



TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

Diseño e implementación de un analizador de dependencias para procesamiento de lenguaje natural en Español

Mediante Máquinas de Soporte Vectoriales

Autor

Alejandro Alcalde Barros

Directores

Salvador García López



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

—
8 de diciembre de 2016

Alejandro Alcalde Barros: *Diseño e implementación de un analizador de dependencias para procesamiento de lenguaje natural en Español*, Mediante Máquinas de Soporte Vectoriales, Grado en Ingeniería Informática, © 8 de diciembre de 2016

DIRECTOR:
Salvador García López

LOCALIZACIÓN:
Granada

ÚLTIMA MODIFICACIÓN:
8 de diciembre de 2016

Ohana means family.
Family means nobody gets left behind, or forgotten.
— Lilo & Stitch

Dedicated to the loving memory of Rudolf Miede.
1939–2005

RESUMEN

En este trabajo se implementa un método para analizar dependencias palabra a palabra con una estrategia de abajo a arriba (*Bottom-Up*) usando *Support Vector Machine* (SVM). En concreto, este trabajo se ha centrado en analizar las dependencias entre palabras en Castellano. El algoritmo no usa información sobre la estructura de las frases, realiza un análisis determinístico de las dependencias del idioma. El funcionamiento consiste en ir generando un árbol de dependencias para la frase mediante la ejecución de tres acciones — DESPLAZAR, IZQUIERDA, DERECHA — Inicialmente se comienza con tantos nodos como palabras tiene la frase, a medida que se ejecuta el algoritmo, se aplican las acciones mencionadas para ir construyendo el árbol. Por ejemplo, al aplicar la acción DERECHA a dos nodos contiguos, el nodo posicionado a la derecha pasa a ser padre del nodo izquierdo. En cuanto a los resultados obtenidos, han sido coherentes a la descripción proporcionada por los autores, obteniéndose resultados muy similares, incluso ligeramente mejores. La implementación original – Yamada y Matsumoto [35] – no se basa en conjuntos de datos en castellano, de modo que las comparaciones se han realizado frente al trabajo de Jain [12].

ETIQUETAS: PNL, SVM, Parseo de dependencias.

ABSTRACT

In this project, a method for analyzing word to word dependencies is implemented using a bottom-up strategy with the help of Support Vector Machine (SVM). In particular, this work has focused in analyzing dependencies between words for spanish language. The algorithm does not use any information about sentence's structure, it performs a deterministic analysis of the language's dependencies. Dependencies are build by generating a dependency tree for the sentence being analized. This tree is build with the help of three parsing actions — SHIFT, LEFT, RIGHT — At the beginning the algorithm has as many nodes as words has the sentence, in each iteration an action is applied to the nodes in order to build the dependency tree. For instance, when RIGHT action is applied to a pair of nodes, the node on the right becomes the parent of the left node. The accuracy obtained by this project its coherent with the one obtained for other authors. Our accuracy its very similar, even a little bit better. The original implementation by Yamada and Matsumoto [35] did not use any

Spanish datasets for its experiments, so comparison has been made against the work of Jain [12].

TAGS: NLP, SVM, Dependency parsing.

*An approximate answer to the right problem
is worth a good deal more than an exact answer
to an approximate problem.*

— John Tukey

AGRADECIMIENTOS

Muchas gracias a mi familia por apoyarme durante todos estos años de carrera, especialmente en los últimos meses de desarrollo de este proyecto. A mi tutor, por apoyar la idea.

Y por último, a Cristina, por estar siempre ahí.

ÍNDICE GENERAL


I	PUESTA EN ESCENA	1
1	MOTIVACIÓN E INTRODUCCIÓN	3
1.1	¿Qué es el Procesamiento del Lenguaje Natural?	3
1.2	Historia del Procesamiento del Lenguaje Natural	6
1.3	Limitaciones	8
1.4	El pipeline genérico	9
1.4.1	Pasos previos	10
1.4.2	Proceso principal del análisis de sentimientos	13
1.5	El pipeline de CORENLP	15
1.6	Estado del arte	17
1.6.1	A nivel del documento	17
1.6.2	A nivel de sentencia	18
1.6.3	A nivel de entidad	19
II	OBJETIVOS	21
2	PARSEO DE DEPENDENCIAS EN ESPAÑOL	23
III	RESOLUCIÓN DEL TRABAJO	25
3	UNA INTRODUCCIÓN A SCALA	27
3.1	¿Por qué SCALA?	27
3.2	Patrones de diseño como miembros de primera clase	27
3.3	Ventajas del uso de TRAITS	28
3.4	Reglas de Visibilidad	29
3.5	Big Data	30
4	ALGORITMO SELECCIONADO: STATISTICAL DEPENDENCY ANALYSIS WITH SUPPORT VECTOR MACHINES	33
4.1	Una introducción a las SVMs	34
4.2	Análisis de dependencias determinístico	35
4.2.1	Acciones para el parseo	35
4.2.2	Descripción del algoritmo	37
4.2.3	Extracción de características	38
4.2.4	Agrupando SVMs para reducir costes	38
4.3	Ejemplo práctico	39
5	IMPLEMENTACIÓN	41
5.1	Planificación	41
5.2	Análisis y Diseño	42
5.3	Implementación	43
5.3.1	DEPENDENCYPARSER	44
5.3.2	NODE	46
6	CASOS DE PRUEBA ORIENTADOS A APRENDIZAJE AUTOMÁTICO	51

6.1	Test-Driven Development	51
6.1.1	El ciclo de Test-Driven Development (TDD)	52
6.2	Desarrollo orientado a comportamiento	53
6.3	TDD aplicado al Aprendizaje Automático	53
6.4	Casos de prueba realizados	53
6.4.1	DATA_PARSER_SPEC	53
6.4.2	DEPENDENCY_PARSER_CHECK_BASE_LINE_SPEC	54
IV	CONCLUSIONES Y VÍAS FUTURAS	59
7	EVALUACIÓN, COMPARACIÓN Y DISCUSIÓN DE RESULTADOS	61
7.1	Conjuntos de Datos	61
7.2	Medidas de Evaluación	61
7.3	Comparación de Resultados	62
8	VÍAS FUTURAS	65
8.1	Trabajo Futuro	65
V	APÉNDICE	67
	BIBLIOGRAFÍA	69

ÍNDICE DE FIGURAS

Figura 1	Ejemplo de parseo de dependencias	8
Figura 2	Ejemplo de parseo de dependencias 2	8
Figura 3	Visualización del <i>pipeline</i> de CORENLP	17
Figura 4	Diagrama de clases PERSONA y TRABAJADOR	29
Figura 5	Estructura en árbol de la frase “Rolls-Royce Motor Cars Inc. said it expects its U.S. sales to remain steady at about 1,200 cars.”	33
Figura 6	Árbol de parseo de dependencias	34
Figura 7	Ejemplo SVM	35
Figura 8	Ejemplo de la acción DESPLAZAR. (a) muestra el estado antes de aplicar la acción. (b) el resultado tras aplicarla	36
Figura 9	Ejemplo de la acción DERECHA. (a) Estado antes de la acción. (b) Estado tras aplicar la acción	36
Figura 10	Ejemplo de la acción LEFT. (a) Estado antes de la acción. (b) Estado tras aplicar la acción	36
Figura 11	Planificación del proyecto	41
Figura 12	Paquetes del proyecto	42
Figura 13	Diagrama de clases completo	49
Figura 14	Frase en formato CoNLL-U.	62
Figura 15	Frase en formato convertido.	62

TODO LIST

 Con las figuras de arriba quizá no haya que poner esta sección. 39

ÍNDICE DE TABLAS

Tabla 1	<i>Pipeline</i> de CORENLP y disponibilidad por lenguaje	9
Tabla 2	Reglas de visibilidad en SCALA	30
Tabla 3	Descripción del tipo de características y sus valores	39

Tabla 4	Comparación de resultados	63
---------	---------------------------	----

ÍNDICE DE CÓDIGO FUENTE

Código 1	Patrón Singleton en Scala.	28	
Código 2	Refactorizando Upper	28	
Código 3	Composición en lugar de herencia	29	
Código 4	WORDCOUNT en JAVA	31	
Código 5	WORDCOUNT en SCALA	31	
Código 6	Estructuras de datos más relevantes del paquete DATAS- STRUCTURES	42	
Código 7	Parámetros para la SVM	43	
Código 8	Codificación de las acciones DESPLAZAR, IZQUIERDA, DERECHA		43
Código 9	Test que comprueba si la entrada de datos es correcta o no	55	
Código 10	Código del test DependencyParserCheckBase- lineSpec	56	
Código 11	Código del test DependencyParserCheckNFea- turesSpec	57	

LISTA DE ALGORITMOS

Algoritmo 1	Algoritmo de parseo	37
-------------	---------------------	----

ACRÓNIMOS

NLP	Natural Language Processing
PNL	Procesamiento del Lenguaje Natural
IA	Inteligencia Artificial

ASR	Automatic Speech Recognition
RVA	Reconocimiento de Voz Automático
POS	Part-Of-Speech
AA	Aprendizaje Automático
NER	Named Entities
DP	Dependency Parsing
HRL	High-Resource Language
LRL	Low-Resource Language
SDS	Spoken Dialogue System
DM	Dialogue Management
TTS	Text-To-Speech
API	Application Programming Interface
HTML	Hyper Text Markup Language
CRF	Conditional Random fields
PTB	Penn Tree Bank
XML	Extensible Markup Language
SO	Semantic Orientation
PMI-IR	Pointwise Mutual Information and Information Retrieval
SVM	Support Vector Machine
SSyntSs	Surface-Syntactic structures
DSyntSs	Deep-Syntactic Structures
TDD	Test-driven Development
BDD	Behavior-Driven Development
FP	Functional Programming
TDD	Test-Driven Development
BDD	Behavior-Driven Development
ROC	ReceIver Operating Characteristic
AUC	Area Under Curve

Parte I

PUESTA EN ESCENA

La memoria se organiza en cuatro partes.

La primera, en el [Capítulo 1](#), es una introducción al Procesamiento del Lenguaje Natural, en ella se definen los tipos de aplicaciones que permite, así como una descripción del *pipeline* usual que suele seguirse en este tipo de sistemas y el estado del arte.

En la segunda parte, [Capítulo 2](#) se listan los objetivos del proyecto.

La tercera parte se compone de los Capítulos ([3](#) [4](#) [5](#) y [6](#)). En ellos se introduce al lenguaje SCALA, se explican los detalles técnicos del algoritmo de parseo de dependencias, se discute la implementación y se muestran los casos de prueba realizados, respectivamente.

Por último, la cuarta parte consta del [Capítulo 7](#) y [Capítulo 8](#), en donde se exponen los resultados obtenidos, se comparan los resultados con el algoritmo original y se proponen trabajos futuros.

MOTIVACIÓN E INTRODUCCIÓN

1.1 ¿QUÉ ES EL PROCESAMIENTO DEL LENGUAJE NATURAL?

El lenguaje natural se refiere a cualquier lenguaje hablado por un humano [24], (por ejemplo, Inglés, Castellano o Chino). El *Natural Language Processing* (NLP) es un campo de la ciencia de la computación e ingeniería desarrollado a partir del estudio del lenguaje y la computación lingüística dentro del campo de la Inteligencia Artificial (IA). Los objetivos del NLP son diseñar y construir aplicaciones que faciliten la interacción humana con la máquinas y otros dispositivos mediante el uso del lenguaje natural. Dentro del amplio campo del NLP podemos distinguir las siguientes áreas principales:

*Procesamiento del
Lenguaje
Natural (PNL)*

El lenguaje natural se refiere a cualquier lenguaje hablado por un humano, (por ejemplo, Inglés, Castellano o Chino). El NLP es un campo de la ciencia de la computación e ingeniería desarrollado a partir del estudio del lenguaje y la computación lingüística dentro del campo de la IA. Los objetivos del NLP son diseñar y construir aplicaciones que faciliten la interacción humana con la máquinas y otros dispositivos mediante el uso del lenguaje natural. Dentro del amplio campo del NLP podemos distinguir las siguientes áreas principales [10, 24]:

RESÚMENES este área incluye aplicaciones que puedan, basándose en una colección de documentos, dar como salida un resumen coherente del contenido de los mismos. Otra de las posibles aplicaciones sería generar presentaciones a partir de dichos documentos. En los últimos años, la información disponible en la red ha aumentado considerablemente. Un claro ejemplo es la literatura científica, o incluso repositorios de información más genérica como *Wikipedia*. Toda esta información escrita en lenguaje natural puede aprovecharse para entrenar modelos que sean capaces de generar hipótesis por sí mismos, generar resúmenes o extraer hechos. Un ejemplo claro puede ser la extracción de hechos básicos que relacionen dos entidades ("*Luís es padre de Cristina*").

TRADUCCIÓN AUTOMÁTICA: Esta fue la principal área de investigación en el campo del NLP. Como claro ejemplo tenemos el traductor de Google, mejorando día a día. Sin embargo, un traductor realmente útil sería aquel que consiga traducir en tiempo real una frase que le dictemos mientras decidimos qué línea de autobús debemos coger para llegar a tiempo a una conferencia en Zurich. La traducción entre lenguajes es quizá una de las formas más transcendentales en las

que las máquinas podrían ayudar en comunicaciones entre humanos. Además, la capacidad de las máquinas para traducir entre idiomas humanos se considera aún como un gran test a la [IA](#), ya que una traducción correcta no consiste en el mero hecho de generar frases en un idioma humano, también requiere del conocimiento humano y del contexto, pese a las ambigüedades de cada idioma. Por ejemplo, la traducción literal “*bordel*” en Francés significa Burdel; pero si alguien dice “*Mi cuarto es un bordel*”, el traductor debería tener el conocimiento suficiente para inferir que la persona se está refiriendo a que su habitación es un desorden.

Conocido como texto
paralelo

La traducción automática fue una de las primeras aplicaciones no numéricas de la computación y comenzó a estudiarse de forma intensiva en la década de los 50. Sin embargo, no fue hasta la década de los 90 cuando se produjo una transformación en este área. IBM se hizo con una gran cantidad de frases en Inglés y Francés que eran traducciones las unas de las otras, lo cual permitió recopilar estadísticas de traducciones de palabras y secuencias de palabras, concediendo así el desarrollo de modelos probabilísticos para la traducción automática. Hasta ese momento, todo el análisis gramático se hacía manualmente.

A la llegada del nuevo milenio, se produjo una explosión de texto disponible en la red, así como grandes cantidades de *texto paralelo*. Se dieron invención a nuevos sistemas para la traducción automática basados en modelos estadísticos basados en frases en lugar de palabras. En lugar de traducir palabra por palabra, ahora se tenían en cuenta pequeños grupos de palabras que a menudo poseen una traducción característica.

En los últimos años, y mediante el uso de *deep learning* se están desarrollando modelos de secuencias basados en este tipo de aprendizaje bastante prometedores. La idea principal del *deep learning* reside en entrenar un modelo con varios niveles de representación para optimizar el objetivo deseado, una traducción de calidad, en este caso. Mediante estos niveles el modelo puede aprender representaciones intermedias útiles para la tarea que le ocupa. Este método de aprendizaje se ha explotado sobre todo en redes neuronales. Un ejemplo claro de *deep learning* usando redes neuronales es el reconocimiento de dígitos, cada capa interna de la red neuronal intenta extraer características representativas de cada dígito a distintas escalas. Podemos ver una demostración de este comportamiento en Pound y Riley [23]

RECONOCIMIENTO DE VOZ: Una de las tareas más difíciles en [NLP](#). Aún así, se han conseguido grandes avances en la construcción de modelos que pueden usarse en el teléfono móvil o en el ordenador. Estos modelos son capaces de reconocer expresiones del lenguaje hablado como preguntas y comandos. Desafortunadamente, los sistemas *Automatic Speech Recognition* ([ASR](#)) funcionan bajo dominios muy acotados y no permiten al interlocutor desviarse de la entrada que espera el sis-

Reconocimiento de
Voz
Automático ([RVA](#))

tema, por ejemplo, “*Por favor, diga ahora la opción a elegir: 1 Para... , 2 para...*”

SDS: Los *Spoken Dialogue Systems* (**SDSs**). El diálogo ha sido un tema popular para el **NLP** desde los 80. En estos sistemas se pretende reemplazar a los usuales buscadores en los que introducimos un texto para obtener algún tipo de respuesta a una pregunta. Por ejemplo, si quisieramos saber a qué hora abre un centro comercial, bastaría con hablarle al sistema en lenguaje natural – nuestro lenguaje natural, ya sea Inglés, Alemán o Castellano y el sistema nos daría respuesta a nuestra pregunta. Aunque ya existen este tipo de sistemas (por ejemplo, *Siri de Apple*, *Cortana de Microsoft*, *Google Now...*) están aún en una situación muy precaria, ya que ninguno entiende por completo el lenguaje natural, solo un subconjunto de frases clave.

SDS: *Sistemas de Diálogo Hablados*

La creación de **SDSs**, ya sea entre humanos o entre humanos y agentes artificiales requiere de herramientas como:

- **ASR**, para identificar qué dice el humano.
- *Dialogue Management* (**DM**), para determinar qué quiere el humano.
- Acciones para obtener la información o realizar la actividad solicitada.
- Síntesis *Text-To-Speech* (**TTS**), para comunicar dicha información al humano de forma hablada.

DM: *Gestión del diálogo*

Leer un texto por una máquina

En la actualidad, Hinton y col. [9] desarrollaron un **SDS** haciendo uso de *deep learning* para asociar señales sonoras a secuencias de palabras y sonidos del idioma humano, logrando avances importantes en la precisión del reconocimiento del habla.

CLASIFICACIÓN DE DOCUMENTOS: Una de las áreas más exitosas del **NLP**, cuyo objetivo es identificar a qué categoría debería pertenecer un documento. Ha demostrado tener un amplio abanico de aplicaciones, por ejemplo, filtrado de *spam*, clasificación de artículos de noticias, valoraciones de películas... Parte de su éxito e impacto se debe a la facilidad relativa que conlleva entrenar los modelos de aprendizaje para hacer dichas clasificaciones.

ANÁLISIS DE SENTIMIENTOS: Gran parte del trabajo en **NLP** se ha centrado en el análisis de sentimientos (identificación de orientaciones positivas o negativas en textos) e identificación de creencias positivas, negativas o neutrales en frases basándose en información léxica y sintáctica. Tanto las creencias como los sentimientos constituyen actitudes hacia eventos y proposiciones, aunque en concreto, los sentimientos pueden también referirse a actitudes hacia objetos tales como personas, organizaciones y conceptos abstractos. La detección

de sentimientos y emociones en texto requiere de información léxica y a nivel de la propia sentencia. Por lo general, el sentimiento puede detectarse a través del uso de palabras expresando orientaciones positivas o negativas, por ejemplo, *triste*, *preocupado*, *difícil* son todas palabras con una connotación negativa, mientras que *cómodo*, *importante*, *interesante* denotan un sentimiento positivo. Las aproximaciones más sofisticadas para el análisis de sentimientos intentan buscar tanto la fuente como el objeto del sentimiento, por ejemplo, quién está expresando un sentimiento positivo sobre alguna persona, objeto, actividad o concepto.

La comunidad del reconocimiento de voz está igualmente implicada en el estudio de actitudes positivas y negativas, centrándose en la identificación de emociones haciendo uso de información acústica y prosódica, es decir un relieve en la pronunciación. Otras investigaciones se han centrado en identificar emociones particulares, específicamente las seis emociones básicas según Ekman – ira, aversión, temor, dicha, tristeza y asombro – las cuales pueden ser reacciones a eventos, proposiciones u objetos. Por contra, la generación de emociones ha demostrado ser un reto mucho mayor para la síntesis TTS.

La clasificación de sentimientos es algo ampliamente usado para identificar opiniones – puntos de vista positivos o negativos hacia personas, instituciones o ideas – en muchos idiomas y géneros. Una de las aplicaciones más prácticas, y de las que más abundan consiste en identificar críticas sobre películas o productos [21, 33].

La minería de datos en redes sociales con el fin de realizar análisis de sentimientos se ha convertido en un tema popular con el objetivo de evaluar el *estado de ánimo* del público – de twitter, por ejemplo. –

El NLP emplea técnicas computacionales con el propósito de aprender, entender y producir lenguaje humano. Las aproximaciones de hace unos años en el campo de la investigación del lenguaje se centraban en automatizar el análisis de las estructuras lingüísticas y desarrollar tecnologías como las mencionadas anteriormente. Los investigadores de hoy en día se centran en usar dichas herramientas en aplicaciones para el mundo real, creando sistemas de diálogo hablados y motores de traducción *Speech-to-Speech*, es decir, dados dos interlocutores, interpretar y traducir sus frases. Otro de los focos en los que se centran las investigaciones actuales son la minería en redes sociales en busca de información sobre salud, finanzas e identificar los sentimientos y emociones sobre determinados productos.

1.2 HISTORIA DEL PROCESAMIENTO DEL LENGUAJE NATURAL

A continuación, citamos algunos de los avances en este campo durante los últimos años según Hirschberg y Manning [10].

Durante las primeras épocas de esta ciencia, se intentaron escribir vocabularios y reglas del lenguaje humano para que el ordenador las

entendiera. Sin embargo, debido a la naturaleza ambigua, variable e interpretación dependiente del contexto de nuestro lenguaje resultó una ardua tarea. Por ejemplo, una estrella puede ser un ente astronómico o una persona, y puede ser un nombre o un verbo.

En la década de los 90, los investigadores transformaron el mundo del NLP desarrollando modelos sobre grandes cantidades de datos sobre lenguajes. Estas bases de datos se conocen como *corpus*. El uso de estos conjuntos de datos fueron uno de los primeros éxitos notables del uso del *big data*, mucho antes de que el Aprendizaje Automático (AA) acuñara este término.

Esta aproximación estadística al NLP descubrió que el uso de métodos simples usando palabras, secuencias del *Part-Of-Speech* (POS) (si una palabra es un nombre, verbo o preposición), o plantillas simples pueden obtener buenos resultados cuando son entrenados sobre un gran conjunto de datos. A día de hoy, muchos sistemas de clasificación de texto y sentimientos se basan únicamente en los distintos conjuntos de palabras o “*bag of words*” que contienen los documentos, sin prestar atención a su estructura o significado. El estado del arte de hoy día usa aproximaciones con AA y un rico conocimiento de la estructura lingüística. Un ejemplo de estos sistemas es *Stanford CORENLP* [18]. CORENLP proporciona un *pipeline* estándar para el procesamiento del NLP incluyendo:

POS: Categorías morfosintácticas en castellano

POS TAGGING: Etiquetado morfosintáctico. Módulo encargado de leer texto en algún lenguaje y asignar la categoría morfosintáctica a cada palabra, por ejemplo, nombre, verbo, adjetivo... aunque por lo general se suelen usar etiquetas más detalladas como “*nombre-plural*”.

NER: *Named Entities* (NER), etiqueta palabras en un texto correspondientes a *nombres de cosas*, como personas, nombres de compañías, nombres de proteínas o genes etc. En concreto, CORENLP distingue de forma muy precisa tres tipos de clases, personas, organizaciones y localizaciones.

PARSEO GRAMATICAL: Resuelve la estructura gramatical de frases, por ejemplo, qué grupos de palabras van juntos formando frases y qué palabras son sujeto u objeto de un verbo. Como se ha comentado, en aproximaciones anteriores se usaban parseadores probabilísticos usando conocimiento del lenguaje a partir de sentencias analizadas sintácticamente a mano. Para así producir el análisis más probable de sentencias nuevas. Actualmente se usan parseadores estadísticos, los cuales aún cometen fallos, pero funcionan bien a rasgos generales.

DP: *Dependency Parsing* (DP) o parseo de dependencias. Analiza la estructura gramatical de una frase, estableciendo relaciones entre pa-

labras principales y palabras que modifican dichas palabras principales. La [Figura 1](#) muestra un ejemplo. La flecha dirigida de la palabra *moving* a la palabra *faster* indica que *faster* modifica a *moving*. La flecha está etiquetada con una palabra, en este caso *advmod*, indicando la naturaleza de esta dependencia. La [Figura 2](#) muestra ejemplos de los distintos módulos del *pipeline* de CORENLP

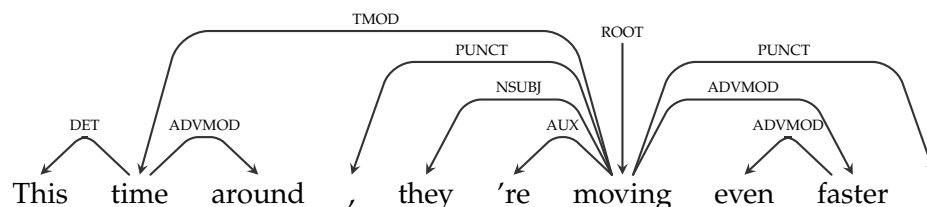


Figura 1: Ejemplo de parseo de dependencias

Part of speech:

NNP NNP RB VBD IN NNP NNP , CC PRP VBZ RB VBG PRP IN PRP .
 Mrs. Clinton previously worked for Mr. Obama, but she is now distancing herself from him.

Named entity recognition:

Person Date Person Date
 Mrs. Clinton previously worked for Mr. Obama, but she is now distancing herself from him.

Co-reference:

Coref Coref Coref Coref
Mention Ment M Mention M
 Mrs. Clinton previously worked for Mr. Obama, but she is now distancing herself from him.

Basic dependencies:

compound nsubj advmod cc conj nmod case compound nsubj aux advmod nmod case
NNP NNP RB VBD IN NNP NNP , CC PRP VBZ RB VBG PRP IN PRP .
 Mrs. Clinton previously worked for Mr. Obama, but she is now distancing herself from him.

Figura 2: Ejemplo de la salida del programa CORENLP. De arriba a abajo, se muestra la categoría morfosintáctica de cada palabra, el nombre de algunas entidades, determina qué entidad hace co-referencia a la misma persona u organización y por último la estructura sintáctica de cada frase, usando un análisis de dependencias gramaticales.

1.3 LIMITACIONES

Aunque se han producido avances, una de las principales limitaciones del NLP hoy día es el hecho de que la mayoría de recursos y sistemas solo están disponibles para los denominados *High-Resource Languages (HRLs)*, estos lenguajes son el Inglés, Francés, Español, Ale-

HRL: Idiomas de altos recursos

Tabla 1: *Pipeline* de CORENLP y disponibilidad por lenguaje

ANNOTATOR	AR	ZH	EN	FR	DE	ES
Tokenize / Segment	✓	✓	✓	✓		✓
Sentence Split	✓	✓	✓	✓	✓	✓
Part of Speech	✓	✓	✓	✓	✓	✓
Lemma			✓			
Named Entities		✓	✓		✓	✓
Constituency Parsing	✓	✓	✓	✓	✓	✓
Dependency Parsing		✓	✓	✓	✓	
Sentiment Analysis			✓			
Mention Detection		✓	✓			
Coreference		✓	✓			
Open IE			✓			

mán y Chino. Por contra, hay una gran cantidad de *Low-Resource Languages* (LRLs) – como Bengali, Indonesio, Punjabi, Cebuano y Swahili – hablados y escritos por millones de personas que no disponen de este tipo de sistemas. Uno de los mayores retos para la comunidad del lenguaje es desarrollar recursos y herramientas para cientos o miles de lenguajes, no solo para unos pocos.

Aún existiendo bastante *software* trabajando con NLP, y para idiomas HRL, suelen obtenerse mejores resultados para un idioma en concreto, el Inglés. Es por ello que este trabajo se ha centrado en desarrollar una fase del *pipeline* que se encuentra en todos los sistemas que realizan análisis de sentimientos, y en general NLP para el idioma Español. Como ejemplo podemos citar el famoso CORENLP [18].

En la Tabla 1 se lista todo el *pipeline* de CORENLP junto con el soporte para cada lenguaje. Como se aprecia, el *pipeline* esta completo únicamente para el Inglés. El objetivo de este trabajo ha consistido en implementar un parseo de dependencias para el Español.

Con la introducción del *pipeline* de CORENLP, se pasa ahora a describir el proceso que todo sistema para NLP debe seguir. Comenzaremos mencionando un proceso genérico, para después profundizar en el *pipeline* de un *software* específico, CORENLP en nuestro caso.

1.4 EL PIPELINE GENÉRICO

En esta sección se comentará el proceso habitual que suele seguirse como *pipeline* en los problemas de NLP. Para ello se describirán los distintos niveles de análisis en los que opera dicho *pipeline*, así como las diferentes aproximaciones que se usan y los problemas más comunes a los que se enfrenta todo sistema que realice NLP.

LRL: Idiomas de
bajos recursos

Liu [16] define una opinión como una quintupla conteniendo el objetivo de la opinión (o *entidad*), el atributo del objetivo al que se dirige la opinión, el sentimiento (o polaridad) de la opinión, pudiendo ser este positivo, negativo o neutral, el poseedor de dicha opinión y la fecha en la que se produjo. Formalmente se podría definir como la tupla:

$$(e_i, a_{ij}, s_{ijkl}, h_k, t_l)$$

donde e_i corresponde con el objetivo de la opinión i -ésima, a_{ij} es el j -ésimo atributo de e_i , h_k el k -ésimo poseedor de la opinión, t_l codifica el tiempo en el que se emitió la opinión y por último, s_{ijkl} es la polaridad de la opinión hacia el atributo a_{ij} para la entidad e_i por el poseedor de la opinión h_k en el momento t_l .

El principal objetivo del análisis de sentimientos consiste en encontrar todas las tuplas $(e_i, a_{ij}, s_{ijkl}, h_k, t_l)$ en un documento o colección de documentos.

1.4.1 Pasos previos

El procesamiento más usual para realizar tareas de análisis de sentimientos se puede dividir en una serie de pasos definidos. Dichos pasos corresponden a la adquisición del *corpus* o datos, preprocesamiento del texto, el proceso principal del análisis de sentimientos, agregación y resumen de los resultados y por último, visualización. En los próximos apartados se mencionarán los tres primeros.

1.4.1.1 Adquisición de Datos

En este paso se debe obtener el *corpus* para el cual se desea realizar el análisis de sentimientos. Actualmente existen dos aproximaciones para realizar esta tarea. Una de ellas consiste en hacer uso de la *Application Programming Interface* (API) de alguna página web de la que se desee extraer el *corpus*. Una de las APIs más populares para este propósito es la de Twitter. La segunda aproximación hace uso de *Web Crawlers* para extraer datos de las webs deseadas.

Ambas aproximaciones presentan sus ventajas y desventajas, y por tanto se debe encontrar un equilibrio en función de cual se decida usar. Veamos algunos de ellos.

Mediante el uso de una API la implementación es sencilla, los datos obtenidos están ordenados y poseen una estructura poco sujeta al cambio, sin embargo, en función del proveedor de la API se presentan ciertas limitaciones. Siguiendo con Twitter, su API limita a 180 consultas cada 15 minutos el número de peticiones que se pueden realizar. Además, su API para *streaming* presenta otras limitaciones. En lugar de imponer límites a la cantidad de peticiones, restringe el número de clientes que se pueden conectar desde la misma dirección IP al

mismo tiempo, así como la velocidad a la que cada uno puede leer los datos. Pese a las limitaciones anteriores, la más importante quizás sea que esta aproximación depende de la existencia de una [API](#) por parte del sitio web.

Por otro lado, la aproximación basada en rastreadores webs son bastante más complejas de implementar, la razón principal se debe a que los datos obtenidos, por norma general tendrán ruido y no estarán estructurados. Como beneficio, esta aproximación tiene la capacidad no imponernos prácticamente ninguna restricción. Si bien es cierto que se deben respetar ciertas normas y protocolos, como las indicaciones del fichero `ROBOTS.TXT`¹ de cada sitio web, no realizar múltiples peticiones al mismo servidor y espaciar las mismas para no someter al servidor a demasiada carga.

1.4.1.2 Preprocesamiento del texto

El segundo paso en el *pipeline* del análisis de sentimientos es el preprocesamiento del texto adquirido. En este paso se realizan varias tareas habituales para el [NLP](#) correspondientes al análisis léxico. Algunas de estas tareas son:

TOKENIZACIÓN: Es una técnica fundamental para la mayoría de tareas en [NLP](#). Encargada de separar las cadenas de texto del documento completo en una lista de palabras. Es muy sencilla de realizar para idiomas delimitados por espacios como el Inglés, Español o Francés, pero se torna considerablemente más compleja para idiomas donde las palabras no son delimitadas por espacios, como el Japonés, Chino y Thai.

Los idiomas anteriores requieren de un proceso llamado segmentación de palabras, el cual es un problema de etiquetado secuencial. Para resolverlo se usan *Conditional Random fields* ([CRF](#)), método que ha demostrado ser superior a los modelos de Markov ocultos y modelos de Markov de máxima entropía [[14](#), [22](#), [28](#)]. Debido a que es una técnica fundamental, existen multitud de herramientas disponibles, para idiomas delimitados por espacios, el tokenizador de Stanford² o el de [OPENNLP](#)³. Para la segmentación de palabras en Chino, existen herramientas como [ICTCLAS](#)⁴, [THULAC](#)⁵ y el segmentador de Stanford⁶.

STEMMING: Proceso heurístico encargado de eliminar los afijos de la palabra para dejarlos en su forma canónica (invariante, o raíz). Por

¹ <http://www.robotstxt.org/robotstxt.html>

² <http://nlp.stanford.edu/software/tokenizer.shtml>

³ <https://opennlp.apache.org/documentation/manual/opennlp.html#tools.tokenizer>

⁴ <http://ictclas.nlpir.org>

⁵ <http://thulac.thunlp.org>

⁶ <http://nlp.stanford.edu/software/segmenter.shtml>

ejemplo, *persona*, *personificar* y *personificación* pasan a ser *persona* una vez acabado este proceso.

LEMATIZACIÓN: Proceso algorítmico para convertir una palabra a su forma de diccionario no-inflexible – *non-inflected* en inglés –. Esta fase es análoga a la anterior (*stemming*) pero se realiza a través de una serie de pasos más rigurosos que incorporan un análisis morfológico de cada palabra.

ELIMINACIÓN DE STOPWORDS: Actividad encargada de borrar las palabras usadas para estructurar el lenguaje pero que no contribuyen de modo alguno a su contenido. Algunos ejemplos de estas palabras pueden ser *de*, *la*, *que*, *el*, *en*, *y*, *a*, *los*, *del*, *se*, *las*, *por*, *un*, *par*, *con*.⁷

SEGMENTACIÓN DE FRASES: Procedimiento que separa párrafos en sentencias. Presenta sus propios retos, ya que los signos de puntuación, como el punto (.) se usan con frecuencia para marcar tanto el fin de una frase como para denotar abreviaciones y números decimales.

POS TAGGING Y PARSEO o etiquetado morfosintáctico. Paso que etiqueta cada palabra de una sentencia con su categoría morfosintáctica, como *adjetivo*, *nombre*, *verbo*, *adverbio* y *preposición*. Estas etiquetas pueden usarse como entrada para procesamientos futuros, como el parseo de dependencias (Objetivo de este trabajo) o como característica para el proceso de AA. De igual manera que la segmentación, es un problema de etiquetado secuencial. El etiquetado morfosintáctico proporciona información léxica, el parseo obtiene información sintáctica. El parseo genera un árbol que representa la estructura gramatical de una sentencia dada con la correspondiente relación entre los distintos constituyentes. Este trabajo se ha centrado en construir un parseo de dependencias para el Español.

Cabe destacar que no es obligatorio aplicar todos y cada uno de los pasos anteriores. En función del tipo de aplicación se ejecutarán unos pasos u otros. Por ejemplo, un sistema basado en AA probablemente aplicará cada uno de estos pasos con el fin de reducir la dimensionalidad y ruido del problema. Por contra, una aproximación no-supervisada quizá necesite la categoría sintáctica de algunas de las *stopwords* para construir reglas de dependencia con el fin de usarlas posteriormente en el proceso principal del análisis. Es claro pues, que la aproximación no-supervisada en este caso deberá omitir la fase eliminación de *stopwords*. En [Otras aproximaciones](#) se describe con

⁷ Para ver una lista completa de palabras visitar: <http://snowball.tartarus.org/algorithms/spanish/stop.txt>

más detalle las diferencias entre aproximaciones supervisadas frente a no supervisadas.

Por otra parte, existen otro tipo de pasos dependientes en su totalidad del origen de los datos y el método de adquisición. En particular, los datos que se obtengan a través de un rastreador web deberán ser procesados con fin de eliminar las etiquetas *Hyper Text Markup Language* (HTML) e información no textual – como imágenes y anuncios – El texto extraído de Twitter necesitará de especial atención en cuanto a *hashtags*, menciones, *retweets*, texto póbrememente escrito, emoticonos, carcajadas escritas y palabras con caracteres repetidos — *siiiiiiiiiii* —

1.4.2 Proceso principal del análisis de sentimientos

La tercera fase en el *pipeline* es el proceso principal del análisis. A continuación se mencionarán los distintos niveles de granularidad en los que actúan las aproximaciones más comunes.

1.4.2.1 Niveles de análisis

Desde que el análisis de sentimientos comenzó a ganar popularidad, se han ido proponiendo distintos niveles para el análisis en las distintas etapas. La primera se realizaba a nivel del documento, donde el objetivo residía en identificar la polaridad general del mismo. Más tarde, el interés se desplazó hacia un nivel más específico, las setencias. Por último, se bajó un paso más en cuanto a la granularidad para interesarse a nivel de entidad. Cabe destacar que los niveles más granulares pueden agruparse o conglomerarse para formar niveles más altos – menos granulares, a mayor escala – Por ejemplo, una análisis de sentimientos podría calcular la media de polaridades en una frase y producir un resultado a nivel de sentencias. Veamos a continuación los distintos niveles.

NIVEL DE DOCUMENTO: A este nivel, el análisis trata de clasificar el documento al completo con una polaridad positiva o negativa. La utilidad de este nivel a menudo está limitada y por normal general se usa en el contexto del análisis de reseñas [17]. Formalmente, el objetivo para este tipo de tareas puede definirse como una versión modificada de la representación introducida en la Sección 1.4 y corresponde a la búsqueda de tuplas

$$(-, \text{GENERAL}, S_{\text{GENERAL}}, -, -)$$

donde la entidad e , el poseedor de la opinión h , y el tiempo t en el que se manifestó la opinión se asumen conocidos o se ignoran. El atributo a_j de la entidad e se corresponde con GENERAL. Todo esto implica que el análisis devolverá sólo la polaridad general del documento.

NIVEL DE SENTENCIA: Análogo al anterior, ya que se podría considerar una sentencia como un documento corto. Sin embargo, este nivel presenta algunos pasos de preprocesamiento consistentes en separar el documento en oraciones, paso que a su vez posee retos similares a la tokenización de idiomas no delimitados por periodos — visto en [Preprocesamiento del texto](#) —

NIVEL DE ENTIDAD Y ASPECTO: El más granular de todos a los que el análisis de sentimientos puede trabajar. A este nivel la tarea no consiste únicamente en encontrar la polaridad de una opinión, también su objetivo — a quién va dirigida — Por esto mismo, la definición de la quintupla en la [Sección 1.4](#) aplica al completo. El análisis a nivel de documento y sentencias funcionan bien cuando el texto analizado contiene una sola entidad y aspecto, pero empeoran para varios [7]. Para resolver este tipo de problemas, algunos sistemas de análisis de sentimientos basados en aspectos intentan detectar cada aspecto mencionado en el texto para asociarlo con una opinión.

El primer trabajo que se ocupó de resolver este problema fue obra de Hu y Liu [11]. Hu y Liu detectaban características de productos – aspectos – comentados con frecuencia por clientes, luego identificaban dichas sentencias con opiniones, las evaluaban en base a su polaridad y finalmente resumían los resultados.

1.4.2.2 Otras aproximaciones

Existen dos aproximaciones para llevar a cabo el proceso de análisis de sentimientos. Una basada en léxico, no supervisada. Esta aproximación depende de reglas y heurísticas obtenidas del conocimiento lingüístico [31]. La otra aproximación es supervisada, usa [AA](#), aquí se utilizan algoritmos que aprenden la información subyacente de datos previamente anotados, permitiéndoles así clasificar instancias nuevas — sin etiquetar [21] —. Aunque estas dos aproximaciones son las más usadas, existen estudios que han demostrado buenos resultados al combinar ambas. Pasamos ahora a describirlas en detalle.

APROXIMACIÓN NO SUPERVISADA BASADA EN EL LÉXICO: También llamada basada en la semántica. Intenta determinar la polaridad del texto usando un conjunto de reglas y heurísticas obtenidas del conocimiento del idioma. Los pasos habituales consisten en marcar primero cada palabra y frase con su correspondiente polaridad con ayuda de un diccionario. El siguiente paso incorpora el análisis de modificadores de sentimientos – como la negación – y su ámbito – intensificadores y negación –. Por último se tratan las conjunciones adversativas – *pero, aunque, mas* – comprendiendo cómo afectan a la polaridad y reflejándolo en la puntuación final del sentimiento [17].

APROXIMACIÓN SUPERVISADA BASADA EN APRENDIZAJE: Igualmente conocida como aproximación basada en [AA](#) o métodos estadísticos para la clasificación de sentimientos. Consiste en algoritmos que aprenden los patrones subyacentes de los datos de entrenamiento — datos cuya clase o etiqueta se conoce para cada instancia — para después intentar clasificar nuevos datos suministrados al algoritmo, esta vez sin estar etiquetados. Los pasos a seguir en una aproximación de este tipo consisten en realizar algo de ingeniería de características que respresenten el objeto cuya clase se quiere predecir. Tras esto, se usan dichas representaciones como como entrada del algoritmo. Algunas de las características más usadas en al análisis de sentimientos son: frecuencia del término, categorías morfosintáctica – *POS tags* – palabras y frases con sentimientos, reglas de opinión, modificadores del sentimiento y dependencias sintácticas, por mencionar algunas [17].

APROXIMACIÓN BASADA EN CONCEPTOS: Relativamente moderna. Consiste en usar *ontologías* para apoyar la tarea del análisis de sentimientos. Las ontologías suelen presentarse como gráfos donde los conceptos se asocian a nodos enlazados por relaciones. El estudio realizado por Zhou y Chaovalit [37] analiza en profundidad las ontologías, así como sus aplicaciones y desarrollo.

Una de las ventajas de usar métodos no supervisados reside en la no dependencia de grandes cantidades de datos para entrenar a los algoritmos. Aún así, sigue siendo necesaria la construcción u obtención de un léxico para los sentimientos. Los métodos no supervisados son menos dependientes del dominio que los métodos supervisados. De hecho, los clasificadores entrenados en un dominio específico muestran de forma consistente peor comportamiento cuando son ejecutados en otros dominios [2].

Una ontología se define como un modelo que conceptualiza el conocimiento de un dominio dado, de tal forma que pueda ser comprendido tanto por humanos como máquinas.

1.5 EL PIPELINE DE CORENLP

En la [Sección 1.4](#) se ha visto el proceso genérico a seguir para problemas de *NLP*, se presenta ahora una breve descripción de los pasos que se mostraron en la [Tabla 1](#), correspondientes al *pipeline* para un software concreto — CORENLP [18] — Como se aprecia, dicho *pipeline* solo está totalmente completo para el Inglés.

CORENLP viene empaquetado con los modelos para Inglés. Es posible descargar modelos para distintos idiomas, pero por separado. El soporte para estos idiomas no es completo. Los pasos del *pipeline* mencionados a continuación se centran en la versión para el Inglés. En [Otras aproximaciones](#) se citaron los distintos modos de realizar tareas para *NLP*, los modelos de CORENLP entrenan modelos usando tanto métodos de [AA](#) supervisados como basándose en reglas.

TOKENIZADOR: *Tokeniza* el texto en secuencias de símbolos. El componente para el Inglés proporciona un *tokenizador* al estilo *Penn Tree Bank* (PTB) que ha sido ampliado para tratar con texto de la web y con ruido. Los componentes correspondientes para el Chino y Árabe proporcionan segmentación de palabras y *clitic*. Como proceso final, el *tokenizador* almacena los desplazamientos de caracteres de cada símbolo en el texto de entrada.

CLEANXML: Elimina las etiquetas *Extensible Markup Language* (XML) del documento.

SSPLIT: Separa una secuencia de símbolos en frases.

TRUECASE: Determina la probabilidad de un símbolo de estar en mayúscula en el texto — es decir, la probabilidad de que, en un texto bien escrito, dicho símbolo debiera estar en mayúscula. — cuando esta información no está disponible, por ejemplo, un texto con todas las letras en mayúsculas. Este proceso se implementa con un modelo discriminativo usando un etiquetador de secuencias CRF [8].

POS: Etiqueta símbolos con su correspondiente categoría morfosintáctica – *POS tag* – haciendo uso de un etiquetador de máxima entropía [27].

LEMMA: Genera la raíz – o *lemma* – para todos los símbolos en la anotación.

GENDER: Añade información sobre el género más probable a los nombres.

NER: Reconoce nombres — PERSONA, LUGAR, ORGANIZACIÓN, MISCELÁNEA — y entidades numéricas — DINERO, NÚMERO, FECHA, HORA, DURACIÓN. — En los etiquetados por defecto, las entidades para nombres se reconocen mediante una combinación de secuencias de CRFs entrenados en varios *corpus* [8]. Las entidades numéricas son reconocidas usando dos sistemas basados en reglas, uno para el dinero y números, y otro sistema *estado del arte* para procesar expresiones temporales [5].

REGEXNER: Implementa un NER simple basado en reglas usando secuencias de símbolos mediante expresiones regulares. El objetivo de este paso del *pipeline* es proporcionar un *framework* que permita al usuario incorporar etiquetas NE que no están anotadas en *corpus* NL tradicionales. Por ejemplo, la lista por defecto de expresiones regulares distribuida en los modelos de CORENLP reconoce ideologías, nacionalidades, religiones y títulos.

PARSE: Proporciona un análisis sintáctico completo, incluyendo representaciones tanto de dependencias como constituyentes. Está basado en un parseador probabilístico [13, 19].

SENTIMENT: Análisis de sentimientos con un modelo compositivo sobre árboles usando *deep learning* [25]. La puntuación asignada a un sentimiento se calcula mediante los nodos de un árbol binario para cada sentencia, incluyendo, en particular, el nodo raíz de cada frase.

DCOREF: Detecta menciones y resolución de coreferencias tanto pronominales como nominales [15]. Se devuelve el grafo de coreferencia del texto al completo, con las palabras principales de las menciones como nodos.

En la Figura 3 se muestra el *pipeline* completo de CORENLP.

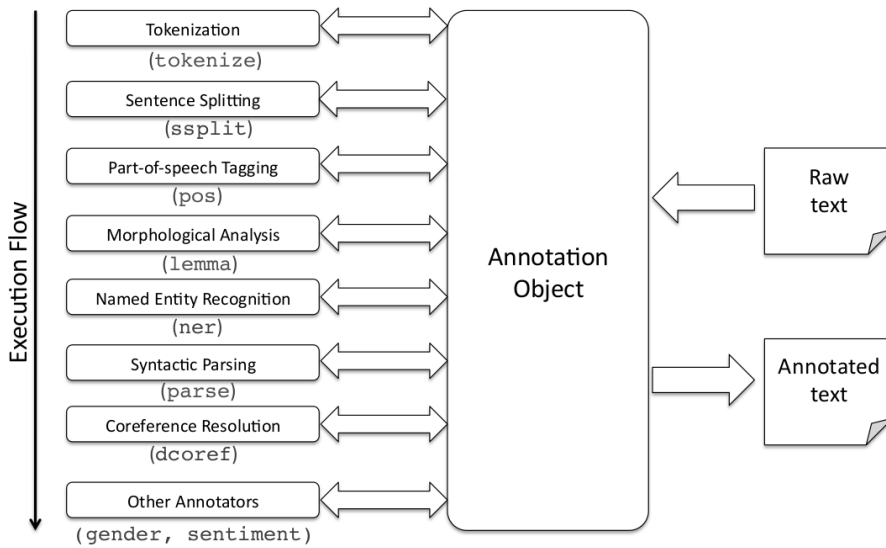


Figura 3: Visión general de la arquitectura. El texto a procesar se añade a un objeto ANNOTATION. Posteriormente una secuencia de etiquetadores añaden información sobre qué fases del *pipeline* deben ejecutarse. El resultado contiene el texto procesado, puede obtener en formato XML o texto plano.

1.6 ESTADO DEL ARTE

En esta sección se describirán las aproximaciones actuales en el análisis de sentimientos, en función de los distintos niveles mencionados en Niveles de análisis.

1.6.1 A nivel del documento

Turney [30] presentó un método no supervisado para clasificar reseñas como recomendadas o no recomendadas. Su investigación de-

Semantic
Orientation (SO):
Orientación
Semántica

termina las polaridades mediante una media de la *Semantic Orientation* (SO) de las frases que aparecen en las reseñas. La SO de un par de frases se calcula mediante *Pointwise Mutual Information and Information Retrieval* (PMI-IR) [29], usando la Ecuación 1

$$SO(\text{frase}) = \log \frac{\text{hits}(\text{frase}, \text{"excelente"})\text{hits}(\text{"mediocre"})}{\text{hits}(\text{frase}, \text{"mediocre"})\text{hits}(\text{"excelente"})} \quad (1)$$

Pang, Lee y Vaithyanathan [21] emplearon métodos de AA para clasificar reseñas como positivas o negativas. Para ello entrenaron los siguientes clasificadores: naïve Bayes, máxima entropía y *Support Vector Machine* (SVM) usando las siguientes características: unigrama, bigrama, POS tag y la posición de la palabra. Los autores indicaron que las ocurrencias de opiniones contradictorias hacen del análisis de sentimientos para documentos una tarea más difícil que la categorización de texto basada en temas – *topic-based*. –

1.6.2 A nivel de sentencia

Yu y Hatzivassiloglou [36] presentaron tres aproximaciones para clasificar la subjetividad, basada en la similitud de sentencias, naïve Bayes y varios clasificadores naïve Bayes, respectivamente. La primera aproximación, basada en similitud, calcula una puntuación de similitud de las frases y las compara con un umbral predeterminado. La aproximación usando naïve Bayes entrena un clasificador con las sentencias de documentos de opinión y fácticos. Las características son unigramas, bigramas, trigramas POS tags y un recuento de palabras expresando opinión. Los múltiples clasificadores naïve Bayes entrena a los distintos clasificadores en diferentes sub conjuntos del espacio de características. Posteriormente las sentencias que tienen etiquetas diferentes con los correspondientes documentos se eliminan del conjunto de test. El proceso de entrenamiento y testeo es iterativo hasta que no haya más frases en el conjunto de test a borrar. Finalmente, los clasificadores entrenados en el conjunto de entrenamiento reducido son los usados para determinar si las sentencias son subjetivas u objetivas.

Aunque existen varias propuestas sobre el parseo de dependencias para Español, quizá una de las más interesantes y con mejores resultados sea la de Ballesteros y col. [3]. Se basan en el estado del arte para el parseo sintáctico de dependencias, que produce como resultado un *Surface-Syntactic structures* (SSyntSs), y en el parseo semántico de dependencias – *Preprocesamiento del texto* –, el cual proporciona estructuras semánticas. El inconveniente de la primera estructura es que se define sobre el vocabulario al completo de un idioma. Además posee funciones gramaticales específicas del idioma. La segunda estructura, es problemática al perder información sobre la estructura lingüística. Por esta razón, Ballesteros y col. definen *Deep-Syntactic*

Structures (*DSyntSs*), más apropiada para tareas de *NLP*, ya que se sitúa en mitad de las dos estructuras anteriores. La finalidad de su trabajo consiste en generar un *DSyntSs* a partir de un *SSyntSs* usando un parseador sintáctico, traduciendo de una estructura a otra.

1.6.3 *A nivel de entidad*

Thet, Na y Khoo [26] propusieron una aproximación lingüística aprovechando la estructura de dependencia gramatical de las cláusulas. La estructura de la dependencia gramatical se obtiene a través de un árbol sintáctico, para luego extraer sub-árboles que representan cláusulas — por ejemplo, dividiendo una frase en oraciones separadas que expresan opiniones hacia el aspecto correspondiente. — La generación del sub-árbol puede considerarse como una descomposición de una sentencia compleja en varias simples, transformando el problema original a una tarea a nivel de sentencias. La extracción del aspecto en este trabajo se simplifica a un proceso de etiquetado semántico. En este proceso se asigna una etiqueta a cada oración mediante un conjunto de aspectos predefinidos indicando palabras.

Wang y col. [34] usaron un modelo no supervisado, basado en máquinas de Boltzmann restringidas. El modelo propuesto extrae conjuntamente sentimiento y aspecto. Además el modelo incorpora como variables latentes las variables aspecto, sentimiento y demás variables de fondo. Sin embargo, estas variables latentes no poseen una relación condicional, es decir, no hay aristas conectándolas en el modelo gráfico.

Parte II

OBJETIVOS

PARSEO DE DEPENDENCIAS EN ESPAÑOL

A continuación se listan los objetivos previstos del trabajo.

REVISIÓN BIBLIOGRÁFICA DEL ESTADO DEL ARTE Y ANTECEDENTES DEL PARSEO DE DEPENDENCIAS EN ESPAÑOL. Este objetivo pretende explorar los métodos existentes que realizan parseado de dependencias, para adquirir un conocimiento previo del [Estado del arte](#) en la literatura. Así como conocer los métodos existentes para el parseo de dependencias en Español [3].

ELECCIÓN Y ANÁLISIS DE REQUERIMIENTOS DE UN PROCEDIMIENTO APROPIADO DE PARSEO DE DEPENDENCIAS Y DISEÑO PARA SCALA Debido a la popularidad del lenguaje de programación SCALA en el área del [AA](#) y la minería de datos, se pretende implementar este trabajo bajo dicho lenguaje. El algoritmo de parseo de dependencias elegido se ha basado en Yamada y Matsumoto [35] y Jain [12]. Una introducción al lenguaje de programación SCALA puede consultarse en el [Capítulo 3](#). La descripción del algoritmo propuesto por Yamada y Matsumoto se detalla en el [Capítulo 4](#).

IMPLEMENTACIÓN Y PROCESOS DE PRUEBA DEL ALGORITMO ESCOGIDO Aquí se implementará el algoritmo escogido para el parseado de dependencias. Así mismo, se pondrá en práctica la técnica de desarrollo al estilo *Test-Driven Development* ([TDD](#)), en concreto se usará *Behavior-Driven Development* ([BDD](#)) orientado a problemas de [AA](#). La implementación se encuentra en el [Capítulo 5](#), mientras que las pruebas pertinentes pueden encontrarse en el [Capítulo 6](#).

EVALUACIÓN DEL ALGORITMO, COMPARACIÓN Y DISCUSIÓN DE RESULTADOS OBTENIDOS EN CASOS PRÁCTICOS Con este objetivo se pretende mostrar los resultados obtenidos con la implementación realizada, así como una comparación de los resultados en la implementación original. Estos resultados se discutirán en la [Sección 7.2](#).

Parte III

RESOLUCIÓN DEL TRABAJO

Esta parte del trabajo se centra en los pasos seguidos para el desarrollo del proyecto. Se organiza como sigue:

En el [Capítulo 3](#) se discuten las características del lenguaje SCALA, y por qué ha sido escogido para el desarrollo.

El [Capítulo 4](#) detalla los aspectos técnicos del algoritmo de parseo de dependencias seleccionado.

En el [Capítulo 5](#) se narra las distintas etapas seguidas para el desarrollo, desde el análisis hasta la implementación.

Por último, el [Capítulo 6](#) explica los distintos casos de prueba llevados a cabo.

UNA INTRODUCCIÓN A SCALA

El nombre SCALA es una concatenación de dos palabras, *Scalable Language*. A continuación se enumeran algunas de las razones por las que se ha elegido este lenguaje, la lista completa puede encontrarse en Wampler y Payne [32].

3.1 ¿POR QUÉ SCALA?

Las principales características de SCALA que lo hacen un buen candidato para este trabajo son las siguientes:

PARADIGMA MIXTO — PROGRAMACIÓN ORIENTADA A OBJETOS:

SCALA soporta al completo el paradigma de la orientación a objetos. Además, mejora el modelo de objetos proporcionado por JAVA con la introducción de TRAITS, un modo muy claro de implementar tipos mediante composiciones mixtas. Todo es un objeto en SCALA, incluso los tipos numéricos.

PARADIGMA MIXTO — PROGRAMACIÓN FUNCIONAL: De igual modo, SCALA soporta al completo *Functional Programming* (FP). En los últimos años, la FP ha resurgido como una de las mejores herramientas para pensar en problemas de concurrencia, *Big Data* y en general para escribir código correcto. Este código correcto, conciso y potente se logra mediante el uso de valores inmutables, funciones de primera clase, funciones sin efectos colaterales, funciones de “orden superior” y colecciones de funciones.

FP: Programación
Funcional

SINTAXIS BREVE, ELEGANTE Y FLEXIBLE: Expresiones que pueden llegar a ser demasiado extensas en JAVA se hacen concisas en SCALA.

ARQUITECTURA SCALABLE: SCALA permite escribir desde *scripts* pequeños, que son interpretados, hasta aplicaciones distribuidas de gran envergadura. Hay cuatro mecanismos inherentes al lenguaje permitiendo esta escalabilidad: 1) composiciones mixtas mediante TRAITS, 2) miembros de tipo abstracto y genéricos; 3) anidamiento de clases y 4) tipos explícitos SELF.

3.2 PATRONES DE DISEÑO COMO MIEMBROS DE PRIMERA CLASE

Otra de las ventajas de SCALA es que incorpora ciertos patrones de diseño en el mismo lenguaje. Por ejemplo, en [Patrón Singleton en Scala](#), se muestra una implementación del patrón.

Código 1 Patrón Singleton en Scala.

```
object Upper {
  def upper(strings: String*) = strings.map(_.toUpperCase())
}
println(Upper.upper("Hello", "World!"))
```

En la primera línea, **object** Upper crea el objeto *singleton*. El patrón *singleton* tiene sentido cuando no es necesario guardar ningún estado del objeto ni el objeto interactúa con el mundo exterior.

Es posible refactorizar el código anterior, como se muestra en [Refactorizando Upper](#)

Código 2 Refactorizando Upper

```
object Upper2 {
  def main(args: Array[String]) = {
    val output = args.map(_.toUpperCase()).mkString(" ")
    println(output)
  }
}
```

3.3 VENTAJAS DEL USO DE TRAITS

Uno de los errores más comunes cuando se usa Programación Orientada a Objetos es abusar de la herencia. Este error suele cometerse cuando se crean relaciones de herencia para añadir estados a las clases derivadas. Para ilustrarlo con un ejemplo – Wampler y Payne [32] – podemos pensar en dos clases, PERSONA y TRABAJADOR. Podría pensarse en establecer una relación de herencia entre PERSONA y TRABAJADOR, pero entonces estaríamos aplicando herencia para añadir un ESTADO a un objeto, este no es el objetivo de la herencia. Una mejor alternativa sería crear subclases del comportamiento de un objeto con el mismo estado. De esta forma, se favorece la composición sobre la herencia, es decir, componer unidades de funcionalidades en lugar de crear jerarquías de clases. Esto se logra mediante el uso de TRAITS – [Sección 3.1](#) – El código quedaría como en [Composición en lugar de herencia](#). El diagrama de clases para el Código 3 sería el de la [Figura 4](#).

Código 3 Composición en lugar de herencia

```

trait PersonState {
  val name: String
  val age: Option[Int]
  val address: Option[Address]
}

case class Person(
  name: String,
  age: Option[Int] = None,
  address: Option[Address] = None) extends PersonState

trait EmployeeState {
  val title: String
  val manager: Option[Employee]
}

case class Employee(
  name: String,
  age: Option[Int] = None,
  address: Option[Address] = None,
  title: String = "[unknown]",
  manager: Option[Employee] = None)
extends PersonState with EmployeeState

```

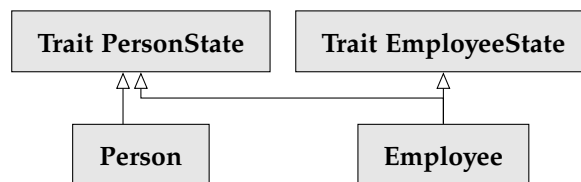


Figura 4: Diagrama de clases PERSONA y TRABAJADOR

Este diseño ha sido el seguido para el proyecto, en concreto, para las estructuras de datos que representan las frases del conjunto de test y train, cuyo diagrama se puede consultar en la [Figura 13](#).

3.4 REGLAS DE VISIBILIDAD

SCALA ofrece un amplio abanico de posibilidades en cuanto a la visibilidad a la que se exponen las clases, objetos y métodos. Este abanico es mucho más amplio que el ofrecido por JAVA y se puede consultar en la [Tabla 2](#)

Tabla 2: Reglas de visibilidad en SCALA

Nombre	Palabra reservada	Descripción
public	ninguna, por defecto	Visibles en cualquier lugar.
protected	protected	Visibles al tipo que los define, tipos derivados y tipos anidados. Solo son visibles dentro del mismo paquete y sub paquetes.
private	private	Solo en los tipos que los definen y tipos anidados. A nivel de paquete son visibles únicamente en el mismo que los define.
scoped protected	protected [scope]	Visibilidad delimitada por el ámbito definido por SCOPE. SCOPE puede ser un paquete, tipo o this — misma instancia —
scoped private	private [scope]	Similar al anterior, pero los objetos derivados no pueden acceder.

3.5 BIG DATA

La principal razón del éxito de SCALA en la comunidad del [AA](#) y *Big Data* reside en la facilidad que ofrece para escribir programas concurrentes usando el paradigma [FP](#). La diferencia entre aplicaciones de *Big Data* escritas en JAVA frente a las escritas en SCALA mediante [FP](#) es abismal. En [Refactorizando Upper](#) se vio un ejemplo de map, funciones como esta y sus compañeras — flatMap, filter, fold... — han sido siempre herramientas para trabajar con datos. Independientemente del tamaño de los datos, se aplica la misma abstracción.

Como ejemplo mostraremos dos versiones del clásico HOLA MUNDO de MAP REDUCE, una en JAVA — [4](#) — y otra en SCALA — [5](#) — Los ejemplos son del libro de Wampler y Payne [[32](#)]. Basta observar el código para notar la verbosidad de JAVA frente a SCALA.

Código 4 WORDCOUNT en JAVA

```

class WordCountMapper extends MapReduceBase
implements Mapper<IntWritable, Text, Text, IntWritable> {

    static final IntWritable one = new IntWritable(1);
    // Value will be set in a non-thread-safe way!
    static final Text word = new Text();

    @Override
    public void map(IntWritable key, Text valueDocContents,
OutputCollector<Text, IntWritable> output, Reporter reporter) {
        String[] tokens = valueDocContents.toString().split("\\s+");
        for (String wordString: tokens) {
            if (wordString.length > 0) {
                word.set(wordString.toLowerCase());
                output.collect(word, one);
            }
        }
    }
}

class WordCountReduce extends MapReduceBase
implements Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text keyWord, java.util.Iterator<IntWritable>
↪ counts,
OutputCollector<Text, IntWritable> output, Reporter reporter) {
        int totalCount = 0;
        while (counts.hasNext()) {
            while (counts.hasNext()) {
                totalCount += counts.next.get();
            }
            output.collect(keyWord, new IntWritable(totalCount));
        }
    }
}

```

Código 5 WORDCOUNT en SCALA

```

class ScaldingWordCount(args : Args) extends Job(args) {
    TextLine(args("input"))
        .read
        .flatMap('line -> 'word) {
            line: String => line.trim.toLowerCase.split("\\s+")
        }
        .groupBy('word){ group => group.size('count) }
        .write(Tsv(args("output")))
}

```

Podríamos decir que estos son los motivos principales – aunque no los únicos – de la elección de este lenguaje de programación en el desarrollo de un parseador de dependencias para Español.

ALGORITMO SELECCIONADO: STATISTICAL DEPENDENCY ANALYSIS WITH SUPPORT VECTOR MACHINES

Yamada y Matsumoto [35] proponen un método para analizar las dependencias palabra-a-palabra mediante una estrategia *bottom-up* – de abajo a arriba. – Para ello se hace uso de la técnica de *AA Support Vector Machine* (SVM). Sus experimentos se basan en árboles de dependencias creados a partir del corpus PTB, logrando una precisión superior al 90 % para dependencias palabra-a-palabra. Aún siendo esta precisión inferior al *Estado del arte*, hay que tener en cuenta que este método no utiliza información sobre la estructura de las frases.

El tipo de anotaciones usadas en este método pueden verse en la Figura 6. Esta forma de ilustrar las dependencias palabra-a-palabra es más sencilla de entender para los anotadores que el usual estilo PTB — Figura 5 — El problema del estilo PTB es que requiere que los anotadores tengan un amplio conocimiento de la teoría lingüística del idioma, así como de la estructura de las frases, además del dominio específico que trata el problema. Como ventaja adicional, la representación del árbol de dependencias de la Figura 6 hace que la construcción de los datos de entrenamiento sea menos ruidosa, al ser el proceso de anotación más simple. En la estructura propuesta por

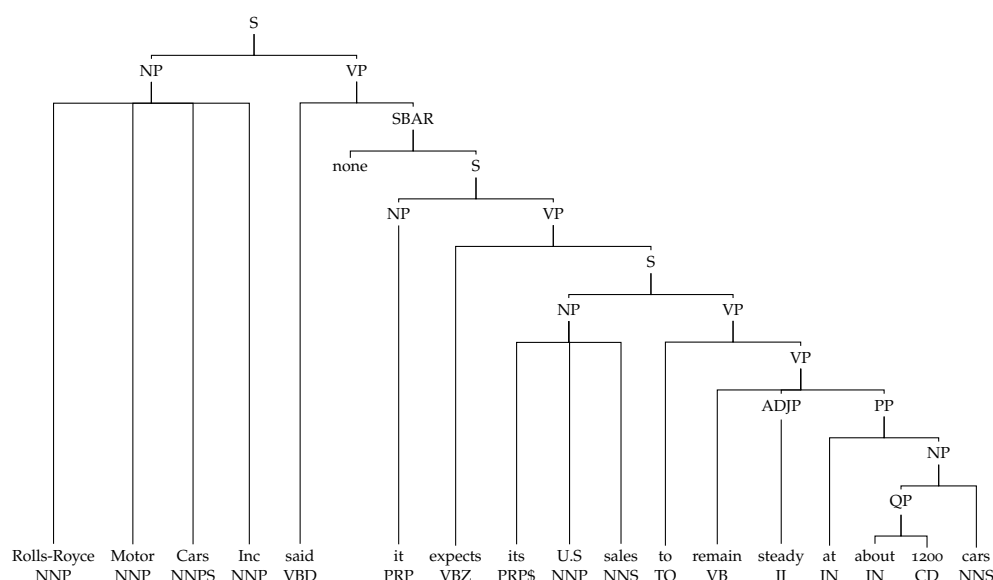


Figura 5: Estructura en árbol de la frase "Rolls-Royce Motor Cars Inc. said it expects its U.S. sales to remain steady at about 1,200 cars."

Yamada y Matsumoto se realiza un análisis estadístico de las depen-

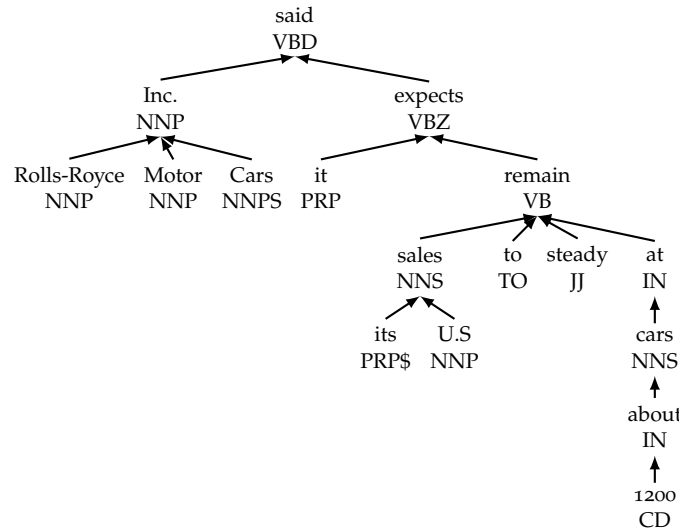


Figura 6: Árbol de parseo de dependencias

dencias de un idioma. Para lograr este análisis, se usa la técnica de aprendizaje conocida como [SVM](#), ya que es capaz de tratar con espacios de características a gran escala. En los siguientes párrafos se hará una breve introducción a esta técnica.

4.1 UNA INTRODUCCIÓN A LAS SVMs

Los modelos lineales son muy poderosos, mediante transformaciones lineales es posible incrementar en gran medida su capacidad expresiva. Sin embargo, incrementar dicha capacidad tiene un precio, el sobre ajuste y más tiempo de cómputo. [SVM](#) usa un cojín de seguridad cuando separa los datos. Este cojín logra que la [SVM](#) sea más robusta al ruido, reduciendo así el sobre ajuste. Además, [SVM](#) es capaz de trabajar con la herramienta conocida como *kernel* — la cual permite operar de forma eficiente con transformaciones no lineales de gran dimensión —. Estas dos características — el cojín de seguridad y el *kernel* — hacen de la [SVM](#) un modelo no lineal muy robusto y potente con regularización automática. Las [SVM](#) son muy populares por su facilidad de uso y su buen rendimiento.

[SVM](#) usa la estrategia del máximo margen ideada por *Vapnik*. Supongamos l datos de entrenamiento (\mathbf{x}_i, y_i) , $(1 \leq i \leq l)$, donde \mathbf{x}_i es un vector de características en un espacio de dimensionalidad n , y_i es la etiqueta de la clase $\{-1, +1\}$ de \mathbf{x} . Las [SVMs](#) encuentran un hiperplano $\mathbf{w} \cdot \mathbf{x} + b = 0$ que separe correctamente los datos de entrenamiento de forma que tengan margen máximo, es decir, con la máxima distancia entre dos hiperplanos $\mathbf{w} \cdot \mathbf{x} + b \geq 1$ y $\mathbf{w} \cdot \mathbf{x} + b \leq -1$, como se aprecia en la [Figura 7](#). Los puntos negros representan la clase negativa, los blancos la positiva y los que se sitúan justo en las líneas del margen se llaman puntos de soporte vectoriales.

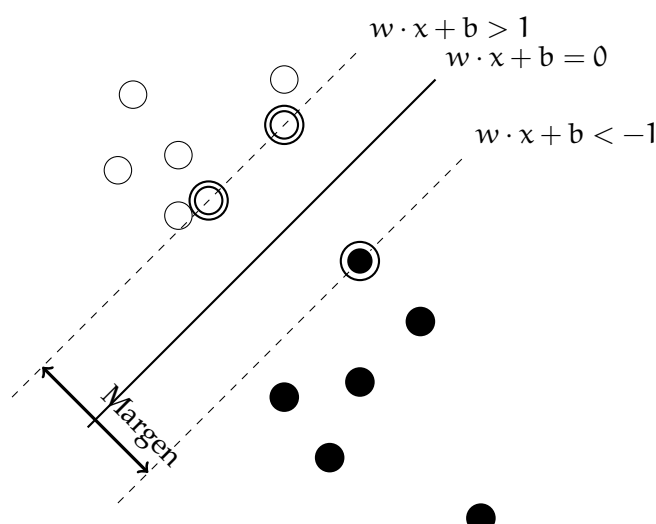


Figura 7: Ejemplo SVM

El uso de SVM para el análisis estadístico de dependencias presenta principalmente dos ventajas. La primera de ellas es un gran poder de generalización en espacios de características de grandes dimensiones. Segunda, gracias al *kernel trick* es posible entrenar al modelo para que aprenda a partir de la combinación de múltiples características.

Para poder trabajar con clasificaciones no lineales uno de los *kernels* posibles es el polinomial $(x' \cdot x'' + 1)^d$. Con este *kernel* se habilita la posibilidad de tener en cuenta combinaciones de d características sin incrementar demasiado el tiempo de cálculo. En el problema que nos ocupa, esto se traduce en la capacidad de entrenar reglas de dependencias usando varias características, como POS tags, las palabras en sí y sus combinaciones.

4.2 ANÁLISIS DE DEPENDENCIAS DETERMINÍSTICO

4.2.1 Acciones para el parseo

El trabajo de Yamada y Matsumoto propone tres acciones para construir el árbol de dependencias. Es aquí cuando entra en juego la SVM, ya que aprenderá las acciones de los datos de entrenamiento y predecirá qué acciones realizar para datos nuevos. El árbol de dependencias se construye de izquierda a derecha, siguiendo la dirección de lectura habitual. Las tres acciones posibles son *Shift*, *Right* y *Left* — DESPLAZAR, DERECHA e IZQUIERDA. — Estas acciones se aplican a dos palabras vecinas, nombradas a partir de ahora nodos objetivo. Se pasa ahora a describir cada una de las acciones.

DESPLAZAR indica que no se ha realizado ninguna construcción en el árbol de dependencias entre los nodos objetivo. La acción para esta situación simplemente desplaza una posición a la derecha la venta-

na que apunta a los nodos objetivo. En la figura [Figura 8](#) muestra un ejemplo concreto — Los ejemplos han sido extraídos de Yamada y Matsumoto [35] —

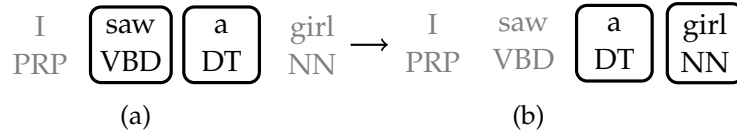


Figura 8: Ejemplo de la acción DESPLAZAR. (a) muestra el estado antes de aplicar la acción. (b) el resultado tras aplicarla

DERECHA construye una relación de dependencia entre los nodos objetivo. De los dos nodos objetivo, el de la izquierda pasa a ser hijo del nodo de la derecha. En la [Figura 9](#) puede observarse el efecto de esta acción. IZQUIERDA construye una relación de dependencia entre

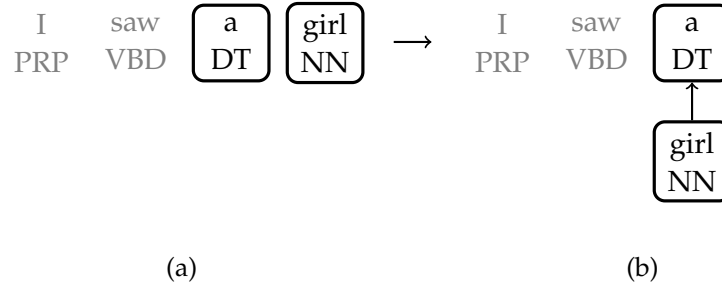


Figura 9: Ejemplo de la acción DERECHA. (a) Estado antes de la acción. (b) Estado tras aplicar la acción

los nodos objetivo. De los dos nodos objetivo, el de la derecha pasa a ser hijo del nodo de la izquierda. En la [Figura 10](#) se ilustra esta situación.

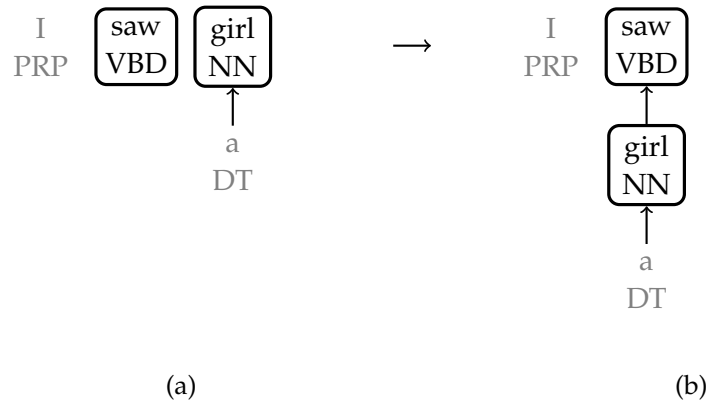


Figura 10: Ejemplo de la acción LEFT. (a) Estado antes de la acción. (b) Estado tras aplicar la acción

4.2.2 Descripción del algoritmo

El Algoritmo 1 consiste de dos partes. Primero se estima la acción apropiada usando la información contextual rodeando los nodos objetivo. Segundo, se construye el árbol de dependencias ejecutando las acciones estimadas en el primer paso.

Algoritmo 1 Algoritmo de parseo

```

1: Input Sentence:  $(w_1, p_1), (w_2, p_2), \dots, (w_n, p_n)$ 
2: Initialize:
3:    $i \leftarrow 1$ 
4:    $\mathcal{T} \leftarrow \{(w_1, p_1), (w_2, p_2), \dots, (w_n, p_n)\}$ 
5:    $\text{no\_construction} \leftarrow \text{true}$ 
6: Start
7:   while  $|\mathcal{T}| \geq 1$  do
8:     if  $i == |\mathcal{T}|$  then
9:       if  $\text{no\_construction} == \text{true}$  then break
10:      end if
11:       $\text{no\_construction} \leftarrow \text{true}$ 
12:       $i \leftarrow 1$ 
13:    else
14:       $\mathbf{x} \leftarrow \text{getContextualFeatures}(\mathcal{T}, i)$ 
15:       $\mathbf{y} \leftarrow \text{estimateAction}(\text{model}, \mathbf{x})$ 
16:       $\text{construction}(\mathcal{T}, i, \mathbf{y})$ 
17:      if  $\mathbf{y} == \text{Left or Right}$  then  $\text{no\_construction} \leftarrow \text{false}$ 
18:      end if
19:    end if
20:  end while

```

Cuando el algoritmo se ejecuta, la variable i representa el índice del nodo de la izquierda del par de nodos objetivos, $i + 1$ el de la derecha, dichos nodos están en \mathcal{T} . \mathcal{T} contiene la secuencia de nodos a los que se debe estimar una acción, estos nodos se corresponden con los nodos raíz de los árboles que se van construyendo a lo largo del proceso de parseado. Como es lógico, inicialmente todos los nodos $t_i \in \mathcal{T}$ están compuestos únicamente por la raíz, sin tener hijos. La información que guarda cada nodo es el par (w_i, p_i) , donde w_i es una palabra y p_i su [POS tag](#).

La estimación sobre qué acción aplicar se lleva a cabo mediante las funciones *getContextualFeatures* y *estimateAction*. La primera extrae características de \mathbf{x} en función del contexto que la rodea por i , en la sección [Extracción de características](#) se profundiza sobre este tema. La segunda función estima la acción más apropiada.

La variable *no_construction* se encarga de comprobar cuando se han producido acciones que hayan resultado en la contrucción de dependencias al terminar de leer la frase. Cuando esta variable tiene valor

verdadero significa que se ha producido la acción DESPLAZAR para todos los nodos objetivo, y por tanto no se han creado nuevas dependencias. En ese caso se detiene el proceso y se devuelven los árboles en \mathcal{T} , ya que no es posible aplicar más acciones. Cuando la variable es falsa, los nodos objetivo se colocan al principio de la frase y se vuelve a repetir el proceso hasta que $|\mathcal{T}| = 1$.

4.2.3 Extracción de características

En el proceso de entrenamiento de las SVM, cada estimación de una acción y en el contexto x se corresponde con una observación (x, y) de la SVM. Al tener tres tipos de acciones, el problema que nos ocupa es de clasificación multi clase. Para resolverlo se crean tres clasificadores binarios correspondientes a cada acción, lo cual se conoce como método por parejas. Los tres clasificadores binarios son IZQUIERDA vs. DERECHA, IZQUIERDA vs. DESPLAZAR y DERECHA vs. DESPLAZAR.

Como se mencionó en la Subsección 4.2.2, el POS tag y la propia palabra se usan como características de los nodos para los contextos de la izquierda y derecha. Se pasa ahora a describir en qué consisten dichos contextos.

Como ya se sabe, i e $i + 1$ son los índices que apuntan los nodos objetivo en \mathcal{T} . El contexto a la izquierda se puede definir como los nodos posicionados a la izquierda de los nodos objetivo, es decir t_l , ($l < i$). El contexto a la derecha, de forma análoga, son los nodos a la derecha de los nodos objetivo t_r , ($i + 1 < r$)

La representación de una característica viene dada por la tupla (p, k, v) . p es la posición partiendo de los nodos objetivo, k codifica el tipo de característica y v almacena el valor para dicha característica. Cuando $p < 0$ se está representando el nodo del contexto izquierdo, $p = \{0-, 0+\}$ representa el nodo izquierdo (0-) o derecho (0+) de los nodos objetivo — recordemos que los nodos objetivo estaban formados por dos nodos. — Análogamente, $p > 0$ indica los nodos en el contexto derecho. La Tabla 3 muestra los valores del tipo de característica k y sus valores v . Las características $ch\{-L, R\}\{-pos, lex\}$ se denominan características hijas y se calculan de forma dinámica durante el análisis. Estas características ayudan a determinar qué acción tomar.

4.2.4 Agrupando SVMs para reducir costes

Debido a la gran cantidad de datos de entrenamiento disponibles, Yamada y Matsumoto proponen dividirlos en varios grupos. La división se realiza en base al POS tag del nodo de la izquierda de los nodos objetivo. Por ejemplo, si el POS tag del nodo izquierdo es “VB”, se estima la acción usando $SVMs^{VB}$

Tabla 3: Descripción del tipo de características y sus valores

Tipo	Valor
pos	<i>POS tag</i>
lex	La palabra
ch-L-pos	<i>POS tag</i> del nodo hijo modificando al padre desde la izquierda
ch-L-lex	Palabra del correspondiente ch-L-pos
ch-R-pos	<i>POS tag</i> del nodo hijo que modifica al padre desde la derecha
ch-R-lex	Palabra del correspondiente ch-R-pos

4.3 EJEMPLO PRÁCTICO

Con las figuras de arriba quizá no haya que poner esta sección.

IMPLEMENTACIÓN

En este capítulo se detalla todo el proceso llevado a cabo para el desarrollo del proyecto, desde la planificación hasta la finalización del mismo. La implementación del algoritmo se ha realizado en SCALA, los detalles del mismo pueden encontrarse en el [Capítulo 4](#). Así mismo, las ventajas del desarrollo en SCALA pueden consultarse en el [Capítulo 3](#).

5.1 PLANIFICACIÓN

En la [Figura 11](#) se muestra un diagrama de *Gantt* con la planificación ideada para el proyecto

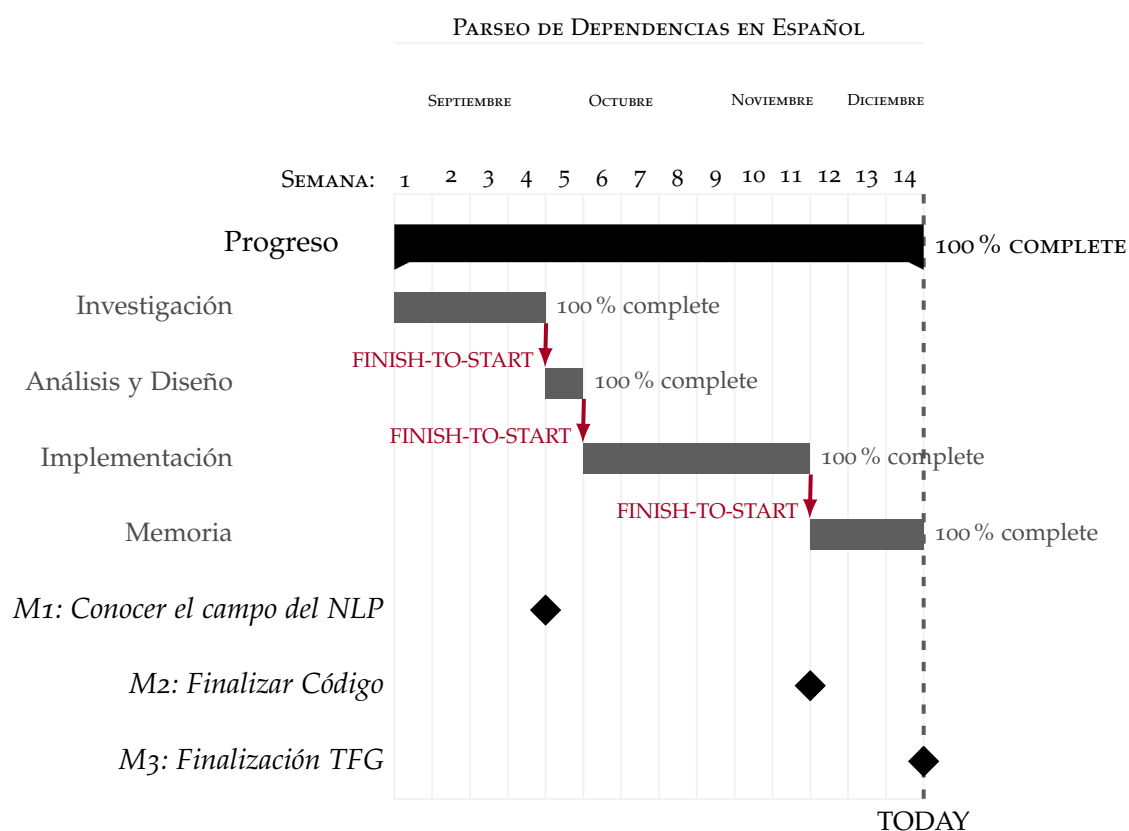


Figura 11: Planificación del proyecto

5.2 ANÁLISIS Y DISEÑO

Los paquetes creados se organizan según la [Figura 12](#).

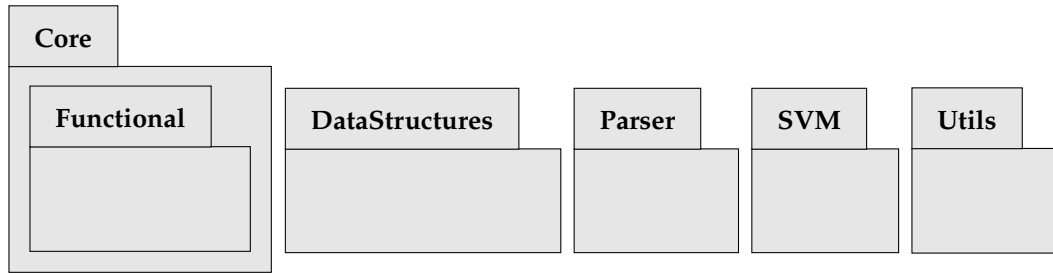


Figura 12: Paquetes del proyecto

En el paquete `CORE.FUNCTIONAL` se definen algunas estructuras de teoría de categorías, actualmente solo hay implementada una mónada – *monads* –.

En `DATASTRUCTURES` se definen las estructuras de datos necesarias para el desarrollo del proyecto, entre otras, aquí se definen las representaciones de las frases para *training* y *test* vistas en la [Figura 13](#). En el [Código 6](#) se listan algunas de las estructuras más relevantes.

Código 6 Estructuras de datos más relevantes del paquete `DATASTRUCTURES`

```
// Información sobre un nodo
case class Node(lex: String,
                position: Int,
                posTag: String,
                var dependency: Int = -1,
                var left: Vector[Node],
                var right: Vector[Node])

// Encargada de almacenar las características para la SVM
final case class Vocabulary(positionVocab: Map[Int, Counter],
                           positionTag: Map[Int, Counter],
                           chLVocab: Map[Int, Counter],
                           chLTag: Map[Int, Counter],
                           chRVocab: Map[Int, Counter],
                           chRTag: Map[Int, Counter])
```

`PARSER` es el paquete principal, contiene la implementación del algoritmo de parseo de dependencias estadístico de Yamada y Matsumoto.

`SVM` encapsula todo lo relacionado con las [SVMs](#), desde el adaptador para los datos hasta la configuración y ajuste de parámetros. Lo más relevante quizá sean los parámetros usados para la [SVM](#), se muestran en el [Código 7](#). Se aprovecha el [Código 7](#) para comentar los parámetros usados:

- `param.svm_type = svm_parameter.C_SVC`: especifica que el tipo de clasificación va a ser multiclase.

Código 7 Parámetros para la SVM

```
object SVMConfig {
  val param = new svm_parameter

  param.svm_type = svm_parameter.C_SVC
  param.kernel_type = svm_parameter.POLY
  param.degree = 2
  param.gamma = 1.0
  param.coef0 = 1.0
  param.cache_size = 4000
  param.eps = 0.001
  param.C = 1.0
  param.shrinking = 1
}
```

- `param.kernel_type = svm_parameter.POLY`: Como se comentó en la [Sección 4.1](#) el *kernel* será de tipo polinómico, de grado 2. El *kernel* se define como $(\gamma \cdot u' \cdot v + \text{coef0})^{\text{degree}}$, cuyos valores pueden consultarse en el código.

UTILS define algunas constantes, tipos de datos, métodos de lectura para los datos de *test* y *training* y encapsula los tres tipos de acciones que puede realizar el parseador. Las acciones se han codificado según el Código 8.

Código 8 Codificación de las acciones DESPLAZAR,IZQUIERDA,DERECHA

```
object Action {

  sealed trait Action

  case object Left extends Action {
    final def value: Int = 0
  }

  case object Shift extends Action {
    final def value: Int = 1
  }

  case object Right extends Action {
    final def value: Int = 2
  }
}
```

El diagrama de clases del proyecto pueden consultarse en la [Figura 13](#)

5.3 IMPLEMENTACIÓN

En esta sección se mostrará en detalle el código implementado del diseño visto en la [Sección 5.2](#). Se comenzará con el código de la cla-

se principal, que implementa el algoritmo de aprendizaje, llamado `DEPENDENCYPARSER`.

5.3.1 DEPENDENCYPARSER

Para facilitar la lectura de la implementación, se dividirá el código en varias partes y se comentarán por separado.

```

1 class DependencyParser(val trainSentences: Vector[LabeledSentence],
2                       val testSentences: Vector[LabeledSentence]) {
3
4   case class Accuracy(private[DependencyParser] val rootAcc: Map[String, Int] =
5     ↳ Map.empty,
6                       private[DependencyParser] val depNAcc: Map[String, Int] =
7     ↳ Map.empty,
8                       private[DependencyParser] val depDAcc: Map[String, Int] =
9     ↳ Map.empty,
10                      private[DependencyParser] val completeD: Int = 0,
11                      private[DependencyParser] val completeN: Int = 0){
12
13     def rootAccuracy: Double = rootAcc.values.sum / testSentences.size.toDouble
14     def dependencyAccuracy: Double = depNAcc.values.sum /
15     ↳ depDAcc.values.sum.toDouble
16     def completeAccuracy: Double = completeN / completeD.toDouble
17   }
18 // ...

```

En la declaración de la clase principal — `DEPENDENCYPARSER` — indica que debe recibir en su constructor dos conjuntos de datos, el de *train* y el de *test*. Así mismo, se declara como clase interna `ACCURACY`, cuya función es calcular las medidas de evaluación – [Sección 7.2.](#) –

El resto de datos miembro de la clase `DEPENDENCYPARSER` son

```

1 //...
2 /**
3  * Train
4  */
5 // 1.1 - Build Vocabulary
6 private[this] val vocabulary = generateVocabulary(trainSentences)
7 // 1.2 - Extract Features
8 private[this] val features = extractFeatures(sentences2)
9 // 1.3 - Train models
10 private[this] val models = train(features._1, features._2)
11
12 val nFeatures = vocabulary.nFeatures
13
14 /**
15  * Test with unseen data
16  */
17 private[this] val inferredTree = test(testSentences)
18 //...

```

El primer paso del algoritmo es calcular el vocabulario a usar en el proceso de entrenamiento, mediante el método `GENERATEVOCABULARY`. El siguiente paso consiste en extraer las características – [Tabla 3](#) – usando el método `EXTRACTFEATURES` – [Subsección 4.2.3](#) –. Por último, solo resta entrenar los modelos mediante el método `TRAIN` usando las características extraídas.

Se procede ahora a mostrar la implementación de `GENERATEVOCABULARY`

```

1 // 1.1 - Build Vocab
2 private[this] def generateVocabulary(sentences: Vector[LabeledSentence]):
  ↳ Vocabulary

```

El modo de calcular el vocabulario usa el Algoritmo 1 propuesto por Yamada y Matsumoto [35]. La estimación de la acción a tomar – Subsección 4.2.1 – se realiza de forma algorítmica, sin involucrar al modelo SVM. Es en este método donde se construye el árbol de dependencias.

Tras construir el vocabulario, se extraen las características con

```

1 // 1.2 - Extract features
2 private[this] def extractFeatures(sentences: Seq[LabeledSentence]): (Map[String,
  ↳ Vector[Vector[Int]]], Map[String, DblVector]) = {

```

El método se encarga de generar las características que le serán proporcionadas a la SVM para entrenarse.

Por último, resta la fase de entrenamiento, de la cual se encarga el método TRAIN. Debido a la importancia de este método, se muestra su código al completo.

```

1 // 1.3 - Train models
2 def train(X: Map[String, Vector[Vector[Int]]], Y: Map[String, DblVector]):
  ↳ Map[String, svm_model] = {
3   logger.info("Training models...")
4
5   val nFeatures = vocabulary.nFeatures
6
7   @tailrec
8   def train0(XKey: Iterable[String], modelsAcc: Map[String, svm_model]):
9     ↳ Map[String, svm_model] =
10     (XKey.toSeq: @switch) match {
11       case head +: tail =>
12
13         logger.debug(s"\t\tPosTags left: $XKey")
14         logger.debug(s"\t\tSize: ${X(head).size}")
15         logger.debug(s"\t\t# features: $nFeatures")
16         (getClass.getResource(s"${Constants.ModelPath}/svm.$head.model") != null:
17           ↳ @switch) match {
18           case true =>
19             val modelPath =
20               ↳ getClass.getResource(s"${Constants.ModelPath}/svm.$head.model").getPath
21             logger.info(s"Loaded model:
22               ↳ ${modelPath.substring(modelPath.indexOf("svm"))}")
23             logger.debug(s"Loaded model: $modelPath")
24             // Load Models
25             train0(tail, modelsAcc + (head -> svm.svm_load_model(modelPath)))
26           case false =>
27             val svmProblem = new SVMProblem(Y(head).size, Y(head).toArray)
28             // Create each row with its feature values Ex: (Only store the actual
29               ↳ values, ignore zeros)
30             // x -> [ ] -> (2,0.1) (3,0.2) (-1,?)
31             //      [ ] -> (2,0.1) (3,0.3) (4,-1.2) (-1,?)
32             //      .....
33             X(head).zipWithIndex.foreach {
34               case (x, i) =>
35                 val nodeCol = createNode(x)
36                 svmProblem.update(i, nodeCol)
37             }
38             val error = svm.svm_check_parameter(svmProblem.problem,
39               ↳ SVMConfig.param)
40             require(error == null, f"${logger.error(s"Errors in SVM
41               ↳ parameters:\n$error")}")

```

```

35
36     val m = modelsAcc + (head -> trainSVM(svmProblem, SVMConfig.param))
37     svm.svm_save_model(s"src/main/resources/models/svm.$head.model",
38                       ↪ m(head))
39
40     train0(tail, m)
41   }
42   case Nil => modelsAcc
43 }
44 train0(X.keys, Map.empty)

```

El método recibe como parámetro las características calculadas en el paso anterior. En su interior reside una función recursiva que recorre todas las características y va generando tantas SVMs como sea necesario, como se explicó en la [Subsección 4.2.4](#). En el caso de que ya existan los modelos de una anterior ejecución, estos son cargados en lugar de calcularse de nuevo, lo cual ahorra tiempo, ya que entrenar los modelos puede llevar hasta dos días.

Una vez se tienen los modelos entrenados, es el momento de comprobar la calidad de las predicciones, esto se logra con el método TEST.

```

1 private[this] def test(sentences: Vector[LabeledSentence]): Vector[Vector[Node]]

```

El método TEST genera un vector de árboles inferidos por los modelos entrenados. Una vez terminado, para calcular las medidas de evaluación se debe llamar al método GETACCURACY

```

1 def getAccuracy: Accuracy

```

Que devuelve un objeto de tipo ACCURACY, encargado de calcular los tres tipos de medidas de evaluación: DEP ACC, ROOT ACC y COMP. RATE.

5.3.2 NODE

La clase NODE representa el árbol de dependencias, debido a su importancia se muestra el código al completo.

```

1 /**
2  * Node of a tree
3  * Can contain any number of children
4  * Left and Right represents the children created due to left and right
5  * dependencies, respectively.
6  *
7  * Created by Alejandro Alcalde <contacto@elbaultdelprogramador.com>.
8  */
9 case class Node(lex: String,
10                position: Int,
11                posTag: String,
12                var dependency: Int = -1,
13                var left: Vector[Node],
14                var right: Vector[Node]) {
15
16   def insertRight(child: Node): Unit = {
17     child.dependency = position
18     right = right :+ child

```

```

19 }
20
21 def insertLeft(child: Node): Unit = {
22   child.dependency = position
23   left = left :+ child
24 }
25
26 def matchDep(goldSentence: LabeledSentence, depAcc: Map[String, Int],
27   ↪ depAccBase: Map[String, Int])
28 : (Map[String, Int], Map[String, Int]) = {
29   @tailrec
30   def matchDep0(acc: Map[String, Int], acc2: Map[String, Int], n: Node)(queue:
31     ↪ Seq[Node])
32   : (Map[String, Int], Map[String, Int]) = {
33     @inline def condition(node: Node): Boolean =
34       node.dependency != -1 &&
35       ↪ !Constants.punctuationTags.contains(goldSentence.tags(node.position))
36     @inline def condition2(node: Node): Boolean =
37       goldSentence.dep(node.position) == node.dependency
38
39     val w = goldSentence.words(n.position)
40     val newAccs = if (condition(n) && condition2(n)) {
41       (acc + (w -> (acc.getOrElse(w, 0) + 1)), acc2 + (w -> (acc2.getOrElse(w,
42         ↪ 0) + 1)))
43     } else if (condition(n)) {
44       (acc, acc2 + (w -> (acc2.getOrElse(w, 0) + 1)))
45     } else {
46       (acc, acc2)
47     }
48
49     (queue: @switch) match {
50       case head :: tail => matchDep0(newAccs._1, newAccs._2, head)(head.left ++
51         ↪ head.right ++ tail)
52       case Nil => (newAccs._1, newAccs._2)
53     }
54   }
55
56   matchDep0(depAcc, depAccBase, this)(left ++ right)
57 }
58
59 /**
60  * Check if the tree is parsed correctly completely (Ignoring punctuation tags)
61  * against the Gold sentence tags
62  *
63  * @param goldSentence The sentences corretly annotated
64  * @return True if the tree is 100% correctly parsed
65  */
66 def matchAll(goldSentence: LabeledSentence): Boolean = matchNodes(goldSentence)
67   ↪ == goldSentence.words.size
68
69 def matchNodes(goldSentence: LabeledSentence): Int = {
70   @inline def condition(n: Node): Boolean =
71     (goldSentence.dep(n.position) == n.dependency) ||
72     ↪ Constants.punctuationTags.contains(goldSentence.tags(n.position))
73
74   @tailrec
75   def match0(acc: Int, n: Node)(queue: Seq[Node]): Int = {
76     val count = if (condition(n)) acc + 1 else acc
77
78     (queue: @switch) match {
79       case head :: tail => match0(count, head)(head.left ++ head.right ++ tail)
80       case Nil => count
81     }
82   }
83 }

```

```

78
79     match0(0, this)(left ++ right)
80 }
81
82 override def toString: String = s"<LEX: $lex, TAG: $posTag, DEP: $dependency,
    ↪ POS: $position, LEFT: $left, RIGHT: $right>"
83 }

```

Esta clase recibe como parámetros:

Lex: La palabra que almacena.

posTag: La categoría morfosintáctica de la palabra.

dependency: Número representando la dependencia de la palabra.

left: Hijos del árbol a la izquierda de este nodo.

right: Hijos a la derecha de este nodo.

En cuanto a sus métodos:

insertRight/insertLeft: Insertan un nuevo nodo a su izquierda o derecha.

matchDep: Comprueba que las dependencias son correctas para las estimaciones dadas por el modelo.

matchAll: Comprueba si el árbol ha sido calculado correctamente al completo, es decir, ha tenido un acierto del 100 % para esa frase.

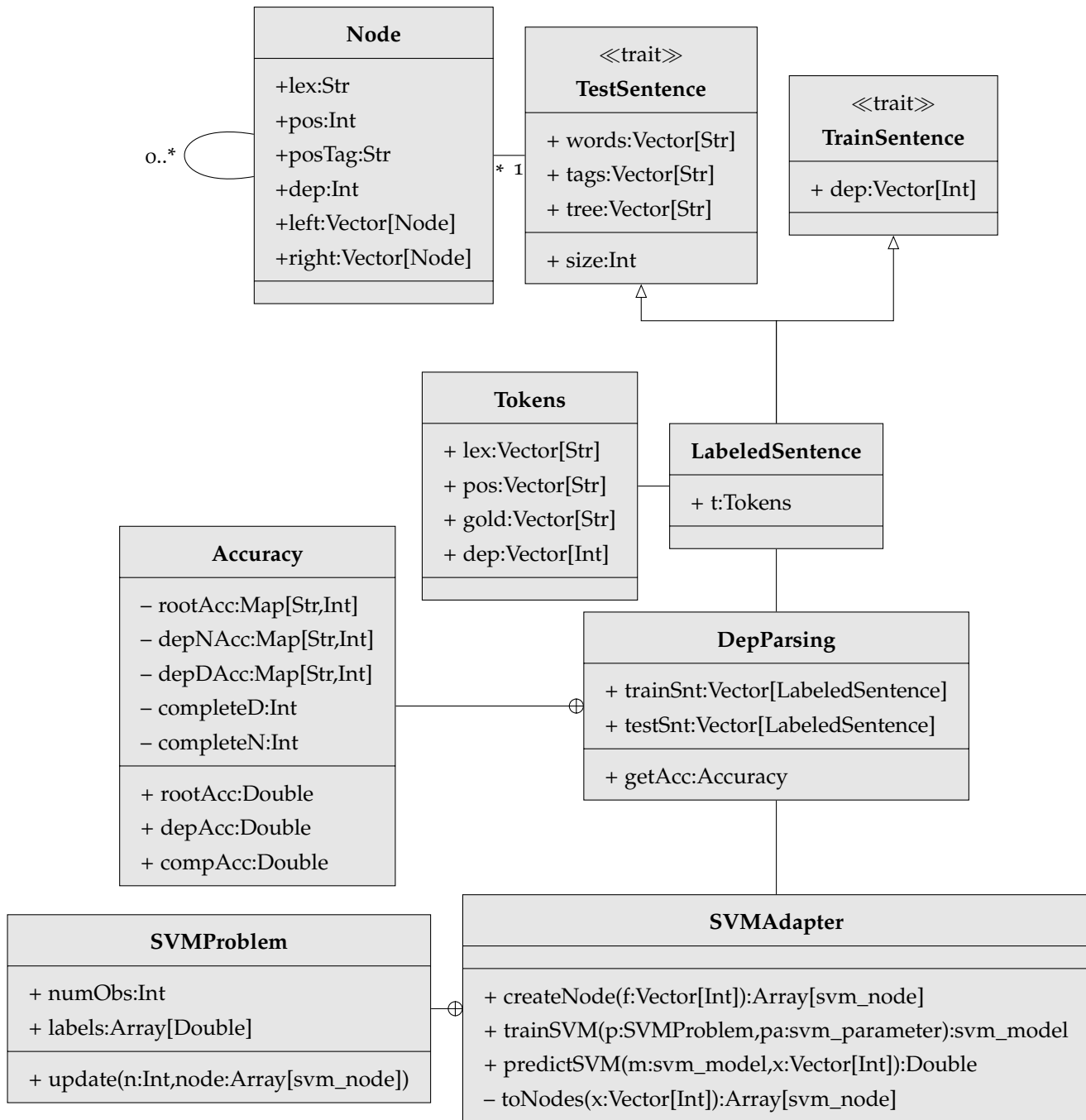


Figura 13: Diagrama de clases completo

CASOS DE PRUEBA ORIENTADOS A APRENDIZAJE AUTOMÁTICO

En este capítulo se verán los casos de prueba implementados para el proyecto. Se ha seguido un estilo de *tests* orientado a problemas de [AA](#). En las próximas secciones se introduce este tipo de práctica.

6.1 TEST-DRIVEN DEVELOPMENT

El *Test-Driven Development* ([TDD](#)) se basa en dos principios muy claros [\[4\]](#):

- No escribir ninguna línea de código nueva a no ser que se tenga un *test* fallido.
- Eliminar duplicidades.

En esencia, [TDD](#) es un proceso en el desarrollo de *software* que permite al desarrollador escribir código que especifica el comportamiento que poseerá el programa antes de que este sea implementado. La ventaja de este estilo de desarrollo reside en que a cada paso que se avanza, se obtiene un *software* completamente funcional, así como el conjunto de especificaciones que lo definen. En el [TDD](#) está inherente en cada momento el desarrollo mediante prueba y error, al igual que en el [AA](#).

El proceso al que nos sometemos al adoptar la filosofía [TDD](#) cambia la forma en la que se piensa al desarrollar código. Además, el *software* diseñado como resultado será mucho más modular, permitiendo tener distintos componentes que se pueden intercambiar en todo el *pipeline*.

Cuando se escribe de antemano la intención del código, antes de implementarlo de verdad, se aplica una presión al diseño del mismo que evita escribir código del llamado “*Por si acaso*”. Con el uso de [TDD](#), primero se piensa en un caso de prueba, se ve que el *software* aún no lo soporta y entonces se corrige. Si no se es capaz de pensar en un caso de prueba, no se añade código.

[TDD](#) opera a varios niveles. Los *tests* pueden escribirse para funciones, métodos, clases, programas, servicios webs, redes neuronales y *pipelines* de [AA](#) al completo. En todo momento, independientemente del nivel, los *tests* se escriben desde la perspectiva del cliente. En este proyecto se ha centrado el tipo de *tests* hacia el [AA](#). En este contexto, los *tests* se escriben para funciones, métodos, clases, implementaciones matemáticas y todos los algoritmos de aprendizaje.

6.1.1 El ciclo de TDD

El ciclo para TDD consiste en escribir trozos pequeños de funciones que intenten hacer algo que aún no está implementado. Normalmente se suele estructurar el *test* en tres partes principales. En la primera se preparan los objetos y datos necesarios. En la segunda se llama al código para el cual se está escribiendo el *test*. Por último, se valida si el resultado del código es el que se esperaba. En la primera fase se escribe el código necesario para hacer pasar el *test* — en este momento no es relevante que el código sea correcto o esté bien diseñado, el único objetivo es hacer pasar el *test*. — Una vez se tiene el *test* correcto, se procede a refactorizar el código. Cabe destacar que en este contexto *refactorizar* significa cambiar la forma en la que se escribió el código, pero bajo ningún concepto cambiar cómo se comporta.

Por tanto, el ciclo del TDD se puede dividir en tres pasos: ROJO, VERDE y REFACTORIZAR.

6.1.1.1 Rojo

Primero se crea un *test* que no funciona. Al más alto nivel en cuanto a AA se refiere, podría ser un punto de partida como conseguir un resultado mejor que una predicción aleatoria. El punto de partida puede ser el que se quiera, el anterior es solo un ejemplo, otros podrían ser: *Predecir siempre lo mismo*.

6.1.1.2 Verde

Una vez se tiene un *test* que está fallando, se procede a arreglarlo, de ahí el nombre de este paso – VERDE. – Si se ha empezado con un *test* a un nivel de abstracción demasiado elevado, se puede dividir el mismo en varios a más bajo nivel y comenzar a arreglarlos. El objetivo aquí es hacer que el *test* pase lo antes posible, es por ello que se permite programar cualquier cosa con tal de obtener un *test* “verde”, es en la siguiente fase donde se procede a refactorizar.

6.1.1.3 Refactorizar

Una vez se tiene el *test* en verde, se procede a refactorizar. Como se mencionó más arriba, hay que tener especial cuidado en esta fase, ya que refactorizar significa cambiar el *software* sin alterar su comportamiento. Si se añade al código una cláusula IF, o cualquier otro tipo de sentencia de control de flujo, ya no se está refactorizando, se está alterando el comportamiento. Un indicativo de que no se está refactorizando correctamente es que *test* escritos en fases anteriores comiencen a fallar — Volverse rojos. — Si esto ocurriese, se debe deshacer lo hecho hasta que los *test* fallidos vuelvan a estar en verde.

6.2 DESARROLLO ORIENTADO A COMPORTAMIENTO

El desarrollo orientado a comportamiento – *Behavior-Driven Development* (BDD) en inglés – consiste en escribir los *test* de forma que expresen el tipo de comportamiento al que afectan. Siguen una estructura muy clara, llamada “GIVEN, WHEN, THEN”. Por ejemplo, un *test* siguiendo esta estructura podría ser “DADO un conjunto de datos vacío, CUANDO se entrena el clasificador, ENTONCES se debería producir una excepción de operación inválida”. Como se aprecia, en la primera parte – GIVEN – se establece un contexto para el *test*, es decir, se preparan los datos de entrada. En la cláusula WHEN, se llama al código cuyo comportamiento se quiere probar y por último, la cláusula THEN comprueba que el resultado del código que se está probando coincide con lo que se esperaba.

El objetivo de este tipo de *test* es que sean lo suficientemente descriptivos para que cualquier persona familiar con el dominio del problema sea capaz de entenderlo y opinar.

6.3 TDD APLICADO AL APRENDIZAJE AUTOMÁTICO

Una vez se han introducido los conceptos TDD y BDD se pasa a describir cómo pueden aplicarse estas filosofías al desarrollo de sistemas de aprendizaje.

En cada algoritmo de AA existe alguna forma de cuantificar la calidad del resultado. En regresión lineal se ajusta el valor R^2 , para problemas de clasificación se utiliza la curva *Receiver Operating Characteristic* (ROC) y el área bajo la misma — Area Under Curve (AUC) — una matriz de confusión u otro tipo de medidas.

Para empezar a desarrollar un sistema, primero se construye un algoritmo muy básico e ignorante. La calidad de este algoritmo será puramente aleatoria. A partir de aquí, se puede comenzar a desarrollar otro algoritmo que se comporte mejor, superando la pura aleatoriedad obtenida anteriormente. Así se comienza un ciclo iterativo en el que se intenta superar a las puntuaciones obtenidas en el paso anterior.

6.4 CASOS DE PRUEBA REALIZADOS

En esta sección se introducen los distintos *tests* que se han realizado al proyecto.

6.4.1 DATAPARSERSPEC

En este *test* se comprueban dos situaciones con cláusulas GIVEN, WHEN, THEN. La primera debe comprobar que cuando no se proporcionan datos para entrenar el modelo, el algoritmo no debe hacer na-

da. La segunda comprueba que cuando sí que se proporcionan datos, se leen correctamente. El código se lista en [9](#).

6.4.2 `DEPENDENCYPARSERCHECKBASELINESPEC`

Este *test* se encarga de establecer un punto de partida para los resultados del modelo. Como se vio en la [Subsubsección 6.1.1.1](#), este punto de partida sirve para comprobar que no se están empeorando las predicciones del modelo conforme vamos desarrollando el sistema. El punto de partida se establece para dos tipos de medidas, la primera, cuyo *test* se puede ojear en el Código [10](#) comprueba la precisión del modelo fijando unas cotas mínimas que deben cumplirse para las medidas de evaluación — pueden consultarse en la [Sección 7.2](#) — La segunda comprueba que el número de características – [Subsección 4.2.3](#) – es siempre el mismo cuando se usa el conjunto de datos para desarrollo, el código de este *test* se muestra en [11](#).

Código 9 *Test que comprueba si la entrada de datos es correcta o no*

```

class DataParserSpec extends Specification
  with GWT
  with StandardRegexStepParsers { def is = s2"""
When no data given, do not launch algorithm  ${testResources.start}
  Given no train data
  Given no test data
  Then should do nothing                      ${testResources.end}
When data given, launch algorithm            ${dataSet.start}
  Given Train file: es_ancora-converted-train1
  Given Test file: es_ancora-converted-test1
  When Launching
  Then should read Correctly                  ${dataSet.end}
"""

val dataSetName = readAs(".*: (.*)$").and((s: String) => s)
val noDataSet = readAs(".*").and((s:String) => "/tmp/aa")

val testResources =
  Scenario("NoData").
    given(noDataSet).
    given(noDataSet).
    when() {case train :: test :: _ =>

      val trainResource =
        if (getClass.getResource(train) == null) ""
        else getClass.getResource(train).getPath
      val testResource =
        if (getClass.getResource(test) == null) ""
        else getClass.getResource(test).getPath

      val r1 = DataParser.readDataSet(trainResource)
      val r2 = DataParser.readDataSet(testResource)

      (r1,r2)
    }.
    andThen() {case _ :: r :: _ => (r._1 must beNone) && (r._2 must
      ↪ beNone) }

val dataSet = Scenario("DATA").
  given(dataSetName).
  given(dataSetName).
  when(aString) {case _ :: t :: tt :: _ =>
    val r1 = DataParser.
      readDataSet(
        getClass.getResource(s"/data/spanish/$t"))
    val r2 = DataParser.
      readDataSet(
        getClass.getResource(s"/data/spanish/$tt"))

    (r1,r2)
  }.
  andThen() {case _ :: r :: _ => (r._1 must beSome) && (r._2 must
    ↪ beSome)}}
}

```

Código 10 Código del *test* DependencyParserCheckBaselineSpec

```

class DependencyParserCheckBaselineSpec extends Specification
  with GWT
  with StandardRegexStepParsers {def is = s2"""
    When training the model, set the following baselines
    ↪  ${featuresBaseline.start}
      Given Train data set: es_ancora-converted-train1
      Given Test data set: es_ancora-converted-test1
      When Genenaring Vocabulary
      Then Dep. Acc should be at least: 70%
      and Root Acc should be at least: 50%
      and Comp. Acc should be at least: 3% ${featuresBaseline.end}
    """}

  val aDataSet = readAs(".*: (.*)$").and((s: String) => s)
  val myD = readAs(".*: (\\d+).*$").and((s: String) => s.toDouble)

  val featuresBaseline =
    Scenario("nFeatures").
      given(aDataSet).
      given(aDataSet).
      when(aString){case _ :: test :: train :: _ =>
        val testSentences =
          ↪  DataParser.readDataSet(getClass.getResource(s"/data/spanish/$test").getPath)
        val trainSentences =
          ↪  DataParser.readDataSet(getClass.getResource(s"/data/spanish/$train").getPath)
        val parser = new DependencyParser(trainSentences.get,
          ↪  testSentences.get)
        parser.getAccuracy
      }.
      andThen(myD){case baseline :: r :: _ => r.dependencyAccuracy*100
        ↪  must be_>(baseline)}.
      andThen(myD){case baseline :: r :: _ => r.rootAccuracy*100 must
        ↪  be_>(baseline)}.
      andThen(myD){case baseline :: r :: _ => r.completeAccuracy*100
        ↪  must be_>(baseline)}

}

```

Código 11 Código del *test* `DependencyParserCheckNFeaturesSpec`

```

class DependencyParserCheckNFeaturesSpec extends Specification
  with GWT
  with StandardRegexStepParsers {def is = s2"""
    When training the model, set the following baselines
    ↪  ${nfeaturesCheck.start}
        Given Train data set: es_ancora-converted-train1
        Given Test data set: es_ancora-converted-test1
        When Genenaring Vocabulary
        Then the number of features must be: 46468
    ↪  ${nfeaturesCheck.end}
    """

    val aDataSet = readAs(".*: (.*)$").and((s: String) => s)

    val nfeaturesCheck =
      Scenario("nFeatures").
        given(aDataSet).
        given(aDataSet).
        when(aString) { case _ :: test :: train :: _ =>
          val testSentences = DataParser.readDataSet(
            getClass.getResource(s"/data/spanish/$test").getPath)
          val trainSentences = DataParser.readDataSet(
            getClass.getResource(s"/data/spanish/$train").getPath)
          val parser = new DependencyParser(trainSentences.get,
            ↪ testSentences.get)
          parser.nFeatures
        }.
        andThen(anInt) { case baseline :: r :: _ => baseline must_== r }
  }

```

Parte IV

CONCLUSIONES Y VÍAS FUTURAS

Esta última parte de la memoria se dedica a presentar los resultados obtenidos con el parseador introducido en [Capítulo 4](#), así como comparar los resultados con la implementación original. Por último, se comentarán algunas ideas para trabajos futuros.

EVALUACIÓN, COMPARACIÓN Y DISCUSIÓN DE RESULTADOS

7.1 CONJUNTOS DE DATOS

El conjunto de datos usado ha sido *Spanish Universal Dependency* [20], para poder ejecutar el algoritmo con estos datos, ha sido necesario hacer una conversión de los mismos, ya que para etiquetar las frases de *test* se ha usado el *POS tagger* de Stanford, y este no etiqueta los datos de acuerdo al estándar¹. El *tagger* de Stanford utiliza las etiquetas de ANCORa 3.0². Para dicha conversión se usó un *script* en PYTHON proporcionado por Jain [12].

En concreto, el conjunto de datos usado se llama UD_SPANISH-ANCORa, puede descargarse a través de <https://lindat.mff.cuni.cz/repository/xmlui/handle/11234/1-1827>. Citando de su documentación, ANCORa consiste en:

ANCORa es un corpus del catalán (ANCORa-CA) y del español (ANCORa-ES) con diferentes niveles de anotación. El corpus de cada lengua contiene 500.000 palabras y están constituidos fundamentalmente por textos periodísticos

Como se ha mencionado en el párrafo anterior, este *dataset* usa el formato CoNLL-U³, formato que ha sido adaptado usando el SCRIPT mencionado anteriormente.

Como ejemplo, se proporciona el formato original de una frase en la Figura 14 frente a la conversión realizada en la Figura 15.

Para las pruebas realizadas, el tamaño del conjunto de datos de entrenamiento es de 14305 frases, los datos de *test* contienen 1721 sentencias.

7.2 MEDIDAS DE EVALUACIÓN

En orden de evaluar los resultados del algoritmo, Yamada y Matsmoto proponen tres tipos de medidas. Precisión de Dependencias, Precisión en la Raíz y clasificación Completa — *Dependency Accuracy*

¹ <http://universaldependencies.org/docs/u/pos/index.html>

² <http://cllc.ub.edu/corpus/en>

³ <http://universaldependencies.org/format.html>

1	Tambores	tambor	NOUN	NOUN	Gender=Masc Number=Plur	0	root	-	-
2	cercanos	cercano	ADJ	ADJ	Gender=Masc Number=Plur	1	amod	-	-
3	,	,	PUNCT	PUNCT	PunctType=Comm	4	punct	-	-
4	en	en	ADP	ADP	AdpType=Prep	1	advmod	-	-
5	todo	todo	PRON	PRON	Gender=Masc Number=Sing PronType=Tot	4	mwe	-	-
6	caso	caso	NOUN	NOUN	-	4	mwe	-	-
7	:	:	PUNCT	PUNCT	PunctType=Colo	10	punct	-	-
8	todo	todo	DET	DET	Gender=Masc Number=Sing PronType=Ind	10	det	-	-
9	un	uno	DET	DET	Gender=Masc Number=Sing PronType=Ind	8	det	-	-
10	muestuario	muestuario	NOUN	NOUN	Gender=Masc Number=Sing	1	appos	-	-
11	de	de	ADP	ADP	AdpType=Prep	13	case	-	-
12	esa	ese	DET	DET	Gender=Fem Number=Sing PronType=Dem	13	det	-	-
13	percusión	percusión	NOUN	NOUN	Gender=Fem Number=Sing	10	nmod	-	-
14	urbana	urbano	ADJ	ADJ	Gender=Fem Number=Sing	13	amod	-	-
15	de	de	ADP	ADP	AdpType=Prep	17	case	-	-
16	nuestro	nuestro	DET	DET	Gender=Masc Number=Sing Number[psor]=Plur Person=1 Poss=Yes PronType=Prs	-	-	-	-
17	↪	17	det	-	-	-	-	-	-
18	tiempo	tiempo	NOUN	NOUN	Gender=Masc Number=Sing	13	nmod	-	-
19	.	.	PUNCT	PUNCT	PunctType=Peri	1	punct	-	-

Figura 14: Frase en formato CoNLL-U.

1	Tambores	NOUN	NOUN	-1
2	cercanos	ADJ	ADJ	0
3	,	PUNCT	PUNCT	3
4	en	ADP	ADP	0
5	todo	DET	PRON	3
6	caso	NOUN	NOUN	3
7	:	PUNCT	PUNCT	9
8	todo	DET	DET	9
9	un	DET	DET	7
10	muestuario	NOUN	NOUN	0
11	de	ADP	ADP	12
12	esa	DET	DET	12
13	percusión	NOUN	NOUN	9
14	urbana	ADJ	ADJ	12
15	de	ADP	ADP	16
16	nuestro	DET	DET	16
17	tiempo	NOUN	NOUN	12
18	.	PUNCT	PUNCT	0

Figura 15: Frase en formato convertido.

(*Dep. Acc.*), *Root Accuracy* (*Root Acc.*) y *Complete Rate* (*Comp. Rate*), respectivamente. — Dichas mediciones se definen como

$$\begin{aligned}
 \text{Dep. Acc} &= \frac{\text{Número correcto de padres}}{\text{Número total de padres}} \\
 \text{Root Acc} &= \frac{\text{Número de nodos raíz correctos}}{\text{Número total de frases}} \\
 \text{Comp. Rate} &= \frac{\text{Número de frases parseadas por completo}}{\text{Número total de frases}}
 \end{aligned}$$

7.3 COMPARACIÓN DE RESULTADOS

Tras realizar varias pruebas para ajustar parámetros, finalmente se fijó el grado del polinomio a 2, como muestra el Código 7. En cuanto a la longitud del contexto, introducido en Subsección 4.2.3 el mejor resultado se obtiene cuando se fija en (2,4), esto es, se usa un contexto a la izquierda de dos nodos, y un contexto a la derecha de cuatro nodos. La Tabla 4 muestra una comparación de los resultados obte-

nidos con el parseador implementado frente a los resultados de Jain. Se menciona aquí a Jain [12] porque implementó el mismo algoritmo aquí expuesto, pero usando conjuntos de datos para el Castellano. De hecho, este trabajo usa el mismo conjunto de datos que Jain, ANCORa, introducido en la Sección 7.1.

KERNEL: $(x' \cdot x'' + 1)^2$, CONTEXTO: (2, 4)	TFG	JAIN
<i>Dep. Acc.</i>	76 %	75 %
<i>Root Acc.</i>	67 %	70 %
<i>Comp. Rate</i>	15 %	11 %

Tabla 4: Comparación de resultados

Como se puede apreciar, los resultados son bastante similares, llegando a mejorar el *Comp. Rate*.

Es de esperar que los resultados sean similares, ya que el algoritmo es el mismo. Las variaciones pueden deberse al método `EXTRACTTEST-FEATURES` — Subsección 5.3.1 —, ya que generan salidas distintas. Es posible que esto se deba al comportamiento de los diccionarios, ya que como bien es conocido, no tienen un orden definido. Por tanto, al acceder a los elementos del mismo, que contienen las características se producen pequeñas variaciones que pueden estar influyendo en el cálculo de las características extraídas.

Otro motivo por el que pueden estar obteniéndose resultados ligeramente distintos es el tipo de implementación para la SVM. Jain hace uso del módulo para PYTHON `SKLEARN`, este proyecto sin embargo, utiliza la implementación original de SVM desarrollado por Chang y Lin, `LIBSVM` [6].

VÍAS FUTURAS

8.1 TRABAJO FUTURO

Como trabajo futuro se podrían implementar distintos tipos de algoritmos y establecer una comparación entre ellos.

Otra posible mejora puede ser aprovechar aún más las características de SCALA, ya que al momento de escribir el programa, el desarrollador se estaba familiarizando con el lenguaje. Hay mucho margen para realizar mejoras en el código. Por ejemplo, se podría hacer todo el código funcional e inmutable.

Ya que este algoritmo se ha encargado de implementar un único proceso del *pipeline* – [Sección 1.4](#) y [Sección 1.5](#) –, se podrían proponer más implementaciones del mismo, hasta llegar a realizar un *software* que implemente un *pipeline* completo para el castellano.

Por último, en orden de mejorar el tiempo de entrenamiento, sería posible trasladarse a tecnologías de *Big Data* e implementar el proceso usando *frameworks* como SPARK.

Parte V

APÉNDICE

BIBLIOGRAFÍA

- [1] Yaser S. Abu-Mostafa, Malik Magdon-Ismael y Hsuan-Tien Lin. *Learning From Data, a short course*. AML, 2012.
- [2] Michael Gamon Anthony Aue. «Customizing Sentiment Classifiers to New Domains: a Case Study». En: *Submitted to RANLP-05, the International Conference on Recent Advances in Natural Language Processing*. Borovets, BG, 2005.
- [3] Miguel Ballesteros, Bernd Bohnet, Simon Mille y Leo Wanner. «Data-driven deep-syntactic dependency parsing». En: *Natural Language Engineering* 22.6 (nov. de 2016), págs. 939-974.
- [4] Justin Bozonier. *Test-Driven Machine Learning*. Packt, 2015.
- [5] Angel X. Chang y Christopher D. Manning. «SUTIME: A Library for Recognizing and Normalizing Time Expressions». En: *In LREC*. 2012.
- [6] Chih-Chung Chang y Chih-Jen Lin. «LIBSVM: A library for support vector machines». En: *ACM Transactions on Intelligent Systems and Technology* 2 (3 2011). Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>, 27:1-27:27.
- [7] Ronen Feldman. «Techniques and applications for sentiment analysis». En: *Communications of the ACM* 56.4 (2013), pág. 82.
- [8] Jenny Rose Finkel, Trond Grenager y Christopher Manning. «Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling». En: *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*. ACL '05. Ann Arbor, Michigan: Association for Computational Linguistics, 2005, págs. 363-370.
- [9] Geoffrey Hinton y col. «Deep Neural Networks for Acoustic Modeling in Speech Recognition». En: *IEEE Signal Processing Magazine* 29 (2012), págs. 82-97.
- [10] J. Hirschberg y C. D. Manning. «Advances in natural language processing». En: *Science* 349.6245 (2015), págs. 261-266.
- [11] Minqing Hu y Bing Liu. «Mining and summarizing customer reviews». En: *Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '04*. Association for Computing Machinery (ACM), 2004.
- [12] Rohit Jain. *Re-Implementing Statistical Dependency Analysis With Support Vector Machines*. Inf. téc. Cornell Tech, 2016.

- [13] Dan Klein y Christopher D. Manning. «Fast Exact Inference with a Factored Model for Natural Language Parsing». En: *Proceedings of the 15th International Conference on Neural Information Processing Systems*. NIPS'02. Cambridge, MA, USA: MIT Press, 2002, págs. 3-10.
- [14] Taku Kudo, Kaoru Yamamoto y Yuji Matsumoto. «Applying conditional random fields to Japanese morphological analysis». En: *In Proc. of EMNLP*. 2004, págs. 230-237.
- [15] Heeyoung Lee, Angel Chang, Yves Peirsman, Nathanael Chambers, Mihai Surdeanu y Dan Jurafsky. «Deterministic Coreference Resolution Based on Entity-centric, Precision-ranked Rules». En: *Comput. Linguist.* 39.4 (dic. de 2013), págs. 885-916. ISSN: 0891-2017.
- [16] B. Liu. En: *Handbook of Natural Language Processing Chapter Sentiment Analysis and Subjectivity* (2010), págs. 627-666.
- [17] Bing Liu. «Sentiment Analysis and Opinion Mining». En: *Synthesis Lectures on Human Language Technologies* 5.1 (2012), págs. 1-167.
- [18] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard y David McClosky. «The Stanford CoreNLP Natural Language Processing Toolkit». En: *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Association for Computational Linguistics (ACL), 2014.
- [19] Marie-Catherine de Marneffe y Christopher D. Manning. «The Stanford Typed Dependencies Representation». En: *Coling 2008: Proceedings of the Workshop on Cross-Framework and Cross-Domain Parser Evaluation*. CrossParser '08. Manchester, United Kingdom: Association for Computational Linguistics, 2008, págs. 1-8. ISBN: 978-1-905593-50-7.
- [20] Joakim Nivre y col. *Universal Dependencies 1.4*. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University in Prague. 2016. URL: <http://hdl.handle.net/11234/1-1827>.
- [21] Bo Pang, Lillian Lee y Shivakumar Vaithyanathan. «Thumbs Up?: Sentiment Classification Using Machine Learning Techniques». En: *Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing - Volume 10*. EMNLP '02. Stroudsburg, PA, USA: Association for Computational Linguistics, 2002, págs. 79-86.
- [22] Fuchun Peng, Fangfang Feng y Andrew McCallum. «Chinese segmentation and new word detection using conditional random fields». En: *Proceedings of the 20th international conference on Computational Linguistics - COLING '04*. Association for Computational Linguistics (ACL), 2004.

- [23] Mike Pound y Sean Riley. *Inside a Neural Network – Computerp-hile*. Youtube. 2016. URL: https://www.youtube.com/watch?v=BFdMrD0x_CM.
- [24] James Pustejovsky y Amber Stubbs. *Natural Language Annotation for Machine Learning*. O'Reilly, 2012.
- [25] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng y Christopher Potts. «Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank». En: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Seattle, Washington, USA: Association for Computational Linguistics, 2013, págs. 1631-1642.
- [26] Tun Thura Thet, J.-C. Na y C. S. G. Khoo. «Aspect-based sentiment analysis of movie reviews on discussion boards». En: *Journal of Information Science* 36.6 (2010), págs. 823-848.
- [27] Kristina Toutanova, Dan Klein, Christopher D. Manning y Yo-ran Singer. «Feature-rich Part-of-speech Tagging with a Cyclic Dependency Network». En: *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*. NAACL '03. Edmonton, Canada: Association for Computational Linguistics, 2003, págs. 173-180.
- [28] Huihsin Tseng, Pichuan Chang, Galen Andrew, Daniel Jurafsky y Christopher Manning. «A Conditional Random Field Word Segmenter for Sighan Bakeoff 2005». En: *SIGHAN Workshop on Chinese Language Processing*. Association for Computational Linguistics, 2005.
- [29] Peter D. Turney. «Mining the Web for Synonyms: PMI-IR versus LSA on TOEFL». En: *Machine Learning: ECML 2001: 12th European Conference on Machine Learning Freiburg, Germany, September 5-7, 2001 Proceedings*. Ed. por Luc De Raedt y Peter Flach. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, págs. 491-502. ISBN: 978-3-540-44795-5.
- [30] Peter D. Turney. «Thumbs Up or Thumbs Down?: Semantic Orientation Applied to Unsupervised Classification of Reviews». En: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. ACL '02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, págs. 417-424.
- [31] David Vilares, Miguel A. Alonso y Carlos Gómez-Rodríguez. «A syntactic approach for opinion mining on Spanish reviews». En: *Natural Language Engineering* 21.01 (2013), págs. 139-163.
- [32] Dean Wampler y Alex Payne. *Programming Scala*. O'Reilly, 2015.

- [33] Hao Wang y Martin Ester. «A Sentiment-aligned Topic Model for Product Aspect Rating Prediction». En: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics (ACL), 2014.
- [34] Linlin Wang, Kang Liu, Zhu Cao, Jun Zhao y Gerard de Melo. «Sentiment-Aspect Extraction based on Restricted Boltzmann Machines». En: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, 2015, págs. 616-625.
- [35] Hiroyasu Yamada y Yuji Matsumoto. «Statistical dependency analysis with support vector machines». En: *Proceedings of IWPT*. Vol. 3. 2003, págs. 195-206.
- [36] Hong Yu y Vasileios Hatzivassiloglou. «Towards Answering Opinion Questions: Separating Facts from Opinions and Identifying the Polarity of Opinion Sentences». En: *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing*. EMNLP '03. Stroudsburg, PA, USA: Association for Computational Linguistics, 2003, págs. 129-136.
- [37] Lina Zhou y Pimwadee Chaovalit. «Ontology-supported polarity mining». En: *Journal of the American Society for Information Science and Technology* 59.1 (2007), págs. 98-110.

DECLARACIÓN

Yo, ALEJANDRO ALCALDE BARROS, alumno del GRADO EN INGENIERÍA INFORMÁTICA de la ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIONES de la UNIVERSIDAD DE GRANADA, declaro explícitamente que el trabajo presentado es original, entendiéndose en el sentido de que no se ha utilizado ninguna fuente sin citarla debidamente.

Granada, 8 de diciembre de 2016

Alejandro Alcalde Barros

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both \LaTeX and \LyX :

<https://bitbucket.org/amiede/classicthesis/>

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

<http://postcards.miede.de/>

Final Version as of 8 de diciembre de 2016 (`classicthesis` version 4.2).

Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL*) are used. The "typewriter" text is typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera". (Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.)