CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Mejora del pipeline de NLP para analisis de historia clínica digital

Autor: Pablo García Encinas

Directora: Ernestina Menasalvas Ruiz

MADRID, JULIO DE 2019

Este trabajo no habría sido posible sin la ayuda de Ernestina y el resto de compañeros del grupo de investigación MIDAS.

Gracias a mi familia por apoyarme siempre. Gracias a mis amigos de la Batmafia, que sin ellos la estancia en la universidad habría sido más larga o muy corta.

Índice general

Índice de figuras

Índice de tablas

Resumen

La digitalización de la historia clínica digital ha permitido que se tenga acceso a mucha información acerca de la historia natural de los pacientes. Sin embargo, poder extraer información de estas historias clínicas requiere sistemas que permitan estructurar la información en forma de texto.

En este sentido en el grupo MIDAS de la UPM se ha desarrollado una herramienta C-liKES que permite estructurar esta información. En este trabajo de fin de grado analizamos C-liKES para añadir funcionalidades y paliar problemas que esta herramienta tiene.

En particular, se ha incluido la funcionalidad de reconocimiento de términos médicos usando la ontología UMLS, que ha servido como base para que una persona sin conocimientos técnicos elevados pudiera definir semánticas encontradas en los textos.

El sistema también ha sido modificado para ser operado por medio de una API, facilitando su gestión y permitiendo automatizar las ejecuciones. Además, siguiendo ese planteamiento, se ha introducido el concepto de usuarios del sistema que pueden darle uso a la vez, para que distintos investigadores tengan la posibilidad de trabajar juntos sin suponer un problema.

Abstract

The digitalisation of the digital clinical records has granted us access to a huge amount of information about the patient's natural history. However, being able to extract information from those clinical records requires systems that allow structuring the information in text form.

In this regard, the MIDAS group from UPM has developed a tool called C-liKES that is able to structure that information. In this final project we analyse C-liKES to add new functionalities and reduce the problems this tool has.

In particular, a new functionality for recognizingmedical terms was added making use of the metathesaurus UMLS, which serves the basis to allow somebody without high rechnical knowledge define semantics to be found in the texts.

The system was also adapted to be controlled through an API, making the administration process simpler and the execution more automatic. Following this approach, users were also added to the system and they can all work simultaneously, this way several researchers are able to work together without any hassle.

1

Introducción y objetivos

1.1 Introducción

Los avances tecnológicos de los últimos años han resultado determinantes para el gran crecimiento de lo que denominamos Big Data. Cada vez generamos y almacenamos más datos, lo que sumado a la conectividad que nos brinda internet supone el caldo de cultivo perfecto para la revolución que nos acontece: sólo en 2017 ya se generaban 2.500 millones de GB al día [1]. Esto acarrea una gran oportunidad pero también un desafío, ya que debemos de ser capaces de seleccionar de un sinfín de datos qué información nos es relevante. Y como es lógico la inmensa mayoría de los datos no son estructurados y forma parte del propio proceso de extracción de datos organizarlos y jerarquizarlos.

Centrándonos en el caso particular del que trata el trabajo, las historias clínicas hospitalarias contienen información muy valiosa que comprende antecedentes familiares del paciente, diagnósticos y tratamientos, así como eventos en el tiempo que permiten inferir la evolución de su estado, sus patologías, etc. Resulta evidente que es de gran relevancia médica ya que es el medio empleado por los profesionales sanitarios para recopilar los datos destacables del caso. Ser capaces de automatizar la extracción y estructuración de estos datos supondría tener una base de conocimiento con mucho potencial en distintos ámbitos de la medicina, como por ejemplo analizar el impacto de los distintos tratamientos en pacientes sin necesidad de un estudio clínico específico.

Como es comprensible, los médicos eligen el lenguaje natural para confeccionar la historia clínica y es aquí donde se presenta el primer reto: el lenguaje natural es ambiguo, subjetivo y a menudo inconsistente. Extraer la información más relevante no es trivial: es necesario conocer tecnicismos, discernir características médicamente relevantes de lo meramente anecdótico, desambiguar términos en función de su contexto, identificar el uso de la negación y relacionar los elementos entre sí.

No existe ninguna solución comercial que extraiga conocimiento de historias clínicas, pero si podemos encontrar herramientas para ello en el ámbito de la investigación. Algunos ejemplos son Savana [2] e IOMED [3], que ofrecen varios servicios de extracción de datos de textos clínicos.

El sistema desarrollado por MIDAS se llama C-liKES (Clinical Knowledge Extraction System) y es una solución integral compuesta por módulos que emplean diversas tecnologías para generar una base de datos estructurada, consistente y relevante a partir de cualquier conjunto de informes médicos. Este trabajo consiste en la ampliación y mejora de estos módulos para conseguir ampliar la información recopilada y mejorar su calidad, reducir el coste computacional total del sistema y facilitar la adaptación de sus componentes con otras aplicaciones.

1.2 Objetivo

El objetivo será mejorar C-liKES para tener más funcionalidades y ser más usable.

Para ello, el primer paso será analizar y comprender el sistema C-liKES para ser capaz de usarlo como usuario, así como la modificación o extensión de sus capacidades como desarrollador.

También será necesario añadir la funcionalidad de búsqueda de términos médicos, que consiste en localizar palabras de los textos médicos que estén recogidas en una extensa base de datos. Concretamente la base de datos UMLS [4], que contiene millones de registros con partes del cuerpo, enfermedades, medicamentos y muchos otros términos médicos.

Se requiere además rediseñar el sistema para simplificar futuras modificaciones, ya que requiere demasiado tiempo para familiarizarse con él y las tareas de alterar y probar el código resultan arduas. Se busca que el sistema sea agnóstico y pueda integrarse en gran variedad de entornos y con distintos lenguajes de programación, dotándole de una versatilidad que amplíe su uso.

2

Preliminares

2.1 Introducción

Un trabajo previo al desarrollo de cualquier modificación del sistema C-liKES es necesariamente estudiar en profundidad su funcionamiento y cada uno de sus componentes.

A continuación se muestra el resultado del estudio del sistema focalizando en el flujo de los datos, seguido de la organización de su arquitectura y sus componentes, y finalmente una valoración de las limitaciones que se han observado en el estudio y que requieren atención a la hora de proponer los cambios a realizar.

2.2 Flujo de datos de C-liKES

Entrada: Documentos médicos en archivos CSV.

Salida: Base de datos relacional orientada a paciente con la información extraída de los textos.

C-liKES es un sistema cuya entrada es un conjunto de documentos clínicos y su salida es una base de datos con información estructurada de cada uno de los pacientes.

Está formado por procesos fundamentales que transforman y añaden datos, los cuales son: *Raw2SourceDocument, SourceDocument2Annotation, Annotation2Document y Document2Patient*. A continuación de describen en detalle estos procesos.

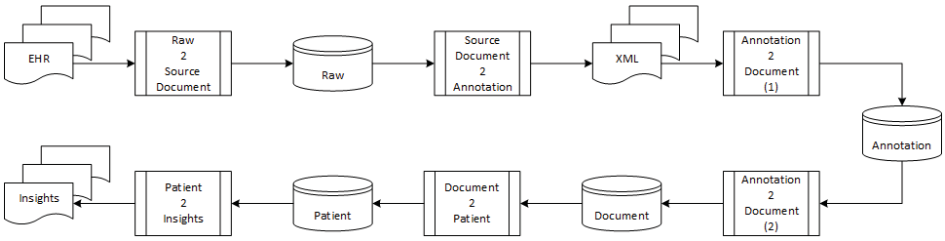

Figura 2.1: Flujo de datos de C-liKES

2.2.1 Raw2SourceDocument

Entrada: Archivos CSV con documentos y sus metadatos.

Salida: Base de datos con los documentos y sus metadatos estructurados.

Proceso: Mediante el uso de scripts SQL se leen los archivos CSV y se normalizan los informes médicos, los cuales pueden presentarse en distintos formatos. De esta manera se recogen todos los documentos en un único modelo (tabla 2.1) llamado *HealthDocument*, que contiene el texto y los metadatos. Como mínimo se debe conocer la fecha de creación, el texto, el idioma y un identificador del paciente.

id	El identificador único de cada documento generado automáticamente.
EHR	El identificador del paciente (Electronic Health Record.)
date	La fecha de creación del documento.
text	El texto del documento a procesar.
birthDate	La fecha de nacimiento del paciente.
gender	El género del paciente.
category	El tipo de documento (Nota o Informe, por ejemplo.)
subcategory	El departamente hospitalario.
hospital	El hospital del que provine el documento.
active	Flag booleano para desactivar documentos sin borrarlos.
language	El idioma del documento (ISO 639-1).

Tabla 2.1: Modelo de HealthDocument de entrada al sistema

Un requisito previo a este proceso es convertir manualmente los archivos que contienen los documentos, típicamente en formato XLS, a archivos CSV [5] para que puedan ser leídos en este proceso.

2.2.2 SourceDocument2Annotation

Entrada: Base de datos con los documentos y sus metadatos.

Salida: Archivos XML que contienen las anotaciones y la información extraída del texto.

Proceso: En este proceso se computan todos los documentos en el pipeline de UIMA [6], en el cual cada anotador añade información al encontrar términos o patrones específicos. Finalmente se recogen los resultados en ficheros XML [7], uno por cada documento. Se pueden distinguir tres subprocesos:

1. Una implementación del componente de UIMA CollectionReader se encarga de consultar la base de datos para obtener los documentos y para cada uno de ellos crea un objeto CAS. Para ello antes limpia el texto de caracteres no imprimibles, etiquetas HTML y demás contenido despreciable.

2. Dichos objetos CAS serán procesados de manera organizada por los anotadores definidos en el pipeline de UIMA (figura 2.2.) Estos anotadores son implementaciones de *analysis engines* que aportan información a porciones del texto, y entre ellos se pueden distinguir los que detectan patrones y los que localizan términos de un diccionario.

3. Finalmente un *CAS consumer* serializa cada objeto CAS en un archivo XML que será el punto de partida en el siguiente paso de la ejecución de C-liKES.

2.2.3 Annotation2Document

Entrada: Archivos XML con anotaciones de los documentos.

Salida: Base de datos relacional con la información extraída orientada a documento.

Proceso: Los archivos XML con las anotaciones conforman una base de datos de documentos no relacional, la cual por medio de BaseX [8] puede ser administrada para consultar los documentos y sus contenidos de manera eficiente.

Un script de python será el encargado de recopilar los documentos XML por medio de consultas a BaseX y relacionar todas sus anotaciones, poblando con ellas una base de datos MySQL orientada a documentos.

2.2.4 Document2Patient

Entrada: Base de datos relacional orientada a documento.

Salida: Base de datos relacional orientada a paciente.

Proceso: Un script de python lee la base de datos que contiene los documentos y toda la información extraída y lo organiza en torno a cada paciente, volcando el resultado final de C-liKES en una base de datos MySQL.

2.3 Arquitectura de C-liKES

El sistema se encuentra organizado (figura 2.2) en contenedores de docker [9] y es orquestado con un archivo de configuración de docker compose [10]. Esto mantiene los módulos de manera independiente y permite emplear comandos para controlar las distintas acciones necesarias para usar el sistema.

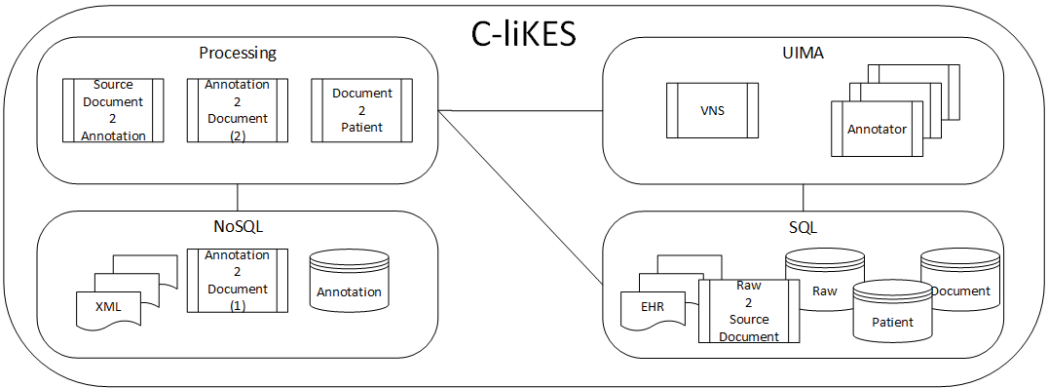

Figura 2.2: Arquitectura del sistema

2.3.1 Contenedor SQL

Servicio MySQL que contiene las bases de datos relacionales para que los componentes de *processing* lean y escriban en ellas:

1. **Raw** alberga la entrada del sistema, compuesta por los documentos normalizados con sus atributos asociados (identificador del paciente, tipo de documento, título, fecha de creación, etc.)

2. **Document** contiene las anotaciones resultantes tras procesar los textos clínicos de forma jerárquica orientada a cada documento: Documento, sección, párrafo y frase (figura 2.3.)

3. **Patient** está compuesta por la información procesada de los documentos orientada a cada paciente.

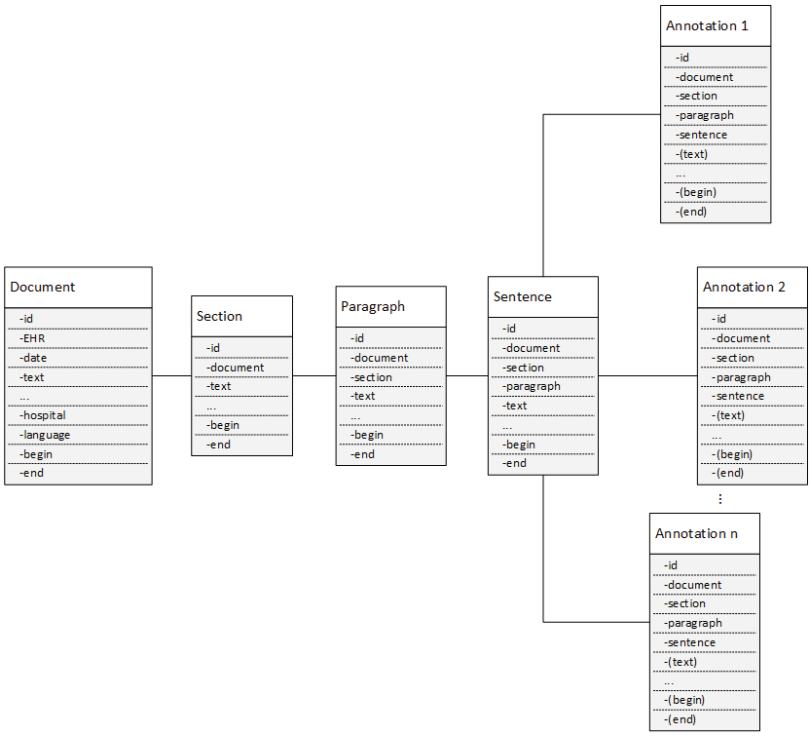

Figura 2.3: Esquema de tablas de la base de datos orientada a documentos

2.3.2 Contenedor NoSQL

En este contenedor se ejecuta BaseX [8], una base de datos no relacional compuesta por los XML resultantes del proceso de anotación de los documentos. Su función es atender a consultas sobre dichos resultados y hacerlo de manera eficiente a pesar del gran volumen de datos con el que trabaja.

2.3.3 Contenedor UIMA

Es aquí donde se ejecutan los anotadores de texto (tabla 2.2) como anotadores remotos del framework UIMA [6] de Apache. Un servicio llamado Vinci resuelve peticiones tanto de C-liKES como de otros equipos (el de un investigador haciendo pruebas, por ejemplo) para que los anotadores realicen trabajos sobre conjuntos de documentos.

Tokenization	Divide los textos en párrafos, frases y tokens.
PartOfSpeech	Realiza análisis morfológico de los tokens.
Section	Agrupa trozos del texto en secciones que tratan un mismo tema.
OncologyTreatment	Detecta tratamientos y sus métricas (Quimioterapia, Radioterapia, etc.)
OncologyDiagnosis	Localiza diagnósticos de cáncer de pulmón con su fecha asociada.
Occupation	Busca la profesión del paciente.
OncologyEcog	Detectan métricas específicas sobre el estado del cáncer y sus tratamientos.
OncologyEgfr	
OncologyStage	
OncologyKarnofsky	
OncologyAlk	
OncologyPerformanceStatus	
HospitalService	Extrae el nombre del servicio hospitaario cuando esté presente.
Comorbidity	Localiza comorbilidades mencionadas en el texto.
FamilyAntecedent	Anota antecedentes familiares.

Tabla 2.2: Lista de anotadores presentes en el pipeline de UIMA

2.3.4 Contenedor Processing

En este contenedor se orquesta la ejecución del sistema mediente el uso de 3 componentes mencionados en el apartado anterior: *SourceDocument2Annotation*, *Annotation2Document* y *Document2Patient*.

Para ello cuenta con un entorno de ejecución en el que dispone de Java y Python así como acceso al resto de recursos del sistema por medio de una red de docker.

2.4 Valoración de C-liKES

Tras haber analizado en profundidad el sistema, su forma de uso y sus componentes, se aprecian algunas limitaciones que deben ser resueltas para conseguir los objetivos planteados:

1. Crear nuevos anotadores y modificar los existentes es complejo y requiere configurar un entorno de desarrollo muy pesado.

2. No hay reconocimiento de entidades nombradas sobre el que construir anotadores de más alto nivel.

3. Es necesaria una persona con conocimiento del sistema para ejecutar y obtener resultados.

4. La integración de los datos de salida del sistema en procesos siguientes no es automática y supone una pérdida de tiempo.

2 Preliminares

3

Enfoque propuesto

3.1 Introducción

El trabajo se ha llevado a cabo en el Centro de Tecnología Biomédica de la UPM en el laboratorio MEDAL.

Tal y como se ha visto en el capítulo anterior C-liKES cuenta con algunas limitaciones que debemos mejorar para conseguir los resultados deseados. El objetivo de este trabajo consiste en analizar e implementar una serie de modificaciones que resulten en un sistema más usable, mantenible, potente y genérico.

Habiendo analizado C-liKES, las mejoras propuestas se detallan a continuación:

- Reconocimiento de términos médicos.
- Semánticas a partir de conceptos.
- C-liKES como servicio.
- Soporte de varios usuarios.
- Librería pública para Java.

3.2 Reconocimiento de términos médicos

Un anotador fundamental para el crecimiento del proyecto que permitiese abarcar otros campos de estudio es el de términos médicos. A continuación se muestra una frase de ejemplo con dichos términos resaltados:

> Cancer diag hace 3 meses, Adenocarcinoma pulmon izqduierdo estadio IIIb T2N3M0, con EGFR Y ALK negativos.

Anotar los términos médicos y poder identificarlos con una referencia única es fundamental para que los resultados puedan ser usados por otros investigadores de distintas instituciones. Además, constituye una buena base sobre la que crear nuevos anotadores de alto nivel abstrayéndose del lenguaje del texto y considerando únicamente los conceptos encontrados en él.

3.2.1 Análisis de requisitos

El primer requisito para poder reconocer términos médicos es utilizar alguna base de datos que se pueda consultar para identificar los propios términos. También es necesario que cada término contenga algo de información:

- Identificador único: Se requiere un identificador del término médico para poder compartir resultados de futuros análisis con otros laboratorios. Es por esto que la base de datos empleada debe de ser accesible fácilmente por otros investigadores.
- Descripción: Es necesario también saber qué constituye el término médico, principalmente para facilitar la comprensión de los resultados de la anotación.

Como se ha visto en el ejemplo anterior, hay términos constituidos por siglas, otros por una sola palabra y otros por varias palabras. La solución a desarrollar debe de ser capaz de encontrar todos ellos e incluso ser redundante, tal y como se describe a continuación:

- Si un término compuesto por varias palabras, como puede ser ”cáncer de pulmón”, contiene otros términos de menor número de palabras (como ”cáncer” y ”pulmón”) deberán localizarse todos.
- Cuando un mismo nombre pueda constituir más de un concepto médico, como por ejemplo *martillo* que puede referirse a un hueso o a una herramienta, deben anotarse todas sus acepciones.

Este trabajo debe realizarlo un nuevo anotador de UIMA que se incluirá en el pipeline de C-liKES y su complejidad debe de ser lineal, ya que de ser cuadrática el tiempo de ejecución del sistema se dispararía con el gran volumen de datos que maneja.

3.2.2 Implementación y desarrollo

El primer paso para desarrollar el anotador propuesto es encontrar una base de datos que cumpla las características descritas. En esta búsqueda las opciones más relevantes fueron SNOMED [11] y UMLS [4], ambas son extensas y se usan día a día en muchos otros proyectos.

Una diferencia fundamental entre las dos es que mientras SNOMED es una terminología médica, UMLS se define como un *metathesaurus*. Esto significa que UMLS es un conjunto de muchas terminologías médicas en diferentes idiomas, y de hecho SNOMED es una de ellas. Por lo tanto, UMLS se considera más completo.

Además, UMLS no solo contiene los conceptos médicos de las terminologías recogidas en él, también cuenta con una red de semánticas que relaciona los conceptos y define una jerarquía entre ellos(figura 3.1). Gracias a esto podemos obtener algo más de información sobre los términos, sus tipos semánticos nos sirven como descripción para completar las anotaciones.

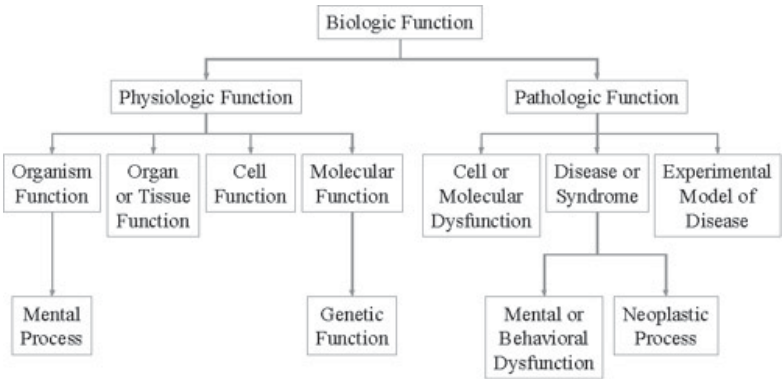

Figura 3.1: Ejemplo de tipos semánticos de UMLS

UMLS está desarrollado por el Instituto de la Salud estadounidense y para adquirirlo es necesario solicitar una licencia de uso en su página web (figura 3.2). Una vez revisan y aprueban la solicitud, podemos descargar una serie de archivos y scripts que, tras ser configurados, generan una base de datos con todos los términos médicos.

Para incluir UMLS en C-liKES es necesario añadir un nuevo contenedor en el docker-compose. Su único cometido será tener un servicio de MySQL al que poder hacer consultas

3 Enfoque propuesto

Figura 3.2: Solicitud de licencia de uso de UMLS

para conocer qué palabras constituyen términos médicos. Esas consultas se harán desde una cuenta sin permisos para hacer modificaciones, es decir, solo con privilegios para ejecutar mandatos SELECT.

El siguiente paso consiste en desarrollar un anotador que cumpla los requisitos descritos para incluirlo en el pipeline de UIMA de C-liKES. Para ello el entorno de desarrollo más adecuado es el IDE Eclipse [12], ya que cuenta con plugins específicos de desarrollo de UIMA, y necesariamente el lenguaje de programación será Java. Una dependencia que el anotador debe satisfacer en su iniciación es la conexión a la base de datos: esta se empleará constantemente para cotejar si los conceptos constituyen un término en UMLS o no.

Para hacer la búsqueda de un concepto en la base de datos se ejecuta un *prepared statement* con la cadena de texto a cotejar como parámetro. A continuación se muestra la query empleada:

```
SELECT MRCONSO.CUI, STY
FROM umls.MRSTY
LEFT OUTER JOIN umls.MRCONSO ON MRSTY.CUI=MRCONSO.CUI
WHERE STR = ? GROUP BY MRCONSO.CUI, STY
```

Cada tupla devuelta, en caso de haberla, constituye un término UMLS válido y contiene dos campos tal y como solicitamos en la cláusula select:

1. **CUI**: Identificador único de UMLS.
2. **STY**: Tipo semántico de UMLS.

Lo único que falta para completar el anotador es definir el algoritmo encargado de procesar los textos. Haciendo uso del análisis morfológico que realiza el anotador *PartOfSpeech* podemos reducir las consultas a la base de datos, descartando verbos, determinantes, conjunciones y adverbios. El resto de tokens, constituidos por sustantivos, adjetivos y numerales serán buscados individualmente y en caso de constituir un término serán anotados con su información asociada. Además de ser anotados individualmente, serán guardados para consultar si la concatenación con el token siguiente constituye también un término.

El algoritmo tiene una complejidad $\mathcal{O}(n)$ ya que solo recorre una vez los tokens del texto, por lo que su inclusión en el pipeline del sistema no aumenta drásticamente el tiempo de ejecución. Los resultados son satisfactorios como muestra el resultado de la anotación sobre la oración que se exponía como ejemplo al comienzo de esta sección:

STR	CUI	STY
Cáncer	C0006826	Neoplastic Process
Adenocarcinoma	C0001418	Neoplastic Process
Pulmón	C0024109	Body Part, Organ, or Organ Component
Pulmón izquierdo	C0225730	Body Part, Organ, or Organ Component
EGFR	C0034802	Receptor
ALK	C0252409	Enzyme

Tabla 3.1: Salida del anotador UMLS sobre la oración de ejemplo

3.3 Semánticas a partir de conceptos

Contar con los términos UMLS encontrados en los textos permite efectuar distintos análisis que anteriormente no eran viables, como por ejemplo detectar la frase en que se habla del diagnóstico de un paciente.

Los médicos pueden referirse al diagnóstico de muchas formas distintas, por lo que no resulta sencillo usar patrones para localizarlo. Algunos ejemplos son: *cáncer de pulmón, cáncer pulmonar, cáncer alveolar, carcinoma pulmonar, adenocarcinoma de pulmón*, etc. Haciendo uso de los tipos semánticos de UMLS podemos reconocer el diagnóstico siempre que una frase contenga un ”proceso neoplásico” y una localización relacionada con ”pumón”.

$$\text{Frase con diagnóstico} \begin{cases} \text{- STY: Proceso Neoplásico} \\ \text{- CUI: Pulmón o Pulmonar} \end{cases}$$

Una ventaja que confiere trabajar sobre los términos médicos ya anotados es que el idioma del texto original es irrelevante.

3.3.1 Análisis de requisitos

Se requiere crear un anotador que evite el trabajo manual de relacionar términos UMLS en las oraciones en búsqueda de semánticas superiores como es el diagnóstico. Este anotador debe de usar el framework de UIMA para poder integrarse en el pipeline de C-liKES.

Un requisito diferencial respecto a otros anotadores es que la configuración que recibe debe poder modificarse de la forma más sencilla posible, sin necesidad de compilar el código y siendo lo suficientemente fácil como para que un investigador dedicado al análisis de los resultados, sin conocimiento sobre el funcionamiento de UIMA, pueda hacerlo.

Dicha configuración además deberá ser lo suficientemente versátil como para definir semánticas complejas, para lo que debe satisfacer estos requisitos:

- Cada semántica estará definida por un conjunto de reglas que deben satisfacerse en la frase, al menos una pero sin límite superior.
- Cada regla define qué términos UMLS la satisfacen, y puede hacerlo mediante un conjunto de CUIs o mediante un conjunto de STYs (ambos conjuntos pueden contener un único elemento.)
- Además, dichas reglas pueden negarse para forzar la no aparición de los términos especificados.

Este último requisito resulta fundamental para evitar falsos positivos en ciertos casos. Por ejemplo, en un contexto de antecedentes familiares se puede estar mencionando el cáncer sin tratarse del diagnóstico, para solucionarlo podemos definir una regla de no aparición de palabras que definen una relación familiar como ”padre”, ’madre” o ”abuelo”.

3.3.2 Implementación y desarrollo

En un anotador de UIMA convencional es habitual definir cualquier parámetro de configuración en el propio código o en alǵún archivo de recurso del proyecto. En ambos casos, la modificación de dichos parámetros requiere de un entorno de desarrollo pesado y conocimiento para compilar un JAR que incluya los nuevos cambios, por lo que para este anotador es necesario otro planteamiento.

Una alternativa viable es que el anotador lea sus configuraciones en el momento de su inicialización de un archivo externo al proyecto, en una ruta predefinida de un volumen de archivos del contenedor *uima* de C-liKES. Esto no solo reduce el trabajo al realizar modificaciones a la configuración, también tiene sentido ya que distintas instancias del sistema

podrían tener distintas configuraciones en función del caso de uso que requiera su proyecto de investigación.

Para serializar las configuraciones en el archivo de texto hay gran variedad de formatos, siendo los más relevantes XML, JSON y YAML:

- **XML**: Es una buena opción ya que se usa en varios componentes del sistema y la versión de UIMA que se emplea requiere ciertas configuraciones con este formato. Sin embargo, aunque en el pasado estaba muy extendido se ha reducido su uso porque resulta demasiado verboso.
- **JSON**: Este formato está basado en la sintaxis de Javascript por lo que su uso en el entorno web es casi total. Resulta muy ligero y sencillo de leer por un humano.
- **YAML**: Cuenta con la facilidad de uso de JSON pero es más potente ya que cuenta con más funcionalidades como datos tipados, estructuras no jerárquicas y el uso de la indentación para estructurar los datos.

XML	JSON	YAML

<Servers>
<Server>
<name>Server1</name>
<owner>John</owner>
<created>123456</created>
<status>active</status>
</Server>
</Servers>
	{
	Servers: [
	{
	name: Server1,
	owner: John,
	created: 123456,
	status: active
	}
	]
	}
		Servers:
		- name: Server1
		owner: John
		created: 123456
		status: active

Figura 3.3: Formatos de serialización XML, JSON y YAML

Aunque los 3 formatos son buenas opciones, en este caso el anotador utilizará JSON ya que teniendo en cuenta la dirección que debe tomar este proyecto, la integración con servicios web será más sencilla si se emplea este formato.

La configuración del anotador viene definida por una lista de semánticas, identificadas con un nombre, que pretenden encontrarse en los documentos. Cada una de estas semánticas está constituida por la lista de reglas que deben darse en una misma frase para considerar encontrada dicha semántica. Cada regla tiene un parámetro ”target” que identifica cuál es el objetivo de la regla, siendo sus posibles valores *cui* cuando se trata del identificador único de UMLS, o *sty* cuando el objetivo de la regla es su tipo semántico. Las reglas también deben contar con el parámetro ”rule” el cual contiene la lista de valores que permiten satisfacer la misma, separadas por una barra vertical en caso de ser más de uno. Por último, el parámetro

opcional ”inverse” puede tomar el valor booleano *true* cuando se pretende que la frase no satisfaga la regla para encontrar la semántica (figura 3.4).

```
{
  "dx": [
    {
      "target": "sty",
      "rule": "B2.2.1.2.1.2"
    },
    {
      "target": "cui",
      "rule": "C0024109|C2709248"
    },
    {
      "target": "sty",
      "rule": "A2.9.3",
      "inverse": true
    }
  ]
}
```

Figura 3.4: Definición de reglas de diagnóstico en archivo JSON

La primera regla define que debe encontrarse un proceso neoplásico, la segunda o bien el término pulmón o pulmonar, y la tercer al estar invertida define que la frase no contenga un término de relación familiar.

El anotador, desarrollado en Java usando el framework UIMA, tendrá una fase de inicialización en la cual leer el archivo de texto e importar de él la configuración de las semánticas y sus reglas. A la hora de procesar cada documento, iterará sobre cada oración comprobando si el conjunto de términos médicos contenidos en ella satisfacen las reglas impuestas por cada semántica, y de hacerlo la anotará con el nombre que identifica a la semántica así como los términos UMLS involucrados.

Gracias a la incorporación de este anotador, llamado *SentenceSemantic*, se han agilizado las labores de análisis posteriores a la ejecución del sistema incorporando varias semánticas como por ejemplo *comorbilidad*, *antecedente familiar* y el *diagnóstico* descrito en los ejemplos.

3.4 C-liKES como servicio

La interacción con el sistema siempre ha sido por medio de comandos en el terminal y modificando distintos archivos con gran variedad de formatos. Es por esto que para operarlo se requiere de un técnico con conocimientos de todos los componentes y es un proceso aparatoso en el que, al hacerse manualmente, es fácil cometer errores que conllevan pérdidas de tiempo.

Sería deseable contar con otros mecanismos para manejar el sistema que no requieran tanta preparación y faciliten la automatización.

3.4.1 Análisis de requisitos

Se requiere definir una interfaz con la que poder controlar el ciclo de vida del sistema sin tener que tener acceso de administración a la máquina huésped:

- **Carga de documentos**: Los reportes médicos que constituyen la entrada del sistema ya no se cargarán mediante consultas SQL leyendo archivos CSV, si no por peticiones al servicio.
- **Definición de reglas semánticas**: La configuración del anotador SentenceSemantic ya no se importará de un archivo de texto, se podrá definir en cualquier momento previo a la ejecución por medio del servicio.
- **Ejecución del pipeline**: Anteriormente consistía en una serie de comandos en la terminal, ahora debe poder hacerse mediante una única petición al servicio.
- **Obtención de resultados**: El servicio proveerá la opción de consultar los resultados de la ejecución de forma cómoda para facilitar la integración en módulos para su análisis. Esto supone una mejora respecto al método anterior, que se basaba en la descarga de un *dump* de la base de datos.

El servicio que implemente dicha interfaz deberá atender peticiones con un protocolo de comunicación muy extendido para simplificar su integración en otras aplicaciones, así como aportar una documentación detallada del uso de cada petición.

3.4.2 Implementación y desarrollo

La interfaz se definirá como una API RESTful, lo cual supone entre otras cosas que el protocolo será HTTP empleando JSON para serializar los datos enviados y recibidos. Esta

elección se debe a que es con gran diferencia la comunicación más usada entre aplicaciones, por lo que hay gran cantidad de librerías que simplifiquen la integración del servicio en muchos lenguajes de programación.

La entrada del sistema, constituida por los reportes médicos, está modelada como *Raw Document* y cuanta con varios métodos (figura 3.5) con los que consultar y modificarla:

- **Verbo PUT**: Con este verbo HTTP se pueden añadir nuevos documentos al sistema o modificar los ya existentes. Cada Raw Document contiene un identificador único aportado por el usuario, con lo que una petición PUT sobre un identificador ya conocido en lugar de crear modificará el recurso.

  Hay un endpoint con el que crear o modificar documentos de uno en uno, el cual resulta más cómodo si la cantidad de datos es pequeña, y otro con el que hacerlo en lote, lo cual agiliza el proceso si la colección de documentos es ingente.

- **Verbo DELETE**: Como su propio nombre indica, los métodos con este verbo eliminan los documentos previamente creados. Cuenta con dos endpoints, permitiendo eliminar documentos selectivamente o todos con una única petición.

- **Verbo GET**: Este verbo permite consultar los documentos contenidos en el sistema, de nuevo permitiendo obtenerlos deforma única o toda la colección a la vez.

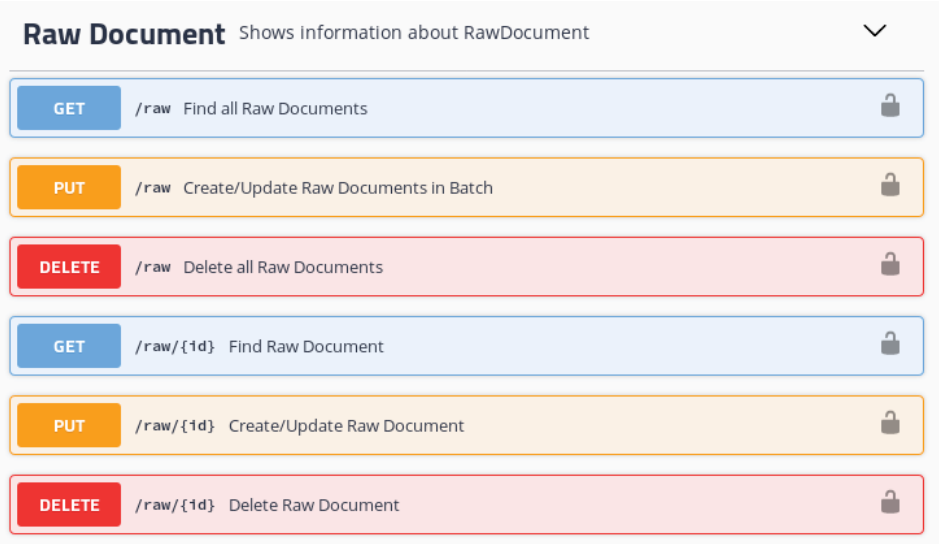

Figura 3.5: Endpoints de Raw Document

La configuración del sistema, actualmente tan solo las semánticas del anotador Sentence-Semantic, puede ser alterada (figura 3.6) con los siguientes métodos:

- **Verbo PUT**: Empleado para definir las semánticas, puede crear nuevas o reemplazar una existente si tienen el mismo nombre.

- **Verbo DELETE**: Permite eliminar alguna semántica definida anteriormente si se especifica su nombre, o borrarlas todas en caso contrario.

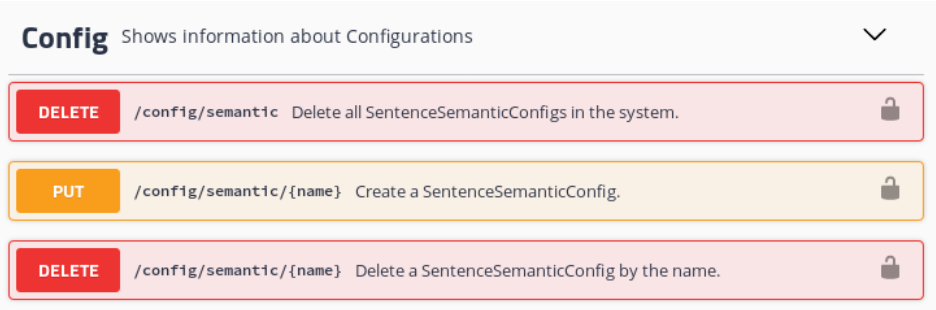

Figura 3.6: Endpoints de configuración de las semánticas

El ciclo de vida del sistema se modela en *Jobs*, que como el resto de recursos pueden crearse y eliminarse (figura 3.7) mediante peticiones al servicio:

- **Verbo POST**: Crea un nuevo Job, lo cual supone empezar una nueva ejecución del sistema. Devuelve un identificador del job creado por el sistema para poder realizar sobre él peticiones en el futuro

- **Verbo GET**: Devuelve información de los jobs existentes, indicando si la ejecución a finalizado o el paso en el que se encuentra en caso contrario. Especificando un identificador puede obtenerse solo la información referente al job indicado.

- **Verbo DELETE**: Borra el job en caso de especificar su identificador o todos los jobs en caso contrario. La eliminación de un Job supone cancelar la ejecución en curso si procede además de eliminar sus resultados asociados si los hubiera.

Figura 3.7: Endpoints del recurso Job

Por último, los resultados pueden ser de dos tipos: términos médicos (UMLS) o semánticas (del anotador SentenceSemantic), cada uno cuenta con un método GET (figura 3.8) el cual recibe como parámetro el identificador del Job al que pertenecen los resultados.

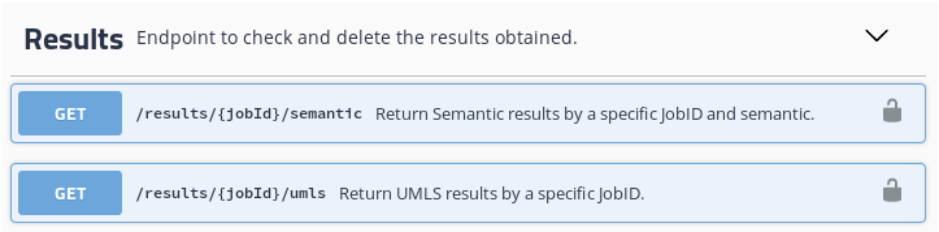

Figura 3.8: Endpoints del recurso Job

Una vez definida la API, comienza la labor de implementar el servicio que atienda las peticiones definidas. Para ello se emplea Spring Boot [13], un framework muy usado para la creación de servicios web en Java, dando lugar a una aplicación que se desplegará en un nuevo contenedor del sistema llamado *http*. Esta se conectará a la base de datos del contenedor *sql*, donde se persistirán los datos de los raw documents, las configuraciones de las semánticas, los jobs y sus resultados.

Además de la inclusión del nuevo servicio web, es necesario hacer modificaciones en varios módulos del sistema para que funcione como está definido:

1. La creación de un script en bash que coordine la ejecución de los componentes: Su función es atender peticiones de la aplicación web para arrancar o detener la ejecución

del sistema, controlando posibles errores y actualizando en la base de datos el estado de cada job. La comunicación se realiza por medio de sockets en puertos no expuestos fuera de los contenedores.

2. Lectura de configuración de semánticas de la base de datos: En lugar de guardar las reglas de las semánticas en un archivo de texto, se recogen en la base de datos como el resto de entidades del sistema y por tanto el anotador *SentenceSemantic* debe conectarse a la misma para importar su configuración.

3. El componente que crea los resultados, *Annotation2Document*, debe ahora asociar cada uno de ellos al job que los ha generado.

Para finalizar, en un endpoint del servicio web se despliega Swagger [14], una utilidad que muestra en el propio navegador todos los métodos que ofrece el servicio con sus parámetros, ejemplos de uso y una herramienta para probarlos en vivo. Esto sirve como documentación y facilita la incorporación de los servicios en otro software.

3.5 Soporte de varios usuarios

C-liKES es un sistema planteado para ser instalado en un laboratorio donde probablemente haya más de un investigador que necesite usarlo, y al estar ya planteado como un servicio resulta lógico orientar su uso a distintos usuarios.

Además, una ventaja añadida es que se puede emplear un usuario para cada caso de uso. Es decir, en lugar de asociar un usuario a cada investigador, se podría designar un usuario a cada caso de estudio que se quiera realizar, ya que se mantendrían documentos y reglas semánticas diferentes para cada uno.

3.5.1 Análisis de requisitos

El requisito fundamental para llevar a cabo esta mejora es garantizar la integridad de los datos entre usuarios. Cada usuario debe tener independencia dentro del sistema en todos los recursos: sus *documentos*, sus *configuraciones semánticas*, sus *jobs* y sus *resultados*.

Será necesario un sistema de autenticación para diferenciar a los usuarios y evitar suplantamientos de identidad, especialmente al manejarse en muchos casos datos sensibles o protegidos por la LOPD. Cada usuario contará con un nombre de usuario y una contraseña secreta para hacer cualquier interacción con el sistema.

Para gestionar los usuarios se implementarán nuevos métodos de la API que únicamente el usuario con el rol de administrador podrá usar. Se requiere poder realizar las siguientes operaciones:

- Consultar los usuarios del sistema.
- Crear un nuevo usuario.
- Eliminar un usuario existente.

3.5.2 Implementación y desarrollo

Para que el sistema de servicio a distintos usuarios y se diferencie absolutamente el contenido de cada uno de ellos, es necesario hacer cambios en la base de datos:

1. Crear una tabla en la base de datos donde almacenar los usuarios con su información asociada: *email* y *password* para la autenticación, *role* para diferenciar al administrador del resto de usuarios, y otros puramente informativos como *name*, *phoneNumber* e *institution*.
2. Relacionar los recursos con la nueva tabla de usuarios para que todos ellos estén asociados únicamente a un usuario.
3. Adaptar todos los componentes del sistema para usar correctamente la nueva estructura de tablas, especialmente la aplicación del servicio web, *SourceDocument2Annotation* y *Annotation2Document*

Todos los métodos de la API deben identificar al usuario que realice cada petición, para ello hay distintos métodos en el entorno web que podemos implementar. A continuación se detallan los tres más relevantes:

- **HTTP Basic Authentication**: En este método cada petición incluye el usuario y contraseña en la cabecera HTTP. No requiere hacer peticiones adicionales ni guardar parámetros de sesión ni cookies.
- **API Keys**: En este caso en vez de incluirse el usuario y la contraseña debe incluirse una cadena de texto única generada para cada usuario llamada *key*. Requiere de métodos para controlar el ciclo de vida de las keys como la creación y el descarte de las mismas. Su principal ventaja sobre *HTTP Basic Authentication* es que aunque alguien pusiera leer las cabeceras no podría conocer el usuario ni la contraseña.

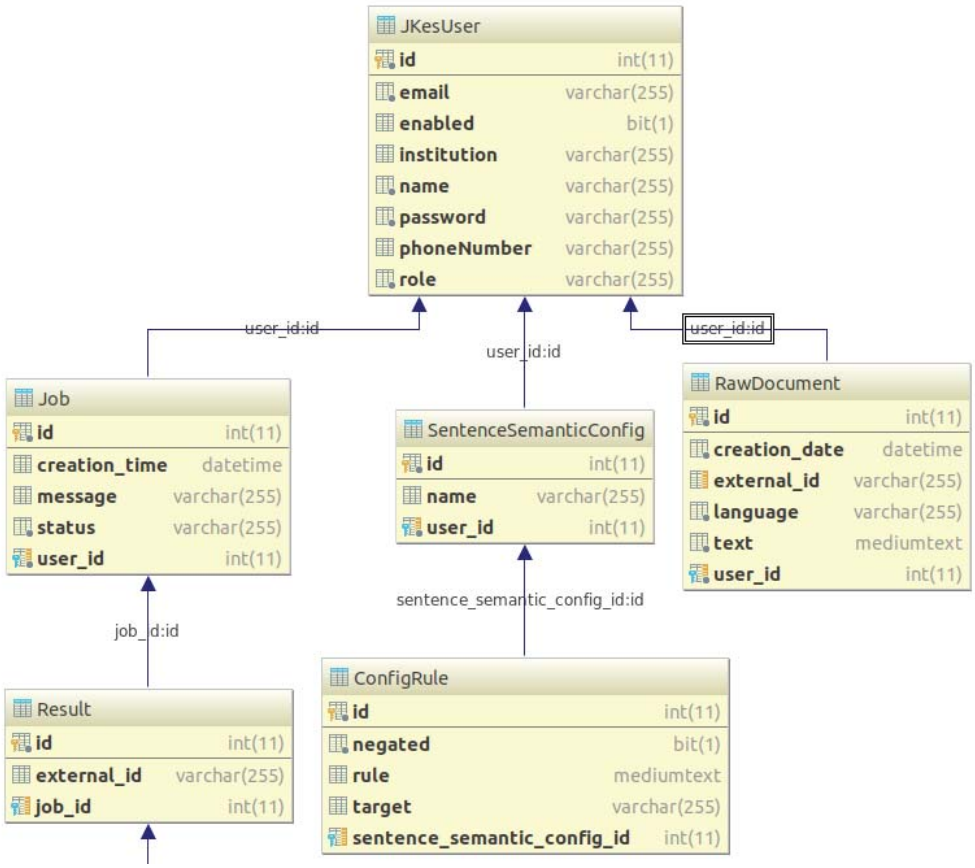

Figura 3.9: Nueva estructura de la base de datos orientada a usuarios

- **OAuth**: Es ampliamente usado en APIs de grandes proveedores de servicios y sin duda es una buena opción cuando la seguridad de miles de clientes es fundamental, sin embargo su implementación es compleja y supone mucho trabajo para el cliente.

De los métodos propuestos se considera que el más adecuado es *HTTP Basic Authentication*, principalmente por dos motivos: Primeramente porque el sistema está planteado para ser usado internamente en laboratorios de investigación, por lo que la seguridad no es la prioridad máxima. Y segundo, porque uno de los objetivos es que su facilidad de uso favorezca su adaptación en distintos proyectos y aplicaciones.

Por lo tanto, todas las peticiones ya existentes de la API ahora requerirán la autenticación enviando los credenciales en las cabeceras HTTP. Este método, aparte de restringir el acceso al sistema únicamente a los usuarios autorizados, también permite diferenciar los distintos clientes cuyos documentos, configuraciones y resultados son diferentes.

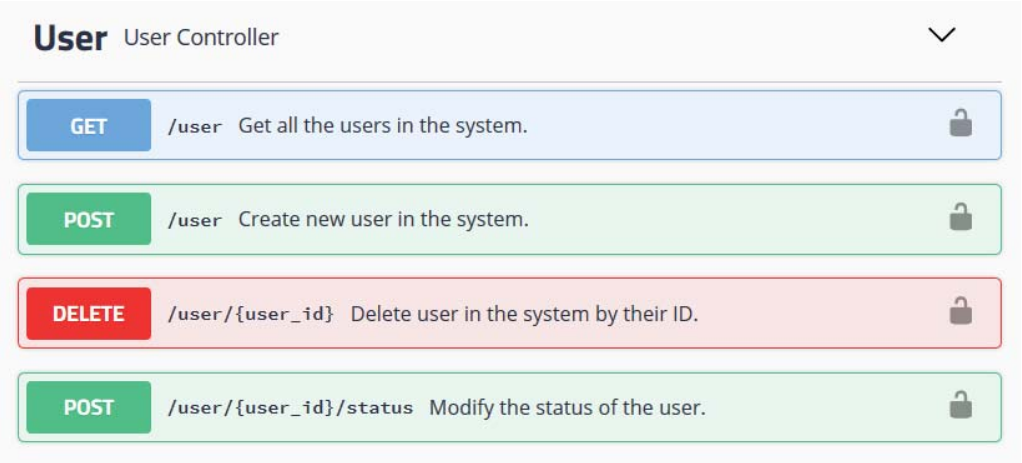

Figura 3.10: Métodos de la API de gestión de usuarios

Por último es necesario que el sistema incluya métodos para gestionar los usuarios como describen los requisitos. Para ello, al crear la instancia del sistema se habilita automáticamente un usuario con el rol de administrador y cuya contraseña vendrá dada por un archivo de configuración del sistema, por defecto *admin*. Este usuario es el único con permiso para realizar las peticiones de gestión de usuarios, las cuales vienen recogidas en la figura 3.10

3.6 Librería pública para Java

Aunque el sistema ya es usable y no se requieren módulos adicionales, se plantea desarrollar una librería para simplificar al máximo la integración a aplicaciones desarrolladas en Java o que ejecuten en la *Java Virtual Machine*.

3.6.1 Análisis de requisitos

La librería debe permitir realizar las operaciones de uso del sistema sin incluir las de gestión. Se considera que la gestión de usuarios, por ejemplo, no requiere ser automatizada ni incluida a una aplicación. Sin embargo si deben facilitarse la subida de documentos al sistema, la definición de configuraciones semánticas, la ejecución del sistema y por supuesto la obtención de los resultados.

Además algo fundamental será facilitar la inclusión de la librería a cualquier proyecto sin tener que copiar código manualmente ni descargar un JAR. Debe ser accesible por sistemas de automatización de construcción de aplicaciones como *Maven* [15] o *Gradle* [16].

3.6.2 Implementación y desarrollo

Para desarrollar la interacción con la API se emplea una librería llamada Retrofit [17], la cual implementa automáticamente la gestión de un cliente HTTP y demás cuestiones para la comunicación aportándole una sencilla configuración y la interfaz de la API (figura 3.11).

Además de efectuar las peticiones al servicio web, la librería también simplificará la gestión de los datos. Concretamente, en la obtención de los resultados el usuario final no tendrá que preocuparse de la paginación, podrá abstraerse empleando una implementación de *Iterable* al que solicitar más elementos y será el cliente el que gestione el índice de las páginas controlando la condición de parada.

```
@GET("raw")
Call<List<RawDocument>> getRawDocuments();

@GET("raw/{id}")
Call<RawDocument> getRawDocument(@Path("id") Integer documentId);

@PUT("raw/{id}")
Call<RawDocument> addRawDocument(@Path("id") String externalId, @Body RawDocument rawDocument);

@PUT("raw")
Call<ResponseBody> addRawDocuments(@Body List<RawDocument> rawDocuments);

@DELETE("raw/{id}")
Call<ResponseBody> deleteRawDocument(@Path("id") String externalId);

@DELETE("raw")
Call<ResponseBody> deleteRawDocuments();
```

Figura 3.11: Porción de la interfaz representante de la API

Para facilitar el uso de la librería en otros proyectos, se mantiene disponible en un repositorio de maven gracias al software Artifactory [18]. Mediante unos scripts configurados en el proyecto, cada nueva versión que se genere será desplegada automáticamente. De esta forma con tan solo incluir dos bloques de código (figura 3.12) en el archivo *pom.xml* de cualquier proyecto maven, la librería podrá usarse y se mantendrá actualizada sin ningún esfuerzo.

```
...

<repositories>
    <repository>
        <id>medal</id>
        <url>http://138.4.130.6:11501/artifactory/libs-release</url>
    </repository>
</repositories>

...

<dependencies>
    <dependency>
        <groupId>es.upm.ctb.midas</groupId>
        <artifactId>jkes-java-client</artifactId>
        <version>0.3.6</version>
    </dependency>
</dependencies>

...
```

Figura 3.12: Incorporación de la librería a un proyecto maven

4

Conclusiones y líneas futuras

4.1 Conclusiones

En este trabajo se ha desarrollado:

1. *Crear nuevos anotadores y modificar los existentes es complejo y requiere configurar un entorno de desarrollo muy pesado.*

   Gracias a las semánticas definidas por medio de la API ya no es necesario programar ni compilar ningún anotador de UIMA. Solo hay que definir ciertas reglas, las cuales pueden ser modificadas de forma sencilla en cualquier momento.

2. *No hay reconocimiento de entidades nombradas sobre el que construir anotadores de más alto nivel.*

   Gracias al anotador de UMLS, los términos médicos son reconocidos e identificados con un CUI único, y sobre ellos se pueden definir anotadores más complejos como SentenceSemantic.

3. *Es necesaria una persona con conocimiento del sistema para ejecutar y obtener resultados.*

Ya no se requiere de un técnico para gestionar el sistema ya que la API permite introducir documentos, ejecutar el procesado del texto y obtener los resultados directamente. Tan solo se requiere conocer dicha API para incorporar el sistema a cualquier proyecto.

4. *La integración de los datos de salida del sistema en procesos siguientes no es automática y supone una pérdida de tiempo.*

   Al aportar una abstracción por medio de la API, se puede usar cualquier lenguaje para programar una aplicación que realice las peticiones al servicio web y obtenga de forma sencilla y rápida los resultados listos para ser procesados y analizados.

Consiguientemente todas las metas plantadas al comienzo se han cumplido con éxito.

4.2 Líneas futuras

A pesar de haber alcanzado las metas con éxito, el desarrollo de este trabajo deja abiertas la siguientes posibilidades de mejora del sistema:

- **Soporte a otros idiomas**: Actualmente el sistema solo funciona con textos en español y, aunque es un idioma bastante extendido, añadir más idiomas aumentaría el uso potencial considerablemente.

  Incluyendo modelos del análisis morfológico para cada nuevo lenguaje así como ampliando los términos de UMLS sería posible generar nuevas versiones de C-liKES que acepten otras lenguas. Otra opción viable es delegar la inclusión de dichos modelos y términos, lo cual acarrea más trabajo para el usuario final pero también aporta más versatilidad.

- **Librerías para otros lenguajes**: En este trabajo se ha desarrollado una librería para Java y, aunque es posible la comunicación directa con la API, el uso de una librería ya preparada facilita la incorporación del sistema en nuevos proyectos.

  Desarrollando librerías análogas para otros lenguajes populares, como por ejemplo Python, se podría extender el uso del sistema y facilitar aun más la integración a proyectos que no utilicen java.

- Interfaces visuales: El sistema requiere de un desarrollador que lo opere, pero en el futuro podrían plantearse una serie de modificaciones que permitan a personas sin conocimientos de programación usar el sistema.

  Sin duda algunas de esas modificaciones deberán incluir una interfaz gráfica, como por ejemplo una web donde introducir textos y definir semánticas. Este paso acercaría más

al personal sanitario el uso directo del sistema sin depender de gente con conocimientos más técnicos.

4 Conclusiones y líneas futuras

Bibliografía

[1] E. Press. (2018). Cada día se generan 2.500 millones de GB de datos: IBM crea una plataforma para que las empresas los aprovechen, dirección: https://www.europapress.es/portaltic/internet/noticia-cada-dia-generan-2500-millones-gb-datos-ibm-crea-plataforma-empresas-aprovechen-20170323162319.html.

[2] Savana. (2019). Transformando el texto de tus historias clínicas en Big Data, dirección: https://savanamed.com.

[3] IOMED. (2019). Acelerando la Investigación Clínica, dirección: https://iomed.es/.

[4] U. D. of Health & Human Services. (2019). Unified Medical Language System, dirección: https://www.nlm.nih.gov/research/umls/.

[5] C. .-.-. C. S. Values. (2019). Datahub, dirección: https://datahub.io/docs/data-packages/csv.

[6] A. S. Foundation. (2019). Apache UIMA, dirección: https://uima.apache.org/.

[7] E. M. L. (XML). (2019). W3C, dirección: https://www.w3.org/XML/.

[8] BaseX. (2019). Enterprise Container Platform - Docker, dirección: https://www.docker.com/.

[9] Docker. (2019). The XML Framework: Lightweight and High-Performance Data Processing, dirección: http://basex.org/.

[10] ——, (2019). Docker Compose - Docker docs, dirección: https://docs.docker.com/compose/.

[11] S. International. (2019). SNOMED, dirección: http://www.snomed.org/.

[12] T. E. Foundation. (2019). Eclipse - The Platform for Open Innovation and Collaboration, dirección: https://www.eclipse.org/.

[13] P. Software. (2019). Spring Boot, dirección: https://spring.io/projects/spring-boot.

[14] SmartBear. (2019). Swagger tools, dirección: https://swagger.io/.

[15] Apache. (2019). Maven - Welcome to Apache Maven, dirección: https://maven.apache.org/.

[16] G. Inc. (2019). Gradle Build Tool, dirección: https://gradle.org/.

[17] Square. (2019). Retrofit, dirección: https://square.github.io/retrofit/.

[18] JFrog, «Universal Artifact Repository Manager», 2019. dirección: https://jfrog.com/artifactory/.

Este documento esta firmado por

Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
Fecha/Hora	Sun Jun 30 17:10:19 CEST 2019
Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
Numero de Serie	630
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)