Integración de Técnicas de Procesamiento de Lenguaje Natural a través de Servicios Web

Tesis entregada para el grado de

Ingeniero en Sistemas

en la Facultad de Ciencias Exactas

Por

Facundo Matías Ramos

Juan Ignacio Velez

Bajo la supervisión de

Director: Dr. Alejandro Rago

Co-director: Dr. Andres Díaz Pace

Universidad Nacional del Centro de la Provincia de Buenos Aires

Tandil, Argentina

Mayo 2016

Agradecimientos

A nuestros padres que nos alentaron y apoyaron durante todos estos años de estudio. A nuestras hermanas Monica, Natalia, y Victoria, que nos acompañaron en esta etapa universitaria.

También queremos agradecer a nuestros directores de tesis, Alejandro y Andres, que nos ayudaron a llevar a cabo este trabajo con mucha dedicación.

A nuestros amigos que estuvieron a lo largo de este recorrido. Gracias por los momentos compartidos.

Por último, queremos darle las gracias a nuestras novias, Pía y Lucila, que siempre estuvieron en los buenos y malos momentos, ayudando a cumplir este objetivo.

A todos ellos, muchísimas gracias.

Índice de contenido

Capítulo 1 - Introducción

El procesamiento de lenguaje natural (NLP del idioma inglés *Natural Language Processing*) es una disciplina que trata la interacción entre la computadora y el lenguaje humano. El NLP está compuesto por un conjunto de técnicas computacionales que permiten el análisis y la representación de los textos. Este área de investigación involucra diferentes disciplinas de las Ciencias de la Computación, como la Inteligencia Artificial y la Lingüística, y a partir de conceptos referidos a cada una de ellas se logra que una máquina pueda intentar comprender lenguaje natural e identificar su significado.

El NLP se utiliza en una variedad de tareas y aplicaciones como pueden ser la recuperación y extracción de información, minería de datos, traducción automática, sistemas de búsquedas de respuestas, generación de resúmenes automáticos, entre otras [Sateli2012]. Los sistemas que utilizan técnicas NLP para analizar documentos textuales, por lo general, son desarrollados en diversos lenguajes de programación y preparados para funcionar en una determinada plataforma de software. Esto plantea una dificultad para los desarrolladores, debido a que cada sistema debe integrar diferentes técnicas de NLP independientes entre sí, siendo en la mayoría de los casos, tecnológicamente incompatibles. Además de la existencia de una gran variedad de técnicas NLP que permiten extraer información específica de un texto, se pueden encontrar diferentes implementaciones para cada técnica dependiendo del grupo de expertos que la desarrolle. El área de NLP se encuentra en constante crecimiento, ya sea desarrollando nuevos módulos o actualizando técnicas existentes. Esto genera una necesidad de mantener actualizado los módulos NLP con el fin de obtener los mejores resultados posibles.

Para cada uno de los sistemas que realizan procesamiento NLP, se utiliza un conjunto de técnicas personalizado dependiendo de la aplicación a desarrollar. Por ejemplo, una aplicación para buscar información dentro de un documento sólo necesita obtener las palabras relevantes para generar un diccionario e indexar dicha información. Otro ejemplo podría ser una aplicación que identifica la temática de un texto determinado, la cual requiere de un conjunto de técnicas de NLP especializadas para llevar a cabo su objetivo. El problema aquí reside en el esfuerzo al desarrollar más de una aplicación que utilice un subconjunto de las mismas técnicas de NLP, debido a que cada desarrollo implica una integración de técnicas ad-hoc que en general no es reutilizable (en otras aplicaciones).

En la actualidad, se pueden encontrar herramientas que realizan procesamiento NLP a través de servicios Web [Dale2015]. Por ejemplo, Aylien, LanguageTools, IBM Watson, entre otras. Cada una de estas herramientas permite enviar texto y obtener el análisis del mismo aplicando técnicas NLP. La invocación al procesamiento de texto se realiza de manera simple a través de una invocación Web. En su mayoría, estas herramientas brindan un servicio gratuito con acceso a un amplio conjunto de técnicas de NLP.

A pesar de las ventajas que presentan las herramientas de procesamiento NLP existentes, se pueden destacar algunas limitaciones. El primer problema es la falta de configurabilidad en las herramientas de

servicios NLP. Las aplicaciones que hacen uso de estas herramientas requieren diferentes tipos de análisis dependiendo de la funcionalidad de cada una [Marcos2016]. Es por eso, que se requiere un procesamiento personalizado para cumplir con todas las necesidades. Mayormente, estas herramientas no permiten la ejecución de una secuencia de módulos que realicen un análisis compuesto por más de una técnica de NLP. Los módulos provisto por estas herramientas deben ser utilizados tal y como están definidos.

Otro problema, es la dificultad a la hora de componer módulos NLP. Algunas de las herramientas permiten agregar módulos nuevos pero no es posible componerlos a partir de módulos pre-existentes. Esto limita la cantidad de módulos y la información del texto que se puede extraer, teniendo en cuenta que muchas técnicas de NLP pueden realizar análisis complejos en base a los datos extraídos por otros analizadores. Además, cada uno de los módulos provistos por las diferentes herramientas poseen escasas variables parametrizables. La parametrización de módulos permite contar con diversas variantes de un mismo módulo.

Teniendo en cuenta estas limitaciones, se evidencia la necesidad de una herramienta que se ajuste a las necesidades generales de la mayoría de las aplicaciones que utilizan procesamiento NLP. En primer lugar, la solución debe ser flexible para poder incorporar, configurar y ejecutar técnicas de NLP desarrolladas por terceros. En segundo lugar, la solución debe ser una solución configurable que permite realizar análisis personalizados. Y en tercer lugar, la solución debe poder exponer las técnicas NLP para que puedan ser consumidas por diversas aplicaciones independientemente de la plataforma en la que se desarrollen. En el presente trabajo se propuso abordar una solución que simplifique la integración de módulos NLP en sistemas externos.

1.1. Enfoque

El objetivo principal de este trabajo es presentar una herramienta, llamada TeXTracT, que permite realizar procesamiento NLP desde cualquier aplicación y plataforma, personalizando el proceso a realizar de acuerdo a las necesidades particulares de cada aplicación que quiera hacer uso de ella. Esta herramienta permite la incorporación, configuración y ejecución de técnicas de NLP desarrolladas por terceros, como así también soporta la composición de módulos de NLP para realizar análisis de texto personalizados de acuerdo a las necesidades de cada aplicación.

La Figura 1.1 ilustra un diagrama conceptual de TeXTracT. Cada uno de los componentes cumplen una función específica para cumplir los objetivos planteados. La interacción de las aplicaciones con la herramienta se realiza mediante el uso de servicios Web, de forma tal de permitir el acceso desde cualquier lenguaje y plataforma de desarrollo [Mulligan2009]. Asimismo, este componente coordina búsqueda, la composición, configuración y ejecución de los módulos NLP. En primer lugar, el componente *Broker* actúa como intermediario entre las aplicaciones que requieren procesar texto y los

módulos de NLP, con el fin de independizar los servicios NLP de las aplicaciones que los invocan. Por otro lado, el componente *Adapter* permite agregar módulos NLP de manera sencilla y manipularlos de manera uniforme (independientemente de la técnica que describa) sin modificar ninguna parte del sistema.

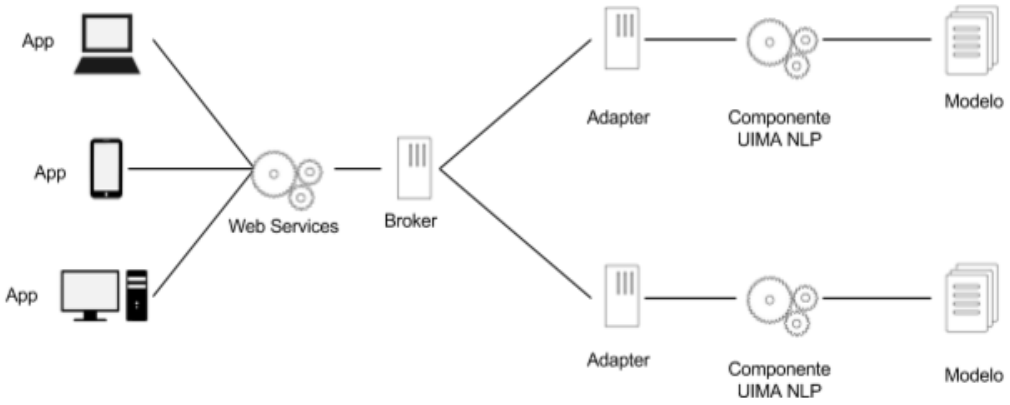


Figura 1.1. Esquema Conceptual de TeXTracT.

Para llevar a cabo este desarrollo, se utilizaron diferentes tecnologías que simplificaron su implementación. Una vez definida la estructura de servicios Web, se optó por utilizar de una biblioteca de servicios REST para Java; que permite exponer métodos de una clase determinada como servicios. En cuanto a la estructuración y análisis de texto, se hizo uso del framework UIMA, el cual define una infraestructura que permite el análisis de información extrayendo datos específicos acerca de un texto [Ferrucci2004].

En la Figura 1.1 se puede ver que el punto de acceso a TeXTracT es a través del componente *Broker*. Este componente es el encargado de recibir la solicitud de una aplicación y obtener los datos correspondientes al análisis, como son el texto y la secuencia de módulos a ejecutar. El texto de entrada es interpretado por el *Broker* dependiendo del formato del mismo. Cuando se identifica el pipeline de procesamiento, el texto es decodificado por el adaptador del módulo en cuestión para que este realice el proceso. Una vez realizado el procesamiento, el *Adapter* codifica el resultado para que pueda ser interpretado por el resto de los componentes del pipeline. Cada uno de los módulos de análisis especificados en la invocación pueden contener parámetros que modifiquen el funcionamiento del mismo. Estos datos son extraídos por el *Broker*, realizando la instanciación correspondiente con los parámetros indicados. Los módulos indicados en la solicitud Web son invocados por el *Broker* en el orden correspondiente, permitiendo así realizar una composición de analizadores de texto. Además, este componente es el encargado de realizar la búsqueda en ejecución de los módulos disponibles. La aplicación que desea realizar el proceso puede solicitar el listado de analizadores disponibles para hacer uso de los que crea necesarios.

Con el fin de evaluar la solución desarrollada, se llevaron a cabo dos casos de estudio. El primero consiste de la implementación de una aplicación nueva con NLP. El objetivo del primer caso de estudio es mostrar

que haciendo uso de TeXTracT, se pueden crear aplicaciones de manera simple y rápida, soportando la evolución de dichos sistemas en el tiempo. La aplicación en cuestión es un corrector online de textos en Inglés. En este caso, se realizaron observaciones de los tiempos que llevó la codificación de la app en comparación con una estimación realizada sobre un desarrollo tradicional cuyo procesamiento NLP es ad-hoc. El segundo caso de estudio, en cambio, tiene el objetivo de establecer la factibilidad y los costos asociados al migrar una aplicación existente a los servicios de NLP en la Web. La aplicación migrada, denominada REAssistant, permite identificar crosscutting concerns a partir de un conjunto de especificaciones de caso de uso. En los experimentos, se tomaron métricas que comparan cantidad de clases, tamaño de la aplicación, consumo de procesador y memoria, etc.. Estas métricas se tomaron entre REAssistant con el procesamiento NLP de manera ad-hoc y REAssistant utilizando TeXTracT.

Los resultados obtenidos de ambas evaluaciones fueron positivos. Para el primer caso, si bien se obtuvo una mejora en cuanto a tiempo de desarrollo, el resultado final de la aplicación fue muy positivo. Mientras un desarrollo de procesamiento NLP ad-hoc requiere el estudio de cada uno de los módulos de análisis y la adaptación de los mismos al sistema que se está desarrollando, cuando se utiliza TeXTracT solo se debe seleccionar un conjunto de módulos de un listado y crear una URL que invoque el procesamiento necesario, evitando problemas de compatibilidad y el estudio de algoritmos NLP.

En el segundo caso de estudio, las mejoras en optimización de recursos físicos de la app migrada fueron considerables, debido a la carga de proceso y utilización de memoria que implica la ejecución de módulos NLP de forma local. Asimismo, también se observaron importantes reducciones en la cantidad de líneas de código, la eliminación de bibliotecas Java dentro del proyecto, dando como resultado una aplicación que necesita menos recursos para ser utilizada.

1.2. Organización del Contenido

El contenido del resto del trabajo se encuentra organizado de la siguiente manera. El Capítulo 2 explica el marco teórico sobre el cual se basa el trabajo final, detallando cada uno de los conceptos que son utilizados durante el desarrollo de los demás capítulos. Inicialmente, se introducen los conceptos de Procesamiento de Lenguaje Natural, y se describen un conjunto de técnicas de procesamiento NLP populares en el desarrollo de apps. Luego, se desarrollan los conceptos de servicios Web, centrándose en la arquitectura y sus diferentes variantes SOAP y REST.

El Capítulo 3 describe los trabajos relacionados. Estos trabajos se organizan en dos líneas de investigación. En primer lugar, se analizan APIs que proveen servicios Web de procesamiento de lenguaje natural, con el fin de observar las distintas alternativas para desarrollar una app que requiere NLP. En segundo lugar, se analizan un conjunto de aplicaciones que hacen uso de este tipo de servicios, y se explora el funcionamiento específico de cada uno de ellos.

El Capítulo 4 presenta el enfoque propuesto y la herramienta desarrollada, explicando cada uno de los componentes principales que permiten la invocación de los servicios Web y el procesamiento del texto. Luego, se detallan las tecnologías utilizadas para el desarrollo de TeXTracT, y se deja en evidencia cuales son los beneficios de haberlas utilizado. Por último, se explica en detalle los componentes más relevantes de TeXTracT.

El Capítulo 5 reporta los resultados experimentales de implementar una app nueva que hace uso de TeXTracT, demostrando la facilidad para crear aplicaciones de manera rápida y sencilla. La app que se desarrolló es un sitio Web para corregir textos escritos en Inglés. Además, se detalla el desarrollo de patrones identificadores de errores gramaticales y su posterior integración a una secuencia de procesamiento. También se describe el desarrollo de la herramienta y las tecnologías utilizadas para lograr el producto final. Por último, se analizan métricas entre el desarrollo de una aplicación ad-hoc y una aplicación que utiliza la infraestructura de NLP de TeXTracT.

En el Capítulo 6 se reportan los resultados experimentales al migrar una aplicación existente a los servicios de NLP en la Web. Primero se describe la arquitectura original de REAssistant y las tareas efectuadas para migrar dicha implementación a una solución que haga uso de TeXTracT. Por último, se analizan métricas entre el funcionamiento que realiza REAssistant al llevar a cabo procesamiento NLP ad-hoc y la versión que hace uso de TeXTracT. Las métricas planteadas son en base a la utilización de los recursos físicos, como así también características estáticas de la aplicación.

Finalmente, el Capítulo 7 presenta las conclusiones de esta tesis, discutiendo las ventajas y limitaciones de la herramienta desarrollada. Asimismo, en este capítulo se enumeran diferentes líneas de trabajo futuro que se desprenden de este trabajo final.

Capítulo 2 - Marco teórico

En este capítulo se presentarán los conceptos que se utilizan en la realización de este trabajo. Primero, se introducirán los conceptos de Procesamiento de Lenguaje Natural, haciendo foco en cada una de las particularidades que esta tarea conlleva. Se presentará un conjunto de técnicas de procesamiento, explicando el funcionamiento en el funcionamiento específico de cada una los resultados que producen. En segundo lugar, se desarrollará el concepto de servicios Web, centrándose en la arquitectura y sus diferentes variantes. Por último, se presentarán diferentes usos y aplicaciones en los que se pueden encontrar procesamiento de texto vía servicios Web con el fin de observar la amplitud del campo de aplicación que posee este área de estudio en la actualidad.

2.1. Natural Language Processing (NLP)

Cuando se habla de lenguaje natural, se hace referencia al lenguaje oral y escrito mediante el cual se pretende comunicar algo. Tanto la comprensión como la composición de mensajes en lenguaje humano es una tarea compleja, ya que requiere millones de conexiones neuronales y procesos corporales de captación. Una de las características principales del lenguaje natural es que surge de modo espontáneo entre la gente, mientras que, por ejemplo, los lenguajes propios utilizados por computadoras para comunicarse entre sí son definidos por un protocolo matemático riguroso.

En el campo de la informática se han realizado diferentes investigaciones para que las computadoras puedan interpretar el lenguaje natural. A pesar de que ciertas características del lenguaje natural tienen reglas estrictas que facilitan su análisis computarizado (ej., gramática), algunas otras facetas son más complejas (ej., significados) [Gobinda2005]. Dentro de las variaciones que presentan los lenguajes naturales, existen para cada uno de ellos gramáticas y diccionarios que se encargan de acotarlos para poder así eliminar ambigüedades. Esto ayuda también a que personas que no tengan como lengua nativa un determinado lenguaje, lo puedan aprender y entender.

El Procesamiento del Lenguaje Natural (PLN) es un área de investigación que explora las posibilidades de una computadora para comprender y manipular el lenguaje natural escrito u oral, de manera que se pueda hacer uso del mismo [Manning1999, Kumar2011]. Esto se realiza con el objetivo de desarrollar técnicas y herramientas que permitan la implementación de sistemas capaces de interpretar y utilizar el lenguaje natural para desempeñar las tareas deseadas, como por ejemplo, un clasificador de noticias, o un identificador de correo indeseado.

Existe una clasificación de los diversos tipos de procesamientos que se pueden realizar sobre un texto. Esta se puede ver más claramente como una estructura de niveles, permitiendo así diferenciar los aspectos que trata cada nivel de proceso y la complejidad que posee cada uno.

2.1.1. Niveles de Procesamiento

Las técnicas de NLP pueden ser organizadas en diferentes niveles de procesamiento. Cada uno de estos niveles representa un tipo de análisis que se debe efectuar al texto de entrada para extraer información específica. Los diferentes niveles de procesamiento existentes son: fonológico, morfológico, léxico, sintáctico, semántico, del discurso y pragmático. Como se puede ver, la Figura 2.1 ilustra todos los niveles nombrados anteriormente. A continuación se describirán los diferentes niveles desde el más bajo, al más alto.

En un principio, el procesamiento del lenguaje natural se pensó como un modelo secuencial que repetía una y otra vez la misma secuencia de procesos para lograr la comprensión de la entrada. Desde el punto de vista psicolingüístico, se admite que el procesamiento del lenguaje es mucho más simple si se plantea una vista de niveles que pueden interactuar en diferentes órdenes. Partiendo del conocimiento existente sobre el lenguaje, un análisis se alimenta de los niveles más altos de procesamiento para asistir a los de más bajo nivel. Por ejemplo, el conocimiento pragmático de que un documento que se está leyendo habla de informática será utilizado cuando una palabra en particular tenga varios sentidos o significados. De esta manera, esta palabra se interpretará con un sentido "informático".

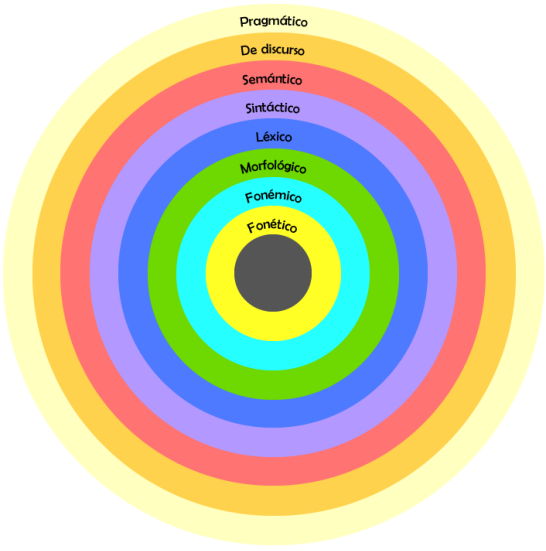


Figura 2.1. Niveles de procesamiento del lenguaje natural.

Siguiendo el orden que muestra la Figura 2.1, a continuación se dará una breve explicación de cada uno de los niveles que forman parte del procesamiento del lenguaje.

Nivel fonético

Se encarga de la interpretación del sonido dentro de las palabras.

Nivel fonémico

Se llama fonemas a las unidades teóricas básicas postuladas para estudiar el nivel fonológico de la lengua humana. Es decir, estudian la variación en la pronunciación cuando las palabras están conectadas. Estos niveles componen el análisis fonológico del lenguaje natural, realizado para los casos en los que la entrada es verbal/sonora.

Nivel morfológico

Se encarga de analizar la composición de las palabras. El análisis de este nivel consiste en determinar la forma, clase o categoría gramatical de cada palabra dentro de una oración. Teniendo en cuenta este nivel, un sistema NLP es capaz de desglosar una palabra y obtener el significado a través del significado de cada uno de sus morfemas. Por ejemplo, si se busca interpretar la palabra "Libros", se obtiene que "Libr-" es el lexema, que "-o-" y "-s" son morfemas dependientes flexivos de masculino y plural.

Nivel léxico

Se encarga del significado individual de cada palabra. Para poder realizar el procesamiento léxico, se debe tratar cada palabra por sí misma, y etiquetarla con una parte del discurso dependiendo del contexto en la cual se encuentra. Cada palabra que compone un texto lleva asociada un conjunto de datos morfológicos, sintácticos y semánticos. El objetivo específico de este nivel es analizar cada una de estas palabras para saber su significado y función dentro de una oración. Por ejemplo, la palabra "lima" puede definirse como un derivado del verbo "limar", pero también puede ser un sustantivo si se refiere al "fruto del limero". Para poder determinar cuál es el rol de cada palabra en un texto en necesario resolver algunas ambigüedades que se presentan en este nivel como lo son la sinonimia, antonimia, entre otras. Por ejemplo, la sinonimia es la relación entre palabras diversas que comparten un mismo significado como puede ser *cerca* y *próximo*, *suave* y *terso*, *dulce* y *azucarado*. En el caso de la antonimia, se trata de una relación de opuestos entre dos palabras; cuando se presenta una antonimia se está haciendo un contraste o comparación entre dos palabras completamente contrarias como son *cerca* y *lejos*, *suave* y *áspero*, *dulce* y *salado*.

Nivel sintáctico

Se encarga de analizar la función de cada palabra dentro de una oración, descubriendo así la estructura u organización de la misma. El resultado de este procesamiento será una representación de la oración analizada que mostrará la relación entre las palabras que la conforman. Por ejemplo:

Se puede observar mediante el ejemplo anterior, como dentro del nivel sintáctico se identifican los componentes de una oración permitiendo conocer la función que cumple el mismo. La sintaxis de esta oración identifica dos artículos "el", dos sustantivos como son "hombre" y "auto", y un verbo "lava". A partir de esta clasificación de palabras, el procesamiento permite determinar el sujeto y el predicado de la oración.

Nivel semántico

Se encarga de obtener el sentido de una oración a partir de la interacción entre las palabras que la conforman. El procesamiento semántico admite sólo un sentido a las palabras con varios significados, y así incluir el sentido en la representación semántica de la oración. Para ilustrar el análisis realizado en este nivel con un ejemplo, se plantea la desambiguación de la palabra "vela" teniendo en cuenta los siguientes dos significados:

1. Cilindro de cera o sebo, atravesado por una mecha que se prende para alumbrar.
2. Pieza de lona o lienzo fuerte que, atada a las vergas, recibe el viento que impulsa la nave.

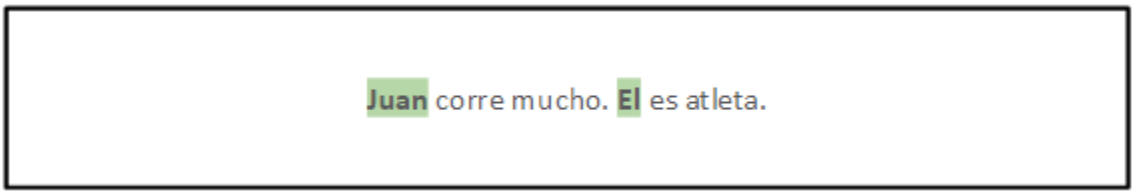
Dos oraciones pueden contener esta palabra con cualquiera de los dos significados. Por ejemplo:

1. Puso dos **velas** a San Pancracio
2. Los egipcios fueron los primeros constructores de barcos de **vela**.

A partir del análisis realizado por este nivel, se puede determinar el significado que tiene la palabra "vela" en cada caso. En la frase n°1, se puede ver que la palabra "vela" está acompañada del nombre propio "San Pancracio", que se refiere a un santo. Teniendo en cuenta esto, la definición que mejor aplica es la n°1, ya que son conceptos relacionados con la religión. En la oración n°2, se puede ver que se encuentra la palabra "barcos" acompañando a la palabra en cuestión. De esta manera, se determina que corresponde mejor con la definición n°2 de vela, ya que hace referencia a un objeto que impulsa una nave.

Nivel de discurso

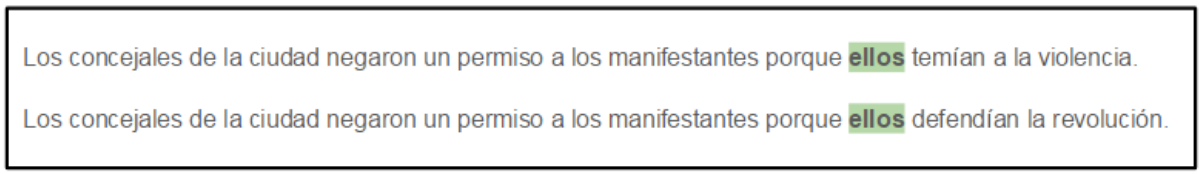
Se encarga de trabajar con unidades de texto más grande que los niveles anteriores. Hace foco en el texto como un todo para alcanzar el significado haciendo conexiones entre las oraciones. Dos de los procedimientos que son realizados por este nivel son la resolución de anáforas, y el reconocimiento de la estructura del texto. La resolución de anáforas consiste en reemplazar pronombres con la entidad a la que hacen referencia. En este caso, el procedimiento resuelve que el pronombre "Él" de la segunda oración, hace referencia a "Juan", presente en la oración anterior:



El reconocimiento de la estructura del discurso trata de identificar la función que cumple cada oración en el texto, sumando información al significado del texto completo. Por ejemplo, un artículo de diario puede estar compuesto por una *iniciativa*, una *historia principal*, *eventos previos*, una *evaluación*, y las *expectativas*. Conociendo la intención de cada uno de estos componentes del discurso, resulta más sencillo comprender la idea que se quiere transmitir.

Nivel pragmático

El nivel pragmático se encarga de analizar las diferentes variables relevantes para la comprensión de un texto o para explicar la elección de determinadas formas de llevarlo a cabo en función de los factores contextuales. Entre las variables se pueden mencionar: la situación, el emisor, el receptor, el enunciado, y en caso de tratarse de lenguaje verbal, el tono en el que se está expresando. Este nivel utiliza el contexto por encima de los contenidos del texto para la comprensión. Para comprender mejor la función del nivel pragmático, se tiene el siguiente ejemplo:



En este caso, se puede observar como la palabra "ellos" en la primer oración hacer referencia a los "concejales", mientras que en la segunda oración, referencia a los "manifestantes". La única manera de poder determinar el sentido de esa palabra dentro del texto, es mediante el contexto. En ambos casos, a partir del conocimiento de cómo se desarrollan las manifestaciones políticas, se puede determinar a quién hacer referencia "ellos".

2.1.2. Técnicas NLP

Para lograr el procesamiento de lenguaje natural, existen un conjunto de técnicas mediante las cuales se extrae del texto información determinada. A continuación, se describirán algunas de las técnicas más comunes utilizadas por los diferentes sistemas NLP para procesar texto escrito en lenguaje natural. Cada técnica de NLP puede implicar uno o varios niveles de procesamiento explicados en la sección anterior.

Detección de oraciones

La detección de oraciones es una de las técnicas básicas correspondiente al nivel de procesamiento sintáctico. Si bien parece una tarea simple, detectar oraciones tiene ciertas dificultades a la hora de procesar títulos, abreviaturas, lista de elementos, y otros componentes que no siguen un patrón de texto plano. Esta técnica funciona recortando una secuencia de caracteres entre dos signos de puntuación. El signo debe estar acompañado por un espacio en blanco. Excluyendo el caso de la primer frase y en posibles ocasiones la última frase. Para determinar las abreviaciones en el texto se utiliza palabras cargadas en el modelo. Por esto la técnica utiliza un modelo por idioma, ya que tiene símbolos o abreviaturas necesarias para detectar las sentencias. En el siguiente ejemplo se puede observar la separación en oraciones de un fragmento de texto.

Pierre Vinken, 61 años, se unirá a la junta como director no ejecutivo 29 noviembre. El Sr. Vinken es presidente de Elsevier NV, el grupo editorial holandés.	<Párrafo inicio="0" fin="156"> <Oración inicio="0" fin="83" /> <Oración inicio="84" fin="156" />

Se puede ver en el ejemplo la delimitación de oraciones dentro de un párrafo indicando el caracter de inicio y final. Un caso especial que se puede encontrar en el fragmento de texto es la abreviatura “Sr.”. El modelo en español determina que el punto que existe en “Sr.” es por una abreviatura a la palabra “Señor”, ignorando de esta manera el signo de puntuación como final de oración.

Segmentación por palabras

Una vez identificadas cada una de las oraciones que componen el texto, el siguiente paso es la segmentación por palabras, más conocida como analizador léxico o “Tokenizer”. Esta técnica pertenece al nivel léxico y consiste en la identificación de *tokens*, los cuales son unidades lingüísticas como palabras, puntuación, números, caracteres alfanuméricos, etc. Una forma de identificar *tokens* en idiomas modernos que utilizan un sistema de escritura basado en el Griego, como el Inglés y otros idiomas Europeos, se realiza delimitando espacios en blanco con límites de palabra, entre comillas, paréntesis y puntuación.

El trato con las abreviaciones es similar a la detección de oraciones, ya que no existen normas universalmente aceptadas para muchas abreviaturas y acrónimos. El enfoque más adoptado para el reconocimiento de abreviaturas es mantener una lista de palabras recortadas reconocidas.

Algunos ejemplos que pueden traer problemas son las direcciones de emails, palabras con apostrofes, URLs, ciudades, etc. Un ejemplo de esta problemática es el siguiente. Suponiendo que dentro de un fragmento de texto se encuentran los valores decimales 7.1 ó 82.4, al segmentar por cada valor específico se obtendrán los *tokens* "7", "1", "82", y "4". Este resultado no es el esperado si se trataba de extraer el valor decimal como un único *token*. Lo mismo ocurre para el valor "$2,023.74", ya que el analizador admite que tanto el punto como la coma son delimitadores, y por lo tanto dividirá el número en tres partes. Asimismo, se debe tener en cuenta el idioma con el cual se está trabajando, ya que por ejemplo en idiomas como el Chino Mandarín, no se definen límites tan claros entre las palabras como en el Español o Inglés, y puede tornar esta tarea más compleja. Para ilustrar el funcionamiento de esta técnica, considere el siguiente ejemplo:

Pierre Vinken, 61 años, se unirá a la junta como director no ejecutivo 29 noviembre. El Sr. Vinken es presidente de Elsevier NV, el grupo editorial holandés.	... <Palabra inicio="0" fin="5" /> <Palabra inicio="7" fin="12" /> <Palabra inicio="15" fin="16" /> <Palabra inicio="20" fin="23" /> ...

En el ejemplo se puede ver la separación por palabras indicada tanto por los espacios en blanco como por signos de puntuación. De esta manera, se obtiene el listado de palabras que componen el párrafo.

Etiquetado gramatical o Part-of-Speech(POS)-tagging

Una vez ejecutadas las dos técnicas previamente explicadas, se puede realizar el proceso de etiquetar las palabras según el rol que cumplen dentro de una oración. Este proceso de NLP se conoce como Etiquetado gramatical o Part-of-Speech(POS tagging), etiquetado de partes del discurso. Este proceso se encarga de asignar a cada una de las palabras de un texto su categoría gramatical de acuerdo a la definición de la misma o el contexto en que aparece, por ejemplo, sustantivo, adjetivo, adverbio, etc. Para ello es necesario establecer las relaciones de una palabra con sus adyacentes dentro de una frase o de un párrafo. Un mismo token puede tener múltiples etiquetas POS, pero solo una es válida dependiendo del contexto.

Son numerosos los sistemas que automatizan la asignación de partes del discurso ("tagging"). Muchos de ellos utilizan técnicas tales como modelos ocultos de Markov (Brants 2000), enfoque de máxima entropía (Ratnaparkhi 1996), y el aprendizaje basado en la transformación (Brill 1994). Sin embargo, la gran mayoría de estos métodos utilizan la misma información para determinar las etiquetas POS, por lo que obtienen niveles de desempeño similares. Un ejemplo de esta técnica es el siguiente:

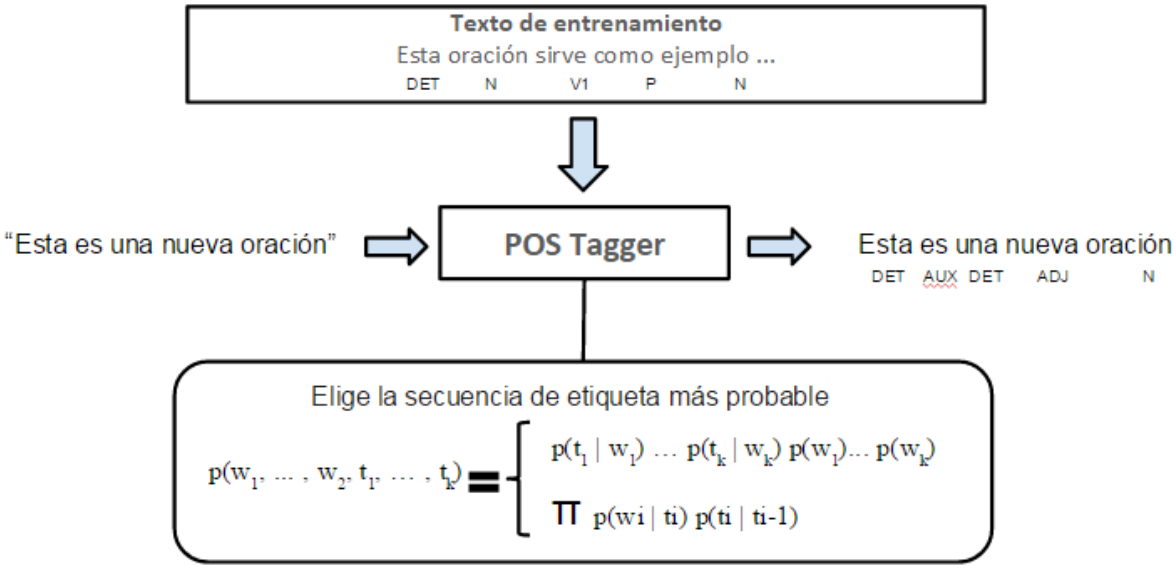


Figura 2.2. POSTagger.

En la Figura 2.2, se puede ver un etiquetado básico de una oración. En primer lugar, las técnicas de etiquetado de partes del discurso poseen un conjunto de textos mediante la cual se entrenan los modelos de predicción. Luego, el etiquetador recibe un texto de entrada para analizar, y busca la secuencia de etiquetas más probable a partir del modelo matemático aprendido. Este punto depende exclusivamente de la técnica en sí, ya que utilizan cada una un modelo distinto.

Segmentación morfológica

Otra técnica de NLP muy utilizada en el procesamiento de texto, es la segmentación morfológica o en morfemas. Un morfema es el fragmento mínimo capaz de expresar el significado de una palabra, es decir, es la unidad significativa más pequeña de un idioma. Un morfema no es idéntico a la palabra, ya que este puede estar acompañado por ejemplo, por prefijos o sufijos, mientras que una palabra, por definición, es independiente. Cuando una palabra se sostiene por sí misma, se considera una raíz porque tiene un significado propio y cuando depende de otro morfema para expresar una idea, es un afijo porque tiene una función gramatical. Cada palabra puede comprender uno o más morfemas.

Los morfemas se clasifican en 2 categorías. Los morfemas "independientes" admiten cierta libertad fonológica del lexema. En algunos casos pueden formar por sí solos una palabra:

Pronombres: cuíde-se, di-le, él, ella.

Preposiciones: desde, a, con, de.

Conjunciones: y, e, o, pero, aunque.

Determinantes: él, ella, ese, un, una.

Los morfemas "dependientes" van unidos o fusionados a otra unidad mínima dotada de significado, también conocidos como monema, para completar su significado. En ciertos casos provocan cambios de

acento, cambios fonéticos en los fonemas adyacentes y sólo pueden aparecer en un orden secuencial concreto. Hay dos tipos de morfemas:

Derivativos: Estos morfemas son facultativos, es decir, añaden algunos matices al significado de los lexemas.

- Prefijos
- Sufijos
- Interfijos

Flexivos: Estos morfemas son constitutivos, es decir, señalan las relaciones gramaticales y sus accidentes entre los diferentes agentes de una acción verbal o una expresión nominal.

- Género
- Número
- Persona
- Modo y tiempo

La identificación de monemas permite el análisis en profundidad de una palabra dentro de un fragmento de texto, proporcionando información específica como puede ser el género, modo y tiempo, entre otras. Mediante el análisis realizado con esta técnica se puede ubicar de manera precisa cada palabra de cada oración.

Eliminación de "Stop Words"

La técnica "Stop Word" es utilizada para excluir palabras muy comunes que suelen tener poco valor para recuperar información que necesita el usuario. La cantidad de ocurrencias de una palabra en el texto determina si es o no una "stop word", dado que cuanto más ocurrencias existan menos relevancia tiene en el texto. Dentro de este grupo se encuentran los artículos, los pronombres, las preposiciones, y las conjunciones. Esta técnica permite reducir el tamaño del texto para analizar, eliminando aproximadamente el 30% o 40% de dichas palabras. Además, se mejora la eficiencia, ya que la selección de palabras claves es más precisa. A continuación se plantea una identificación de "stop words" para poder observar de manera clara cuál es la ventaja de esta técnica.

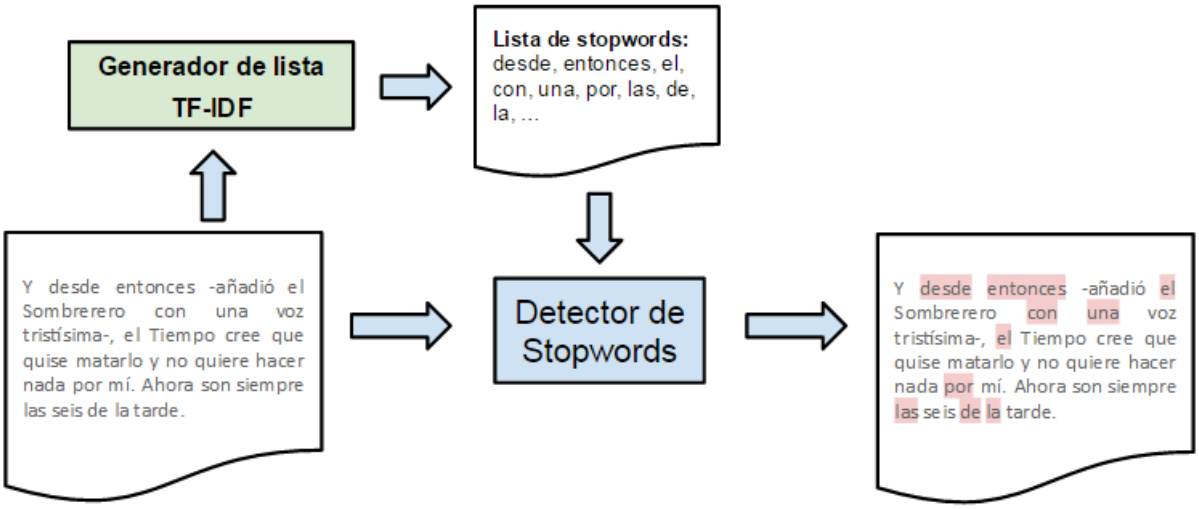


Figura 2.3. Detección de Stopwords.

Esta detección de *stop words* está realizada con un listado en español que se encuentra en la Web[1]. Estos listados proveen un conjunto de palabras recurrentes en un idioma específico. La Figura 2.3, ilustra cómo a partir de un listado de palabras *stopwords*, estas son detectadas dentro del texto indicando que es posible su eliminación. En este caso en particular, al listado de palabras de uso común, se le agrega un conjunto de palabras propias del documento que se desea analizar. Esto se lleva a cabo mediante la medida numérica TF-IDF (Term Frequency - Inverse Document Frequency), que permite determinar que palabras son importantes para un documento dado de acuerdo a la frecuencia de aparición dentro del texto. Estas es una de las técnicas que se pueden utilizar para generar una lista de *stop words*, como así también lo es el Modelo de Espacio Vectorial (MSV) para determinar la relevancia de una palabra, entre otros.

Reconocimiento de Entidades Nombradas (NER)

Es una subtarea de la extracción de información que busca y clasifica elementos del texto que pertenecen a categorías predefinidas como pueden ser nombres de personas, entidades, organizaciones, lugares, expresiones temporales, cantidades, porcentajes, etc.

Para poder reconocer las diferentes entidades se utilizan una serie de aproximaciones. En primer lugar, algunas entidades simples se pueden reconocer mediante patrones codificados con expresiones regulares para encontrar entidades de fecha, tiempo, velocidad, etc. También suelen haber técnicas que utilizan una lista ordenada para reconocer nombres de personas, lugares, organizaciones, etc. Por último, existen reconocedores de entidades que utilizan un algoritmo de entropía máxima para clasificar cada uno de los tokens como un tipo de entidad particular en caso de que así sea.

[1] http://www.ranks.nl/stopwords/spanish

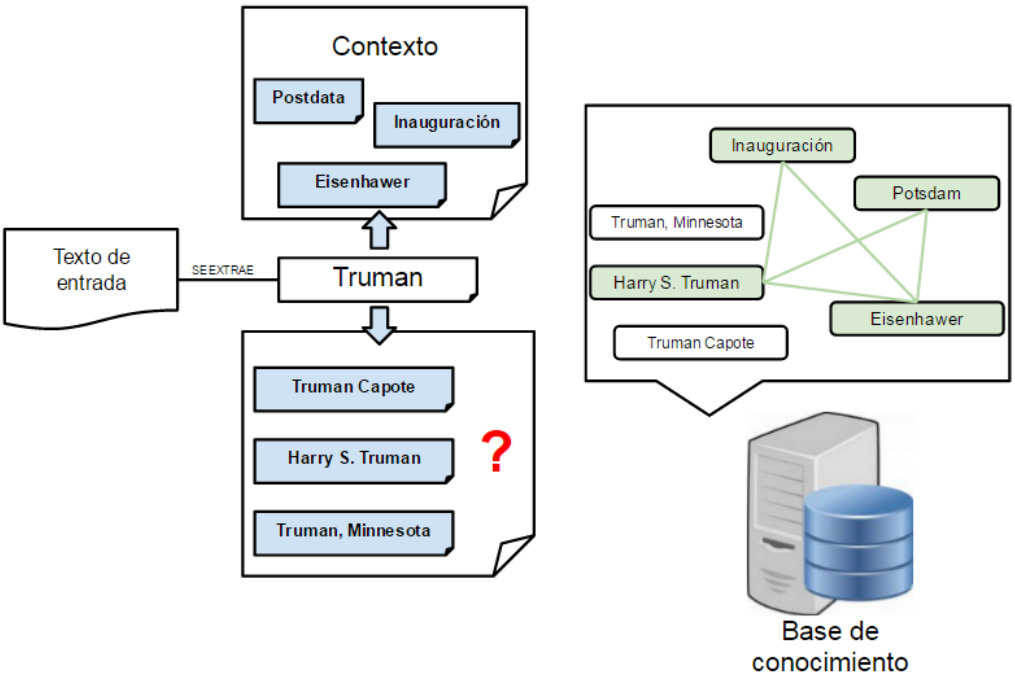
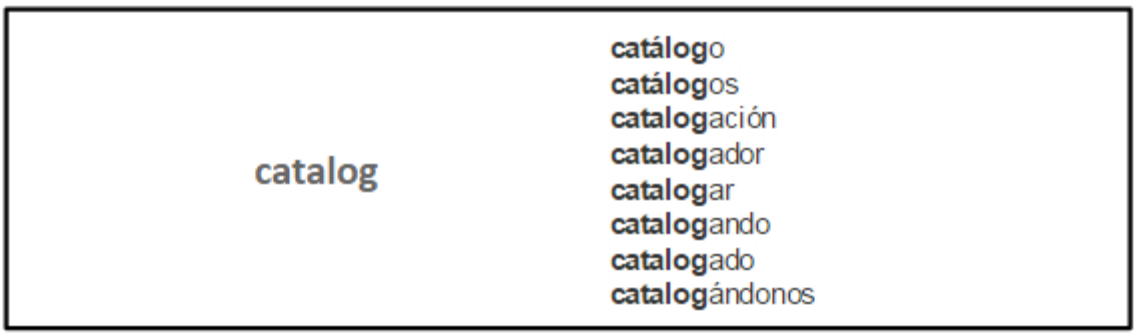


Figura 2.4. Reconocimiento de entidades nombradas.

Como se puede ver en el ejemplo de la Figura 4, muestra el funcionamiento de una técnica de NER. En este caso se extrae la entidad "Truman" de un texto de entrada la cual permite realizar varias interpretaciones. En este caso "Truman" puede referirse a "Truman Capote", "Harry S. Truman" o "Truman, Minnesota". Para poder realizar el análisis es necesario tener una noción del contexto en el cual se encuentra esta entidad para poder determinar a qué se refiere. Dentro de las posibles entidades se realiza una asociación con los conceptos del contexto dentro de una base de datos de conocimiento. En este caso, "Harry S. Truman" se relaciona con el concepto de "Inauguración", con la ciudad de "Potsdam" y con el apellido "Eisenhawer", de manera que la entidad "Truman" hallada en el texto se asocia con esta entidad.

Stemming

Las palabras están morfológicamente estructuradas en prefijos, sufijos y una raíz. La técnica de Stemming busca un concepto de la palabra eliminando tanto prefijos como sufijos y obteniendo la raíz. De esta manera, se efectúa una reducción de la palabra a su mínimo elemento con significado. Un término que es reducido a su común denominador simplifica la recuperación de documentos cuyas palabras tenga la misma raíz. Por ejemplo:



Como se puede ver en este ejemplo, todos los términos derivan de la raíz "catalog", haciendo posible la recuperación de ocho palabras que comparten una misma raíz como derivados con el mismo significado. Aun así, esta técnica no siempre funciona correctamente ya que hay palabras que poseen raíces compartidas por más de un significado, como se puede ver en la siguiente Tabla 1:

Término con prefijo	Raíz/Stem	Término con el que causaría confusión
Prevalencia	**valenc**	**Valencia, valencia, valenciano, ambivalencia, polivalencia,**
Precatalogar	**catalog**	**Descatalogar, catalogo,**

Tabla 2.1.

Uno de los métodos más conocidos para llevar a cabo la reducción morfológica es el algoritmo de Martin Porter [2]. También existe un lenguaje llamado Snowball, que permite el desarrollo de reglas para la extracción de *stems* de manera sencilla. Una vez que se tienen los algoritmos, estos son compilados por Snowball traduciendo el contenido a C o Java, permitiendo así incluir el análisis desarrollado en un proyecto desarrollado en dichos lenguajes.

2.1.3. Fundamentos de NLP

Los fundamentos de NLP vienen dado por varias disciplinas como los son las ciencias de la Computación y la Información, Lingüística, Matemática, Inteligencia Artificial, entre otras. Hasta la década de 1990, los sistemas de NLP fueron construidas manualmente con diccionarios y reglas hechas a mano. Al crecer el tamaño de la información, los investigadores comenzaron a utilizar técnicas de aprendizaje automático a construir automáticamente los sistemas de NLP. Hoy en día, la gran mayoría de la de sistemas NLP utilizan algoritmos de aprendizaje de máquina (Machine Learning).

[2] http://tartarus.org/martin/PorterStemmer/

El aprendizaje automático o aprendizaje de máquinas, más conocido como *Machine Learning*, es una rama de la inteligencia artificial que tiene como objetivo el desarrollo de técnicas que permitan a las computadoras adquirir conocimiento. Esto se lleva a cabo mediante el uso de modelos estadísticos que buscan automatizar partes del método científico.

A lo que refiere dentro del estudio del Procesamiento del Lenguaje Natural, el aprendizaje de máquina se introdujo a partir de la construcción manual de gramáticas y bases de conocimiento en un principio. Posteriormente, los procedimientos se automatizan completamente mediante el uso de métodos de aprendizaje estáticos entrenados con grandes contenidos de lenguaje natural. Además, el procesamiento automático contribuyó al desarrollo de un número importante de métodos y técnicas capaces de resolver una gran variedad de problemas de adquisición y comprensión del lenguaje como pueden ser: extracción automática de conocimiento léxico, desambiguación léxica y estructural (POS Tagging, Word Sense Disambiguation, etc), recuperación y extracción de información, traducción por máquina, entre otras.

Los enfoques de aprendizaje se categorizan generalmente como métodos estáticos, y simbólicos. Dentro de los métodos estáticos se encuentran Naive-Bayes, principio de máxima entropía, Modelos de Markov, etc. En el grupo de los métodos simbólicos se pueden encontrar métodos de árboles de decisión, listas de decisión, aprendizaje basado en instanciaciones, separadores lineares, entre otros. Cada uno de estos métodos tiene su campo de aplicación determinado, como por ejemplo es el caso de el Principio de Máxima Entropía que es utilizado para el etiquetado de partes del discurso, extracción de morfemas y categorización de texto. De la misma manera, el método simbólico de árboles de decisión permite un análisis del texto mucho más detallada, permitiendo realizar reconocimiento del discurso, desambiguación del sentido de las palabras (WSD) y resúmenes automáticos, entre otras tareas.

Desde sus diferentes enfoques, el aprendizaje de máquina o *Machine Learning*, ha permitido el desarrollo de métodos que tienen una mayor precisión en diferentes niveles NLP. La posibilidad de automatizar el análisis de grandes cantidad de texto, permite obtener un mayor nivel de comprensión y de aprendizaje.

2.2. Web Services

Existen múltiples definiciones de los Servicios Web, dado que debido a su complejidad es difícil encontrar una caracterización que englobe todo lo que son e implican [W32015]. Un servicio web se puede definir como una aplicación interoperable a través de la Web. UDDI (Universal Description, Discovery and Integration) Consortium define un servicio Web como una aplicación de negocio modular y autocontenida, con interfaces estandarizadas orientadas a Internet. Esta definición hace énfasis en la necesidad de compatibilidad entre las partes a partir de estándares de Internet. Sin embargo, no queda claro el concepto de aplicación de negocio autocontenida. Es por eso, que en la búsqueda de la definición correcta de servicios Web, aparece la siguiente:

"Un Servicio Web (Web Service [WS]) es una aplicación software identificada por un URI (Uniform Resource Identifier), cuyas interfaces se pueden definir, describir y descubrir mediante documentos XML. Los Servicios Web hacen posible la interacción entre "agentes" software (aplicaciones) utilizando mensajes XML intercambiados mediante protocolos de Internet."

Definición: (del World Wide Web Consortium [W3C]

Esta definición de W3C es más precisa porque, especifica mejor cómo deben trabajar los servicios Web. Según esta definición, los servicios Web se deben poder "definir, describir y descubrir", dejando en claro la idea de "accesible" que se planteó anteriormente. Además, especifica que estos servicios proporcionan mecanismos de comunicación estándares, como XML, entre diferentes aplicaciones que interactúan entre sí para presentar información dinámica al usuario [Allen2006, Marks2006].

Para ejemplificar el funcionamiento y la utilidad de los servicios web se planteó el siguiente ejemplo:

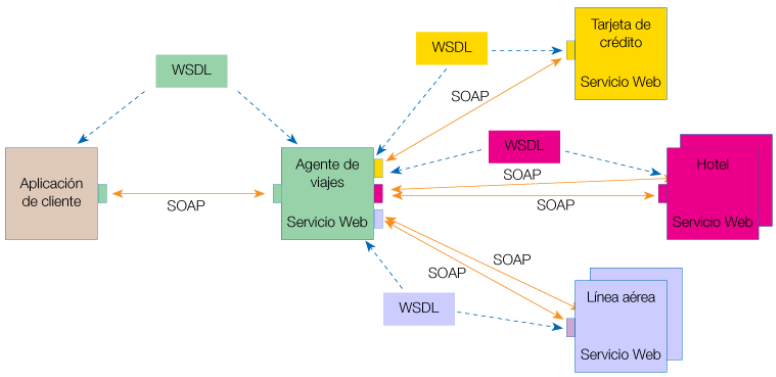


Figura 2.5. Diagrama de sistema de agencia de viajes.

En la figura 2.5 un usuario, solicita información sobre un viaje que desea realizar haciendo una petición a una agencia de viajes que ofrece sus servicios a través de Internet. En este caso, el usuario juega el papel de cliente dentro de los servicios Web. La agencia de viajes ofrecerá a su cliente la información requerida. Para proporcionar al cliente dicha información esta agencia de viajes solicitará a su vez información a otros recursos en relación con el hotel y la compañía aérea. La agencia de viajes obtendrá información de estos recursos, lo que la convierte a su vez en cliente de esos otros servicios Web que le van a proporcionar la información solicitada sobre el hotel y la línea aérea. Por último, el usuario realizará el pago del viaje a través de la agencia de viajes que servirá de intermediario entre el usuario y el servicio web que gestionará el pago.

De esta manera, se puede apreciar como un usuario accede desde una aplicación a un servicio que opera con otros, de la misma manera en la que la aplicación del cliente lo hace con él. Para proporcionar interoperabilidad y extensibilidad entre estas aplicaciones, y que al mismo tiempo sea posible su combinación para realizar operaciones complejas, es necesaria una arquitectura de referencia estándar.

2.2.1. Arquitectura Web Service

Un servicio Web es una interfaz que describe con conjunto de operaciones accesibles a través de la web mediante mensajes a través de protocolos web, como puede ser HTML o TCP-IP, que puede llevar a cabo una o varias tareas. La descripción de un servicio web se hace a partir de estándares formales de notación, como puede ser XML, o JSON. Esta descripción provee todos los detalles necesarios para poder interactuar con el servicio, incluyendo el formato de los mensajes, los protocolos utilizados y la ubicación.

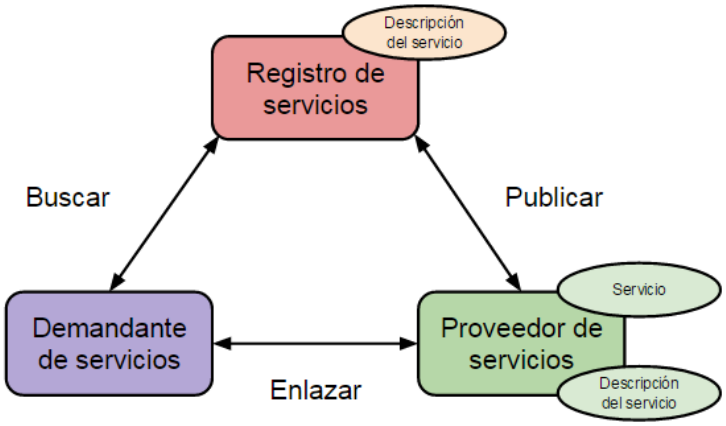


Figura 2.6. Diagrama de arquitectura Web Service.

Como se puede ver en la Figura 2.6, esta arquitectura establece tres componentes y tres operaciones. Los componentes son el proveedor de servicios, el demandante de servicios y el registro de servicios. En cuanto a las operaciones que realizan los actores sobre los servicios y sus descripciones son buscar, publicar y enlazar.

El "proveedor de servicios" es la entidad que crea los servicios Web y el encargado de hacer su descripción en algún formato estándar como puede ser UDDI. En esta descripción se encuentran todos los detalles referidos a los servicios a prestar por el proveedor, indicando entre otras cosas, la manera en la que se accede y los parámetros necesarios para realizar las tareas.

El "registro de servicios" es el lugar donde el "proveedor de servicios" publica las descripciones de los servicios que tiene creados. Además de contener esta información, el "registro de servicios" se encarga de realizar la búsqueda y generar los enlaces necesarios para llamar a estos servicios desde una aplicación. Esta aplicación que invoca al "registro de servicios", cumpliría la función de "demandante de servicios".

El "demandante de servicios" es el encargado de invocar y/o interactuar con el servicio a través de un navegador o una interface de programa. Este componente obtendrá los datos del procesamiento realizado por el "proveedor de servicios", en un formato estandarizado para poder comprender el resultado de la operación. El componente demandante obtiene las especificaciones de uso de los servicios y los enlaces al proveedor del registro.

Una vez que la unidad demandante tiene todos los datos para poder invocar un determinado conjunto de servicios, se enlaza con el proveedor para interactuar en tiempo de ejecución. En esta comunicación, las dos partes utilizan la información del descriptor generado en un principio por el proveedor para poder llevar a cabo la comunicación deseada. Debido a la abstracción que provee el hecho de que estos servicios se basen en interfaces estándar, no influye que un servicio esté escrito en Java y el navegador que lo solicite esté en C++, como así tampoco que el servicio se encuentre en UNIX y el navegador en Windows. Los servicios web permiten la interoperabilidad entre plataformas a tal punto que la plataforma pasa a ser irrelevante.

2.2.2. Arquitecturas SOAP

Uno de los protocolos de servicios Web más conocido es el Simple Object Access Protocol (SOAP) [Huang2003]. Este es un protocolo de empaquetamiento de mensajes que se comparten entre aplicaciones. La especificación sólo define un XML con la información a ser transmitida, un conjunto de reglas para traducir el mensaje y información específica de la representación del contenido. El diseño de este protocolo hace posible el intercambio de mensajes entre un conjunto amplio de tipos de aplicaciones. Debido a esto, SOAP logró la gran popularidad que tiene hoy en día en materia de servicios Web. SOAP es una aplicación de especificaciones XML. La definición del protocolo se basa en los estándares XML como lo son XML Schema y XML Namespaces, para su definición y funcionamiento.

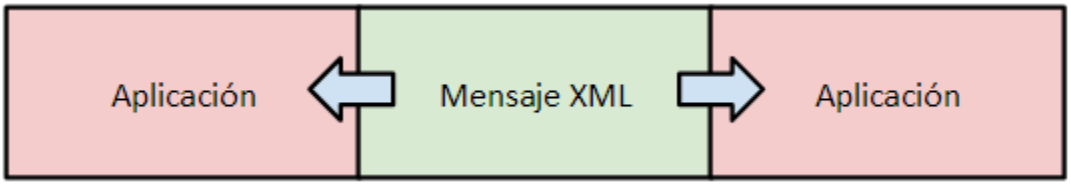


Figura 2.7.

Como se puede observar en la Figura 2.7, las aplicaciones intercambian información mediante mensajes XML. Una aplicación SOAP debe incluir la descripción del formato en el cual se escriben los mensajes que genera para que el receptor pueda interpretar correctamente el mensaje enviado. Este receptor, además, debe estar preparado para procesar estos mensajes y así poder extraer la información contenido por el XML. Los mensajes SOAP están compuestos por una envoltura que contiene un cuerpo, y opcionalmente una cabecera, como se puede ver en la 2.8.

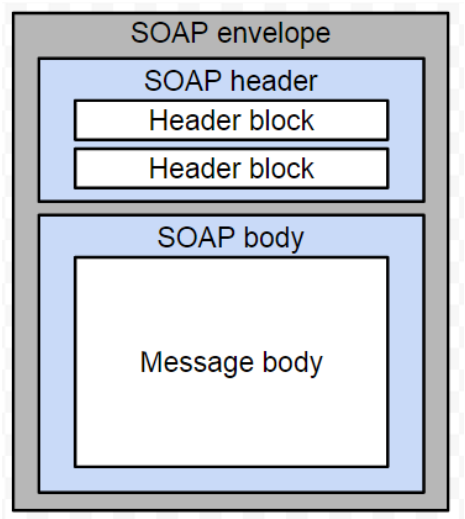


Figura 2.8. Composición de mensaje SOAP.

Las cabeceras contienen bloques de información que indica cómo se debe procesar el mensaje. Estas especificaciones pueden ser configuraciones de ruteo y entrega, datos de autenticación, contextos de transacciones, entre otros. El cuerpo contiene el contenido del mensaje que debe ser entregado y procesado.

La sintaxis para expresar un mensaje SOAP está basada en "http://www.w3.org/2001/06/soap-envelope". Este espacio de nombres XML identifica puntos para un esquema de este tipo que define la estructura del formato de un mensaje SOAP. A continuación, se puede ver un ejemplo de un mensaje SOAP con un ejemplo de una orden de compras.

```
<s:Envelope xmlns:s=''http://www.w3.org/2001/06/soap-envelope''>
<s:Header>
        <m:transaction
xmlns:m=''soap-transaction''s:mustunderstand="true">
                <transactionID>1234</transactionID>
        </m:transaction>
</s:Header>
<s:Body>
        <n:purchaseOrder xmlns:n=''urn:OrderService''>
                <from>
                        <person>Christogher Robin</person>
                        <dept>Accounting</dept>
                </from>
                <to>
                        <person>Pooh Bear</person>
                        <dept>Honey</dept>
                </to>
                <order>
                        <quantity>1<quantity>
                        <item>Pooh Stick</item>
                </order>
        </n:purchaseOrder>
</s:Body>
</s:Envelope>
```

En este fragmento de código se puede apreciar resaltado con color rojo la etiqueta "Envelope" que enmarca el mensaje SOAP. Luego, se encuentra el "Header" o cabecera, que contiene datos, como se dijo anteriormente, referidos a la transacción. La etiqueta "transaction" tiene atributos "xmlns" y "mustunderstand", que indican el tipo de transacción que se debe realizar. Además, la etiqueta "transactionID" contiene un identificador como dato, que será de utilidad para el receptor. Seguido a la

cabecera, se encuentra el cuerpo con los datos específicos de la orden de compra. Como se ve, el contenido especifica el emisor de la orden de compra dentro de la etiqueta “purchase Order”, con la etiqueta “from”. De la misma manera, se especifican los datos del receptor dentro de la etiqueta <to>. En <order> van los datos referidos a la orden, como son el artículo que desea comprar y la cantidad que solicita el comprador.

Además de los mensajes, SOAP provee un mecanismo de manipulación de errores denominado faltas. Estos son mensajes SOAP que indican alguna falla que se obtuvo en el procesamiento del mensaje enviado. Estos mensajes tiene un conjunto de especificaciones dentro del protocolo de manera tal que puedan ser interpretados por el emisor, y de esta manera, se pueda corregir el error.

Debido al uso de XML, este protocolo permite invocar procedimientos remotos sea cual sea el lenguaje en el que el procedimiento este escrito, dando esto como resultado una gran interoperabilidad. La utilización del protocolo HTTP para la comunicación hace que este protocolo sea fácilmente escalable. Por otro lado, SOAP es mucho más lento que otros middleware como puede ser el caso de CORBA (Common Object Request Broker Architecture) o ICE (Internet Communications Engine), ya que los datos binarios se codifican como texto. Para mejorar sobre este aspecto se desarrolló un XML que contiene datos binarios.

Después de que SOAP se introdujo por primera vez, pasó a ser parte de la capa subyacente de un conjunto más complejo de web services basada en WSDL (Web Service Description Language) y UDDI (Universal Description Discovery and Integration) [Graham2005].

2.2.3. REST (Representational State Transfer)

REST (Representational State Transfer) o Transferencia de Estado Representacional, es una arquitectura de software para sistemas distribuidos [Sandoval2009]. El concepto fue introducido en el año 2000 en la tesis doctoral de Roy Fielding, y pasó a ser ampliamente utilizado por la comunidad de desarrollo. REST define un conjunto de principios arquitectónicos mediante los cuales se diseñan servicios web haciendo foco en los recursos del sistema, como por ejemplo el método de acceso a dichos recursos y cómo se transfieren hacia los clientes escritos en diversos lenguajes de programación.

En los últimos año, las arquitecturas REST emergieron como modelo predominante para el diseño de servicios, debido a que esta arquitectura es simple de usar. Específicamente, REST ha logrado desplazar a SOAP y las interfaces basadas en WSDL.

Su creador evaluó todas los recursos existentes en la red y las tecnologías disponibles para crear aplicaciones distribuidas hasta llegar a REST. A partir de estos estudios, propuso una arquitectura que permitiera romper las limitaciones existentes hasta el momento, cómo podía ser el desarrollo de aplicaciones complejas difíciles de mantener y extender. Teniendo en cuenta esto, Fielding propuso los siguientes principios para la definición de sistemas REST:

- Debe ser un sistema cliente-servidor.
- No debe mantener estado.
- Debe soportar un sistema de cache.
- Debe ser accesible de manera uniforme.
- Debe ser diseñado por capas.
- Codigo on-demand.

Estos principios no determinan qué tipo de tecnología se debe utilizar, sino que define cómo debe ser transferida la información entre los componentes y cuáles son los beneficios de tomar estos principios como guía de diseño.

Para poder observar en detalle la arquitectura REST, Fielding plantea una visión a partir del sistema como un todo, entrando en detalle y especificando cada unas de las restricciones que se plantean a medida que sea necesario. Dicho esto, la descripción comienza por el denominado "estilo Nulo", que representa un conjunto vacío de restricciones y limitaciones (Figura 2.9). En este caso, no existen límites entre los diferentes componentes del sistema.

Figura 2.9. Estilo Nulo.

Se dice que la arquitectura REST es híbrida porque se compone de otras arquitecturas. En este caso, y cumpliendo con el principio nombrado anteriormente, plantea una arquitectura cliente-servidor (Figura 2.10).

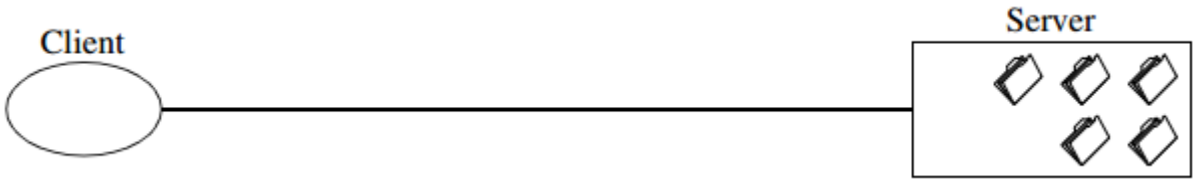


Figura 2.10. Cliente-Servidor.

El principio de la arquitectura cliente-servidor se basa en una buena separación de responsabilidades (Figura 2.11). Separando correctamente la interfaz del usuario de todo lo referido al manejo y almacenamiento de datos, mejora la portabilidad del sistema, como así también mejora la escalabilidad simplificando los componentes del lado del servidor.

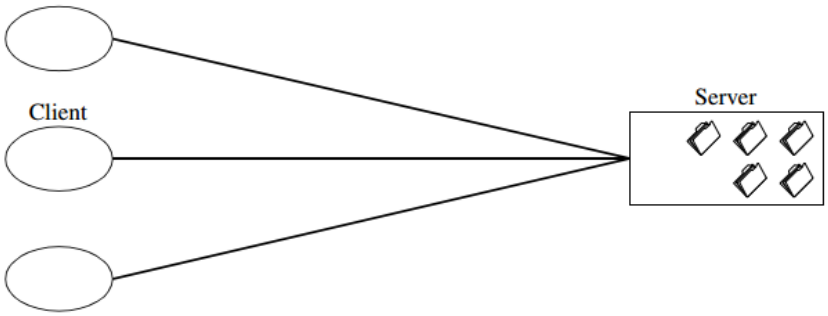


Figura 2.11. Stateless.

La próxima restricción que se aplica a esta arquitectura, es que no debe almacenar estados. Es decir, que cada solicitud que se haga del cliente al servidor debe tener todos los datos necesarios para comprender lo que se solicita sin poder hacer uso de ningún contexto almacenado en el servidor. El estado de sesión se mantiene exclusivamente en el cliente. De esta manera se mejora la visibilidad, la confiabilidad y la escalabilidad.

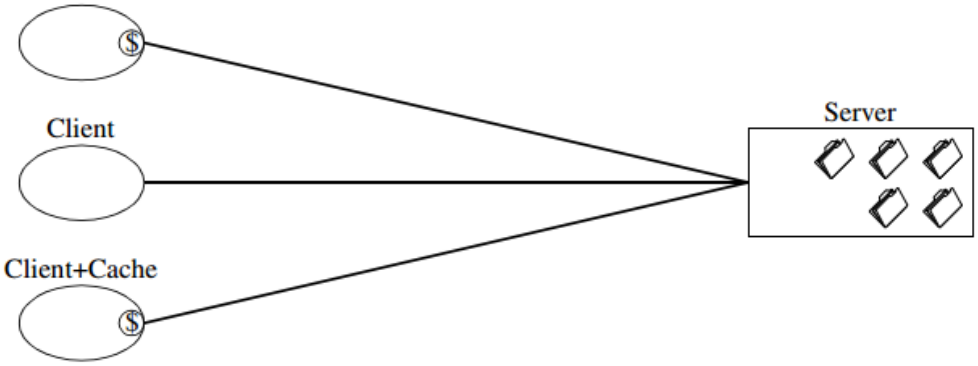


Figura 2.12. Cliente-Servidor con uso de caché.

Para mejorar la eficiencia de la red, se agrega el uso de caché (Figura 2.12). El uso de este componente de memoria del lado del cliente, permite etiquetar la información transmitida a través de una respuesta a una solicitud, determinando si es "cacheable" o "non-cacheable". Si la respuesta es "cacheable", el cliente asigna memoria caché para reusar la información en solicitudes posteriores. La ventaja del uso de caché, es que se tiene la capacidad de eliminar algunas interacciones entre clientes y servidores, mejorando así la eficiencia, la escalabilidad, y la performance que percibe el usuario, ya que se reduce la latencia promedio de respuesta frente a una serie de interacciones.

La arquitectura creada a partir de este conjunto de restricciones da como resultado el diseño de WWW temprano, anterior a 1994, que tenía como objetivo una arquitectura cliente-servidor sin almacenamiento de estados para el intercambio de archivos estáticos a través de la Internet.

Una de las principales diferencias entre el diseño de la arquitectura WWW temprana y REST es el uso de interfaces uniformes entre los componentes (Figura 2.13). Aplicando principio de generalidad de la Ingeniería de Software a las interfaces de los componentes, la arquitectura del sistema se simplifica,

mejorando también la visibilidad de las interacciones entre componentes. La implementación de cada componente se desacopla del servicio que provee el mismo.

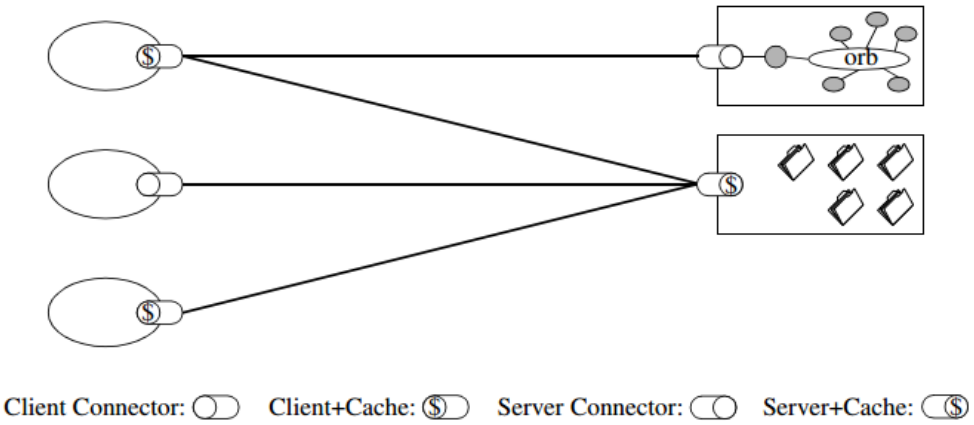


Figura 2.13. Cliente-servidor con interfaces uniformes entre los componentes.

Para mejorar el comportamiento de los sistemas que siguen este diseño arquitectónico, se agrega la noción de diseño por capas (Figura 2.14). Este estilo arquitectónico permite que un sistema esté compuesto por una jerarquía de capas, de manera que cada una de las capas solamente se puede comunicar con una capa contigua, sin saltar niveles.

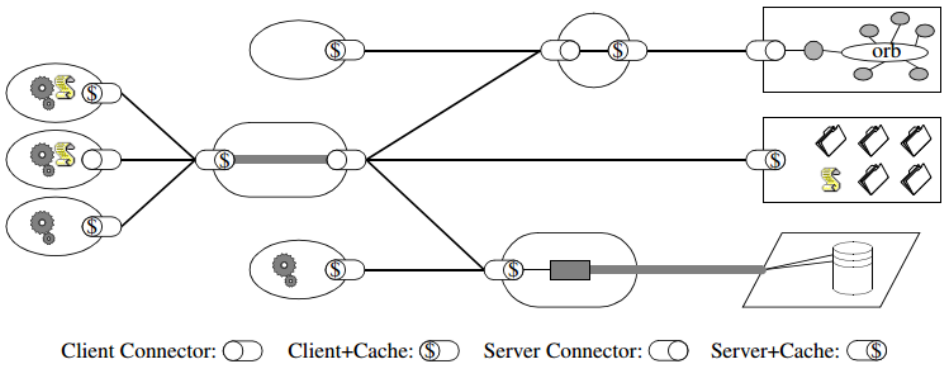


Figura 2.14. Cliente-servidor con jerarquía de capas.

Las capas se pueden utilizar para encapsular servicios, y protegerlos de ciertos usuarios. También mejora la escalabilidad del sistema, permitiendo el balanceo de carga de servicios a través de múltiples redes y procesadores.

Por último, REST permite extender la funcionalidad de los clientes mediante la descarga y ejecución de código en forma de applets o scripts. Esto simplifica el lado del cliente ya que se reduce el número de componentes que se deben pre-implementar. Esta es una restricción opcional en la arquitectura REST, ya que mejora la extensibilidad, pero reduce la visibilidad.

Un concepto importante de resaltar en REST es la existencia de elementos de información denominados *recursos*, que pueden ser accedidos utilizando un identificador global llamado URI (Uniform Resource

Identifier). Estos elementos de información pueden ser manipulados por los componentes de la red que se comunican a través del estándar HTTP e intercambian representaciones de estos recursos. De esta manera una aplicación puede interactuar con un recurso conociendo su identificador.

2.3. Aplicaciones de NLP en la Web

Existen un conjunto de herramientas que generan síntesis de textos extensos a partir de la utilización de técnicas NLP. Por ejemplo, *Tool4Noobs* es una herramienta libre que provee un servicio de resúmenes parametrizable. Dentro de los parámetros configurables por el usuario se encuentra el largo del artículo, la cantidad de palabras y el numero de lineas y/o oraciones que debe contener el resumen. *Stremor Automated Summary and Abstract Generator* es una herramienta que realiza resúmenes al igual que la anterior con la diferencia que esta API genera un resumen de 350 caracteres desde URLs o texto plano. En este campo de desarrollo existe un conjunto muy grande de herramientas que varían a la hora de analizar dependiendo las técnicas que se aplican brindando diferentes resultados[3].

Otra categoría de herramientas que se pueden encontrar son los procesadores de texto que se encargan de obtener datos de interés, como pueden ser las palabras más utilizadas, o una nube de conceptos de uso común. *Skyttle* es una SaaS (Software as a Service) que provee servicios para análisis de texto que extraen patrones de interés del texto y los almacena en un formato estructurado para un análisis profundo de la información [4].

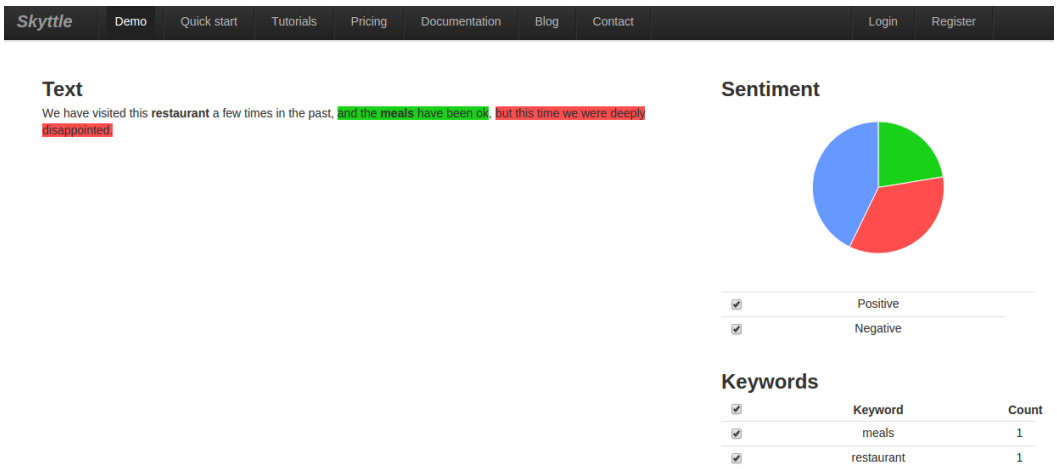


Figura 2.16. Skyttle.com.

En la Figura 2.16 se puede ver un análisis realizado desde una Demo que se encuentra en la web oficial de la herramienta. En este caso, además de realizar el análisis de palabras claves, Skyttle permite extraer los

[3] http://www.howtosummarize.info/list-of-10-great-summarize-generators/

[4] http://www.skyttle.com

sentimientos que se expresan dentro del texto. En este caso, se puede ver que fragmentos de la oración transmiten ideas positivas, y cual transmite ideas negativas.

Otra aplicación a la cual se ajusta NLP es la interpretación y generación de preguntas y respuestas automáticas. En este área se pueden encontrar herramientas como *WebKnox,* la cual permite al usuario ingresar una pregunta, y obtener una respuesta directa, rápida y completa (Figura 2.17). Esta herramienta emplea un conjunto de técnicas NLP para satisfacer esta funcionalidad. En primer lugar, busca a través de servicios externos si la respuesta ya existe. En segundo lugar, busca responder la pregunta utilizando una base de conocimientos propia. Por último, busca coincidir algunos de los conceptos que se tratan en la pregunta y genera una respuesta acorde en tiempo real.

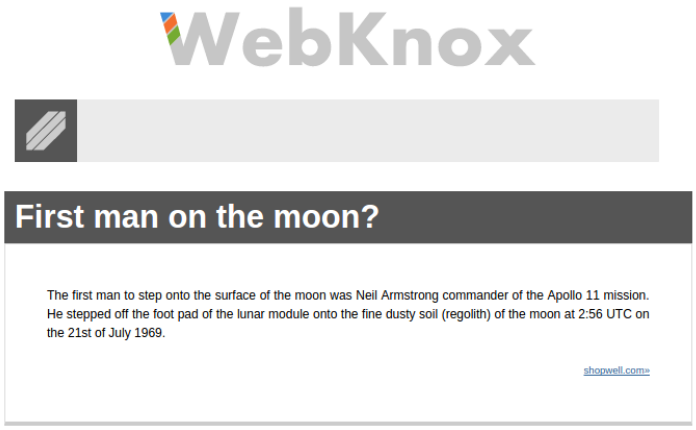


Figura 2.17. Webknox.com.

Si se tiene en cuenta las técnicas NLP de reconocimiento de voz se pueden encontrar herramientas como Jeannie, una aplicación de Android que permite al usuario interactuar con su dispositivo (Figura 2.18). Mediante acciones de voz, Jeannie es capaz de responder a preguntas, enviar correos electrónicos, configurar alarmas, recordatorios, escuchar música de forma automática, entre otras acciones. Esta herramienta realiza un reconocimiento del comando de voz a partir de técnicas NLP de procesamiento de audio y a partir de la transcripción realiza la acción deseada.

Figura 2.18. Jeannie Android Application.

En la actualidad, una gran variedad de artículos son publicados a diario en Internet, dando como resultado un conjunto de objetos de análisis a los cuales se les puede aplicar procesamiento de lenguaje natural para

extraer una información determinada. Diffbot es una aplicación que permite extraer datos de interés de una página Web[5]. Con Diffbot se puede extraer el contenido de una página, identificar los componentes que una página posee, extraer el contenido del texto en limpio, entre otras acciones. Esta herramienta devuelve como resultado un conjunto de datos estructurados en formato JSON.

Como se puede ver, el espectro de aplicaciones en los cuales se puede aplicar NLP es muy amplio. Desde simples aplicaciones para generar síntesis de textos, hasta reconocedores de voz, el NLP es una área que crece día a día. Cada una de estas aplicaciones aumenta su conocimiento con el uso de las herramientas agrandando su base de conocimientos. Además de estas aplicaciones específicas, existe un conjunto de herramientas de uso cotidiano que utilizan técnicas NLP como pueden ser los editores de texto Microsoft Word, que realizan correcciones en el texto sugiriendo modificaciones. Los buscadores de Internet también hacen uso de técnicas de análisis de la información para poder etiquetar contenidos y generar una solución precisa y acertada a la búsqueda pretendida por el usuario.

[5] http://www.diffbot.com/

Capítulo 3 - Trabajos Relacionados

En este capítulo se realiza un análisis de las diferentes plataformas, arquitecturas y aplicaciones que se pueden encontrar en la actualidad que utilizan NLP mediante servicios Web. Este estudio discute cuáles son las características que posee cada uno de estos componentes y sus limitaciones, abarcando un conjunto amplio de sistemas basados en procesamiento de texto.
El conjunto de sistemas NLP presentados a continuación se divide en dos grandes grupos. En primer lugar, se exploran APIs que proveen servicios Web de procesamiento de lenguaje natural. De esta manera, se observan las distintas opciones existentes a la hora de consumir un servicio NLP, reconociendo sus ventajas y desventajas. En segundo lugar, se analizan un conjunto de aplicaciones que hacen uso de este tipo de servicios con el fin de lograr un funcionamiento específico.

3.1. Servicios NLP (APIs)

En esta sección se presentará un conjunto de herramientas que proveen servicios web para llevar a cabo el procesamiento de texto en lenguaje natural. Dentro del gran conjunto de aplicaciones de este tipo encontrados en la Web, se seleccionaron las herramientas más relevantes en el contexto de este trabajo final, de acuerdo a la capacidad de procesamiento del mismo y el uso de cada uno.

3.1.1. Aylien Text Analysis API

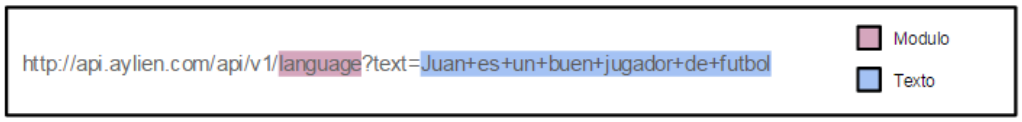
AYLIEN Text Analysis es un conjunto de herramientas de procesamiento de lenguaje natural, recuperación de información y *Machine Learning*, que permiten extraer significado y conocimiento tanto de contenidos textuales como visuales de manera automatizada [Aylien2015]. AYLIEN se encuentra desarrollado con una arquitectura REST que permite el acceso a los servicios a través de métodos HTTP. Además, provee un conjunto de SDKs para plataformas en los lenguajes Python, NodeJS, Ruby, PHP, Java, Go y .NET (C#), cuyo funcionamiento se desarrollará más adelante. A continuación se lista el conjunto de técnicas provistas por la herramienta para poder analizar documentos.

- **Extracción de artículos:** permite extraer el contenido y datos específicos (nombre del autor, contenidos multimedia, etc) de un artículo en un sitio Web eliminando información irrelevante como son los componentes de navegación de un sitio y las publicidades.

- **Extracción de entidades:** permite extraer entidades relevantes de un texto, como pueden ser nombres de personas, lugares, productos y organizaciones, a partir de identificación de patrones.
- **Extracción de conceptos:** permite realizar una identificación de entidades vinculando los conceptos que trata el texto con información adicional de fuentes como puede ser Wikipedia u otras enciclopedias online.
- **Resumen:** permite obtener un resumen del texto a partir de la extracción de un número pequeño de frases claves de un artículo.
- **Clasificación:** permite categorizar el texto a partir de las palabras que lo conforman. La API clasifica el texto en 500 categorías preestablecidas para textos escritos en Inglés. Por ejemplo, deportes, empleo, política, etc.
- **Análisis de sentimientos:** permite detectar sentimientos en un documento en base al análisis de la polaridad de los términos y su subjetividad estableciendo así las intenciones y emociones expresadas por el autor.
- **Sugerencias de hashtag:** permite clasificar publicaciones en las redes sociales mediante el análisis de las etiquetas que cada uno de estos posee. Estas etiquetas, también conocidas como *hashtags*, contienen una o más palabras claves concatenadas sobre los temas a los cuales hace referencia el mensaje, permitiendo así realizar la clasificación del contenido.
- **Detección del lenguaje:** detecta el lenguaje principal de un documento y lo devuelve en formato ISO 639-1.
- **Etiquetado de imágenes:** permite identificar formas y objetos en una imagen devolviendo un listado de palabras junto a su valor de confianza, que indica cuán seguro se encuentra el sistema acerca de la identificación realizada.

Esta herramienta tiene desarrollado cada uno de los módulos antes mencionados para un conjunto determinado de idiomas. Los idiomas son: Alemán, Francés, Italiano, Español y Portugués. Los módulos de extracción de artículos, resumen, clasificación, extracción de entidades, extracción de conceptos, sugerencias de *hashtags* y detección del lenguaje, se encuentran desarrollados para todos los idiomas mencionados anteriormente, mientras que, por ejemplo, el módulo de etiquetado semántico se encuentra implementado sólo para el idioma Inglés.

Para poder hacer uso de los diferentes servicios que esta herramienta provee, existen diferentes maneras dependiendo de la plataforma en la que se desarrolla el sistema. Sin embargo, todas las alternativas comparten el uso de los servicios Web REST. Como fue explicado en el Capítulo 2, los servicios Web son invocados a través de una URL. Si se quiere ejecutar la técnica de detección de lenguaje, se debe indicar el módulo a utilizar y enviar el texto o la URL que tiene el contenido que se desea procesar. Por ejemplo, la URL para evaluar la frase “Juan es un buen jugador de fútbol” sería la siguiente:



Además de esta URL, se deben agregar en la cabecera de la solicitud el ID y la clave de la aplicación en la que se hará uso de estos servicios a modo de autenticación. Por simplicidad, estos parámetros no son mostrados en el ejemplo. Una vez realizada la configuración completa, se podrá realizar la solicitud obteniendo como resultado la siguiente salida en formato JSON:

```
{
 "text": "Juan es un buen jugador de futbol",
 "lang": "es",
 "confidence": 0.99999517941328
}
```

Se puede ver que la herramienta especifica que el texto ingresado esta en el idioma Español con un confiabilidad del 99%. Para ilustrar el modo de uso de los SDKs, a continuación, se puede ver un ejemplo implementado con el kit para PHP. Este kit consta de una carpeta con un conjunto de clases para invocar los servicios de procesamiento de la siguiente manera:

```
$text = "John is a very good football player!";
$textapi = new AYLIEN\TextAPI("5d3293c8", "35990a06e4205d2069cfc09721b9d295");
$sentiment = $textapi->Sentiment(array("text" => $text));
$language = $textapi->Language(array("text" => $text));
echo 'Sentiment: ', $sentiment->polarity, ' (', $sentiment->polarity_confidence, ')' . PHP_EOL;
echo 'Language: ', $language->lang, ' (', $language->confidence, ')' . PHP_EOL;
```

El SDK provee la clase TextAPI, que se inicializa con los datos del usuario y los de la aplicación, permitiendo de esta manera al proveedor mantener un control sobre la cantidad de acceso a servicios realizada. Una vez que se obtiene un objeto del tipo TextAPI, se realizan los llamados a los diferentes módulos que se deseen aplicar. En este caso se realizan dos tipos de análisis: uno de sentimientos y otro de lenguaje. El resultado obtenido se imprime al final obteniendo la siguiente salida:

```
Sentiment: positive (0.99999882727649) Language: en (0.99999768923747)
```

La API permite aplicar las diferentes técnicas NLP disponibles que provee esta herramienta. La herramienta tiene como gran ventaja la posibilidad de invocar los servicios a través del protocolo HTTP.

Su implementación a partir de la arquitectura RESTful da la posibilidad al desarrollador de un sistema hacer uno o varios llamados a servicios NLP sin necesidad de realizar grandes adaptaciones en cuanto a la implementación. Los módulos que esta herramienta posee no tienen una estructura que permita el acoplamiento de varios de los mismos en un solo llamado. Si bien, mediante los SDK es posible combinar módulos a partir de múltiples invocaciones a métodos sobre el mismo texto, en el caso de los llamados HTTP, se vuelve una tarea compleja.

El conjunto de técnicas de análisis que AYLIEN poseé es estático y depende el grupo de los módulos provistos por los desarrolladores. Esto significa que no es posible agregar nuevas técnicas o variantes a las ya existentes. Además, para una tarea específica de procesamiento no existen variantes, limitando así la selección de técnicas a utilizar por el desarrollador a los métodos preestablecidos.

3.1.2. Language Tools

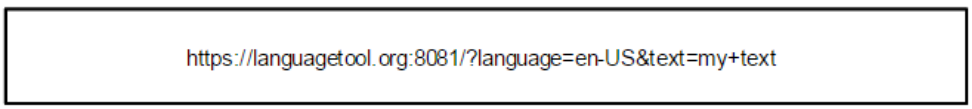
LanguageTools es una herramienta de código abierto que verifica gramática, ortografía y estilo mediante un conjunto de técnicas NLP que permiten identificar errores que los correctores tradicionales no pueden detectar [Language2015]. Esta herramienta se encuentra disponible para múltiples plataformas permitiendo así la adaptación a distintos tipos de aplicaciones.

En el sitio Web de la herramienta, se pueden encontrar add-ons para LibreOffice y OpenOffice, una versión stand-alone y una extensión para el navegador Mozilla Firefox. Cualquiera de estas aplicaciones son para el uso de un usuario final que desea realizar correcciones con el fin de verificar su escritura. Adicionalmente, LanguageTools provee interfaces para que los desarrolladores puedan integrar sus aplicaciones con el motor de análisis gramatical. LanguageTools permite realizar el análisis de texto a partir de un conjunto de módulos desarrollados en Java, entre los cuales se pueden mencionar:

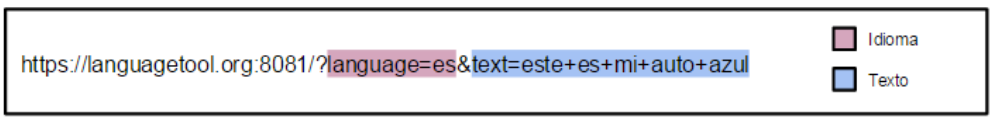
- **AnalyzedSentence**: permite obtener el conjunto de oraciones que constituyen el texto ingresado analizadas y separadas por *tokens*.
- **AnalyzedToken**: permite generar un listado de palabras con el análisis de cada una dentro del contexto de la oración que conforma. Esta tarea se lleva a cabo con técnicas de POS-Tagging e identificadores de lemas.
- **AnalyzedTokenReadings**: permite obtener un arreglo de *AnalyzedTokens* utilizado para almacenar múltiples POS y lemas para una determinada palabra o *token.*
- **JLanguageTool**: permite realizar análisis sobre el texto a partir de diferentes reglas. Estas puedan ser generadas en lenguaje Java, reglas creadas a partir de archivos XML, e incluso reglas implementadas por el usuario.

- **Language**: permite obtener los componentes de análisis necesarios dependiendo del lenguaje original del texto. Por ejemplo, se puede pedir el POS-Tagger de un lenguaje, o un analizador de lemas.
- **MultiThreadedJLanguageTool**: permite el análisis de texto en múltiples hilos para poder alcanzar el resultado de manera mas rapida. Este tipo de análisis se realiza cuando no hay problemas de carga de procesamiento.

Existen otros módulos de análisis dentro de la herramienta como: identificadores de idiomas, segmentadores de texto, anotadores de oraciones, entre otros. Además, cada uno de estos módulos soporta todos los idiomas considerados en LanguageTools. Esta Aplicación permite utilizar los servicios a través de una API Web pública. Los servicios son accesibles mediante peticiones HTTPS y tecnología REST que permite utilizar el corrector. La comunicación con la herramienta mediante el protocolo HTTPS se realiza a través de la siguiente URL:



El servicio corre en el puerto 8081 de languagetool.org, donde se pueden ajustar algunas variables a través del método GET. En primer lugar, se debe especificar el idioma del texto que se desea analizar. En la imagen de arriba se puede observar que trata de un texto en inglés estadounidense "*en-US"*. También, se permite agregar la variable "autodetect=1" para que la herramienta determine cuál es el idioma predominante en el texto de entrada. El modo GET solo debe utilizarse para hacer pruebas porque no es un método seguro de comunicación con el servidor. Los sistemas en producción que requieran seguridad deben utilizar el método POST para transferir información. Una limitación de LanguageTools es que solamente admite texto plano, y no es posible procesar otros formatos como XML, HTML o formatos específicos. A continuación, se puede ver un ejemplo que muestra el funcionamiento de la herramienta:



Para observar de mejor manera el funcionamiento de esta API, se propone analizar la frase "Este es mi auto azul" especificando que el lenguaje es Español (es). Como se puede ver, el texto ingresado presenta errores introducidos adrede para visualizar la forma que tiene la herramienta de indicar problemas y correcciones. El resultado obtenido luego de invocar el servicio REST de LanguageTools es:

```
<matches software="LanguageTool" version="3.1-SNAPSHOT" buildDate="2015-06-29 22:01">
  <language shortname="es" name="Spanish"/>
  <error fromy="0" fromx="0" toy="0" tox="4" ruleId="UPPERCASE_SENTENCE_START"
  msg="Esa frase no se inicia con mayúscula" replacements="Este" context="este es mi
  auto asul" contextoffset="0" offset="0" errorlength="4" category="Mayúsculas y
  minúsculas" locqualityissuetype="typographical"/>
</matches>
```

En el encabezado del XML se encuentran los datos de la versión de la herramienta que se está utilizando y la fecha de compilación. Luego, se ubica la etiqueta “language” que indica el idioma del texto analizado, en este caso, es Español. A continuación, se pueden observar los errores encontrados indicando el inicio y el fin de la cadena que se desea corregir (fromY, toY, fromX, toX), la regla a la cual hace referencia el error (rule-id) y el mensaje de error (msg) correspondiente. En este caso, la anotación indica que la frase no comienza con mayúscula, y por eso sugiere reemplazar la palabra “este” por “Este”. A partir de esta respuesta XML uno puede realizar diferentes acciones según lo requiera una aplicación, o simplemente mostrar el conjunto de errores que la entrada posee.

Dentro de las ventajas que posee LanguageTools se destaca la posibilidad de realizar análisis por múltiples reglas sin necesidad de hacer múltiples llamados indicando en cada uno que modulo se desea invocar. Esto da la pauta que la arquitectura de la herramienta es configurable, ya que se puede seleccionar el conjunto de reglas a ejecutar. Este conjunto de reglas se puede ejecutar de manera secuencial o en simultáneo en diferentes hilos de ejecución. Esto permite una mejora en la performance ya que la carga de trabajo es menor, reduciendo así también el tiempo de análisis.

LanguageTools da soporte para generar nuevas reglas, permitiendo al usuario personalizar uno o más módulos con el fin de extraer la información necesaria. Por ejemplo, en el caso de Java, una clase abstracta que brinda el conjunto de métodos a implementar para poder agregar una nueva regla. En otro de los casos, se puede insertar un archivo XML que posee patrones que describen una regla y de esta manera se marcan las coincidencias dentro del texto. Esto permite al usuario de la herramienta extender la funcionalidad del sistema implementando sus propias reglas. La documentación de la herramienta no explica si existe la forma de realizar reglas incrementales que utilicen información de otra ejecutadas con anterioridad. LanguageTools no permite agregar módulos de análisis externos implicando esto que los módulos de análisis son fijos. Es decir, que cada regla o técnicas de análisis que se desee incluir, debe ser desarrollada completamente por el usuario.

3.1.3. MeaningCloud

MeaningCloud es un SaaS (Software as a Service) que permite a los desarrolladores embeber módulos de análisis de texto y procesamiento semántico en una aplicación de manera sencilla y poco costosa.

MeaningCloud funciona a partir de un conjunto de APIs con funciones estándar que pueden ser combinadas por los desarrolladores [Meaning2015]. Esta herramienta también provee APIs optimizadas para diferentes industrias y escenarios de trabajo. Dentro de estas API, se pueden encontrar diccionarios, taxonomías, y secuencias de procesos específicos para cada aplicación.

Sobre el núcleo de MeaningCloud, se encuentra desarrollada una interfaz mediante la cual se puede acceder a un conjunto de servicios web que permiten hacer uso de las diferentes técnicas de análisis. Los módulos de análisis que provee esta API son los siguientes:

- **Lematización, POS Tagger y Parsing:** permite extraer información acerca de las palabras que conforman un texto.
  - Lematization: obtiene lemas de las palabras que conforman el texto.
  - Parsing: genera un árbol sintáctico a partir de análisis sintáctico y morfológico.
  - POS Tagger: identifica a qué función cumple una palabra en una oración.

  Estos tres módulos son configurables, dando la oportunidad al usuario de obtener la información que necesite.
- **Identificación de Idioma:** permite identificar de manera automáticas el idioma principal del texto. Esta herramienta es capaz de identificar el idioma dentro de un conjunto de sesenta idiomas.
- **Clasificación de Textos:** permite asignar una o más categorías a un documento de acuerdo a su contenido. Esta categorización se realiza a partir una taxonomía preestablecida, con la ayuda de las tareas de preprocesamiento como pueden ser *tokenization, lemmatization*, entre otras.
- **Extracción de Temas:** permite la extracción de diferentes elementos presentes en la fuente de información. Este proceso se lleva a cabo a partir de la combinación de técnicas NLP, identificando así elementos relevantes en el texto.
- **Análisis de Sentimientos:** permite analizar texto para determinar si expresa sentimientos positivos, negativos o neutros. Para lograr esto, se evalúa la polaridad de cada una de las frases del texto y la relación entre las mismas, dando un valor de polaridad global de todo el texto.
- **Corrección de Textos:** permite realizar correcciones de texto automatizado. Este módulo utiliza tecnologías NLP multilenguaje para chequear ortografía, gramática y estilo de texto con gran precisión.
- **Reputación Corporativa:** permite identificar organizaciones dentro de un texto y medir la misma mediante diferentes variables que representan la reputación de la misma. Estos datos son utilizados para generar reportes de cada una de las organizaciones mencionadas en un texto.

Cada una de las clasificaciones de textos se generan a partir de modelos de predicción generados con un conjunto de datos de entrenamiento. Como se puede ver en la Figura 3.1, el proceso de aprendizaje consta de un módulo de entrenamiento, el cual es derivado de un corpus de texto de diferentes categorías, y un conjunto de reglas de clasificación. A partir de esto, se obtiene un modelo de predicción que será aplicado en un módulo de clasificación, el cual responderá al texto que se desee analizar con un listado de categorías con un valor de confianza. Este valor hace referencia a la seguridad con la que la herramienta clasifica un componente del texto. Es posible importar modelos a la herramienta y de esta manera personalizar el clasificador generado a partir del entrenamiento.

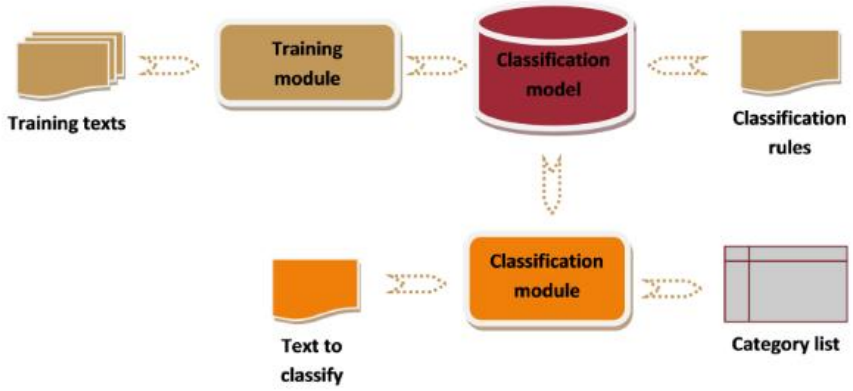


Figura 3.1. Workflow del proceso de clasificación (Meaning Cloud).

Luego de realizar el análisis de un texto, la herramienta devuelve el resultado en formato JSON. Por ejemplo, si se analiza la frase "*The 85th Academy Awards ceremony took place February 24, 2013.*" con el módulo de clasificación de texto, se obtendría una lista de categorías con las temáticas que puede tratar el texto de entrada. Como se puede ver en el siguiente ejemplo, el resultado indica que el texto corresponde con las categorías de "arte, cultura y entretenimiento - premios de entretenimiento" y "interés humano - galardón y premio".


```
{
    "status": { msg: "OK" },
    "category_list": [
        {
         "label": "arts, culture and entertainment - entertainment award"
        },
        {
            "label": "human interest - award and prize",
        }
    ]
}
```

Una de las características más relevantes de esta herramienta es la posibilidad que tiene el usuario de crear sus propios modelos. La herramienta posee un conjunto de modelos predefinidos que permiten analizar y categorizar un texto de acuerdo a las categorías que el modelo reconoce [6]. Pero en el caso que se desee llevar a cabo una categorización personalizada, se puede utilizar la API de clasificación de texto con modelos creados por el usuario. Para poder crear un modelo, es necesario registrarlos con un nombre, una descripción, se debe seleccionar el lenguaje que soporta, y además se puede ingresar un archivo que contenga un conjunto de categorías sobre las cuales se realizará la categorización.

MeaningCloud trabaja con tres tipos de modelos: estáticos, basados en reglas o híbridos. Los modelos estáticos comparan el texto de entrada con los textos de entrenamiento incluidos en el modelo. Los modelos basados en reglas utilizan exclusivamente reglas de clasificación, considerando solamente los términos definidos en el modelo para realizar el análisis. En el caso de los modelos híbridos, como su nombre lo indica, se refiere a la combinación de los dos tipos de modelos anteriores. Un modelo híbrido permite utilizar textos de entrenamiento para cubrir gran parte de la clasificación, con los beneficios de un modelo estático, y aplicando clasificación basada en reglas en los casos en los que el modelo estático es impreciso. Esto trae como ventaja la posibilidad de generar modelos personalizados para identificar los datos que el usuario requiera.

MeaningCloud permite al usuario crear su propio diccionario con entidades y conceptos personalizados para que pueda ser utilizado mediante la API de extracción de tópicos, o el análisis de sentimientos[7]. La precisión de los resultados obtenidos con esta herramienta de análisis se ve limitada por el uso de recursos genéricos, que si bien cubren muchos aspectos a tener en cuenta pueden carecer de información a la hora de realizar análisis en dominios específicos o en términos relevantes para el interés del usuario. La creación de diccionarios impacta directamente sobre los resultados obtenidos de cualquiera de las APIs que lo utilice. Una de las características interesantes de la creación de diccionarios es la posibilidad de generar entradas con tanto detalle como el usuario desee. Los diccionarios están compuesto de dos tipos de entradas: entidades y conceptos. Al igual que para la generación de modelos, es necesario ingresar un archivo que contenga los contenidos con el detalle que usuario desee. Tanto en para la creación de modelos como para la de diccionarios, los archivos ingresados son separados por tabulaciones (tvs, tab-separated values).

MeaningCloud tiene desarrollados SDKs para los lenguajes PHP, Java, Python y VisualBasic. También se puede obtener un cliente NodeJS llamado Kikobeats. Además de proveer soporte para estas plataformas, los llamados a servicios Web públicos se pueden realizar mediante protocolos HTTP o HTTPS a través de métodos POST. Esto permite adaptar cualquiera de los módulos de análisis que esta herramienta provee a un conjunto grande de plataformas de desarrollo.

[6] https://www.meaningcloud.com/developer/resources/models

[7] https://www.meaningcloud.com/developer/resources/dictionaries

Esta herramienta no permite agregar módulos NLP para utilizarlos durante el análisis. Si bien, se destaca la posibilidad de generar reglas propias con diccionarios personalizados por el desarrollador, no se pueden agregar módulos de análisis externos.
Otra limitación es que la comunicación entre componentes de análisis de MeaningCloud no es buena. En cada plataforma se debe realizar un llamado por vez a cada módulo. En el caso de querer recopilar información de diferentes tipos para el mismo texto, se deberán llamar tantos servicios como componentes de análisis se quieran utilizar.

3.1.4. IBM Watson Developer Cloud

Watson es un sistema informático de Inteligencia Artificial capaz de responder preguntas formuladas en lenguaje natural [Arnold2016]. Se destaca por su capacidad de analizar y comprender lenguaje escrito u oral, reconocer información estructurada, publicaciones en redes sociales, y fuentes multimediales como videos, imágenes. De esta manera, la herramienta asimila el material durante todo el análisis del contenido y almacena datos de interés en cada ejecución. Dentro del conjunto de herramientas que se han desarrollado en base a Watson se encuentra Developer Cloud [IBM2015]. Esta API ofrece una variedad de servicios para el desarrollo de aplicaciones cognitivas. Para acceder a cada uno de los servicios, Developer Cloud ofrece una interfaz REST mediante la cual se produce la interacción entre desarrolladores y aplicaciones de Watson. Esta interacción se puede realizar en distintos lenguajes de programación, tales como Java, PHP, Ruby, Python, entre otros.
A continuación, se lista el conjunto de servicios que ofrece Watson, teniendo en cuenta que algunos de estos módulos de análisis se encuentran en una fase preliminar de desarrollo.

- **Expansión de conceptos:** permite determinar el contexto conceptual de un fragmento de texto determinado, proporcionando un conjunto de palabras o frases adicionales que amplían la información que este contiene.
- **Percepción de conceptos:** permite trabajar con conceptos e identificar asociaciones conceptuales dentro del contenido del texto de entrada. Por ejemplo, un texto que habla de "Fútbol" lo vincula con conceptos "Asociación de Fútbol", "FIFA", etc.
- **Traducción de lenguaje:** permite traducir textos a partir de la selección dentro de un conjunto de modelos de traducción que trabajan con técnicas estáticas diferentes. Por ejemplo, permite indicar si es un texto de una conversación o una noticia, y dependiendo de esto realiza la traducción más adecuada para el campo de aplicación determinado.
- **Percepción de personalidades:** permite obtener un perfil de una personalidad determinada a partir del análisis de publicaciones en redes sociales y contenidos en general.

- **Reconocimiento visual:** permite analizar imágenes o capturas de vídeo para extraer información de su contenido.
- **Analizador de tono:** permite realizar análisis lingüísticos para detectar e interpretar señales emocionales que se encuentran en el audio de entrada.

Para hacer uso de estas técnicas, el usuario debe registrarse en una plataforma en la nube llamada BlueMix. Por medio de esta plataforma, se pueden crear, ejecutar y gestionar aplicaciones Web o móviles que necesiten realizar análisis de texto. BlueMix cuenta con un panel de gestión detallado donde se brinda documentación de ayuda para desarrollar un sistema [Gheith2016]. Un ejemplo de la comunicación con la API REST de Watson, puede verse en la Figura 3.2:

```
def POST(self, text=None):
    """
    Send 'text' to the Personality Insights API
    and return the response.
    """
    try:
        profileJson = self.service.getProfile(text)
        return json.dumps(profileJson)
    except Exception as e:
        print "ERROR: %s" % e
        return str(e)
```

```
def getProfile(self, text):
    """Returns the profile by doing a POST to /v2/profile with text"""

    if self.url is None:
        raise Exception("No Personality Insights service is bound to this app")
    response = requests.post(self.url + "/v2/profile",
                     auth=(self.username, self.password),
                     headers = {"content-type": "text/plain"},
                     data=text
                     )
    try:
        return json.loads(response.text)
    except:
        raise Exception("Error processing the request, HTTP: %d" % response.status_code)
```

Figura 3.2. Método POST al servicio Personality Insights.

En dicho código, se puede observar apreciar que el método POST se encarga de llamar a otro método getProfile que hará un pedido al servicio *Personality Insights*. Este llamado requiere de una autenticación y contraseña por parte del usuario que lo quiera utilizar, ya que Watson no es un servicio gratuito. Luego, el servicio retorna contenido JSON con los resultados del análisis de personalidad.

Si bien Watson Developer Cloud es una herramienta muy potente, se pueden mencionar varias limitaciones. Por ejemplo, Developer Cloud no permite realizar llamados que involucren más de un módulo de NLP. Es decir, cada procesamiento es un llamado independiente y no pueden componerse para combinar diferentes técnicas. La API permite introducir módulos de clasificación personalizados a partir de uno de sus servicios, pero no se permite agregar módulos específicos fuera de Watson. Dentro de las ventajas de los módulos personalizados, observamos que trabajan con distintos lenguajes de programación, facilitando la adaptación a distintos proyectos. Watson posee la ventaja de ser una herramienta escalable y, por lo tanto, hace escalables a las aplicaciones desarrolladas bajo su plataforma.

3.1.5. Comparación de APIs NLP

A continuación, se muestra un cuadro comparativo (Tabla 3.1) en el cual presentan diferentes características de las herramientas de forma tal de tener una visión general de los trabajos analizados.

	Aylien	LanguageTools	DeveloperCloud	MeaningCloud
Extensible	X	X	X	X
Escalable	-	-	✓	✓
Personalizable	X	✓	X	X
Plataformas	Python, NodeJS, Ruby, PHP, Java, Go, C#	Java	Python, NodeJS, Ruby, PHP, Java, Go	-
Arquitectura WS	REST	REST	REST	REST
Servicio pago	Free/Pago	Free	Free/Pago	Free/Pago
Código abierto	No	Si	No	No
Acceso público	No	Si	-	No
Formato I/O	Texto plano	Texto plano	Texto Plano	Texto Plano, Markup

Tabla 3.1. Comparación de APIs NLP.

En primer lugar, se hace una evaluación de la arquitectura según la descripción obtenida de la documentación de cada API particular. Dentro del conjunto de herramientas evaluadas, DeveloperCloud y MeaningCloud están diseñadas con una arquitectura escalable. Mientras que para el caso de las dos restantes no se encontró información suficiente para determinar este atributo. La ventaja que tienen todas las herramientas analizadas es la posibilidad de invocarse mediante protocolos HTTP. Esto permite que se puedan hacer múltiples llamados desde diversos tipos de aplicaciones. En cuanto a la extensibilidad de las APIs, ninguna permite agregar componentes de análisis de otro grupo de desarrollo que no sea el propio. Si bien, alguna de ellas permite personalizar modelos, ninguna da la posibilidad de incorporar un analizador externo. Otra dificultad aparece a la hora de componer módulos nuevos a partir de módulos existentes, ya que ninguna de las herramientas lo expone en su documentación. A diferencia de las demás,

LanguageTools, posee interoperabilidad entre sus componentes, de manera que se puede seleccionar un conjunto de los mismos y ejecutarlos tanto secuencialmente como en paralelo. Ninguna herramienta de las evaluadas permite hacer invocaciones a múltiples módulos en una sola petición.

También, para esta comparación, se tuvieron en cuenta otras características importantes a la hora de seleccionar una API para el desarrollo de herramientas NLP. Tanto Aylien como Developer Cloud, poseen diferentes kits de desarrollo para plataformas específicas. Esto es una ventaja de Aylien y DeveloperCloud sobre las otras dos herramientas. En cambio, cuando se trata de accesibilidad al código de la herramienta, LanguageTools es la única cuya implementación puede ser modificada por los usuarios.

Para las demás características analizadas, todas las APIs evaluadas son similares en el sentido que fueron desarrolladas con tecnología REST. Asimismo, dichas herramientas contemplan como entrada documentos en texto plano. Salvo LanguageTools, la cual es una herramienta gratuita, las demás poseen servicios gratuitos y pagos. Los servicios gratuitos brindan un acceso limitado a los servicios que tiene la herramienta en cuanto a cantidad de llamadas por día, o cantidad de módulos de análisis que se pueden utilizar. Las membresías pagas, por lo general, permiten un acceso al uso completo a la herramienta y sin límites de cantidad de invocaciones a servicios.

3.2. Herramientas que utilizan APIs NLP

En esta sección, se analizaron un conjunto de herramientas que hacen uso de distintas APIs para explorar las virtudes que tienen al adquirir procesamiento NLP. El objetivo principal de esta sección, es demostrar el uso masivo de aplicaciones que utilizan el procesamiento de texto, como así también, la variedad de disciplinas que esta tarea abarca. En cada caso, se puede ver la funcionalidad de cada una de las aplicaciones, la API sobre la cual se encuentra desarrollada, y los servicios que utilizan.

3.2.1. Checkmate (LanguageTools)

CheckMate es una aplicación multi-plataforma que permite revisar la calidad de documentos traducidos. A partir de la API provista por LanguageTools para invocar los servicios de análisis, esta herramienta es capaz de realizar verificaciones en textos traducidos tales como el reconocimiento de patrones en el texto original que deben corresponder con la traducción realizada, o fragmentos del texto traducido más cortos o más largos que la fuente original, entre otros. Por otra parte, esta herramienta permite reconocer palabras repetidas, caracteres incorrectos, espacios iniciales y finales, faltantes, etc.

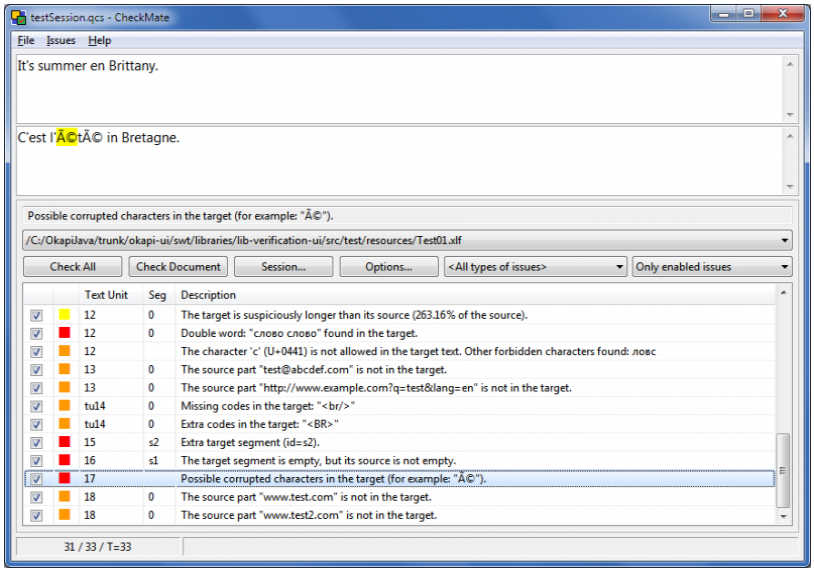


Figura 3.3. CheckMate.

CheckMate permite ingresar textos en todos los idiomas que la API soporta. En la Figura 3.3 se puede ver el análisis de una oración y su traducción realizada con CheckMate. En la parte superior se puede ver el texto original, más abajo su traducción, y por último el listado de sugerencias y correcciones devuelto por LanguageTools.

3.2.2. MeaningCloud Excel Plugin

Los desarrolladores de la API MeaningCloud implementaron un plugin para Excel que permite analizar el texto que se encuentra dentro de la plantilla de cálculo. Este plugin ofrece cuatro tipos de análisis que se pueden realizar al contenido del documento: clasificación de texto, análisis de sentimientos, identificación del lenguaje y extracción de tópicos. En la Figura 3.4, se muestra la salida de un análisis de sentimientos del texto seleccionado por un usuario donde se determina si la polaridad de una frase es positiva, negativa o neutra. Este componente extrae información suficiente para determinar el nivel de afinidad de una frase con el texto, como así también el nivel de confianza, subjetividad e ironía. El usuario podrá configurar el plugin para indicar los datos de interés que tiene, y así exponerlos en una hoja de cálculo.

	A	B	C	D	E	F	G
1	ID	Text	Polarity	Agreement	Subjectivity	Confidence	Irony
2	478558031840419840	I get really sad when I look at the ikea bedroom gallery	N+	AGREEMENT	SUBJECTIVE	98	NONIRONIC
3	478557926798274560	I needa take someone with me to Ikea...	NONE	AGREEMENT	OBJECTIVE	100	NONIRONIC
4	478557799643762688	@Shaggy_locs @LilMissBoojiee yea ikea they not even 5 yet	NONE	AGREEMENT	OBJECTIVE	100	NONIRONIC
5	478557776424103936	RT @arstechnica: IKEA waits 8 years, then shuts down IKEAhackers site with trademark claim http://t.co/pz884KXl6z by @joemullin	N	AGREEMENT	OBJECTIVE	100	NONIRONIC
6	478557735039295488	High school #grads! Already dreaming of #dorm life? Get #ideas for your space	NONE	AGREEMENT	OBJECTIVE	100	NONIRONIC
7	478557662318047232	@JulianCardillo @SeanLDonahue Yeah, that's been there for a while. Curtatone has been coy about why the road got renamed from IKEA Way.	NONE	AGREEMENT	OBJECTIVE	100	NONIRONIC
8	478557555489505280	does anyone else hate how low to the ground ikea malm beds are? 😩	N	AGREEMENT	SUBJECTIVE	100	NONIRONIC
9	478557478729560064	@dandegas1 @MarkHalliwell1 liking the Ikea lamp in the corner.	P	AGREEMENT	SUBJECTIVE	100	NONIRONIC
10	478557308466003968	RT @LeahIsWrong: Happy birthday @iK EA_R! http://t.co/ym2lOudGts	NONE	AGREEMENT	OBJECTIVE	100	NONIRONIC
11	478557177624686593	RT @harprielle96: Look at this! Gorgeous, and tastes even better than it looks! Courtesy of @IKEA_R http://t.co/wUXdFGRx0E	P	AGREEMENT	SUBJECTIVE	100	NONIRONIC
12	478557177225822209	Can't wait to see this new room at Ikea! http://t.co/GyYfuvpa3d	NONE	AGREEMENT	OBJECTIVE	100	NONIRONIC
13	478557166287482880	RT @Natalie_Scala: Excellent news re Ikea coming to Sheffield. Now I won't have to traipse so far for bumper bags of tea-lights! http://t.c…	P+	AGREEMENT	SUBJECTIVE	100	NONIRONIC
14	478557154346299392	@lucytodd96 hahahahhaa I know!! And we are stunnahs.	NONE	AGREEMENT	OBJECTIVE	100	NONIRONIC
15	478557110767480832	Saw this kitchen at Ikea Sunday. I think I'll just live here, thank you. #Love	P+	AGREEMENT	SUBJECTIVE	100	NONIRONIC
16	478557103368699904	£60m Ikea store set for Sheffield.	NONE	AGREEMENT	OBJECTIVE	100	NONIRONIC
17	478557084816904192	IKEA waits 8 years, then shuts down IKEAhackers site with trademark claim	N	AGREEMENT	OBJECTIVE	100	NONIRONIC
18	478557084582420480	On hold to bloody IKEA and they put mr brightside on🔧 not so angry anymore	P	DISAGREEMENT	SUBJECTIVE	98	NONIRONIC
19	478556996971425794	@w_stanley Since this is a UK order, we're unable to access any of your records. IKEA UK can be reached here: http://t.co/V8zpU6wunW So sry!	NEU	DISAGREEMENT	OBJECTIVE	100	NONIRONIC

Figura 3.4 Salida del análisis de sentimientos.

3.2.3. Aylien Google Sheets Plugin

Otra herramienta interesante es un plugin para Google Sheets (planilla de cálculo de Google Drive) que utiliza los servicios NLP que provee la estructura AYLIEN. Este complemento permite aplicar técnicas de clasificación y resumen de documentos y artículos, análisis de sentimientos y extracción de entidades, y contextos para seis idiomas diferentes. Con esta herramienta acoplada a la hoja de cálculos de Google, se pueden extraer datos de interés, dando así la opción al usuario de insertar el contenido en gráficos para poder llevar a cabo un análisis más completo. En la Figura 3.5, se puede observar el modo de uso de la herramienta. El servicio de AYLIEN que permite clasificar el texto que contiene la URL indicada en la columna URLs. En este ejemplo, se ingresa un conjunto de URLs para determinar la temática del contenido de cada una. Una vez analizado, el resultado se puede ver en una columna del documento. Como se destacó anteriormente, en la Figura 3.5 se puede ver un gráfico generado a partir de los resultados provenientes del análisis.

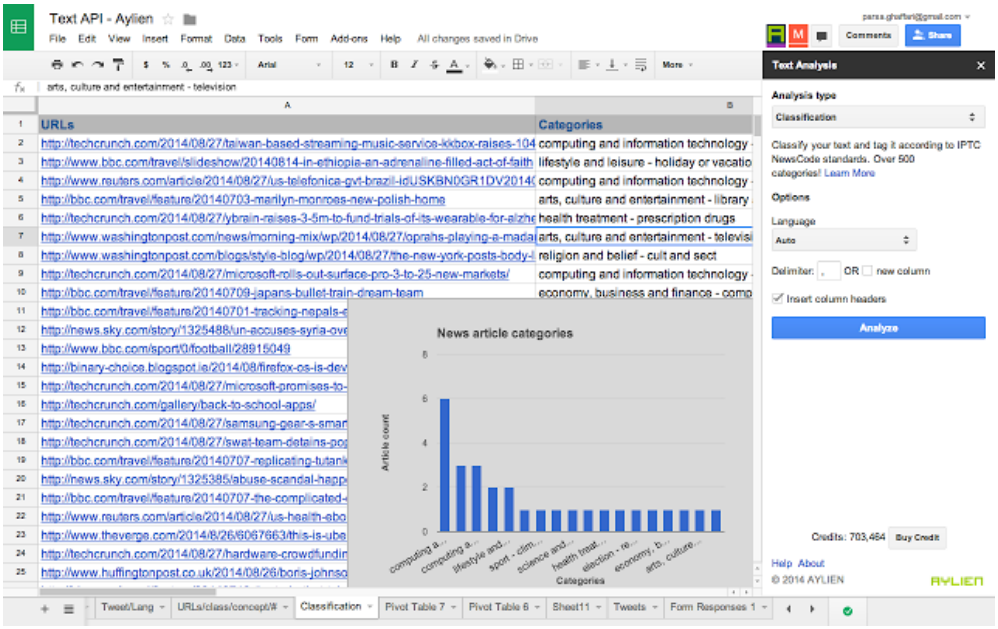


Figura 3.5. Complemento para Google Sheets.

3.2.4. Cognitive Head Hunter

A partir de la API Watson Developer Tools se han desarrollado aplicaciones cognitivas para diversos campos de aplicación. Una aplicación simple es Cognitive Head Hunter, mediante la cual se puede buscar trabajo cargando el perfil del usuario. También se puede agregar el vínculo a la red LinkedIn para que la herramienta obtenga más información y pueda dar una respuesta precisa.

Para utilizar la herramienta, esta requiere una descripción del perfil del usuario. A partir de estos datos se hace uso de los servicios *Cognitive Insights* y *Personality Insights* de Watson para encontrar un trabajo. *Cognitive Insights* captura los datos del texto de entrada y los inserta en un gráfico conceptual basado en Wikipedia. El servicio identifica vínculos a los conceptos mencionados. Mientras que el servicio *Personality Insights* extrae y analiza un espectro de atributos de diferentes personalidades para ayudar a descubrir características de personas y entidades. *Cognitive Head Hunter* hace uso de estos servicios, ingresando los datos obtenidos en gráficos como se pueden ver en la Figura 3.6 A partir de estos datos, la herramienta sugiere dentro de un conjunto de empleos ingresados en el sistema, cuales son los más apropiados para el cliente.

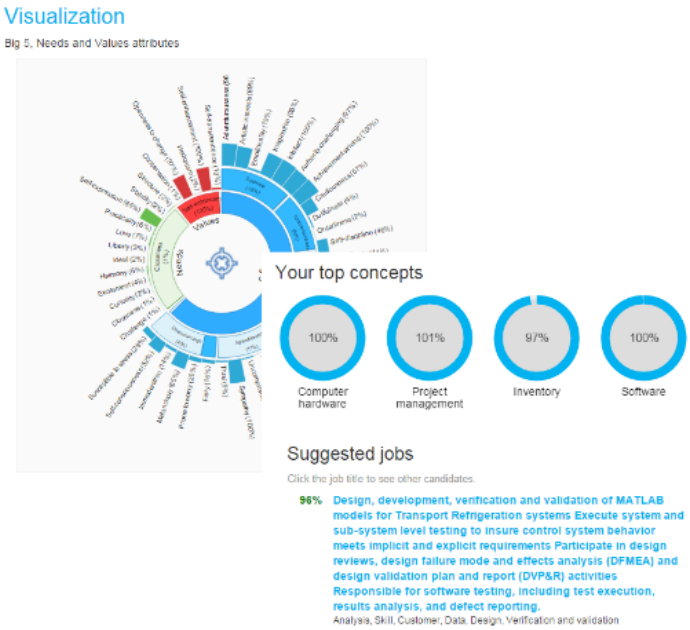


Figura 3.6. Cognitive Head Hunter.

3.2.5. MD Anderson's Oncology Expert

Una aplicación de mayor complejidad es la desarrollada por el centro médico de lucha contra el cáncer MD Anderson sobre la plataforma IBM Watson, denominada MD Anderson's Oncology Expert™. Esta herramienta es utilizada para analizar decisiones clínicas tomadas por los médicos a partir de la evaluación de documentación almacenada de pacientes anteriores. Este análisis es llevado a cabo mediante el proceso del texto de las historias médicas con Watson.

IBM Watson hace uso de los componentes de proceso de texto para identificar cuales son las características de la enfermedad que tiene un determinado paciente. A partir de estos datos, realiza una comparación con los casos anteriores para poder obtener un tratamiento específico que se realizó para un caso similar. Además, se tienen en cuenta los análisis realizados al paciente actual, para poder tener en cuenta el trabajo que se viene realizando hasta el momento. Esta herramienta no es utilizada solamente por el centro médico, sino que se puede utilizar como teleconsulta para obtener una segunda opinión al respecto de la patología.

El análisis de texto abarca muchos componentes de NLP provistos por Watson. De esta manera, se obtiene una asociación más precisa con otros diagnósticos. Cada uno de los diagnósticos ingresados queda almacenado para poder ser utilizado más adelante.

En la Figura 3.7, se puede observar una captura de pantalla de la aplicación. Como se dijo anteriormente, a partir de los datos suministrados por el paciente, se buscan casos similares para poder ver las historias médicas que se encuentran en la base de datos. A partir de eso, en la pantalla principal se sugieren

distintos tipos de terapias ponderadas a partir de los conocimientos obtenidos del análisis de textos realizado por la plataforma de IBM.

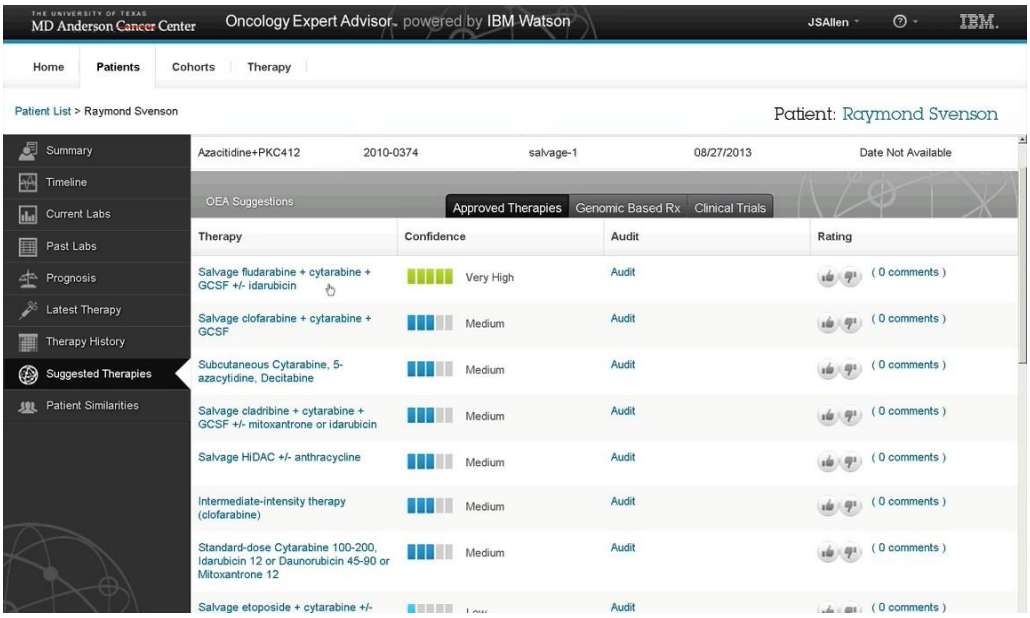


Figura 3.7. MD Anderson's Oncology Expert.

3.2.6. ClearTK

ClearTK no es una aplicación como las analizadas a lo largo de este capítulo, ya que es un framework de desarrollo [Ogren2008]. Este framework basado en UIMA permite realizar análisis de contenidos mediante el aprendizaje automático y el procesamiento NLP. Este framework provee interfaces para acceder a bibliotecas de aprendizaje automático como son SVMlight, LIBSVM, LIBLINEAR, OpenNLP MaxEnt, y Mallet.

Otra característica de este framework es utilizar múltiples bibliotecas de extracción de información que pueden ser usadas con cualquiera de los clasificadores de aprendizaje automático que la herramienta provee. ClearTK interpreta cada una de estas bibliotecas, pudiendo así realizar la traducción de la salida obtenida a un formato apropiado para el modelo que esté siendo usado por el usuario del framework. Dentro de los componente de análisis NLP provistos en ClearTK se encuentran Snowball Stemmer, OpenNLP Tools, MaltParser Dependency Parser, y Stanford CoreNLP. Además, provee un conjunto de módulos de UIMA dentro de los cuales se encuentran Penn Treebank, ACE 2005, CoNLL 2003, Genia, TimeBank y TempEval. ClearTK es de uso frecuente en la actualidad ya que contiene un conjunto de componentes de análisis que se pueden acceder desde cualquier aplicación y obtener un resultado sencillo de interpretar.

3.2.7. Resumen

Dentro del variado conjunto de aplicaciones que hacen uso de servicios de procesamiento de texto, se puede observar los diferentes campos de aplicación como así también las plataformas para las cuales se desarrollan. Esto expone la necesidad de unificar un conjunto de componentes de análisis de uso común que puedan ser accedidos tanto desde una aplicación de escritorio, de una aplicación o sitio Web, como así también desde aplicaciones móviles.

Para cada una de las herramientas mencionadas anteriormente existen necesidades particulares en cuanto a componentes de análisis se refiere. Por ejemplo, las herramientas que realizan correcciones de texto, necesitan menos analizadores que la herramienta desarrollada por el instituto MD Anderson de lucha contra el cáncer. Esta última debe realizar una relación entre conjuntos grandes de informes médicos y encontrar asociaciones precisas en los mismos para poder alcanzar un resultado lo más exacto posible. La necesidad de una estructura configurable para poder unificar el uso de servicios de procesamiento de texto es otra de las conclusiones alcanzadas mediante el análisis realizado.

La composición de módulos NLP es otra de las dificultades existentes a la hora de hacer uso de las herramientas para el procesamiento de texto. Algunas herramientas permiten agregar módulos nuevos pero ninguna hace referencia a la composición de módulos a partir de los que ya existen. Los módulos nuevos que deseen generarse deben cumplir con el formato indicado por cada una de las herramientas. Esto permite aumentar la cantidad de módulos y la información que se extrae con cada uno de estos dentro de las limitaciones que posee cada proceso de desarrollo. Por ejemplo, si las nuevas reglas deben ser descriptas mediante XMLs, el usuario deberá adaptar su modelo al formato requerido por la herramienta.

Cada uno de los módulos provistos por las diferentes herramientas poseen escasas variables parametrizables. En la mayoría de los casos, la parametrización viene dada por la seguridad de los módulos, obligando que para el uso correcto de la herramienta se envían las credenciales de usuario y/o aplicación a la cual responde. Esta es una limitación, ya que podría existir un número mayor de variables parametrizable conociendo la cantidad de variantes existentes sobre un mismo módulo de análisis.

La escalabilidad es otro atributo que no es tenida en cuenta todas las herramientas de procesamiento analizadas. En este caso, DeveloperCloud de Watson y MeaningCloud son las dos herramientas que poseen una arquitectura escalable. En alguno de los casos esta característica varía de acuerdo al servicio solicitado por el usuario (pago o gratuito).

Ninguna de las herramientas desarrolladas en este capítulo parece permitir el agregado de nuevos módulos de análisis ya desarrollados. Por ejemplo, si StanfordNLP desarrollara un nuevo clasificador de texto, ninguna de estas herramientas permite agregarlo para su posterior uso del mismo. Esta es una limitación importante ya que el desarrollo de técnicas y módulos en este área es constante.

Capítulo 4 - Herramienta TeXTracT

Las demandas actuales del mercado de desarrollo de software requieren que muchas aplicaciones tengan que contar con la habilidad de procesar y comprender texto escrito en lenguaje natural (dichas aplicaciones son denominadas NLP). En la práctica, los desarrolladores deben dedicar tiempo y esfuerzo para codificar este tipo de características en los productos, creando la necesidad de re-utilizar herramientas específicas para este propósito. Desafortunadamente, la gran mayoría de dichas soluciones presentan limitaciones que dificultan el desarrollo rápido y la evaluación de aplicaciones. Entre los problemas más importantes, se destacan los siguientes. Primero, cada una de las herramientas descritas en el capítulo anterior son poco configurables, debiendo desarrollar para cada una de las aplicaciones que realicen procesamiento NLP una adaptación particular. Segundo, ninguna de las herramientas NLP existentes permite la composición de módulos NLP, siendo esta una gran ventaja a la hora de personalizar el proceso para extraer la información necesaria. Tercero, los módulos provistos por las herramientas no son por lo general parametrizables, restringiendo el uso de estos. Por último, ninguna de las herramientas NLP desarrolladas hasta el momento permite el agregado de nuevos módulos de análisis. Esta es una desventaja importante teniendo en cuenta que los módulos de NLP se actualizan constantemente, incorporando mejoras a sus modelos y algoritmos. Por lo tanto, en dominios de aplicación variados se requiere trabajar con las últimas novedades de la literatura.

Una alternativa interesante para resolver estas limitaciones consiste en desarrollar una solución que permita: (i) incorporar, configurar y ejecutar técnicas de NLP desarrolladas por terceros, (ii) componer técnicas de NLP para realizar análisis de texto punta-a-punta personalizados para cada aplicación, y (iii) exponer las técnicas de NLP para que puedan ser consumidas por diversas aplicaciones y desde diferentes plataformas.

4.1. Propuesta

Teniendo en cuenta las limitaciones encontradas en la literatura, se construyó una herramienta denominada TeXTracT, la cual tiene como objetivo simplificar la integración de tecnologías NLP en el desarrollo de aplicaciones Web y standalone en distintos tipos de plataforma. TeXTracT está desarrollada en base a tres pilares, a saber: la interoperabilidad de la herramienta, la adaptación de los pipelines de procesamiento de texto, y la flexibilidad respecto a la incorporación de técnicas de NLP.

En primer lugar, se decidió aprovechar algunos de los lineamientos definidos en el patrón *Broker*, de forma tal uniformizar y simplificar la comunicación entre las aplicaciones y los diferentes servicios de procesamiento NLP provistos por TeXTracT. Esta idea surge de la necesidad de contar con una

herramienta que pueda interpretar mensajes de aplicaciones en los cuales se indica el pipeline de proceso que se desea aplicar.

Otro aspecto que se tuvo en cuenta para satisfacer las necesidades de los desarrolladores de aplicaciones NLP, fue que se pueda listar los componentes de NLP de manera dinámica, con el objetivo de poder agregar y reconocer módulos nuevos sin necesidad de recompilar o detener la ejecución de TeXTracT. Esta funcionalidad permite hacer un descubrimiento de los servicios disponibles y presentarlos a la aplicación que lo solicite.

Asimismo, para lograr una herramienta interoperable, se tomó la decisión de desarrollar una API accesible mediante servicios Web de forma tal de que las técnicas de NLP puedan ser invocadas desde cualquier aplicación, independientemente de la tecnología, lenguaje, o sistema operativo utilizado en su desarrollo.

TeXTracT permite personalizar las solicitudes de procesamiento, de manera tal que sea posible agregar o configurar un módulo de procesamiento de NLP. Por ejemplo, si se desea utilizar un modulo de POS Tagging dentro de un pipeline de proceso que invoca servicios NLP existentes en la herramienta, es posible especificar otro modelo que describa el funcionamiento del mismo. En la invocación a este módulo de POS Tagging, se agrega un parámetro con el directorio de este nuevo modelo en particular. Esta parametrización de módulos, posibilita el uso de distintas versiones de un módulo según el parámetro que se utilice durante el llamado.

4.2. Arquitectura de la Herramienta

Durante el desarrollo de TexTRacT, se tomaron diferentes decisiones de diseño para abordar las características definidas anteriormente. Dichas decisiones quedan reflejadas en la arquitectura ilustrada en la Figura 4.1. En la misma, se puede ver un diagrama de componentes arquitectónicos en el cual se pueden apreciar la organización de los elementos que componen la herramienta y las conexiones entre los mismos.

Como se mencionó anteriormente, se aprovecharon algunas de las características principales del patrón de diseño *Broker* para organizar los componentes [Bass2012]. Específicamente, se desarrolló un componente del mismo nombre que actúa como intermediario entre las aplicaciones que requieren procesar texto y los módulos de NLP particulares. Desde un punto de vista arquitectónico, este componente define un protocolo de comunicación Web independiente de las tecnologías de implementación de las aplicaciones o de las técnicas de NLP. Este tipo de organización también establece que las técnicas de NLP individuales son consideradas módulos independientes que pueden ser invocadas desde las aplicaciones. Asimismo, el componente provee funcionalidad para el descubrimiento de las técnicas de NLP disponibles y permite encadenar su ejecución a pesar de que sean módulos independientes y que no se

conizcan entre sí. Desde una perspectiva funcional, este componente tiene la responsabilidad de recibir las solicitudes de procesamiento de una aplicación determinada, coordinando la secuencia de análisis, configurando las técnicas de NLP según los requerimientos de la aplicación y enviando los llamados de ejecución a las mismas, recolectando los resultados para ser devueltos en un formato estandarizado. De esta manera, se disminuye el grado de acoplamiento entre la herramienta y la aplicación que hace la invocación a los servicios. El componente de comunicación recibe un mensaje y realiza el proceso independientemente del tipo de aplicación que lo haya invocado.

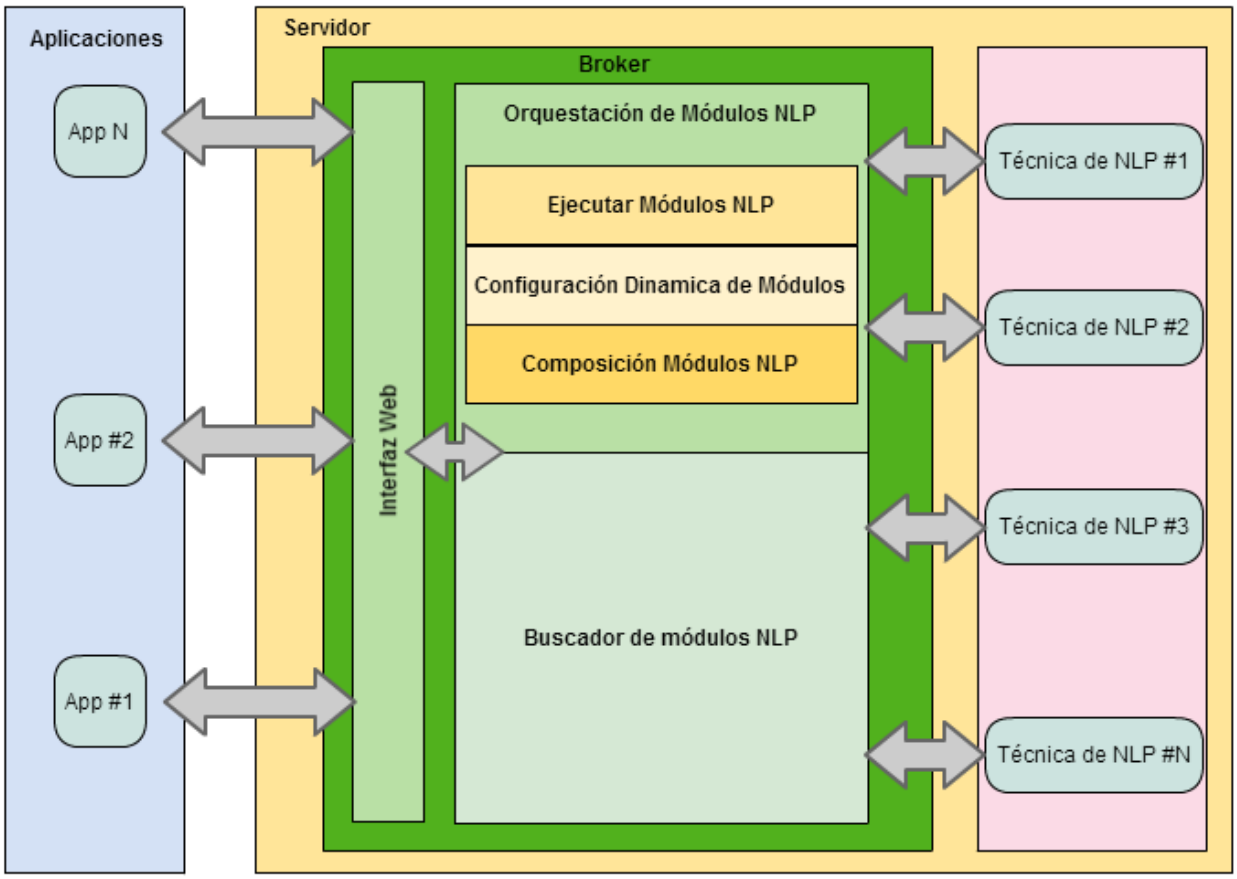


Figura 4.1. Arquitectura Conceptual de TexTRacT.

Una de las responsabilidades más importantes del componente *Broker* es aquella de orquestar la secuencia de ejecución de técnicas NLP de acuerdo a las necesidades de cada aplicación específica. Esto significa que TexTRacT debe encargarse de la ejecución y la conexión dinámica de cada una de las técnicas de NLP solicitadas, combinando sus entradas y salidas de forma tal componer un pipeline secuencial de procesamiento de texto que permite realizar un análisis de texto personalizado de punta a punta. Otro aspecto central considerado durante el diseño de la arquitectura fue permitir que las aplicaciones puedan adaptar las técnicas de NLP, como por ejemplo definiendo un modelo de aprendizaje para identificar etiquetas POS específicas para un dominio particular y diferente al provisto por defecto en los algoritmos. Esencialmente, esta adaptabilidad frente a los cambios tiene como propósito posibilitar la alteración del

comportamiento de un módulo individual de procesamiento de texto sin necesidad de realizar modificaciones en el código fuente. Para lograr este tipo de flexibilidad, se definió el componente *Configuración Dinámica de Módulos*, el cual tiene la responsabilidad de abstraer las diferentes formas de configurar las técnicas de NLP mediante el pasaje de parámetros. Cada módulo puede encadenarse con otros, utilizando la información extraída previamente para generar un conjunto nuevo de anotaciones. Por ejemplo, el módulo encargado de identificar dependencias, hace referencia a los *tokens* identificados por el anotador correspondiente.

Una funcionalidad importante en la arquitectura es aquella que permite listar los módulos disponibles para el análisis. Con este propósito, se definió el componente *Buscador de Módulos NLP*, el cual tiene la responsabilidad de descubrir las técnicas de NLP que se encuentran cargadas en la herramienta. Una característica relevante del buscador es que permite encontrar las técnicas de NLP de forma dinámica, listando en cada llamado aquellas técnicas disponibles en ese momento en el cargador de clases. Por dicha razón, se decidió utilizar Java como tecnología de implementación. Esto significa que es posible agregar nuevas técnicas de NLP o remover aquellas previamente cargadas sin necesidad de alterar el código de TexTRacT, como así también sin que sea necesario reiniciar la aplicación.

4.3. Tecnología

Para instanciar la arquitectura de TeXTracT, se seleccionaron diversas tecnologías las cuales simplifican su desarrollo y permitieron materializar el diseño propuesto. En esta sección, se mostrará en detalle los componentes utilizados durante la implementación de la herramienta de procesamiento Web.

4.3.1. Apache UIMA

A raíz de la investigación de tecnologías para la manipulación y análisis de texto, se decidió utilizar el framework Apache UIMA (Unstructured Information Management) como base para la abstracción de las técnicas NLP y como mecanismo de comunicación que uniformiza las salidas y entradas de NLP. UIMA es una infraestructura que permite el análisis de información extrayendo datos específicos a partir del contenido de los documentos. UIMA es un proyecto de código abierto preparado para analizar grandes volúmenes de información no estructurada (texto, audio, vídeo), con el fin de descubrir y organizar lo relevante para el usuario final o la aplicación [UIMA2015].

Para comprender la función de los componentes UIMA involucrados en el procesamiento NLP, a continuación se describe su flujo de trabajo. Inicialmente los datos no estructurados son analizados de forma tal de detectar conceptos de interés, como pueden ser nombres de personas, organizaciones,

productos, entre otros. Estos elementos identificados son almacenados en una estructura que contiene toda la información obtenida.

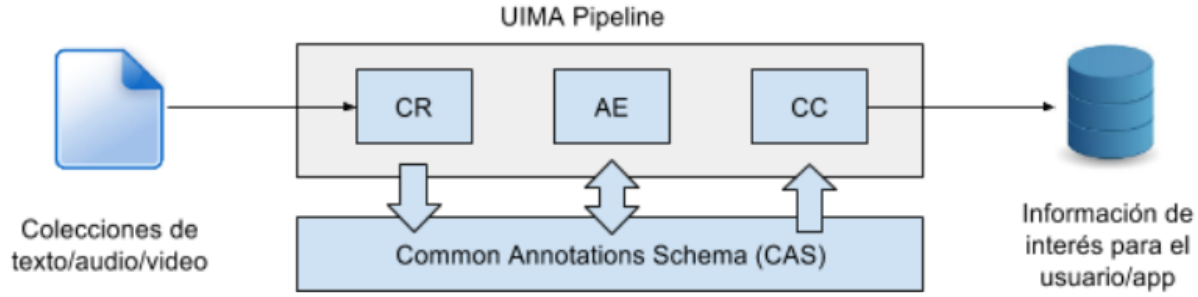


Figura 4.2. Flujo de trabajo UIMA.

En la Figura 4.2 se puede apreciar los diferentes componentes que forman parte de la arquitectura de procesamiento de texto de UIMA. Particularmente, los componentes de este proceso son:

- **Análisis Engine (AE):** Este componente es el encargado de analizar la estructura del texto y registrar meta-información del documento. Cada AE realiza el análisis sobre los atributos descriptivos del documento de entrada y agrega información resultante del análisis en forma de anotaciones sobre el texto. Un conjunto de AEs interconectados secuencialmente permiten realizar analisis de texto más complejos de forma incremental.
- **Common Analysis Structure (CAS):** Este componente es una estructura de datos que permite la representación de propiedades y valores, denominados anotaciones en la jerga de UIMA. El CAS contiene el documento que se analiza y todas las anotaciones que produce cada uno de los *Analisis Engine*.
- **Collection Reader (CR):** Este componente es el encargado de leer información de fuentes variadas, como pueden ser documentos de texto plano, fragmentos de audio o base de datos. Tiene la responsabilidad de inicializar el CAS con el contenido del documento a procesar. A través de cada CR la herramienta puede interpretar el contenido de la entrada según su codificación particular y así poder realizar el análisis correspondiente.

Una ventaja de UIMA aprovechada en TeXTracT es que permite generar cada AE como una aplicación independiente que realiza las actividades correspondientes a una técnica NLP. Esto fue posible a partir de la utilización de las interfaces provistas, que permiten uniformizar la entrada y la salida de los anotadores, creando así componentes de análisis desacoplados del resto. Al ingresar el texto que se desea analizar, se escribe en el CAS y se procesa por un AE que agrega las anotaciones generadas en la misma estructura.

Una vez terminado el procesamiento, se serializa el contenido del CAS de manera tal de que pueda ser interpretado por la aplicación que solicitó el procesamiento. Asimismo, en TeXTracT también se aprovecharon los CollectionReaders de UIMA. En este caso, se implementaron dos CR para interpretar el texto de entrada: un lector de texto plano y un lector de anotaciones UIMA. En cuanto a la salida, UIMA da soporte para obtener el contenido del CAS en formato XML. Estos componentes simplifican la comunicación de TeXTracT con las aplicaciones que lo invocan, como así también, la comunicación entre anotadores.

Los AE pueden generar una nueva anotación, o agregar un atributo a una anotación existente. Por ejemplo, el AE que identifica cada palabra del texto ingresado genera una anotación nueva en el CAS marcando su posición en el documento. Mientras que un AE que identifica partes del discurso no genera una nueva anotación, sino que agrega un atributo a cada anotación de palabra que indica qué función sintáctica cumple en el texto. En base a las técnicas NLP más utilizadas, se definieron un conjunto de anotaciones que permiten registrar dicha información independientemente del algoritmo usado. En la Tabla 4.1 se puede ver la descripción de cada una de las anotaciones utilizadas en TeXTracT.

Anotación	**Proposito**	**Atributos**
Sentence	Oraciones dentro del texto de entrada.	-Begin: # carácter de inicio de oración. -End: # caracter final de oración.
	Por ejemplo: My name is John. **<Sentence begin=’17’ end=’32'>**I wrote a book**.</Sentence>** I am very happy.	
Token	Palabras individuales dentro de las oraciones y sus propiedades.	-Lemma: lema de la palabra. -Pos: función de la palabra dentro de la oración.
	Por ejemplo: My **<Token begin=’3’ end=’7’ lemma=’name’ pos=’NN’ >**name**</Token>** is John. I wrote a book**.** I am very happy.	
Chunk	Fragmento de texto con una función sintáctica en la oración.	-Chunk: tipo de fragmento (por ejemplo frase sustantivo, verbal, etc).
	Por ejemplo: <Chunk chunk=’NP’ begin=’0’ end=’7’ >My name**</Chunk>** is John. I wrote a book**.** I am very happy.	
CoNLLDependency	Relaciones sintácticas entre palabras basada en el modelo de Conference on Natural Language Learning.	-Relation: tipo de relación(por ejemplo sujeto de, objeto de, etc). -Source: ID del Token base de la relación. -Target: ID del Token objetivo del atributo *Source*.

	Por ejemplo: My name is John. **<CoNLLDependency relantion='SBJ' source='105' target='94'><Token id='94'>** I **</Token><Token id='105'>** wrote**</Token></CoNLLDependency>** a book**.** I am very happy.	
Argument (SRL)	Expresión o elemento sintáctico en una frase que cumple un rol específico respecto al predicado.	-Label: tipo de argumento (por ejemplo A0, A1, etc). -Root: Token raíz del argumento. -Description: definición del rol jugado en la oración.
	Ej: My name is John. **<Argument id='316' label='A0' root='94' description='writer'>**I**</Argument>** wrote **<Argument label='A1' root='116' description='thing written'>**a book**</Argument>.** I am very happy.	
Predicate (SRL)	Representa la acción principal de la oración.	-Label: base del predicado(variante del verbo). -Root: *token* raíz del fragmento de texto. -PassiveVoice: es voz pasiva. -Description: significado del predicado.
	Ej: My name is John. I **<Predicate label='write.01' root='105' description='set pen to paper' passiveVoice='false'>**wrote**</Predicate>** a book**.** I am very happy.	

Tabla 4.1. Anotaciones UIMA.

Para poder automatizar la utilización de UIMA a lo largo del proyecto, se utilizó una biblioteca de Java denominada UIMAFit la cual simplifica la instanciación de los componentes (anotadores, lectores de colecciones, etc.) sin que sea necesario utilizar archivos de configuración estáticos (XML) [UIMAFit2015]. Los descriptores que especifican la estructura de los componentes UIMA son generados dinámicamente, permitiendo tratar los componentes como objetos y facilitando su manipulación. Todos los anotadores utilizados en TeXTracT utilizan la biblioteca UIMAFit.

4.3.2. Jersey

Para poder exponer los servicios NLP de la herramienta como servicios Web se exploraron distintas alternativas con respecto a la tecnología a utilizar. La forma tradicional de declarar servicios Web es SOAP, un protocolo estándar que permite definir cómo dos objetos en diferentes procesos pueden comunicarse por medio del intercambio de datos XML. Por otro lado, se encontró una manera más simple y dinámica para declarar los servicios Web denominada REST (Representational State Transfer). Este protocolo permite declarar una interfaz entre sistemas que utilicen directamente HTTP para obtener datos o indicar la ejecución de operaciones sobre los datos en cualquier formato (XML, JSON, etc). Además, es un protocolo *stateless* o sin estado, es decir que el servidor no almacena información del estado del cliente, dando la posibilidad de dar servicio a cualquier cliente en cualquier momento. Una diferencia

notable entre estos dos protocolos la dificultad para definir cada uno de ellos. Para definir un servicio Web SOAP es necesario definir un conjunto de componentes del mensaje como son la envoltura, el encabezado y el cuerpo del mensaje. Por el contrario, un servicio REST se define directamente con URL accesible a través del protocolo HTTP, sin necesidad de declarar componentes adicionales. Debido a la necesidad de la arquitectura propuesta para TeXTracT, este fue el protocolo seleccionado.

La implementación de los servicios REST se basó en la tecnología definida en Jersey. Jersey es un framework de código abierto creado con el fin de simplificar el desarrollo de clientes en Java que consuman y provean este tipo de servicios REST [Jersey2015]. Jersey cuenta con una API compatible con JAX-RS (Java API for RESTful Web Services), la cual es estándar y portable. Esta API permite aprovechar las anotaciones de código Java con el fin de facilitar el desarrollo y el despliegue de los cliente en los puntos finales del servicio. Este framework facilitó en gran parte el desarrollo de los servicios Web de TeXTracT, adaptándose a las necesidades planteadas.

4.4. Workflow de la Herramienta

Al utilizar TeXTracT, las aplicaciones consumidoras deben seguir un protocolo de intercambio de mensajes (es decir, un flujo preestablecido de información) para coordinar y ejecutar los módulos de NLP requeridos. La Figura 4.3 presenta un diagrama de secuencia donde una aplicación hace uso de los servicios NLP expuestos en TeXTracT.

El *Broker* se encarga de la comunicación entre los módulos de análisis y la aplicación que los demanda. Inicialmente, la aplicación envía un mensaje a través de un servicio Web al *Broker* solicitando listar los componentes NLP disponibles en el servidor. Para obtener el listado de los componentes de manera dinámica, TeXTracT utiliza una API de Java denominada Reflection. Esta API permite inspeccionar y manipular clases e interfaces en tiempo de ejecución, sin conocer a priori los tipos y/o nombres de las clases específicas con las que está trabajando. Los módulos NLP se encuentran encapsulados en *Adapters* para poder identificarlos en tiempo de ejecución como componentes de análisis. Entonces, el *Broker* solicita la información a cada uno de los *Adapters* existentes en el sistema, y a partir de esta información extrae los datos del módulo NLP al que contiene. La principal ventaja de usar dicha API es que facilita la incorporación de módulos en tiempo de ejecución.

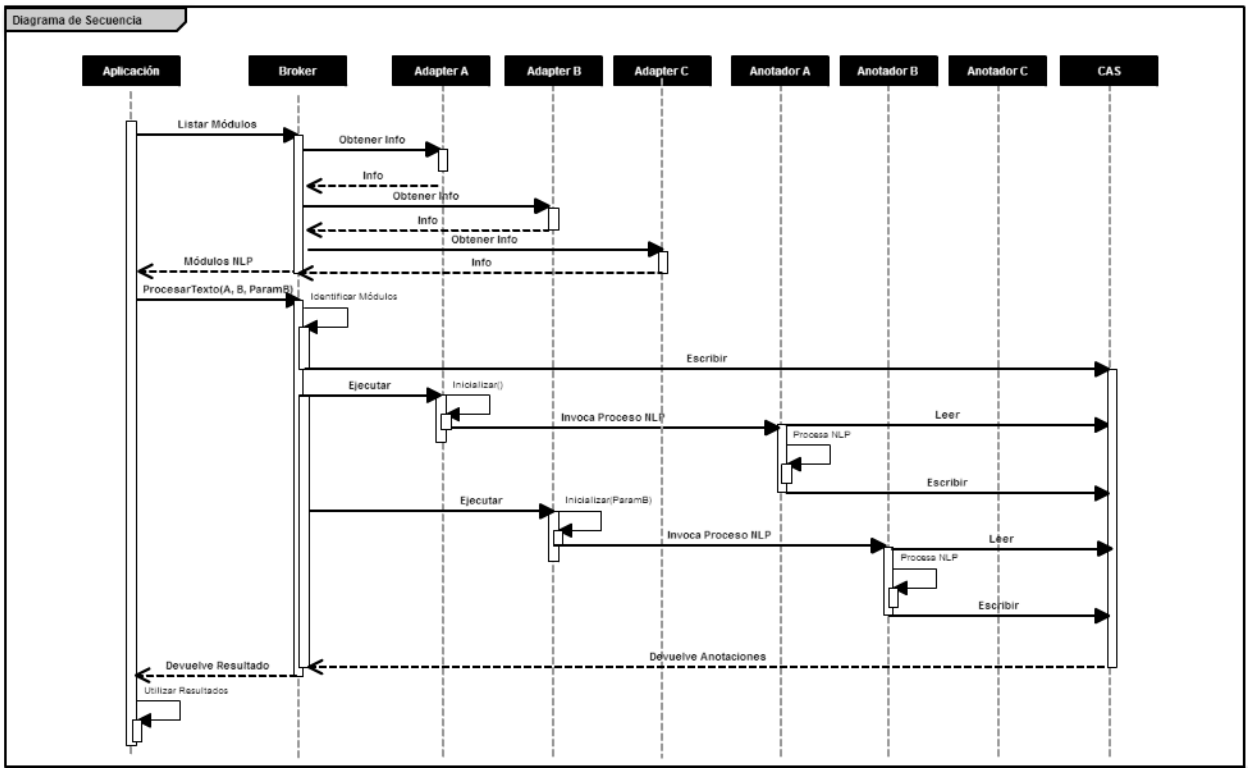


Figura 4.3. Diagrama de secuencia del flujo estandar para la invocación de servicios NLP expuestos en TeXTracT.

Una vez que se obtiene la lista de módulos NLP, la aplicación determina el subconjunto de módulos NLP que necesita para cumplir sus objetivos. Esta secuencia se especifica a través del servicio Web en el que se solicita el procesamiento mediante una URL, indicando el nombre de cada uno de los componentes de proceso requeridos. En esta misma solicitud también se envía el texto sobre el cual se va a realizar el análisis mediante información codificada en el request POST. En el diagrama se puede ver el mensaje "Procesar(A, B, ParamB)" que indica que se requiere ejecutar de los módulos NLP A y B secuencialmente. En este mensaje se ve también el "ParamB" que representa una configuración particular para el segundo módulo solicitado. Esto sirve para manejar variantes de las técnicas de NLP, como por ejemplo, el modelo utilizado para extraer dependencias entre palabras, o el diccionario que se debe utilizar para extraer el significado de una palabra en el contexto de una oración. El *Broker,* a partir de estos datos, se encarga de identificar los módulos solicitados por la aplicación. Luego, el *Broker* inicializa el CAS sobre el cual se van a agregar las anotaciones resultantes del análisis de texto. Posteriormente, se comienza con la secuencia de análisis enviando un pedido de ejecución al adaptador del primer módulo NLP, en este caso, el *AdapterA*. Este adaptador inicializa el módulo de análisis al que hace referencia, e invoca el procesamiento NLP correspondiente. El módulo, en este caso *AnotadorA*, lee el contenido del CAS, realiza el procesamiento correspondiente a la técnica de análisis y escribe los resultados obtenidos en el CAS. Una vez realizado el primer análisis, se invoca el método "Ejecutar" del *AdapterB*. Como se observó anteriormente, este módulo fue invocado con parámetros adicionales. El proceso es llevado a

cabo de la misma manera que en el caso del *AnotadorA*, realizando el procesamiento y escribiendo el resultado en el CAS. Una vez finalizada la secuencia de módulos de análisis se obtiene el conjunto de anotaciones generado a lo largo del procesamiento, y se devuelven los resultados a la aplicación en formato XML.

4.5. Diseño Detallado

Una vez construido el diseño arquitectónico de la herramienta, cada uno de los componentes que la conforman y las tecnologías con las cuales se llevaría a cabo la implementación, se procedió a la materialización de TeXTracT. En primer lugar, en esta sección se detallará cómo se utilizó la biblioteca Jersey para poder exponer la funcionalidad como servicios REST. Luego, se explicará la lógica para listar las técnicas NLP disponibles con reflexión. Por último, se detallará el diseño de los adaptadores de anotadores UIMA utilizados en el proyecto, los cuales permiten encapsular las técnicas de NLP y exponerlas como servicios.

4.5.1. Implementación con Jersey

Con el fin de exponer métodos de Java como servicios Web Jersey provee un conjunto de anotaciones definidas por JAX-RS que posibilitan el acceso mediante el protocolo HTTP. En TeXTracT se utilizaron las anotaciones *@Path*, *@Get*, *@Post* y *@Produce*. Con *@Path* se especifica la ruta relativa de la clase a la que se quiere invocar. Mediante estas anotaciones, se permite asociar una clase Java con un recurso Web dejando así expuestos los diferentes servicios disponibles. Por ejemplo, en la Figura 4.4 se puede observar que se determina que la ruta "<ruta raíz del servicio>/broker" hace referencia a la clase "Broker".

```
@Path("/broker")
public class Broker {
```

Figura 4.4 Declaración de ruta relativa.

Las anotaciones @GET y @POST, especifican el tipo de petición HTTP de un recurso. La anotación @GET fue utilizada para declarar el método *listar* porque no es necesario mandar información (Figura 4.5). Por su parte, la anotación @POST fue usada para invocar los anotadores y su ejecución ya que se debía mandar el texto que se quiere analizar (Figura 4.6).

```
@GET
@Path("/listar")
@Produces(MediaType.TEXT_XML)
public String listarServicios() throws ClassNotFoundException, IOException{
```

Figura 4.5 Declaración GET con Jersey.

Tanto en la Figura 4.5 como en la Figura 4.6, se puede visualizar la utilización de la anotación @PRODUCES. Esta tiene la función de especificar el tipo de medio MIME que se obtiene en la respuesta del servicio. En el caso del método GET, el listado es devuelto en formato XML como se puede leer.

```
@POST
@Path("/exec/{anotador}")
@Produces(MediaType.TEXT_XML)
public String ejecutar(@FormParam("input")String contenido, @PathParam("anotador")String anot)
```

Figura 4.6 Declaración POST con Jersey.

También se utilizaron las anotaciones @FormParam y @PathParam para pasar información de las aplicaciones al *Broker*. La anotación @FormParam se encarga de enlazar un parámetro enviado a través de un formulario con una variable del sistema. La Figura 4.6, muestra como la identificación del elemento del formulario es *input*, mientras que una vez enviado al recurso, el valor del parámetro se asocia a la variable *contenido*. La anotación @PathParam se encarga de enlazar un atributo codificado en la ruta mediante la cual se llama al servicio con una variable. De esta manera, el contenido existente en el lugar *{anotador}* se instancia en la variable de tipo String *anot*. Además, en la imagen se ilustra un ejemplo de este mecanismo. La anotación @PathParam fue muy útil para definir la composición de módulos de NLP, ya que dicha secuencia se encuentra declarada en la URL mediante la que invoca el procesamiento.

Para dar comienzo al procesamiento, la cadena de texto se divide mediante un carácter especial que permite separar el nombre de cada uno de los módulos a ejecutar. El primer componente de esta secuencia, por lo general, es un lector de colecciones, mediante el cual se instancia el lector a utilizar dependiendo del formato de la entrada. Por ejemplo, si se desea procesar un texto plano, la cadena *anot* será la siguiente: *RawTextCollectionReader&AnnotatorA*, quedando especificado que el lector de colecciones utilizados es *RawTextCollectionReader*.

Luego de determinar los módulos de NLP se procede a configurar y ejecutar los mismos. Una vez realizada la separación de los módulos por el simbolo especial se realiza un bucle a través de este listado, obteniendo el nombre del componentes y realizando la instanciación correspondiente. Por ejemplo, si el listado contiene el nombre *InitTokenAnnotator*, se realiza la instanciación del módulo *TokenAnnotator*.

4.5.2. Adaptadores UIMA

La integración de los anotadores de UIMA se llevó a cabo mediante el uso de un componente basado en el patrón de diseño Adapter [Gamma1995]. Principalmente, esta organización permite abtraer los algoritmos NLP unificando la interfaz para que el *Broker* pueda invocar el procesamiento sin necesidad de conocer cada una de las técnicas. Consecuentemente, este diseño permite encapsular cada uno de los algoritmos NLP dentro de una interfaz que adiciona funcionalidad y permite hacer uso de los módulos como aplicaciones independientes. A continuación se puede ver un diagrama conceptual de un adaptador en TeXTracT.

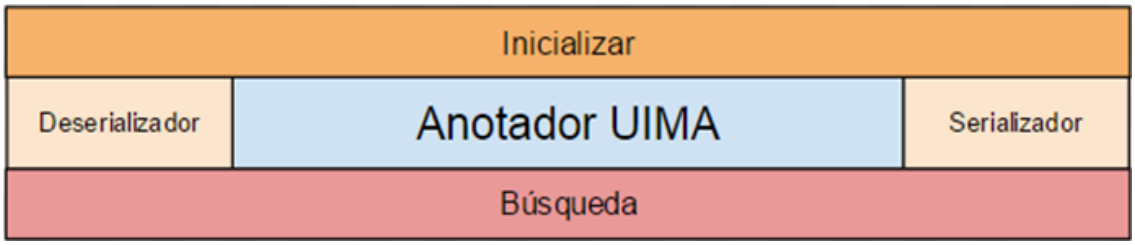


Figura 4.7. Adaptador UIMA

Como se puede ver en la Figura 4.7, el adaptador agrega dos componentes que permiten interpretar la entrada y codificar la salida. El *deserializador* recibe un conjunto de anotaciones como XML junto con el texto de entrada que se desea analizar. Entonces, ese conjunto es convertido a un formato que la técnica de NLP pueda entender. En el caso de los *analysis engines*, se deserializa dicha entrada y se instancia el CAS en memoria. Una vez realizado el procesamiento del texto, el conjunto de anotaciones producidos es codificado por el *serializador*, dando como resultado un CAS con las anotaciones generadas por el anotador encapsulado. En caso de ejecutar un pipeline de anotadores de UIMA, por cuestiones de performance, se tomó la decisión de suprimir la serialización y deserialización intermedia, decodificando la entrada en el primer componente y codificando la salida luego de la ejecución del último anotador.

El adaptador también adiciona la funcionalidad para inicializar una técnica NLP de manera dinámica cuando la invocación de una aplicación lo solicite. El *Broker* identifica la técnica a ejecutar, y crea una instancia de adaptador que contiene dicha técnica. Por ejemplo, se recibe por parámetro *TokenAnnotator* y se crea una instancia de adaptador que contenga la técnica de identificación de *Tokens*. Además, es posible personalizar técnicas de NLP mediante el envió de parámetros en la solicitud de proceso. Estos parámetros son identificados por el *Broker* y pueden hacer referencia a diferentes variables. Por ejemplo, se puede enviar por parámetro el directorio donde se encuentra el modelo al cual responde la técnica NLP. El envío de parámetros permite modificar anotadores existentes sin necesidad de modificar el código y/o recompilar TeXTracT.

Por último, el adaptador permite identificar los anotadores disponibles en TeXTracT. Para realizar esta identificación, el *Broker* utiliza mecanismos de reflexión de Java. La reflexión es una funcionalidad que permite inspeccionar y manipular clases e interfaces en tiempo de ejecución. De esta manera, se buscan las instancias de los adaptadores cargados en la JVM (Java Virtual Machine) permitiendo listar todas las técnicas disponibles. Además, el uso de reflexión permite listar las técnicas de NLP agregadas en ejecución sin necesidad de recompilar TeXTracT.

Capítulo 5 - Caso de Estudio Nº 1

Para poder comprender el desempeño y facilidad de la solución desarrollada en esta tesis, se llevó a cabo un caso de estudio con la implementación de una herramienta nueva. La hipótesis del caso de estudio es que la arquitectura de servicios de NLP permite crear diversos tipos de aplicaciones de manera rápida y sencilla, soportando la evolución de dichos sistemas en el tiempo. El sistema que se utilizó para probar esta hipótesis comprende el desarrollo de un sitio Web para corregir textos escritos en Inglés.
A partir de la división del desarrollo en un conjunto de tareas fijas, se determinó el tiempo implicado para la implementación de un sistema utilizando TeXTracT y uno que realice el procesamiento NLP *ad-hoc*. Durante la implementación de la herramienta, se tomaron métricas respecto al análisis de requerimientos, tiempo de codificación, testing de componentes, entre otros aspectos de interés que determinan el nivel de complejidad del desarrollo de una aplicación NLP.

5.1. TextChecker

En base al procesamiento de texto que permite hacer TeXTracT, se desarrolló una herramienta Web denominada *TextChecker* que permite obtener correcciones y/o sugerencias gramaticales a partir del análisis de texto. Algunas de las funcionalidades provistas por la herramienta son:

- **Carga de texto**: Permite cargar el texto que se desea analizar a través de la interfaz Web.
- **Información de errores**: Lista el conjunto de los posibles problemas gramaticales que la herramienta puede detectar en el texto. Cada uno de los errores contiene una breve descripción del mismo.
- **Análisis del texto**: Permite analizar el texto ingresado mediante la herramienta TeXTracT. *TextChecker* hace uso de los módulos de NLP precargados y módulos de detección de problemas gramaticales específicamente desarrollados para tal fin.
- **Resultado del análisis**: Permite visualizar los errores detectados por TeXTracT de manera práctica, resaltando el contenido del texto con una leyenda que indica los problemas gramaticales y su tipo.

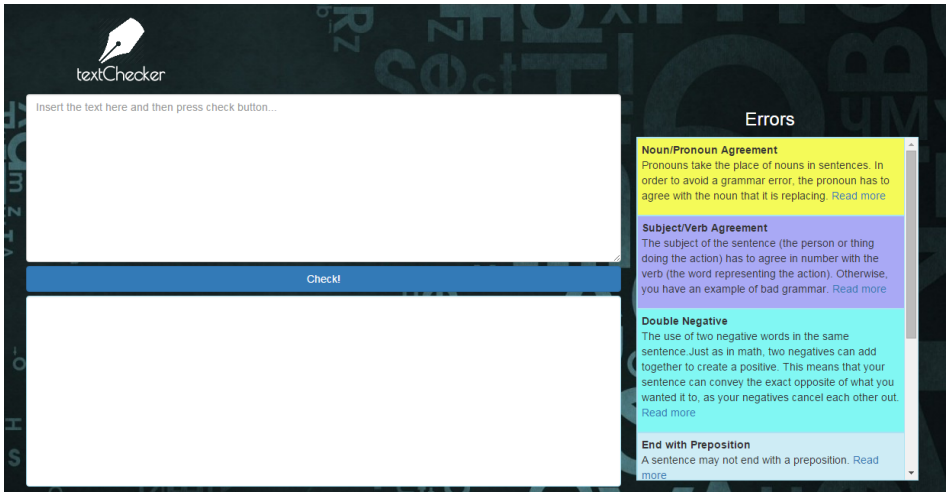


Figura 5.1. TextChecker.

En la Figura 5.1, se puede observar una captura de pantalla de la herramienta. En la parte superior izquierda, se encuentra un campo de texto en el cual se debe ingresar el documento que se desea corregir. En la derecha, la página contiene una lista de los tipos de error que TextChecker puede identificar con una descripción breve que explica de qué se trata. Cada uno de los errores que aparecen en la lista contiene un enlace que permite ampliar la información del error que se describe. En la Figura 5.2, se aprecia cómo TextChecker presenta los resultados de una corrección (parte inferior izquierda).

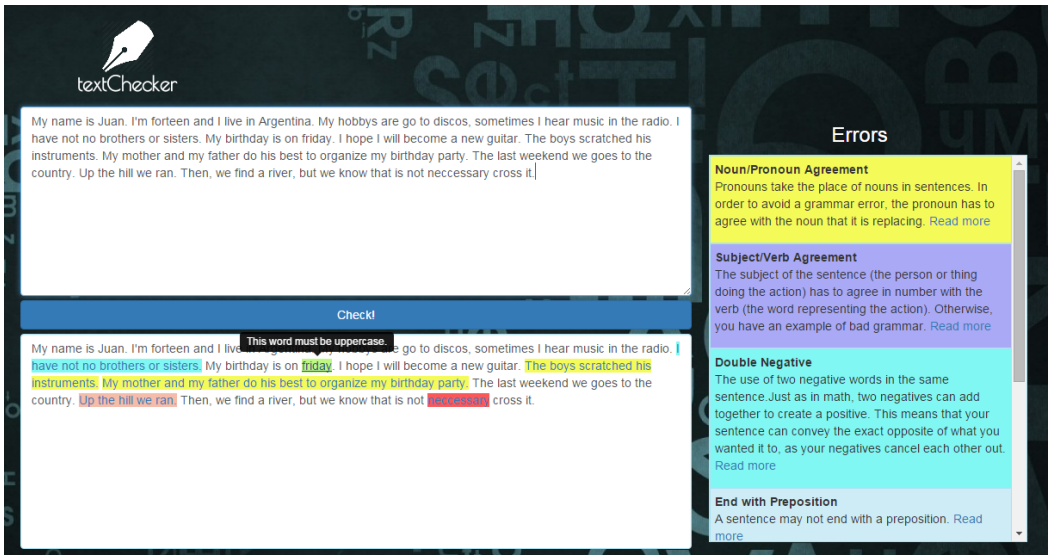


Figura 5.2. Corrección en TextCheker.

Por ejemplo, en el texto de la oración "*I have not no brothers or sisters*" se encuentra resaltado por contener dos palabras con forma de negación También se resalta la palabra "*friday*" por estar en minúsculas, ya que los días de la semana deben comenzar con mayúsculas.

5.2. Diseño

Para desarrollar el corrector TextCheck, se planteó utilizar una arquitectura cliente-servidor en la cual desde una página Web se invocan los servicios Web de NLP que provee TeXTracT para procesar el texto a corregir. Una vez obtenidos los resultados de análisis, el cliente Web separa los datos de interés de las anotaciones NLP relevantes, y resalta las correcciones correspondientes. La Figura 5.3 ilustra la arquitectura del corrector en un diagrama de componentes.

La integración de los servicios NLP necesarios según los requisitos de TextChecker fue directa. En primer lugar, se listan los módulos de NLP disponibles en la herramienta para identificar aquellos de utilidad para corregir el texto.

Debido a que TeXTracT no contaba con módulos para identificar patrones en el texto, se decidió integrar identificadores de patrones dentro de textos con el lenguaje RUTA. Esta tarea se realizó de manera simple debido a la flexibilidad de TeXTract. Desde el cliente Web, los módulos *Sentence, Token, Chunk, POSTagger, CoNLLDependency* y los patrones de identificación son invocados en un mismo llamado indicando la secuencia para obtener como resultado los errores gramaticales.

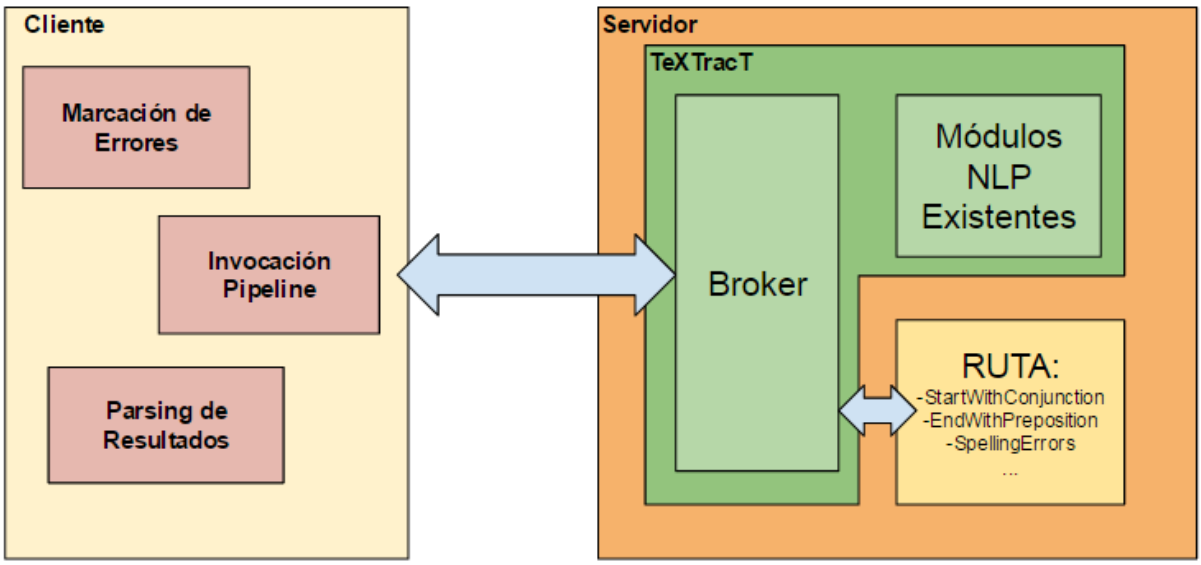


Figura 5.3. Arquitectura de TextChecker

5.2.1. Servidor

En el lado del servidor, inicialmente se realiza el procesamiento de lenguaje natural necesario para desglosar un texto en Inglés y facilitar la búsqueda de los fragmentos de texto con errores. Al analizar las distintas reglas de gramática y ortografía del lenguaje, se obtuvo un conjunto de módulos de análisis

necesarios para implementar los patrones identificadores de errores. Luego de explorar potenciales alternativas para implementar las reglas ortográficas y gramaticales de manera sencilla, se decidió utilizar un lenguaje basado en reglas denominado RUTA (Rule-based Text Annotation), que permite la declaración de patrones de anotaciones UIMA, y a partir de un conjunto de condiciones, generar nuevas anotaciones en el texto. RUTA trae consigo diversas ventajas, como por ejemplo, la simplificación del diseño de los módulos ya que está basado en la misma la tecnología UIMA. Otra ventaja es la simplicidad a la hora de acoplar estas reglas RUTA a la secuencia de análisis reutilizando los módulos preexistentes.
Una vez implementado este módulos, se definió que el acceso al servidor se realiza mediante un llamado a una URL determinando cuales son los componentes a utilizar. De esta manera, una vez realizado el proceso por TeXTracT, se obtienen los resultados solicitados por TextChecker.

5.2.2. Cliente

Dado que el análisis del texto para buscar errores gramaticales u ortográficos se realiza en su totalidad en el servidor, las responsabilidades del cliente se limitan al envío y recepción de información, como así también de la presentación de los resultados y la interacción con el usuario. El sitio Web consta de un formulario que permite ingresar el texto que se desea analizar. Este contenido textual es enviado al servidor mediante un método POST de HTTP. La página se comunica con el servidor mediante llamados asincrónicos utilizando tecnología AJAX. Luego de haber procesado el texto, el servidor producirá como respuesta un XML con el contenido del texto procesado y las anotaciones correspondiente a las correcciones gramaticales.
A partir de los resultados, el cliente tendrá la responsabilidad de descomponer el XML para poder identificar las anotaciones de los errores en el texto detectadas por los módulos de análisis. Cada una de las anotaciones presenta un identificador y propiedades particulares del error gramatical encontrado en el texto. Por ejemplo, una anotación *EndWithPreposition* indica que la oración está potencialmente mal escrita, ya que la última palabra de la misma se corresponde con una preposición, siendo esta una mala práctica que no debe llevarse a cabo en textos formales. Cada anotación contiene los caracteres de principio y el final de la anotación para identificar la oración afectada. Una de las responsabilidades del cliente Web será utilizar dichos los límites para poder ubicar el fragmento de texto al cual hace referencia la anotación y marcarlo con el tipo de error apropiado. Una vez identificados cada uno de los errores, la aplicación marca visualmente las correcciones sugeridas por la herramienta. A continuación, se detallarán las reglas implementadas haciendo foco en cada componente del lenguaje RUTA.

5.2.3. Apache UIMA Ruta

RUTA es un lenguaje basado en reglas diseñado para el desarrollo ágil de aplicaciones de procesamiento de textos con UIMA [Kluegl2015, Toepfer2014]. Mediante esta tecnología se pueden definir patrones de anotaciones de UIMA de manera intuitiva y flexible. RUTA cuenta con varios plugins para Eclipse construidos para facilitar la escritura de reglas incluyendo editores con correctores sintácticos del lenguaje y mecanismos para evaluar las reglas escritas.

Las reglas codifican un patrón de anotaciones con condiciones adicionales. Si este patrón se corresponde con el texto, una acción determinada se ejecutará sobre la/s anotación/es correspondiente/s. Una regla particular está compuesta de cuatro partes: la condición a verificar, un cuantificador opcional, una lista de condiciones y una lista de acciones. Una condición se refiere a una anotación individual la cual se pretende encontrar en el texto. El cuantificador especifica la frecuencia con la que dicha anotación debe ocurrir para considerarla una coincidencia. La lista de condiciones consta también de un conjunto adicional de restricciones determinada por el primer componente de la regla que el fragmento de texto o una anotación debe cumplir. La lista de acciones describe las consecuencias de la regla, las cuales pueden referirse a la creación de nuevas anotaciones o la modificación de anotaciones existentes.

Para ejemplificar el funcionamiento de una regla de RUTA, se explicará el desarrollo de una regla para identificar pronombres masculinos, destacando las diferentes partes de la misma. En primer lugar, para declarar un componente del lenguaje, se utiliza la palabra *DECLARE* seguida del nombre del nuevo tipo. De esta manera, se pueden crear las anotaciones necesarias para indicar la existencia de un pronombre masculino. Por ejemplo, para declarar una nueva anotación denominada “SingularPronoun” se utiliza el siguiente código:

```
DECLARE MalePronoun;
```

Una vez definidos los tipos de anotación que se van a utilizar (sean importados o definidos con la cláusula DECLARE), se puede comenzar a codificar las reglas de búsqueda. Cada regla consta de una anotación sobre la cual se verifica la lista de condiciones. En siguiente ejemplo se puede ver que la anotación denominada W, la cual representa cualquier palabra de un documento.

```
W{REGEXP("he") -> MARK(MalePronoun)}
```

En el ejemplo anterior, se encuentra entre llaves la lista de condiciones y la lista de acciones. La lista de condiciones para este caso particular es REGEXP, una función de expresión regular que verifica si los caracteres dentro de la anotación W se corresponden con la cadena “he”. Después de las condiciones, se encuentra la lista de acciones correspondiente. En este ejemplo particular, es una acción simple que indica que se debe crear una nueva anotación *MalePronoun* que marque la palabra W.

RUTA permite describir las reglas utilizando diferentes estructuras, según las necesidades las aplicaciones. Por ejemplo, la misma regla explicada anteriormente podría ser descripta de la siguiente manera:

```
W{ -> MARK(MalePronoun)}<- {REGEXP("he")}
```

Como se puede ver, dentro de las llaves que se encuentran al lado de la anotación a analizar, solamente se encuentra la lista de acciones. A diferencia del estilo de codificación anterior, donde la condición precede a la lista de acciones, en esta regla se puede ver como la lista de condiciones pasa a estar del lado derecho de la regla.

5.3. Desarrollo

En esta sección, se detalla la implementación del corrector de texto. En primer lugar, se mostrarán las reglas gramaticales seleccionadas y el desarrollo de las mismas con el lenguaje UIMA RUTA. Luego, se describirán los pasos llevados a cabo para la integración de dichos módulos, evaluando la dificultad que esta tarea implica para el desarrollo de una aplicación nueva. Por último, se ejemplifica la aplicación de corrección de texto sobre un documento que presenta errores para visualizar el funcionamiento de la herramienta.

5.3.1. Procesamiento de Reglas de Anotaciones en UIMA

Para el análisis del texto en Inglés, se investigaron un grupo significativo de problemas ortográficos y gramaticales recurrentes para que la aplicación desarrollada pueda detectar diferentes tipos de errores en los documentos. Teniendo en cuenta que el lenguaje a utilizar para describir las reglas de RUTA utiliza scripts de patrones de anotaciones, se exploraron diferentes tipos de reglas gramaticales y se estudió su adaptación al lenguaje específico de anotaciones. Cada una de estas reglas se integraron a la secuencia de procesamiento NLP para poder utilizar los resultados parciales de los módulos de NLP, y partir de los mismos generar las anotaciones que hacen referencia a los errores gramaticales que posee el texto de entrada.

5.3.2. Integración de RUTA en el Pipeline de NLP

En esta sección se explican las modificaciones realizadas en la infraestructura de TeXTracT para integrar RUTA. En primer lugar, se implementó una clase genérica llamada *RutaAnnotator*, que permite instanciar cada uno de los componentes de corrección de texto RUTA. Esta clase permite manipular los módulos de

corrección de texto de la misma manera que a los módulos de NLP. De esta manera, se podrá insertar cada uno de los módulos RUTA en cualquier secuencia de proceso.

```
@Override
public void process(JCas jCas) throws AnalysisEngineProcessException {
    InputStream in = null;
    in = this.getClass().getClassLoader().getResourceAsStream(script);
    String result = getStringFromInputStream(in);
    try {
        Map<String, Object> parameters = new HashMap<String, Object>();
        parameters.put(RutaEngine.PARAM_DEBUG, debug);
        Ruta.apply(jCas.getCas(),result,parameters);
```

Como cada error gramatical va a tener un script de RUTA asociado se implementó un método llamado "*getStringFromInputStream*" que permite leer dichos archivos del servidor. Una vez obtenido el contenido, la biblioteca de RUTA permite ejecutar el script sobre las anotaciones que se pasan por parámetro. Como resultado, los scripts crearán anotaciones para los errores detectados.

Con el fin de identificar cada uno de los módulos de corrección de texto dentro de TeXTracT, se planteó que cada uno haga uso de una clase *InitAnnotator* de la misma manera que lo hace cada módulo de NLP. Esta clase, a través de una *Factory*, permite generar un analizador a partir de un descriptor del componente que se desea crear. Para el caso específico de una regla RUTA, se debe indicar la ruta en la cual se encuentra el archivo con el código correspondiente.

A continuación se muestra el fragmento de código que declara un anotador en base a la regla *StartWithConjunction*. La clase lleva el nombre "*InitStartWithConjunctionRutaAnnotator*" siguiendo la convención que permite la identificación del nombre de los componentes disponibles dinámicamente. Mediante la reflexión utilizada en TeXTracT, se podrá identificar el módulo como "*StartWithConjunctionRutaAnnotator*", y así poder hacer uso de este a través de solicitudes HTTP.

```
21 public class InitStartWithConjunctionRutaAnnotator extends InitAnnotator {
22
23     @Override
24     public AnalysisEngine getEngine(TypeSystemDescription typeSystemDescription, TypePriorities typePriorities)
25         AnalysisEngineDescription aeDescription =
26                 AnalysisEngineFactory.createEngineDescription(RutaAnnotator.class,
27                         typeSystemDescription, typePriorities, RutaAnnotator.RUTA_SCRIPT,
28                             getModelsPath()+ "$RUTA_DIR$/StartWithConjunction.ruta", RutaAnnotator.RUTA_DEBUG, false);
29         AnalysisEngine analysisEngine = UIMAFramework.produceAnalysisEngine(aeDescription);
30         return analysisEngine;
31     }
32
33 }
```

Una vez que se genera el anotador, el componente *Broker* podrá hacer uso del mismo mediante la invocación del método *process*. El método *process* realiza el análisis sobre el *JCas* pasado por parámetro, en base al script de RUTA especificado en el método *getEngine* de *InitAnnotator*. Como resultado de esta operación se obtiene un nuevo conjunto de anotaciones dentro del CAS que indican los errores correspondientes a la regla *StartWithConjunction*.

5.3.3. Exploración de Reglas Ortográficas y Gramaticales

Dentro del conjunto de reglas gramaticales exploradas, se tuvo en cuenta la probabilidad de ocurrencia de cada uno de los errores tratando de buscar aquellos errores más comunes y cotidianos en el texto, como así también considerando aquellos que puedan ser revelados mediante patrones de anotaciones. A continuación, se dará una breve explicación de cada una de las normas gramaticales seleccionadas.

Finalizar una oración con una preposición (EndWithPreposition)[8]: Aunque en conversaciones y documentos informales es aceptable terminar una oración con una preposición, esta afirmación no es válida en la escritura formal. Por ejemplo, la oración *"This is a pleasant city to live in."* es válida en un contexto informal. Sin embargo, si se quiere comunicar la misma idea en un contexto formal, una forma más correcta de escribir la oración sería: "*This is a pleasant city in which to live.*".
Gramaticalmente, las oraciones que terminan con una preposición están sujetas a errores de comprensión por parte de los lectores, los cuales pueden perder la referencia al sujeto. Por ende, las oraciones finalizadas con preposiciones frecuentemente derivan en malas interpretaciones cuando se trata comunicar ideas en ambientes técnicos, de difusión, o incluso hay varios actores en una misma oración.

Relación entre sustantivo y pronombre dentro de una oración (NounPronounAgreement)[9]: el propósito de un pronombre es reemplazar o referenciar a un sustantivo nombrado anteriormente en una oración. Al igual que los sujetos y los verbos, los sustantivos y los pronombres deben coincidir en número dentro de una misma oración. En la siguiente oración se puede observar un posible uso de los pronombres.

When an **individual** is sick, the **individual** should call a doctor.
When an **individual** is sick, **he** or **she** should call a doctor.

Como se puede observar, en la primer oración se hace referencia al sujeto "*individual*" en las dos partes, mientras que en la segunda se utilizan los pronombres "*he*" y "*she*" para reemplazar al sujeto. En este caso, la correspondencia entre sujeto y pronombre es correcta, ya que "*individual*" es un sujeto singular y los pronombres "*he*" y "*she*" son ambos singulares. A continuación se puede ver una oración que no cumple con la regla.

When **individuals** are sick, **he** or **she** should call a doctor.

[8] http://www.grammar-quizzes.com/preps-placement.html

[9] http://academicguides.waldenu.edu/writingcenter/grammar/nounpronounagreement

En este caso, se aprecia que "*individuals*" referencia a múltiples personas y los pronombres son, al igual que en la oración anterior, singulares. En este caso, a menos que la oración esté dentro de un contexto que aclare que "*he*" o "*she*" hacen referencia a otra persona, estamos ante la presencia de un error gramatical.

Relación entre sujeto y verbo dentro de una oración (SubjectVerbAgreement)[10]: existe una regla básica del Inglés que indica que los sujetos singulares deben corresponderse con verbos singulares y de la misma manera, para sujetos plurales corresponden verbos plurales. Esta regla, al igual que la de los pronombres, relaciona el sujeto de una oración con el verbo, verificando que los dos concuerdan en número. Por ejemplo, considere las siguientes oraciones.

My brother is a nutritionist. **My sisters are** mathematicians.

Se puede ver que "*My brother*" es un sujeto singular y concuerda con el verbo "*is*", que es la conjugación singular del verbo "*to be*". En la segunda oración, "*My sisters*" es un sujeto plural que concuerda con el verbo "*are*" que es la conjugación plural del verbo "*to be*". En cambio, si se consideran las siguientes oraciones dicha correspondencia no se cumple.

My brother ~~are~~ a nutritionist. **My sisters ~~is~~** mathematicians.

En estas oraciones, se puede observar que las conjugaciones del verbo "*to be*" no concuerdan. En la primer oración, el sujeto singular "*My brother*" está relacionado con "*are*", siendo este un verbo plural. En la segunda oración, el sujeto plural "*My sisters*" está relacionado con el verbo singular "*is*".

Empezar una oración con una conjunción (StartWithConjunction)[11]: por definición, una conjunción es una palabra que sirve para unir oraciones o partes de una oración. En muchas ocasiones, una oración puede comenzar con una conjunción y transmitir una idea correctamente. Desafortunadamente, un problema común que suele presentarse en los documentos es que erróneamente una oración comience con una conjunción sin unir dos o más ideas de oraciones adyacentes. A continuación se puede observar dos oraciones unidas con una conjunción.

He did not come to work. **Because** he was ill.

[10] http://www.grammarbook.com/grammar/subjectVerbAgree.asp

[11] http://www.englishgrammar.org/common-mistakes-conjunctions-3/

En este caso, la oración que comienza con una conjunción es una subordinada y esta carece de sentido por sí misma de forma aislada. La manera correcta de escribir esta oración sería:

He did not come to work **because** he was ill.

En esta oración, la conjunción es "*because*" qué significa "porque" e indica el motivo por el cual el sujeto no fue a trabajar "*He did not come to work*" ¿por qué? "*because he was ill"*. Esta regla ortográfica no tiene una eficacia del 100%, ya que es necesario comprender el contexto de la oración para determinar su correctitud. Sin embargo, la detección de dichas malas prácticas al comenzar oraciones permite advertir a los escritores de los posibles errores acarreados por el uso de conjunciones como primer palabra dentro de una oración. Para saber si estamos cometiendo un error de este tipo, lo ideal es establecer si las oraciones tienen sentido por sí solas cuando se les remueve su contexto.

Dobles negativos (DoubleNegative)[12]: este error se refiere a aquellas oraciones en las que se utilizan dos formas de negación en la misma oración. La doble negación no intensifica la negación de la oración, sino que por el contrario, hace positivo el significado de la oración y la idea que se pretendía transmitir. Por ejemplo:

I **don't not** love you. We **ain't** got **no** story.

En la primer oración, se puede ver como la palabra "*don't*" qué es la abreviación de "*do not*" y "*not*" se anulan entre sí, cambiando el significado de la idea que se quiere comunicar. De la misma manera, las negaciones "*ain't*" y "*no*" de la segunda oración se anulan entre si, causando el mismo efecto.

I **don't** **not** love you. We **ain't** got **no** story.

Errores de escritura (SpellingMistakes): además del conjunto de reglas encontrado a partir del estudio de errores comunes en el idioma Inglés, se encuentra un listado de palabras que con frecuencia son escritas incorrectamente. Por esta razón, se compiló un listado exhaustivo de equivocaciones comunes al escribir palabras a partir de varios documentos en Inglés para facilitar su búsqueda mediante el lenguaje de anotaciones.

Palabras en mayúsculas (UpperCaseWords): en el idioma Inglés, existe un conjunto de palabras que siempre deben ser escritas con la letra inicial en mayúscula. Entre estas palabras, se pueden mencionar

[12] http://grammar.yourdictionary.com/grammar-rules-and-tips/double-negative-trouble.html

los días de la semana (como puede ser "*Monday*" (lunes)), los meses del año (por ejemplo "*January*"(enero)) y fechas especiales del año. Para encontrar este tipo de entidades en el texto, también existen listados exhaustivos que indican la forma singular de escritura que tiene cada palabra.

5.3.4. Codificación de Reglas en RUTA

Teniendo en cuenta los mecanismos de búsqueda provistos por el lenguaje de anotaciones RUTA, en esta parte del Capítulo se describirán con mayor detalle la definición de los scripts para detectar problemas gramaticales y ortográficos en módulos que utilizan reglas basadas en NLP para efectuar su detección automática. Con este propósito, se ilustran cada una de las reglas gramaticales junto con una explicación del funcionamiento de los patrones de anotaciones para cada caso particular. Para crear las reglas gramaticales descritas en la sección anterior se utilizaron los módulos de NLP *Sentence, Token, POSTagger, Lemmatizer, Chunker, CoNLLDependency* y *CoNLLSRL.*

Módulo EndWithPreposition: este módulo corresponde a la regla gramatical que indica los problemas existentes al terminar una oración con una preposición. El script de RUTA para detectar este tipo de problemas es el siguiente:

```
5  DECLARE EndPreposition(STRING desc);
6
7  Sentence{-> CREATE(EndPreposition, "desc" = "A sentence cannot end with a preposition.")}<-{
8      (edu.isistan.uima.unified.typesystems.nlp.Token.pos == "IN")+
9          Token.pos == ".";
10 };
```

Este primer fragmento de código muestra en primer lugar la declaración del tipo de error "*EndWithPreposition*". En todas los módulos desarrollados la primer línea se corresponde a la declaración de una anotación que corresponde al error que hace referencia. Cada una de estas anotaciones tiene un atributo "*desc*" que almacenará una frase descriptiva del error encontrado.

En la segunda línea se indica mediante la anotación "*Sentence*" que para cada una de las oraciones del texto evalúe la posibilidad de crear una anotación del tipo "*EndPreposition*" con su respectiva descripción, siempre y cuando se cumpla una condición dicha acción se ejecuta con el método "*CREATE*" y agregando como parámetro un texto que explica que una oración no debería terminar con una preposición.

A continuación de la lista de acciones, se encuentra la condición que debe cumplir la oración. En este caso particular, se verifica que la oración contenga una anotación "*Token*" cuyo atributo "*pos*" sea "*IN*"(es decir, una preposición). El operador "+" indica la concatenación de anotaciones en el lenguaje RUTA. Particularmente, se pretende encontrar aquellas oraciones que su última palabra es una

preposición, mirando si el token que le sigue a una oración es un punto que denota su terminación. En caso de cumplirse la condición, se generaría una anotación como se muestra a continuación.

```
<EndWithPreposition begin="64" end="72" desc="Sentence end with a preposition."/>
```

Para mayor información acerca de las implementación de las reglas ortográficas en RUTA, el lector puede consultar el Apéndice A.

5.3.5. Interfaz de Usuario

La arquitectura cliente-servidor de TextChecker simplifica el desarrollo de la UI debido a que el procesamiento y análisis del texto no se realiza en el cliente. Esto significa que las responsabilidades del cliente Web son obtener el texto ingresado por el usuario, armar una secuencia de proceso específica para la corrección de dicho texto, realizar la llamada al servidor y visualizar los resultados obtenidos.
La construcción de la secuencia de proceso NLP se define de manera estática dentro de TextChecker, ya que los módulos utilizados son siempre los mismos. En la implementación, esto implica que la URL que realiza la solicitud al servidor está predefinida para ejecutar los módulos de *RawText, Sentence,*etc (ver ejemplo a continuación).

```
$URL_BASE/RawTextCollectionReader&InitOpenNlpSentenceAnnotator&InitOpenNlpTokenAnnotator&
        InitOpenNlpPosAnnotator&InitMatetoolsLemmaAnnotator&InitOpenNlpChunkAnnotator&
        InitMatetoolsCoNLLDependencyAnnotator&InitCoNLLSRLAnnotator&
        InitUppercaseRutaAnnotator&InitSVARutaAnnotator&InitStartWithRutaAnnotator&
        InitSpellingMistakesRutaAnnotator&InitDNegativeRutaAnnotator&
        InitEWPrepositionsRutaAnnotator& InitNounRutaAnnotator&InitCleanerRutaAnnotator
```

La invocación con la URL es enviada a través de un llamado AJAX junto con el texto ingresado por el usuario. Esta tecnología permite realizar una solicitud HTTP asincrónicamente sin necesidad de recargar el contenido la interfaz de usuario. Esto da como resultado una ejecución más fluida de la herramienta ya que se realiza la petición al servidor y se obtiene la respuesta en segundo plano, modificando los datos obtenidos de manera dinámica y rápida con Javascript. A continuación, se puede ver el código referido a este llamado asincrónico:

```
function procesarTexto(){
  var urlToProcess = url;
  var form = $("form").serialize();
  $.ajax({
    async:true,
    cache:false,
    type: 'POST',
```

```
            url: urlToProcess,
            data: form,
            success:  function(respuesta){
                pintarTexto(respuesta);
            },
            error:function(objXMLHttpRequest){}
        });
    }
```

En este código se obtiene la URL dentro de la variable *urlToProcess* y el texto ingresado por el usuario en un formulario *form*. Con estos datos se realiza la llamada asincrónica indicando que el método HTTP solicitado es POST, ya que el texto es contenido por un formulario en la interfaz de usuario. Como el llamado es asincrónico, se declara una función a ejecutar cuando el servidor envíe la respuesta a la solicitud enviada, en este caso, se ejecuta la función *pintarTexto*.

La función *pintarTexto* recibe como parámetro el resultado del procesamiento realizado por TeXTracT con cada una de las anotaciones referidas a las correcciones gramaticales indicadas en un principio. El método es el siguiente:

```
function pintarTexto(respuesta){
  var contenido = textoDeEntrada;
  for (var i = xml.length - 1; i >= 0; i--) {
    var inicio =  xml[i].getAttribute('begin');
    var fin = xml[i].getAttribute('end');
    var error = xml[i].getAttribute('error');
    agregarEtiqueta(inicio, fin, error,contenido);
  };
  return contenido;
}
```

Este método obtiene el texto ingresado por el usuario en la interfaz del corrector para poder etiquetarlo. Luego, itera sobre el contenido XML de la respuesta emitida por el servidor, y para cada una las anotaciones generadas se obtiene el carácter inicial, el carácter final y el tipo de error al que se refiere. Posteriormente, se agrega sobre el texto de salida una etiqueta en el fragmento de texto con el error correspondiente, permitiendo así visualizar el error.

5.4. Experiencias del desarrollo

De forma tal de entender las bondades de utilizar nuestra solución de servicios para desarrollar aplicaciones NLP, se llevó a cabo una comparación entre el desarrollo hipotético de una aplicación ad-hoc y una aplicación que aprovecha la infraestructura de NLP de TeXTracT. La hipótesis es que la herramienta propuesta puede mejorar significativamente ciertas tareas del desarrollo de las aplicaciones, reduciendo los tiempos de desarrollo y aliviando el esfuerzo de los programadores. Para determinar la validez de esta hipótesis, primero se planificaron las tareas necesarias para el desarrollo de la aplicación

de forma ad-hoc y se estimaron los tiempos de codificación y el esfuerzo particular de cada tarea. Luego, se recabaron métricas concretas de tiempo y esfuerzo durante la implementación real de la aplicación usando TeXTracT, con el objetivo de poder cuantificar las diferencias con la planificación. La estimación de la duración de cada tarea fue realizada a partir del conocimiento propio en las tecnologías necesarias para llevar a cabo la implementación de la herramienta con el procesamiento de texto ad-hoc. La estimación es un buen parámetro de comparación ya que indica la mejora en tiempo de desarrollo para este caso de estudio en el que se propone la realización de una nueva herramienta.

Desde una perspectiva cualitativa, la implementación de la herramienta utilizando TeXTracT fue sencilla. Muchos de los componentes existentes en la infraestructura fueron reutilizados y combinados, con el fin de obtener la información necesaria para identificar los errores gramaticales utilizando las reglas de RUTA. Respecto a la implementación de componentes nuevos, se aprovechó la arquitectura flexible y configurable con la cual se diseñó TeXTracT para poder incorporar RUTA como un nuevo anotador. Para lograr esto se utilizo un adaptador, que mediante la instanciación con un script RUTA particular, generó cada uno de los módulos de reglas gramaticales. De esta manera, la herramienta de corrección permite la combinación de estos nuevos módulos en el pipeline de procesamiento agregando las anotaciones correspondientes al resultado.

Con respecto a la codificación de reglas ortográficas, RUTA simplificó el agregado de los nuevos componentes de reglas gramaticales ya que está basado en UIMA al igual que TeXTracT. Con este lenguaje se redujo el trabajo necesario a la hora de agregar el módulo de RUTA a un pipeline de procesamiento de texto.

La materialización de la aplicación fue bastante rápida, esencialmente porque con algunos conocimientos básicos de HTML y JavaScript fue posible invocar el pipeline de NLP e interpretar los resultados para su presentación al usuario. La implementación de la interfaz de usuario con estas tecnologías dió como resultado una aplicación ágil que utiliza mínimos requisitos de procesamiento del lado cliente.

A partir de este caso de estudio es interesante evaluar no solo la simplicidad del desarrollo de una aplicación que haga uso de los servicios Web, sino también con el fin de comprobar la dificultad para agregar módulos de análisis a la herramienta de procesamiento de texto.

Tarea	Descripción	Ad-hoc	TeXTract	Etapa
T1	Analizar el dominio de la aplicación de corrección de textos	4 días	4 días	Análisis de Requerimientos
T2	Relevamiento bibliográfico acerca de de reglas gramaticales en inglés	4 días	4 días	
T3	Relevamiento bibliográfico de técnicas de NLP	4 días	4 días	

T4	Investigación de mecanismos para identificar patrones en el texto	8 días	8 días	
T5	Definición de una infraestructura para procesar texto en lenguaje natural	4 días	1 día	Diseño
T6	Selección y adaptación de algoritmos concretos de NLP	4 días	1 día	
T7	Integración entre los diferentes módulos de NLP	4 días	1 día	
T8	Selección e integración de tecnología para buscar problemas gramaticales	4 días	4 días	
T9	Comunicación entre aplicación cliente y backend	4 días	2 días	
T10	Desarrollo del procesamiento NLP e infraestructura de proceso ad-hoc	8 días	1 día	Implementación
T11	Desarrollo reglas gramaticales	8 días	6 días	
T12	Desarrollo de interfaz de usuario	4 días	4 días	
T13	Implementación de invocación remota para el análisis del texto	6 días	2 días	
T14	Desplegar módulos de reglas gramaticales	4 días	2 días	Despliegue
T15	Desplegar aplicación en el servidor	2 días	2 días	
T16	Pruebas sobre módulos de reglas gramaticales en pipeline de proceso	4 días	2 días	Testing
T17	Pruebas sobre aplicación cliente	2 días	2 días	

Tabla 5.1. Tareas para el desarrollo de la herramienta.

En la Tabla 5.1, se puede ver el detalle de cada una de las tareas realizadas durante el desarrollo de TextChecker. En el gráfico de barras (Figura 5.4), permiten visualizar la ganancia (aproximada) obtenida a la hora de desarrollar una aplicación con procesamiento NLP ad-hoc contra la utilización de TeXTracT. A partir de las tareas que implica el desarrollo de TextChecker, en la Tabla 5.1, se puede ver a qué etapa de desarrollo corresponde cada tarea junto al tiempo en días que lleva realizarla.

En la Figura 5.4 se puede observar que en la etapa de requerimientos no se obtuvieron mejoras, ya que la identificación de problemas y las necesidades que tiene el desarrollo del corrector son independientes de

la forma de implementación seleccionada. En cambio, en la etapa de diseño, se comenzaron a obtener mejoras en el desarrollo utilizando TeXTracT. La definición de una infraestructura para procesar texto se realizó de manera más rápida ya que TeXTracT prescribe una organización y arquitectura fija y lo único que se debe hacer es aprender cómo opera. Mientras que el desarrollo ad-hoc implica un esfuerzo mayor debido a que es una decisión relevante sobre la cual se basará la herramienta. Lo mismo ocurre con la selección e integración de los algoritmos NLP, ya que TeXTracT contiene un conjunto de módulos que responden a diferentes algoritmos y técnicas de procesamiento estándar, y sólo se requiere explorar cuales son útiles y cuáles se deben agregar/modificar. En cambio, en un desarrollo ad-hoc se deben explorar cada uno de los módulos de análisis existentes, investigar el funcionamiento de cada uno, y adaptarlos a la aplicación, implicando un esfuerzo mayor.

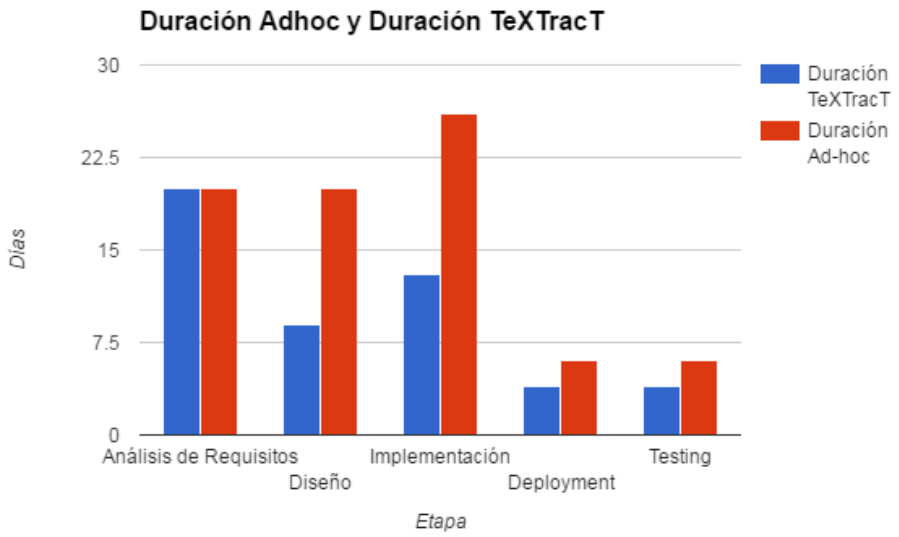


Figura 5.4 Gráfico por etapas de desarrollo.

La comunicación entre la aplicación y el servidor es otra decisión de diseño que se simplifica con TeXTracT ya que se realiza a través de una interfaz Web. En esta etapa se identificó la mayor ganancia en tiempo ya que la utilización de la herramienta Web tiene definidas muchas cuestiones de diseño a las cuales se debe adaptar el corrector. La variedad de módulos de análisis y la flexibilidad del sistema permiten componer un pipeline de proceso que cumpla los requisitos del sistema. Por el contrario, a la hora de realizar un desarrollo ad-hoc, se debe tener en cuenta las interfaces de comunicación de entrada y salida con la que trabaja cada uno de los módulos tanto para poder combinarlos entre sí como para obtener el resultado solicitado. Esta tarea puede ser compleja si se tiene en cuenta que cada uno de los módulos son desarrollados sin tener en cuenta una notación estándar.

En la etapa de implementación se obtuvo una ganancia mayor que en la etapa de diseño. El tiempo de desarrollo de las reglas gramaticales y la interfaz de usuario no difieren demasiado entre el desarrollo ad-hoc y la utilización de TeXTracT, ya que son tareas independientes del procesamiento de texto. Sin

embargo, se obtuvo una ganancia importante a la hora de desarrollar la infraestructura y los módulos NLP, ya que TeXTracT permite la utilización de los componentes de análisis sin necesidad de crear una estructura propia. También, se obtiene ganancia durante la comunicación remota para el análisis de texto. Cuando se utiliza TeXTracT la comunicación se realiza mediante la interfaz que define la herramienta, mientras que en la implementación a medida se deben definir cada una de las políticas de comunicación.

Durante el despliegue del sistema, la ventaja que tiene TeXTracT por sobre el desarrollo a medida es que su diseño flexible permite agregar los módulos de reglas gramaticales de manera simple y sin necesidad de grandes esfuerzos en tiempo de programación. El desarrollo *ad-hoc* implica el despliegue tanto de los módulos de NLP como el conjunto de módulos de reglas gramaticales, implicando un esfuerzo mayor. A la hora de desplegar el frontend en el servidor no se observaron mejoras considerables, ya que su implementación no depende del procesamiento del texto realizado por TeXTracT.

En la etapa de pruebas, se obtuvieron mejoras cuando se realizan las pruebas para identificar los errores gramaticales, ya que la utilización de TeXTracT garantiza el correcto funcionamiento de las técnicas de NLP y la comunicación entre las mismas. Por lo tanto, solo es necesario comprobar el funcionamiento correcto de las reglas gramaticales agregadas. Mientras que en el caso de las pruebas sobre la aplicación cliente no se obtuvieron ventajas. En cambio, cuando se realiza el desarrollo *ad-hoc* se deben probar el funcionamiento de cada uno de los módulos de NLP individuales, cada uno de los módulos de reglas gramaticales y la comunicación entre todos los componentes.

El desarrollo con procesamiento ad-hoc se planificó para ser realizado en 78 días, mientras que con el uso de TeXTracT se realizó en 50 días, siendo una mejora considerable a la hora de desarrollar una aplicación desde cero. Esta diferencia en tiempo de desarrollo no debe necesariamente ser interpretada como significativa a favor de TeXTracT, ya que pueden existir diferentes variables, como por ejemplo, la experiencia de un desarrollador, que acortan los tiempos de implementación finales. La ganancia de tiempo más importante es a futuro, para casos en los que se quiera reutilizar módulos de análisis como así también cualquier componente de la estructura de análisis.

La realización de cambios sobre el procesamiento de texto en TextChecker, es una tarea sumamente compleja en una implementación *ad-hoc*, ya que para modificar/agregar módulos de procesamiento es necesario cambiar la estructura del sistema. Mientras que la utilización de TeXTracT soporta cambios de manera simple y sin necesidad de realizar grandes cambios sobre la plataforma.

Capítulo 6 - Caso de Estudio Nº 2

El segundo caso de estudio tiene dos objetivos principales. El primero es analizar posibles dificultades que se tiene al migrar una aplicación existente a TeXTracT. El segundo objetivo es mostrar las mejoras en tiempo de ejecución y la reducción de hardware que se obtiene al utilizar TeXTracT. El sistema seleccionado para migrar se denomina REAssistant. Dicha aplicación permite identificar crosscutting concerns (aspectos de interés en los requerimientos que quedan dispersos en múltiples documentos) a partir de un conjunto de especificaciones de caso de uso.

Experimentalmente, se recabaron métricas para comparar las dos variantes de REAssistant (la versión original, y la migrada a TeXTracT). Algunas de las métricas permiten evaluar la herramienta en funcionamiento, como por ejemplo el uso de CPU, uso de HEAP, uso de memoria RAM, entre otras; mientras que existe un conjunto de métricas "estáticas" correspondientes al tamaño de espacio en disco, número de clases implicadas, cantidad de bibliotecas, etc.

6.1. REAssistant

La herramienta REAssistant (REquirements Analysis Assistant) permite buscar *crosscutting concerns* en casos de uso escritos en lenguaje natural. Esta herramienta cuenta con diferentes servicios para realizar un análisis lingüístico de los casos de uso y así lograr la identificación de *concerns* deseada [Rago2012, Rago2013, Rago2016].

Cuando el usuario carga los requerimientos en el sistema, internamente se ejecutan una serie de analizadores de texto sobre las especificaciones de casos de uso. Este análisis es llevado a cabo por un conjunto de módulos de NLP construidos sobre la plataforma UIMA. Básicamente, estos módulos descomponen los casos de uso en oraciones y buscan identificar la estructura de la oración.

Una vez que el texto ha sido analizado, REAssistant provee dos técnicas diferentes para identificar los *concerns* de manera semiautomática. La primera técnica, denominada *Semantic Clustering*, aplica un algoritmo que relaciona palabras según la semántica, es decir, teniendo en cuenta la definición y su intención dentro de un contexto. La segunda técnica, denominada *Semantically-Enriched Queries*, define un motor de búsqueda basado en reglas para que el usuario realice consultas sobre un conjunto de *concerns* predefinidos a partir de conocimiento semántico. La consultas permiten codificar el conocimiento acerca de los *concerns* y cómo se relacionan semánticamente con expresiones del lenguaje natural.

Mediante la interacción con REAssistant, la herramienta presenta un listado de *concerns* para que el usuario pueda realizar una investigación más profunda utilizando las múltiples funciones de visualización

que la herramienta provee. La idea es que el usuario pueda visualizar y filtrar con diferentes niveles de detalle, utilizando distintos colores para cada *concern*, como se muestra en la Figura 6.1.

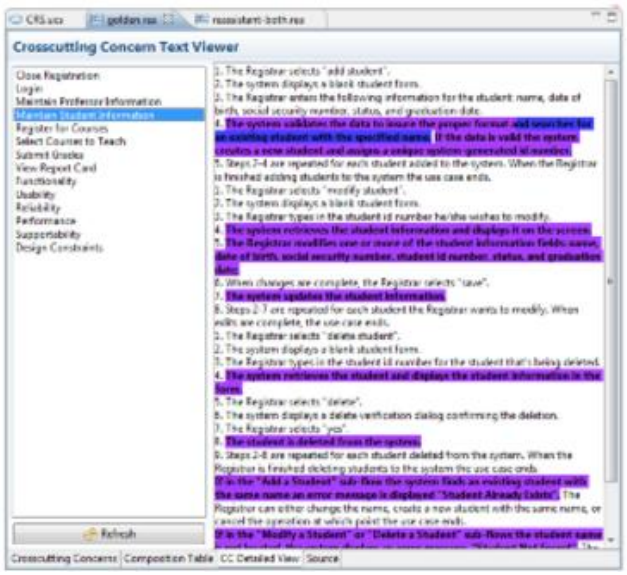


Figura 6.1 REAssistant.

6.2. Arquitectura de REAssistant

Como se mencionó anteriormente, la herramienta REAssistant está desarrollada como un conjunto de plugins de Eclipse y se basa en el framework UIMA para analizar el texto de los casos de uso. Su arquitectura, ilustrada en la Figura 6.2., separa el procesamiento de texto de la identificación de concerns y su presentación.

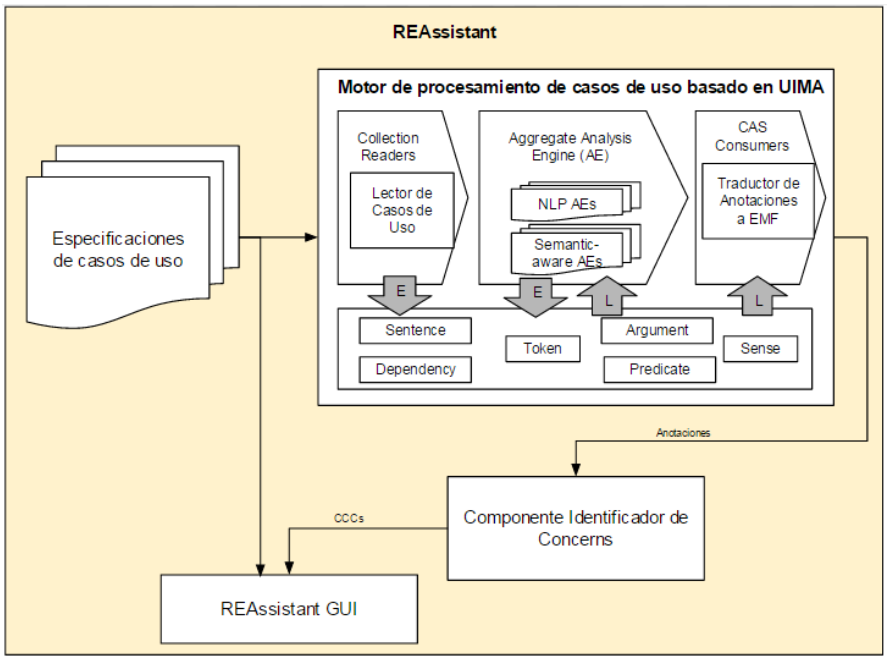


Figura 6.2 Arquitectura de REAssistant.

Los módulos de NLP permiten extraer información de las especificaciones de casos de uso ingresada. Cada uno de estos módulos genera anotaciones (marca en el texto) que indican un elemento identificado por una técnica NLP. Por ejemplo, el módulo *TokenAnnotator* genera anotaciones de tipo *Token* y las agrega al resultado parcial. Los módulos más complejos hacen uso de estas anotaciones previas para realizar el análisis, como por ejemplo, el módulo *Argument* hace referencia a los *Tokens* que se encuentran involucrados. En el Figura 6.3 se ilustran los diferentes módulos de NLP de REAssistant y las dependencias entre los mismos. La mayoría de los módulos ya se encuentran implementados de manera similar en TeXTracT. Se puede ver también que para cada técnica de identificación de *concerns* se requieren determinados módulos. Es decir, dependiendo de la técnica utilizada se hace uso de un determinado conjunto de anotaciones.

Los módulos que utilizan las técnicas de identificación de *concerns* son las siguientes:

- **Sentence-AE:** identifica las oraciones del texto.
- **Token-AE:** identifica las palabras dentro de cada oración.
- **Stopwords-AE:** marca las palabras irrelevantes.
- **Stemmer-AE** y **Lemmatizer-AE:** estos reconocen palabras que comparten la misma raíz y asignan una misma representación mínima.
- **Part-Of-Speech-AE:** identifica los roles sintácticos de una palabra dentro de una oración.
- **DependencyParsing-AE:** identifica relaciones gramaticales.
- **DomainAction-AE:** identifica las acciones de dominio de los casos de uso [13].

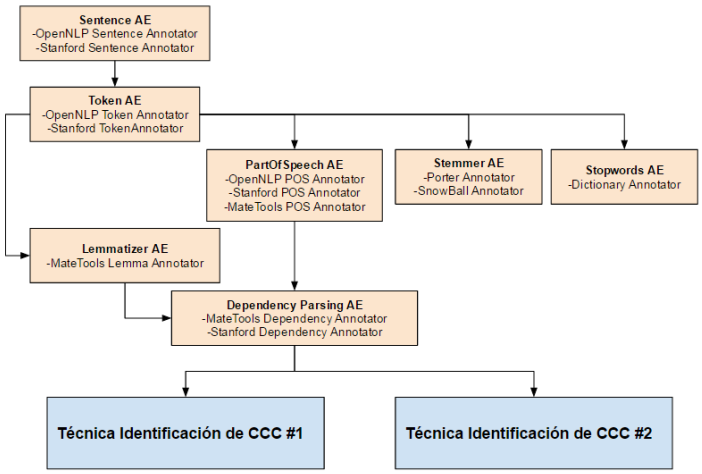


Figura 6.3 Módulos de análisis.

[13] Una acción de dominio (DA) es la abstracción de una interacción recurrente entre los actores y el sistema. Una DA puede agrupar pasos de casos de uso relacionados a la gestión de información, interacciones con los usuarios, o la presentación de datos, entre otras cuestiones. Además permite identificar abstracciones dispersas en los documentos relacionadas con atributos de calidad y concerns del dominio.

6.3. Migración de REAssistant

A continuación, se presenta la migración del procesamiento de texto *ad-hoc* en REAssistant a la invocación de los servicios de TeXTracT. Se presentarán los problemas y limitaciones que tiene el desarrollo original de la aplicación seguido de las modificaciones necesarias para integrar TeXTracT.

6.3.1 Problemas y Limitaciones

En la versión original de REAssistant los módulos de procesamiento de lenguaje natural fueron implementados y orquestados mediante el framework UIMA. Esto representa una ventaja al realizar la migración ya que TeXTracT también se encuentra implementado con UIMA. Por lo tanto la integración tanto de la entrada como la salida del proceso se realiza de manera transparente, y con pocas modificaciones.

REAssistant lleva a cabo el procesamiento de texto utilizando un conjunto de módulos NLP, en donde cada uno corresponde con una técnica de NLP. Cada una de estas técnicas son descritas a partir de modelos que se utilizan para instanciar un módulo particular. Esto trae consigo uno de los principales problemas, ya que estos modelos ocupan mucho espacio en disco. Además, cada modelo debe ser cargado en memoria cada vez que se ejecuta REAssistant, dando como resultado un consumo elevado de recursos físicos.

Otra limitación es que el procesamiento de texto *ad-hoc* en REAssistant no permite actualizar y/o agregar módulos de análisis de manera simple, siendo esta una necesidad dado que se realizan mejoras constantemente en este campo de estudio. Para poder realizar una modificación de este tipo, se debe recompilar REAssistant y volver a cargar el plugin en Eclipse.

6.3.2 Modificaciones de la Migración

Este caso de estudio implica la migración del procesamiento NLP realizado en REAssistant a TeXTracT. Para lograr esta adaptación, en primer lugar se analizaron cada uno de los módulos disponibles en TeXTracT para poder definir cuales podían reutilizarse y cuáles se deben agregar para extraer las anotaciones que permiten la identificación de *concerns*. La mayoría de estos módulos de análisis se encontraban disponibles en la herramienta y pudieron ser utilizados mediante la adaptación de los archivos XML que describen los componentes de análisis. Para los módulos que no existían en TeXTracT, se debieron cargar bibliotecas, anotadores y los archivos que utilizan estos anotadores para la ejecución de cada uno, como puede ser: el modelo, diccionarios, descriptores, etc. En primer lugar, se

debió ajustar la clase a la herencia de la interfaz *InitAnnotator* para que la herramienta pueda listar estos nuevos anotadores y hacer uso de los mismo de la misma manera que lo hace con los demás anotadores. En segundo lugar, se deben modificar los anotadores para que lean los archivos necesarios del nuevo *classpath* del servidor. En tercer lugar, se tuvo que agregar y adaptar la clase encargada del procesamiento del texto, la cual recibe el CAS correspondiente al análisis realizado hasta el momento, y agrega las anotaciones correspondientes dependiendo de la técnica descrita. De esta manera, se puede adherir a cualquier pipeline de proceso los nuevos anotadores y así obtener la salida requerida.

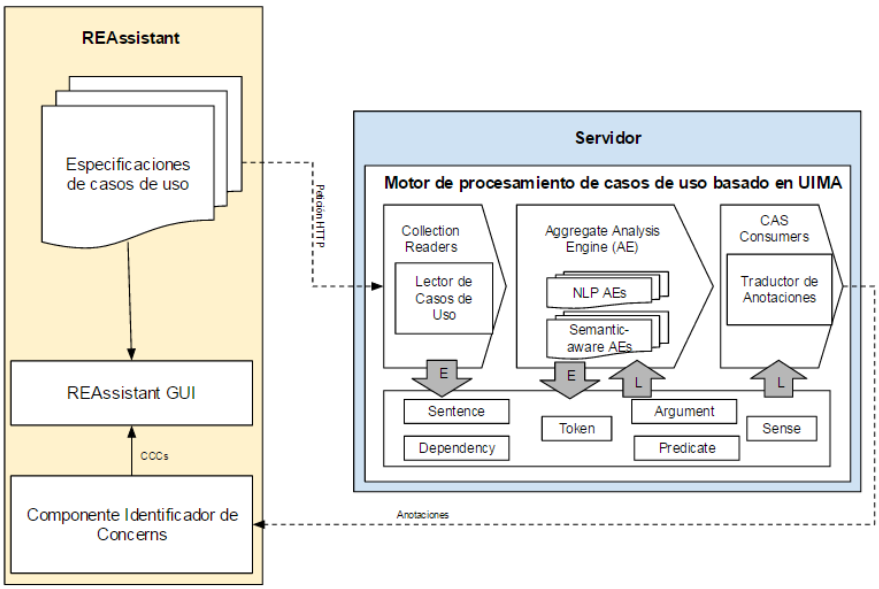


Figura 6.4 REAssistant con NLP Web Service.

En este caso particular, el módulo de proceso que se debió agregar a TeXTracT para poder llevar a cabo la identificación de *concerns* en REAssistant fue *DomainAction*. Para poder agregar este módulo se agrego el modelo correspondiente mediante el cual se describe el funcionamiento del anotador.

En la Figura 6.4, se puede observar la estructura de REAssistant con el uso del procesamiento NLP a través de servicios Web. La funcionalidad del plugin se reduce a la creación de la URL especificando cada uno de los módulos de NLP que se quieren utilizar junto con el texto que se desea analizar. Una vez que obtiene la respuesta del servidor con el conjunto de anotaciones, estos datos son procesados por el *Componente Identificador de Concerns* que se encarga de identificar los *concerns* a los que hace referencia un caso de uso en particular. Finalmente, el conjunto de concerns identificados es enviado al módulo *REAssistant GUI* que expone los resultados de manera simple para que el usuario pueda trabajar con los mismos.

6.3.3. Integración TeXTracT en REAssistant

Originalmente, para realizar el procesamiento NLP, REAssistant hacía uso de un método de UIMA que permite describir la secuencia de anotadores con los cuales se realizaba el análisis. Este método da como resultado un *string* con el conjunto de anotaciones resultante sobre el cual se realiza la identificación de *concerns*. Por lo tanto, en la migración a TeXTracT solo se debe suplantar fragmento de código que invoca el método UIMA, por una invocación al servicio Web de TeXTracT.

Tanto la información de entrada como de salida del procesamiento NLP en REAssistant es realizada mediante archivos. Por lo tanto, se utilizaron frameworks de Java que permiten crear, modificar y leer archivos de manera simple.

Para realizar esta integración se utilizó una biblioteca que permite la invocación de servicios REST y la interpretación de la respuesta de los mismo en una aplicación Eclipse. A continuación, se puede ver el código implementado con dicho propósito.

```
URL url = new URL(textractAccess);
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setRequestMethod("POST");
conn.setDoInput(true);
conn.setUseCaches (true);
conn.setDoOutput(true);
conn.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
```

La biblioteca utilizada para lograr la comunicación fue *java.net*. Esta biblioteca declara las clases *URL*[14], *HttpURLConnection*[15] y *URLConnection*[16], que posibilitan la interacción con servicios Web. Primero, mediante el uso de la *URL* se creo un objeto del mismo nombre con la dirección de acceso al servidor donde se encuentra TeXTracT. A partir de este objeto URL, se crea una instancia de *HttpURLConnection*, mediante la cual se puede realizar solicitudes HTTP. Sobre esta conexión se especifican cada uno de los parámetros de la solicitud, como por ejemplo, el tipo de solicitud (POST o GET), el uso de caché, el formato de la entrada, entre otros. Además, se envía como parámetro una variable que tiene el contenido de las especificaciones de caso de uso que se desean analizar. Con cada una de las variables para realizar la solicitud HTTP especificadas, se lleva a cabo la invocación del servicio que ejecuta el pipeline especificado. La URL correspondiente a la invocación en REAssistant es la siguiente:

[14] https://docs.oracle.com/javase/7/docs/api/java/net/URL.html

[15] https://docs.oracle.com/javase/7/docs/api/java/net/HttpURLConnection.html

[16] https://docs.oracle.com/javase/7/docs/api/java/net/URLConnection.html

$URL_BASE/RawTextCollectionReader&InitOpenNlpSentenceAnnotator&InitOpenNlpTokenAnnotator&InitMatetoolsLemmaAnnotator&InitOpenNlpPosAnnotator&InitOpenNlpChunkAnnotator&InitMatetoolsCoNLLDependencyAnnotator&InitCoNLLSRLAnnotator&InitDomainActionAnnotator

Originalmente, REAssistant utilizaba un componente escritor de colecciones de UIMA que permite escribir el contenido de un CAS en un archivo. Es así, que la aplicación crea el archivo de extensión *uima* a partir del cual se realiza la identificación de *concerns*. En la variante que utiliza TeXTracT, se hace uso de la clase *HttpURLConnection*, que hereda de la biblioteca *URLConnection* el método *getOutputStream* y permite obtener la salida del procesamiento realizado durante la conexión. Con estos datos de salida se crea un archivo con la extensión *uima*, que contiene cada una de las anotaciones que necesita REAssistant para lograr la identificación *concerns*.

6.4. Experiencias Obtenidas

Para comparar el funcionamiento original de REAssistant y la versión migrada a TeXTracT, se plantearon un conjunto de métricas mediante las cuales se pueden apreciar tanto la utilización de los recursos físicos en cada ejecución, como así también características "estáticas" del plugins. Estas métricas son: uso de CPU, uso de HEAP, uso de memoria RAM, tamaño final de los plugins, cantidad de líneas de código y cantidad de bibliotecas utilizadas en la implementación.
Con el objetivo de realizar las pruebas bajo las mismas condiciones para cada una de las variantes de REAssistant, se definió provisoriamente un escenario de ejecución, donde la secuencia de acciones a realizar es la siguiente:

- Abrir Eclipse IDE
- Generar archivo UIMA a partir de una especificación de casos de uso
- Generar un archivo REA con la identificación de *CCCs*.

Teniendo en cuenta que REAssistant/Eclipse es una herramienta desarrollada en Java, se utilizó la aplicación *VisualVM* para obtener información acerca de la utilización de recursos de la máquina virtual, con el fin de extraer métricas de su funcionamiento. Esta herramienta de monitoreo establece en tiempo real el uso de recursos de cada escenario de las dos variantes de REAssistant.
El análisis de la utilización de CPU resulta de gran importancia teniendo en cuenta que el procesamiento NLP que requiere REAssistant es complejo y genera una carga importante en términos de recursos físicos. Existen módulos de NLP que realizan un conjunto grande de acciones debido a la complejidad de los mismos, Algunos ejemplos son: la identificación de acciones de dominio o las dependencias. El Figura

6.5 ilustra el porcentaje de uso de CPU de las dos variantes de REAssistant con respecto al tiempo (expresado en segundos). Según el escenario planteado, primero se abre Eclipse IDE realizando la carga del área de trabajo en la cual se encuentra la especificación de caso de uso sobre las cuales se realizará el procesamiento. En la Figura 6.1, la marca #1 hace referencia a esta primera etapa de proceso.

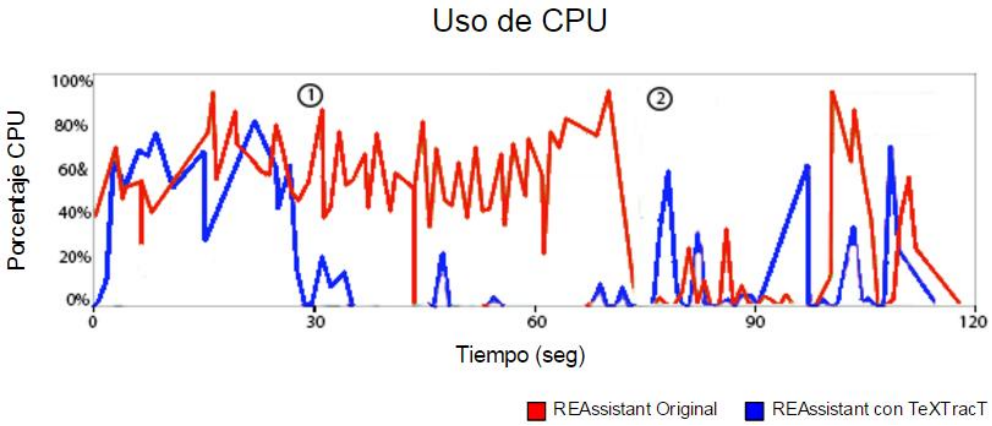


Figura 6.5

Hasta la marca #1, la carga de procesamiento es similar en ambas variantes. Esto era esperable teniendo en cuenta que es una actividad en la que el procesamiento NLP no influye directamente. Luego, se procede a la invocación de la creación del archivo *.uima* desde la especificación de casos de uso. Esta acción es el punto crítico de análisis, ya que se trata del lapso de tiempo donde se realiza el procesamiento NLP. Desde la marca #2, se ve el uso de CPU para la variante original de REAssistant no solo se sostiene en el tiempo sino que alcanza los puntos máximos de uso de CPU. Diferentemente, la variante migrada con TeXTracT, en el mismo lapso de tiempo alcanza la menor carga de CPU, dado que es el momento en el que se invoca al servicio Web que descompone y ejecuta los módulos NLP. Esta diferencia se encontraba dentro de los resultados esperados, debido a que, como se mencionó anteriormente, el proceso NLP que requiere REAssistant involucra módulos de alta complejidad. Por su parte, la versión que utiliza TeXTracT delega todo la carga de procesamiento al servidor, dejando el CPU prácticamente ocioso hasta que recibe la respuesta con las anotaciones correspondientes. La marca #2 en adelante, se presenta el comportamiento asociado a la generación del archivo *.rea* a partir de la especificación de casos de uso y el archivo *.uima* producido como resultado del análisis NLP. La Figura 6.1, se observa que la carga de CPU en esta última actividad es similar debido a que el detector de *concerns* es independiente del proceso NLP.

En segundo lugar, se evaluó la utilización de memoria RAM a lo largo del tiempo durante la ejecución del escenario de uso. La Figura 6.6 muestra que Eclipse requiere como mínimo de 2 GB de RAM independientemente de la actividad realizada por el mismo. A partir de esta referencia, se puede visualizar que la utilización de memoria no sufre mayores alteraciones durante la ejecución de REAssistant con TeXTraxT. Por el contrario, cuando el procesamiento se realiza en la misma computadora en la que se ejecuta Eclipse, la utilización de la memoria aumenta considerablemente de 2Gb a 3Gb. Es un resultado

predecible ya que se cargan los modelos en memoria de muchas de las técnicas de NLP. Además, se puede observar que la aplicación original no libera la memoria luego de realizar el procesamiento, manteniendo la memoria ocupada hasta el cierre de Eclipse.

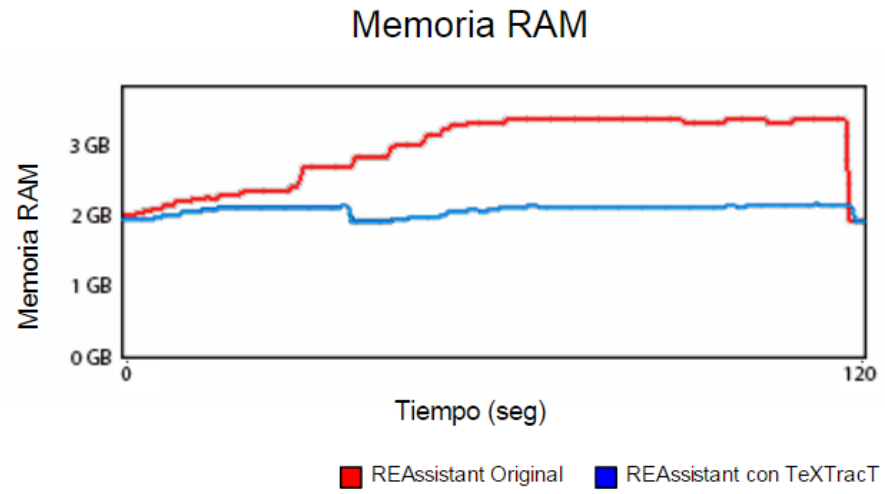


Figura 6.6

En tercer lugar, se realizó la evaluación sobre el uso de HEAP para poder observar la variación entre ambas implementaciones en términos de asignación de memoria. Teniendo en cuenta que el HEAP indica la cantidad de memoria utilizada en tiempo de ejecución, se tenía la impresión que se observarían diferencias entre las variantes, debido a la cantidad de variables y el tamaño de los modelos que describen el funcionamiento de las diferentes técnicas NLP.

En la Figura 6.7 se puede apreciar el consumo en términos de asignación de memoria de las dos variantes de REAsisstant. Originalmente, para realizar el procesamiento NLP se configuró Eclipse para asignar un máximo de 3 GB para el HEAP. En el momento en que se realiza el procesamiento NLP se utilizó el total de la memoria asignada dado que se deben cargar en memoria todos los módulos de análisis utilizados. En cambio, la variante que utiliza TeXTracT, crea en memoria las variables correspondientes a REAssistant, sin contar todo lo referido al procesamiento NLP, ya que esta carga se deriva al servidor donde se aloja la herramienta de procesamiento. Es por esta razón, que configurando un máximo de 512 MB de HEAP se puede realizar el procesamiento sin mayores complicaciones.

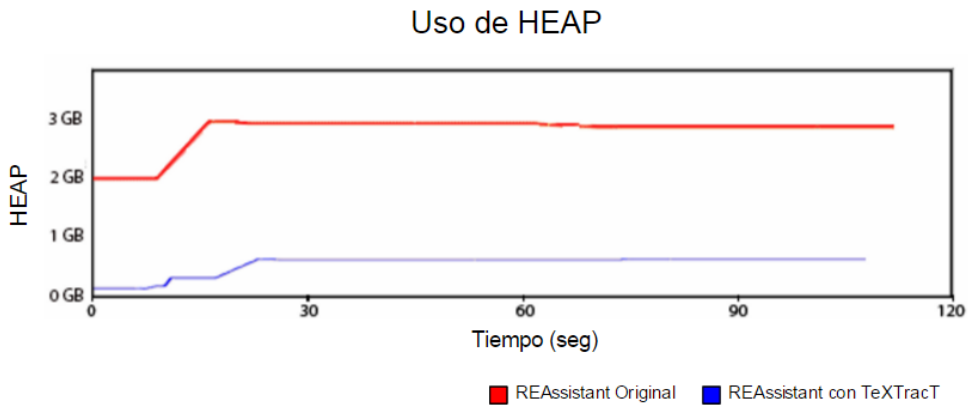


Figura 6.7

Cada una de las métricas descritas anteriormente permite evaluar la utilización de recursos durante la ejecución de REAssistant en cada una de sus versiones. Pero también, se plantearon métricas "estáticas" que se muestran en la Tabla 6.1 se indican las mejoras en la implementación de la versión que utiliza TeXTracT sobre la que realiza el procesamiento ad-hoc.

La Tabla 6.1 muestra la comparación entre las dos variantes de REAssistant. Se pueden observar varios puntos que destacan las mejoras de la variante migrada. Inicialmente se migraron los proyectos referidos al procesamiento NLP, como es el caso de *edu.isistan.daclassifier*, que trabaja sobre el anotador *CoNLLSRLAnnotator*, y *edu.isistan.uima.libs*, que contiene cada una de las librerías que utilizan los anotadores NLP. Además, se migró el proyecto *REAssistant-models* que contiene todos los modelos utilizados para instanciar los anotadores desarrollados. Esta migración tuvo un impacto importante ya que reduce el tamaño del proyecto en un 96% de la variante original de REAssistant. Se realizaron modificaciones en el proyecto *edu.isistan.uima.unified* que contiene la implementación de cada uno de los componentes de análisis junto a la clase *UIMAProcessor*, que realiza la invocación al pipeline de procesamiento NLP. En este caso, se migraron todos los componentes de análisis que se encuentran en TeXTracT. La única clase que compone el proyecto *unified* después de la migración es *UIMAProcessor*, en donde se sustituyó la ejecución del pipeline de módulos, por la invocación al servicio Web de TeXTracT.

	Líneas de Código	Cantidad de paquetes	Cantidad de Clases	Tamaño (KB)
edu.isistan.daclassifier (variante original)	257473	98	1633	12.565
edu.isistan.dal	2615	7	43	55
edu.isistan.dal.edit	1625	2	19	51
edu.isistan.dal.editor	1620	1	14	63
edu.isistan.reassistant	5110	14	43	355
edu.isistan.reassistant.ccdetector	3007	8	21	594
edu.isistan.reassistant.ccdetector.model	1232	3	19	33
edu.isistan.reassistant.ccdetector.model.edit	637	1	7	20
edu.isistan.reassistant.evaluator	2427	11	38	556
edu.isistan.reassistant.model	1764	3	26	44
edu.isistan.reassistant.model.edit	875	1	10	25
edu.isistan.ucseditor	4071	7	36	152
edu.isistan.uima.editor	220	1	2	9

edu.isistan.uima.libs (variante original)	0	0	0	19.171
edu.isistan.uima.model	9013	25	165	199
edu.isistan.uima.model.edit	5295	9	60	127
edu.isistan.uima.unified (variante original)	11062	27	144	341
edu.isistan.uima.unified (variante migrada)	51	1	1	50
edu.isistan.uima.wizards	465	3	7	20
REAssistant-models (variante original)	0	0	0	1.099.649
Total (variante original)	308511	221	2287	1.134.029
Total (variante migrada)	40027	97	511	2.353
Reducción	87,03%	56,11%	77,66%	99,79%

Tabla 6.1. Código involucrado en el desarrollo de REAssistant en sus dos variantes.

Finalmente, la reducción del código en el paquete *unified*, junto a la migración de las bibliotecas utilizadas, y los modelos NLP, que se encuentran en el servidor que aloja TeXTracT, el tamaño de la variante original pasa de ser 1.134.029 KB a 2.353 KB. En términos de cantidad de código, se puede ver en la Tabla 6.1, que mejora notablemente, disminuyendo un 87% la cantidad de líneas de código. Además, se utiliza un 77,66% menos de clases a lo largo del proyecto, obteniendo como resultado un proyecto simple y fácil de mantener. Esta migración permite derivar el procesamiento de texto al servidor, liberando de esta manera recursos de la máquina que ejecuta REAssistant. Las métricas obtenidas, demuestran que la disminución de esfuerzo en cada uno de los recursos es considerable teniendo en cuenta que el procesamiento de texto es una tarea de gran esfuerzo. Una de las particularidades encontradas a partir de las pruebas, fue el tiempo que demora el procesamiento en cada caso. Desde un primer momento se estimó que el tiempo de procesamiento en el servidor sería significativamente menor, sin embargo, ambas ejecuciones rondaron los dos minutos.

En conclusión, la migración del procesamiento de una herramienta que realiza procesamiento NLP a TeXTracT es una práctica importante para llevar a cabo, ya que la herramienta generada consume menos recursos, contiene menos cantidad de código, y el tamaño total se reduce considerablemente. También, la adaptación se realiza de manera simple sin necesidad de grandes esfuerzos de los desarrolladores para invocar los servicios Web, y trabajar con la salida obtenida.

Capítulo 7 - Conclusión

En este trabajo se desarrolló una plataforma Web de procesamiento de lenguaje natural que permite consumir técnicas de análisis de texto desde diversos tipos de aplicaciones. A partir de las herramientas existentes en la actualidad, y las características de las aplicaciones que hacen uso de este tipo de procesamiento, se propuso el desarrollo de una herramienta flexible, configurable, extensible e interoperable, para abarcar las limitaciones existentes hasta el momento. Con esta finalidad, se logró desarrollar TeXTracT, una herramienta que permite agregar módulos de análisis y/o parametrizar los componentes de análisis disponibles, y así obtener el resultado deseado.

Para poder desarrollar la herramienta deseada, se realizó un diseño basado en el patrón *Broker*, con un componente de acceso a la herramienta que permite independizar los servicios NLP de las aplicaciones que solicitan el procesamiento. Este componente es el encargado de recibir la solicitud de proceso, identificar los módulos requeridos y orquestar los componentes de análisis combinando la entrada y la salida de información de los mismos. Ante la necesidad de manipular cada uno de los módulos de NLP de la misma manera, se implementó una interfaz mediante la cual se adaptó cada uno de los componentes de análisis. Este diseño dio la posibilidad de agregar analizadores adicionales de manera simple mediante la instanciación de un adaptador parametrizable en el cual se puede especificar el modelo de análisis a seguir. Adicionalmente, la herramienta desarrollada permite listar cada uno de los componentes disponibles en cada momento, utilizando un mecanismo de búsqueda de analizadores dentro de las clases que componen el sistema.

La herramienta fue evaluada experimentalmente utilizando dos casos de estudio. El primero consiste en el desarrollo de una aplicación que consuma servicios NLP con el propósito de demostrar que la arquitectura de TeXTracT permite crear diversos tipos de aplicaciones de manera rápida y sencilla, soportando la evolución de dichos sistemas en el tiempo. El sistema en cuestión es un sitio Web para corregir texto escrito en Inglés. Para extraer métricas que reflejen el esfuerzo desarrollo con los servicios de NLP, se realizó una división del desarrollo de un conjunto de tareas fijas. Inicialmente, se estimaron los tiempo de desarrollo de cada tarea en base a nuestra experiencia propia. A partir de esta referencia, se comparó el tiempo correspondiente al desarrollo del corrector utilizando TeXTracT y el tiempo de desarrollo en una aplicación que realice el procesamiento *ad-hoc*. La implementación del corrector ortográfico involucra la generación de reglas a partir de un identificador de patrones de elementos UIMA (lenguaje RUTA). Esta tecnología simplifica el desarrollo y la incorporación de las reglas ortográficas a TeXTracT dado que está basado también en UIMA, dando la posibilidad de acoplarlos a cualquier pipeline de proceso y aprovechar los módulos de NLP pre-existentes.

Una vez desarrollado el módulo del corrector, se implementó una interfaz de usuario que da la posibilidad de ingresar el texto que se desea corregir y mostrar los errores encontrados, remarcando el fragmento de

texto erróneo, como así también, el tipo de error al que se refiere. Los resultados obtenidos de esta evaluación fueron prometedores debido a la flexibilidad de la herramienta para incorporar nuevos módulos de análisis, como así también la facilidad con la que se generó una secuencia de proceso para invocar mediante un simple llamado a un servicio Web de TeXTracT. La materialización de la herramienta fue bastante rápida, ya que con algunos conocimientos básicos de HTML y Javascript fue posible invocar el procesamiento y exponer los resultados. Se esperaba una mejora significativa en cuestión de tiempo, pero no fue así en este caso de estudio. La mejora más importante se obtuvo en la etapa de diseño e implementación, teniendo en cuenta que el uso de TeXTracT define una arquitectura fija y permite la utilización de módulos NLP sin necesidad de crear una estructura de proceso propia. Sin embargo, en las etapas de desarrollo restantes la diferencia en tiempo de desarrollo fue mínima dado que las tareas son independientes de la forma de implementación seleccionado. El tiempo de desarrollo con el procesamiento de texto *ad-hoc* fue de 78 días, mientras que utilizando TeXTracT llevó 50 días. Se pudo determinar que una vez estudiado el funcionamiento de la API de procesamiento se puede implementar una herramienta de manera simple y en poco tiempo. Otra ventaja significativa fue la facilidad con la que se puede agregar componentes de análisis, entre ellos, los módulos identificadores de errores, dando la posibilidad de extender la funcionalidad del corrector ortográfico cuando se desee.

Mediante el segundo caso de estudio, se buscó establecer los costos asociados a la migración de una aplicación existente a los servicios NLP de TeXTracT. Para esto, se utilizaron un conjunto de plugins de Eclipse que materializan la herramienta denominada REAssistant. Esta herramienta realiza la identificación de *crosscutting concerns* a partir de un conjunto de especificaciones de casos de uso. Para realizar la migración del procesamiento de texto de dicha herramienta se debieron agregar en TeXTracT algunos módulos de análisis necesarios para el correcto funcionamiento del componente identificador de *concerns*. En REAssistant, se realizaron las modificaciones correspondientes para cambiar el procesamiento *ad-hoc*, por la invocación al servicio Web de NLP.

Para comparar el funcionamiento de ambas implementaciones, se recolectaron métricas tales como utilización de memoria, uso de CPU, tamaño de plugins, entre otras, permitiendo así determinar la diferencia relacionada con el consumo de recursos físicos requeridos por el proceso. Se planteó un circuito de tareas a realizar en ambas aplicaciones (original y migrada) tomando las mediciones correspondientes en tiempo de ejecución. Como resultado, se pudo observar una mejora significativa en el consumo de recursos en la herramienta que realizaba el procesamiento a través de TeXTracT. Esto se debe a que la carga de proceso que implica el análisis de texto, con los módulos necesarios para identificar *concerns*, es derivada al servidor. Durante el procesamiento NLP el uso promedio del CPU de la aplicación original es aproximadamente de 70%, mientras que con el uso de TeXTracT el uso máximo fue del 20%. Se pudo observar también que la utilización de memoria no sufre mayores alteraciones durante la ejecución de REAssistant con TeXTraxT, manteniéndose en 2 GB a lo largo de proceso. Por el contrario, en la versión original el uso de memoria alcanzó los 3 GB durante el funcionamiento del plugin. Además, la versión original de REAssistant contiene en su interior cada uno de los archivos que

describen el funcionamiento de cada una de las técnicas de NLP, como así también las bibliotecas Java necesarias. En la versión Web, estos recursos se encuentran en el servidor, dando como resultado un plugin con la misma funcionalidad pero mucho más liviano. El tamaño de la aplicación original es de 97 MB, mientras que la versión que utiliza TeXTracT tiene un tamaño de 63 MB.

7.1. Contribuciones

Durante el desarrollo de los casos de estudio, se logró observar un conjunto de ventajas que posee TeXTracT con respecto a otras soluciones para realizar NLP en una aplicación determinada. A continuación se listan las contribuciones más importantes de nuestra solución:

- La herramienta permite utilizar módulos NLP que son implementados por distintos grupos de especialistas. Además, los desarrolladores tienen la oportunidad de implementar e integrar en TeXTracT sus propios módulos rápidamente. El uso del patrón Broker en conjunto con la tecnología UIMA da facilidades a la hora de querer agregar o quitar módulos NLP de la herramienta TeXTracT.
- Las aplicaciones que hacen uso de TeXTracT pueden configurar los módulos requeridos para lograr el procesamiento NLP necesario según el dominio. Cada una de estas aplicaciones puede utilizar los módulos NLP que se encuentran disponibles en la herramienta y configurar módulos individualmente según su conveniencia.
- Las aplicaciones no tienen que realizar múltiples llamados para resolver un procesamiento de texto que involucra varios módulos NLP. Particularmente, la herramienta provee funcionalidad para componer módulos individuales en un pipeline siguiendo los mecanismos de comunicación de UIMA.

7.2. Limitaciones

Luego de desarrollar la herramienta TeXTracT y evaluarla sobre los distintos casos de estudio, se identificaron ciertas limitaciones. A continuación, se listan las diferentes limitaciones encontradas:

- Se esperaba una velocidad mayor de ejecución de un pipeline de desarrollo comparado con un desarrollo *ad-hoc* que realiza el mismo procesamiento. Esto se debe a que la estructura de TeXTracT agrega un overhead importante con respecto a un desarrollo *ad-hoc* para una

aplicación determinada. Mayormente, este problema surge de la cantidad de código extra que se genera por hacer uso de la tecnología UIMA en TeXTracT, a diferencia de la utilización de módulos individuales incluidos en las bibliotecas originales, como lo hacen los desarrollos a medida.

- Debido a que la invocación a un pipeline de proceso es realizada mediante servicios REST, no es posible almacenar estados, es decir, no se pueden almacenar los anotadores creados. Esto trae como consecuencia que en cada ejecución tengan que ser cargados cada uno de los componentes de análisis. La particularidad de este problema está ligada a que muchos módulos de NLP implementados tienen archivos que describen el comportamiento de los mismos, y deben ser cargados en memoria en cada ejecución.
- Para el manejo de la información de los textos se utilizó el sistema UIMA y así poder extraer los datos específicos en formato XML. Los propios analizadores, como así también cada uno de los componentes utilizados por estos, están declarados bajo esta estructura. Esto genera una dependencia de la herramienta a UIMA que implica una limitación teniendo en cuenta que existen o pueden existir sistemas mejores. Con el fin de realizar la comunicación entre los componentes NLP se utilizan archivos en formato XML que contienen cada una de las anotaciones resultantes. A medida que avanza el procesamiento, estos archivos se hacen muy grandes provocando una transferencia de datos lenta.
- La entrada de texto a la herramienta es realizada a través del servicio Web. En la presente versión, se admiten tres formatos de entrada: texto plano, archivos UCS y archivos XMI. Esto se debe a que cada entrada debe ser interpretada por un lector de colecciones. Esto es una problemática importante, ya que se limitan los tipos de contenidos que puede ser ingresados para realizar NLP.

7.3 Trabajo Futuro

La herramienta desarrollada propone una solución novedosa y práctica para aquellas aplicaciones que necesitan procesamiento NLP. Es por esta razón que existen diferentes líneas de trabajo futuro que se derivan de esta tesis tales como:

- **Escalabilidad:** Si bien UIMA es una tecnología escalable, no fue uno de los puntos en los cuales se hizo hincapié durante el desarrollo de la herramienta TeXTracT. Es importante lograr la escalabilidad de la herramienta para así aumentar la cantidad de aplicaciones que quieran hacer uso de TeXTracT en simultaneo. Para esto se debe realizar un balanceo de cargas para evitar los fallos producidos y la caída del servidor con componentes redundantes.

- **Soportar diferentes formatos de entrada:** Se deberían desarrollar múltiples lectores de colecciones que permitan ingresar a la herramienta diversos formatos de documentos de entrada. Para esto se deben desarrollar diversos lectores de colecciones que interpreten múltiples formatos de entrada y así volcar el contenido en la estructura de análisis.
- **Optimizar la creación de componentes de análisis en tiempo de ejecución:** La creación de una gran cantidad de archivos de tamaño considerable en cada ejecución provoca una reducción de performance de la herramienta. Para mejorar este aspecto, se podrían evaluar diferentes maneras de mantener en memoria alguno de los analizadores utilizados recientemente sin que sea necesaria la carga por completo de todo el pipeline de proceso cada vez que se realice una invocación. Por ejemplo, los anotadores de oraciones y *tokens* se utilizan en la mayoría de los pipelines de proceso, razón por la cual se podría tener pre-cargados en memoria para optimizar el funcionamiento.

Bibliografía

[Aylien2015] Aylien text analysis API. http://aylien.com/text-api, Fecha de acceso: 15-11-2015.

[IBM2015] IBM developer cloud. https://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud, Fecha de acceso: 15-11-2015

[Language2015] Language tools API. https://www.languagetool.org, Fecha de acceso: 18-11-2015

[Meaning2015] Meaning cloud. https://www.meaningcloud.com, Fecha de acceso: 20-11-2015

[Jersey2015] Guia de usuario Jersey. https://jersey.java.net/documentation/latest/index.html, Fecha de acceso: 22-07-2015

[UIMAFit2015] Guía UimaFit. https://uima.apache.org/d/uimafit-current/tools.uimafit.book.html, Fecha de acceso: 18-12-2015

[W32015] Guía Breve de Servicios Web. http://www.w3c.es/Divulgacion/GuiasBreves/ServiciosWeb, Fecha de acceso: 10-10-2015

[Bass2012] Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. SEI Series in Software Engineering, Addison-Wesley Professional, 3rd edn. (October 2012)

[Dale2015] Dale Robert. NLP meets the cloud. Natural Language Engineering, pp 653-659, 10.1017/S1351324915000200 (2015).

[Ferrucci2004] Ferrucci, D., Lally, A.: UIMA: an architectural approach to unstructured information processing in the corporate research environment. Natural Language Engineering 10(3-4), 327–348 (2004)

[Kluegl2015] Kluegl, P., Toepfer, M., Beck, P.D., Fette, G., Puppe, F.: UIMA RUTA: Rapid development of rule-based information extraction applications. Natural Language Engineering pp. 1–40 (2015), 10.1017/S1351324914000114

[Rago2013] A. Rago. Herramientas para la identificación de crosscutting concerns en especificaciones de casos de usos . Tesis Master en Ingeniería de Software, Universidad UNICEN Instituto de investigación ISISTAN (2013).

[Rago2016] Rago, A., Marcos, C., Diaz-Pace, A.: Assisting requirements analysts to find latent concerns with REAssistant. Automated Software Engineering 23(2), 219–252 (2016)

[Marcos2016] Rago, A., Marcos, C., Diaz-Pace, A.: Opportunities for analyzing hardware specifications with NLP techniques. In: 3rd Workshop on Design Automation for Understanding Hardware Designs (DUHDe'16). Design, Automation and Test in Europe Conference and Exhibition (DATE'16), Dresden, Germany (2016)

[Rago2012] Rago, A., Marcos, C., Diaz-Pace, A.: Text analytics for discovering concerns in requirements documents. In: XIII Argentine Symposium on Software Engineering (ASSE'12). La Plata, Argentina (September 2012)

[Sateli2012] Sateli, B., Angius, E., Rajivelu, S.S., Witte, R.: Can text mining assistants help to improve requirements specifications? In: Mining Unstructured Data (MUD 2012). Canada (2012)

[Gamma1995] Gamma, E. Design patterns. Reading, Mass.: Addison-Wesley. (1995).

[UIMA] Apache UIMA™ Development Community. UIMA Tutorial and Developers' Guides, Versión 2.8. (2015)

[Gobinda2005] Gobinda G. Chowdhury. Natural Language Processing. Dept. of Computer and Information Sciences University of Strathclyde, Glasgow, UK. (2005).

[Manning1999] Chris Manning y Hinrich Schütze. Foundations of Statistical Natural Language Processing. Cambridge, MA. (1999).

[Kumar2011] Ela Kumar. Natural Language Processing. Nueva Delhi, India. (2011).

[Allen2006] P. Allen, S. Higgings, P. McRae, H. Schlamman. "Service Orientation : Winning Strategies and Best Practices". Cambridge University Press New York, NY, USA (2006).

[Marks2006] Eric A.Marks, Michael Bell. "Service-Oriented Architecture : A Planning and Implementation Guide for Business and Technology". University of Alabama, Alabama, USA (2006).
[Gheith2016] A. Gheith et al., IBM Bluemix Mobile Cloud Services IBM Journal of Research and Development, pp. 7:1-7:12, (3 2016), 10.1147/JRD.2016.2515422.

[Arnold2016] B. Arnold , Building the IBM Containers cloud service, IBM Journal of Research and Development, pp. 9:1-9:12, (3 2016), 10.1147/JRD.2016.2516943.

[Sandoval2009] Sandoval, J., RESTful Java Web Services: Master Core REST Concepts and Create. Packt Publishing Ltd, (11 2009)

[Graham2005] Graham S., Davis D., Simeonov S., Danick G., Brittenham P., Nakamura Y. Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI, Sams Publishing Second Edition (2005).

[Mulligan2009] G. Mulligan, A comparison of SOAP and REST implementations of a service based interaction independence middleware framework, Proceedings of the 2009 Winter Simulation Conference (WSC), (2009), pp. 1423-1432, 10.1109/WSC.2009.5429290.

[Huang2003] Nan-Chao Huang. A Cross Platform Web Service Implementation Using SOAP By Submitted in partial fulfillment of the requirements , Master of Science in Computer and Information Science Knowledge Systems, Institute Skokie: 2003, IL 60076 (2003).

[Ogren2008] P. V. Ogren, P. G. Wetzler and S. J. Bethard. ClearTK: A UIMA Toolkit for Statistical Natural Language Processing, Workshop Towards Enhanced Interoperability for Large HLT Systems: UIMA for NLP (2008).

[Toepfer2014] Toepfer M., Kluegl P., Beck M., Fette G., puppe F. UIMA Ruta Workbench: Rule-based Text Annotation, The 25th International Conference on Computational Linguistics (2014)

Apéndice A

Reglas para Detectar Problemas Gramaticales y Ortográficos

A.1. Módulo NounPronounAgreement

Este módulo verifica la correspondencia entre sujetos y pronombres dentro de una oración. En el siguiente fragmento de código vemos la declaración del tipo de error, en este caso denominado "*NounPronounError*" con su atributo para la descripción. Además, se puede ver la declaración de componentes auxiliares "*SingularPronoun*" y "*PluralPronoun*" en las líneas 9 y 11 respectivamente, que permiten simplificar la escritura del script.

```
4  WORDLIST SingularIndefinitePronounsList = 'singular_indefinite_pronouns.txt';
5
6  DECLARE NounPronounError(STRING desc);
7
8  //PRONOMBRES
9  DECLARE SingularPronoun;
10 Token{->MARK(SingularPronoun)}<-{(Token.pos == "PRP" | Token.pos == "PRP$");};
11 DECLARE PluralPronoun;
12 Token{INLIST(PluralPronounsList, Token.lemma)->MARK(PluralPronoun), UNMARK(SingularPronoun)};
```

Estas reglas se ejecutan para cada "*Token*" dentro del texto, creando una anotación del tipo *SingularPronoun* si el atributo POS es PRP o PRP$ (correspondientes a pronombres personales o pronombres posesivos, respectivamente). En la línea 12 se puede ver una regla aplicada a cada "*Token*" la cual indica que de deben marcar todos los pronombres del listado "*PluralPronounsList*" como pronombres singulares. Acto seguido, se declara el tipo "*PluralPronoun*" el cual hace referencia a los pronombres plurales. En la línea 12, se crea una anotación de pronombre plural y se borra, en caso de existir, la anotación de pronombre singular sobre el mismo "*Token*", ya que la identificación a partir de las etiquetas PRP y PRP$ puede referirse a un pronombre plural.

Inicialmente, se confeccionó una lista de posibles pronombres plurales en Inglés y se la almacenó en un archivo de texto. Luego, utilizando el predicado INLIST de RUTA, el cual permite detectar si una palabra o propiedad de una anotación está dentro de un archivo de texto, se verifica si el lema de una anotación *Token* pertenece al listado de pronombres plurales. Luego de identificar los pronombres y crear las respectivas anotaciones, se procede a identificar los sustantivos.

```
16 //Singular NOUN
17 DECLARE SingularNoun;
18 Token{Token.pos == "NN"->MARK(SingularNoun)};
19 Token{Token.pos == "NNP"->MARK(SingularNoun)};
20
21 //Plural NOUN
22 DECLARE PluralNoun;
23 Token{Token.pos == "NNS"->MARK(PluralNoun)};
24 Token{Token.pos == "NNPS"->MARK(PluralNoun)};
25
26 // AND. PLURAL NOUN
27 Argument{->MARK(PluralNoun)}<-{(
28     CoNLLDependency.relation == "NMOD"
29     Token.lemma == "and"
30     CoNLLDependency.relation == "NMOD"
31 );};
```

En este fragmento de código se anotan los sujetos singulares y plurales. Al igual que para los pronombres, esta tarea se lleva a cabo para simplificar la escritura y lectura de las reglas. Los sujetos singulares son identificados mediante la búsqueda de las etiquetas "NN" y "NNP" en el atributo POS de los *Token*, para sujetos singulares y propios, respectivamente. Los sujetos plurales son identificados mediante la búsqueda de las etiquetas "NNS" y "NNPS" de los *Token*, para sujetos plurales y sujetos plurales propios, respectivamente. En el caso que la oración presente un sujeto compuesto, conformado por dos sujetos singulares unidos por una conjunción, se utiliza una estrategia alternativa para identificarlo. Desde un punto de vista gramatical, el sujeto compuesto debe considerarse como un sustantivo plural ya que combina dos sujetos singulares. La alternativa para encontrar este tipo de sujetos aprovecha las anotaciones de NLP generadas previamente, específicamente, aquellas enfocadas en las dependencias entre palabras (modificadores de sustantivos) y el análisis de roles semánticos (argumentos y predicados). Sobre las anotaciones de argumentos se verifican la existencia de dos sustantivos con sus respectivos modificadores unidos por una conjunción. Utilizando una línea de razonamiento análoga, cuando se tiene una oración con dos sustantivos unidos por una disyunción, se anota el sujeto compuesto como singular.

Una vez que se anotaron las pronombres y sujetos, las reglas para detectar los errores gramaticales son sencillas. En el siguiente fragmento de código se muestra la identificación de errores cuando dentro de una oración existe una sujeto singular y un pronombre plural.

```
40 // RULES1. Singular Noun + Plural Pronoun
41 Sentence{->CREATE(NounPronounError, "desc" = "Sentence with singular noun and plural pronoun")}<-{(
42     Argument{Argument.label == "A0", IS(SingularNoun)} Predicate{Predicate.passiveVoice == false}
43     Argument{Argument.label == "A1", CONTAINS(PluralPronoun)});
44 };
```

Esta regla crea anotaciones "*NounPronounError*" a partir de las anotaciones de tipo "*Argument*". Este tipo de anotaciones indica que si el valor de la etiqueta "*label*" es "*A0*", el argumento hace referencia al sujeto de la oración. Mientras que si el valor de la etiqueta "*label*" es "*A1*", se trata de un argumento que hace referencia al objeto de la oración. A partir de esta información, se escribieron las reglas para anotar *NounPronounError* reconociendo el sujeto de una oración (A0) y su correspondiente objeto (A1). Sobre el sujeto de la oración se utilizar el predicado *IS* de RUTA que permite saber si el componente analizado

se encuentra anotado con una etiqueta particular, en este caso es *SingularPronoun*. Sobre el objeto de la oración se utiliza el predicado *CONTAINS* de RUTA que verifica la existencia de una anotación dentro del componente que está siendo analizado. En este caso, se verifica la existencia de un *PluralPronoun* dentro del argumento con *label* igual a *A1*. Se puede ver entre los argumentos *A0* y *A1* una condición sobre el predicado que une estos dos componentes. Esto se lleva a cabo para comprobar que la frase no esté escrita en voz pasiva, dado que en ese caso la regla gramatical no se cumple.

La regla que verifica que un sujeto plural y un pronombre singular es muy similar al caso anterior, pero altera ciertos parámetros en la lista de condiciones de la regla. La lista de condiciones verifica que el argumento que actúa como sujeto sea un "*PluralNoun*", que el predicado principal de la oración esté escrito en voz activa y cuyo argumento de objeto directo tenga pronombres en singular.

A.2. Módulo StartWithConjunction

Este módulo permite identificar aquellas oraciones que comienzan con una conjunción.

```
6  DECLARE Conjuntion;
7  Token{->CREATE(Conjuntion)}<-{Token.pos == "IN";};
8
9  DECLARE StartWithConjunction(STRING desc);
10 Sentence{STARTSWITH(Conjuntion) ->CREATE(StartWithConjunction, "desc" = "A sentence may not start with a conjunction.")};
```

Primero, se identifican aquellos *Tokens* cuyo atributo POS sea "IN", los cuales se corresponden a conjunciones, y se los marca con la anotación "*Conjuntion*". Acto seguido, se declara el tipo de error *StartWithConjunction* y su correspondiente atributo de descripción. Haciendo uso del predicado *STARTSWITH* la cual indica si un componente comienza con una determinada anotación, se verifica que la oración presente el problema con la conjunción. Si la oración efectivamente comienza con una conjunción, entonces esta es marcada con la anotación *StartWithConjunction*.

A.3. Módulo DoubleNegative

Este módulo permite identificar oraciones que presenten síntomas de doble negación. Con este propósito, se compiló una lista de palabras con sentido negativo denominada *NegativeList*.

```
4  DECLARE DoubleNegative(STRING desc);
5
6  Sentence{->CREATE(DoubleNegative,"desc" = "This sentence have double negation.")}<-{(
7      Token{INLIST(NegativeList, Token.lemma)} # Token{INLIST(NegativeList, Token.lemma)}
8  );};
9
10 Sentence{->CREATE(DoubleNegative,"desc" = "This sentence have double negation.")}<-{(
11     Argument.label == "AM-NEG" # Token{INLIST(NegativeList, Token.lemma)}
12 );};
```

El script de RUTA desarrollado busca aquellas oraciones que contengan dos palabras existentes en la lista de palabras negativas. Para lograr este objetivo se utiliza el predicado *INLIST* que verifica la existencia del lema del *Token* dentro de la lista *NegativeList*. La primer parte del script busca aquellas oraciones que contienen dos *Token* con lemas negativos según el listado de negaciones, sin importar las palabras que se encuentren en el medio. Para poder omitir las palabras que unen las dos negaciones, se utiliza el símbolo numeral (#), que en lenguaje RUTA significa "*wild card*". La segunda parte del script complementa los resultados anteriores analizando los argumentos negativos de una oración. Específicamente, el script verifica que en la misma oración exista un argumento de etiqueta "AM-NEG", que refiere a una estructura de negación, y una palabra de la lista de negativas. En ese caso, crea una anotación para marcar el error.

A.4. Módulo SpellingMistakes y Módulo UpperCaseWords

Los módulos para verificar la existencia de errores de escritura y palabras que deben escribirse con mayúscula son similares, y por lo tanto, fueron agrupados durante el desarrollo. Ésto ocurre porque tanto los errores de escritura como las palabras que deben escribirse con mayúscula pueden ser detectados utilizando listas de palabras precargadas en archivos de texto y utilizando el predicado INLIST provisto en RUTA.

Tanto los errores de escritura comunes, como las palabras que siempre van escritas en mayúscula, funcionan en base a una lista de palabras.

```
3 WORDLIST SpellingMistakesList = 'common_misspelling.txt';
4
5 DECLARE SpellingMistakes(STRING desc);
6 Token{INLIST(SpellingMistakesList)->CREATE(SpellingMistakes,"desc" = "Spelling error.")};
```

Como se puede ver en este fragmento de código, se declara una *WORDLIST* llamada *SpellingMistakesList* la cual carga un listado de palabras desde un archivo de texto. En la línea 5, se puede ver la declaración de la anotación *SpellingMistakes* correspondiente a los errores de escritura. Mediante el uso del predicado *INLIST* se verifica la existencia de un *Token* determinado dentro del listado de palabras con errores de escritura. En caso de cumplir esta condición, se genera una anotación del tipo *SpellingMistakes*.

A diferencia de los *SpellingMistakes*, para la anotación *UpperCaseWord* solamente se cambió el tipo de anotación y el archivo utilizado para generar la lista de palabras sobre la cual se evalúan los *Token* ("uppercase_words.txt"). Asimismo, para que la regla sea más eficiente, el predicado *INLIST* es aplicado sobre cada una de las palabras en minúscula que contiene el texto, identificada mediante las anotaciones *SW*.

```
3  WORDLIST UpperCaseList = 'uppercase_words.txt';
4
5  DECLARE UpperCaseWords(STRING desc);
6  SW{INLIST(UpperCaseList)->CREATE(UpperCaseWords,"desc" = "This word must be uppercase.")};
```

A.5. Módulo SubjectVerbAgreement

Este módulo permite verificar la correspondencia entre el sujeto y el verbo de una oración. Básicamente, la idea para detectar este tipo de error es aprovechar las anotaciones de dependencias entre palabras y extraer la información que permite verificar la existencia de errores de este tipo particular. El siguiente script permite analizar las anotaciones *CoNLLDependency* cuyo tipo de relación sea *SBJ*, que denota la vinculación de dos palabras donde una es el sujeto de la otra. Para cada una de estas anotaciones se debe verificar que las partes de dicha dependencia (es decir, los Tokens) cumplan el acuerdo entre sujeto y verbo. Esto significa que la fuente y el destino o bien son un verbo plural y sujeto singular, o un verbo singular y un sujeto plural.

```
3  DECLARE SubjectVerbAgreement;
4
5  //Basic rules.
6  CoNLLDependency{->MARK(SubjectVerbAgreement)}<-{(
7      CoNLLDependency.relation == "SBJ" &
8      (
9          (CoNLLDependency.source.pos == "VBP" & CoNLLDependency.target.pos == "NN")|
10         (CoNLLDependency.source.pos == "VBZ" & CoNLLDependency.target.pos == "NNS")
11     ));
12 };
```

En esta regla también se buscó simplificar la lectura y escritura del código mediante el resumen de algunas anotaciones. Por ejemplo, como se puede ver en el siguiente script, se anotaron como sujetos todas la anotaciones *Chunks* con atributo *chunk* de valor *NP* (sintagmas nominales), como así también se anotaron los *Tokens* con atributos *POS* de valor *NNP* (nombre propio singular).

```
DECLARE Sujeto;
Chunk{->MARK(Sujeto)}<-{Chunk.chunk == "NP";};
Token{->MARK(Sujeto)}<-{Token.pos == "NNP";};
```

Otro caso que se tuvo en cuenta al desarrollar este script fueron las oraciones subordinadas. Una oración subordinada es aquella que depende estructuralmente del núcleo de otra oración, y pierde el sentido si se la saca de la oración principal. Un ejemplo puede ser “Juan, quien trabajó en la empresa, me visitara por la tarde.”, donde la oración subordinada es “quien trabajó en la empresa”.

Para esta regla, las oraciones subordinadas no son analizadas, ya que pueden generar confusiones a la hora de verificar la correspondencia entre verbos y sujetos. Es por eso, que el siguiente script declara un tipo *Subordinada*, el cual hace referencia a fragmentos de textos entre dos comas.

```
DECLARE Subordinada;
(COMMA # COMMA){->MARK(Subordinada)};
```

La conjunción de dos sujetos dentro de una oración debe ser tratado como un único sujeto plural a la hora de verificar la concordancia entre sujeto y verbo. El siguiente script verifica la existencia de una conjunción de sujetos donde el verbo asociado es un verbo singular.

```
24 // SUJETO + AND + SUJETO + VERBO
25 (Sujeto Y Sujeto Token){->MARK(SubjectVerbAgreement)}<-{Token.pos == "VBZ";};
26
27 // SUJETO + AND + SUJETO + PREDICATE(con verbo singular)
28 (Sujeto Y Sujeto Predicate){->MARK(SubjectVerbAgreement)}<-{Predicate.root.pos == "VBZ";};
29
30 //SUJETO y SUJETO VERBO. Con subordinadas.
31 (Sujeto Subordinada Y Sujeto Predicate){->MARK(SubjectVerbAgreement)}<-{Predicate.root.pos == "VBZ";};
32 (Sujeto Y Sujeto Subordinada Predicate){->MARK(SubjectVerbAgreement)}<-{Predicate.root.pos == "VBZ";};
```

Las anotaciones *Token* y *Predicate* indican que el verbo es singular cuando el valor del atributo *POS* es *VBZ* (Verbo singular). En las líneas 31 y 32, se verifican las oraciones que contienen oraciones subordinadas, realizando el mismo control sobre los *Token* y *Predicate* sin tener en cuenta el fragmento correspondiente a la subordinada.