TRABAJO FIN DE GRADO

INGENIERÍA INFORMÁTICA

Algoritmos para Procesamiento de Lenguaje Natural en Español: Tokenizador, POS tagger y lematizador

Autor

Mª Cristina Heredia Gómez

Director

Salvador García López

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada,12 de Diciembre de 2016

Director: Salvador García López Autor: Mª Cristina Heredia Gómez

Algoritmos para Procesamiento de Lenguaje Natural en Español: Tokenizador, POS tagger y lematizador

Resumen

En este trabajo se presenta una implementación en Scala de tres algoritmos que desarrollan tres fases iniciales y fundamentales en tareas de procesamiento del lenguaje natural. En primer lugar se presenta un algoritmo de tokenización que en tres fases extraerá los tokens presentes en un texto dado y las frases del mismo. En segundo lugar, se presenta un POS tagger que etiquetará todas las palabras extraídas en el algoritmo anterior atendiendo a su categoría morfosintáctica, usando el formato de etiquetas Parole por defecto. Para ello aplicará reglas léxicas, morfológicas y contextuales, estas últimas obtenidas mediante un algoritmo de Brill. En tercer y último lugar se presenta un lematizador que pasará cada palabra a su forma base, haciendo uso de las etiquetas presentes en las mismas como consecuencia del proceso anterior para desencadenar unas u otras reglas de lematización.

Las tres herramientas se desarrollan para el español, obteniendo resultados similares y en casos, ligeramete superiores a la implementación original propuesta en la versión para español de Smedt & Daelemans[38] (Pattern.es). Por último, se integran las tres herramientas en un parseador, por lo que es posible probar cualquiera o todos los algoritmos desde línea de comandos.

Palabras Clave: Procesamiento del lenguaje natural, Scala, tokenizador, pos etiquetado, lematizador

Director: Salvador García López Autor: Mª Cristina Heredia Gómez

Algorithms for natural language processing in Spanish: Tokenizer, POS tagger and lemmatizer

Abstract

In this work is exposed a Scala implementation of three initial and essential algorithms in natural language processing tasks. On the first hand, a tokenization algorithm, that iterates whithin three steps extracting tokens whithin a given text and its sentences is exposed. On the second hand, is exposed a POS Tagger algorithm that tags all the extracted words in the previus step by their morphosyntactic information, using Parole tagset by default. In order to do this, tagger applies lexical, morphological and contextual rules, where contextual ones are acquired by a Brill algorithm. In last place, a lemmatizer that gets the basic form of given words is exposed. In order to achieve that, it takes conscience about given word's tags in previous tagging proccess to apply one lemmatization rules set or another.

The three algorithm are developed for Spanish language, achieving similar results and sometimes quite superior results to the original implementation proposed in Spanish version of Smedt & Daelemans[38] (Pattern.es). Last but not at least, the three algorithms are integrated whithin a parser, so it could be possible to taste one or all the implemented algorithms by command line.

Keywords: Natural language processing, Scala, tokenizer, pos tagger, lematizer

Contents

Lista de figuras

Agradecimientos

Gracias a mis padres, especialmente a Manolo por el apoyo mostrado durante este trabajo y las dosis de Serenia. Gracias también a mi tutor, Salva, por el apoyo y trabajar los domingos, y a Juanjo, por ser tan comprensivo y vivaz. Gracias a Alex, por aguantarme cuando ni yo lo hacía y ser mi pilar.

1

Introducción y motivación

1.1 Introducción al PLN

El proyecto desarrollado se engloba en el campo del Procesamiento del lenguaje natural(PLN). El lenguaje natural es cualquier lenguaje usado por los humanos para comunicarse (Alemán, Inglés, Español, Hindi...). Dado que estos lenguajes se transmiten entre generaciones y van experimentando evoluciones, resulta difícil obtener reglas que los describan. PLN es, por tanto, el área de estudio y aplicación que engloba cualquier tipo de manipulación computacional del lenguaje natural.

Es decir, PLN abarca desde aplicaciones simples, como contar el número de ocurrencias de las palabras en un texto para comparar diferentes estilos de escritura, a aplicaciones más complejas, como comprender expresiones humanas completas para poder dar respuestas útiles a preguntas. Bird et al.[4] Como por ejemplo, el asistente Siri de iPhone.

La lingüística computacional o PLN comenzó en 1980, sin embargo en los últimos 20 años ha crecido enormemente, despertando un gran interés en el ámbito de la investigación científica pero también en el ámbito práctico, ya que cada vez son más los productos, especialmente los tecnológicos,

que incorporan algún tipo de aplicación basada en NLP. Por ejemplo, traductores como el traductor de Skype, o asistentes de voz inteligentes (Cortana de Microsoft, Google Now de Google o el ya mencionado Siri de Apple).

Este crecimiento en el campo del procesamiento del lenguaje natural se debe principalmente a que en los últimos años, con el uso de redes sociales como Facebook, SnapChat, Twitter, Google plus, Linked in... y de sitios web comerciales como Amazon o Booking, los usuarios han generado una gran cantidad de contenido mayoritariamente subjetivo, el cual se puede aplicar en muchos ámbitos como márketing, política, gestión de crisis, soporte, atención al cliente, etc. También han influido en su crecimiento el aumento de capacidad de procesamiento y cómputo que ha habido en los últimos años y el desarrollo de técnicas de machine learning más complejas y potentes.

En la actualidad, según lo descrito en Hirschberg & Manning[16] éstas son algunas de las principales áreas en PLN:

Traducción automática

La traducción automática es el área del PLN que tiene como objetivo el empleo de sofware para ayudar a traducir de un lenguaje natural a otro, ya sea en texto o hablado. Ésto supone una gran dificultad, ya que para que una traducción sea correcta, no basta con traducir palabra a palabra, sino que hay que tener en cuenta el sentido de la palabra y el contexto de ésta, pues hay casos en los que la misma palabra significa varias cosas dependiendo del contexto. Por ejemplo, en las frases *"compra una lata de refresco"* y *"deja ya de dar la lata"* aparece la palabra *lata* desempeñando una función distinta:

En la primera frase, *lata* es un nombre, por lo que se entendería como un envase hecho de hojalata, mientras que en la segunda frase aparece como una locución verbal, por lo que se entendería como "molestar" o "importunar".

El campo de la traducción automática se empezó a estudiar a finales de 1950s, sin embargo inicialmente no tuvo mucho éxito debido a que los traductores construidos eran sistemas basados en gramáticas escritas a mano. Fue a partir de 1990 y gracias a que los científicos de IBM consiguieron una cantidad suficientemente grande de frases de traducciones entre dos lenguajes, cuando construyeron un modelo probabilístico de traducción automática.

A partir de entonces se siguió investigando y se descubrieron los traductores máquina basados en frases, que en lugar de ir traduciendo palabra a palabra, detectaban los pequeños subgrupos de palabras que solían ir juntas y que tenían una traducción especial. Esto se utilizó para desarrollar el traductor de Google.

Actualmente, el estado del arte en este campo está en traductores máquinas que usan deep learning, entrenando un modelo de varios niveles para optimizar un objetivo (la calidad de la traducción), donde luego el modelo pueda aprender por sí mismo más niveles que le sean útiles para desarrollar la tarea. Esto ha sido estudiado especialmente para redes neuronales, habiendo conseguido en varios casos obtener los mejores resultados hasta el momento, empleando redes neuronales distribuidas. Como por ejemplo, en Luong et al.[23].

Sistemas de reconocimiento del habla

Esta área, muy conocida desde 1980s, estudia como permitir y mejorar la comunicación entre humanos y máquinas. Aunque siempre se ha pensado, por ejemplo, en aplicaciones como robots que ayudan en casa o a personas con movilidad reducida,no muchos años atrás se expandió al ámbito de los smartphones (mencionábamos en la introducción a los asistentes de voz para móvil más conocidos).

El reconocimiento del habla necesita principalmente de :

- Una herramienta de reconocimiento automático del habla (RAH) para identificar que está diciendo el humano.
- Una herramienta de manejo de diálogo (MD) para identificar lo que quiere el humano.
- Acciones para realizar la actividad solicitada.
- Una herramienta de síntesis texto a voz para que la máquina pueda comunicar al humano el resultado de forma hablada.

Sin embargo, aún se está investigando como hacer estas herramientas más precisas. Añadiéndole a lo anterior las dificultades propias de reconocer lenguaje humano hablado: pausas, coletillas, coordinación, toma de turnos... desemboca en que los sistemas de reconocimiento de habla aún no

han tenido gran éxito interactuando en dominios abiertos, donde los usuarios pueden hablar de cualquier cosa, aunque en dominios cerrados donde conocían el tema han mostrado resultados mejores.

En los últimos años se ha aplicado deep learning en estos sistemas, mapeando señales de sonido a secuencias de palabras y sonidos del lenguaje humano Hinton et al.[15], aunque actualmente el enfoque más usado es el proceso de decisión de Markov, que hace identificación del diálogo (pregunta, sentencia, acuerdo..) mediante una probabilidad de distribución sobre todos los posibles estados del sistema, que va actualizando según se desarrolla el diálogo. Young et al.[47].

Lectura Automática

La lectura automática es el área que tiene como objetivo que las máquinas puedan integrar o resumir información a los humanos, mediante la lectura y comprensión de las grandes cantidades de texto disponibles.

Esta idea atrae especialmente a los científicos, ya que es complicado llevar el ritmo de todas las publicaciones que se hacen, aunque sólo sea en su campo, por lo que sería de gran utilidad que un sistema pudiera resumir e identificar los datos más relevantes de las publicaciones. El objetivo inicial de estos sistemas es la extracción de relaciones, es decir, ser capaz de extraer relaciones entre dos entidades, como por ejemplo ”A es hermano de B”, lo cual ya se ha realizado con éxito en dominios específicos. Aunque hay técnicas que escriben los patrones de las relaciones a mano (por ejemplo: <PERSONA>, el hermano de <PERSONA>), se ha demostado que aplicando Machine learning se obtienen mejores resultados, ya que se pueden obtener relaciones basadas en características extraidas de secuencias de palabras y secuencias gramaticales de una frase. Culotta & Sorensen[8].

Los sistemas más recientes han usado inferencia probabilística sofisticada para distinguir qué claúsulas textuales se asocian a qué factores de la base del conocimiento, por ejemplo, Niu et al.[28] y apuestan por técnicas de extracción de hechos más simples pero más escalables que no requieren etiquetado manual de los datos, o las extraen usando NLP. Etzioni et al.[11].

Minería de datos en medios sociales

La minería de datos es el campo que tiene como objetivo descubrir patrones en grandes volúmenes de datos. Hoy en día, la gran cantidad de datos disponibles a través de redes sociales (Facebook, Twitter, Instagram, Youtube..), blogs o foros se puede descargar usando técnicas de web scrapping y se usa, aplicando técnicas de machine learning e inteligencia artificial, para aprender a detectar información demográfica a partir del lenguaje (como sexo o edad), hacer un seguimiento de las tendencias más populares u opiniones más populares sobre política o sobre productos, e incluso, como hizo Google (`www.google.org/flutrends/`) para ver como se difunde la gripe a través de los tweets de los usuarios y sus búsquedas en internet Elhadad et al.[10].

A pesar de que este campo tiene innumerables aplicaciones, muchas de las cuales podrían ser de gran interés (como por ejemplo, detectar grupos que hacen bullying a otros o fomentan el odio), están aumentando los poblemas de privacidad y se está limitando el acceso a esos datos. Por ejemplo, Twitter ya ha limitado el periodo de tiempo del que se pueden descargar tweets. Instagram también a modificado su API con este propósito.

Otra dificultad con la que cuenta este campo, es la validación. Muchas veces no hay forma de comprobar que la información presente en internet es cierta, por ejemplo las reseñas sobre hoteles, productos o restaurantes. En la actualidad, Facebook está ideando un modelo para detectar noticias falsas en su red social. Aunque se ha probado a agregar información de distintas fuentes para intentar validar la información, de momento no ha tenido mucho éxito.

Análisis de sentimientos

Este campo (también conocido como minería de opiniones) analiza las opiniones, sentimientos, valoraciones, actitudes y emociones de la gente frente a entidades como productos, servicios, organizaciones, individuos, eventos, temas, cuestiones...

Liu[22] emplea el término *opinión* para referirse al concepto de sentimento, evaluación, valoración o actitud e información asociada (objetivo de la opinión o persona que da la opinión) en su totalidad, y el término *sentimiento* para referirse al sentido positivo o negativo subyacente en una opinión. Por ejemplo *"Apple lo está haciendo muy bien en esta economía pobre"* es una opinión que contiene

dos sentimientos, uno positivo con Apple como objetivo y otro negativo sobre la economía actual.

Los estudios sobre este campo comenzaron en el año 2000, principalmente debido a que para entonces se empezó a recoger texto subjetivo en formato digital. Actualmente hay muchos campos relacionados con este cuyas tareas difieren ligeramente, por ejemplo análisis de opiniones, análisis de subjetividad, minería de sentimientos... aunque gran parte del trabajo se concentra en el análisis de sentimientos.

Los enfoques más simples tratan de identificar si lo expresado en el texto (por ejemplo, en un tweet) tiene una orientación positiva o negativa usando dicccionarios de sentimientos como Whissell[44]. Otros enfoques más complejos tratan de identificar la polaridad del sentimiento así como el objeto de éste.Wiebe et al.[45]. También se han realizado trabajos recientes tratando de indentificar algunas emociones en particular, como las de Ekman (furia, aversión, miedo, felicidad,tristeza y sorpresa) y se ha investigado sobre reconocer esas emociones clásicas usando características como la edad, la personalidad , el género las condiciones mentales o médicas del usuario. Hirschberg & Manning[16].

Las aplicaciones de este campo son innumerables y abarcan desde identificar valoraciones en productos Wang[42] a predecir los precios del mercado o evaluar el estado mental de una comunidad. Bollen et al.[5].

1.2 Historia del PLN

De acuerdo a lo descrito en Hirschberg & Manning[16], el PLN comienza en 1980s como intersección entre la inteligencia artificial y la lingüística. Durante las primeras décadas, los investigadores escribían a mano las reglas y el vocabulario del lenguaje humano. Sin embargo, no se obtuvo éxito, debido a la variabilidad y dificultad del lenguaje humano. Por ejemplo, el traductor palabra a palabra de ruso a inglés que no tenía en cuenta el contexto, el léxico o la morfología y que tradujo la frase bíblica : *"El espíritu está dispuesto, pero la carne es débil"* como *"El vodka es agradable, pero la carne es estropeado"* según Nadkarni et al.[27].

Es a partir de 1990 cuando el PLN sufre una transformación cuando los investigadores comienzan a tener la posibilidad de obtener grandes cantidades de datos del lenguaje en formado digital y construyen modelos sobre estos. Surge así el PLN estadístico o PLN basado en corpus, lo que supuso un éxito en el uso del "big data", aunque ese término se introduciría más adelante. Con lo anterior

surgen métodos que usan el Part-Of-Speech (POS) de las palabras, es decir, su categoría morfosintáctica(si son sustantivo, un adjetivo, un verbo, una preposición...) adquiriendo notables resultados cuando se entrena con un conjunto de datos suficientemente grande.

Actualmente, muchos clasificadores de texto y sentimientos se basan únicamente en los diferentes conjuntos de palabras que presenta el texto (bolsas de palabras) sin tener en cuenta estructuras a nivel de frase, de documento o de significado. Sin embargo, los mejores enfoques actuales usan técnicas sofisticadas de machine learning y un buen entendimiento de la estructura lingüística subyacente, identificando información sintáctica, semántica y de contexto. Algunos de los sofwares más conocidos acutalmente son:

- Stanford CoreNLP Manning et al.[24] es una suite de herramientas de PLN basada en machine learning, que pueden ser incorporadas en aplicaciones con necesidades de procesamiento de texto. Desarrollada en Java y con algunas herramientas disponibles para los lenguajes más hablados, también incluye métodos para entrenar modelos sobre corpus de datos.
- Python NLTK Bird[3] es una suite de PLN de las más completas. Programada en Python, soporta un gran número de librerías para texto que incluyen las herramientas más comunes. Su éxito se debe en parte a que está bien documentada, incluye tutoriales y a su comunidad activa de usuarios.

1.3 Aproximaciones más comunes en PLN

Debido a la gran aplicabilidad que tiene el análisis de sentimientos, se ha despertado un gran interés por este área del PLN, proponiendo en los últimos años numerosas aproximaciones que emplean varias técnicas desde diferentes áreas de la informática para resolver el problema.

En la actualidad, las aproximaciones más comunes son dos Ribeiro et al.[34]:

1.3.1 Aplicación de métodos de machine learning supervisados

Estos enfoques aplican métodos de clasificación supervisada, por lo que tienen como desventaja que requieren de datos (corpus) etiquetados (con información morfosintáctica, lema...) para entrenar los clasificadores, lo que resulta costoso en tiempo dado que son cantidades grandes de datos.

De esta forma se consigue que los algoritmos aprendan de los datos etiquetados para luego ser capaces de clasificar otros datos de entrada. Como ventaja, estos métodos tienen la habilidad de adaptar y crear modelos entrenados para objetivos y contextos concretos.

1.3.2 Aplicación de métodos basados en léxico

Estos enfoques tienen en común que emplean listas predefinidas de palabras, donde cada palabra está asociada a un sentimiento específico. Los métodos basados en léxico varían según el contexto en el que se crean, por ejemplo, LIWC Tausczik & Pennebaker[39] fue originalmente propuesto para analizar patrones de sentimientos en textos formales escritos en inglés, mientras que PANAS-t Gonçalves et al.[14] fue adaptado al contexto web.

A pesar de que no tienen la parte negativa de necesitar un conjunto de datos etiquetados, tienen la dificultad de crear un diccionario basado en léxico que sea aplicable en múltiples contextos.

1.4 Estado del arte

En la sección anterior se describían las aproximaciones más comunes actualmente en el análisis de sentimientos. Ya que tenemos una idea, en esta sección se ilustra brevemente el estado del arte.

En investigación, el análisis de sentimientos se ha desarrollado a tres niveles, según Liu[22] y Westerski[43] :

1.4.1 Análisis de sentimientos a nivel de documento

El objetivo de este análisis es clasificar todos los sentimientos expresados por los autores a lo largo del documento, concluyendo si lo expresado en el documento es positivo, negativo o neutro, sobre una entidad que puede ser un producto o servicio. La mayoría de las técnicas de análisis de sentimientos a nivel de documento obtienen un acierto de clasificación entre el 70% y el 80%,cuando son aplicadas a un solo tipo de texto, dependiendo de la cantidad de texto que se tenga como entrada y del tipo de texto.

Algunos de las soluciones más destacadas en este área, son:

Por una parte, el trabajo desarrollado por Turney[40] para clasificación de críticas, donde presenta un algoritmo de tres pasos que procesa los documentos sin supervisión humana. Este se basa en la

medida de la distancia entre los adjetivos encontrados en el texto a palabras preseleccionadas con polaridad conocida. Brevemente, los pasos que sigue son:

- Paso 1: Extrae los adjetivos del documento aplicando una serie de patrones predefinidos (como nombre-adverbio, nombre-adjetivo...etc)
- Paso 2: Mide la orientación semántica. Para ello se mide la distancia a palabras cuya polaridad se conoce (”excelente” y ”pobre”). Obtiene la dependencia entre dos palabras analizando el número de ocurrencias con el motor de búsqueda AltaVista para documentos que contienen dos palabras relativamente próximas.
- Paso 3: Finalmente cuenta la orientación semántica media para todos los pares de palabras y clasifica la crítica como recomendada o no recomendada.

Por otro lado, Pang et al.[31] presentaron otro trabajo basándose en técnicas conocidas de clasificación. En la propuesta se testea un grupo seleccionado de algoritmos de aprendizaje automático, comprobando si producen buenos resultados cuando se aplica el análisis de sentimientos a nivel de documento.

Presentan resultados para Naive Bayes Lewis[21], Máxima entropía Berger et al.[2] y Máquinas de soporte vectorial Joachims[19], con un acierto de clasificación final de 71%-85%, dependiendo de la técnica y el conjunto de datos empleados.

1.4.2 Análisis de sentimientos a nivel de oración

Dos son los objetivos de este análisis: el primero es identificar si la frase contiene opiniones (es subjetiva) u objetiva. El otro es clasificar la frase si es subjetiva, en positiva, negativa o neutral. Al igual que el anterior nivel de análisis descrito, la mayoría de las soluciones propuestas a nivel de oración aplican técnicas de machine learning.

Entre los trabajos más conocidos centrados en encontrar frases subjetivas estáRiloff & Wiebe[35] donde propusieron un método que usa clasificadores de alta precisión basados en listas hechas de palabras indicativas de subjetividad, para extraer las oraciones subjetivas. En la primera fase del algoritmo, las frases con alto nivel de confianza son etiquetadas por dos clasificadores (primero, se etiquetan las subjetivas con alto nivel de confianza, y luego las objetivas con alto nivel de confianza).

Las frases que no tengan un alto nivel de confianza de pertenecer a una u otra categoría quedan sin etiquetar en esta primera fase.

En la segunda etapa del algoritmo, con los datos etiquetados obtenidos en la fase anterior se entrena un algoritmo de extracción que genera patrones para oraciones subjetivas, patrones que después son usados para buscar más sentencias subjetivas en el texto. Tras procesar todo el conjunto de datos de entrenamiento, se ordenan los patrones extraídos basándose en su ocurrencia, entre otras condiciones, y se toman los mejores para la siguiente iteración del análisis.

A diferencia de la técnica anterior, Yu & Hatzivassiloglou[48] presentaron un trabajo en el que se clasificaba la frase en objetiva o subjetiva, pero también la orientación de las frases subjetivas(positiva,negativa o neutral). Para la primera, los autores presentan resultados para detección de similaridad en oraciones, naive bayes y naive bayes multiclase. Para la segunda tarea, emplean una técnica similar a la de Turney[40].

1.4.3 Análisis de sentimientos a nivel de aspecto

Este nivel de análisis es el más detallado, lo que lo convierte en el más útil pero también el más complicado, ya que el objetivo no es solo determinar qué frases son objetivas o subjetivas y si estas últimas tienen orientación positiva, negativa o neutral, sino que también tiene como objetivo detectar qué es esactamente lo que le gusta o no al autor.

Una de las primeras soluciones más conocidas a este problema fue Hu & Liu[17], un enfoque basado en el léxico que utilizaba la mayor frecuencia de las entidades, aspectos o servicios. Sobre éste se publica una mejora 4 años más tarde Ding et al.[9], presentando así un método que funciona bastante bien en la práctica y consta de cuatro pasos:

- Paso 1. Marcar expresiones de sentimientos: consiste en marcar todas las expresiones de sentimientos en cada frase que contengan uno o más aspectos con +1 o -1. Por ejemplo: *"el altavoz de este teléfono no es bueno"* sería marcada con un (+1) dado que "bueno" es una palabra asociada a polaridad positiva.

- Paso 2. Aplicar desplazadores de sentimientos: esto son, palabras o frases que pueden cambiar la orientación del sentimiento. Por ejemplo, "no" es un desplazador. Ahora la frase anterior sería marcada con (-1).

- Paso 3. Manejo de palabras y frases adversativas: palabras y frases que indican contrariedad necesitan ser manejadas, ya que suelen cambiar la orientación de los sentimientos. Por ejemplo, ”pero” o”sin embargo”..son conjunciones adversativas. Un ejemplo de manejo sería *”el altavoz de este teléfono no es bueno (-1), sin embargo la cámara es estupenda (+1)”*

- paso 4. Agregar puntuación de los sentimientos: para ello se suele aplicar una función como

$$puntuacion(a_i, s) = \sum_{se_j} \frac{se_j.ss}{dist(se_j, a_i)}$$

  donde se_j es la expresión de un sentimiento en la frase s, a_i es el aspecto(objeto, servicio...) i-ésimo de la frase s , $se_j.ss$ es la puntuación del sentimiento se_i y $dist(se_j, a_i)$ es la distancia entre el aspecto a_i y el sentimiento se_j en la oración s.

Si la puntuación final es positiva, entonces la opinión sobre el aspecto a_i en la frase s es positivo. Si la puntuación es negativa, la opinión sobre el aspecto es negativa. Si es 0, la opinión sobre ese aspecto es neutral.

También se han planteado métodos que usan aprendizaje automático en este nivel de análisis, como por ejemplo Jiang et al.[18] que emplea un árbol de parseo sintáctico para generar un conjunto de características que representen algunas relaciones sintácticas de la entidad o aspecto objetivo, y otras palabras.

2

Objetivos del trabajo

En este trabajo se implementan las tres primeras herramientas para un software de PLN en Español. El trabajo se implementa usando el lenguaje Scala, a partir de las propuestas de Smedt & Daelemans[38] y se puede consultar una planificación temporal de los objetivos del trabajo en la parte de resolución del trabajo, en el capítulo 5, sección de Planificación. Igualmente se puede consultar una introducción a Scala en esta parte, en el capítulo 4.

Los objetivos del trabajo se detallan brevemente a continuación de manera lineal:

Documentación y revisión bibliográfica

Aquí el objetivo es adquirir conocimiento general sobre la temática del procesamiento del lenguaje natural, tokenización, lematización y pos etiquetado para Español, documentarse sobre qué trabajos se han realizado anteriormente, la metodología a seguir, qué trabajos hay ahora y el estado del arte.

Elección de técnicas a implementar y lenguaje de programación, planificación del trabajo, análisis de requisitos y diseño

El segundo objetivo consiste en:

- elegir tres técnicas: de tokenización, Pos tagger y lematización respectivamente. Finalmente se decide implementar las tres técnicas para estas herramientas que presenta Smedt & Daelemans[38] para español.
- elegir el lenguaje de programación: se elige el lenguaje Scala para la implementación de este trabajo por sus múltiples ventajas a pesar de que su extructura resulta inicialmente compleja. Esta decisión se detalla en el capítulo resolución del trabajo, en la parte de Introducción a Scala.
- realizar un análisis de los requisitos y de diseño de las herramientas a implementar y su integración conjunta. Se detalla dentro del capítulo de resolución del trabajo, en la parte de Análisis y diseño.

Implementación en código, integración y evaluación de las técnicas implementadas

Aquí se implementan los diferentes módulos de tokenización, etiquetado morfosintáctico y parseo, siguiendo un desarrollo guiado por pruebas (TDD) escribiendo primero los test, usando para ello la librería Scalatest y el estilo FunSuite. Esto se detalla en el capítulo de resolución del trabajo, en la parte de implementación y pruebas.

Análisis y exposición de resultados, comparaciones y conclusión

Aquí se discuten y analizan los resultados obtenidos, comparandolos con los del paper original y con los de otros softwares de NLP disponibles, y se exponen vías futuras del mismo. Se detalla en el capítulo conclusiones y vías futuras.

Resolución del trabajo

A partir de aquí el trabajo se estructura como sigue:

- En el Capítulo 3 se hace una revisión bibliográfica.
- En el Capítulo 4 se hace una introducción a Scala, para visualizar el porqué de su elección.
- En el Capítulo 5 se habla de como se ha planificado el proyecto, análisis de requisitos, diseño y metodología de desarrollo.
- En el Capítulo 6 se explica el material empleado para la resolución de este trabajo, corpus de datos, ficheros para el contexto, el léxico o la morfología...todo lo que se ha utilizado pero no se ha implementado en este proyecto.
- En el Capítulo 7 se detalla el proceso de implementación y pruebas de los tres algoritmos desarrollados.
- En el Capítulo 8 se exponen los resultados del trabajo y marco experimental.
- En el Capítulo 9 se exponen conclusiones y vías futuras.

3

Revisión bibliográfica

3.1 El pipeline Genérico

El objetivo final de toda suite de PLN es tener un sofware que contenga un conjunto de herramientas para poder realizar análisis de sentimientos o algún tipo de extracción de información. Esto se alcanza a través de varias tareas de procesamiento del texto, empezando por hacer el contenido uniforme y acabando por identificar las funciones de las palabras y como se organizan. Aquí se describen las tareas más comunes del pipeline de un software de PLN junto con las aproximaciones más comunes.

3.1.1 Obtención de datos

La obtención de datos es la primera tarea a realizar, mediante la cual se obtendrá el corpus de texto, etiquetado o no (depende de la aproximación a aplicar) que se va a usar para crear la herramienta de PLN, así como los datos sobre los que se quiere hacer el análisis de sentimientos o la extracción de información deseada.

La forma general de obtener los datos directamente de internet es a través de la propia API de los sitios web de los que se desee descargar información por ejemplo la Api de twitter si queremos descargar tweets, o la api de instagram para imágenes y etiquetas. Sin embargo, a pesar de que hay manuales que explican como consultarlas, estas APIs han limitado mucho las descargas últimamente, alegando cuestiones de privacidad de los datos de usuario. Otro método es usar arañas web para rastrear la información deseada de forma similar a lo que hace google con su conocida araña Googlebot, con la que visita todas las webs frecuentemente para añadirlas a su índice y percatarse de los cambios. Ahora bien, este último método requiere unos conocimientos necesarios para crear una araña web y ocupa mucho ancho de banda.

En cuanto a corpus ya etiquetados en inglés se pueden encontrar muchos, entre los más populares están The Penn Treebank Marcus et al.[25] o Wikicorpus http://www.cs.upc.edu/~nlp/wikicorpus/, mientras que para español son conocidos Wikicorpus, Ancora http://clic.ub.edu/ancora o SemEval Màrquez et al.[26].

3.1.2 Procesamiento del texto

Teniendo el texto, se le aplica un procesamiento, separándolo en unidades relevantes, añadiendo información sintáctica y morfológica, extrayendo entidades y relaciones entre estas... En este apartado se describen las fases de procesamiento más comunes según describe Rodrigues & da Silva Teixeira[36].

Tokenización

En esta fase se separa el texto del documento en sus unidades atómicas, los tokens (palabras, números, símbolos), por lo que es una fase totalmente necesaria en casi cualquier suite de PLN. Aunque no es una tarea muy compleja para lenguajes que emplean espacios entre palabras, como la mayoría de lenguajes que usan el alfabeto latino, sí que resulta mucho más complicado en lenguajes que no usan espacios entre palabras, como el Chino. Chang et al.[7].

La tokenización se sirve de heurísticas simples, como considerar que todos los strings de caracteres del alfabeto contiguos forman parte del mismo token (igualmente para los números) y que todos los tokens van separados unos de otros por espacios, saltos de línea o por signos de puntuación que

no sean abreviaciones. Algunas herramientas actuales que hacen tokenización son Freeling Padró & Stanilovsky[30], Apache OpenNLP Baldridge[1] y los ya citados NLTK y StanfordNLP. No hay ninguna herramienta especialmente dedicada a tokenizar, ya que la tokenización puede ser razonablemente bien hecha usando expresiones regulares cuando se procesan lenguajes que usan el alfabeto latino. Para otros lenguajes más complejos, como el Árabe o el Chino, sí se precisan de tokenizadores más complejos, como el Stanford Word Segmenter que aplica segmentación de palabras.

Detección de límites de oraciones

En esta fase se aborda el problema de establecer los límites de una oración en el texto. En algunos casos esto se incluye dentro de la fase de tokenización. Encontrar los límites de una oración no es una tarea trivial, ya que las marcas de puntuación que delimitan el final de una frase suelen ser ambiguas en muchos lenguajes. En español lo son, ya que por ejemplo, el punto se puede usar como marcador de final de oración pero también como separador entre parte entera y decimal en números reales o en iniciales y abreviaciones (por ejemplo, Srta.).

Algunos sisemas que han demostrado buenos resultados en separación de frases sobre diferentes lenguajes naturales, son Punkt Kiss & Strunk[20] e iSentenizerWong et al.[46].

Lematización

Lematización es el proceso mediante el cual se determina el lema de cada palabra. El lexema de una palabra se define como la unidad mínima que es parte común en todas las palabras de una misma familia. Por ejemplo, *"arte"* , *"artístico"* o *"artista"* son tres palabras distintas que comparten el lexema *"art"*. Lema es la representación de la forma canónica (o forma de diccionario) de los lexemas, y tiene por tanto significado. Por ejemplo, *"escribían"*, *"escribieron"* o *"escribirán"* son palabras derivadas de un mismo lexema cuyo lema es *"escribir"*.

Este proceso es importante porque reduce el número de términos a procesar, ya que muchos se reducen al mismo lema, acortando así la complejidad computacional del problema. La dificultad de este proceso varía dependiendo del lenguaje, resultando mucho más sencillo para lenguajes con una morfología inflexiva simple, como el Inglés, y complicándose más para lenguajes morfológicamente más ricos como el Español o el Alemán.

Stemming

Este método es común en lugar de la lematización e incluso en algunos casos se incluye dentro de ésta o como complemento. El Stemming es un proceso simple mediante el cual se trata de reducir la palabra a su forma base eliminando sus sufijos. Se obtiene así una forma de la palabra que no es necesariamente la raíz de la palabra, pero que suele bastar, dado que palabras de la misma familia tienen esa parte en común, o un conjunto reducido de partes en común, si son palabras irregulares. Por ejemplo: *"sleep"*, *"sleeping"*, *"sleeped"*, *"sleeps"*... tienen el común el stem *"sleep"*, que es la forma base del verbo. En lenguajes con pocas inflexiones, como el Inglés, es muy probable que el lema y el stem coincidan. Sin embargo, en lenguajes con más inflexiones, como el Español, eso rara vez ocurre.

POS Tagging

POS Tagging es un proceso muy importante en tareas de PLN, ya que etiqueta mediante algoritmos cada palabra con su PART-OF-SPEECH, es decir, con su categoría morfosintáctica (nombre, verbo, preporisición, adjetivo...) junto a otras propiedades que dependen de esta. De hecho es necesario para realizar el parseo sintáctico (el paso posterior).

Aunque lo usual es aplicar un proceso de lematización o stemming antes de esta fase, hay sistemas que aplican primero esta fase, como el desarrollado en este proyecto.

Los principales desafíos que se presentan en esta etapa, son:

- Tratar la ambigüedad, dado que las palabras pueden desempeñar distinta función morfosintáctica y por tanto, tener diferentes POS tags dependiendo del contexto.
- Asignar una etiqueta morfosintáctica a palabras de las que el sistema no tiene conocimiento

Para tratar de solventar los problemas anteriores, normalmente se tiene en cuenta el contexto dentro de la frase de la palabra a etiquetar, y se selecciona la etiqueta morfosintáctica más probable dada esa palabra y su contexto.

Que sea una tarea muy necesaria hace que esté muy investigada, por lo que actualmente la precisión de etiquetado ronda el 90%, aunque esto se debe a que la precisión se mide en porcentaje de palabras bien etiquetadas. Si se midiera por porcentaje de frases completamente bien etiquetadas, la cifra descendería al 55%-57% de precisión, como manifestaban Giesbrecht & Evert[12].

La mayoría de estas herramientas han sido desarrolladas para el Inglés y evaluadas usando los datos de Penn Treebank. Destacan por ejemplo Stanford POS Tagger que tiene modelos de etiquetado para 6 lenguajes diferentes, SVMTool Giménez & Marquez[13] que se basa en clasificadores de máquinas de soporte vectorial y TreeTager, basado en modelos de Markov Schmid[37].

Parseo Sintáctico

El parseo sintáctico es una tarea que consiste en analizar las oraciones para producir estructuras que representen como se organizan las palabras en las frases, dada una gramática formal. Las gramáticas tienen dos posibles formalismos estructurales:

- Constituyente: es una unidad dentro de una estructura jerárquica que está compuesta por una palabra o un grupo de palabras.
- De dependencia: estas gramáticas desciben la estructura de una frase en términos de los enlaces entre las palabras, ya que cada enlace refleja una relación de dominancia/dependencia entre un término y otro dependiente de él. Estas gramáticas son las que normalmente se emplean para parsear texto.

Por ejemplo, para la frase "this book has two authors" ("este libro tiene dos autores")

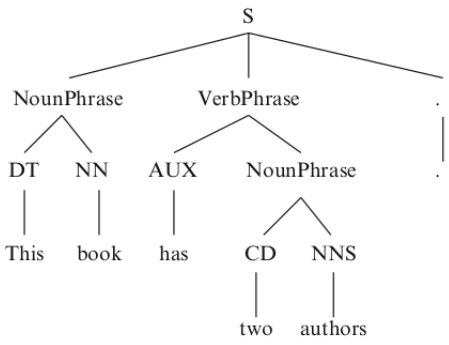


Figure 3.1: Posible árbol de gramática Constituyente

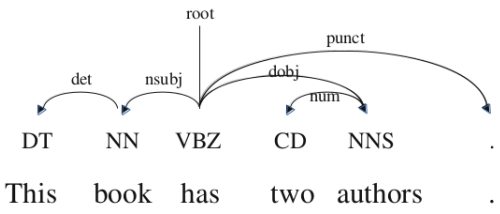


Figure 3.2: Posible árbol de gramática de Dependencias

El parseo sintáctico una tarea computacionalmente muy intensiva, por lo que a veces es deseable sustituirla por otra menos costosa en cómputo que produzca resultados similares, como localizar patrones textuales. Sin embargo, esta sustitución no siempre es posible, dado que deben de tenerse fuentes de información semiestructuradas, estructuradas, o generadas por una máquina.

Algunas herramientas para parseo sintáctico son StanfordParser, MaltParser Nivre et al.[29] o TurboParser `http://www.cs.cmu.edu/~ark/TurboParser/`, estas dos últimas implementan gramáticas de dependencias.

Aquí concluye la parte genérica del pipeline. Además del pipeline genérico, hay procesos que se engloban en una parte que se conoce como pipeline dependiente del dominio, algunos de los cuales se detallan en la siguiente sección.

3.2 El pipeline de OpenNLP

En la sección anterior vimos el pipeline genérico que suelen seguir los softwares de NLP. En esta se describe OpenNLP `http://opennlp.apache.org/`, un conocido framework desarrollado por Apache.

Apache OpenNLP es una librería desarrollada en Java que incorpora un kit de herramientas basadas en aprendizaje automático para hacer procesamiento del texto en lenguaje natural, permitiendo al usuario crear su propio pipeline de PLN, entrenar sus propios modelos o evaluarlos. Como consecuencia, no ofrece soporte para ningún lenguaje específico, sino que ofrece algoritmos bastante genéricos que podrían funcionar con cualquier lenguaje. Sí que tiene algunos modelos pre entrenados para algunos lenguajes.

3.2.1 Herramientas que incorpora

las herramientas de OpenNLP son accesibles a través de su API o por línea de comandos. Estas son las herramientas que incorpora:

Detector de frases

Un detector de frases capaz de detectar si un signo de puntuación marca el final de una frase o no. Se entiende por frase la mayor secuencia de caracteres separados por espacios en blanco entre dos signos de puntuación. La frase inicial y final de un texto son excepcionales, los primeros caracteres que no sean caracteres en blanco se asumen como el inicio de una frase, mientras que los últimos caracteres no blancos se asumen como el final de la frase. Una vez que se han detectado correctamente las fronteras de las oraciones, se formatea el texto debidamente.

Tokenizador

OpenNLP incorpora tres tokenizadores que segmentan una entrada de cadena de caracteres en tokens (palabras, signos, números..etc).

- Tokenizador de espacios en blanco: toma como tokens toda secuencia que no sean caracteres en blanco.
- Tokenizador simple: basado en clases de caracteres, donde las secuencias con la misma clase de caracter se consideran tokens.
- Tokenizador basado en aprendizaje: tokenizador de máxima entropía que detecta las fronteras de los tokens basándose en un modelo probabilístico.

Detokenizador

esta herramienta hace justo lo contrario que el tokenizador, es decir, construye el texto original no tokenizado a partir de una secuencia de tokens. La implementación se basa en reglas que definen como los tokens deben añadirse a la cadena de caracteres: *MERGE_TO_LEFT* añade el token por la

izquierda, *MERGE_TO_RIGHT* añade el token por la derecha y *RIGHT_LEFT_MATCHING* añade el token a la derecha d la primera ocurrencia y a la izquierda de la segunda ocurrencia.

Name entity recognition: NER

Es la herramienta que se encarga de buscar nombres de entidades y números en el texto. Para ello necesita un modelo que dependerá del lenguaje y el tipo de entidad que se quiera identificar. OpenNLP incluye algunos modelos pre entrenados sobre varios corpus libres disponibles. Para encontrar nombres en filas de texto, el texto debe ser previamente segmentado en frases y tokens.

Categorizador de documento

El categorizador de documento de OpenNLP puede clasificar texto dentro de categorías predefinidas. Se basa en un framework de máxima entropía. De nuevo, para clasificar el texto se necesita de un modelo, pero OpenNLP no provee de ninguno pre entrenado, dado que para clasificar se requiere de requisitos específicos.

POS tagger

El POS tagger que incorpora OpenNLP marca los tokens con su correspondiente tipo de palabra (según su categoría morfosintáctica) y para ello se basa en información del propio token y del contexto del mismo. Para resolver la ambiguedad que se presenta cuando un token puede tener múltiples etiquetas morfosintácticas, se aplica un modelo de probabilidad para predecir la etiqueta correcta. Para limitar el posible número de etiquetas que puede recibir un token, se emplea un diccionario de etiquetas.

Un *diccionario de etiquetas* es un diccionario de palabras donde se especifica qué etiquetas puede tener cada posible token.

Chunker

Esta herramienta separa el texto en sintagmas correlados, como el sujeto y el predicado de la frase, pero no especifica su estructura interna ni su función en la frase principal.

Parseador

OpenNLP tiene dos implementaciones para el parseado: una llamada ”chunking parser” que genera como salida el texto parseado separado por espacios, y el ”tree insert parser” que genera un árbol como resultado, pero este último está aún en fase experimental. En ambos caso se requiere el entrenamiento de un modelo con datos en formato de OpenNLP (formato Penn treebank pero limitado a una frase por línea).

Resolución de coreferencias

La resolución de coreferencias es una herramienta que incorpora OpenNLP para linkar las múltiples referencias que se hacen a una misma entidad a lo largo del documento, aunque se limita a menciones de sintagmas nominales, por ejemplo "el gato".

3.2.2 Modelos que incorpora

Además de las herramientas anteriores, OpenNPL incorpora modelos preentrenados para los siguientes idiomas: Inglés, Alemán, Portugués, Danés, Holandés, Sueco y Español. `http://opennlp.sourceforge.net/models-1.5/`. Aunque en algunos enfoques se invierte el orden, todos los modelos aquí son entrenados aplicando primero detección de frases y luego tokenización (además de otras fases).

Para el Inglés incorpora modelos para todas las fases: tokenizador, detección de frases, pos Tagger, NER, chunker, parser y resolución de Coreferencias. Para el Sueco, Portugués, Alemán o Danés incluye tokenizador, detección de frases y pos Tagger, mientras que para el Holandés incluye los anteriores y además el NER. Para el Español, sólo incluye el NER.

3.3 Limitaciones

En Hirschberg & Manning[16] dicen que la mayor limitación de los sistemas de PLN actuales es que dan soporte a los lenguajes más hablados y más extendidos, como el Inglés, Francés, Alemán, Español o Chino, pero otros como el Swahili, Bengali o Indonesio hablados por millones de personas no tienen disponible una herramienta de este tipo.

Sin embargo, aunque el Español está entre los focos de mira de las suites de PLN más usadas actualmente, rara vez hay un pipeline completo para análisis de sentimientos en español que proporcione resltados similares a los que se obtienen para el Inglés. Por ejemplo, en la sección anterior explicábamos Apache OpenNLP, que directamente no incluye herramientas como lematización o análisis de sentimientos para ningún idioma.

Quizá la suite de NLP más conocida en la actualidad y referenciada junto con Python NLTK es Stanford CoreNLP, debido a que es bastante completa, está bien documentada y tiene módulos para muchos de los idiomas más usados:

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			✔			

Figure 3.3: Disponibilidad de herramientas e idiomas en Stanford CoreNLP

Pero de nuevo vemos que sólo está completo para Inglés, faltando métodos como la lematización o el análisis de sentimientos para Español. Otro paquete menos conocido que incluye funcionalidad para minería web, procesamiento del lenguaje natural y machine learning es Pattern Smedt & Daelemans[38] implementada en Python, que según el benchmark de métodos de análisis de sentimientos que hicieron Ribeiro et al.[34] da mejores resultados en media que Stanford CoreNLP, aunque el benchmark solo hace comparaciones para el Inglés.

Pattern incluye una serie de herramientas para Español, pero esta tampoco incluye análisis de sentimientos para este lenguaje.

Ante estas limitaciones el proyecto desarrollado en este trabajo se centra en implementar las tres primeras etapas del pipeline de un software NLP para análisis de sentimientos en español: tokenizador, lematizador y POS tagger.

Teniendo como finalidad a largo plazo construir un software que haga como etapa final análisis de sentimientos en Español.

"If I were to pick a language to use today other than Java,
it would be Scala."

James Gosling

4

Introducción a Scala

SCALA, (Scalable language) es un lenguaje de programación de propósito general cuya primera versión fue lanzada en 2004, y desde entonces ha ido creciendo enormemente en usuarios. Estas son algunas razones por las que se decide usar Scala en este proyecto Wampler & Payne[41]:

Es escalable

Tal y como su nombre indica, este lenguaje ha sido diseñado para crecer con las demandas de sus usuarios, por lo que es una buena opción para escribir desde scripts pequeños a grandes sistemas, abordar desafíos actuales como el Big data o proporcionar servicios con gran disponibilidad y robustez. Esta es la principal razón por la que se escoge este lenguaje para este proyecto, dado que la intención es ir ampliándolo.

Soporta un paradigma mixto

Por una parte Scala soporta programación orientada a objetos (POO), mejorando los objects de Java incluyendo los traits, una forma clara de implementar los tipos usando composiciones mixtas. En Scala todo son objetos realmente, incluso los tipos numéricos. Por otro lado, también soporta totalmente programación funcional (FP), herramienta que se ha convertido en la mejor forma de pensar en concurrencia, Big data y corrección del código en general (empleo de inmutabilidad, funciones de primera clase, funciones de alto orden...).

Tiene un sofisticado sistema de tipos

Extiende el sistema de tipos de Java con otros tipos genéricos más flexibles y otras mejoras para mejorar la corrección del código. Además Scala incorpora un mecanismo de inferencia de tipos.

Es estaticamente tipado

Scala incorpora el tipado estático como herramienta para crear aplicaciones más robustas, pero añade algunas modificaciones para hacerlo más llevadero, como incorporar la inferencia de tipos y hacerlo más flexible, permitiendo identificación de patrones y nuevas formas de escribir y componer tipos.

Un lenguaje JVM y Javascript

Explota las funcionalidades y optimizaciones de JVM, así como la gran cantidad de liberías y herramientas disponibles para Java. Además tiene un puerto para JavaScript (Scala.js).

Sintaxis concisa, elegante y flexible

Si de algo hablan los programadores de Scala es de su sintaxis y de las reducciones de código que experimentan con respecto a Java. Por ejemplo:

```
// código en Java
 class Persona {
     private int edad;
```

```
    private String nombre;

    public Persona(int edad, String nombre) {
        this.edad = edad;
        this.nombre = nombre;
    }
}

// código en Scala
  class Persona(edad: Int, nombre: String)
```

Sí, dado este código en Scala el compilador creará dos atributos privados edad(entero) y nombre(cadena de caracteres) y un constructor que tomará los valores que se le pasen inicialmente para inicializar esas variables. Más rápido de escribir, de leer, y menos errores.

Más ejemplos de Scala

A continuación se muestran algunos ejemplos simples de Scala obtenidos de Raychaudhuri[32]

4.0.1 Encontrar caracter en mayúscula

Éste es un ejemplo en Java y Scala del problema de encontrar un caracter en mayúcula en una cadena de texto:

```
// código en Java
 boolean hasUpperCase = false;
for(int i = 0; i < name.length(); i++) {
    if(Character.isUpperCase(name.charAt(i))) {
        hasUpperCase = true;
        break;
    }
}
// código en Scala
    val hasUpperCase = name.exists(_.isUpper)
```

Mientras que el código en Java itera sobre cada caracter del string, comprobando uno a uno hasta que encuentra una mayúscula, modifica el flag boolenano y se sale del bucle, en Scala basta con llamar a la función exist sobre el string name, devolviendo un booleano. La _ representa cada caracter de la cadena. De nuevo Scala seduce con su sintaxis.

4.0.2 Contar las líneas de un fichero

```ruby
# código en Ruby
count = 0
File.open "someFile.txt" do |file|
   file.each { |line| count += 1 }
end
```

```scala
// código en Scala
val count = scala.io.Source.fromFile("someFile.txt").getLines().map(x => 1).sum
```

En Ruby también se realiza de forma breve, pero no tan elegante, ya que necesita una variable contador que va incrementando en cada vez que cuenta una línea. En Scala, una posible forma de hacerlo es usando map, una función que a cada línea del fichero le hace corresponder un 1. Finalmente los suma todos llamando a sum.

Además, Scala soporta composición mixta a través de los traits. Los traits son parecidos a las clases abstractas con implementación parcial. Por ejemplo, podríamos crear un nuevo tipo de colección que permitiera acceder al contenido del fichero como un iterable, mezclando el trait Iterable de Scala:

```scala
class FileAsIterable {
  def iterator = scala.io.Source.fromFile("someFile.txt").getLines()
}
```

Ahora si lo mezclamos con el trait Iterable de Scala, al crear un objeto de esa clase, éste será un Iterable, teniendo acceso a los métodos de Iterable:

```
val newIterator = new FileAsIterable with Iterable[String]
newIterator.foreach { line => println(line) }
```

donde el método foreach al que llama, es de Iterable.

Scala y la concurrencia

Uno de los grandes problemas de la concurrencia es que si no se coordina bien el acceso a los recursos puede haber cambios inesperados e indeseados en los mismos, por acción de alguna de las hebras. Scala ayuda en esto con la inmutabilidad, de hecho, por defecto lo hace todo inmutable. Aunque en Scala se puede usar cualquier mecanismo de Java, Scala incorpora también sus propias herramientas específicas para concurrencia. Algunas de ellas son: Wampler & Payne[41]

4.0.3 Futures

La API scala.concurrent.Future simplifica la concurrencia en código. Los Futures empiezan a correr concurrentemente cuando son creados, aunque lo hacen de forma asíncrona. Se puede hacer que las tareas sean independientes y sin bloqueo o por el contrario, bloquear, y además la API ofrece muchas funcionalidades para manejar los resultados (que pueden ser un Future), como callbacks que pueden ser invocados cuando el resultado esté listo. Veamos un ejemplo simple en el que se lanzan concurrentemente 5 tareas:

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
def sleep(millis: Long) = {
Thread.sleep(millis)
}
// Busy work ;)
def doWork(index: Int) = {
sleep((math.random * 1000).toLong)
index
}
```

```
(1 to 5) foreach { index =>
val future = Future {
doWork(index)
}
  future onSuccess {
    case answer: Int => println(s"Success! returned: $answer")
  }
  future onFailure {
    case th: Throwable => println(s"FAILURE! returned: $th")
  }
}
sleep(1000) // Wait long enough for the "work" to finish.
println("Finito!")
```

Se usa un método SLEEP para simular la ocupación por un tiempo. El método DOWORK llama a sleep con un número aleatorio de milisegundos. Se itera con foreach en un rango de enteros (1 a 5 inclusive) y se llama a scala.concurrent.Future.apply. En este caso, Future.apply recibe una función de trabajo a realizar (doWork), y devuelve un nuevo objetoFuture, el cual ejecuta el cuerpo de doWork(index) en otra hebra, devolviendo el control inmediatamente al bucle. Finalmente se usan onSucess (si la tarea se completa correctamente) y onFailure (si falla) para registrar los callbacks.

4.0.4 MODELO DE ACTORES

Una regla de la concurrencia es: si puedes, no compartas. Los actores son entidades software independientes que no comparten unos con otros información mutable que se pueda alterar. En su lugar se comunican mediante mensajes, eliminando la necesidad de sincronizar el acceso a datos, estados e información mutable. De esta forma es más fácil crear aplicaciones concurrentes robustas. Cada actor cambia su estado según lo necesite, pero sólo si tiene acceso exclusivo a ese estado y sus invocaciones se garantizan seguras.

Scala y Big data

Las funciones map, flatMap, filter, fold, reduce...etc siempre han sido funciones para trabajar con datos, independientemente de que estos fueran grandes o pequeños. Es por esto que una vez que se entiende Scala y sus colecciones, resulta fácil enganchar una API de Big data basada en Scala, sin embargo, sabiendo Java, enganchar con MapReduce Java API resulta más complicado por ser más difícil de usar y estar a más bajo nivel.

Pero la principal ventaja de Scala en Big data frente a otros lenguajes no es la curva de aprendizaje, sino la programación funcional. Scala permite escribir lo mismo (o más), con menos código que otros lenguajes. El ejemplo simple de contar el número de palabras, sacado de Wampler & Payne[41] bastará para evidenciar a simple vista la ventaja de las APIs Big data basadas en Scala frente a las basadas en Java.

Esto es sólo parte de la versión Hadoop. El original se compone de unas 60 líneas de código, sin imports ni comentarios.

```
// src/main/java/progscala2/bigdata/HadoopWordCount.javaX
...
class WordCountMapper extends MapReduceBase
   implements Mapper<IntWritable, Text, Text, IntWritable> {

 static final IntWritable one = new IntWritable(1);
 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) {
   totalCount += counts.next.get;
}
```

Ahora una versión Scalding, basado en Scala. El código orignial tiene 12 líneas de código, contando el import:

```
// src/main/scala/progscala2/bigdata/WordCountScalding.scalaX
import com.twitter.scalding._

class WordCount(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")))
}
```

"How well we communicate is determined not by how well we say things, but how well we are understood."

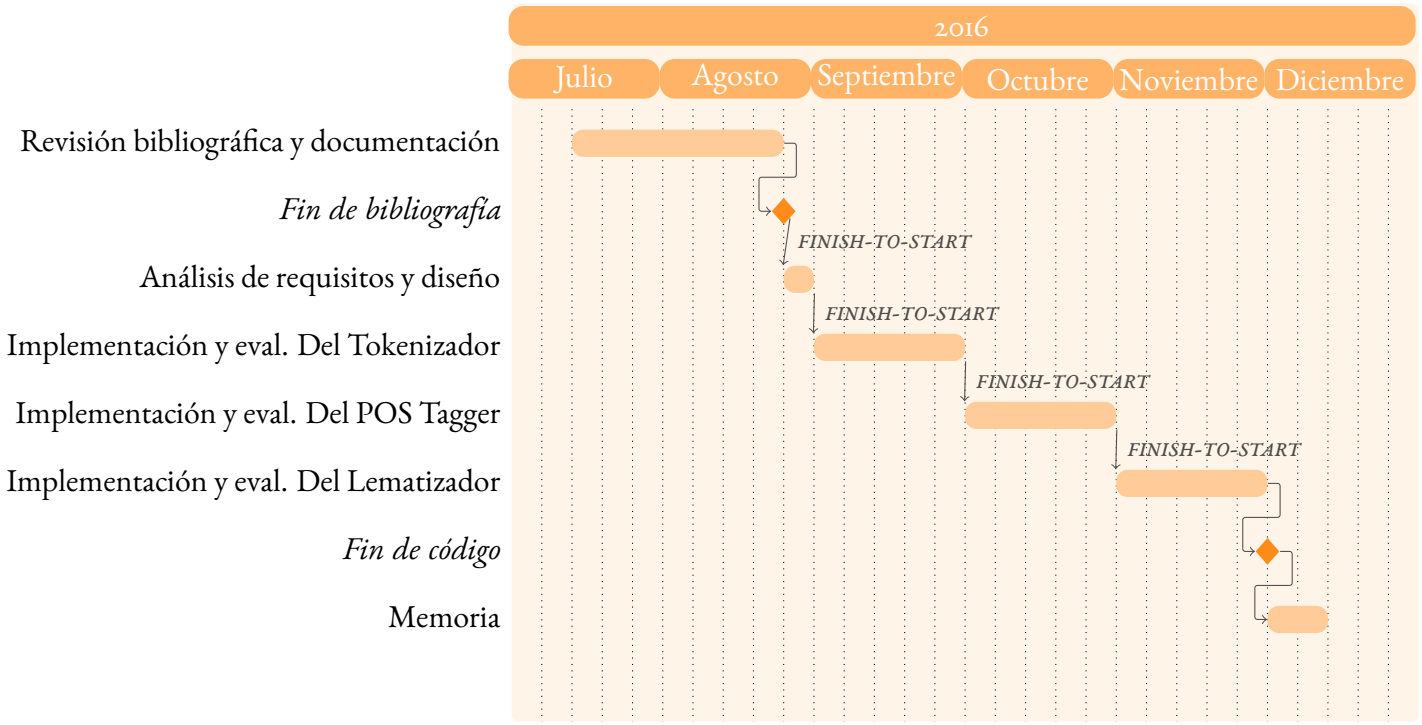
Andrew Grove

5

Planificación, análisis y diseño

Planificación

A continuación se expone mediante un diagrama de Gantt como se ha distribuido el tiempo durante estos 5 meses para cada tarea de este proyecto, a partir de mediados de Julio hasta la segunda semana de Diciembre.



Análisis de requisitos

En esta sección se muestra la especificación de requisitos para el proyecto de manera formal, a través de plantillas que muestran información relevante del requisito, como la descripción, su prioridad, estado o riesgo. Al ser demasiados, se exponen solo los más importantes.

REQUISITOS FUNCIONALES

Identificador: RF-01	
Necesidad: Alta	Autor: Cristina Heredia
Descripción: El sistema incluirá un método de parseado de texto, al que el usuario especificará una cadena de texto y cual de las tres tareas desea realizar(tokenizado, etiquetado,lematización) mediante parámetros booleanos. Devuelve el resultado de aplicarle al texto especificado la acción solicitada. Si no se especifica opción, devuelve el texto tal como lo especificó el usuario.	
Prioridad: Alta	Riesgo: Medio
Dependencias: RF-02, RF-03, RF-07	Estado: Hecho

Identificador: RF-02	
Necesidad: Alta	Autor: Cristina Heredia
Descripción: El sistema incluirá un método de procedimiento de tokenización de texto, que separará el texto en sus unidades mínimas; los tokens. Si el texto contiene emoticonos, los cambia por el sentimiento que tengan asociado. Recibe como parámetro una cadena de texto especificada por el usuario. Devuelve una lista de listas de tokens, donde cada lista de tokens se corresponde con una frase del texto del usuario.	
Prioridad: Alta	Riesgo: Medio
Dependencias:	Estado: Hecho

Identificador: RF-03	
Necesidad: Alta	Autor: Cristina Heredia
Descripción: El sistema incluirá un método de etiquetado morfosintáctico de texto (POS Tagger), que etiquetará cada palabra(token) con su categoría morfosintáctica según el léxico, una morfología y un contexto que recibirá como parámetro. También recibe como parámetro una lista con los tokens del texto introducido por el usario, por lo tanto requiere de una tokenización previamente. Devuelve una lista de pares (token,etiqueta).	
Prioridad: Alta	Riesgo: Medio
Dependencias: RF-02,RF-04,RF-05, RF-06	Estado: Hecho

Identificador: RF-04	
Necesidad: Alta	Autor: Cristina Heredia
Descripción: El sistema incluirá un método de aplicación de un contexto a una lista de tokens ya etiquetados morfosintácticamente, para mejorar el etiquetado de los mismos. Como resultado devuelve una lista con los tokens y sus nuevas etiquetas, en un par (token,tag).	
Prioridad: Alta	Riesgo: Medio
Dependencias: RF-02, RF-05	Estado: Hecho

Identificador: RF-05	
Necesidad: Alta	Autor: Cristina Heredia
Descripción: El sistema incluirá un método de creación de un diccionario de léxico, a partir de la lectura y manipulación de un fichero de 85877 líneas de reglas léxicas. Éste será usado en el POS tagger para realizar un etiquetado inicial.	
Prioridad: Alta	Riesgo: Medio
Dependencias:	Estado: Hecho

Identificador: RF-06	
Necesidad: Alta	Autor: Cristina Heredia
Descripción: El sistema incluirá un proceso de aplicación de reglas morfológicas a un token dado con una etiqueta dada, que se reciben como parámetros. Ésto sirve de nuevo para mejorar el etiquetado, esta vez teniendo en cuenta la morfología.	
Prioridad: Alta	Riesgo: Medio
Dependencias: RF-02, RF-05	Estado: Hecho

Identificador: RF-07	
Necesidad: Alta	Autor: Cristina Heredia
Descripción: El sistema incluirá un proceso de lematización que obtendrá la forma base de la palabra, aplicando un diccionario y varios algoritmos. Requiere de un etiquetado previo de las palabras, por lo que recibe como argumento una lista de (palabra,etiqueta) y devuelve una lista de (palabra,etiqueta,lema).	
Prioridad: Alta	Riesgo: Medio
Dependencias: RF-03	Estado: Hecho

REQUISITOS NO FUNCIONALES

Identificador: RNF-01	
Necesidad: Alta	Autor: Cristina Heredia
Descripción: El método de Pos etiquetado a implementar para el español debe tener un acierto medio mayor o igual a 0.80.	
Prioridad: Alta	Riesgo: Medio
Dependencias:	Estado: Hecho

Identificador: RNF-02	
Necesidad: Alta	Autor: Cristina Heredia
Descripción: El método de lematización para español a implementar debe tener un acierto medio mayor o igual a 0.90.	
Prioridad: Alta	Riesgo: Medio
Dependencias:	Estado: Hecho

Identificador: RNF-03	
Necesidad: Alta	Autor: Cristina Heredia
Descripción: El sistema debe implementarse usando siempre inmutabilidad.	
Prioridad: Media	Riesgo: Medio
Dependencias:	Estado: Propuesto

Diseño

En este apartado se explica lo relativo al diseño de clases y métodos realizado para este trabajo junto con la metodología que se ha seguido para su implementación.

Metodología usada

En este trabajo se ha tratado de seguir la metodología de desarrollo dirigido a documentos (TDD), un conocido procedimiento de desarrollo cuya idea es hacer las pruebas inicialmente, de forma previa a la implementación. En concreto, se hacen test unitarios para testear las funcionalidades más importantes del trabajo.

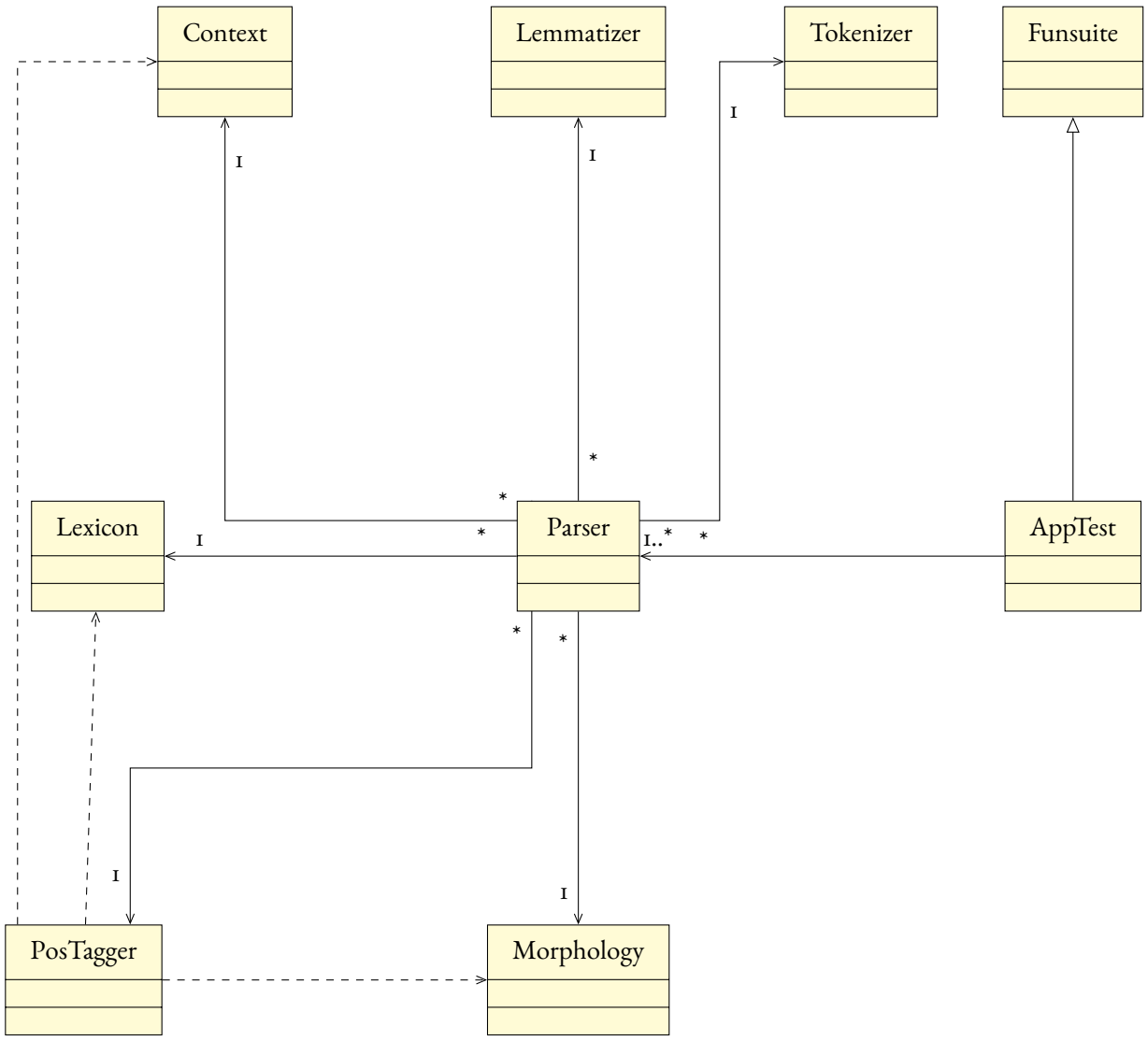
El procedimiento seguido es el siguiente: en primer lugar se elige una funcionalidad a implementar, por ejemplo, el tokenizado de una cadena de texto. Luego se escribe una prueba para testear si dicha funcionalidad es correcta o no, prueba que inicialmente no pasará al no estar implementada aún dicha funcionalidad:

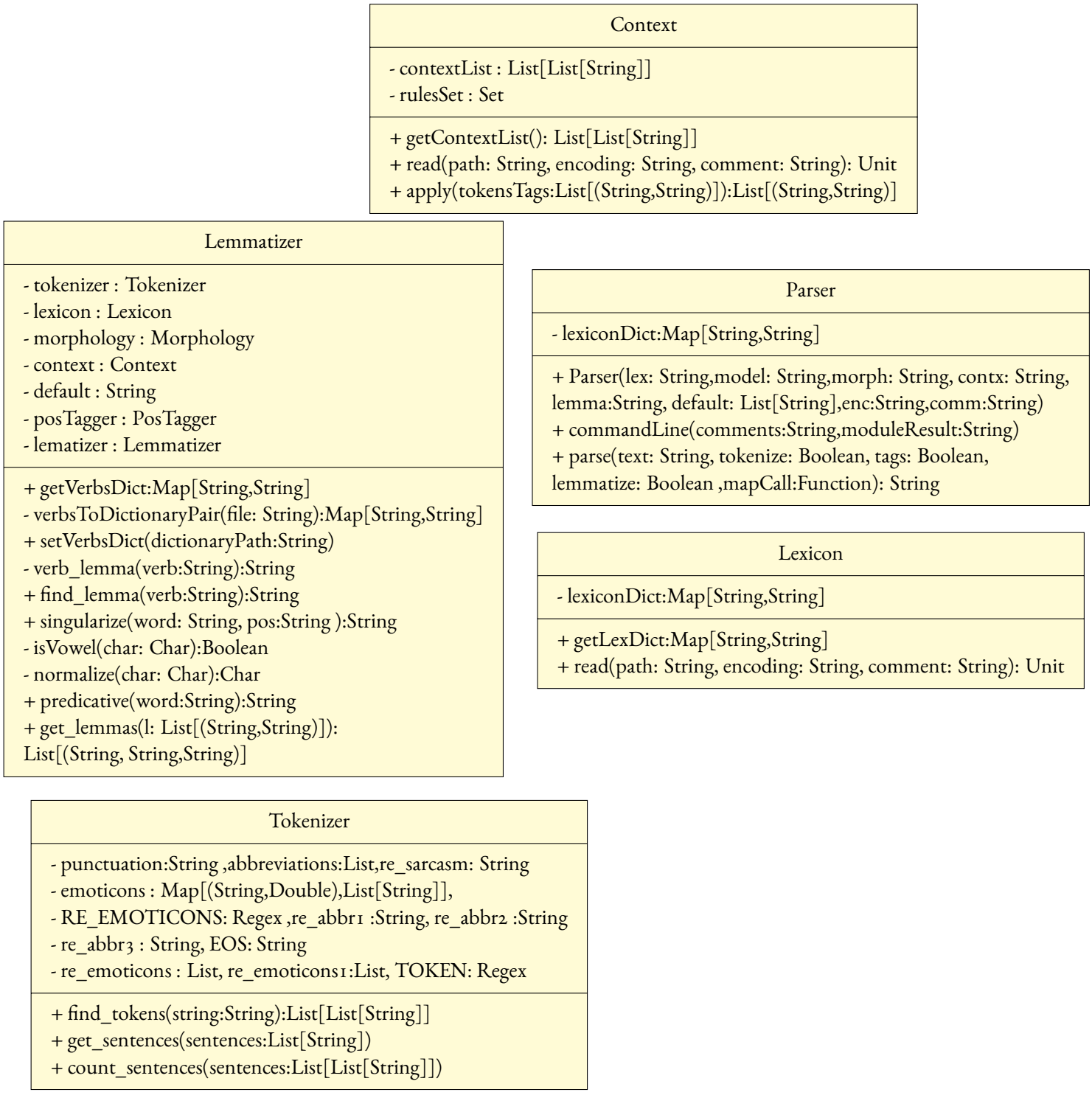
```
//testing find tokens method
  test("find tokens test") {
    assert(tokenizer.find_tokens(string1) == List(List("Los", "gatos", "negros",
     ↪  "son", "bonitos", ".")))
    assert(tokenizer.find_tokens(string2) == List(List("Nadie", "podrá",
     ↪  "vivir", "eternamente")))
    //....
```

El siguiente paso es implementar el método find_tokens. Una vez implementado, se vuelve a ejecutar el test, y si lo pasa se refactoriza y se vuelve a ejecutar el test. Después se toma otra funcionalidad y se repite el proceso hasta que se hayan implementado todas las funcionalidades. Para los test se ha utilizado la librería ScalaTest y el estilo FunSuite.

Diseño de clases

Se ha intentado mantener privados todos los datos miembro así como todos los métodos que no fueran necesarios de ser accedidos fuera de la clase que los implementa. A continuación se expone un diagrama de clases. Aunque las clases no se definen de forma detallada en este digrama por su tamaño, se incluyen más detalladas en las siguientes páginas.




Context
- contextList : List[List[String]]
- rulesSet : Set
+ getContextList(): List[List[String]]
+ read(path: String, encoding: String, comment: String): Unit
+ apply(tokensTags:List[(String,String)]):List[(String,String)]
Lemmatizer
- tokenizer : Tokenizer
- lexicon : Lexicon
- morphology : Morphology
- context : Context
- default : String
- posTagger : PosTagger
- lematizer : Lemmatizer
+ getVerbsDict:Map[String,String]
- verbsToDictionaryPair(file: String):Map[String,String]
+ setVerbsDict(dictionaryPath:String)
- verb_lemma(verb:String):String
+ find_lemma(verb:String):String
+ singularize(word: String, pos:String):String
- isVowel(char: Char):Boolean
- normalize(char: Char):Char
+ predicative(word:String):String
+ get_lemmas(l: List[(String,String)]):
List[(String, String,String)]
Parser
- lexiconDict:Map[String,String]
+ Parser(lex: String,model: String,morph: String, contx: String,
lemma:String, default: List[String],enc:String,comm:String)
+ commandLine(comments:String,moduleResult:String)
+ parse(text: String, tokenize: Boolean, tags: Boolean,
lemmatize: Boolean ,mapCall:Function): String
Lexicon
- lexiconDict:Map[String,String]
+ getLexDict:Map[String,String]
+ read(path: String, encoding: String, comment: String): Unit
Tokenizer
- punctuation:String ,abbreviations:List,re_sarcasm: String
- emoticons : Map[(String,Double),List[String]],
- RE_EMOTICONS: Regex ,re_abbr1 :String, re_abbr2 :String
- re_abbr3 : String, EOS: String
- re_emoticons : List, re_emoticons1:List, TOKEN: Regex
+ find_tokens(string:String):List[List[String]]
+ get_sentences(sentences:List[String])
+ count_sentences(sentences:List[List[String]])

PosTagger
- tagset:String, UNIVERSAL : String, NOUN : String - CD: Regex, X : String - parole: mutable.Map[String,String] - VERB : String, ADJ : String, ADV : String - PRON : String, DET : String, PREP : String - ADP : String, NUM : String, CONJ : String - INTJ : String, PRT : String, PUNC : String
+ setTagset(tagsetname:String) + parole2penntreebank(token:String,tag:String):(String, String) + penntreebank2universal(token: String, tag: String):(String,String) + parole2universal(token:String, tag:String):(String,String) + find_tags(tokens:List[String], lexicon:Lexicon, model:String, morphology:Morphology, context:Context, entities:String, default:List[String], mapCall:(String,String)=>(String,String)): List[(String,String)]

Morphology
- morphologyList:List[List[String]] - rulesSet: Set
+ getMorphology:List[List[String]] + read(path: String, encoding: String, comment: String) + apply(token:String, tag:String, previus:(String,String), next:(String,String), morphology:List[List[String]], lexicon:Map[String,String]): String

6

Material empleado

Anteriormente se mencionaba que los objetivos inicialmente previstos en este trabajo de PLN eran tres: desarrollar un método de tokenizado de texto, otro de POS etiquetado (etiquetado morfosintáctico) y otro de lematización usando el lenguaje Scala. Para ello, se toman como base de referencia las herramientas para Español (Pattern.es) de Pattern Smedt & Daelemans[38].

En este capítulo se detalla como se obtienen los datos que se han usado para la implementación de este proyecto pero que no forman parte de ésta, y que son importantes para el correcto funcionamiento de los algoritmos. Las implementaciones de los algoritmos se detallan en el capítulo siguiente. Vayamos detallando según las fases del proyecto:

6.1 Tokenizador

Pattern presenta un Tokenizador para español basado en reglas, que no necesita más que una cadena de texto a tokenizar.

6.2 Pos Tagger

Determinar el tipo de palabra que es cada token de un texto ya no es una tarea que pueda resolverse sólo aplicando algunas reglas consecutivas (al menos obteniendo resultados algo competitivos), dado que la función de las palabras varía según el contexto o su posición en una frase, la morfología...

El método propuesto en Pattern para el tagger español consiste en usar un corpus etiquetado de palabras suficientemente grande y entrenar un algoritmo de machine learning que aprenda reglas y patrones presentes en esos datos. Concretamente, como corpus usa el Wikicorpus Reese et al.[33] para español, y como algoritmo usa un Algoritmo de Brill:

Wikicorpus

Existen tres variantes para tres lenguas distintas: Español, Catalán e Inglés. El corpus en su versión actual cuenta con unas 750 millones de palabras en total (120 millones para el español) y está construido a partir de texto de la Wikipedia etiquetado con información lingüística, concretamente de lema y POS tag, usando la librería Freeling Padró & Stanilovsky[30].

Algoritmo de Brill

Este método fue propuesto por Eric Brill Brill[6]. Es un algoritmo de aprendizaje también conocido como aprendizaje basado en transformación, dado que cada palabra recibe inicialmente una etiqueta que luego va modificándose a menudo que se aplican los pasos del algoritmo. Normalmente, la condición de parada es que alcance una tasa de error inferior a una obtenida por parámetro. También necesita recibir un corpus etiquetado como parámetro, ya que lo usará para el aprendizaje, un diccionario léxico al que a cada palabra le corresponde su tipo más común (verbo, nombre, etc), y una lista de plantillas de reglas.

El funcionamiento es el siguiente:

1. Etiquetado inicial: El algoritmo comienza asignando a cada palabra su etiqueta más común, para ello la busca en el diccionario léxico que recibe como parámetro. En esta etapa se etiquetan mal muchas palabras, pues no se tiene en cuenta para nada el contexto de las mismas. Además, hay palabras que no estarán en el diccionario.

2. Etiquetar las desconocidas: Padró & Stanilovsky[30] propone dos procedimientos para ello. En primer lugar se considera que las palabras que no estaban presentes en el diccionario léxico y empiezan por mayúscula son nombres propios. En segundo lugar, se etiquetan las palabras desconocidas restantes asignándoles la etiqueta más común en palabras conocidas con las mismas tres últimas letras. Por ejemplo, "roncaba" se etiquetaría como verbo si "cantaba" fuera conocida.

3. Aplicación de reglas iterativamente para mejorar el etiquetado: Se cambiará una etiqueta de a a b cuando se cumpla alguna de las siguiente reglas de la lista de plantillas de reglas:

    - La palabra siguiente(anterior) está etiquetada como z
    - La segunda palabra previa(posterior) está etiquetada como z
    - Una de las dos palabras anteriores(siguientes) está etiquetada como z
    - Una de las tres palabras anteriores(siguientes) está etiquetada como z
    - La siguiente está etiquetada como z y la posterior como w
    - La anterior(siguiente) está etiquetada como z y las dos anteriores(posteriores) a ella están etiquetadas como w
    - La palabra empieza (o no empieza) por mayúscula
    - La palabra anterior empieza (o no empieza) por mayúscula

Donde z, w son dos categorías cualesquiera (VB,NN..). Cabe destacar que las reglas son aplicables tanto a palabras como a etiquetas de las mismas, y se expresan con la notación interna del algoritmo: `"PREVTAG", "NEXTTAG", "PREV1OR2TAG", "NEXT1OR2TAG", "PREV1OR2OR3TAG", "NEXT1OR2OR3TAG", PREV2TAG","NEXT2TAG", "CURWD", "PREVWD", "NEXTWD", "PREV1OR2WD", "NEXT1OR2WD"`.

Un ejemlo de instancia de una regla sería: `TO IN NEXTTAG AT`, que se interpretaría como que la etiqueta `"TO"` cambiaría a `"IN"` si la etiqueta de la palabra anterior es `"AT"`.

Tras la fase del etiquetado inicial y etiquetar las desnococidas, se comparan las etiquetas estimadas con las reales, obtieniendo una lista de errores. A continuación, por cada error de esa lista (por cada palabra mal etiquetada) se instancia cada una de las reglas de la lista de plantillas de reglas, y se evalúan, para determinar cuál de ellas reduce más el error. La ganadora se aplica al texto y se añade a la lista de resultados. Después se vuelve a calcular una lista de errores sobre el nuevo corpus resultante de haber aplicado la mejor regla.

Para determinar cual es la mejor regla, a cada regla se le asigna una puntuación, que se calcula como los aciertos que tiene menos los fallos.

Algorithm 1 Algoritmo de Brill

```
1: procedure BrillTagger(corpus,lexicon,rate_error,rulesTlist)
2:     C_i ← initial tagging of corpus
3:     errors ← compute errors of C_i
4:     Result ← ø
5:     i ← 0
6:     nErrors ← number of errors
7:     best ← ø
8:     while nErrors>rate_error do
9:         for each error in errors do
10:            for each rule in instances(rulesTlist) do
11:                score(rule) ← good corrections of rule - bad corrections of rule in C_i
12:        best ← rule with best score
13:        C_{i+1} ← Apply(best, C_i )
14:        Results ← Results+best
15:        i ← i+1
16: return Results
```

6.2.1 Lectura del corpus

El Wikicorpus para español contiene unos 50 ficheros de texto. Cada uno pesa entre 25MB y 102MB y cada línea de cada uno de ellos es una palabra con su lema y su POS etiqueta en notación Parole http://www.cs.upc.edu/~nlp/SVMTool/parole.html. Un fragmento de ejemplo:

```
 Ibiševi ibiševi NP00000 0
, , Fc 0
nacido nacer VMP00SM 00249716
el el DA0MS0 0
6_de_agosto_de_1984 [??:6/8/1984:??.??:??] W 0
en en SPS00 0
Vlasenica vlasenica NP00000 0
, , Fc 0
Yugoslavia yugoslavia NP00000 0
( ( Fpa 0
actual actual AQ0CS0 01667781
Bosnia bosnia NP00000 0
```

```
y y CC 0
Herzegovina herzegovina NP00000 0
) ) Fpt 0
es ser VSIP3S0 01775973
un uno DI0MS0 0
futbolista futbolista NCCS000 0
bosnio bosnio AQ0MS0 02731920
que que PR0CN000 0
juega jugar VMIP3S0 00727813
como como CS 0
delantero delantero NCMS000 00466114
```

Aunque en este ejemplo no se aprecia, algunas líneas de estos ficheros son metadatos(”<doc>”, ”</doc>”, ”ENDOFARTICLE”, ”REDIRECT”...), por lo que deben ser ignoradas. A continuación se muestra un pseudocódigo de la lectura de este corpus, obteniendo finalmente una lista de pares (palabra, etiqueta). El código en python que emplea Pattern se puede encontrar en la web de CLiPS (`http://www.clips.ua.ac.be/`).

Algorithm 2 Lectura Wikicorpus

```
1:  procedure readWikicorpus(corpusPath=”./tagged.es”,words=1000000
2:      S ← ø
3:      i ← 0
4:      for f in corpusPath do
5:          for line in open(f) do
6:              if line == ” \n ” or startsWith(metadata) then
7:                  continue
8:              w, lemma, tag ← line splitted by (” ”)
9:              if tag starts with(”Fp”) then tag=2 first characters of tag
10:             else if tag starts with(”V”) then tag=3 first characters of tag
11:             else if tag starts with(”NC”) then tag=2 first characters of tag + 3° character of tag
12:             elsetag=2 first characters of tag
13:         for w in split tag by (”_”) do
14:             S ← S+(w,tag)
15:             i ← i+1
16:         if tag == ”Fp” and w == ”.” then S ← S + []
17:         if i ≥ words then return S
```

6.2.2 Obtención del Léxico

El método de Pos tagger español propuesto en Smedt & Daelemans[38], y por lo tanto, el implementado en este proyecto, hace uso de reglas léxicas, morfológicas y contextuales para mejorar la calidad del etiquetado. Pattern usa las herramientas de PythonNLTK para obtener estos ficheros de reglas. En esta sección se explica como se construye el diccionario léxico usado en este trabajo a partir del corpus leído en la sección anterior.

El procedimiento a seguir es el siguiente:

1. Se cuenta la ocurrencia de cada palabra en el Wikicorpus español
2. Se cuenta la ocurrencia de las etiquetas de las palabras en el mismo corpus. Hay palabras que pueden tener varias etiquetas
3. Se construye el diccionario tomando las palabras más frecuentes y su etiqueta más frecuente

Este diccionario léxico se utilizará luego para etiquetar inicialmente los datos en el algoritmo de Brill. De nuevo, se expone una descripción en pseudocódigo de este proceso:

Algorithm 3 Obtención del Léxico

```
1:  procedure createLexicon(corpus, maxLines=100000)
2:      Lexicon ← ø
3:      top ← ø
4:      for sentence in corpus do
5:          for w, tag in sentence do
6:              increase ocurrences of Lexicon(w, tag)
7:      for w, tags in Lexicon do
8:          freq ← ∑ all ocurrences of tags for w
9:          tag ← the tag with bigger ocurrence for w
10:         top ← top + (freq,w,tag)
11:     get the maxLines with bigger ocurrency
12:     write to file
```

Debido a que el resultado que se obtiene es muy grande en tamaño(85877 líneas), el léxico se guarda en el fichero es-lexicon.txt, y se usa directamente en el Pos tagger que implementa este proyecto.

Lo mismo sucede con el contexto y la morfología, ya que entrenar el modelo cada vez que se quiera usar el Pos Tagger es un proceso muy lento y poco recomendable. Por tanto, los algoritmos se ejecutan una vez, guardando los resultados en ficheros para reutilizarlos posteriormente.

Algunas líneas de ejemplo de es-lexicon.txt

```
óleos NCP
ómnibus NC
ónice NCS
ópera NCS
óperas NCP
óptica AQ
ópticas AQ
óptico AQ
```

Obtención de reglas contextuales

Para obtener las reglas que tienen en cuenta los alrededores de la palabra a etiquetar (el contexto), se usa el léxico obtenido en el apartado anterior y el algoritmo de Brill, ya descrito también anteriormente.
El procedimiento a seguir es el siguiente: en primer lugar, se anonimizan los nombres propios presentes en el corpus, ya que la intención es aprender reglas generales del tipo ”un nombre va seguido de un adjetivo” , y no ”Granada va seguido de un adjetivo”. Este proceso de describe en pseudocódigo como:

Algorithm 4 Anonimización de nombres

```
1: procedure ANONIMIZE(CORPUS)
2:     ANONIMOUS ← ”anonymous”
3:     i ← 0
4:     for sentence in corpus do
5:         for (w, tag) in sentence do
6:             if tag == ”Np” then
7:                 s[i] ← (ANONYMOUS,”NP”)
```

En segundo lugar, se entrena el algoritmo de Brill para aprender las reglas contextuales presentes en el corpus. El Pos tagger para español de Pattern Smedt & Daelemans[38] no hace una implementación propia de Brill, sino que utiliza el FastBrillTaggerTrainer de Python NLTK Bird et al.[4], utilizando el léxico obtenido en el apartado anterior para hacer la fase de etiquetado inicial del corpus, un conjunto de reglas como las que explicaron en la explicación del algoritmo, y el corpus resul-

tante del proceso anterior de anonimización. El resultado (de 250 líneas) se guarda en es-context para su reutilización.

Algunas líneas de ejemplo de es-context.txt

```
VMI VMP PREV1OR2WD llevada
W SP CURWD con
Zm NCP PREVWD en
DD PD NEXTTAG PP
W Z SURROUNDTAG Fpa Fpt
NCS RG PREVWD on
SP RG NEXTWD facto
AQ NCS SURROUNDTAG DI AQ
DD PD NEXTTAG P0
NCP Zu PREVTAG Zu
PR CS SURROUNDTAG VMP DA
CS PR CURWD donde
PI DI CURWD una
RG SP CURWD según
```

Obtención de reglas morfológicas

Las reglas morfológicas son aquellas basadas en los prefijos y sufijos de las palabras. Por ejemplo, palabras acabadas en -mente son advervios en Español. En Pattern.es se propone su uso para mejorar la precisión del algoritmo de Pos etiquetado, concretamente, para las palabras desconocidas que se etiquetan como nombres por defecto.

El primer paso es detectar los sufijos y contar su ocurrencia en las palabras que no estén etiquetados como nombres propios. Para detectar si es un sufijo, se toman las 5 últimas letras de la palabra, y se comprueba si son menos letras que las de la palabra original. Por ejemplo: dada la palabra ”flor” sabremos que no es un sufijo porque sus últimas 5 letras (y 4) son ”flor”, y eso mide igual que la palabra inicial. Dicho de otra forma, la palabra en sí no puede ser un sufijo.
Una vez tenemos los sufijos, el segundo paso es calcular la frecuencia de la etiqueta que más aparece con el mismo, creando una nueva regla. Por último, se filtran las reglas, descartando las que no casen al menos con 10 palabras y las que tengan como etiqueta más frecuente una que no se dé al menos en un 80%. También se descartan aquellas cuya etiqueta sea "NCS" (nombre común singular). A

continuación se presenta un pseudocódigo del proceso. El código en python se puede consultar en la web de CLiPS.

Algorithm 5 Obtención de las reglas morfológicas

```
1:  procedure createMorphology(corpus, maxLines=125)
2:      suffix ← ø
3:      f1 ← ø
4:      f2 ← ø
5:      for sentence in corpus do
6:          for w, tag in sentence do
7:              x ← last 5 words of w
8:              if length(x) < length(w) and tag ≠ "NP" then
9:                  increase occurrences of suffix(x,tag)
10:     for x, tags in suffix do
11:         tag ← the tag with bigger ocurrence for x
12:         f1 ← ∑ all ocurrences of tags for words ending in x
13:         f2 ← (number of ocurrences of tag)/f1
14:         top ← top + (f1,f2,x,tag)
15:     top ← order rules by descendent ocurrency
16:     top ← filter (f1,f2,x,tag) which f1 ≥ 10 and f2 > 0.8
17:     top ← get the top maxLines rules
18:     top ← format top
19:     write to file
```

El resultado son reglas formateadas como `[x fhassuf y tag]`, donde `x` es cualquier sufijo e `y` es la longitud del mismo. El resultado se guarda en es-morphology.txt

Algunas líneas de ejemplo de es-morphology.txt

```
NC ado fhassuf 3 VMP x
NC ido fhassuf 3 VMP x
NC ico fhassuf 3 AQ x
NC ivo fhassuf 3 AQ x
NC osa fhassuf 3 AQ x
NC oso fhassuf 3 AQ x
NV ía fhassuf 2 VMI x
```

6.3 Lematizador

Por último, para el lematizador, se usa un fichero de 576 líneas (es-verbs.txt),público en el repositorio de CLiPS, con las conjugaciones de 576 verbos españoles obtenidos de la web `http://users.ipfw.edu/jehle/verblist.htm`, para crear un diccionario interno al que a cada verbo conjugado se le asocia su forma base(lema). Ésto se explica en la parte de implementación.

Algunas líneas de ejemplo de es-verbs.txt

```
abrazar,abrazo,abrazas,abraza,abrazamos,abrazáis,abrazan,abrazando,abracé,abrazaste...
abrir,abro,abres,abre,abrimos,abrís,abren,abriendo,abrí,abriste,abrió,abrimos...
aburrir,aburro,aburres,aburre,aburrimos,aburrís,aburren,aburriendo,aburrí...
```

7

Implementación y pruebas

En este capítulo se explican la implementaciones llevadas a cabo en este trabajo y las pruebas que se han realizado sobre las mismas. Los algoritmos por se describen por orden en el pipeline.

7.1 Tokenizador

El proceso de tokenización implementado recibe como entrada una cadena de texto y da como salida los tokens de la cadena. Si la cadena se compone de varias frases, entonces da como salida el texto segmentado en frases tokenizado. Por ejemplo, si recibe como entrada:

```
"El cielo es azul. El agua es transparente y las amapolas son rojas."
```

Dará como salida: `["El", "cielo", "es", "azul", "."]`
`["El", "agua", "es", "transparente", "y", "las", "amapolas", "son", "rojas", "."]`
El algoritmo de tokenizado se puede dividir en 4 fases:

1. Preprocesado el string: manejo de comillas (simples y dobles), espacios en blanco, retornos de carro, saltos de línea.

2. Obtención de los tokens del texto

3. Separación de frases

4. Manejo de sarcasmo y emoticonos

Tanto el algoritmo como las estructuras usadas por éste se implementan en una clase Tokenizer. Igual se ha hecho con el resto de clases, pretendiendo así aislar todo la funcionalidad relativa a una clase en la implementación de la misma. Veamos el algoritmo en detalle:

7.1.1 Preprocesado del texto recibido

```
def find_tokens(string:String):List[List[String]]={
    var s=string
    val punc=punctuation.replace(".","").toCharArray

  //handle unicode quotes
    if(s.contains("“")) s.replace("“"," “ ")
    if(s.contains("”")) s.replace("”"," ” ")
    if(s.contains("‘")) s.replace("‘"," ‘ ")
    if(s.contains("’")) s.replace("’"," ’ ")
   //collapse whitespace
    s="""\r\n""".r.replaceAllIn(s,"\n")
    s="""\n{2,}""".r.replaceAllIn(s,EOS)
    s="""\s+""".r.replaceAllIn(s," ")
```

Donde punctuation es una cadena con signos de puntuación, que se define dentro de la clase como:

```
  private[this] val punctuation = ".,;:!?()[]{}`'\"@#\$^&*+-|=~_"
```

En la segunda línea de la función se elimina el punto de entre los signos de puntuación, dado que el punto lo manejaremos aparte, para evitar confusiones como creer que "..." es un final de frase al leer su primer punto. Con eso se construye un array de caracteres punc.
El siguiente paso es espaciar las comillas del texto, ya que el tokenizador busca tokens entre espacios en blanco. Por ejemplo, la frase: `Como dice la canción,"vive y sé feliz".` Quedaría: `Como dice la canción, " vive y sé feliz " .`

Finalmente se hace uso de expresiones regulares para sustituir la presencia de `\r\n`, que es el separador de línea que suele usar Windows por un salto de línea, así como cambiar uno o más caracteres en blanco por uno solo y dos o más saltos de línea por EOS, definida anteriormente dentro de la clase como:

```
  private[this] val EOS = "END-OF-SENTENCE"
```

7.1.2 Obtención de los tokens

Para obtener los tokens se define el token con una expresión regular de la siguiente forma:

```
private[this] var TOKEN="""(\S+)\s""".r
```

Es decir, uno o más caracteres que no sean espacios en blanco seguidos de un espacio en blanco. Ésta expresión regular se usa para encontrar todas las coincididencias de palabras seguidas de un espacio en blanco en el texto, en este caso s. Luego, se itera sobre cada una de las coincidencias encontradas de la siguiente manera:
Se comienza eliminando el espacio en blanco final de la palabra: por ejemplo "gatos " $\Rightarrow$ "gatos", y se comprueba si la primera letra de la palabra es un signo de puntuación distinto del punto. Si lo es, se quita la primera letra de la palabra y se mete en la lista de tokens, repitiendo el mecanismo hasta que la primera letra de la palabra no sea un signo de puntuación. Después comienza de nuevo un proceso iterativo, en el que se comprueba si la última letra de la palabra es un signo de puntuación, incluido el punto. Aquí pueden darse tres casos:

- Si el final de la palabra son puntos suspensivos: Por ejemplo "cantaba..." en este caso se quitan los tres puntos finales y se añaden a una lista para ser añadidos después.
- Si el final de la palabra es un signo de puntuación pero no es ningún punto: se quita el signo de puntuación y se añade a una la lista, para ser añadido después.
- Si la palabra acaba con un punto: en ese caso se comprueba si es una abreviación (Por ejemplo: Srta.) para ello se hace uso de una lista de abreviaciones y expresiones regulares definidos en la clase:

```
private[this] val abbreviations=List("a.C.", "a.m.", "apdo.", "aprox.",
 ↪  "Av.", "Avda.", "c.c.", "D.", "Da.", "d.C.",
   "d.j.C.", "dna.", "Dr.", "Dra.", "esq.", "etc.", "Gob.", "h.", "m.n.",
    ↪  "no.",
```

```
        "núm.", "pág.", "P.D.", "P.S.", "p.ej.", "p.m.", "Profa.", "q.e.p.d.",
         ↪  "S.A.",
        "S.L.", "Sr.", "Sra.", "Srta.", "s.s.s.", "tel.", "Ud.", "Vd.", "Uds.",
         ↪  "Vds.",
        "v.", "vol.", "W.C.")

    private[this] val re_abbr1="""^([A-Za-z]\.)+\$""".r
    private[this] val re_abbr2="""^[A-Z][a-z]{1,3}\.\$""".r
```

donde re_abbr1 empareja abreviaciones como D.,U.S.,c.c., y re_abbr2 empareja abreviaciones como Dr.,Sr.,Dra.,Uds.. Si se reconoce como abreviación, se mete en la lista de tokens, mientras que si no es una abreviación se quita el punto y se añade a una la lista, para ser añadido después.

- finalmente se añaden tanto la palabra como el signo de puntuación que se ha ido guardando para después en la lista de tokens. La desripción en código es la siguiente:

```
var tokens=List[String]()

  //get tokens and handle punctuation marks
  TOKEN.findAllIn(s+" ").foreach(t => if(t.length>0){
   var tail=mutable.MutableList[String]()
   var t2=t.stripSuffix(" ")
   while (punc.contains(t2.head)){
    tokens::=t2.head.toString
    //tokens=tokens.reverse
    t2=t2.tail
   }
   breakable {
    while (punc.contains(t2.last) || t2.endsWith(".")) {
     if (t2.endsWith("...")) {
      tail += ("...")
      t2 = t2.substring(0, t2.length - 3)
```

```
    }
    else if (punc.contains(t2.last)) {
     tail += (t2.substring(t2.length - 1))
     t2 = t2.substring(0, t2.length - 1)
    } //split elipsis (...) before splitting period

    //split period(if not an abbreviation)
    if (t2.endsWith(".")) {
     if ((abbreviations.contains(t2) || re_abbr1.findAllMatchIn(t2).length > 0
      ↪  || re_abbr2.findAllMatchIn(t2).length > 0) != true) {
      tail += (t2.substring(t2.length - 1))
      t2 = t2.substring(0, t2.length - 1)
     }else break()
    }
   }
  }

  if( t2.compareTo("")!=0){
   tokens::=t2
  }
  if(!tail.isEmpty) {
   tail.foreach(u=> tokens::=u.toString )
  }
 })
```

Puesto que en el código se ha ido insertando por la cola, sólo quedaría invertir la lista para tener los tokens:

```
tokens=tokens.reverse
```

7.1.3 Separación de frases

Obtenida la lista de tokens, queda segmentarla por frases. El procedimiento es el siguiente: se itera sobre la lista de tokens obtenida, comprobando si el token es algún signo de fin de sentencia

("...","","","!","","?","","EOS"). Cuando se encuentra un signo de fin de sentencia, se toman todos los tokens leídos (incluído el de final de sentencia) como una frase. A partir de aquí empieza a considerarse que empieza la frase siguiente. El proceso termina cuando se ha iterado sobre todos los tokens. Su descripción en código, es:

```
var j=0
var i=0
var sentences=List[String]()
 while (j < tokens.length) {

  if (tokens(j) == "..." || tokens(j) == "." || tokens(j) == "!" || tokens(j)
   ↪  == "?" || tokens(j) == EOS) {
   j += 1
   sentences ::= tokens.slice(i, j).filter(t => t != EOS).mkString(" ")
   i = j
  }
  j += 1
 }
sentences::=tokens.slice(i,j).filter(t=>t!=EOS).mkString(" ")
```

Hay frases que no tienen signo de puntuación final. Se podría considerar que si no tiene signo de puntuación final no está bien estructurada y por lo tanto no es válida, pero se ha preferido considerar que si no tiene signo de puntuación final, la frase se acaba cuando no hay más tokens. Por ejemplo `"Ese boli negro escribe muy bien"` se entendería como una única frase.

Debido a esto, se añade la última línea de código anterior, ya fuera del bucle. Para textos bien estructurados (con frases con una marca de fin) esta sentencia última resulta en añadir un sentencia vacía al conjunto de sentencias, sin embargo esta se filtra antes de devolver el resultado de forma muy cómoda en `Scala`.
Por último, en la clase tokenizador también se implementan dos métodos que imprimen las sentencias de tokens y cuentan las sentencias de tokens, respectivamente.

7.2 POS Tagger

El algoritmo de Pos tagger realiza el etiquetado morfosintáctico del texto. Para ello recibe una lista de tokens y devuelve una lista de tuplas (token, etiqueta) de los mismos. Para el etiquetado se usan

etiquetas en el formato de PAROLE http://www.lsi.upc.edu/~nlp/SVMTool/parole.html , pero se añade opción de especificar una función que las mapee a otro formato como PennTreebank o el formato universal.
Los parámetros que recibe el algoritmo son:

- una lista de tokens
- un objeto de la clase léxico
- un objeto de la clase morfología
- un objeto de la clase contexto
- una lista de string con tres etiquetas por defecto(”NCS”,”NP”,”Z”)
- una función para mapear el formato de las etiquetas a otro formato(opcional)

El funcionamiento es el siguiente:

1. Etiquetado inicial: se itera sobre los tokens, etiquetando las palabras conocidas. Para ello se buscan en el diccionario de léxico que se especifica por parámetro, asignándole la etiqueta que dicha palabra tenga el diccionario léxico. Si no se encuentra la palabra en el diccionario, esta se etiqueta como `"None"` en esta primera fase del algoritmo.
2. Etiquetar las desconocidas: para ello se itera sobre la colección obtenida en el paso anterior. Si una palabra tiene etiqueta desconocida pero es una palabra que empieza por mayúscula, se cambia su etiqueta a `"NP"` (nombre propio). Si por el contrario, casa con la expresión regular:

   ```
   private[this] val CD = """^[0-9\-\,\.\:\/\%\$]+\$"""
   ```

   es decir, es un dígito, símbolo o combinación de ambos, se etiqueta como `"Z"` (dígito). Si se ha proporcionado una morfología al algoritmo, se aplican las reglas morfológicas presentes en la misma para cambiar la etiqueta de la palabra. Por último, si ninguna de las anteriores aplica, se etiqueta como `"NCS"`.
3. Aplicación de reglas contextuales: Si se ha especificado un contexto, éste se aplica para mejorar el etiquetado anterior de todas las palabras.

4. Si se ha especificado una función para mapear las etiquetas a otro formato, ésta se aplica sobre el resultado anterior.

El algoritmo en código se encuentra dentro de la clase PosTagger y es el siguiente:

```
def find_tags(tokens:List[String], lexicon:Lexicon, morphology:Morphology,
 ↪  context:Context, default:List[String],
                mapCall:(String,String)=>(String,String)):List[(String,String)]={

    var tagged=List[(String,String)]()
    var taggedMorp=List[(String,String)]()
    var taggedCntxt=List[(String,String)]()
    var taggedfin=List[(String,String)]()
    // Tag known words.
    tokens.foreach(t=>
     ↪  tagged::=(t,lexicon.getLexDict.getOrElse(t,lexicon.getLexDict.getOrElse(t.toLowerCase,"None"
    //Tag unknow words
    tagged=tagged.reverse
    taggedMorp=tagged.map(t=> {
      var prev = ("None", "None")
      var next = ("None", "None")
      if (tagged.indexOf(t) > 0) prev = tagged(tagged.indexOf(t) - 1)
      if (tagged.indexOf(t) < (tagged.length - 1)) next =
       ↪  tagged(tagged.indexOf(t) + 1)
      if (t._2 == "None") {
        //use NP for capitalized words
        if (t._1.matches("""^[A-Z][a-z]+.\$""")) (t._1, default(1))
        //use CD for digits and numbers
        else if (t._1.matches(CD)) (t._1, default(2))
        //use suffix rules (ej, -mente=ADV)
        else if (!morphology.getMorphology.isEmpty) (t._1,morphology.apply(t._1,
         ↪  default(0), prev, next, lexicon.getLexDict))
          // Use most frequent tag (NCS).
```

```
        else (t._1, default(0))

      } else (t._1,t._2)
    })
    //Tag words by context
    if(!context.getContextList.isEmpty ) taggedCntxt=context.apply(taggedMorp)
    else taggedCntxt=taggedMorp
    //Map tag with a custom function
    if(mapCall != null){
      taggedCntxt.foreach(t => taggedfin ::= mapCall(t._1, t._2))
      taggedfin=taggedfin.reverse
    } else taggedfin=taggedCntxt
    return taggedfin
  }
```

7.2.1 Aplicación del léxico

En la clase Lexicon se implementa un método que lee el fichero es-lexicon.txt y crea un Diccionario Léxico como dato miembro de la clase. La cabecera del método de lectura es la siguiente:

```
def read(path: String, encoding: String, comment: String): Unit
```

recibiendo como parámetros la ruta del fichero, la codificación y el formato de los comentarios del mismo. La aplicación del léxico en el método anterior es trivial, pues no es más que consultar un diccionario.

7.2.2 Aplicación de la morfología

En la clase Morphology se implementa un método que lee el fichero es-morphology.txt y crea una lista interna a la clase de reglas morfológicas (morphologyList). La cabecera es igual que la descrita justo arriba. En esta clase se definen los comandos de las reglas morfológicas, teniendo en cuenta los prefijos-sufijos de las palabras, como:

```
private [this] var rulesSet= Set(
    "word", // Word is x.
```

```
    "char", // Word contains x.
    "haspref", // Word starts with x.
    "hassuf", // Word end with x.
    "addpref", // x + word is in lexicon.
    "addsuf", // Word + x is in lexicon.
    "deletepref", // Word without x at the start is in lexicon.
    "deletesuf", // Word without x at the end is in lexicon.
    "goodleft", // Word preceded by word x.
    "goodright"// Word followed by word x.
  )
   rulesSet.foreach(r=> rulesSet.+=("f"+r.mkString))
```

Estas reglas se emplean en el método apply de esta clase, que aplica la morfología a un par (palabra,etiqueta), dado el par previo a la misma de (palabra,etiqueta), y el posterior.
El proceso es éste: se itera sobre la lista de reglas extraídas del fichero (morphologyList), y se comprueba si el comando de la regla está en rulesSet. Las reglas son del tipo: `ly hassuf 2 RB x` o `NN s fhassuf 1 NNS x`, donde el comando es "hassuf y fhassuf" respectivamente. Luego se aplican las reglas de rulesSet a la palabra que se quiere etiquetar y si se cumple alguna, se cambia la etiqueta de la palabra a la que indique la regla actual de morphologyList. El proceso en código es:

```
def apply(token:String,tag:String,previus:(String,String),
 ↪  next:(String,String),lexicon:mutable.Map[String,String]): String ={

   var f = false
   var x =""
   var cmd=""
   var pos=""
   var realTag=""

   morphologyList.foreach(l=> {
     if (rulesSet.contains(l(1))) { // Rule = ly hassuf 2 RB x
       f = false
       x = l(0)
       pos = l(l.length - 2)
```

```
      cmd = l(1).toLowerCase
    }
    if (rulesSet.contains(l(2))) { // Rule = NN s fhassuf 1 NNS x
       f = true
       x = l(1)
       pos= l(l.length -2)
       cmd= l(2).toLowerCase.stripPrefix("f")
    }
    if( f==false || tag.compareTo(l(0))==true){
      if((cmd=="word" && x==token) ||
        (cmd=="char" && token.contains(x)) ||
        (cmd=="haspref" && token.startsWith(x)) ||
        (cmd=="hassuf" && token.endsWith(x)) ||
        (cmd=="addpref" && lexicon.contains(x+token)) ||
        (cmd=="addsuf" && lexicon.contains(token+x)) ||
        (cmd=="deletepref" && token.startsWith(x) &&
         ↪  lexicon.contains(token.substring(x.length))) ||
        (cmd=="deletesuf" && token.endsWith(x) &&
         ↪  lexicon.contains(token.substring(0,token.length-x.length))) ||
        (cmd=="goodleft" && x==next._1) ||
        (cmd=="goodright" && x==previus._1)
      ){
        realTag=pos
      }
    } else realTag=tag
  })
  return realTag
}
```

7.2.3 Aplicación del contexto

En la clase Context se implementa un método que lee el fichero es-context.txt creando una lista de reglas contextuales en la clase (`ContextList`). Recordemos que el algorito de brill creaba reglas de

la forma: `VBD VB PREVTAG IN => una palabra etiquetada como VBD cambia a VB si la etiqueta de la anterior a ella es IN.`. Por tanto, en la clase se definen las reglas basadas en contexto como:

```
private[this] var rulesSet= Set(
    "prevtag", // Preceding word is tagged x.
    "nexttag", // Following word is tagged x.
    "prev2tag", // Word 2 before is tagged x.
    "next2tag", // Word 2 after is tagged x.
    "prev1or2tag", // One of 2 preceding words is tagged x.
    "next1or2tag", // One of 2 following words is tagged x.
    "prev1or2or3tag", // One of 3 preceding words is tagged x.
    "next1or2or3tag", // One of 3 following words is tagged x.
    "surroundtag", // Preceding word is tagged x and following word is tagged y.
    "curwd", // Current word is x.
    "prevwd", // Preceding word is x.
    "nextwd", // Following word is x.
    "prev1or2wd", // One of 2 preceding words is x.
    "next1or2wd", // One of 2 following words is x.
    "next1or2or3wd", // One of 3 preceding words is x.
    "prev1or2or3wd", // One of 3 following words is x.
    "prevwdtag", // Preceding word is x and tagged y.
    "nextwdtag", // Following word is x and tagged y.
    "wdprevtag", // Current word is y and preceding word is tagged x.
    "wdnexttag", // Current word is x and following word is tagged y.
    "wdand2aft", // Current word is x and word 2 after is y.
    "wdand2tagbfr", // Current word is y and word 2 before is tagged x.
    "wdand2tagaft", // Current word is x and word 2 after is tagged y.
    "lbigram", // Current word is y and word before is x.
    "rbigram", // Current word is x and word after is y.
    "prevbigram", // Preceding word is tagged x and word before is tagged y.
    "nextbigram" // Following word is tagged x and word after is tagged y.
  )
```

El procedimiento de aplicación de las reglas contextuales es similar al de las morfológicas:
se tiene una lista de tuplas de (token, etiqueta) a la que se le añaden tres delimitadores por cada extremo y se itera sobre ella.
En cada elemento de esa lista se itera sobre cada regla de `contextList` , y si la etiqueta de la tupla (token, etiqueta) actual no es un delimitador y ésta coincide con la primera etiqueta de la regla de `contextList` sobre la que se itera, se comprueban las reglas contextuales correspondientes a la etiqueta considerada y el comando de la regla actual de `contextList`. Si empareja alguna, se modifica la etiqueta de la tupla con la etiqueta que indique la regla. Finalmente se eliminan los delimitadores del resultado. El proceso en código es como sigue:

```
def apply(tokensTags:List[(String,String)]):List[(String,String)]={
    val o=List(("STAART", "STAART"),("STAART", "STAART"),("STAART", "STAART"))
     ↪  //empty delimiters for look ahead/back

   var t=o.++(tokensTags).++(o)
  // var mapped=mutable.MutableList[(String,String)]()
  var mapped=List[(String,String)]()
    var cmd=""
    var x=""
    var y=""
    var r1=""
    var matches=false
    var index=0

   t.foreach(token=> { this.contextList.foreach(r=>
     if ((token._2 != "STAART") && (token._2 == r(0) || r(0) == "*")) {
       cmd = r(2).toLowerCase
       x = r(3)
       y = if (r.length > 4) r(4) else ""

       if ((cmd == "prevtag" && x == t(index - 1)._2) ||
         (cmd == "nexttag" && x == t(index + 1)._2) ||
         (cmd == "prev2tag" && x == t(index - 2)._2) ||
         (cmd == "next2tag" && x == t(index + 2)._2) ||
```

```
    (cmd == "prev1or2tag" && ((t(index - 1)._2, t(index -
     ↪  2)._2).toString().contains(x))) ||
    (cmd == "next1or2tag" && ((t(index + 1)._2, t(index +
     ↪  2)._2).toString().contains(x))) ||
    (cmd == "prev1or2or3tag" && ((t(index - 1)._2, t(index - 2)._2, t(index
     ↪  - 3)._2).toString().contains(x))) ||
    (cmd == "next1or2or3tag" && ((t(index + 1)._2, t(index + 2)._2, t(index
     ↪  + 3)._2).toString().contains(x))) ||
    (cmd == "surroundtag" && (x == t(index - 1)._2 && y == t(index + 1)._2))
     ↪  ||
    (cmd == "curwd" && (x == t(index)._1)) ||
    (cmd == "prevwd" && (x == t(index - 1)._1)) ||
    (cmd == "nextwd" && (x == t(index + 1)._1)) ||
    (cmd == "prev1or2wd" && (t(index - 1)._1, t(index -
     ↪  2)._1).toString().contains(x)) ||
    (cmd == "next1or2wd" && (t(index + 1)._1, t(index +
     ↪  2)._1).toString().contains(x)) ||
    (cmd == "prevwdtag" && (x == t(index - 1)._1 && y == t(index - 1)._2))
     ↪  ||
    (cmd == "nextwdtag" && (x == t(index + 1)._1 && y == t(index + 1)._2))
     ↪  ||
    (cmd == "wdprevtag" && (x == t(index - 1)._2 && y == t(index)._1)) ||
    (cmd == "wdnexttag" && (x == t(index)._1 && y == t(index + 1)._2)) ||
    (cmd == "wdand2aft" && (x == t(index)._1 && y == t(index + 2)._1)) ||
    (cmd == "wdand2tagbfr" && (x == t(index - 2)._2 && y == t(index)._1)) ||
    (cmd == "wdand2tagaft" && (x == t(index)._1 && y == t(index + 2)._2)) ||
    (cmd == "lbigram" && (x == t(index - 1)._1 && y == t(index)._1)) ||
    (cmd == "rbigram" && (x == t(index)._1 && y == t(index + 1)._1)) ||
    (cmd == "prevbigram" && (x == t(index - 2)._2 && y == t(index - 1)._2))
     ↪  ||
    (cmd == "nextbigram" && (x == t(index + 1)._2 && y == t(index + 2)._2))
  ) {matches=true ;r1=r(1)}
})
```

```
    if(matches){ mapped::=Tuple2(token._1,r1) ; matches=false} else
     ↪  mapped::=token
  index+=1
  })
   return mapped.reverse.filter(p=>p._1!="STAART")
 }
```

7.2.4 Funciones para mapear los tagsets

Por último, mencionar que en la clase PosTagger se implementan también algunas funciones que mapean las etiquetas de un tagset a otro, en concreto:

```
def parole2penntreebank(token:String,tag:String):(String, String)={
   return (token,parole.getOrElse(tag,tag))
 }
```

mapea las etiquetas del tagset PAROLE a PENNTREEBANK

```
def penntreebank2universal(token: String, tag: String):(String,String)def
 ↪  penntreebank2universal(token: String, tag: String):(String,String) ={
    if (tag.startsWith("NNP-") || tag.startsWith("NNPS-")) return
     ↪  (token,(NOUN.concat(tag.split("-").toString.formatted("%s-%s"))))
      //return (token, "%s-%s" % (NOUN, tag.split("-")[-1]))
    if(List("NN", "NNS", "NNP", "NNPS", "NP").contains(tag)) return (token, NOUN)
    if(List("MD", "VB", "VBD", "VBG", "VBN", "VBP", "VBZ").contains(tag)) return
     ↪  (token, VERB)
    if(List("JJ", "JJR", "JJS").contains(tag)) return (token, ADJ)
    if(List("RB", "RBR", "RBS", "WRB").contains(tag)) return (token, ADV)
    if(List("PRP", "PRP\$", "WP", "WP\$").contains(tag)) return (token, PRON)
    if(List("DT", "PDT", "WDT", "EX").contains(tag)) return (token, DET)
    if(List("IN").contains(tag)) return (token, PREP)
    if(List("CD").contains(tag)) return (token, NUM)
    if(List("CC").contains(tag)) return (token, CONJ)
```

```
    if(List("UH").contains(tag)) return (token, INTJ)
    if(List("POS", "RP", "TO").contains(tag)) return (token, PRT)
    if(List("SYM", "LS", ".", "!", "?", ",", ":", "(", ")", "\"", "#",
     ↪  "\$").contains(tag)) return (token, PUNC)
    return (token, X)
  }
```

mapea las etiquetas de PENNTREEBANK al UNIVERSAL

```
def parole2universal(token:String, tag:String):(String,String)= {

    if(tag == "CS") return (token, CONJ)
    if(tag == "DP") return (token, DET)
    if(List("P0", "PD", "PI", "PP", "PR", "PT", "PX").contains(tag)) return
     ↪  (token, PRON)

    var paroletreebank=parole2penntreebank(token, tag)
    return penntreebank2universal(paroletreebank._1,paroletreebank._2)
  }
```

mapea de PAROLE al UNIVERSAL.

7.3 LEMATIZADOR

Aunque en algunos softwares de PLN la lematicación se hace previamente al etiquetado, en este trabajo se realiza después, ya que se hace uso de las etiquetas de los tokens en el algoritmo de lematización.
En la clase Lematizer se tiene un diccionario de lemas como dato miembro, que se construye a través del fichero es-verbs.txt.

```
private[this] var mappedVerbs = Map[String, String]()
  }
```

El mecanismo de lematización en general es el siguiente: se recibe una lista de tuplas (token, etiqueta) y se itera sobre sus elementos. Entonces si la etiqueta es `"DT"(determinante)` o `"NNS"(nombre`

común en plural) se llama al método singularize, que pasa la palabra a su forma en singular. Si la etiqueta es "JJ"(adjetivo) se llama a predicative, que se encarga de buscar la forma base de los adjetivos y si la etiqueta es "VB" o "MD" (verbo o modal) se llama a get_lemma, que busca la forma base de los verbos. En código:

```
def get_lemmas(l: List[(String,String)]): List[(String, String,String)] = {
    var word :String=""
    var lemma:String=""
    var pos:String=""
    var lemmalist: List[(String, String, String)] = List()

  l.foreach( i => {word=i._1;pos=i._2;lemma=i._1;
  if (pos.startsWith("DT")) lemma=singularize(word,"DT")
  if (pos.startsWith("JJ")) lemma=predicative(word)
  if (pos.startsWith("NNS")) lemma=singularize(word,"NNS")
    if (pos.startsWith("VB") || pos.startsWith("MD")) lemma= verb_lemma(word)
    lemmalist= (word,pos,lemma)::lemmalist
 })
    return lemmalist.reverse
  }
  }
```

Veamos a continuación cada una de las funciones con más detalle.

7.3.1 Singularize

Pasa nombres y determinantes a su forma singular. Para ello comprueba si es un determinante. Si lo es y es "la","las" o "los" lo pasa a "el". Análogamente con "una","unas" o "unos" los pasa a "un".
Luego se comprueba si la palabra acaba en "es"(suele ser nombre plural) y si la subcadena de la palabra desde el inicio de la misma hasta la antepenúltima palabra acaba en "br","i","j","t" o "zn", y si es así se quita la última letra (suele ser una *s*). Posteriormente se itera sobre una lista de terminaciones comunes que son irregulares, por ejemplo, ”unes” → ún (comunes → común) tratando de encontrar si la palabra es uno de esos casos. Finalmente, se añaden cuatro reglas más:

Si la palabra acaba en `"esis","osis","isis"` se deja como está (esclerosis, hipótesis, electrólisis...). Si acaba en `"ces"`, se quitan las tres últimas y se añande `"z"` (luces → luz, perdices→ perdiz, felices → feliz...). Si la palabra acaba en `"es"` se elimina ésta terminación de la palabra (hospitales → hospital, caudales → caudal) y si acaba en `"s"`, se elimina ésta (datos → dato). El proceso en código es:

```
def singularize(word: String, pos:String ):String={
    val w=word.toLowerCase()
    //los gatos=> el gato
    if (pos=="DT"){
      if(List("la","las","los").contains(w)) return "el"
      if(List("una","unas","unos").contains(w)) return "un"
    }
    //hombres=>hombre
    if (w.endsWith("es") && (w.substring(0,w.length-2).endsWith("br") ||
     ↪  (w.substring(0,w.length-2).endsWith("i")) ||
      (w.substring(0,w.length-2).endsWith("j")) ||
       ↪  (w.substring(0,w.length-2).endsWith("t")) ||
       ↪  (w.substring(0,w.length-2).endsWith("zn"))))
      return w.substring(0,w.length-1)
    //gestiones=>gestión
    val endings=List(("anes", "án"),
      ("enes", "én"),
      ("eses", "és"),
      ("ines", "ín"),
      ("ones", "ón"),
      ("unes", "ún"))
    endings.foreach(e=> {if(w.endsWith(e._1)) return
     ↪  w.substring(0,w.length-4).concat(e._2)})
    //hipotesis=>hipotesis
    if (w.endsWith("esis") || w.endsWith("osis") || w.endsWith("isis")) return w
    //luces=>luz
    if(w.endsWith("ces")) return w.substring(0,w.length-3).concat("z")
```

```
    //hospitales=>hospital
    if (w.endsWith("es")) return w.substring(0,w.length-2)
    //gatos=>gato
    if (w.endsWith("s")) return w.substring(0,w.length-1)

   else return w
  }
```

7.3.2 Predicative

Éste método singulariza los adjetivos. Para ello se aplican cuatro casos: si la palabra acaba en `"os"`,`"as"`, se elimina la *s* final (histéricos → histérico, agobiados → agobiado). Si acaba en `"o"`, se deja igual, pero si acaba en `"a"`, ésta se cambia por una `"o"` (agobiada → agobiado). Por último, si la palabra acaba en `"es"`, tiene más de 4 letras y las letras en posición 3° y 4° por la cola son consonantes se elimina la última letra de la misma (horribles → horrible). En caso contrario, se eliminan las dos últimas (humorales → humoral, sociales → social). El proceso en código se decribe como sigue:

```
def predicative(word:String):String={
  var w=word.toLowerCase()
    //histéricos=>histérico
    if (w.endsWith("os") || w.endsWith("as")) w=w.substring(0,w.length-1)
    // histérico=>histérico
    if (w.endsWith("o")) w= w
    //histérica=>histérico
    if(w.endsWith("a")) w= w.substring(0,w.length-1).concat("o")
    //horribles=>horrible, humorales=>humoral
    if(w.endsWith("es")){
      if (w.length >= 4 && !(isVowel(normalize(w.charAt(w.length - 3)))) &&
       ↪  !(isVowel(normalize(w.charAt(w.length - 4))))) w= w.substring(0,
       ↪  w.length - 1)
      else w= w.substring(0,w.length-2)
    }
    return w
```

```
}
```

7.3.3 Lematización de verbos

Para lematizar una forma verbal, se comprueba si esta está en el diccionario interno de (forma verbal, lema) de la clase (mappedVerbs). Si se encuentra, se devuelve su lema asociado. Si no se encuentra, se inicia un proceso de lematización del verbo basado en reglas como sigue:

- Si el verbo acaba en `"ar","er","ir"` es un infinitivo y por tanto esa es la forma base
- Se definen en la clase una serie de inflexiones regulares típicas como:

```
private[this] val irregular_inflections=List(
    ("yéramos", "ir"   ), ( "cisteis", "cer"   ), ( "tuviera", "tener"), (
    ↪  "ndieron", "nder" ),
    ( "ndiendo", "nder" ), ("tándose", "tarse" ), ( "ndieran", "nder" ), (
    ↪  "ndieras", "nder" ),
    ("izaréis", "izar" ), ( "disteis", "der"   ), ( "irtiera", "ertir"), (
    ↪  "pusiera", "poner"),
    ( "endiste", "ender"), ( "laremos", "lar"   ), ("ndíamos", "nder" ),
    ↪  ("icaréis", "icar" )
    ....
```

  y si coincide alguna con el verbo a lematizar, se le aplica.

- Si el verbo a lematizar contiene `"zco", "zca", "zcá"` ésto se reemplaza por `"ce"`. Por ejemplo, conozco → conoce. Ésto se hace porque luego se tratan con una regla los verbos en 3° persona.
- Si el verbo contiene `"ldr" o "ndr"`, por ejemplo *valdrá, contendrá, compondrá, pondrá, mantendrá...* se mantiene la `"l" o "n"` respectivamente, y el `"dr"` en adelante se cambiará por `"er"`. Obteniendo *valer, contener, componer, poner, mantener....*
- Muchos verbos cuyo infinitivo acaba en *-ar* tienen inflexiones regulares. Por ejemplo, `cantabas, cantó, presentaremos, invitarás, ahorraríais` cuyas inflexiones son `"as","ó","aremos","arás","aríais"`. En este paso se define una lista de inflexiones regulares para verbos acabados en *-ar* y se aplica. Si coincide alguna, se elimina la inflexión y se añade el infinitivo.

- Parecido al paso anterior pero con una lista de inflexiones regulares para verbos que acaben en *-er*. Sin embargo, muchas de estas inflexiones también son comunes en verbos cuyo infinitivo acaba en *-ir*. Para detectar de que conjugación es, se comprueba si la palabra sin la inflexión es mayor que dos y si la penúltima letra de ésta es `"i"`. En ese caso, Se le añade el infinitivo `"-ir"`. En caso contrario, se le añade `"-er"`. Por ejemplo, *"correríais"* quedaría como *"corr-"* sin inflexión, y eso tiene longitud mayor que 2. Pero como su penúltima letra es una `"r"`, se lematizaría como `"correr"`.

- Se aplica un proceso esactamente igual al de los infinitivos acabados en `"-ar"` pero con inflexiones típicas de verbos con infinitivos acabados en `"-ir"`.

- Las palabras acabadas en `"a"` u `"o"` son consideradas de la primera conjugación, oor lo que se elimina dicha terminación y se les añade `-ar`.

- Igual que el anterior para verbos acabados en `"as", "an"`.

- Para verbos presente,3° persona del singular acabados en `"-e"` se comprueba si la palabra sin la inflexión es mayor que dos y si la antepenúltima letra de ésta es `"i"`, y en ese caso se elimina `"-e"` para añadirle `"-ir"`. En caso contrario, se eleminiaría `"-e"` para añadir `"-er"`.

- Se hace un proceso similar a los dos anteriores pero para las terminaciones `"-es", "-en"`.

- Por último se crea una lista de inflecciones verbales de tiempos en presente para 1° y 2° personas del plural, asociándole las conjugaciones a las que suelen pertenecer con más frecuencia:

```
val present_plural_inflection_o=List(
  ("amos", "áis"),
  ("emos", "éis"),
  ("imos", "ís")
)
val terminations=Array("ar","er","ir")
```

  y se itera sobre estas inflexiones, aplicando la conjunción correspondiente al verbo a lematizar si éste presenta alguna terminación que case con las de la lista de inflexiones.

A continuación se ilustra el proceso en código Scala:

```
def find_lemma(verb:String):String={
  // Spanish has 12,000+ verbs, ending in -ar (85%), -er (8%), -ir (7%).
  // Over 65% of -ar verbs (6500+) have a regular inflection.
  var v = verb.toLowerCase
  if(verb.endsWith("ar") || verb.endsWith("er") || verb.endsWith("ir")) return
   ↪  verb //verb is infinitive
  //set of rules for irregular inflections +10%

  irregular_inflections.foreach(a=> if(v.endsWith(a._1)) return
   ↪  v.substring(0,v.length-a._1.length).concat(a._2))
  //reconozco=>reconocer
  v=v.replace("zco","ce")
  //reconozcamos=>reconocer
  v=v.replace("zca","ce")
  //reconozcáis=>reconocer
  v=v.replace("zcá","ce")
  //valdrá => valer
if(v.contains("ldr")) return v.substring(0,v.indexOf("ldr")+1).concat("er")
  //contendrá=>contener
  if(v.contains("ndr")) return v.substring(0,v.indexOf("ndr")+1).concat("er")
//many verbs end in -ar and have a regular inflection
val regular_inflection_ar=List (
  "ando", "ado", "ad",                                  // participle
  "aré", "arás", "ará", "aremos", "aréis", "arán", // future
  "aría", "arías", "aríamos", "aríais", "arían",    // conditional
  "aba", "abas", "ábamos", "abais", "aban",         // past imperfective
    "é", "aste", "ó", "asteis", "aron",               // past perfective
    "ara", "aras", "áramos", "arais", "aran")       // past subjunctive

regular_inflection_ar.foreach(u=> if (v.endsWith(u)) return
 ↪  v.substring(0,v.length-u.length).concat("ar"))
//Many verbs end in -er and have a regular inflection
```

```
val regular_inflection_er=List(
    "iendo", "ido", "ed",                                  // participle
"eré", "erás", "erá", "eremos", "eréis", "erán", // future
"ería", "erías", "eríamos", "eríais", "erían",    // conditional
"ía", "ías", "íamos", "íais", "ían",              // past imperfective
  "í", "iste", "ió", "imos", "isteis", "ieron",        // past perfective
  "era", "eras", "éramos", "erais", "eran")       //past subjunctive

regular_inflection_er.foreach(u=>if(v.endsWith(u)){
  val difLength=v.length-u.length
  if(v.substring(0,difLength).length>2 &&
   ↪  v.substring(0,difLength).charAt(difLength-2)=='i')
return v.substring(0,difLength).concat("ir") else return
 ↪  v.substring(0,difLength).concat("er")} )

//Many verbs end in -ir and have a regular inflection
val regular_inflection_ir=List(
  "iré", "irás", "irá", "iremos", "iréis", "irán", //future
"iría", "irías", "iríamos", "iríais", "irían")  //past subjunctive

regular_inflection_ir.foreach(u=>if(v.endsWith(u)) return
 ↪  v.substring(0,v.length-u.length).concat("ir"))
//Present 1sg -o: yo hablo, como, vivo => hablar, comer, vivir.
if(v.endsWith("o")) return v.substring(0,v.length-1).concat("ar")
//Present 2sg, 3sg and 3pl: tú hablas.
if(v.endsWith("a")) return v.substring(0,v.length-1).concat("ar")

if(v.endsWith("as") || v.endsWith("an")){
  if (v.endsWith("as")) v =v.stripSuffix("s").dropRight(1)
  else if( v.endsWith("an")) v=v.stripSuffix("n").dropRight(1)
 return v.concat("ar")
}
// Present 2sg, 3sg and 3pl: tú comes, tú vives.
```

```
if(v.endsWith("e")){
  if(v.substring(0,v.length-1).length>2 &&
   ↪  v.substring(0,v.length-1).charAt(v.length-3)=='i') return
   ↪  v.substring(0,v.length-1).concat("ir")
  else return v.substring(0,v.length-1).concat("er")
}
if(v.endsWith("es") || v.endsWith("en")){
  if (v.endsWith("es")) v =v.stripSuffix("s").dropRight(1)
  else if( v.endsWith("en")) v=v.stripSuffix("n").dropRight(1)
  if(v.length>2 && v.charAt(v.length-2)=='i') return v.concat("ir")
else return v.concat("er")
}
// Present 1pl and 2pl: nosotros hablamos.

val present_plural_inflection_o=List(
  ("amos", "áis"),
  ("emos", "éis"),
  ("imos", "ís")
)
val terminations=Array("ar","er","ir")
//terminations{1}
present_plural_inflection_o.foreach(u=> {
  if (v.endsWith(u._1)) return v.substring(0, v.length -
   ↪  u._1.length).concat(terminations {
    present_plural_inflection_o.indexOf(u)
  })

  if (v.endsWith(u._2)) return v.substring(0, v.length -
   ↪  u._2.length).concat(terminations {
    present_plural_inflection_o.indexOf(u)
  })

})
```

```
  return v
}
```

7.4 Parser

Por último, estos tres procesos se integran en una clase Parser, donde se define un método parse con la siguiente cabecera:

```
def parse(text: String, tokenize: Boolean, tags: Boolean, lemmatize: Boolean
 ↪  ,mapCall:(String,String)=>(String,String)): String
```

Que recibe como parámetros el texto a parsear, tres booleanos indicando si se desea hacer tokenización, pos etiquetado y lematización y una función para mapear las etiquetas al tagset deseado.

7.5 Pruebas

Las pruebas realizadas son todos los test unitarios incluídos en la clase AppTest. Para los test se usa la librería ScalaSuite y el estilo de test FunSuite. En los test se emplean aserciones generales para hacer las comprobaciones(assert). En la clase AppTest se incluyen toda la declaración de variables necesarias para los test y test implementados. Se implementan test sólo para los métodos más importantes, implementando un test por cada funcionalidad relevante. Los test más destacados son:

```
test("test find tags"){
     var wikicorpus=List[List[String]]()
     var i=0
     var n=0

     // Assert the accuracy of the Spanish tagger.

     val lexiconSpan-
      ↪  ish=Lexicon.read("../Spanish_Lematizer/data/es-lexicon.txt","utf-8",";;;")
     val morphologySpan-
      ↪  ish=morphology.read("../Spanish_Lematizer/data/es-morphology.txt","utf-8",";;;")
```

```
    val contextSpan-
     ↪  ish=context.read("../Spanish_Lematizer/data/es-context.txt","utf-8",";;;")


     ↪  scala.io.Source.fromFile("../Spanish_Lematizer/corpus/tagged-es-wikicorpus.txt").getLines()
     ↪  => wikicorpus ::= line.split(" ").toList)
    wikicorpus=wikicorpus.reverse
    wikicorpus.foreach(sentenceList=>{
      var s1= sentenceList.map(f=>f.split("/"))
      var s1ToTag=List[String]()
      s1.foreach(f=> s1ToTag::=f(0))
      s1ToTag=s1ToTag.reverse
      val tagged=tagger.find_tags(s1ToTag, Lexicon, morphology,context,
       ↪  List("NCS","NP","Z"), null)

      var j=0
      s1.foreach(s=>{
        if(s(1)==tagged(j)._2) i=i+1
        n=n+1
        j=j+1
      })
    })
    assert(i.toFloat/n>0.91)
    print("accuracy tagger:" + i.toFloat/n)
  }
}
```

Donde se calcula el porcentaje de acierto del método de pos etiquetado implementado sobre el corpus etiquetado Wikicorpus http://www.cs.upc.edu/~nlp/wikicorpus/.

```
test("singularize") {
    var wordforms = List[List[String]]()
    var testDict = mutable.Map.empty[String, List[String]]
```

```
    //read file

     ↪  scala.io.Source.fromFile("../Spanish_Lematizer/test/data/wordforms-es-davies.csv").getLines
     ↪  => wordforms ::= line.split(" ").toList)
    wordforms.reverse.foreach(l => {
      val splitted = l(0).split(",").map(u =>
       ↪  u.tail.stripPrefix("\"").stripSuffix("\""))
      val w = splitted(0)
      val lemma = splitted(1)
      val tag = splitted(2)
      val f = splitted(3)

      if (tag == "n") {
        testDict += (lemma -> (w +: testDict.getOrElse(lemma, List.empty)))
      }
    })

    var i = 0
    var n = 0
    testDict.foreach(f => {
      if (lemmatizer.singularize((f._2.sortWith(_.length < _.length)).head,
       ↪  "NN") == f._1) i = i + 1
      n = n + 1
    })
    assert(i.toFloat / n > 0.93)
    print("singularize accuracy:" + i.toFloat/n )
  }
```

Donde se calcula el porcentaje de acierto para el método singularize sobre 3000 palabras etiquetadas con su lema obtenidas del corpus wordforms-es-davies.csv http://www.wordfrequency.info/files/spanish/spanish_lemmas20k.txt.

```
test("predicative") {
    var wordforms = List[List[String]]()
```

```
    var testDict = mutable.Map.empty[String, List[String]]
    //read file

     ↪  scala.io.Source.fromFile("../Spanish_Lematizer/test/data/wordforms-es-davies.csv").getLines
     ↪  => wordforms ::= line.split(" ").toList)
    wordforms.reverse.foreach(l => {
      val splitted = l(0).split(",").map(u =>
       ↪  u.tail.stripPrefix("\"").stripSuffix("\""))
      val w = splitted(0)
      val lemma = splitted(1)
      val tag = splitted(2)
      val f = splitted(3)

      if (tag == "j") {
        testDict += (lemma -> (w +: testDict.getOrElse(lemma, List.empty)))
      }
    })
    var i = 0
    var n = 0
    testDict.foreach(f => {
      if (lemmatizer.predicative(f._2.sortWith(_.length < _.length).head) ==
       ↪  f._1) i = i + 1
      n = n + 1
    })
    assert(i.toFloat / n > 0.92)
    print("predicative accuracy:" + i.toFloat/n  )
  }
```

Donde se testea el método predicative, obteniendo su porcentaje de acierto sobre el mismo corpus.

```
test("find lemma") {
    var i = 0
    var n = 0
```

```
    var faults = List[(String, String)]()
    lemmatizer.getVerbsDict.foreach(u => {
      if (lemmatizer.find_lemma(u._1) == u._2) i = i + 1 else faults ::= u
      n = n + 1
    })

     assert(i.toFloat/n > 0.80)
     print("accuracy lematization of verbs: " + i.toFloat/n )
}
```

Por último, el test donde se comprueba que la implementación de lematización de verbos obtiene la presición de etiquetado deseada. Para ello se testea sobre las 23435 entradas de verbos del diccionario creado a partir de es-verbs.txt. Los resultados obtenidos se discuten en el capítulo siguiente.

8

Marco experimental y resultados

8.1 Marco experimental

A continuación se dan unas pautas para probar las herramientas implementadas si se desea. Se ha intentado simplificar el procedimiento, por lo que se incluye un fichero con un object ScalaApp que implementa un método main en el que se incluyen las rutas de todos los paths de los ficheros que se usan ya especificados. Para probar la funcionalidad, hay que ejecutar ScalaApp.scala especificando las siguientes opciones:

- Texto que se desea parsear, sin poner comillas, pues ya las pone el programa.
- Opciones que se desean aplicar separadas por comas. Es decir, si se quiere hacer hasta la lematización, se especificaría true,true,true
- Seleccionar la notación que queremos para las etiquetas. Hay que introducir un número entre 1 y 3, donde 1 representa el tagset parole, 2 representa el tagset penntreebank y 3 representa el tagset universal.

```
/opt/jdk/bin/java ...
Introduce el texto que desees parsear:llegan ya las navidades con turrón y mazapanes
Introduce las opciones que deseas aplicar, separadas por comas:true,true,true
¿Qué tagset deseas usar? 1:parole 2:penn treebank 3:universal2
Texto parseado: llegan/VB/llegar
ya/RB/ya
las/DT/el
navidades/NNS/navidad
con/IN/con
turrón/NN/turrón
y/CC/y
mazapanes/NNS/mazapán
Process finished with exit code 0
```

Figure 8.1: Ejemplo de ejecución desde el main

donde cada palabra aparece en su forma original introducida, junto con su etiqueta y su lema, separados por /.

8.2 Resultados obtenidos

A continuación se representan los resultados finales obtenidos por los algoritmos implementados en este trabajo frente a los resultados obtenidos por esos métodos en trabajo tomado como referencia. No se incluye comparación para el tokenizador dado que no hay costumbre de comparar resultados de tokenizadores, pues suelen funcionar bastante bien en lenguajes espaciados entre palabras, como el nuestro.

Resultados de los algoritmos

	Pattern.es	Este trabajo
singularize	0.9390681	0.9390681
predicative	0.9323671	0.9468599
find lemma	0.8082355	0.8082782
tagger	0.9279443	0.9199328

Por lo general los resultados son muy similares, obteniendo exactamente el mismo resultado para el algoritmo de singularización y prácticamente el mismo para el algoritmo de lematización de verbos y etiquetado,siendo el de este trabajo algo muy levemente superior.
Quizá la mayor diferencia esté en el algoritmo predicativo, aunque tampoco es una diferencia alarmante, y en este caso, es a favor de nuestro trabajo. Ésta diferencia se debe casi con toda seguridad a que se ha decidido etiquetar inicialmente las palabras desconocidas que no sean nombre propios ni números como `"NCS"` (nombre común en singular) en lugar de `"NN"` como hace Pattern.es.

9

Conclusiones y vías futuras

9.1 Conclusiones

En este trabajo se implementan e integran tres algoritmos en el lenguaje Scala que resuelven las tres primeras fases de una herramienta para procesamiento del lenguaje natural. En concreto, se desarrolla un algoritmo de tokenización, otro de Pos etiquetado y otro de lematización, probando que no hace falta utilizar el algoritmo más complejo existente para ser capaz de dar unos resultados decentes.
En todos los algoritmos, especialmente en el lematizador, se hace uso de muchas reglas conocidas existentes en nuestro lenguaje para implementar el algoritmo. Aunque es imposible cubrir todos los posibles casos con reglas, se pueden obtener resultados atractivos sólo empleando algunas de ellas.

9.2 Vías futuras

Como vías futuras se podría implementar la extracción de los ficheros de léxico, morfología y contexto que aplica el Pos tagger, en lugar de tomarlos ya construidos. Concretamente para el contexto se podría implementar un algoritmo de Brill en Scala que se incluya en la herramienta, y algún otro algoritmo más potente que hiciera posible una comparativa.

Otras posibles mejoras son:
añadir manejo de emoticonos en el tokenizador, intentar hacer todo el código inmutable, dado que Scala anima a ello pero es tan complicado visualizarlo al principio que, de haberlo hecho en este trabajo, no nos habría dado tiempo a cumplir los plazos, y seguir ampliando el trabajo con el resto de módulos de PLN hasta tener un software completo para análisis de sentimientos para español. Una razón por la que se elige Scala.

Declaración

Yo, Mª Cristina Heredia Gómez, alumno de l Grado en Ingeniería informática de la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación 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.

Fdo: Mª Cristina Heredia Gómez

Granada a 12 de Diciembre de 2016 .

Referencias

[1] Baldridge, J. (2005). The opennlp project. *URL: http://opennlp. apache. org/index. html,(accessed 2 February 2012)*.

[2] Berger, A. L., Pietra, V. J. D., & Pietra, S. A. D. (1996). A maximum entropy approach to natural language processing. *Computational linguistics*, 22(1), 39–71.

[3] Bird, S. (2006). Nltk: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive presentation sessions* (pp. 69–72).: Association for Computational Linguistics.

[4] Bird, S., Klein, E., & Loper, E. (2009). *Natural language processing with Python*. " O'Reilly Media, Inc.".

[5] Bollen, J., Mao, H., & Pepe, A. (2011). Modeling public mood and emotion: Twitter sentiment and socio-economic phenomena. *ICWSM*, 11, 450–453.

[6] Brill, E. (1992). A simple rule-based part of speech tagger. In *Proceedings of the workshop on Speech and Natural Language* (pp. 112–116).: Association for Computational Linguistics.

[7] Chang, P.-C., Galley, M., & Manning, C. D. (2008). Optimizing chinese word segmentation for machine translation performance. In *Proceedings of the third workshop on statistical machine translation* (pp. 224–232).: Association for Computational Linguistics.

[8] Culotta, A. & Sorensen, J. (2004). Dependency tree kernels for relation extraction. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics* (pp. 423).: Association for Computational Linguistics.

[9] Ding, X., Liu, B., & Yu, P. S. (2008). A holistic lexicon-based approach to opinion mining. In *Proceedings of the 2008 international conference on web search and data mining* (pp. 231–240).: ACM.

[10] Elhadad, N., Gravano, L., Hsu, D., Balter, S., Reddy, V., & Waechter, H. (2014). Information extraction from social media for public health. In *KDD at Bloomberg Workshop, Data Frameworks Track (KDD 2014)*.

[11] Etzioni, O., Fader, A., Christensen, J., Soderland, S., & Mausam, M. (2011). Open information extraction: The second generation. In *IJCAI*, volume 11 (pp. 3–10).

[12] Giesbrecht, E. & Evert, S. (2009). Is part-of-speech tagging a solved task? an evaluation of pos taggers for the german web as corpus. In *Proceedings of the fifth Web as Corpus workshop* (pp. 27–35).

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

[14] Gonçalves, P., Benevenuto, F., & Cha, M. (2013). Panas-t: A pychometric scale for measuring sentiments on twitter. *arXiv preprint arXiv:1308.1857*.

[15] Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., et al. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6), 82–97.

[16] Hirschberg, J. & Manning, C. D. (2015). Advances in natural language processing. *Science*, 349(6245), 261–266.

[17] Hu, M. & Liu, B. (2004). Mining and summarizing customer reviews. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 168–177).: ACM.

[18] Jiang, L., Yu, M., Zhou, M., Liu, X., & Zhao, T. (2011). Target-dependent twitter sentiment classification. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1* (pp. 151–160).: Association for Computational Linguistics.

[19] Joachims, T. (1998). Text categorization with support vector machines: Learning with many relevant features. In *European conference on machine learning* (pp. 137–142).: Springer.

[20] Kiss, T. & Strunk, J. (2006). Unsupervised multilingual sentence boundary detection. *Computational Linguistics*, 32(4), 485–525.

[21] Lewis, D. D. (1998). Naive (bayes) at forty: The independence assumption in information retrieval. In *European conference on machine learning* (pp. 4–15).: Springer.

[22] Liu, B. (2015). *Sentiment Analysis: Mining Opinions, Sentiments, and Emotions*. Cambridge University Press.

[23] Luong, M.-T., Sutskever, I., Le, Q. V., Vinyals, O., & Zaremba, W. (2014). Addressing the rare word problem in neural machine translation. *arXiv preprint arXiv:1410.8206*.

[24] Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J. R., Bethard, S., & McClosky, D. (2014). The stanford corenlp natural language processing toolkit. In *ACL (System Demonstrations)* (pp. 55–60).

[25] Marcus, M. P., Marcinkiewicz, M. A., & Santorini, B. (1993). Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2), 313–330.

[26] Màrquez, L., Villarejo, L., Martí, M., & Taulé, M. (2007). Semeval-2007 task 09: Multilevel semantic annotation of catalan and spanish. In *Proceedings of the 4th International Workshop on Semantic Evaluations* (pp. 42–47).: Association for Computational Linguistics.

[27] Nadkarni, P. M., Ohno-Machado, L., & Chapman, W. W. (2011). Natural language processing: an introduction. *Journal of the American Medical Informatics Association*, 18(5), 544–551.

[28] Niu, F., Zhang, C., Ré, C., & Shavlik, J. W. (2012). Deepdive: Web-scale knowledge-base construction using statistical learning and inference. *VLDS*, 12, 25–28.

[29] Nivre, J., Hall, J., & Nilsson, J. (2006). Maltparser: A data-driven parser-generator for dependency parsing. In *Proceedings of LREC*, volume 6 (pp. 2216–2219).

[30] Padró, L. & Stanilovsky, E. (2012). Freeling 3.0: Towards wider multilinguality. In *LREC2012*.

[31] Pang, B., Lee, L., & Vaithyanathan, S. (2002). Thumbs up?: sentiment classification using machine learning techniques. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10* (pp. 79–86).: Association for Computational Linguistics.

[32] Raychaudhuri, N. (2013). *Scala in Action: Covers Scala 2.10*. Manning Publications.

[33] Reese, S., Boleda Torrent, G., Cuadros Oller, M., Padró, L., & Rigau Claramunt, G. (2010). Word-sense disambiguated multilingual wikipedia corpus. In *7th International Conference on Language Resources and Evaluation*.

[34] Ribeiro, F. N., Araújo, M., Gonçalves, P., Gonçalves, M. A., & Benevenuto, F. (2016). Sentibench-a benchmark comparison of state-of-the-practice sentiment analysis methods. *EPJ Data Science*, 5(1), 1–29.

[35] Riloff, E. & Wiebe, J. (2003). Learning extraction patterns for subjective expressions. In *Proceedings of the 2003 conference on Empirical methods in natural language processing* (pp. 105–112).: Association for Computational Linguistics.

[36] Rodrigues, M. J. F. & da Silva Teixeira, A. J. (2015). *Advanced Applications of Natural Language Processing for Performing Information Extraction (SpringerBriefs in Electrical and Computer Engineering)*. Springer.

[37] Schmid, H. (1995). Treetagger| a language independent part-of-speech tagger. *Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart*, 43, 28.

[38] Smedt, T. D. & Daelemans, W. (2012). Pattern for python. *Journal of Machine Learning Research*, 13(Jun), 2063–2067.

[39] Tausczik, Y. R. & Pennebaker, J. W. (2010). The psychological meaning of words: Liwc and computerized text analysis methods. *Journal of language and social psychology*, 29(1), 24–54.

[40] Turney, P. D. (2002). Thumbs up or thumbs down?: semantic orientation applied to unsupervised classification of reviews. In *Proceedings of the 40th annual meeting on association for computational linguistics* (pp. 417–424).: Association for Computational Linguistics.

[41] Wampler, D. & Payne, A. (2014). *Programming Scala: Scalability = Functional Programming + Objects*. O'Reilly Media.

[42] Wang, H. (2015). *Sentiment-aligned Topic Models for Product Aspect Rating Prediction*. PhD thesis, Applied Sciences: School of Computing Science.

[43] Westerski, A. (2007). Sentiment analysis: Introduction and the state of the art overview. *Universidad Politecnica de Madrid, Spain*, (pp. 211–218).

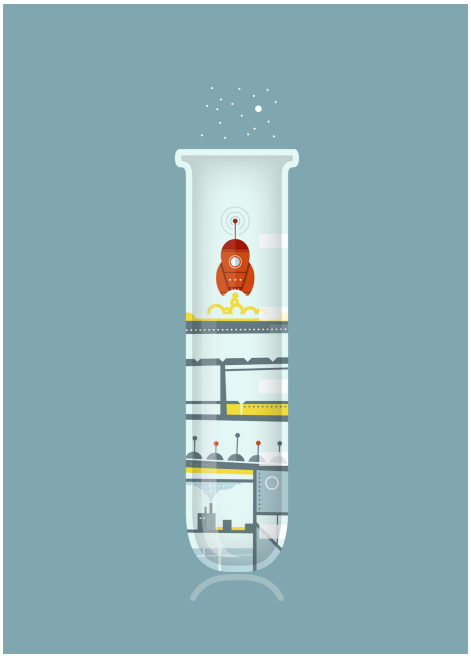
[44] Whissell, C. (1989). The dictionary of affect in language. *Emotion: Theory, research, and experience*, 4(113-131), 94.

[45] Wiebe, J., Wilson, T., & Cardie, C. (2005). Annotating expressions of opinions and emotions in language. *Language resources and evaluation*, 39(2-3), 165–210.

[46] Wong, D. F., Chao, L. S., & Zeng, X. (2014). isentenizer-: Multilingual sentence boundary detection model. *The Scientific World Journal*, 2014.

[47] Young, S., Gašić, M., Thomson, B., & Williams, J. D. (2013). Pomdp-based statistical spoken dialog systems: A review. *Proceedings of the IEEE*, 101(5), 1160–1179.

[48] Yu, H. & Hatzivassiloglou, V. (2003). Towards answering opinion questions: Separating facts from opinions and identifying the polarity of opinion sentences. In *Proceedings of the 2003 conference on Empirical methods in natural language processing* (pp. 129–136).: Association for Computational Linguistics.



This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Egenolff-Berner Garamond, a revival of Claude Garamont's humanist typeface. The above illustration, *Science Experiment 02*, was created by Ben Schlitter and released under CC BY-NC-ND 3.0. A template that can be used to format a PhD dissertation with this look & feel has been released under the permissive AGPL license, and can be found online at github.com/suchow/Dissertate or from its lead author, Jordan Suchow, at suchow@post.harvard.edu.