.

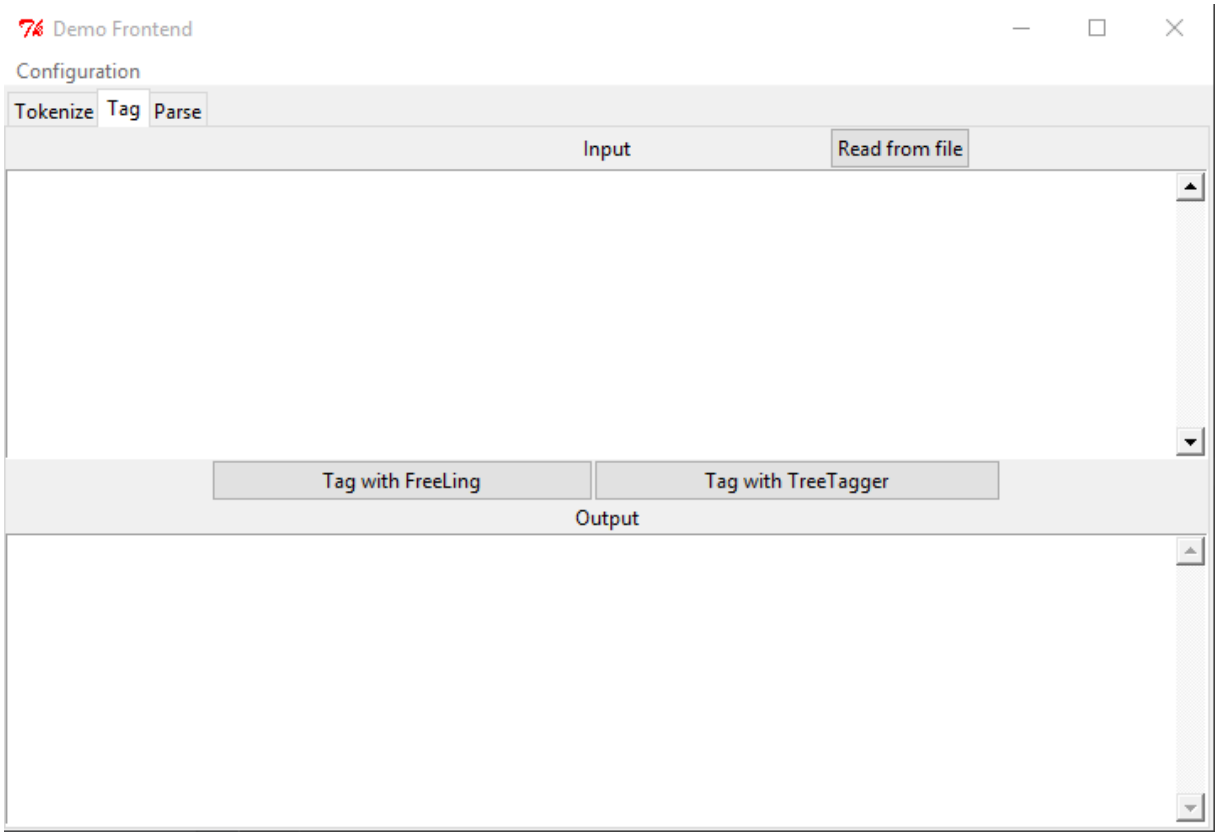

Figura 10: Pestaña de POS-Tagging

Desde la pestaña de Tokenize se puede utilizar únicamente a FreeLing, siendo la única herramienta integrada que permite tokenizar. Desde Tag se puede utilizar tanto FreeLing como TreeTagger. Finalmente, en Parse se puede utilizar FreeLing, MaltParser y Stanford Shift-Reduce Parser. Si se intenta realizar una operación sin configurar previamente la ruta al binario y/o modelos, se recibirá un error de ejecución.

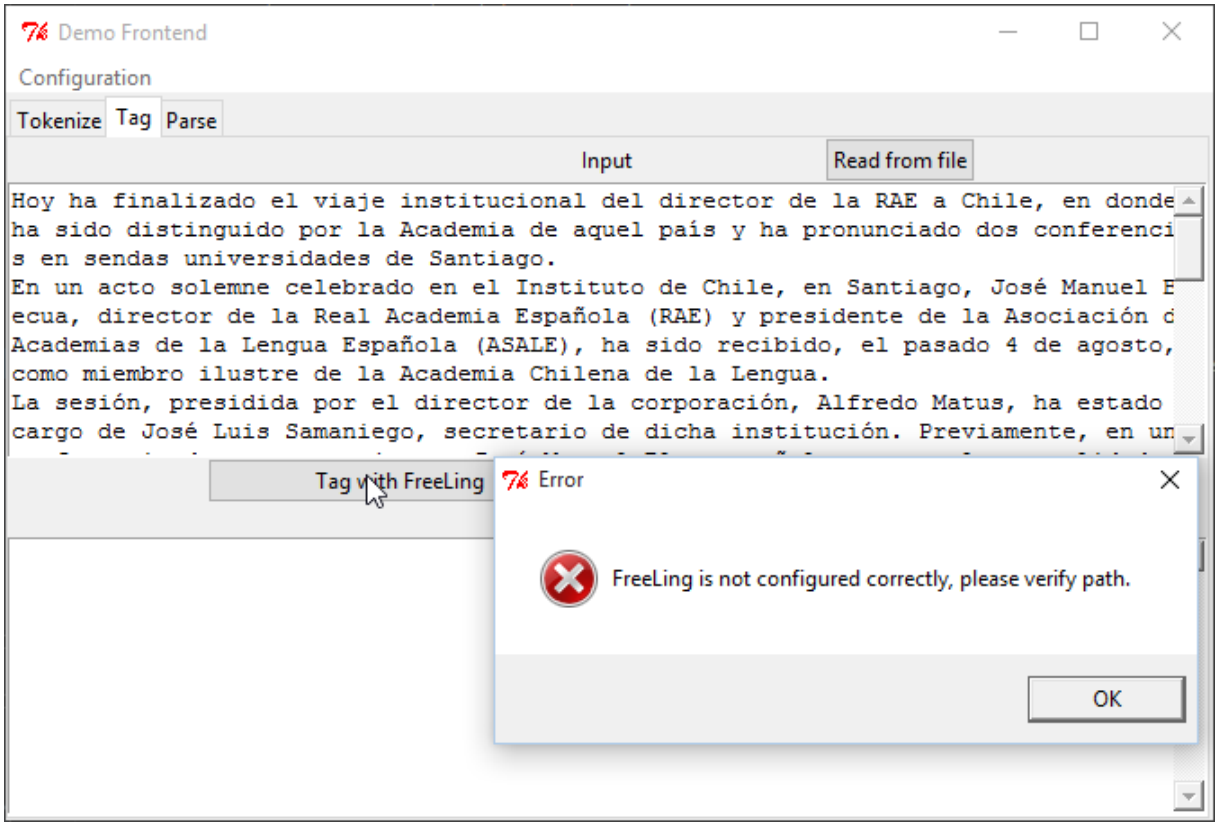

Figura 11: Error de configuración de FreeLing

Para configurar las diferentes herramientas, en la *toolbar* debajo del título está alojado el botón "*Configuration*" que permite establecer todas las rutas.

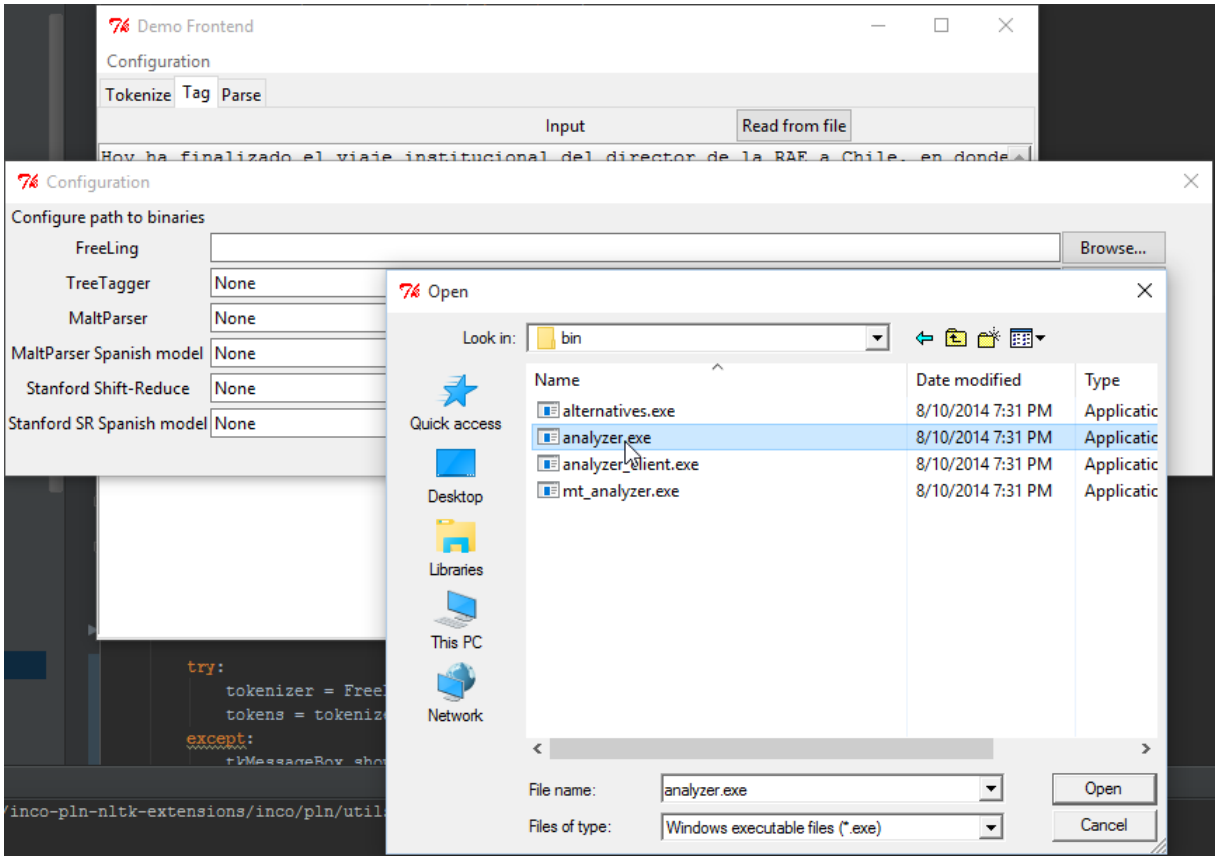

Figura 12: Configurando FreeLing

Una vez configurada la herramienta, está lista para ser usada.

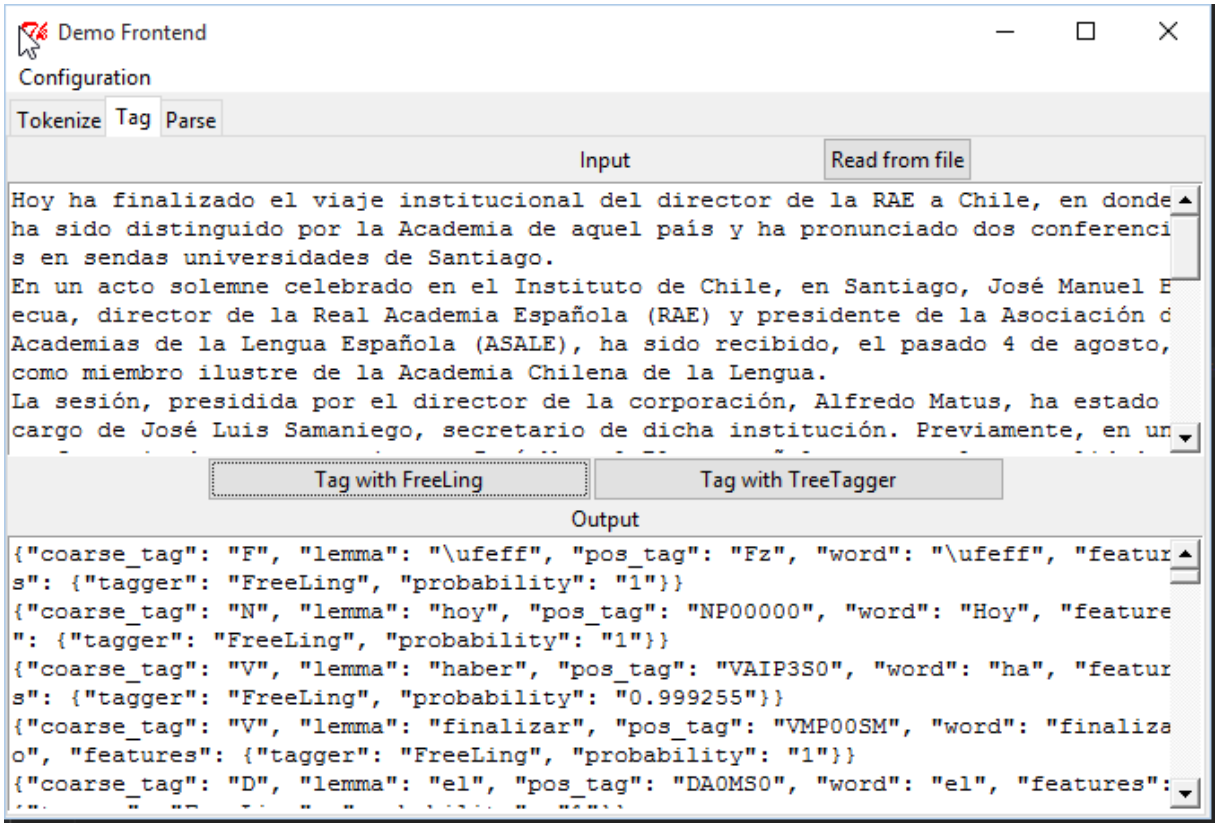

Figura 13: Utilizando FreeLing para realizar POS-Tagging

La pantalla de Tokenize es muy simple y similar a la de Tag, pero la de Parse es significativamente más compleja, desde ella se puede elegir tanto el *parser* como el POS-*Tagger* a utilizar y, una vez generado el árbol sintáctico, se puede convertir éste a formato DOT[46], o renderizarlo en pantalla mediante NLTK.

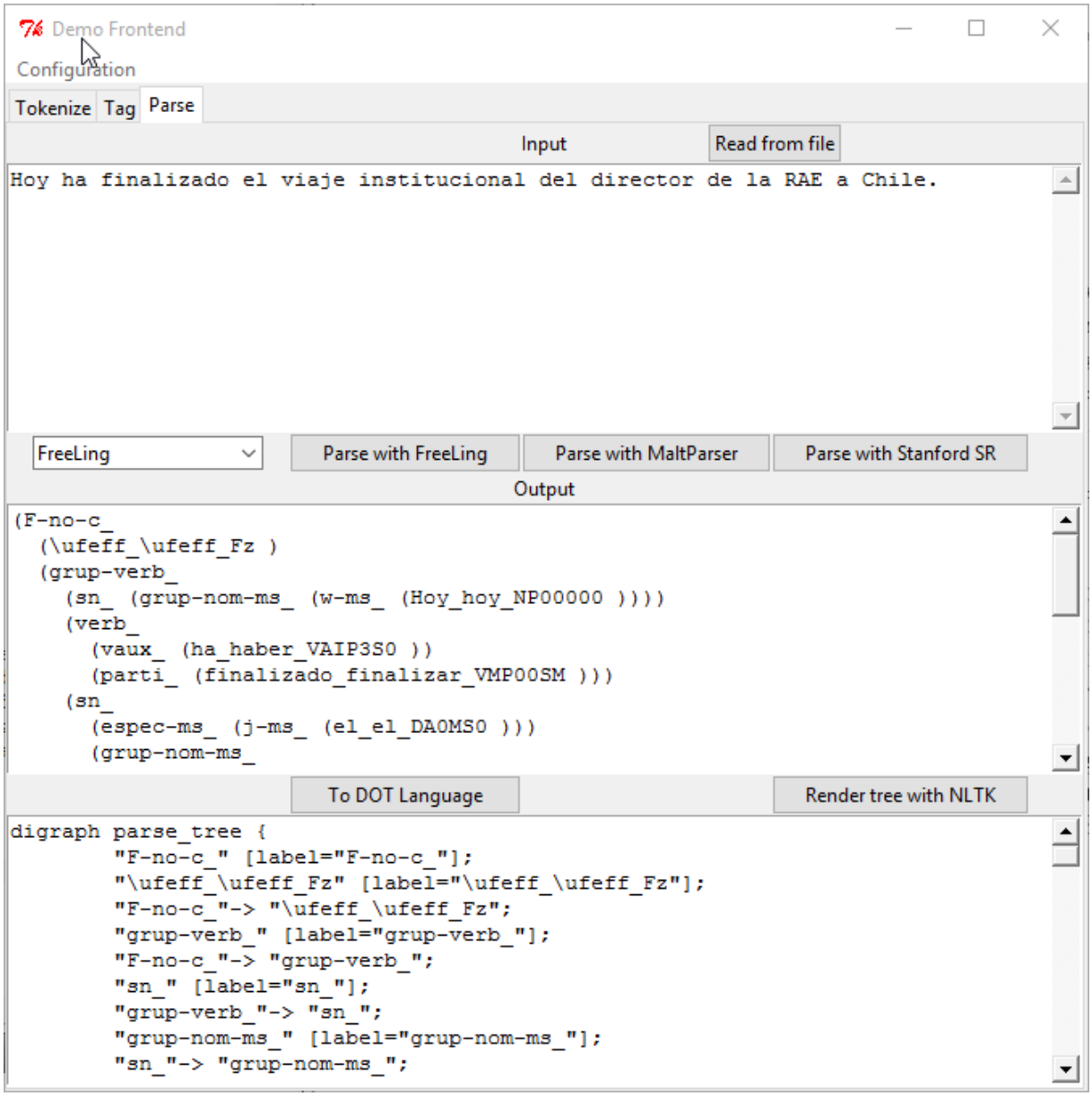

Figura 14: Realizando análisis sintáctico

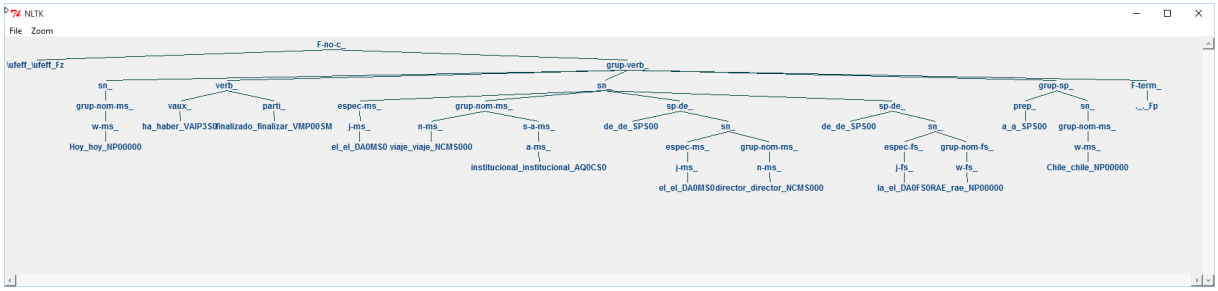

Figura 15: Árbol renderizado con NLTK

Glosario

Accuracy Medida para sistemas de extracción de información, también conocida como Precisión se define de la siguiente forma: # de respuestas correctas dadas por el sistema / # de respuestas dadas por el sistema.

AI-completo Familia de problemas que básicamente requieren todo el conocimiento que posee un humano, y las habilidades para usarlo. Resolverlos equivale a resolver el problema de crear una inteligencia artificial tan inteligente como un humano..

ALPAC Automatic Language Processing Advisory Committee.

API Application Programming Interface.

chunking Ver shallow-parsing.

corpus Conjunto grande de textos. Pueden ser anotados, esto implica que cada palabra contiene información léxica, como ser su part-of-speech, su forma base, etc. Los corpus anotados son útiles para el entrenamiento de herramientas de PLN, las cuales aprenden en base a éstos..

F-Score Métrica que balancea recall y accuracy de la siguiente forma: $F = ((\beta^2+1)PR)/(\beta^2 P+R)$ con β siendo un parámetro que otorga más peso a Recall o a Accuracy.

GATE General Architecture for Text Engineering.

IDE Integrated Development Environment.

InCo Instituto de Computación de la Universidad de la República.

integrated development environment Ambiente de desarrollo integrado, software que provee múltiples herramientas de desarrollo diseñadas para trabajar en conjunto, como ser: editor de texto con resaltado de sintaxis, debugger, colaboración de equipos, reporte de defectos, etc.

JVM Java Virtual Machine.

Labeled Attachment Score Medida para parsers de dependencias, se define como el porcentaje de tokens para los cuales el sistema predijo correctamente el head y la etiqueta de dependencia[26].

LAS Labeled Attachment Score.

n-grama Un modelo de n-gramas es un modelo probabilístico en el cual se intenta predecir cual es la siguiente palabra o etiqueta en una secuencia. Esto se consigue tomando un corpus y contando las ocurrencias de palabras o etiquetas y sus posiciones, y con estos datos, cuando se quiere

predecir una palabra o etiqueta, se miran los n items anteriores y se calcula la probabilidad de que una cierta palabra o etiqueta ocurra dada la subsecuencia de tamaño n que ocurre inmediatamente antes. Cuando la secuencia de items que se miran "hacia atrás" es de tamaño 2 se le dice bigrama, cuando es de tamaño 3 trigrama, y así[8].

NER Named Entity Recognition.

NLTK Natural Language Toolkit.

part-of-speech Categoría léxica a la cual pertenece una palabra, por ejemplo: sustantivo, verbo, adverbio, etc.

PLN Procesamiento del Lenguaje Natural.

POS Part-Of-Speech.

pull request En el contexto de algunos sistemas de control de versionado de código (version control system) distribuidos, un pull request es una solicitud al dueño del sistema para que acepte una modificación al código realizado por un tercero.

Python Lenguaje de alto nivel de propósito general. Interpretado y dinámico.

Recall Medida para sistemas de extracción de información que se define de la siguiente forma: # de respuestas correctas dadas por el sistema / total # de posibles respuestas correctas en el texto.

shallow-parsing También conocido como chunking o parseo parcial, el shallow-parsing es un análisis sintáctico en el cual no se extrae todas las características sintácticas de un texto, sino que se obtienen unas pocas. Por ejemplo, para realizar NER puede que baste con obtener los grupos nominales, dado que las entidades con nombre tienden a ser nombres propios.

Single Classification Ripple Down Rules Enfoque incremental a la construcción de bases de conocimiento en el cual el conocimiento se almacena en una estructura de datos arborescente similar a un árbol de decisión, cada nodo posee una regla que es satisfecha por la información siendo considerada y exactamente dos ramas, según la regla fue satisfecha o no. Si un caso satisface una regla erróneamente, un experto agrega una regla nueva en la rama donde el proceso terminó[47].

tagset Conjunto de etiquetas utilizado para marcar tokens y asignarlos a una categoría gramatical. Se asignan durante la tarea de etiquetado gramatical, y a su vez son principalmente utilizados durante la tarea de análisis sintáctico.

UAS Unlabeled Attachment Score.

Unlabeled Attachment Score Medida para parsers de dependencias, se define como el porcentaje de tokens para los cuales el sistema predijo correctamente el head, sin importar la etiqueta de dependencia[26].

WordNet Base de datos léxica que agrupa palabras en conjuntos de sinónimos, adicionalmente provee definiciones y ejemplos de uso.

wrapper También conocido como adaptador. Objeto que implementa el patrón de diseño Adapter, el cual permite que una interfaz de una clase existente sea utilizada desde otra interfaz sin modificar su código fuente. Habilita a que dos componentes, normalmente incompatibles, interactuen.

árbol de constituyentes La idea de un constituyente es que una frase puede ser dividida en subfrases de acuerdo a una gramática libre de contexto. Ésta consiste en un conjunto de reglas de producción, cada cual indicando como se pueden agrupar los símbolos del lenguaje y un lexicón de palabras y símbolos[8]..

árbol de dependencias A diferencia del caso de los constituyentes, en los árboles de dependencias se utilizan gramáticas de dependencias, las cuales describen la estructura sintáctica de una oración en términos de palabras y relaciones semánticas o sintácticas entre estas palabras (llamadas dependencias léxicas)[8]..

Referencias

[1] Claude E Shannon y Warren Weaver. *The mathematical theory of communication*. University of Illinois press, 2015.

[2] Warren Weaver. "Translation". En: *Machine translation of languages* 14 (1955), págs. 15-23.

[3] P.J.Hancox. *A brief history of Natural Language Processing*. [Online; Último acceso 2-setiembre-2015]. 2015. URL: http://www.cs.bham.ac.uk/~pjh/sem1a5/pt1/pt1_history.html.

[4] Karen Sparck Jones. "Natural language processing: a historical review". En: *Current issues in computational linguistics: in honour of Don Walker*. Springer, 1994, págs. 3-16.

[5] Noam Chomsky. *Syntactic structures*. Walter de Gruyter, 2002.

[6] W John Hutchins. *Early years in machine translation: memoirs and biographies of pioneers*. Vol. 97. John Benjamins Publishing, 2000.

[7] John Hutchins. "ALPAC: the (in) famous report". En: *Readings in machine translation* 14 (2003), págs. 131-135.

[8] Daniel Jurafsky y James H. Martin. *Speech and Language Processing*. 2nd Edition. Prentice Hall, 1999.

[9] Erik F Tjong Kim Sang y Fien De Meulder. "Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition". En: *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*. Association for Computational Linguistics. 2003, págs. 142-147.

[10] Inderjeet Mani. "Summarization evaluation: An overview". En: (2001).

[11] *web CRATER*. URL: http://ucrel.lancs.ac.uk/corpora.html.

[12] *web CALLHOME*. URL: https://catalog.ldc.upenn.edu/LDC97S42.

[13] *Probabilistic Part-of-Speech Tagging Using Decision Trees*. URL: http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/data/tree-tagger1.pdf.

[14] *Improvements In Part-of-Speech Tagging With an Application to German*. URL: http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/data/tree-tagger2.pdf.

[15] S.B. Pham D.Q. Nguyen y col. "Ripple Down Rules for Part-of-Speech Tagging". En: *Proceedings of the 12th International Conference on Intelligent Text Processing and Computational Linguistics*. 2011, págs. 190-201.

[16] S.B. Pham D.Q. Nguyen y col. "RDRPOSTagger: A Ripple Down Rules-based Part-Of-Speech Tagger". En: *Proceedings of the Demonstrations at the 14th Conference of the European Chapter of the Association for Computational Linguistics*. 2014, págs. 17-20.

[17] Georgiana Dinu Grzegorz Chrupala y Josef van Genabith. *Learning Morphology with Morfette*. 2008.

[18] Grzegorz Chrupala. *Towards a Machine-Learning Architecture for Lexical Functional Grammar Parsing*. 2008.

[19] Jin-Dong Kim y col. "Introduction to the Bio-entity Recognition Task at JNLPBA". En: *COLING 2004 International Joint workshop on Natural Language Processing in Biomedicine and its Applications (NLPBA/-BioNLP) 2004*. Ed. por Nigel Collier, Patrick Ruch y Adeline Nazarenko. Geneva, Switzerland: COLING, ago. de 2004, págs. 73-78.

[20] A. Bies S. Kulick y col. *Integrated Annotation for Biomedical Information Extraction*. 2004.

[21] Yuka Tateishi Yoshimasa Tsuruoka y col. *Developing a Robust Part-of-Speech Tagger for Biomedical Text*. 2005.

[22] Yoshimasa Tsuruoka y Jun'ichi Tsujii. *Bidirectional Inference with the Easiest-First Strategy for Tagging Sequence Data*. 2005.

[23] Kristina Toutanova y Christopher D. Manning. "Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger". En: *Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000)*. 2000, págs. 63-70.

[24] Jesús Giménez y Lluís Màrquez. "SVMTool: A general POS tagger generator based on Support Vector Machines". En: *Proceedings of the 4th LREC*. Lisbon, Portugal, 2004.

[25] JOAKIM NIVRE y col. "MaltParser: A language-independent system for data-driven dependency parsing". En: *Natural Language Engineering* 13 (02 jun. de 2007), págs. 95-135. ISSN: 1469-8110. DOI: 10.1017/S1351324906004505. URL: http://journals.cambridge.org/article_S1351324906004505.

[26] Sabine Buchholz y Erwin Marsi. "CoNLL-X shared task on multilingual dependency parsing". En: *Proceedings of the Tenth Conference on Computational Natural Language Learning*. Association for Computational Linguistics. 2006, págs. 149-164.

[27] *Evaluación de MaltParser para el español*. URL: http://www.iula.upf.edu/recurs01_mpars_uk.htm.

[28] Giuseppe Attardi. "Experiments with a multilanguage non-projective dependency parser". En: *Proceedings of the Tenth Conference on Computational Natural Language Learning*. Association for Computational Linguistics. 2006, págs. 166-170.

[29] Dan Klein y Christopher D. Manning. "Accurate Unlexicalized Parsing". En: *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*. ACL '03. Sapporo, Japan: Association for Computational Linguistics, 2003, págs. 423-430. DOI: 10.3115/1075096.1075150. URL: http://dx.doi.org/10.3115/1075096.1075150.

[30] Richard Socher y col. “Parsing With Compositional Vector Grammars”. En: *In Proceedings of the ACL conference*. 2013.

[31] Dan Klein y Christopher D Manning. “Fast exact inference with a factored model for natural language parsing”. En: *Advances in neural information processing systems*. 2002, págs. 3-10.

[32] *Stanford Shift-Reduce Parser*. URL: http://nlp.stanford.edu/software/srparser.shtml.

[33] Carlos Subirats. “Spanish FrameNet: A frame-semantic analysis of the Spanish lexicon”. En: *Berlin/New York: Mouton de Gruyter* (2009), págs. 135-162.

[34] Thomas S Morton. “Coreference for NLP applications”. En: *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics. 2000, págs. 173-180.

[35] Adwait Ratnaparkhi. “Learning to parse natural language with maximum entropy models”. En: *Machine learning* 34.1-3 (1999), págs. 151-175.

[36] Adwait Ratnaparkhi. “Maximum entropy models for natural language ambiguity resolution”. Tesis doct. University of Pennsylvania, 1998.

[37] Jeffrey C Reynar y Adwait Ratnaparkhi. “A maximum entropy approach to identifying sentence boundaries”. En: *Proceedings of the fifth conference on Applied natural language processing*. Association for Computational Linguistics. 1997, págs. 16-19.

[38] Jeffrey C Reynar. “Topic segmentation: Algorithms and applications”. En: *IRCS Technical Reports Series* (1998), pág. 66.

[39] Adwait Ratnaparkhi. “A simple introduction to maximum entropy models for natural language processing”. En: *IRCS Technical Reports Series* (1997), pág. 81.

[40] Michael Collins. “Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms”. En: *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*. Association for Computational Linguistics. 2002, págs. 1-8.

[41] Steven Bird. “NLTK: The Natural Language Toolkit”. En: *Proceedings of the COLING/ACL on Interactive Presentation Sessions*. COLING-ACL ’06. Sydney, Australia: Association for Computational Linguistics, 2006, págs. 69-72. DOI: 10.3115/1225403.1225421. URL: http://dx.doi.org/10.3115/1225403.1225421.

[42] Edward Loper y Steven Bird. “NLTK: The Natural Language Toolkit”. En: *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*. ETMTNLP ’02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, págs. 63-70. DOI: 10.3115/1118108.1118117. URL: http://dx.doi.org/10.3115/1118108.1118117.

[43] Xavier Carreras y col. “FreeLing: An Open-Source Suite of Language Analyzers.” En: *LREC*. 2004.

[44] Hamish Cunningham. “GATE, a general architecture for text engineering”. En: *Computers and the Humanities* 36.2 (2002), págs. 223-254.

[45] Christopher D. Manning y col. “The Stanford CoreNLP Natural Language Processing Toolkit”. En: *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. 2014, págs. 55-60. URL: http://www.aclweb.org/anthology/P/P14/P14-5010.

[46] *Definición Lenguaje DOT*. URL: http://www.graphviz.org/doc/info/lang.html.

[47] Paul Compton y col. “Ripple down rules: Turning knowledge acquisition into knowledge maintenance”. En: *Artificial Intelligence in Medicine* 4.6 (1992), págs. 463-475.