ULPGC Universidad de Las Palmas de Gran Canaria | Escuela de Ingeniería Informática

Inteligencia artificial aplicada a la ley de protección de datos

Miguel Ángel Medina Ramírez

Grado de Ingeniería informática

Trabajo Final de Grado

Las Palmas de Gran Canaria, junio 2020

Tutor: Cayetano Nicolás Guerra Artal

Agradecimientos

A mi familia, por los ánimos y esfuerzos dados en estos cuatro años, y por no preguntar demasiado por el estado de este TFG.

A Cayetano, mi tutor, por haber despertado en mí el interés por la inteligencia artificial y porque sin su ayuda este trabajo no sería posible.

A Acoran y a Mario, por los consejos dados para que este trabajo siguiera adelante.

A mis amigos y compañeros de carrera.

Resumen

Dada la nueva ley de protección de datos, la publicación y el uso de datos referentes a personas en documentos tanto oficiales como públicos pueden violar sus derechos de privacidad. En este trabajo se ha planteado el prototipo de una herramienta que permita eliminar o reducir al mínimo el riesgo de identificación de los datos personales en diferentes documentos con distintos formatos, centrándose en nombres, apellidos y carnés de identidad. Se han utilizados técnicas de *Deep Learning* e Ingeniería del conocimiento aplicadas al procesamiento de lenguaje natural con el fin de lograr un sistema inteligente híbrido que localice estos datos en textos y proceda a su tratamiento. El conjunto de sus operaciones se encuentra agrupados en un servicio Web con estructura de API REST.

Palabras claves: ley de protección de datos, datos personales, Deep Learning, Ingeniería del conocimiento, sistema inteligente híbrido y servicio Web.

Abstract

Given the new data protection law, the publication and use of data concerning individuals in both official and public documents may violate their privacy rights. In this work we have proposed a prototype of a tool that will make it possible to eliminate or reduce to a minimum the risk of identifying personal data in different documents with different formats, focusing on names, surnames and identity cards. Deep Learning and Knowledge Engineering techniques applied to natural language processing have been used in order to achieve an hybrid smart system that locates these data in texts and processes them. All its operations are grouped in a Web service with an API REST structure.

Key*Words*: data protection law, personal data, Deep Learning, knowledge engineering, hybrid smart system and web service.

Índice de Figuras

Figura 2.1: Cálculo de la función objetivo para tokenizar textos. 5
Figura 2.2: Representación del espacio vectorial de palabras. 6
Figura 2.3:Extracción de pares de palabras. 7
Figura 2.4: Red skip-Gram. 8
Figura 2.5: Etiquetado basado en el contexto. 9
Figura 2.6: Partes de una neurona. 11
Figura 2.7: Diagramas de modelos neuronales. 12
Figura 2.8: Típica arquitectura de una red convolucional. 13
Figura 2.9: Convolución. 13
Figura 2.10: Diagrama de una red neuronal recurrente. 14
Figura 2.11: Diagrama de una red neuronal recurrente desenrollada. 14
Figura 2.12: Diagrama de una LSTM. 15
Figura 2.13: Celda de estado. 15
Figura 2.14: Operaciones sobre la celda de estado. 16
Figura 2.15: Operaciones de la puerta del olvido. 16
Figura 2.16: Operaciones de la puerta del recuerdo. 17
Figura 2.17: Operaciones de la puerta de salida. 17
Figura 3.1: Detalles sobre la precisión del modelo es_core_news_md. 24
Figura 3.2: Detalles sobre el modelo es_core_news_md. 24
Figura 4.1: Estructura del buscador de datos personales. 28
Figura 4.2: Método para modificar texto plano. 31
Figura 4.3: Método para extraer entidades en texto plano. 31
Figura 4.4: Función para recorrer un documento .docx. 32
Figura 4.5: Algoritmo para extraer entidades en un documento Word. 33
Figura 4.6: Algoritmo para modificar documentos en Word. 34
Figura 4.7: Código para la extracción de información en documentos pdf. 36
Figura 4.8: Código para modificar documentos pdf. 37
Figura 4.9: Implementación de la clase TokenizerHtml. 39
Figura 4.10: Métodos de extracción y modificación de datos en HTML. 40
Figura 4.11: Código de la clase ColumnSelectorDataFrame. 41
Figura 4.12: Código de la clase DocumentHandlerCsv. 42
Figura 4.13: Expresión regular para buscar DNI. 43
Figura 4.14: Tablas para la verificación del DNI. 44
Figura 4.15: Código para realizar el filtro de entidades con el diccionario. 46
Figura 4.16: Código para buscar sinónimos es SpaCy. 48
Figura 4.17: Ejemplo de Patrones de SpaCy. 49
Figura 4.18: Arquitectura de tubería simple para un sistema de extracción de información. 50
Figura 4.19: Tubería de procesamiento de SpaCy. 50
Figura 4.20: Código para las extracciones de nombres mediante aprendizaje automático y extracciones de carnés de identidad mediante expresiones regulares. 52

Figura 4.21: Reglas para buscar nombres y apellidos mediante el etiquetador de SpaCy. 53
Figura 4.22: Código para las extracciones de nombres mediante patrones y extracciones de carnés de identidad mediante expresiones regulares. 54
Figura 4.23: Contenido del fichero heuristicMeasures.py. 54
Figura 4.24: Implementación de los algoritmos de ofuscación, disgregación y anonimización de datos. 56
Figura 4.25: Estructura del proyecto base con Flask-Restplus. 57
Figura 4.26: Configuración del servidor. 58
Figura 4.27: Código para generar la base de datos. 62
Figura 4.28: Comandos necesarios para la migración de la BD. 62
Figura 4.29: Contenido del fichero LogService.py, capa de servicio. 63
Figura 4.30:Ejemplo de la interfaz en pantallas grandes. 65
Figura 4.31: Ejemplo de la interfaz en dispositivos móviles. 66
Figura 5.1: Gráfica que muestra una comparativa con todas las pruebas hechas con datos no estructurados sacados de cartas. 71
Figura 5.2: Gráfica que muestra una comparativa con todas las pruebas hechas con datos no estructurados sacados de la web. 72
Figura 5.3: Gráfica que muestra una comparativa con todas las pruebas hechas con datos estructurados. 74
Figura 5.4:relación de falsos positivos y falsos negativos en textos para la red convolucional de SpaCy. 76
Figura 5.5: relación de falsos positivos y falsos negativos en web con la red convolucional de SpaCy. 76
Figura 5.6:relación de falsos positivos y falsos negativos en web para el sistema de reglas. 78
Figura 5.7: relación de falsos positivos y falsos negativos en textos para el sistema de reglas. 78

Índice de Tablas

Tabla 1.1 Planificación del TFG 3
Tabla 4.1: Estructura general del directorio del servidor 61
Tabla 4.2: Estructura del Frontend de la aplicación 65
Tabla 4.3: Explicación de las pruebas unitarias. 67
Tabla 5.1:: Resultados sin un diccionario de respaldo. 69
Tabla 5.2: Resultados para datos no estructurados con un umbral de coincidencia del 25%. 70
Tabla 5.3: Resultados para datos no estructurados con un umbral de coincidencia del 50%. 70
Tabla 5.4: Resultados para datos no estructurados con un umbral de coincidencia del 75%. 70
Tabla 5.5: Resultados para datos no estructurados con un umbral de coincidencia del 90%. 71
Tabla 5.6: Resultados de las métricas de las pruebas realizadas con datos no estructurados..... 73
Tabla 5.7: Mediciones de tiempo en segundos con la red convolucional de SpaCy, usando 100 documentos con formatos distintos. 76
Tabla 5.8: Mediciones de tiempo en segundos con el sistema basado en reglas, usando 100 documentos con formatos distintos. 76

Índice General

Agradecimientos I
Resumen II
Abstract III
Índice de Figuras IV
Índice de Tablas VI
Capítulo 1 Descripción del TFG 1
1.1 Introducción 1
1.2 Motivación Inicial 1
1.3 Objetivos 2
1.4 Planificación Temporal 3
1.5 Justificación de Competencias 3
1.6 Estructura del documento 4
Capítulo 2 Marco Teórico y Estado del Arte 5
2.1 ¿Qué es el NLP? 5
2.3 Segmentación y Tokenización 5
2.2 Modelos de Lenguajes 6
2.2.1 Word2Vec – El modelo skip-Gram 7
2.4 Etiquetador de palabras 8
2.4.1 Modelo N-Gram 9
2.5 Reconocimiento de Entidades Nombradas 9
2.6 Campos Condicionados aleatorios 10
2.7 Redes Neuronales Artificiales 10
2.7.1 Redes Neuronales Convolucionales 12
2.7.2 Redes Neuronales Recurrentes 14
2.7.3 Redes Neuronales de memoria a corto y largo plazo 15
2.10 Aprendizaje multitarea 18
2.11 Heurística 18
2.12 Arquitecturas para la reducción del riesgo de identificación de datos 18
Capítulo 3 Herramientas y Tecnologías usadas 21
3.1 Python 21
3.2 JQuery 21
3.3 Bootstrap y SCSS 21
3.4 Librerías para leer textos con formatos 22

3.4.1 Particularidades con documentos *Pdf* 22
3.5 Librerías de procesamiento del lenguaje Natural 23
3.5.1 SpaCy 23
3.5.2 NLTK 24
3.5.3 Standford CoreNLP 25
3.6 SQlite 25
3.7 Base de datos de nombres 25
3.8 Flask 25
3.9 Git 26
3.10 Trello 26
Capítulo 4 Diseño e Implementación 27
4.1 Extracción y modificación de información en Documentos 27
4.1.1 Diseño y clases 27
4.1.2 Documentos en Texto Plano 29
4.1.3 Documentos en Word 31
4.1.4 Documentos en pdf 34
4.1.5 Documento en HTML 37
4.1.6 Hojas de cálculo 40
4.2 Técnicas para la detección de entidades 42
4.2.1 Localización de Entidades y estudio sobre los modelos a utilizar 43
4.2.2 La extracción de información en textos sin contexto 46
4.2.3 La extracción de información en textos con contexto 49
4.2.4 Heurísticas utilizadas 54
4.3 Algoritmos de ofuscación, disgregación y anonimización de datos personales 55
4.4 Servicio Web 56
4.4.1 Api REST 57
4.4.2 Estructura de un proyecto en Flask-restPlus 57
4.4.3 Buscador de datos personales 59
4.5 Arquitectura y Diseño de Software 59
4.5.1 Model-Controller-Service 60
4.5.2 Estructura del servicio web 61
4.5.3 Base de Datos y Modelo 61
4.5.4 Servicio 63
4.5.5 Controlador 64

4.5.6. Frontend 65
4.6 Pruebas y Despliegue 66
Capítulo 5 Experimentos y Resultados 68
5.1 Pruebas de rendimiento 68
5.1.1 Datos no estructurados 69
5.1.2 Datos estructurados 73
5.2 Pruebas de tiempo 75
5.3 Estudio sobre los falsos negativos y positivos 76
Capítulo 6 Conclusiones 80
6.1 Valoración personal 80
6.2 Trabajos Futuros 81
Bibliografía 82
ANEXO 86
ANEXO I: Manual de Usuario 86
Ejecución 86
Guía de Usuario 87
ANEXO II: Repositorio 89
ANEXO III: Diagrama de clases 89

Capítulo 1 Descripción del TFG

1.1 Introducción

La información es un recurso fundamental en la sociedad actual para la toma de decisiones y otros aspectos de esta, esto es cada día más relevante sobre todo con el ascenso del *Big Data* y los nuevos avances en inteligencia artificial. La información nos puede brindar múltiples beneficios siempre que se mantenga el respeto por los derechos de las personas, su privacidad y la protección de sus datos personales. En este contexto, los procesos de reducción de riesgos de identificación adquieren un valor muy especial como fórmula que garantice el correcto avance de la sociedad de la información sin menoscabar el respeto a la protección de datos, tal como se indica en (Agencia Española De Protección De Datos 2016) [30] y cumpliendo con los aspectos recogidos en la "Ley Orgánica de Protección de Datos Personales y garantía de los derechos digitales" [41] y la declaración universal de derechos humanos [42].

En este trabajo se habla de conceptos como la anonimización, la ofuscación y la disgregación de datos personales. La anonimización consiste en el proceso general de transformación de dichos datos de manera irrecuperable ya sea borrándolos del texto o sustituyéndolos por algún conjunto de caracteres, la ofuscación es la transformación de los datos de tal manera que se oculte parte de la información que estos puedan dar, y la disgregación es un concepto parecido a la ofuscación, pero siguiendo una serie de pasos para garantizar que el titular de los datos pueda reconocerlos.

Con estos conceptos en mente, se pretende haciendo uso de los conocimientos adquiridos en el grado y en la mención de Computación, la creación de un sistema aplicando inteligencia artificial e ingeniería del software para la reducción del riesgo de identificación de datos personales en diferentes documentos, centrándose en los carnés de identidad (DNI) y en los nombres y apellidos de personas.

1.2 Motivación Inicial

Los procesos de anonimización/ofuscación/disgregación se suelen hacer a mano, esto requiere un esfuerzo extra por parte de las organizaciones o personas, localizando y tratando los datos personales en un sinfín de documentos. Aunque existan herramientas que ayudan a agilizar el proceso, suelen estar diseñadas para un ámbito o modelo de negocio muy concreto y su uso no es del todo automático, por ello la motivación para la realización de este proyecto es la de crear una herramienta que permita la completa automatización de esta tarea, proporcionando a nuestro ambiente socio-económico mecanismos útiles para cumplir con las leyes y reducir tiempos administrativos, por otro lado, a nuestro ambiente académico ofrece una propuesta de sistema inteligente híbrido aplicado a la búsqueda de entidades en textos, integrando conceptos de *Deep Learning* e Ingeniería del conocimiento.

La reducción del riesgo de identificación puede resultar de utilidad en cualquier ámbito donde se generen, gestionen o se trabaje de alguna manera con bases de datos documentales, donde los documentos contienen información confidencial. Particularmente donde dichos documentos tienen un valor intrínseco de índole científica o técnica, que es independiente de la información confidencial contenida en éstos como se remarca en (Vico, H. 2013) [40], algunos ejemplos serían:

- **Ámbito judicial y Administrativo:** Existen una gran cantidad de documentos legales o burocráticos, que necesitan de alguna herramienta que permita identificar datos personales que ponen en riesgo la integridad de los ciudadanos.
- **Ciencias de la Salud:** En el ámbito médico se generan gran cantidad de documentos con información relevante para consulta profesional. El ejemplo más notable es el de las historias clínicas. Estos documentos guardan información médica de un paciente, que puede ser utilizada para realizar un mejor diagnóstico o tratamiento de otros pacientes con síntomas similares. Es evidente que es necesario guardar la confidencialidad médico-paciente que puede ser vulnerada si la información contenida en dichos documentos llega a manos ajenas.
- **Ámbito Académico:** Tanto en universidades, escuelas e institutos se generan una gran cantidad de documentos con notas e informes sobre alumnos y profesores, que vulneraría la ley de protección de datos si se usasen fuera del ámbito de las instituciones académicas.

1.3 Objetivos

El objetivo que persigue este proyecto será el de eliminar o reducir al mínimo los riesgos de identificación de los titulares de los datos. Se trata de construir una herramienta con la que se podrá reducir los riesgos de identificación asociados a datos personales en diferentes tipos de documentos. Es decir, se pretende crear un sistema cuya entrada sea documentos (*docx*, *pdf*, *txt*, hojas de cálculo…) y la salida será la ofuscación/disgregación/anonimización de datos personales, en este caso nombres propios, apellidos y DNI. Se ha desarrollado aplicando técnicas de procesamiento del lenguaje natural, y librerías de *Python* relacionadas con este ámbito. De la misma forma se han utilizado herramientas aplicadas a la creación de servicios web.

Durante el desarrollo se ha contado con el asesoramiento de una empresa dedicada a la protección de datos llamada *Acoran*, con lo cual se han llevado a cabo diversas entrevistas a fin de validar todos los objetivos de este proyecto.

1.4 Planificación Temporal

Las 300 horas para el desarrollo del trabajo de fin de grado se han distribuido de la siguiente forma: 25 horas para la fase de estudio y análisis, 215 para diseño e implementación, 20 para la evaluación y 40 para la documentación y presentación. Durante el desarrollo del proyecto la distribución temporal de las diferentes fases se ha mantenido sin variaciones. Las fases y las tareas están definidas en la tabla 1.1.

Fases	***Duración Estimada*** ***(horas)***	***Tareas*** ***(nombre y descripción, obligatorio al menos una por fase)***
Estudio previo / Análisis	25	Tarea 1.1: Estudio del estado del arte.
		Tarea 1.2: Estudio sobre las técnicas de procesamiento natural.
		Tarea 1.3: Estudio sobre el procesamiento de lenguaje natural a la búsqueda de entidades (NER) en un texto.
Diseño / Desarrollo / Implementación	215	Tarea 2.1: Realización de una herramienta mediante *Python* que permita la lectura/escritura de textos en diferentes formatos como docx, *pdf*, txt, hojas de cálculo…
		Tarea 2.2: Desarrollo del código que permita la detención de entidades.
		Tarea 2.3: Integrar las aplicaciones/herramientas anteriores en un servicio web.
Evaluación / Validación / Prueba	20	Tarea 3.1: Estudiar el rendimiento de las técnicas de procesamiento de lenguaje natural utilizadas.
		Tarea 3.2: Estudiar el rendimiento del servicio.
Documentación / Presentación	40	Tarea 4.1: Realizar la memoria correspondiente de todo el proceso.

Tabla 1.1 Planificación del TFG

1.5 Justificación de Competencias

En este trabajo se han cubierto las siguientes competencias específicas propias de la mención de computación:

"CP01: Capacidad para tener un conocimiento profundo de los principios fundamentales y modelos de la computación y saberlos aplicar para interpretar, seleccionar, valorar, modelar, y crear nuevos conceptos, teorías, usos y desarrollos tecnológicos relacionados con la informática."

Esta competencia se ve cubierta pues en este proyecto se ha planteado toda una arquitectura para la reducción y la extracción de datos personales mediante modelos computacionales relacionados con la inteligencia artificial. Asimismo, se ha creado una tecnología útil para cumplir con los objetivos propuestos.

"**CP03:** Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos."

Esta competencia se ve cubierta pues para todo el desarrollo del sistema se ha tenido en consideración algoritmos y estrategias que garanticen un orden de complejidad adecuado para obtener un sistema lo más eficiente posible y en tiempo real.

"**CP04:** Capacidad para conocer los fundamentos, paradigmas y técnicas propias de los sistemas inteligentes y analizar, diseñar y construir sistemas, servicios y aplicaciones informáticas que utilicen dichas técnicas en cualquier ámbito de aplicación."

Esta competencia se ve cubierta mediante las tareas de investigación, comprensión y uso de diversos sistemas inteligentes como redes neuronales o técnicas basadas en ingeniería del conocimiento, todas ellas aplicadas al ámbito del procesamiento del lenguaje natural. Una vez completadas las tareas anteriores, se ha integrado todos estos conceptos en un software especializado en la anonimización/ofuscación/disgregación o extracción de datos personales.

1.6 Estructura del documento

Esta memoria pretende documentar y exponer el proceso que se ha llevado a cabo en este trabajo de fin de grado. Para ello se divide en varias partes concisas: el resumen, el *abstract*, los índices de figuras y tablas, y los capítulos que se exponen a continuación.

- **Introducción:** En este capítulo se presenta una introducción al problema a tratar y la motivación que llevó a abordarlo, se explica los objetivos a los que se quiere llegar y la planificación para conseguirlos, se justifica su utilidad para nuestro entorno socioeconómico y científico, y finalmente se justifican las competencias cubiertas.
- **Marco Teórico y Estado del Arte:** En este capítulo se incluyen conceptos básicos para comprender el procesamiento del lenguaje natural, así como también las diferentes técnicas aplicadas a la búsqueda de entidades nombradas. Se presenta el estado del arte actual en los procesos de reducción de datos personales y su relación con las técnicas de inteligencia artificial.
- **Herramientas y tecnologías usadas:** En este capítulo se exponen las herramientas y tecnologías más importantes en el desarrollo del proyecto.
- **Diseño e Implementación:** En este capítulo se explican los diferentes algoritmos y técnicas de inteligencia artificial que se han creado o usado para cumplir con los objetivos. Asimismo, se explican los conceptos de ingeniería del software, arquitectura y diseño con los que se han podido llevar a cabo todo el desarrollo.
- **Experimentos y Resultados:** En este capítulo se explican los diferentes experimentos realizados para las distintas técnicas y sus resultados.
- **Conclusiones:** En este capítulo se presentan las conclusiones, la valoración del trabajo y las posibles líneas futuras.

Capítulo 2 Marco Teórico y Estado del Arte

2.1 ¿Qué es el NLP?

Se entiende por lenguaje natural a todo lenguaje cuya comunicación se fundamenta entre humanos, estos lenguajes han evolucionado a medida que pasan de generación en generación y son difíciles de precisar con reglas concretas. El procesamiento de lenguaje natural es el conjunto de técnicas para la manipulación del lenguaje natural mediante la informática. Esta disciplina varía entre contar frecuencias de palabras que se repiten en un texto hasta hacer que un sistema informático "comprenda" las expresiones humanas más complejas (Bird et al. 2009) [1].

2.3 Segmentación y Tokenización

La *tokenización* es la tarea de dividir una cadena de caracteres en unidades lingüísticas identificables, por ejemplo, en palabras. Existen varias técnicas que se pueden aplicar, entre ellas se encuentra el uso de expresiones regulares que tengan en cuenta los espacios en blanco, retornos de carro, saltos de líneas y demás representaciones textuales, así pues, el nivel de complejidad de las expresiones regulares usadas aumenta con respecto a la calidad de la solución. Aunque parezca trivial, no existe una solución única que funcione bien para todos los ámbitos.

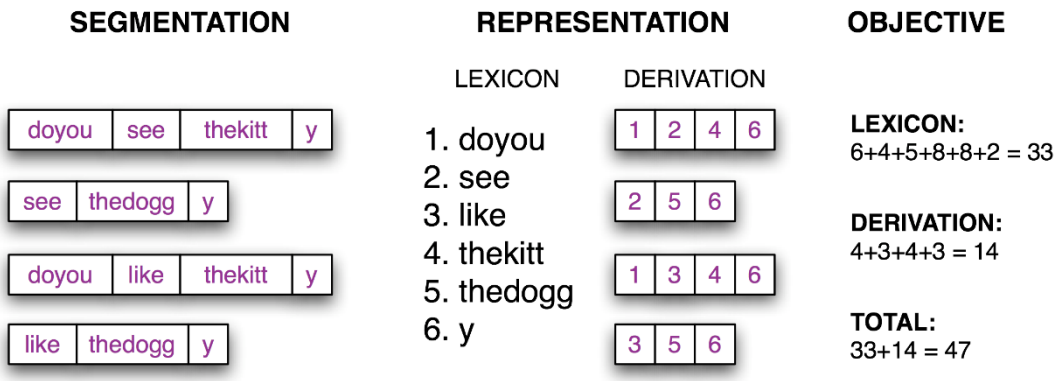


Figura 2.1: Cálculo de la función objetivo para tokenizar textos.

Fuente: (Bird et al. 2009) [1].

La *tokenización* es una instancia de un problema más general denominado *segmentación* como se indica en (Bird et al. 2009) [1], que abarcaría la división de textos en oraciones o palabras para cualquier lenguaje dado. Una manera de plantear la segmentación es considerándolo un problema de búsqueda donde tenemos que

encontrar la cadena binaria que representa la forma de segmentar un texto (donde un 0 simbolice que estamos ante una unidad lingüística y un 1 un punto a fragmentar). A continuación, podemos definir una *función objetivo*, una función de puntuación cuyo valor se tendrá que minimizar, en función del tamaño del léxico (número de caracteres en las palabras más un carácter delimitador adicional para marcar el final de cada palabra) y la cantidad de información necesaria para reconstruir el texto fuente del léxico, como se ilustra en la figura 2.1.

Haciendo uso de técnicas de búsquedas como algoritmos genéticos o búsquedas informadas se puede llegar a soluciones muy buenas relativamente rápido. La segmentación es muy necesaria para lenguajes que no dispongan de ningún delimitador de palabras como podría ser el chino o el japonés.

2.2 Modelos de Lenguajes

Un *modelo de lenguaje* no es más que la representación en un espacio vectorial de un conjunto finito de palabras donde cada una de ellas está representada como un vector en el espacio (McCormick, C. 2016) [17]. Lo interesante de este concepto es que se puede entrever dos propiedades básicas:

1. Las palabras similares o sinónimas tendrán vectores similares.
2. Se puede operar matemáticamente entre vectores para extraer relaciones entre palabras o formar otras nuevas. Así si se opera sobre la palabra rey y mujer y teniendo información que rey es hombre se podrá crear la palabra reina (Shure, L. 2017) [6], como lo ilustra la figura 2.2.

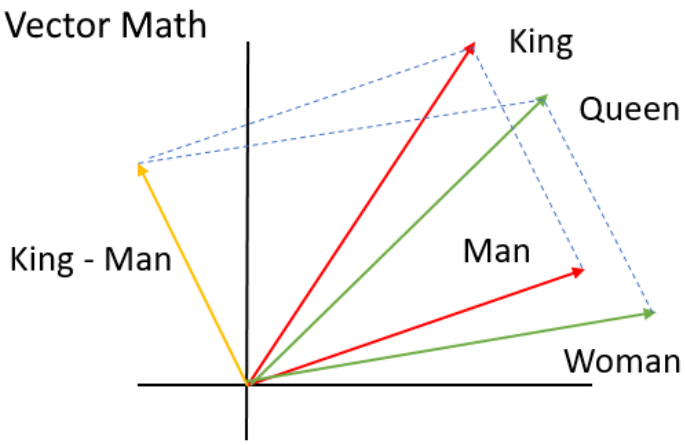


Figura 2.2: Representación del espacio vectorial de palabras.

Fuente:(Shure, L. 2017) [6]

Existen varias técnicas para generar un modelo de lenguaje, la más usada y con la cual se obtiene mejores resultados se llama *Word2Vec*.

2.2.1 Word2Vec – El modelo skip-Gram

El *Word2Vec* es una técnica de generación de modelos de lenguaje basada en aprendizaje no supervisado, de forma similar al tipo de entrenamiento usado para los autoencoders donde se entrena un codificador automático para comprimir un vector de entrada en la capa oculta y descomprimirlo de nuevo al original en la capa de salida. Después de entrenarlo, se quita la capa de salida y simplemente se usa la capa oculta: es un truco para aprender buenas características de imágenes sin tener que etiquetar los datos de entrenamiento.

Se trata, en primer lugar, de entrenar un predictor de palabras, es decir, dada una palabra perteneciente a un vocabulario conocido la red será capaz de inferir la palabra más próxima dentro del mismo vocabulario. El tamaño del vocabulario que puede utilizar la red entra dentro de los datos de capacitación utilizados, que usualmente suelen ser documentos o frases y oraciones.

Se entrena haciendo uso de pares de palabras extraídas de los datos de entrenamiento donde el tamaño del contexto vendrá dado por el tamaño de ventana usada en la extracción, la figura 2.3 muestra un ejemplo.

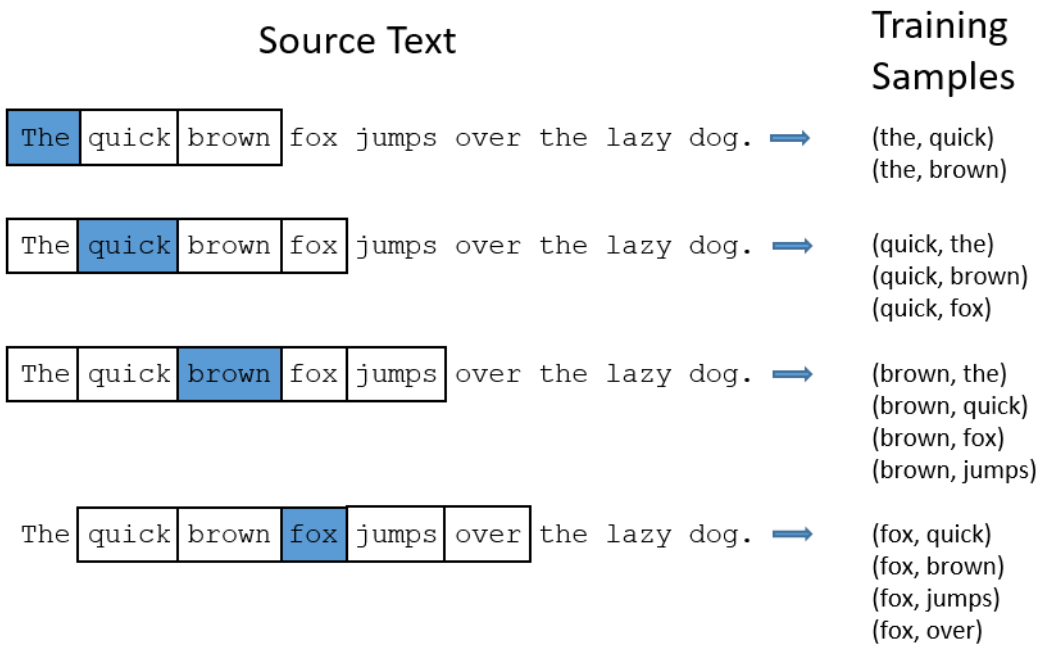


Figura 2.3:Extracción de pares de palabras.

Fuente: (McCormick, C. 2016) [17]

La red aprenderá las estadísticas del número de veces que aparece cada emparejamiento. Por ejemplo, la red probablemente obtendrá muchas más muestras de entrenamiento de ("Unión", "Soviética") que dé ("Ardilla", "Soviética"). Cuando finalice el entrenamiento, si se le da la palabra "soviético" como entrada, entonces generará una

probabilidad mucho mayor para "Unión" o "Rusia" que para "Ardilla" (McCormick, C. 2016) [17].

El tipo de modelo usado para crear el predictor suele ser el mismo, salvando una serie de mejoras o retoques a nivel de cálculo de pesos y de optimizadores que pueden ser diferentes según el idioma o el tipo de problema que nos encontremos. La red consiste en un modelo clásico de tres capas como se define en (McCormick, C. 2016) [17], visible en la imagen 2.4:

- **La capa de entrada**: su tamaño está definido según el número de palabras sacadas de los datos de entrenamiento. Se usa el one hot para codificar las palabras de entrada.
- **La capa intermedia**: que en este caso no dispone de función de activación.
- **La capa de salida**: con una softmax como función de salida. En el entrenamiento se utiliza el vector one hot de la pareja de cada par de palabras como etiqueta.

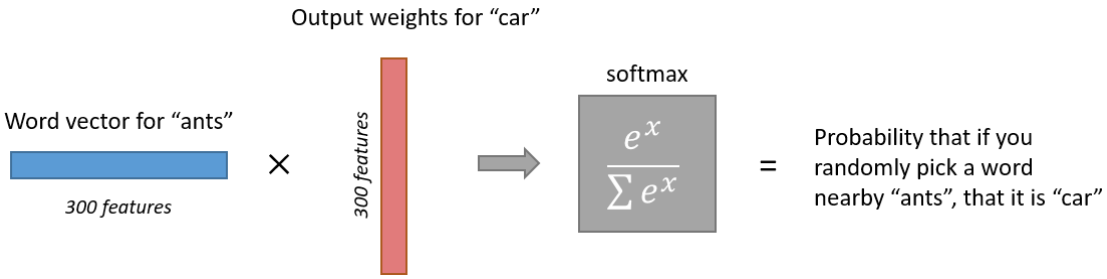


Figura 2.4: Red skip-Gram.

Fuente: (McCormick, C. 2016) [17]

Lo que se pretende con esta técnica es despreciar los resultados de la capa de salida y aprovechar los pesos de la capa intermedia como matriz de búsqueda, de tal forma que cada palabra del vocabulario tendrá sus representación en ese espacio vectorial, que al haberse entrenado como unos pesos dedicados a predecir “cercanías” de palabras según el contexto, aparte de representar palabras, los pesos se agruparán si tienen contexto similares, por lo que dos palabras que sean sinónimas tenderán a tener vectores parecidos.

2.4 Etiquetador de palabras

Etiquetar un texto es el mapeo o asociación de los tokens extraídos a una etiqueta universal de discurso, se entiende por etiquetas de discurso a marcar cada token para saber si representan a verbos, adjetivos, sustantivos, etc.

Existen muchas variantes de asociación, las más clásicas varían entre usar expresiones regulares en cada token para asociarlos a una etiqueta o una base de conocimiento para ayudar a mapearlos, pero se suele apostar más por técnicas que se apoyan en el contexto.

2.4.1 Modelo N-Gram

Este tipo de etiquetado se basa en utilizar el contexto para ayudar a inferir la etiqueta más adecuada a cada token. Se plantea un tamaño de ventana N (suele ser entre 4 o 5), se recorre todo el texto extrayendo el token correspondiente más todos los anteriores respetando el tamaño de ventana. Este tipo de modelo se puede basar en métodos estadísticos o en redes neuronales para realizar la inferencia, pero necesitan de un proceso de capacitación para hacer bien la predicción, además se suelen apoyar en modelos de lenguajes para mejorar su etiquetado.

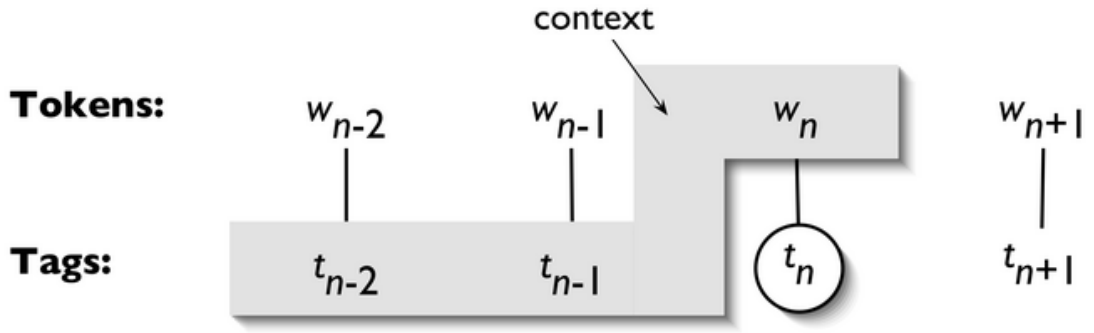


Figura 2.5: Etiquetado basado en el contexto.

Fuente: (Bird et al. 2009) [1].

Las técnicas basadas en el contexto se pueden aplicar para etiquetar también las dependencias entre las palabras, ejemplos de etiquetadores y su implementación se presentan en (Bird et al. 2009) [1].

2.5 Reconocimiento de Entidades Nombradas

El reconocimiento de entidades nombradas (NER) es la tarea de localizar y categorizar términos importantes de un texto (Hassan, F. et al. 2018) [2]. Entre las posibilidades caben: personas, localidades, emails, cantidades, expresiones de tiempo, etc. Existen varias técnicas y aproximaciones, pero se pueden agrupar en tres grandes grupos:

1. **Técnicas clásicas**: mediante ingeniería del conocimiento y sistemas expertos, fundamentalmente aproximaciones basadas en reglas. Son técnicas que suelen dar buenos resultados para búsquedas de entidades complejas pero que no pueden salir del dominio del lenguaje en el que están implementadas (Hassan, F. et al. 2018) [2].
2. **Basadas en aprendizaje automático**: mediante redes neuronales y otras técnicas basadas en unidades de aprendizaje. Es muy común el uso de LSTM o modelos secuenciales, los cuales suelen dar buenos resultados. Su principal desventaja es la necesidad de un gran corpus textual de entrenamiento si estamos bajo un enfoque supervisado, ejemplos de estos están en (Lample, G. et al. 2016) [29] o (Arevalo, M. et al. 2002) [35]. Se podría hablar en este grupo también de algunos métodos basados en aprendizaje evolutivo, que consisten en plantear la detención de entidades como un espacio de estados y optimizar la búsqueda mediante algoritmos genéticos, un ejemplo sería el trabajo realizado en (Hasanuzzaman, M. et al. 2010) [3].
3. **Técnicas híbridas**: Se aprovechan del uso de algún sistema basado en reglas y de técnicas de aprendizaje automático. Están diseñados para aprovechar las ventajas de los grupos anteriores, pero siguen teniendo el problema del dominio del lenguaje (Hassan, F. et al. 2018) [2].

2.6 Campos Condicionados aleatorios

En procesamiento de lenguaje natural, hay dos modelos comunes para resolver tareas de etiquetado de palabras y entidades: *modelos de Markov ocultos* y *campos condicionados aleatorios* (CRFs). Los etiquetadores basados en CRFs son ampliamente usados y aplicados, y normalmente dan mejores resultados en varios campos (Hassan, F. et al. 2018) [2].

Los CRFs son modelos de grafos no dirigidos entrenados de forma condicionada que a menudo se aplican en el reconocimiento de patrones. Estos modelos se usan para calcular la probabilidad condicionada de valores en determinados nodos de salida dados los valores asignados a otros nodos de entrada, como se explica en (Lafferty, J. et al. 2001) [5].

2.7 Redes Neuronales Artificiales

Las *redes neuronales artificiales* (RNA) constituyen un paradigma de computación inspirado en las neuronas biológicas y su interconexión. Las neuronas biológicas son células compuestas principalmente de tres partes: soma (cuerpo celular), dendritas (canales de entrada) y axón (canal de salida) como se puede ver en la figura 2.6. Descrito de una forma muy simplificada, las neuronas transmiten información a través de procesos electroquímicos. Cuando una neurona recibe, a través de las dendritas, una cantidad de estímulos mayor a un cierto umbral, ésta se despolariza excitando, a través del axón, a otras neuronas próximas conectadas a través de las sinapsis.

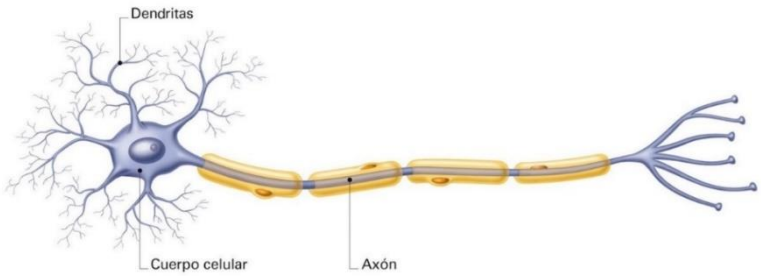


Figura 2.6: Partes de una neurona.

Fuente: (Guerra-Artal, C.) [18]

Inspirados por esta idea se concibió el modelo de neurona artificial. Fundamentalmente, consiste en una unidad de cálculo que admite como entrada un vector de características $\vec{e}$ cuyos valores se suman de forma ponderada mediante unos pesos $\vec{w}$ y, si esta suma supera un cierto umbral θ, genera un cierto valor de salida, por ejemplo 1 y, si no lo supera, genera otro valor, por ejemplo, un 0 (Guerra-Artal, C.) [18]. Cuando la neurona no está conectada a otras conformando una red, actúa como un clasificador lineal. La expresión más básica de la neurona artificial es la siguiente:

$$f(e)=\begin{cases} 1, \sum_{i=0}^{n} w_i e_i \geq \theta \\ 0, en\ caso\ contrario \end{cases} \quad (2.1)$$

Se trata de un artefacto que simboliza una recta en dos dimensiones que pretende separar linealmente un conjunto de datos, de esa manera el método de aprendizaje consiste en mover dicha recta para poder realizar una mejor inferencia con las muestras dadas mediante la actualización de los pesos. El método más común para la actualización es el *descenso por el gradiente*, para aplicar esta técnica se necesita que la salida de la neurona sea una función derivable, denomina función de activación. La principal desventaja que presenta este modelo es que generalmente, solo se puede presentar funciones linealmente separables. Para solventar este problema surge la idea de las redes neuronales, un conjunto de neuronas interconectadas formando una red, se caracteriza porque al menos disponen de tres capas básicas: una capa de entrada, una capa oculta y una capa de salida. Cada una de ellas con un número determinado de neuronas. La ventaja de disponer de un modelo multicapa es que aumenta el espacio de hipótesis que puede representar nuestra red y por tanto puede actuar en funciones no lineales (Russell, S. J. & Norvig, P. 2004) [28].

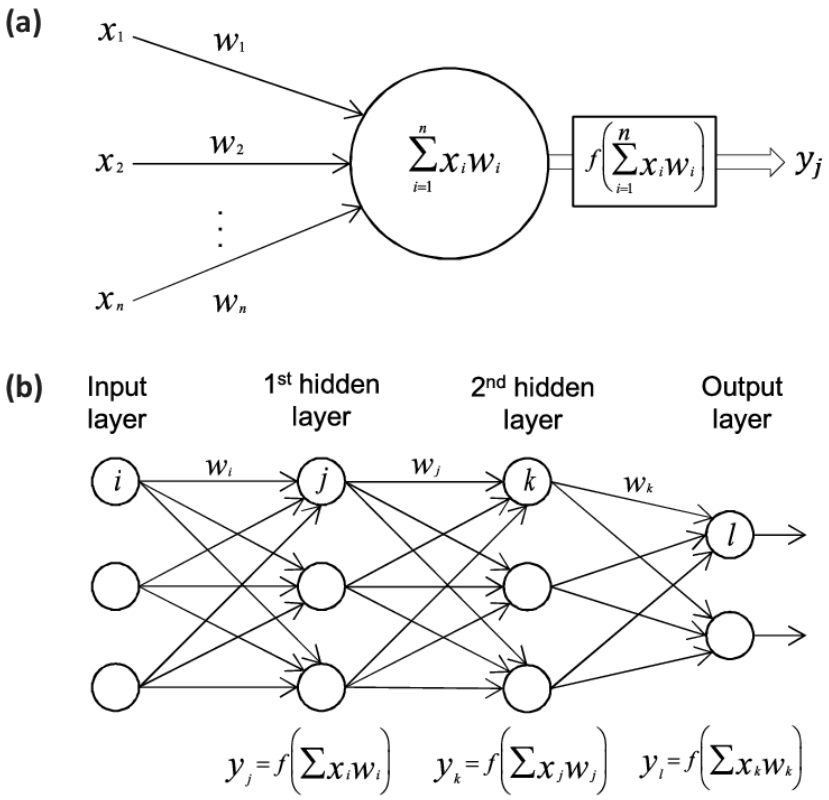


Figura 2.7: Diagramas de modelos neuronales.

Fuente: (Vieira et al. 2017) [19].

Las redes neuronales utilizan una técnica de aprendizaje supervisado denominada *backpropagation* (Jurafsky, D. & Martin, J. H. 2019) [25]. Esta técnica nos permite modificar los pesos de todas las capas, no solo los de la capa de salida. Este algoritmo se utiliza junto a algoritmos de optimización, como puede ser el *Descenso por Gradiente Estocástico* (SGD). Una vez obtenida la salida de nuestra red, se compara el resultado real y el devuelto con la función de coste, obteniendo el error de la red. Al obtener este valor, se propaga hacia atrás hacia la capa de neuronas anterior, sirviendo para ajustar los pesos de cada una de ellas, que generalmente se fijan aleatoriamente al comienzo del entrenamiento. El error se sigue propagando hacia atrás hasta llegar a la capa de entrada. Este proceso se realiza con todas las instancias del conjunto de entrenamiento. El aprendizaje de los pesos correspondientes en las capas ocultas permite a las redes aprender nuevas características del conjunto de datos. Se denomina aprendizaje supervisado debido a que en el entrenamiento se utilizan las etiquetas, que determina a qué clase pertenecen las instancias, para poder calcular el error con el que se modifican los pesos para que la red aprenda.

2.7.1 Redes Neuronales Convolucionales

Las *Redes Neuronales Convolucionales* son un tipo de redes neuronales profundas muy utilizadas cuando se trata con datos con una alta dimensionalidad, como videos o

imágenes. Sin embargo, también son útiles a la hora de tratar con texto y hacer tareas propias del NLP.

Las CNN disponen de una arquitectura básica: una capa de entrada, múltiples capas ocultas y una capa de salida. Su diferencia principal es que estas redes disponen de capas de convolución. Una capa de convolución consta de un conjunto de filtros, llamado *kernels*, que aplican una operación de convolución a una entrada determinada para generar un mapa de características como salida, en términos matemáticos estas capas están diseñadas para producir tensores. Los filtros son en realidad una matriz de pesos que se verán modificados en el entrenamiento, encargados de extraer las características de los datos. En la figura 2.9 se puede observar cómo se realiza la función de convolución. Básicamente consiste en pasar los kernels por el mapa de características realizando una suma ponderada entre los pesos de los kernels y las subregiones del mapa hasta completarlo, extrayendo los resultados en varios vectores numéricos que se usarán para construir el tensor de salida. Como los tensores generados pueden llegar a tener una gran dimensionalidad se suelen usar capas de *Maxpolling* para reducir las dimensionalidades de los mapas de características, su funcionamiento consiste en obtener una representación más pequeña de las características principales descartando aquellos menos relevantes, lo que nos puede proporcionar un mejor rendimiento y ayuda a evitar el sobreajuste. Al final de la red se dispone una capa *fully connect* que se encargará de clasificar el mapa de características, en la figura 2.8 se muestra la arquitectura de esta red (Saha, S. 2018) [21].

Este tipo de redes no son solo aplicables a imágenes sino también a textos. Para ello una de las técnicas más usadas es transformar los tokens en vectores para crear un mapa de características (es decir hacer uso de modelos de lenguaje) con el que la CNN puede hacer inferencias. Aunque las CNN aplicadas a textos son complicadas de entrenar, tienen una ventaja crucial y es que dan un rendimiento temporal asombroso, y se pueden hacer uso de ellas para tareas propias del NLP como son el etiquetado de palabras o de entidades. En este proyecto se usan CNN basados en aprendizaje multitarea proporcionados por la librería *SpaCy* de (Explosion AI) [14].

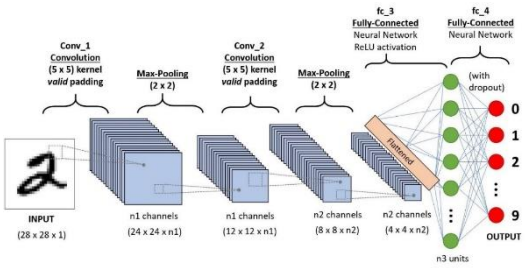


Figura 2.8: Típica arquitectura de una red convolucional.

Fuente: (Saha, S. 2018) [21].

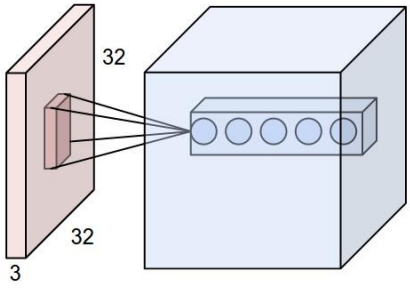


Figura 2.9: Convolución.

Fuente: (Saha, S. 2018) [21].

2.7.2 Redes Neuronales Recurrentes

Los seres humanos no aprenden a pensar desde cero cada segundo, sino que sus pensamientos tienen persistencia, por lo que cualquier actividad relacionada con la compresión del lenguaje se vuelve mucho más sencilla. Las *redes neuronales recurrentes* (Olah, C. 2015) [16] abordan este problema. Son redes con bucles que permiten que la información persista.

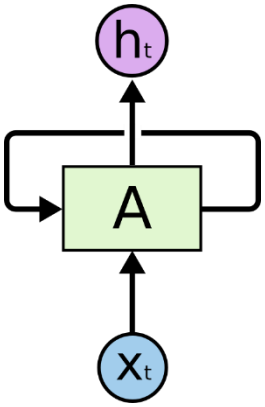


Figura 2.10: Diagrama de una red neuronal recurrente.

Fuente: (Olah, C. 2015) [16].

En el diagrama anterior, un trozo de red neuronal, A, recibe una entrada x_t y genera un valor h_t. Un bucle permite que la información pase de un estado de la red al siguiente. Una red neuronal recurrente puede considerarse como un conjunto de copias múltiples de la misma red, cada una de las cuales pasa un mensaje a un sucesor.

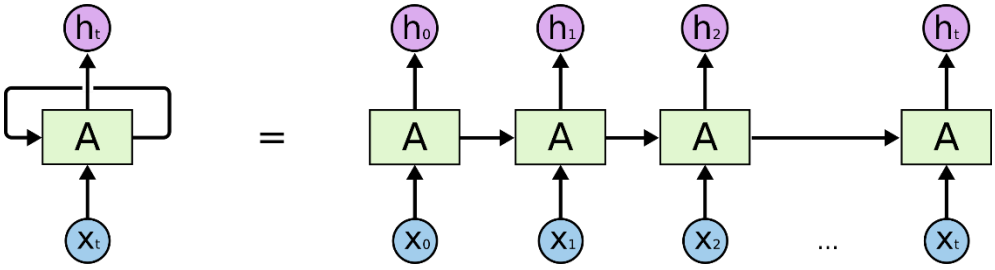


Figura 2.11: Diagrama de una red neuronal recurrente desenrollada.

Fuente: (Olah, C. 2015) [16].

Esta naturaleza en cadena revela que las redes neuronales recurrentes están íntimamente relacionadas con secuencias y listas. Son la arquitectura más adecuada a utilizar para dichos datos.

Las RNN son ideales para problemas donde se necesite un contexto, siempre y cuando dicho contexto no sea demasiado amplio, debido a que como solo disponen de información del instante anterior la capacidad de guardar conocimiento previo es bastante limitada por lo que para problemas reales no van a dar un buen rendimiento, por ello se desarrollaron las LSTM.

2.7.3 Redes Neuronales de memoria a corto y largo plazo

Las *redes de memoria a corto y largo plazo*, generalmente llamadas LSTM, son un tipo especial de RNN, capaces de aprender dependencias a largo plazo. Fueron introducidas en (Hochreiter, S. & Schmidhuber, J. 1999) [4], y refinadas y popularizadas por muchas personas en futuros trabajo. Funcionan extremadamente bien en una gran variedad de problemas, ahora se usan ampliamente.

Las LSTM están diseñadas explícitamente para evitar el problema de dependencia a largo plazo, también disponen de una estructura tipo cadena, pero el módulo de repetición tiene una composición diferente. En lugar de tener una sola capa, hay cuatro que interactúan de una manera muy especial (Olah, C. 2015) [16].

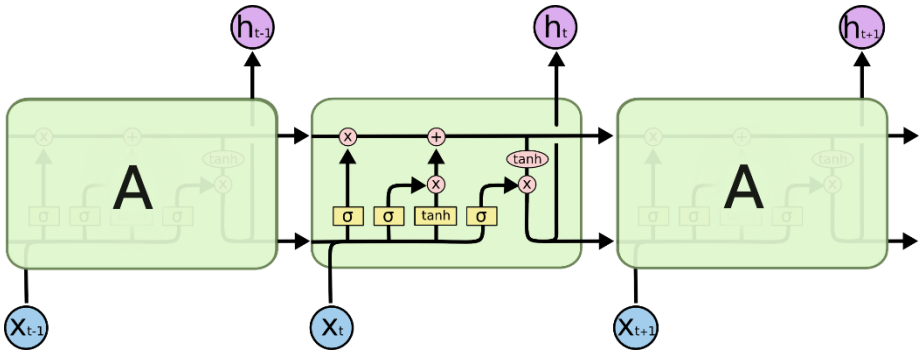


Figura 2.12: Diagrama de una LSTM.

Fuente: (Olah, C. 2015) [16].

La idea central de las LSTM es la *cell state* (célula de estado). Es una especie de cinta transportadora donde la LSTM tiene la posibilidad de añadir o retirar información, cuidadosamente regulado por las *gates* (puertas).

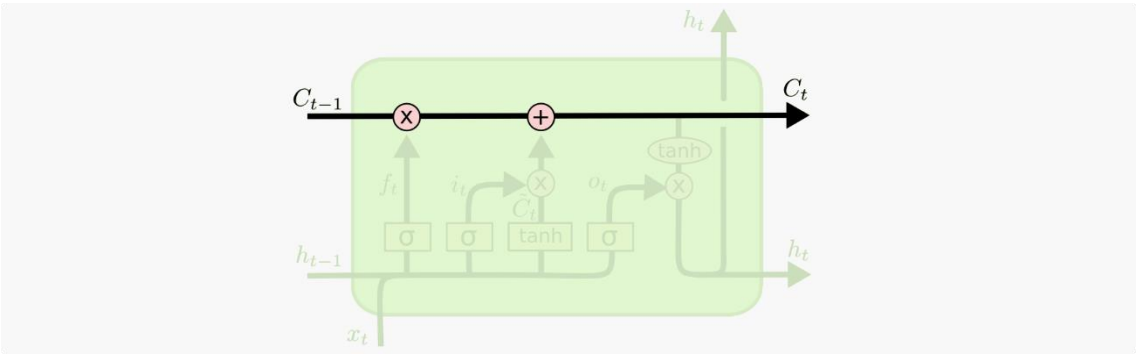
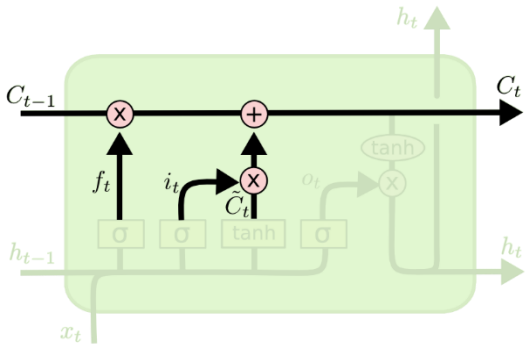


Figura 2.13: Celda de estado.

Fuente: (Olah, C. 2015) [16].

Las puertas son una forma de, opcionalmente, dejar pasar la información. Están compuestas por una capa de red neuronal y una operación de multiplicación sobre cada

elemento. La capa *sigmoide* genera números entre 0 y 1, describiendo cuánto de cada componente debe pasar.



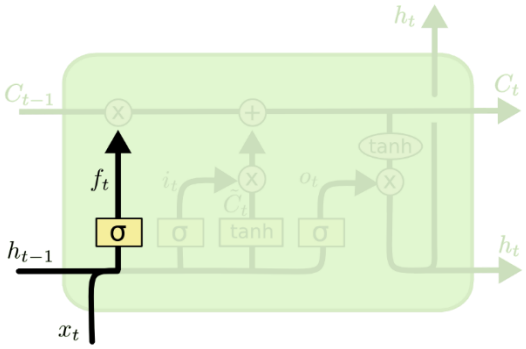
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figura 2.14: Operaciones sobre la celda de estado.

Fuente: (Olah, C. 2015) [16].

La puerta del olvido

El primer paso en la LSTM es decidir qué información vamos a dejar pasar desde la *cell state*. Esta decisión es tomada por la capa *sigmoide* llamada *forget gate layer*. Un ejemplo, suponiendo que tenemos un modelo del lenguaje, la *cell state* podría incluir el género del sujeto para usar el pronombre correcto. Cuando vemos un nuevo sujeto querremos "olvidar" el género del anterior.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

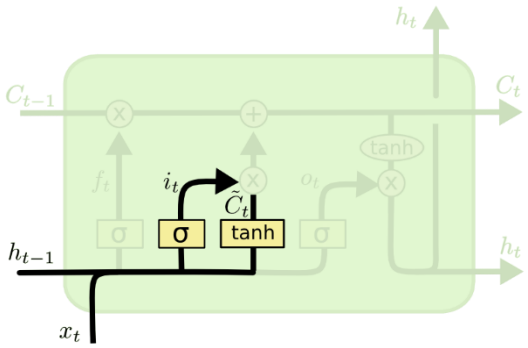
Figura 2.15: Operaciones de la puerta del olvido.

Fuente: (Olah, C. 2015) [16].

La puerta del recuerdo

El siguiente paso es decidir qué vamos a guardar (recordar) en la *cell state*. Esto conlleva dos partes. Primero, una capa *sigmoide*, llamada *input gate layer* decide qué valores se actualizarán. Después, una capa *tanh* creará un vector de nuevos valores candidatos, C_t, que podrían ser añadidos al estado. En nuestro ejemplo, aquí añadiríamos el género de nuestro nuevo sujeto.

Ahora es el momento de actualizar la *cell state* antigua por la nueva. Multiplicamos el antiguo estado por ft, olvidando las cosas que hemos decidido olvidar y añadiendo la información nueva C_t.



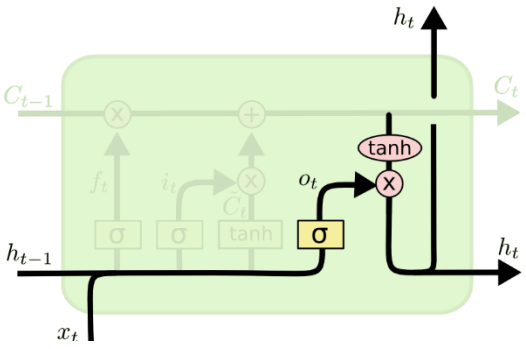
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Figura 2.16: Operaciones de la puerta del recuerdo.

Fuente: (Olah, C. 2015) [16].

La salida

Finalmente, necesitamos decidir lo que vamos a devolver como salida. Esta salida estará basada en nuestra *cell state*, pero con una versión filtrada. Primero, una capa *sigmoide* decide qué partes de la *cell state* devolverá. Después, pasamos la *cell state* por la función *tanh* para pasar los valores al intervalo [-1,1].



$$o_t = \sigma\left(W_o \; [\, h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figura 2.17: Operaciones de la puerta de salida.

Fuente: (Olah, C 2015) [16].

2.10 Aprendizaje multitarea

En el aprendizaje automático básico el uso de los modelos de aprendizaje está orientado a una única tarea, es decir, el entrenamiento y el ajuste de los hiperparámetros se realizan enfocados a minimizar una única función de pérdida, de esta forma se pueden entrenar modelos para clasificar imágenes o analizar textos. Pero cuando tenemos una tarea muy compleja el entrenamiento puede llegar a ser muy duro e ineficiente.

El aprendizaje multitarea es una subrama dentro del aprendizaje automático que busca dividir o separar en varias una tarea que por el contrario sería muy compleja de realizar para un único modelo. El enfoque multitarea busca disponer de varios modelos de aprendizaje que se pasen información entre ellos, es decir, que entre ellos dispongan de la información de los pesos de los otros con el objetivo de actualizar los suyos propios. Es ideal en problemas con más de una función de pérdida. De esta forma a la hora de analizar una oración podemos hacer el análisis sintáctico o de dependencias a la vez que con el de entidades, con el objetivo de que la inferencia sea más precisa gracias a esta técnica.

2.11 Heurística

Se puede definir la heurística como un conjunto de criterios, métodos o principios para decir cuál de entre diversos cursos de acción alternativos "promete" ser el más efectivo para alcanzar algún objetivo. En suma, una heurística es alguna idea que nos guía basada en la intuición o en la experiencia pero que realmente no tiene algún tipo de explicación fundamentada.

La heurística es muy común en varias ramas de la inteligencia artificial, sobre todo en búsquedas. En este proyecto se utiliza en varias partes tanto para analizar nombres en tablas, como para leer documentos y descartar nombres que no parecen que lo sean.

2.12 Arquitecturas para la reducción del riesgo de identificación de datos

En esta sección se presenta algunos ejemplos de arquitecturas y softwares para la reducción de identificación de datos personales en castellano, todas ellas están relacionadas o pensadas para un ámbito bio-médico, pero nos pueden servir como punto de partida para crear una arquitectura aplicada a nuestro dominio del problema.

Arquitectura ANONIMYTEXT

Se trata una arquitectura completa para un software de anonimización de información médica no estructurada, se utiliza una combinación de técnicas para lograr la identificación de la información sensible en el texto (Vico, H. 2013) [40]. El sistema "Anonimytext" se compone de los siguientes subsistemas:

- **Módulo de inducción por diccionarios:** En una primera instancia un experto del negocio (un médico) utiliza el módulo de inducción por diccionarios, para definir una lista de conceptos "sensibles" que aparecen en los documentos, por ejemplo: nombres, direcciones, fechas…. Para cada uno de ellos el experto deberá proveer una lista de ejemplos de estos que se utilizan como "semillas" para generar un diccionario "inducido". Un diccionario inducido es un concepto que permite, a partir de una semilla y utilizando un diccionario obtener un diccionario de "equivalencias" adaptado al dominio particular (Vico, H. 2013) [40].
- **Etiquetador:** Detecta y delimita los datos sensibles dentro de los documentos. Este módulo puede utilizar técnicas de diccionario, heurísticas o mediante búsqueda de entidades nombradas.
- **Modulo Revisor:** El objetivo de este módulo es presentar a un experto del negocio el resultado del proceso de etiquetado sobre un texto ya procesado, y permitir aceptar o rechazar la inferencia de datos sensibles que pueda haberse hecho. Este módulo podría aprovechar el conocimiento del experto para retroalimentar al sistema, utilizando otro componente que presentaremos más adelante que llamaremos retroalimentador.
- **Anonimizador:** Sustituye los datos sensibles del texto, previamente marcados y aceptados, por algún término genérico, o por alguna referencia que permita revertir el proceso contando con una autorización o clave.
- **Retroalimentación:** La propuesta de ANONYMITEXT contempla la retro-alimentación del sistema en base a la salida "rechazo" desde el módulo revisor. En caso de que un experto determine que el proceso de etiquetado no identificó correctamente algún término, debería existir una interfaz que le permita indicar el tipo de error que ocurrió y de esa forma reiniciar el proceso con esta nueva información incorporada (Vico, H. 2013) [40].

Arquitectura MOSTAS

Se trata de un framework de procesamiento de textos en castellano, con la siguiente distribución de módulos (Vico, H. 2013) [40]:

- **Analizador morfo-semántico:** Este componente recibe las notas clínicas en formato no estructurado. Mediante una herramienta llamada STILUS, se realizan búsquedas de todas las palabras en un diccionario general del lenguaje español. De esta manera se identifican los términos generales que no tienen valor desde el punto de vista biomédico.
- **Motor de búsqueda de términos:** Las palabras no reconocidas por el analizador morfo-semántico, se buscan en otros diccionarios más específicos, de siglas, abreviaturas y acrónimos biomédicos. Si se encuentran definiciones en estos últimos, se guardan en un documento XML los significados.
- **NER (Anonimización):** Nuevamente, alimentado por los términos que aún no han sido reconocidos por el motor de búsqueda anterior, se procesan y se reconocen entidades nombradas (NER). En este componente se utiliza una vez más el software STILUS.
- **Verificador ortográfico:** El sistema MOSTAS añade un concepto interesante con este módulo corrector. Se plantea la posibilidad de que aquellos términos que no han sido reconocidos por los tres módulos anteriormente mencionados pueden estar mal

escritos. Lo que se hace es buscar por similitud los términos restantes en los recursos de información médica que se utilizaron anteriormente.

Arquitectura HIDE

Se trata de otro framework de procesamiento de textos en castellano, en este caso es *open source* bajo licencia MIT. La arquitectura propuesta es la siguiente (Vico, H. 2013) [40]:

- **Interfaz de anotación:** En primera instancia HIDE utiliza un proceso iterativo de etiquetado (tagging), basado en el entrenamiento mediante textos previamente etiquetados por un experto. Por lo tanto, se ofrece un módulo que le permite al experto etiquetar textos identificando la información sensible.
- **Extracción de características:** Este módulo lo que hace es identificar una serie de "características" de interés para los términos que se encuentran, tanto en los textos etiquetados de entrenamiento como en los textos nuevos a ser anonimizados.
- **Módulo Clasificador:** El texto con sus características, y los textos de entrenamiento alimentan un módulo que clasifica los términos, utilizando un método estadístico/probabilístico. La salida de este módulo es el texto etiquetado y listo para ser anonimizado.
- **Data Linking / Identifier Extraction:** La arquitectura de HIDE provee un módulo "Data Linker" que vincula toda la información sensible a una entidad. El sistema realiza un trabajo iterativo, extrayendo atributos del texto (con un módulo llamado Identifier Extractor), y vinculando los atributos a las entidades con el Data Linker. Lo que se hace es vincular internamente todos los atributos que son de una misma entidad (ejemplo, edad, dirección y nombre de una persona), generando las estructuras de datos adecuadas para representar a las distintas entidades nombradas que se encuentren en el texto. El módulo Linker utiliza técnicas probabilísticas para vincular los atributos, y se utiliza una base de datos externa con información de entidades para la tarea (ej. base de datos de personas, ubicaciones geográficas, etc).
- **Vista de Identificadores:** Ya teniendo los atributos identificados y vinculados a entidades nombradas, se pasa a un módulo de presentación al usuario que se denomina "Vista de Identificadores". Allí se presenta el texto con sus identificadores y se permite elegir una estrategia de anonimización. Se permite realizar una anonimización completa o parcial (de algunos atributos críticos).
- **Anonimizador**: Finalmente el texto pasa al módulo anonimizador, el cual elimina los atributos sensibles en base al criterio indicado en la vista de identificadores.

Capítulo 3 Herramientas y Tecnologías usadas

3.1 Python

Todo este trabajo se encuentra desarrollado en *Python* debido a que actualmente es el lenguaje referencia en el campo de la inteligencia artificial y la ciencia de datos, además su sencillez y versatilidad funcionan muy bien para el prototipado de aplicaciones sobre todo para servicios web *REST*. Asimismo, es un lenguaje *multiparadigma* que nos brinda varias librerías para el procesamiento de documentos y textos en varios formatos. En este TFG se ha utilizado la versión 3.6 de este lenguaje debido a que algunas librerías que se han utilizados son estables en esta versión. Cabe destacar que el entorno de desarrollo que he usado durante el proyecto ha sido Visual Studio Code, por su simpleza y la comodidad para escribir código en *Python*.

3.2 JQuery

JQuery es una biblioteca de *JavaScript* rápida, pequeña y rica en funciones. Se encarga de diferentes funcionalidades como el desplazamiento y la manipulación de documentos HTML, el manejo de eventos, la animación y que las peticiones Ajax sean mucho más simples, con una API fácil de usar que funciona en una multitud de navegadores (The *jQuery* Foundation 2020) [36]. En este trabajo se ha usado esta librería para programar parte del *frontend* del software.

3.3 Bootstrap y SCSS

Bootstrap es un marco *frontend* gratuito para un desarrollo web más rápido y fácil, incluye plantillas de diseño basadas en HTML y CSS para tipografía, formularios, botones, tablas, navegación, modales, carruseles de imágenes y muchos otros, así como complementos de *JavaScript* opcionales. *Bootstrap* también nos brinda la capacidad de crear fácilmente diseños receptivos (W3C 2020) [37]. SCSS o Sass es un lenguaje de hojas de estilos que se inspira en CSS pero añade más versatilidad y nuevas funcionalidades al lenguaje estándar de diseño. Se han usado estas tecnologías para el diseño de la interfaz web.

3.4 Librerías para leer textos con formatos

Este proyecto se enfoca al manejo de aquellos textos considerados de formato, es decir propios de *Word*, *Excel*, *Pdf*, *Csv* y HTML. Como el lenguaje usado para el tratamiento de documentos con estas características es *Python* se han escogido cinco librerías fundamentales para este lenguaje:

- ***Python*-docx**: esencial para leer y crear textos con extensión docx, trata a los documentos como un conjunto de páginas divididas en diferentes objetos: párrafos, imágenes y tablas. Cada uno de esos objetos dispone de una serie de métodos para su manejo propio de estos documentos: añadir estilo, cambiar la fuente, modificar el tipo o el tamaño de letra, aparte de manejar información textual. Los párrafos en particular están compuestos de *runs*, pequeñas porciones de textos, con los mismos atributos y métodos que un párrafo, pero manejan una menor cantidad de información (Canny, S. 2013) [7].
- **Pandas**: librería muy usada en ciencia de datos, con ella se puede leer, modificar y crear archivos *excel* y *csv*. Se usa está librería para iterar sobre las celdas y poder modificarlas, dispone de una gran cantidad de métodos para ello (McKinney, W. 2017) [8].
- ***Pdf*miner.six**: su fundamento es muy parecido a *Python-docx*, itera sobre el documento *pdf* según una serie de objetos: párrafos, imágenes y tablas. *PDFMiner* intenta reconstruir algunas de esas estructuras mediante el uso de heurística en el posicionamiento de los caracteres. Las estructuras sobre las que infiere son líneas, texto, rectángulos o cajas de caracteres e imágenes (Shinyama, Y et al. 2019) [9].
- ***Pdf*Plumber**: es una librería que actúa como capa de abstracción de *pdfminer.six*, con el objetivo de recrear estructuras de documentos más complejas como tablas o textos sobre las ya encontradas en *pdfminer* (Singer-Vine, J. et al. 2020) [23]. Así mismo usa un algoritmo similar a la librería tabula (Ariga, A. 2019) [10] otra librería enfocada a la extracción de tablas.
- **Beautiful Soup:** es una biblioteca de *Python* para extraer datos de archivos HTML y XML. Funciona con varios analizadores diferentes para proporcionar formas idiomáticas de navegar, buscar y modificar el árbol de etiquetas (Richardson, L. 2004-2020) [38].

3.4.1 Particularidades con documentos *Pdf*

Los documentos *pdf* tienen una particularidad con respecto a otros formatos, se trata de un tipo de formato compuesto, es decir, que admite texto, mapas de bits, imágenes vectoriales, contenido multimedia, etc. A efectos prácticos es un conjunto gráfico de distintos elementos con la idea de ser independiente al sistema operativo (Peréz-Porto, J. & Gardey, A. 2019) [11].

Debido a su naturaleza es complicado modificar y crear documentos *pdf* desde código de una manera cómoda, por ello se ha trabajado sobre una aplicación que modifica

archivos *pdf* mediante expresiones regulares, *pdf-redactor*. Creada por (Tauber, J. 2019) [12], y originariamente una aplicación de consola, su funcionamiento consiste en modificar metadatos y textos en documentos *pdf* mediante expresiones regulares. Hace uso de la librería *pdfrw* (Maupin, P. et al. 2018) [13] destinada a modificar y crear *pdf*.

3.5 Librerías de procesamiento del lenguaje Natural

3.5.1 SpaCy

SpaCy es una librería *open source* para el procesamiento del lenguaje Natural, centrada en desarrollar código en producción. Está librería está enfocada al procesamiento de grandes cantidades de textos para construir sistemas de extracción de información y comprensión del lenguaje. Su estructura se basa en la tubería de procesamiento, compuesta por una serie de elementos:

- **Tokenizer:** módulo destinado a tokenizar textos, es decir, separar por palabras.
- **Etiquetador**: encargado de etiquetar el tipo de cada palabra: verbos, sustantivo, adjetivos…
- **Parser**: etiquetador de dependencias entre palabras.
- **NER**: etiquetador de entidades, identifica diferentes entidades en el texto.

Si bien algunas de las características de *SpaCy* funcionan de manera independiente, otras requieren la carga de modelos estadísticos, lo que permite a *SpaCy* predecir anotaciones lingüísticas, por ejemplo, si una palabra es un verbo o un sustantivo. *SpaCy* actualmente ofrece modelos estadísticos para una gran variedad de lenguajes, que pueden instalarse como módulos individuales de *Python*, aunque en este proyecto todos los modelos son en español. Por lo general, incluyen los siguientes componentes:

- **Pesos Binarios:** se usan para el analizador de dependencias y el reconocedor de entidades.
- **Entradas léxicas:** para las palabras, como su ortografías y forma.
- **Vectores de palabras**: representación de significado multidimensionales de palabras que le permiten determinar su similitud entre ellas.

En este proyecto se usan modelos centrados en el identificador de entidades para poder reconocer aquellas entidades que representen algún dato personal. Se utiliza un modelo basado en una CNN multitarea para el español llamado *es_core_news_md*, capacitado en el corpus *AnCora* y *WikiNER*. Asigna vectores de token específicos de contexto, etiquetas POS, análisis de dependencia y entidades con nombre. Admite la identificación de entidades como personas, localizaciones, organizaciones y miscelánea. En las siguientes figuras se puede ver el modelo en detalle y su precisión.

SYNTAX ACCURACY	
LAS	87.88
UAS	90.81
POS	97.09

NER ACCURACY	
NER F	89.71
NER P	89.86
NER R	89.57

Figura 3.1: Detalles sobre la precisión del modelo *es_core_news_md*.

Fuente: (Explosion AI) [14].

LANGUAGE	ES Spanish
TYPE	CORE Vocabulary, syntax, entities, vectors
GENRE	NEWS written text (news, media)
SIZE	MD 74 MB
PIPELINE	`tagger`, `parser`, `ner`
VECTORS	534k keys, 20k unique vectors (50 dimensions)
SOURCES	Spanish UD AnCora Corpus </> WikiNER
AUTHOR	Explosion
LICENSE	GPL

Figura 3.2: Detalles sobre el modelo *es_core_news_md*.

Fuente: (Explosion AI) [14].

3.5.2 NLTK

NLTK es una de las librerías de procesamiento del lenguaje natural más utilizadas para *Python*. Aunque está pensada para la enseñanza, debido a su gran número de herramientas y recursos se puede encontrar en varias aplicaciones y en otras librerías.

Proporciona interfaces fáciles de usar con un gran número de recursos léxicos y corpus textuales como *WordNet*, junto con un conjunto de bibliotecas de procesamiento

de texto para clasificación, tokenización, derivación, etiquetado, análisis, razonamiento semántico y envoltorios para bibliotecas NLP de potencia industrial.

3.5.3 Standford CoreNLP

Stanford CoreNLP proporciona un conjunto de herramientas aplicadas al procesamiento del lenguaje natural. Integra muchas de las herramientas de NLP de Stanford, incluido el etiquetador de discurso (POS) , el reconocedor de entidad nombradas (NER) , el sistema de resolución de correferencia , el análisis de sentimientos , el aprendizaje de patrones de arranque y la extracción de información. Este hecha en Java y está diseñada para ser una librería propia de ese lenguaje, sin embargo, existe un *wrapping* para que desde *Python* se puedan utilizar sus modelos.

3.6 SQlite

Se trata de un motor de bases de datos relacionales pequeño, ligero, autónomo y de alta confiabilidad. Muy útil para manejar bases de datos con pocas tablas y si se quiere utilizar un enfoque de consulta SQL. Actualmente *Python* ya dispone de una librería para conectar con este tipo de base de datos de forma rápida y cómoda.

3.7 Base de datos de nombres

Para este proyecto se ha utilizado una base de datos completa de nombres y apellidos tanto de hombres y mujeres sacada del Instituto Nacional de Estadística [20] con datos del 2018. Además, se ha pasado toda la información al motor *SQlite* para aprovechar el lenguaje de consulta y su estructura relacional para agrupar en dos tablas, nombre y apellidos, la información a tratar. Estos datos se encuentran ordenados según la frecuencia de uso de los nombre y apellidos, de esa forma los nombres más comunes aparecerán los primeros frente a los más extraños que estarán organizados al final de la tabla. Cabe decir que por optimización ambas tablas se encuentran indexadas para conseguir mejores tiempos en las consultas.

3.8 Flask

Es un *framework* para *Python* destinado a crear servicios web aprovechándose del lenguaje de programación. Se trata de un envoltorio para *Werkzeug* y *Jinja*, dos librerías para el diseño web y de plantillas. En este proyecto se ha utilizado una extensión de *Flask* llamada *Flask-restplus* destinada a construir API *REST* de manera sencilla y rápida bajo un marco de buenas prácticas.

3.9 Git

Git es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente . Su propósito es llevar registro de los cambios en archivos fuentes de código y coordinar el trabajo que varias personas realizan sobre archivos compartidos. Se ha utilizado *Git* y *GitHub* como plataforma de control de versiones del software producido en este proyecto.

3.10 Trello

El concepto fundamental para sacar un proyecto adelante es la organización, en el caso del software no podría ser de otra forma. *Trello* es un sistema de administración que actúa como interfaz web enfocado en la creación y la administración de tarjetas, listas y tableros que permiten organizar proyectos de manera flexible, en este TFG ha sido especialmente útil para estructurar las distintas tareas y aclarar ideas en los meses de desarrollo.

Capítulo 4 Diseño e Implementación

4.1 Extracción y modificación de información en Documentos

En esta sección, se van a plantear una serie de técnicas de extracción y modificación de textos que varían según el tipo de documento que queramos tratar, las cuales serán las más adecuadas posibles teniendo en cuenta que estamos ante un sistema cuyo comportamiento debe ser en tiempo real. Asimismo, se ha pensado todos estos algoritmos desde una perspectiva de ingeniería del software, cumpliendo con los principios de diseño y las metodologías ágiles.

Durante los siguientes apartados se hablará sobre las diferentes formas de tratar la información en cada tipo de documento, sin embargo, para todos ellos, excepto para las hojas de cálculo, se ha optado por una estrategia común a la hora de modificar las entidades que el modelo ha extraído, y es tratar estas entidades como una expresión regular a buscar y modificar en los documentos. Este enfoque tiene un ligero impacto en el tiempo de ejecución, no obstante, con ello se va a obtener una mejor eficiencia a la hora de reducir el riesgo de identificación de los datos, debido a que puede haber nombres y apellidos repetidos en el texto que con las técnicas utilizadas se localizan un número menor de veces de las que aparecen realmente, aunque no ocurre de manera regular es una situación a tener en cuenta, de esto se hablará más en el siguiente capítulo.

4.1.1 Diseño y clases

El problema que tenemos en esta parte del servicio es que debemos disponer de un sistema que sea capaz de estructurar un documento decidiendo que partes tienen un contexto asociado y que partes no poseen uno para aplicarles las técnicas de detección y procesamiento más adecuadas, con el objetivo de extraer los datos personales o de modificar dichos datos para reducir el riesgo de identificación y de obtener un documento idéntico al entrante, pero con esas entidades transformadas. En la figura 4.1 se puede ver la arquitectura del sistema planteado, en el diagrama se tienen en cuenta los detalles vistos en la sección anterior.

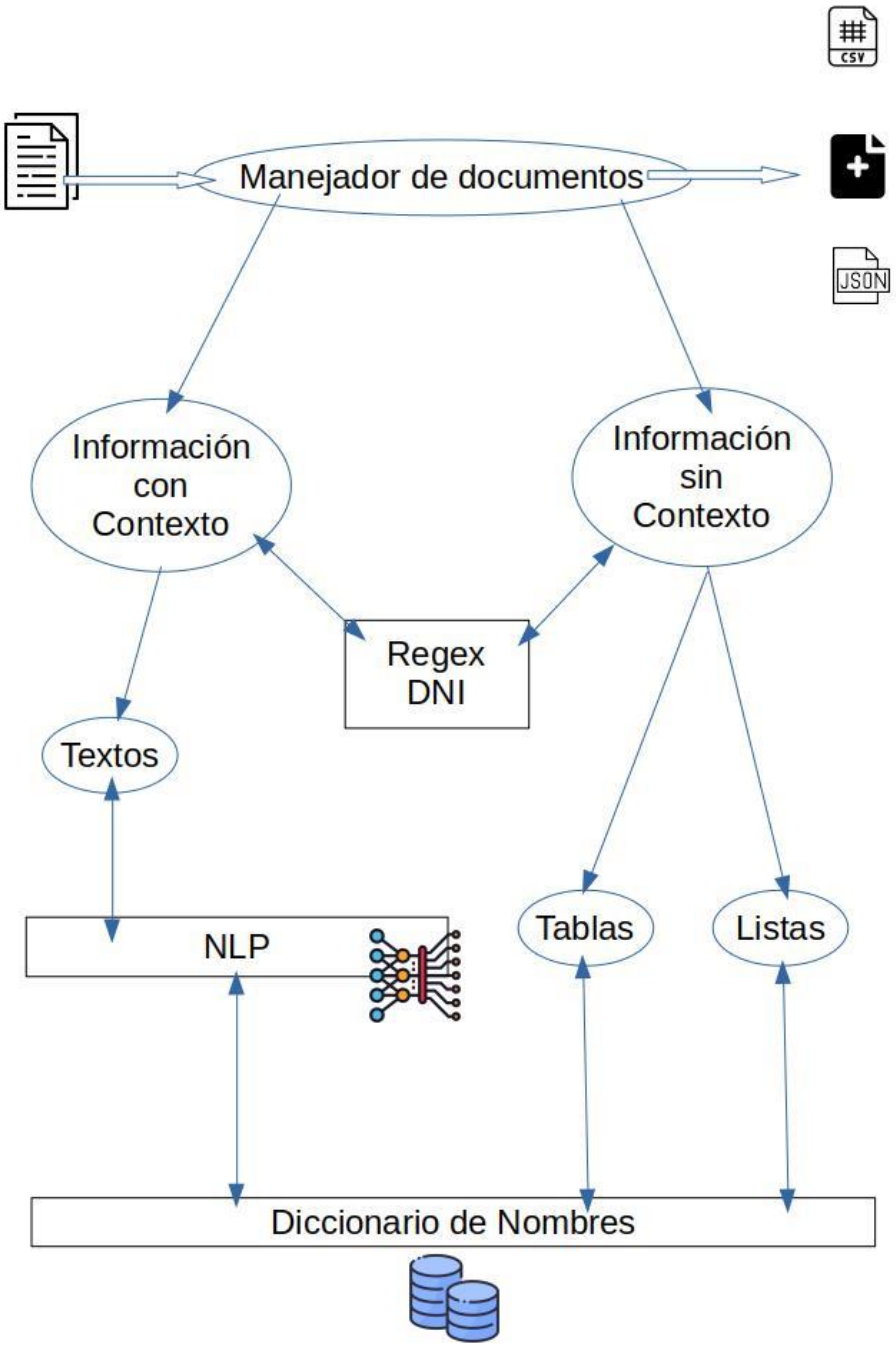


Figura 4.1: Estructura del buscador de datos personales.

Para tratar con diferentes documentos se ha planteado el siguiente esquema de clases:

- Por un lado, necesitamos una clase que se ha capaz de manejar los documentos para realizar las dos operaciones básicas de extracción y modificación. De esta tarea se encarga la clase *DocumentHandler* que a partir de ella se va a establecer una jerarquía, donde por cada tipo de documento que el servicio pueda procesar existirá un manejador con sus métodos adecuados para el tipo de documento
- Debería existir una clase encargada de la búsqueda de entidades, la cual reciba un texto y devuelva una lista con los datos personales encontrados, de esto se encarga la clase *PersonalDataSearch* que establece una jerarquía según el método de extracción de entidades que se utilice: por técnicas de aprendizaje o por reglas, a su vez filtra los resultados con los datos del instituto nacional de estadística mediante la clase *SpanishNamesDB*. Todas las estrategias algorítmicas de localización de entidades que se explicarán en el apartado 4.2 están agrupadas en esta clase y en su jerarquía.
- Para poder seleccionar el tipo de manejador más adecuado se ha utilizado el patrón de diseño *Factory Method*, por lo que se ha creado una clase llamada *CreatorDocumentHandler* que sustituye la creación de manejadores por un método llamado *create*, de esta forma se desacopla el código y la lógica de selección se reduce a observar el tipo de documento y usar el método *create* más adecuado.
- Por último, se necesita de una clase que cargue los modelos e implemente una serie de operaciones útiles para el resto del sistema que tengan relación con el procesamiento de texto y sea cercana a los modelos usados, esta clase se llama *LenguageBuilder*. Se utiliza el patrón *singleton* para hacer que solo se tengan que cargar los modelos una única vez cuando se levanta el servidor. Asimismo, está diseñada de tal forma que, si se quiere cambiar los modelos usados, toda la lógica acoplada a esta clase está dentro de ella por ello su acoplamiento es mínimo, garantizando un mantenimiento más cómodo.

Aún quedan un par de clases que explicar relacionados con los manejadores de documentos, pero su explicación se expondrá más adelante en esta memoria en sus apartados correspondientes. En el anexo III se encuentra un diagrama UML de toda la aplicación.

4.1.2 Documentos en Texto Plano

La mayoría de los documentos con los que va a tratar el servicio están en formato *pdf* o *Word*, sin embargo, los documentos en texto plano son tan comunes y fácilmente tratables a nivel de programación que incluirlos en el conjunto de formatos posibles para este software le añade valor a un coste muy bajo, además durante todo el desarrollo este formato nos ha servido para testear y probar las estrategias usadas. Se ha elegido trabajar sobre documento *.txt* por son los ficheros más usuales en sistemas Windows y también se utilizan en la gran mayoría de las versiones de Unix.

En las figuras 4.2 y 4.3 se pueden ver los métodos de la clase *DocumentHandlerTxt* que hereda de la clase abstracta *DocumentHandler*, y es la

encargada de extraer las entidades de los documentos de texto plano y de modificarlos. *Python* ya dispone de formas para leer y modificar estos documentos ya sea por líneas o procesando todo el texto mediante la función *open*, como no es usual que estos documentos presenten algún tipo de dato estructurado se presupone que toda la información que se encuentra en ellos es simplemente texto, por lo que se puede leer por líneas los ficheros para localizar las entidades. El método *extractData* es el encargado de extraer las entidades (Nombres y Documentos de identidad) y devolver una tupla con estos datos. Hemos optado por procesa un texto plano línea por línea porque se necesita saber si una línea tiene un contexto asociado o si es tan solo una expresión simple para aplicar las técnicas de localización de entidades más adecuadas. El método *documentsProcessing* se encarga de utilizar las entidades encontradas para crear un nuevo fichero de texto plano con la información actualizada. Se modifica la lista de nombres para que no haya repetidos y se ordena en orden descendente utilizando la longitud de los caracteres del nombre, para que si hubiera un nombre más largo formado por otros más pequeños estos no modificasen todas sus ocurrencias en el texto, distorsionando nombres similares con mayor longitud. Un ejemplo, sería que si en un texto se encontrará el nombre “Kevin” y otro nombre formado por el mismo: “Kevin Zarco”; en este caso si no controlásemos el orden y se procesase “Kevin” antes, afectaría a todas las ocurrencias y no se podría detectar la expresión “Kevin Zarco”. Finalmente, se procesa todo el documento, agrupando en una expresión regular las entidades utilizando el token ‘|’ que denota la operación *or*.

Como *Python* tiene una longitud para las expresiones regulares limitada, se escogen como máximo un total de 4000 entidades para usar en una sola expresión regular, posteriormente se obtendrán otras 4000, así sucesivamente hasta procesar todas las entidades. Se ha elegido este valor debido a que la longitud de los carnés es siempre igual y en el caso de los nombres como máximo puede está formado por 5 o 6 palabras en nuestra lengua, aunque esto no es muy común, de todas formas, es muy probable que se pueda utilizar una cantidad mucho mayor, pero se trata de un valor empírico con el que hemos logrado buenos resultados sin encontrar problemas en el rendimiento.

```
def documentsProcessing(self):
    def updateTxt(regex:str, text:Text) -> Text:
        regexName = re.compile(regex)
        return regexName.sub(lambda match: self.anonymizationFunction(match.group()), text)

    if not self.anonymizationFunction:
        return

    with open(self.path, 'r', encoding='utf8') as file, open(self.outfile, 'w',encoding='utf8') as outfile:
        data              = []
        fileText          = str(file.read())
        maxLength         = MAXIMUM_NUMBER_OF_ELEMENTS_IN_A_REGEX
        listNames,idCards = self.extractData()
        listNames = list(set(listNames))
        listNames.sort(
                key     = lambda value: len(value),
                reverse = True
            )
        data[len(data):],data[len(data):] = listNames,idCards
        if len(data) > maxLength:
            intial = 0
            for numberRange in range(maxLength,len(data),maxLength):
                fileText   = updateTxt('|'.join(data[intial:numberRange]), fileText)
                intial     = numberRange
            fileText = updateTxt('|'.join(data[intial:]), fileText)
        else:
            fileText = updateTxt('|'.join(data), fileText)
        outfile.write(fileText)
```

Figura 4.2: Método para modificar texto plano.

```
def extractData(self) -> tuple:
    listNames = []
    idCards   = []
    with open(self.path, 'r',encoding='utf8') as file:
        for line in file:
            line = line[0:len(line)-1]
            if LanguageBuilder().hasContex(line):
                listNames[len(listNames):],idCards[len(idCards):] = self.dataSearch.searchPersonalData(line)
            elif self.dataSearch.isName(line):
                listNames.append(line)
    return listNames,idCards
```

Figura 4.3: Método para extraer entidades en texto plano.

4.1.3 Documentos en Word

Para trabajar con documentos *Word*, la clase *DocumentHandlerDocx* usa la librería *Python-docx*, esta librería solo puede trabajar con documentos con formato *.docx*, lo cual es muy útil porque son el tipo de formato en el que nos centramos en este TFG y una gran mayoría de documentos oficiales y administrativos están en este formato antes de pasarse a *pdf*.

Todos los ficheros *Word* que se procesen bajo esta librería, serán tratados como un árbol de objetos, estos pueden ser: imágenes, tablas, filas, celdas, párrafos y texto, cada uno tiene una serie de propiedades: la fuente de letra, su tamaño, el estilo, etc. Con esto se puede plantear la lectura de un documento *Word* como un proceso de iteración entre los elementos del árbol, descartando aquellos que no nos sean útiles como las imágenes.

En la figura 4.4 se muestra el código necesario para recorrer el árbol, se ha optado por utilizar la instrucción *yield* para transformar la función que recorre el árbol en un generador, haciendo que cuando un nodo sea un párrafo o una tabla la función envíe el objeto y reanude la ejecución una vez que el método principal termine de procesarlo. Como no hay muchos objetos distintos posibles en un documento y estos no están compuestos por muchos a su vez, para cualquier documento siempre se va crear un árbol que crece mucho en anchura pero que es poco profundo, a su vez como solo se explora el primer nivel del árbol para buscar párrafos o tablas su complejidad tanto espacial como temporal serían $O(b)$ siendo b el factor de ramificación, por lo que estamos hablando que el recorrido del árbol es en tiempo polinómico para cualquier documento.

```
def itemIterator(parent):
    if isinstance(parent, _Document):
        parent_elm = parent.element.body
    elif isinstance(parent, _Cell):
        parent_elm = parent._tc
    elif isinstance(parent, _Row):
        parent_elm = parent._tr
    else:
        raise ValueError("the Docx document have object what didn't implemented in the iterator")
    for child in parent_elm.iterchildren():
        if isinstance(child, CT_P):
            yield Paragraph(child, parent)
        elif isinstance(child, CT_Tbl):
            yield Table(child, parent)
```

Figura 4.4: Función para recorrer un documento .docx.

El método *extractData* es el encargado de procesar los documentos y de mediante la clase *dataSearch* de encontrar y enviar las entidades nombradas. Para ello en el caso de que estemos ante un párrafo, se busca las entidades en todo el texto que este contiene, mediante el atributo *text*, si fuera una tabla, como la única manera posible de explorarla en este formato es mediante filas, se usa una clase auxiliar llamada *DataPickerInTables* que procesa las cabeceras de la tabla y va almacenando las celdas correspondientes a las cabeceras que puedan almacenar nombres. Una vez terminada la exploración de todas las celdas se selecciona aquellas que correspondan a un nombre mediante la base de conocimiento y se buscan carnés de identidad en las restantes. Terminado el procedimiento tendríamos dos listas por cada entidad que se enviarán en una tupla. Esta estrategia de utilizar una clase auxiliar para recorrer una tabla por filas se utiliza también para procesar tablas en documentos *pdf* y en documentos *html*. El método *documentsProcessing* utiliza expresiones regulares, al igual que en los ficheros de texto plano, para modificar la información textual en el documento: se itera en el árbol y se modifica el contenido de los objetos mediante el atributo *text* de los párrafos o las celdas y se almacena en el destino correspondiente, de esa forma tenemos un fichero exactamente idéntico al original, pero con las entidades anonimizadas, ofuscadas o disgregadas. Las siguientes figuras son las implementaciones de los algoritmos

explicados, se ha hecho uso de funciones *filter* y *map* para lograr obtener un mejor rendimiento evitando, en la medida de lo posible, bucles en *Python*.

```
def extractData(self) -> tuple:
    lastKey    = []
    listNames  = []
    listIdCard = []
    for block in itemIterator(self.document):
        if isinstance(block, Paragraph):
            if LanguageBuilder().hasContex(block.text):
                listNames[len(listNames):],listIdCard[len(listIdCard):] = self.dataSearch.searchPersonalData(block.text)
            elif self.dataSearch.isName(block.text):
                listNames.append(block.text.strip())
            else:
                _,listIdCard[len(listIdCard):] = self.dataSearch.searchPersonalData(block.text)
        elif isinstance(block, Table):
            namePicker = DataPickerInTables()
            for index, row in enumerate(block.rows):
                rowText = [cell.text for cell in row.cells]
                if index == 0:
                    lables = list(
                        filter(
                            lambda cell: list(
                            filter(
                                lambda x: LanguageBuilder().semanticSimilarity(cell,x) > MEASURE_TO_COLUMN_KEY_REFERS_TO_NAMES,listOfVectorWords))
                                , rowText
                            )
                        )
                    key = list(map(lambda cell: rowText.index(cell),lables))
                    if not key:
                        key = lastKey
                        namePicker.addIndexesColumn(key)
                    else:
                        lastKey = key
                        namePicker.addIndexesColumn(key)
                        continue

                nameRow = list(filter(lambda cell: namePicker.isColumnName(rowText.index(cell)), rowText))
                for cell in nameRow:
                    namePicker.addName(rowText.index(cell), cell)

                listIdCard[len(listIdCard):] = list(
                    itertools.chain.from_iterable(
                        map(lambda cell: self.dataSearch.giveIdCards(cell),
                        filter(lambda cell: cell not in nameRow, rowText))
                    )
                )

            listNames[len(listNames):] = namePicker.getAllNames(self.dataSearch.checkNamesInDB,MEASURE_FOR_TEXTS_WITHOUT_CONTEXTS)
        else:
            continue
    return listNames,listIdCard
```

Figura 4.5: Algoritmo para extraer entidades en un documento Word.

```
def documentsProcessing(self):
    def updateDocx(regex:str, text:Text) -> Text:
        regexName = re.compile(regex)
        return regexName.sub(lambda match: self.anonymizationFunction(match.group()), text)

    if not self.anonymizationFunction:
        return

    data              = []
    maxLength         = MAXIMUM_NUMBER_OF_ELEMENTS_IN_A_REGEX
    listNames,idCards = self.extractData()
    listNames         = list(set(listNames))
    listNames.sort(
            key     = lambda value: len(value),
            reverse = True
        )
    data[len(data):],data[len(data):] = listNames,idCards
    theExpressionShouldBeTrodden      = True

    if len(data) <= maxLength:
        theExpressionShouldBeTrodden = False
        regex     = '|'.join(data)
        regexName = re.compile(regex)

    for block in itemIterator(self.document):
        if isinstance(block, Paragraph):
            if theExpressionShouldBeTrodden:
                intial = 0
                for numberRange in range(maxLength,len(data),maxLength):
                    block.text = updateDocx('|'.join(data[intial:numberRange]), block.text)
                    intial     = numberRange
                block.text = updateDocx('|'.join(data[intial:]), block.text)
            else:
                block.text = regexName.sub(lambda match: self.anonymizationFunction(match.group()), block.text)
        elif isinstance(block, Table):
            for row in block.rows:
                for cell in row.cells:
                    if theExpressionShouldBeTrodden:
                        intial = 0
                        for numberRange in range(maxLength,len(data),maxLength):
                            cell.text = updateDocx('|'.join(data[intial:numberRange]), cell.text)
                            intial = numberRange
                        cell.text = updateDocx('|'.join(data[intial:]), cell.text)
                    else:
                        cell.text = regexName.sub(lambda match: self.anonymizationFunction(match.group()), cell.text)

        else:
            continue
    self.document.save(self.outfile)
```

Figura 4.6: Algoritmo para modificar documentos en *Word*.

En un documento *Word* o en *pdf* una tabla puede estar dividida en varias páginas si su tamaño es demasiado grande, como para estos dos formatos la estrategia de lectura es ir página a página, una misma tabla será detectada como dos objetos diferentes. Para poder localizar esta situación se valora si en encabezado de una tabla alguna columna pueda almacenar nombres, si no es así y en la tabla inmediatamente anterior sí hubo, se escoge las columnas bajo el rango de coincidencia de la tabla anterior. Con esta estrategia heurística se consigue paliar este evento, lo único que hace falta es almacenar el índice de las posibles columnas en cada iteración para que la siguiente tabla pueda tener este método de respaldo, bajando así el número de falsos negativos.

4.1.4 Documentos en pdf

Para trabajar sobre *pdf* y poder obtener la información textual en estos documentos se utiliza la librería *pdfPlumber,* este conjunto de herramientas para *Python* permite leer

documentos con formato *pdf* por páginas y de obtener la información de diferentes maneras. Está construida sobre *pdfminer* y *pdfminer.six,* dos librerías que usan métodos heurísticos para poder extraer datos a bajo nivel centrándose únicamente en textos, lo que ofrece *pdfPlumber* es una capa de abstracción que permite manejar estructuras difíciles de localizar como son las tablas, permitiendo encontrar párrafos y tablas en cada página del documento. El enfoque de la detección de tablas se basa en gran medida en la tesis de maestría de (Nurminen, A. 2013) [39] y está inspirado en (Yildiz, B. et al. 2005) [27], funciona de la siguiente manera:

1. Para cualquier página *pdf* dada, se buscan las líneas que están (a) explícitamente definidas y / o (b) implícitas en la alineación de palabras en la página.
2. Se fusionan las líneas superpuestas o casi superpuestas.
3. Se encuentran las intersecciones de todas esas líneas.
4. Se encuentra el conjunto de rectángulos más granular (es decir, celdas) que usan estas intersecciones como vértices.
5. Se agrupan celdas contiguas en tablas.

Esta estrategia sirve para buscar tablas pero esta librería es un poco tosca a la hora de localizar textos, esto se traduce en devolver todo lo que considera texto incurriendo en información redundante entre las tablas y los párrafos, para quitar estas ocurrencias se utilizan las filas de las tablas como expresiones regulares, como estas siempre están acompañadas de espacios en blancos, tabulaciones, o caracteres extraños que se caracterizan por que separan cada celda, construir una expresión regular para borrarlas es sencillo con un riesgo de pérdida de información mínimo. En la figura 4.7 está el código para iterar entre las páginas de un documento y obtener la información deseada, se ha utilizado un generador por las mismas razones que en los documentos *docx*.

El siguiente paso es bastante similar a los ficheros *.docx*, buscar las entidades en tablas y textos con los algoritmos explicados y obtener una tupla con el conjunto de las entidades nombradas. A la hora de modificar estos documentos se utiliza una aplicación en *Python* creado por (Tauber, J. 2019) [12], que se ha alterado para pasar de ser una aplicación de consola a un conjunto de clases útiles en este proyecto, se llama *pdf-redactor*. El uso de la aplicación es muy sencillo, mediante expresiones regulares y funciones que indican las transformaciones que van a sufrir las ocurrencias encontradas se puede alterar el contenido de un *pdf* escribiendo el resultado en otro, usa la librería *pdfrw* que permite leer *pdf* y alterarlos a muy bajo nivel, el algoritmo que sigue a rasgos generales es el siguiente:

1. Serializa el contenido del *pdf*, tanto los datos como los metadatos
2. Transforma todo el contenido en una serie de tokens, estos tokens pueden ser tanto palabras como caracteres y los agrupa en un diccionario.
3. Aplica las expresiones regulares y las funciones como una serie de filtros
4. Vuelca el contenido serializado y modificado en un nuevo documento *pdf*.

Este algoritmo se encuentra optimizado por lo que para una gran mayoría de documentos se obtendrá un buen rendimiento independientemente del tamaño de este.

Alterar archivos *pdf* no es tarea fácil, puede llegar a ser muy complejo sobre todo en *Python*, aunque disponga de buenas librerías para leerlos no son tan buenas para modificar el contenido de estos, por ello y porque el problema que se plantea en este proyecto está orientado a tratar con una gran variedad de documentos, se ha decidido reutilizar software *open source* para facilitarnos el trabajo y obtener mejores resultados. Si se quiere hacer un estudio más detallado el código se encuentra en los ficheros fuentes del proyecto, está debidamente comentado explicando cada una de sus partes y algoritmos usados, asimismo existe un repositorio en GitHub perteneciente a (Tauber, J. 2019) [12]. En la figura 4.8 está el código utilizado.

```python
def readPdf(path:str) -> Text:
    try:
        with pdfplumber.open(path) as pdf:
            for page in pdf.pages:
                text   = page.extract_text()
                tables = []
                for pdf_table in page.extract_tables():
                    table = []
                    cells = []
                    for row in pdf_table:
                        if not any(row):
                            if any(cells):
                                table.append(cells)
                                cells = []
                        elif all(row):
                            if any(cells):
                                table.append(cells)
                                cells = []
                            table.append(row)
                        else:
                            if len(cells) == 0:
                                cells = row
                            else:
                                for i in range(len(row)):
                                    if row[i] is not None:
                                        cells[i] = row[i] if cells[i] is None else cells[i] + row[i]
                    for regex in map(lambda row: ".*" + ".*".join(row) + ".*", table):
                        text = re.sub(regex,'',text)
                    tables.append(table)
                yield (text, tables)
    except Exception:
        return
```

Figura 4.7: Código para la extracción de información en documentos *pdf*.

```
def documentsProcessing(self):
    def updatePdf(regex:str):
        self.options.content_filters = [
            (
                re.compile(regex),
                lambda m: self.anonymizationFunction(m.group())
            )
        ]
        pdf_redactor.redactor(self.options, self.path, self.outfile)
        self.path = self.outfile

    if not self.anonymizationFunction:
        return

    maxLength = MAXIMUM_NUMBER_OF_ELEMENTS_IN_A_REGEX
    listNames,idCards = self.extractData()
    if not listNames and not idCards:
        pdf_redactor.redactor(self.options, self.path, self.outfile)
        return
    listNames = list(set(listNames))
    listNames.sort(
            key     = lambda value: len(value),
            reverse = True
        )
    data                              = []
    data[len(data):],data[len(data):] = listNames,idCards
    if len(data) > maxLength:
        intial = 0
        for numberRange in range(maxLength,len(data),maxLength):
            updatePdf('|'.join(data[intial:numberRange]))
            intial = numberRange
        updatePdf('|'.join(data[intial:]))
    else:
        updatePdf('|'.join(data))
```

Figura 4.8: Código para modificar documentos *pdf*.

4.1.5 Documento en HTML

Gran parte del texto en la web está en forma de documentos HTML, este formato se caracteriza por ser un lenguaje de etiquetas o de marcado para diseñar páginas web, el conjunto de estas etiquetas representa la estructura de la página web que tendrá que representar el navegador (W3C 2020) [37]. En este apartado se van a explorar las técnicas de extracción de información en páginas HTML (Web scraping) para extraer textos y reorganizar estos para poder localizar datos estructurados.

Para aplicar técnicas web scraping se utiliza la librería *Beautiful Soup,* con ella se pueden extraer datos de archivos XML o HTML usando diferentes analizadores, en nuestro caso hemos elegido utilizar *lxml* porque permite analizar con eficiencia los archivos HTML estando estos bien formados, cumpliendo con el estándar, o no. *Beautiful Soup* analiza todo el contenido del fichero y construye un árbol en *Python* donde cada nodo representa una etiqueta con una serie de propiedades, permite a su vez navegar por él y realizar diferentes operaciones de filtrado como extraer únicamente las etiquetas que contengan texto, con ello podemos podar parte del árbol conservando aquellos nodos que nos sean útiles. De ese modo la estrategia para extraer datos será la siguiente:

1. Procesar el fichero HTML con la librería y el analizador, generando el árbol de etiquetas.
2. Filtrar aquellos nodos que posean texto.
3. Averiguar las etiquetas padres de cada uno para saber si tenemos alguna etiqueta *th* o *td* y por tanto una tabla.
4. Procesar la información obtenida.

Para cumplir con el paso 3 se tiene que buscar en todos los nodos padres de una etiqueta, debido a que el subárbol dentro de una etiqueta *th* o *td* puede contener otros nodos antes de llegar al texto, el procedimiento que se ha construido pasa por evaluar para cada nodo si en el camino de él hasta la raíz existe alguna de estas dos etiquetas, si es así las devuelve, si no, devuelve únicamente el padre, con nodo raíz nos referimos a la etiqueta *[document]*. Este método únicamente localiza tablas que estén basadas en los estándares HTML, es decir, usando etiquetas *th* para las cabeceras y *td* para las filas. Asimismo, no todos los nodos que contengan textos nos pueden servir, dentro todo el conjunto de etiquetas HTML existe un subconjunto que no nos aportan nada útil para buscar entidades y por tanto no se van a procesar, como son: *'noscript', 'header','style', 'html', 'meta', 'head', 'input', 'script', 'link','lang', 'code'*…

La clase *TokenizerHtml* de la figura 4.9 recoge el algoritmo anterior y lo implementa transformando los nodos del árbol en tokens, cada token será una unidad formada por una lista de textos representando las filas de una tabla, junto a una variable que indique si estamos ante una cabecera de una tabla, una fila o ante ninguna de ellas, siendo la lista de textos de un único elemento en el último caso. Devolverá en cada iteración a la clase *DocumentHandlerHtml* un token que se procesará según las técnicas vistas para *pdf* o *Word* usando un objeto auxiliar para procesar las filas de las columnas y hacer consultas al diccionario, en caso de no tratarse de una tabla se utilizará el modelo. En la figura 4.10 se puede apreciar tanto la implementación de esta clase como el método *extractData* que recoge esta estrategia.

Para modificar documentos HTML *Beautiful Soup* dispone de un formateador, un procedimiento que permite pasarle una función para modificar el texto que contienen las etiquetas, siendo este el caso, la implementación transformando las entidades en expresiones regulares se convierte en un problema trivial, está recogido en el método *documentsProcessing* de la figura 4.10.

```python
class TokenHtml:
        def __init__(self,listOfText:list,isTable:TableToken):
            self.text    = listOfText
            self.isTable = isTable

class TokenizerHtml:
    def __init__(self, soup:BeautifulSoup):
        self.soup = soup
        self.blacklist = [
                    '[document]', 'noscript', 'header','style',
                    'html', 'meta', 'head', 'input', 'script', 'link',
                    'lang', 'code','th', 'td'
        ]

    def getToken(self) -> TokenHtml:
        def giveParentName(label) -> str:
            auxLable = label
            while auxLable.parent.name != '[document]':
                if auxLable.parent.name in ['th','td']:
                    return auxLable.parent.name
                auxLable = auxLable.parent
            return label.parent.name

        labelList  = list(filter(lambda label: label and label != "\n", self.soup.find_all(text=True)))
        tags       = list(map(lambda label: giveParentName(label),labelList))
        headList   = []
        rowList    = []
        lenHead    = 0
        for index,htmlLabel in enumerate(zip(labelList,tags)):
            label,tag = htmlLabel
            if tag not in self.blacklist:
                lenHead = 0
                headList.clear()
                rowList.clear()
                yield TokenHtml([str(label)],TableToken.NONE)

            elif tag == 'th':
                headList.append(str(label))
                lenHead = len(headList)
                try:
                    if tags[index+1] == 'th':
                        continue
                    aux = headList[:]
                    headList.clear()
                    yield TokenHtml(aux,TableToken.HEAD)
                except IndexError:
                    aux = headList[:]
                    headList.clear()
                    yield TokenHtml(aux,TableToken.HEAD)

            elif tag == 'td':
                rowList.append(str(label))
                try:
                    if tags[index+1] == 'td' and len(rowList) < lenHead:
                        continue
                    aux = rowList[:]
                    rowList.clear()
                    yield TokenHtml(aux,TableToken.ROW)
                except IndexError:
                    aux = rowList[:]
                    rowList.clear()
                    yield TokenHtml(aux,TableToken.ROW)
```

Figura 4.9: Implementación de la clase TokenizerHtml.

```python
def documentsProcessing(self):

    if not self.anonymizationFunction:
        return

    formatter = HTMLFormatter(self._processEntities)
    listNames,idCards = self.extractData()
    listNames = list(set(listNames))
    listNames.sort(
            key     = lambda value: len(value),
            reverse = True
        )
    data = []
    data[len(data):],data[len(data):] = listNames,idCards
    self._buildRegex(data)
    with open(self.outfile, "w") as f:
        f.write(self.soup.prettify(formatter=formatter))

def extractData(self) -> tuple:
    def cleanPicker():
        if not picker.isEmpty():
            listNames[len(listNames):] = picker.getAllNames(self.dataSearch.checkNamesInDB,MEASURE_FOR_TEXTS_WITHOUT_CONTEXTS)
            picker.clear()

    listNames = []
    idCards   = []
    picker    = DataPickerInTables()
    tokenizer = TokenizerHtml(self.soup)
    for token in tokenizer.getToken():
        if token.isTable == TableToken.NONE:
            if LanguageBuilder().hasContex(token.text[0]):
                listNames[len(listNames):],idCards[len(idCards):]  = self.dataSearch.searchPersonalData(token.text[0])
            elif self.dataSearch.isName(token.text[0]):
                listNames.append(token.text[0])
            cleanPicker()
        elif token.isTable == TableToken.HEAD:
            cleanPicker()
            keys = list(filter(lambda text: list(
                    filter(lambda x:LanguageBuilder().semanticSimilarity(text,x) > MEASURE_TO_COLUMN_KEY_REFERS_TO_NAMES,
                    listOfVectorWords)), token.text))
            if keys:
                for key in keys:
                    picker.addIndexColumn(token.text.index(key))
        elif token.isTable == TableToken.ROW:
            for index in picker.getIndexesColumn():
                try:
                    picker.addName(index,token.text[index])
                except IndexError:
                    continue
            idCards[len(idCards):] = list(
                itertools.chain.from_iterable(
                    map(
                        lambda id: self.dataSearch.giveIdCards(id),
                        [text for index,text in enumerate(token.text) if not index in picker.getIndexesColumn()]
                    )
                )
            )
    return listNames,idCards
```

Figura 4.10: Métodos de extracción y modificación de datos en HTML.

4.1.6 Hojas de cálculo

Las hojas de cálculo son diferentes a los demás documentos, como únicamente son tablas el proceso de lectura es mucho más simple y se puede realizar únicamente con la librería *pandas*, está librería permite leer, modificar y guardar diferentes hojas de cálculo en los formatos más usuales: hojas de *Excel* y *csv*. Teniendo esto en mente se crea una clase por cada formato posible, en este caso *DocumentHandlerExcel* es la encargada de manejar archivos con extensión *xlsm*, *xlsx* y *xls* que son los más comunes si hablamos de archivos *Excel*, por otro lado, la clase *DocumentHandlerCsv* manejará ficheros *csv* que usen la coma como carácter delimitador (RCF 4180), las dos heredan de *DocumentHandlerSpreadsheets* que tiene el método común *save* para almacenar los documentos, este método utiliza el procedimiento *to_csv* para archivos *csv* y el

ExcelWriter en caso de los *Excel*. Asimismo, el software está orientado a procesar hojas que representen una única tabla que es el formato más común en documentos oficiales.

Como únicamente se tiene que implementar el algoritmo para la búsqueda de entidades en tablas, es el módulo que mejor rendimiento ofrece. A su vez, no necesita de ninguna clase auxiliar pues *pandas* permite iterar entre encabezados y columnas con relativa facilidad, de todas formas, se ha creado la clase *ColumnSelectorDataFrame* que implementa la lógica necesaria para indicar en qué columnas hay que buscar nombres y en cuáles no, como el manejo de columnas en *pandas* es trivial el algoritmo para extraer entidades no es difícil de implementar:

1. Se limpia las columnas de posibles valores nulos, para que no influyan en la selección.
2. Si se trata de una columna de nombres, se juzga si su coincidencia con la base de conocimiento supera a la heurística, mediante las técnicas que se verán en la sección 4.2.2.
3. Si no es una columna de nombres, se procede a buscar carnés de identidad.
4. Se devuelve una tupla con las entidades encontradas.

Para anonimizar este tipo de archivos no se usan expresiones regulares pues no son necesarias, se puede hacer la búsqueda junto a la modificación, además de que en términos de eficiencia como los resultados van a depender únicamente de la base de conocimiento no es necesario utilizar el mismo método que en los otros formatos, por lo que al algoritmo anterior habría que añadirle un último paso y es el de modificar las celdas seleccionadas. La figura 4.11 y 4.12 muestran el código para seleccionar las columnas y la implementación de la clase *DocumentHandlerCsv*, la clase *DocumentHandlerExcel* es similar a excepción de que en un documento *Excel* puede haber más de una hoja, por ello se itera primero por hojas y luego para cada una se procesa el algoritmo.

```
class typeOfColumn:
    def __init__(self, key:str, isName:bool):
        self.key    = key
        self.isName = isName

class ColumnSelectorDataFrame:

    def getPossibleColumnsNames(self, df: pd.DataFrame) -> typeOfColumn:
        for key, typeColumn in zip(df.keys(), df.dtypes):
            if typeColumn == object:
                listOfWordSemantics = list(
                        filter(
                                lambda x: LanguageBuilder().semanticSimilarity(key, x) > MEASURE_TO_COLUMN_KEY_REFERS_TO_NAMES,
                                listOfVectorWords
                            )
                    )
                if listOfWordSemantics:
                    yield typeOfColumn(key,True)
                yield typeOfColumn(key,False)

    def columnSearch(self, df: pd.DataFrame, comparateFuntion: classmethod) -> int:
        return len(comparateFuntion(list(df)))
```

Figura 4.11: Código de la clase ColumnSelectorDataFrame.

```python
class DocumentHandlerCsv(DocumentHandlerSpreadsheets):

    def __init__(self, path: str, outfile: str = "",anonymizationFunction=None):
        super().__init__(path, outfile=outfile, anonymizationFunction=anonymizationFunction)
        self.df = pd.read_csv(path)

    def extractData(self, personalData: PersonalData = PersonalData.all) -> tuple:
        listNames = []
        idCards   = []
        for typeColumn in self.selector.getPossibleColumnsNames(self.df):
            dfNotNull   = self.df[typeColumn.key][self.df[typeColumn.key].notnull()]
            if personalData != PersonalData.idCards and typeColumn.isName:
                auxdf = dfNotNull
                if len(dfNotNull) > MAXIMUM_NUMBER_OF_POSSIBLE_NAMES_FOR_A_QUERY:
                    auxdf = sample(list(dfNotNull),round(len(dfNotNull) * SAMPLE_DATA_TO_CHOOSE_NAMES))
                countOfName = self.selector.columnSearch(auxdf,self.dataSearch.checkNamesInDB)
                if countOfName / len(auxdf) > MEASURE_FOR_TEXTS_WITHOUT_CONTEXTS:
                    listNames[len(listNames):] = dfNotNull
            elif personalData != PersonalData.names:
                idCards[len(idCards):] = list(
                    itertools.chain.from_iterable(map(lambda idCards: self.dataSearch.giveIdCards(idCards),dfNotNull))
                )
        return listNames,idCards

    def documentsProcessing(self, personalData: PersonalData = PersonalData.all):
        if not self.anonymizationFunction:
            return

        for typeColumn in self.selector.getPossibleColumnsNames(self.df):
            dfNotNull   = self.df[typeColumn.key][self.df[typeColumn.key].notnull()]
            if personalData != PersonalData.idCards and typeColumn.isName:
                auxdf = dfNotNull
                if len(dfNotNull) > MAXIMUM_NUMBER_OF_POSSIBLE_NAMES_FOR_A_QUERY:
                    auxdf = sample(list(dfNotNull),round(len(dfNotNull) * SAMPLE_DATA_TO_CHOOSE_NAMES))
                countOfName = self.selector.columnSearch(auxdf,self.dataSearch.checkNamesInDB)
                if countOfName / len(auxdf) > MEASURE_FOR_TEXTS_WITHOUT_CONTEXTS:
                    self.df[typeColumn.key].replace({str(name): self.anonymizationFunction(str(name)) for name in dfNotNull}, inplace=True)
            elif personalData != PersonalData.names:
                idCards = list(
                    itertools.chain.from_iterable(map(lambda idCards: self.dataSearch.giveIdCards(idCards),dfNotNull))
                )
                self.df[typeColumn.key].replace({str(idCard): self.anonymizationFunction(str(idCard)) for idCard in idCards}, inplace=True)
        self.save()


    def save(self):
        self.df.to_csv(self.outfile, index=False)
```

Figura 4.12: Código de la clase DocumentHandlerCsv.

4.2 Técnicas para la detección de entidades

Tanto para la extracción como para la tratar con las entidades lo principal es obtener la información para que el software pueda hacer algún tipo de inferencia, dicha información se puede presentar de dos maneras:

- **No Estructurada:** Hace referencia a aquella información que no dispone de algún esquema para su procesamiento, normalmente se refiere a textos escritos en lenguaje natural.
- **Estructurada:** Existe en los datos algún tipo de organización regular y predecible que nos da información sobre los propios datos, un ejemplo clásico es una tabla.

El problema a la hora de trabajar con datos no estructurados y estructurados es que si se trata de un texto este posee un contexto y una gramática basada en algún idioma, lo cual es útil para buscar expresiones o entidades abstractas como lo son los nombres o apellidos. Este hecho es una ventaja si se dispone de técnicas de procesamiento del lenguaje natural, pero que en datos estructurados no se puede explotar de la misma manera. Por ello en esta sección se plantea los algoritmos usados para buscar nombres, apellidos y DNI, haciendo una división clara según el tipo de información que se puede encontrar en un documento, esta separación está centrada en aquella que no disponga de una estructura, pero sí de un contexto que se pueda explotar, como son las frases y las

oraciones, y aquella que no presenta ningún tipo de contexto, pero sí de una estructura como una tabla.

Para la detección de entidades se ha planteado un enfoque de inteligencia artificial híbrido, basado por un lado en técnicas de aprendizaje que permiten realizar inferencias y clasificaciones sobre textos y por otro en ingeniería del conocimiento haciendo uso de heurísticas simples, reglas sobre el lenguaje y de una base de conocimiento que permita mejorar la fiabilidad de los resultados. Todo esto con el objetivo de obtener un sistema que reduzca los riesgos de identificación asociados a datos personales.

4.2.1 Localización de Entidades y estudio sobre los modelos a utilizar

Como ya se ha comentado en este TFG se ha apostado por el uso de sistemas híbridos para la clasificación y búsqueda de entidades en textos, no solo por el hecho de tener a disposición una base de conocimientos de nombres y apellidos, sino también porque se ha usado diferentes técnicas para la localización de diferentes entidades.

El objetivo de este proyecto es la búsqueda de datos personales centrada únicamente en nombres, apellidos y carnés de identidad (DNI), los primeros destacan por ser unos objetos semánticos abstractos muy difíciles de localizar haciendo uso de metodologías clásicas, sin embargo, los DNI son expresiones formales que siempre tienen una forma concreta de ser escritos, por ello con una simple expresión regular se pueden localizar con eficiencia. En la figura 4.13 se expone la expresión regular que se ha construido para localizar estas expresiones. Se podría haber reutilizado mediante reentrenamiento el mismo modelo neuronal que encuentra nombres y apellidos para localizar los documentos de identidad, pero consideramos que en el campo del NER si existen entidades que se pueden representar mediante reglas y dichas reglas son relativamente sencillas es mejor utilizar esta técnica clásica pues dará mejores resultados a corto plazo de eficiencia y rendimiento.

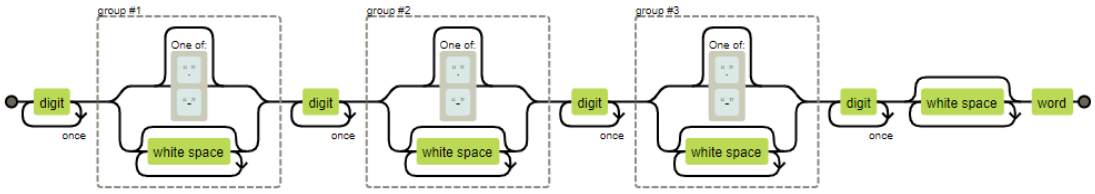


Figura 4.13: Expresión regular para buscar DNI.

La figura 4.13 se corresponde con la siguiente expresión *\d{2}([\.-]?|\s*)\d{2}([\.-]?|\s*)\d{2}([\.-]?|\s*)\d{2}\s*\w,* se ha diseñado para que sea lo más general posible y reconozca la mayoría de formas de escribir el número de identificación del DNI: con puntos, rayas entre los números, escribiendo los números de dos en dos o todos juntos…, y se ha obligado que el DNI disponga de la letra final. Asimismo como la expresión regular no garantiza que la entidad encontrada sea un DNI válido se ha añadido la lógica

necesaria para descartar los DNI que no sean correctos, para ello el artículo 11 del Real Decreto 1553/2005[31], del 23 de diciembre, establece que el Documento Nacional de Identidad recogerá el número personal del DNI y el código de verificación correspondiente al número de Identificación Fiscal, cuyo algoritmo de cálculo del dígito de control y por tanto de verificación es el siguiente: se divide el número entre 23 y el resto se sustituye por una letra que se determina por inspección mediante las tablas de la figura 4.14:

RESTO	0	1	2	3	4	5	6	7	8	9	10	11
LETRA	T	R	W	A	G	M	Y	F	P	D	X	B

RESTO	12	13	14	15	16	17	18	19	20	21	22
LETRA	N	J	Z	S	Q	V	H	L	C	K	E

Figura 4.14: Tablas para la verificación del DNI.

Con respecto a los nombres y apellidos como son entidades más abstractas dependientes del contexto se han utilizado modelos de aprendizaje entrenados para poder localizarlos.

El primer modelo que se planteó utilizar fue el Stanford Core NLP, un conjunto de herramientas de análisis lingüístico creada por la universidad de Stanford, consta de una tubería de procesamiento con análisis de etiquetas semánticas, análisis de dependencia y reconocimiento de entidades nombradas entre otros, el cual está construido sobre campos condicionados aleatorios(CRF), aunque sea una herramienta muy común en esta disciplina y disponga de un modelo de lenguaje para el español, es bastante lenta y está orientada a la investigación, para un sistema de localización y modificación de datos en tiempo real no es una buena idea utilizarlo.

Otros de los posibles modelos planteados fue utilizar algunos de los que ofrece la librería NLTK típicamente utilizada en *Python*, en la que se pueden elegir modelos hechos con diversas técnicas de aprendizaje como redes bayesianas, sin embargo encontrar algunos entrenados para el español es muy complicado y probablemente el rendimiento no sería adecuado, se planteó a su vez hacer uso de redes LSTM que actualmente son las que, en términos de rendimiento, generan mejores resultados para el NER, no obstante el inconveniente de usar estas redes neuronales es que son bastante complejas de entrenar y necesitan de un corpus textual muy grande para obtener resultados válidos.

Finalmente, se ha escogido la librería *SpaCy* para *Python*, *SpaCy* dispone de una serie de modelos CNN en diferentes idiomas entrenados bajo un enfoque multitarea aplicados al procesamiento del lenguaje natural, además ofrece modelos de lenguaje hechos mediante *Word2Vec*, dispone de un entorno con enfoque en producción y la posibilidad del reentrenamiento de los modelos, si fuera necesario. Pero se ha elegido *SpaCy* sobre todo porque ofrece las técnicas de aprendizaje más eficiente con respecto al

tiempo y al rendimiento, la única desventaja es que los modelos en español suelen ser menos eficientes en comparación con los aplicados al inglés.

Una vez que se ha dispuesto algún modelo funcional para realizar tareas de NER realmente se puede localizar nombres y apellidos con facilidad pues el modelo en español que nos ofrece *SpaCy* puede localizar personas como entidades en textos, con eso es suficiente para encontrar nombres, sin embargo, suele incurrir en bastantes falsos positivos sobre todo si se encuentra con palabras que comienzan con mayúsculas y actúan como sustantivos en oraciones. Por ello se hace uso de una base de conocimiento con más de cincuenta mil nombres y más de setenta mil apellidos distintos tanto de hombres como de mujeres para filtrar las entidades y obtener aquellas en las que se tengan certeza que sean nombres de personas.

El enfoque para realizar el filtro es el de aprovechar el optimizador de consultas que nos ofrece el motor SQL en las bases de datos relacionales, para ello dada una posible entidad etiquetada como persona esta se divide en palabras, se normaliza a Unicode las ristras para que la base de datos pueda entenderlas, en este caso los datos en las tablas están en mayúsculas y sin acentos, y se realiza la consulta conjuntamente con el resto de entidades de todo el conjunto encontrado tanto en las tablas de nombres como de apellidos, escogiendo un subconjunto que representan las entidades encontradas por el modelo y que están en la base de datos. Como pueden existir nombres y apellidos que no estén en la base de datos, sobre todo si no son muy usuales, se ha establecido una heurística de selección por la cual el porcentaje de coincidencia de una entidad con la base de conocimiento tiene que ser mayor a un determinado valor para ser considerada una entidad válida, en la figura 4.15 se encuentra el código del filtro. Como disponemos de índices en cada tabla para acelerar las consultas deberíamos garantizar un orden logarítmico con esta técnica. Asimismo, a la hora de normalizar los datos de un texto para poder realizar consultas adecuadamente si se pretende quitar los acentos de una ristra hay que tener cuidado con que el método utilizado respete el carácter ‘ñ’, pues si se pretende pasar los caracteres de las ristras a Unicode este no reconoce la ‘ñ’ como un carácter válido.

Hay que tener en cuenta que este método se aprovecha del optimizador del motor SQL, que está presente en multitud de gestores de base de datos, aunque en este TFG se ha usado *SQlite* esta técnica también es aplicable a otras bases de datos relacionales, teniendo en consideración que existe un límite con respecto al tamaño que ocupa la consulta que se va a realizar. En el servidor solo se puede realizar una consulta con un máximo de 4000 palabras para garantizar que no se exceda este límite, el método *generateWordAsString* de la figura 4.15 garantiza que se cumpla esta restricción.

```
def _convertName(self,name:str) -> list:
    normalizeName = normalizeUnicode(str(name)).upper()
    words         = list(
        filter(
                lambda n: n not in self.keywords,
                normalizeName.replace('-', ' ').replace(',','').replace('\'','').split()
            )
        )
    return words

def _checkNameInSubset(self,name:list, nameSubset:list) -> bool:
    if not name: return False
    namesFound = list(filter(lambda n: n in nameSubset, name))
    return len(namesFound)/len(name) >= self.errorRange


def checkNamesInDB(self, names:list) -> list:
    listNames = list(map(lambda x: self._convertName(x), names))
    words     = list(chain.from_iterable(listNames))
    if not words: return []
    nameSubset = []

    for strName in generateWordsAsString(list(set(words))):
        try:
            sentence                    = "select distinct names from names where names in %s;" % (strName)
            senteceResult               = self.connection.query(sentence)
            nameSubset[len(nameSubset):] = [queryResult[0] for queryResult in senteceResult.fetchall()]
            sentence                    = "select distinct surnames from surnames where surnames in %s;" % (strName)
            senteceResult               = self.connection.query(sentence)
            nameSubset[len(nameSubset):] = [queryResult[0] for queryResult in senteceResult.fetchall()]
        except lite.OperationalError as identifier:
            print(identifier)

    finalNames = list(
        filter(
            lambda name: self._checkNameInSubset(name,nameSubset), listNames
        )
    )
    return [names[listNames.index(strName)] for strName in finalNames]
```

Figura 4.15: Código para realizar el filtro de entidades con el diccionario.

4.2.2 La extracción de información en textos sin contexto

En esta sección se van a explicar los algoritmos utilizados para la extracción de entidades en datos estructurados, que se caracterizan porque no disponen de un contexto asociado con el que se pueda realizar algún tipo de inferencia. Dentro de este conjunto de textos hemos agrupado también a las listas, debido a que para ciertos documentos sobre todo académicos, estas suelen presentar rasgos escritos que confunden a los clasificadores como pueden ser el uso de mayúsculas, o un orden no usual de los apellidos y el nombre de la persona. Para realizar la detección de entidades en textos con estas características se han utilizado técnicas de diccionario, heurísticas simples y técnicas de *Word embedding*, inspiradas en (Wong, W. 2009) [26]. En el caso de los documentos nacionales de identidad la forma más eficiente de buscarlos en un texto, independientemente de su formato, es con expresiones regulares.

NER para las tablas

En el caso de los datos estructurados, como el software está destinado a tratar con documentos oficiales éstos deben presentar ciertas reglas y una de ellas es que la información esté bien presentada, eso quiere decir que en el caso de encontrarse en un documento alguna tabla, esta dispondrá de una cabecera y de varias columnas donde su

contenido será algún concepto o alguna expresión cuyo significado debería estar relacionado con la cabecera correspondiente. Por ello para buscar entidades se ha elegido procesar las columnas ya sea para detectar DNI o nombres. En el caso de los primeros como *Python* dispone de una librería bastante eficiente para las expresiones regulares y de funciones como *filter* o *map* y de compresiones de lista que ayudan a disminuir la complejidad temporal, es viable procesar toda una tabla indistintamente de su tamaño, por otra parte en el caso de los nombres se ha optado por hacer consultas al diccionario de nombres para poder comprobar si en una celda existe algún nombre o no, la heurística que se plantea es la siguiente: si una columna tiene una gran cantidad de coincidencia con el diccionario lo más probable es que la columna contenga nombres, esto es debido a que en el caso de que la columna contenga ciudades o calles, por lo grande que es el diccionario, lo más probable es que algunas celdas tengan una entrada en la base de datos, con esto en mente si se elige un valor heurístico que represente el porcentaje mínimo de coincidencia de todas las celdas de una columna en la base de datos estaremos garantizando una gran disminución de falsos positivos en el resultado.

Si hablamos de complejidad temporal cada columna se va a procesar dos veces una para comparar cada celda con el diccionario y otra para comparar cada celda con la expresión regular, la complejidad de la búsqueda en la base de datos puede variar según el tipo de consulta o estructura de esta. En nuestro proyecto disponemos de dos tablas indexadas y realizamos las consultas de tal forma que por toda una lista de nombres sea el optimizador el que se encarga de procesar la consulta, lo cual nos debería garantizar como máximo un orden logarítmico, y en caso de las expresiones regulares como se trata de un autómata finito determinista el orden tendría que ser igualmente polinómico (Russ, C. 2007)[34] aunque puede variar según la forma de implementarlo en la librería *re* de *Python* y los métodos usados.

Para un documento con muchas tablas o para alguna hoja de cálculo con muchos datos este algoritmo puede llegar a ser ineficiente si por cada celda tenemos que consultar a la base de datos. Pero en términos estadísticos como solo estamos buscando un porcentaje de coincidencia no es necesario utilizar todas las muestras para poder clasificar si la columna contiene nombres o no, si el número de muestras es suficientemente grande con tan solo una muestra aleatoria se puede lograr resultados en menos tiempo, es decir, si una columna tiene más de 10000 celdas si escogemos una muestra aleatoria de un tamaño significante por ejemplo el 50% y el número de coincidencias con la base de datos es superior a un determinado valor heurístico, se puede inferir que toda esa columna almacena nombres.

La clave para conseguir buenos resultados con este algoritmo es jugar bien con los valores heurísticos para disminuir la relación entre posibles falsos positivos y negativos. De esta forma se consigue reducir el número de consultas a la base de datos, pero la columna se debería seguir procesando entera para buscar DNI. Una forma de mejorar la eficiencia es conseguir dividir el conjunto de columnas en dos subconjuntos: uno destinado a buscar posibles nombres y otro que tan solo busque los carnés de identidad, como estamos ante documentos con un formato claro y donde las columnas de las tablas representan un único concepto, una forma de lograr la división es analizando la cabecera de cada columna, para lograr esto hacemos uso de un modelo de lenguaje. *SpaCy* nos ofrece un modelo de lenguaje entrenado en español con una dimensión de R^{50}, gracias

a esto se puede lograr dividir el conjunto de palabras buscando sinónimos, es decir, seleccionamos varias expresiones que puedan ser cabeceras de una tabla cuyas columnas albergan nombres como: personas, personal, sujetos… y filtramos las columnas si su cabecera tienen un grado de similitud muy elevado con alguna de las expresiones seleccionadas, de esa forma las columnas escogidas interactúan con la base de datos y las restantes solamente con la expresión regular, obteniendo un rendimiento adecuado sin realizar demasiadas consultas a la base de datos, en la figura 4.16 se puede ver un ejemplo de código para ver similitudes de palabras usando *SpaCy*.

```
def semanticSimilarity(self, text: str, textToCompare: str) -> float:
        if not text.strip():
            return 0.0

        doc = self.vectorialSpace(text)

        if not doc.vector_norm:
            return False

        docToCompare = self.vectorialSpace(textToCompare)

        return doc.similarity(docToCompare)
```

Figura 4.16: Código para buscar sinónimos es *SpaCy*.

Con estas técnicas se ha conseguido procesar datos estructurados obteniendo un buen rendimiento y con una eficiencia adecuada para un servicio en tiempo real que tendrá que trabajar con varios documentos a la vez.

NER para las listas

Las listas son un tipo de dato no estructurado frecuente en un sinfín de documentos, el modelo *es_core_news_md* no tiene problemas a la hora de encontrar nombres en listas si estos están bien formados, sin embargo, como es muy común listar nombres de personas, sobre todo en documentos académicos, en mayúsculas o utilizando primero los apellidos y separando nombre y apellidos por comas, el modelo no es capaz de etiquetar adecuadamente los nombres debido a que no está entrenado para encontrar estas entidades en este tipo de contexto. Para poder superar esto, antes de usar el clasificador se analizan los textos para comprobar si poseen algunas de las características anteriores, si fuera ese el caso se tendría que usar únicamente la base de conocimiento para poder lograr una búsqueda adecuada.

El algoritmo que se utiliza para hacer esto es bastante trivial, por un lado, *SpaCy* dispone de un etiquetador básico por reglas, su funcionamiento es simple, utiliza los tokens y sus etiquetas semánticas para crear entidades customizadas, de esto se hablará más detenidamente en la siguiente sección. *SpaCy* dispone de una etiqueta para asociar un token a un nombre propio(PROPN) o para asignar signo de puntuación(IS_PUNCT), entre otras, por lo que para poder buscar expresiones como: “Medina Ramírez, Miguel Ángel”, simplemente usamos el objeto *matcher* de *SpaCy* y cargamos los patrones a buscar siendo estos muy similares a las expresiones regulares, en la figura 4.17 están los

patrones y el código necesario para utilizar el *matcher*, el patrón OP se usa para denotar frecuencia de ocurrencia. Asimismo, hemos añadido una nueva restricción y es que, si una ristra dispone de una gran cantidad de letras en mayúsculas o bien puede ser un título, una expresión importante o un nombre en mayúsculas vale la pena comprobarlo porque si existe un nombre escrito en esa forma el modelo no será capaz de identificarlo, por lo que solo nos queda recurrir al diccionario.

```
self.matcher = Matcher(self.nlp.vocab)
        patterNotContext = [
                {'POS'      : 'PROPN', 'OP': '+'},
                {"IS_PUNCT" : True},
                {'POS'      : 'PROPN', 'OP': '+'}
            ]
self.matcher.add("withoutContext", None, patterNotContext)
```

Figura 4.17: Ejemplo de Patrones de *SpaCy*.

4.2.3 La extracción de información en textos con contexto

A la hora de abordar una tarea de etiquetado de entidades nombradas lo primero que se debe tener en cuenta es que la entrada para nuestro sistema será una ristra de tamaño variable, esta ristra deberá ser procesada de alguna manera para obtener una estructura de datos con las entidades. En (Bird et al. 2009) [1] se propone una arquitectura que consiste en tratar a los diferentes procesos que intervienen en la búsqueda como una tubería de procesamiento que maneje los datos de diferente forma para obtener el resultado deseado, en la figura 4.18 se puede apreciar la arquitectura y las diferentes estructuras de datos que maneja. En ella se propone que una ristra se segmente en frases, luego se obtenga los tokens correspondientes, estos tokens serán etiquetados mediante algún algoritmo de etiquetado semántico, una vez mapeado cada token con sus etiquetas de discurso estas se utilizaran como metadato para lograr localizar entidades nombradas, y por último obtener la relación de dependencia semántica entre los tokens del discurso.

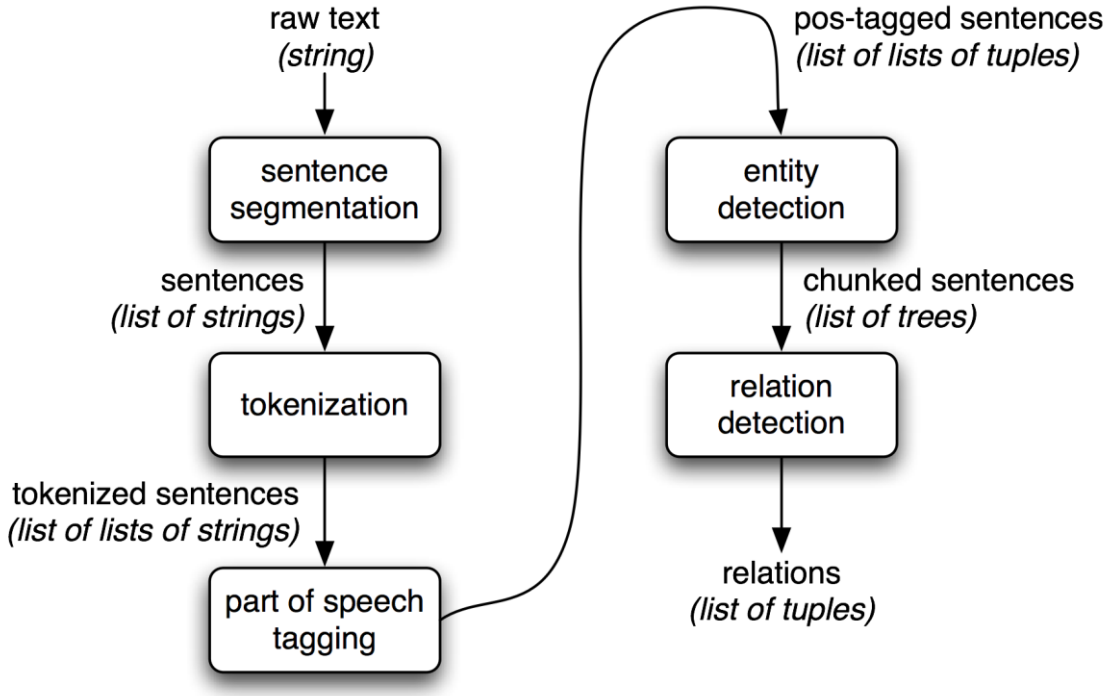


Figura 4.18: Arquitectura de tubería simple para un sistema de extracción de información.

Fuente: (Bird et al. 2009) [1].

SpaCy utiliza una arquitectura similar como se puede ver en la figura 4.19 donde la detención de dependencias se realiza antes de la detección de entidades, además se centra en que la salida obtenida sea una instancia de la clase *Doc* que almacenará la lista de tokens con sus metadatos correspondientes y el conjunto de entidades nombradas que haya encontrado. *SpaCy* dispone a su vez de un modelo de lenguaje que permite mapear los tokens a un espacio vectorial de 50 dimensiones para lograr mejores inferencias y que en cada parte de la tubería se puedan utilizar modelos de aprendizaje automático.

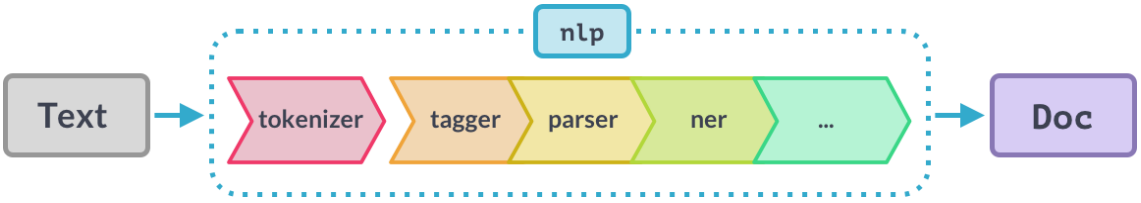


Figura 4.19: Tubería de procesamiento de SpaCy.

Fuente: (Explosion AI) [14].

NER basado en Aprendizaje automático

El detector de entidades del modelo *SpaCy* que se usa en este proyecto permite buscar entre personas, localizaciones, organizaciones y entidades diversas: eventos, nacionalidades, productos u obras de arte. Sin embargo, solo nos interesan las personas, en concreto, por lo que dado una ristra solamente tenemos que extraer las personas y devolver una lista con todas ellas, dado que el modelo *es_core_news_md* suele ser

bastante general con estas entidades y confundirlas con sustantivos acompañados de verbos o palabras que empiezan en mayúsculas, para evitar una explosión de falsos positivos se utiliza una comparación con la base de datos de nombres. La dificultad de esta técnica está en utilizar un valor heurístico adecuado en la comparación con la base de datos para que el conjunto de nombres sea lo más fiel a la base de conocimiento logrando reducir al máximo el número de falsos positivos, pero que a su vez permita que nombres y apellidos que no estén en el diccionario también sean considerados como tales y evitar así aumentar innecesariamente el número de falsos negativos, de esto se hablará más en el capítulo 5.

Un detalle que nos dimos cuenta a la hora de probar el rendimiento del modelo fue que muchas veces los fallos venían porque no lograba dividir bien grandes cadenas de textos, esto es debido a que el algoritmo para tokenizar textos que tiene *SpaCy* simplemente divide en palabras y símbolos, lo cual funciona muy bien para oraciones o párrafos no demasiado grandes, pero como las técnicas para procesar documentos que vimos en las sesiones anteriores extraen la información por páginas, el tokenizador se enfrenta en varios casos a textos con mucha información por lo que en ocasiones no consigue obtener tokens que permitan a los modelos hacer una buena inferencia, esto suele ser recurrente también en otras partes de la literatura como en (Mandal, J. & Mukhopadhyay, S. 2020) [32]. Una forma de solucionarlo es añadir a la arquitectura un proceso previo de segmentación como sugiere la figura 4.18, para una cadena de textos el primer paso sería dividir la cadena en oraciones o frases permitiendo al tokenizador de *SpaCy* obtener mejores resultados. Un buen algoritmo para lograr la segmentación es el propuesto por (Kiss, T. & Strunk, J. 2006) [33] un tipo de técnica no supervisada para la detención de oraciones y frases en distintos idiomas se puede utilizar con la librería NLTK descargando el módulo *punk*. En la figura 4.20 se puede ver el código para la extracción de entidades, el método *replaceUnnecessaryCharacters* se utiliza para quitar de los nombres encontrados posibles caracteres extraños como corchetes o paréntesis, también se ha reemplazado el carácter barra horizontal por una coma, esto es debido a que suele haber fallos en el tokenizador si el texto procesado es un diálogo, por ello se sustituye ese carácter por otro semánticamente similar.

```python
class PersonalDataSearchByEntities(PersonalDataSearch):

    def __init__(self):
        super().__init__()
        self.nlp = LanguageBuilder().getlanguage()

    def searchPersonalData(self, text: Text) -> tuple:
        listNames = []

        for tokenText in sent_tokenize(text.replace('–', ',') ,language='spanish'):
            doc = self.nlp(tokenText)
            listNames[len(listNames):] = self.checkNamesInDB([
                replaceUnnecessaryCharacters(ent.text.strip('\n'))
                for ent in doc.ents if ent.label_ == 'PER'
            ])

        idCards = [
            idCard.group()
            for idCard in list(filter(lambda x: isDni(x.group()) , re.finditer(self.regexIdCards,text)))
        ]

        return (listNames,idCards)
```

Figura 4.20: Código para las extracciones de nombres mediante aprendizaje automático y extracciones de carnés de identidad mediante expresiones regulares.

NER basado en Reglas

Como en este TFG se pretende comparar la eficiencia de las técnicas para la detección de entidades nombradas con aprendizaje automático y aquellas basadas en reglas, también se ha ideado una forma de lograr tener otro buscador de entidades basado en ingeniería del conocimiento haciendo uso de reglas y gramáticas, para ello *SpaCy* consta de un etiquetador de tokens basado en patrones, es decir, para una frase tokenizada y que disponga de etiquetas del discurso se puede emplear reglas para buscar cualquier tipo de entidad personalizada. En la figura 4.21 se muestran las reglas aplicadas y como usar el etiquetador o *matcher* que posee *SpaCy*, asimismo para lograr un mejor rendimiento se ha desactivado la parte de la tubería encargada del NER. Esta técnica se basa en el etiquetador morfosintáctico (parte de las tuberías PARSER y TAGGER) que posee *SpaCy* el cual está diseñado sobre redes convolucionales, por lo que entraría en el rango de técnica NER híbrida.

```
def defineRulesOfNames(self):
        names      = [
            {'POS' : 'PROPN', 'OP': '+'},
            {'TEXT': {'REGEX': 'de|del|-
|el|los|de todos los'}, 'OP': '?'},
            {'POS' : 'PROPN', 'OP': '*'}
        ]
        ruler      = EntityRuler(self.nlpRules)
        self.label = "NAME"
        patterns = [
            {"label": self.label, "pattern":names}
        ]
        ruler.add_patterns(patterns)
        self.nlpRules.add_pipe(ruler, after='tagger')
```

Figura 4.21: Reglas para buscar nombres y apellidos mediante el etiquetador de *SpaCy*.

Las reglas para seleccionar nombres están basadas en encontrar tokens que sean nombres propios y una serie de proposiciones que pueden llevar los nombres compuestos. La estructura de datos que hay que pasar a la clase *EntityRuler* para que pueda crear las reglas es una lista de patrones secuenciales, lo que significa que el primer patrón de la lista corresponde a los primeros tokens de la entidad, el segundo a los tokens que cumplan ese patrón después del primero, y así sucesivamente, a continuación, se explica cada patrón de búsqueda por separado:

- **{'POS': 'PROPN', 'OP': '+'}:** Este patrón busca una serie de tokens cuya etiqueta de discurso (POS) será un nombre propio (PROPN) que se repita una o más veces ('OP': '+').
- **{'TEXT': {'REGEX': 'de|del|-|el|los|de todos los'}, 'OP': '?'}:** En este patrón se usa una expresión regular para buscar las palabras destacadas, las cuales se pueden repetir entre 0 o 1 vez.
- **{'POS': 'PROPN', 'OP': '*'}:** Este patrón busca los últimos nombres propios de un nombre completo, corresponde a los posibles apellidos no compuestos que puede tener las entidades, pueden aparecer entre 0 o más veces.

La intersección de estos patrones en tokens seguidos corresponderá a un nombre, por ello se le asigna al conjunto resultante la etiqueta *NAME*. Como esta es una metodología NER basada en reglas puede ser menos precisa que una perspectiva basada únicamente en redes neuronales, por ello y para evitar la detección de falsos positivos se utiliza el diccionario al igual que en el método anterior. En la figura 4.22 se puede ver el código para la detección de entidades, como se puede apreciar se sigue utilizando el tokenizador *punk* para que los modelos de *SpaCy* logren una mejor inferencia.

```python
class PersonalDataSearchByEntities(PersonalDataSearch):

    def __init__(self):
        super().__init__()
        self.nlp = LanguageBuilder().getlanguage()

    def searchPersonalData(self, text: Text) -> tuple:
        listNames = []

        for tokenText in sent_tokenize(text.replace('–', ',') ,language='spanish'):
            doc = self.nlp(tokenText)
            listNames[len(listNames):] = self.checkNamesInDB([
                replaceUnnecessaryCharacters(ent.text.strip('\n'))
                for ent in doc.ents if ent.label_ == 'PER'
            ])

        idCards = [
            idCard.group()
            for idCard in list(filter(lambda x: isDni(x.group()) , re.finditer(self.regexIdCards,text)))
        ]

        return (listNames,idCards)
```

Figura 4.22: Código para las extracciones de nombres mediante patrones y extracciones de carnés de identidad mediante expresiones regulares.

4.2.4 Heurísticas utilizadas

En secciones anteriores se han explicado las heurísticas asociadas a la búsqueda de entidades nombradas que se han aplicado en este proyecto. Todas ellas están orientadas a valores numéricos que simbolizan un porcentaje, con este dato se pueden hacer descartes a la hora de valorar un nombre o un texto. En el servicio son tratadas como constantes que se pueden modificar en cualquier momento cumpliendo con el principio de diseño abierto-cerrado, están agrupadas en el fichero *heuristicMeasures.py*, la figura 4.23 muestra un ejemplo del estado de este fichero, todas las variables que representen porcentajes tienen un valor de entre 0 y 1, algunas de estas no están relacionadas con las heurísticas utilizadas, pero son constantes esenciales para el correcto funcionamiento del sistema.

```python
ERROR_RANGE_PERCENTAGE_DB: float             = 0.5
MEASURE_FOR_TEXTS_WITHOUT_CONTEXTS: float    = 0.8
MEASURE_TO_COLUMN_KEY_REFERS_TO_NAMES: float = 0.75
MINIMAL_UPPER_CHAR_DENSITY: float            = 0.8
MAXIMUM_NUMBER_OF_ELEMENTS_IN_A_REGEX: int   = 4000

SAMPLE_DATA_TO_CHOOSE_NAMES:float            = 0.25
MAXIMUM_NUMBER_OF_POSSIBLE_NAMES_FOR_A_QUERY: float = 10000
```

Figura 4.23: Contenido del fichero heuristicMeasures.py.

- **ERROR_RANGE_PERCENTAGE_DB:** representa el valor mínimo o umbral de coincidencia que debe tener un nombre con la base de conocimiento.
- **MEASURE_FOR_TEXTS_WITHOUT_CONTEXTS:** representa el valor mínimo de coincidencia de toda una columna con la base de conocimiento.

- **MEASURE_TO_COLUMN_KEY_REFERS_TO_NAMES:** representa el porcentaje para valorar si una cabecera de una columna es sinónimo de las listas de expresiones que denotan cabeceras que almacenen nombres.
- **MINIMAL_UPPER_CHAR_DENSITY:** representa la densidad de caracteres en mayúsculas para poder considerar que un texto no tiene suficiente contexto para que el modelo pueda hacer una inferencia adecuada.
- **SAMPLE_DATA_TO_CHOOSE_NAMES:** representa el tamaño de la muestra aleatoria que se escoge para valorar una columna si esta es demasiado grande.
- **MAXIMUM_NUMBER_OF_POSSIBLE_NAMES_FOR_A_QUERY:** representa la dimensión máxima que puede tener una columna para que se valore cada celda.
- **MAXIMUM_NUMBER_OF_ELEMENTS_IN_A_REGEX:** representa el tamaño máximo posible de una expresión regular que puede procesar *Python*.

4.3 Algoritmos de ofuscación, disgregación y anonimización de datos personales

Según la Agencia Española de protección de datos 2016[30], la finalidad de estos procesos es la de eliminar o reducir al mínimo los riesgos de reidentificación de los datos manteniendo la veracidad de los resultados del tratamiento de estos.

Hemos escogido tres algoritmos uno por cada tipo de procedimiento, en el caso de la anonimización hemos elegido sustituir los datos por cadenas de la misma longitud que el dato original del carácter '*', en el caso de la ofuscación para que el usuario de los datos pueda reconocerlos si se trata de un nombre hemos decidido dejar las primeras letras de las palabras de la entidad y el resto sustituirlo por el carácter '*', al igual que los carnés de identidad, y por último, para disgregar los datos la Agencia Española de protección de datos 2016[30] recomienda dejar los cuatro últimos números del DNI visibles y a anonimizar el resto de caracteres, en el caso de los nombres se anonimiza la expresión entera, en la figura 4.24 está la implementación de estos procedimientos. Existen más algoritmos, algunos incluyen el uso de funciones hash o claves de cifrado para poder recuperar la información, sin embargo, hemos decidido escoger procedimientos sencillos y usuales que cumplan con los objetivos. Existe una pequeña particularidad en los documentos *pdf* por lo cual el carácter '*' no se va a procesar bien en el software de (Tauber, J. 2019) [12]. En su defecto, a la espera que solucionen este problema, en el caso

de los *pdf* las entidades se anonimizarán, ofuscarán o disgregarán utilizan el carácter vacío.

```
def encode(text: str) -> str:
    return "*" * len(text)

def markInHtml(text: str) -> str:
    return '<mark style="background: #7aecec;">' + text + '</mark>'

def disintegration(text:str) -> str:
    if isDni(text):
        chars      = list(text)
        chars[0:3] = 3*'*'
        chars[7:]  = 2*'*'
        return ''.join(chars)
    else:
        return encode(text)

def dataObfuscation(text:str) -> str:
    if isDni(text):
        chars      = list(text)
        chars[1:7] = 7*'*'
        return ''.join(chars)
    else:
        result = map(
            lambda name: '%s%s' %(name[0],len(name[1:])*'*'),
            text.split()
        )
        return ' '.join(result)
```

Figura 4.24: Implementación de los algoritmos de ofuscación, disgregación y anonimización de datos.

4.4 Servicio Web

Un servicio web es un sistema software diseñado para soportar la interacción máquina-a-máquina, a través de una red, de forma interoperable. Cuenta con una interfaz descrita en un formato procesable por un equipo informático (específicamente en WSDL), a través de la que es posible interactuar con el mismo mediante el intercambio de mensajes SOAP, típicamente transmitidos usando serialización XML sobre HTTP conjuntamente con otros estándares web (W3C 2020) [37].

Se ha elegido implementar un servicio web para el prototipo de este proyecto, a pesar de que los algoritmos y técnicas de búsqueda de datos en documentos pueden ser perfectamente implementables bajo cualquier otra tecnología o estándar. Una api o servicio web otorga por un lado independencia a una entidad o equipo con respecto al hardware y a las dependencias que pueden suscitar las técnicas usadas, y por otro lado, brinda la posibilidad de construir un sistema de anonimización o extracción de datos personales abstrayéndose de detalles propios de la localización de datos y el manejo de la información textual en los documentos. Aunque en este proyecto queda por abordar temas

relativos a la seguridad y modelo de negocio, se han centrado los esfuerzos sobre todo en el diseño de software y modelos de detección de información en textos.

4.4.1 Api REST

En los años 2000, Roy Fielding propuso la transferencia de estado representacional (REST) como enfoque de arquitectura para el diseño de servicios web. REST es un estilo de arquitectura para la creación de sistemas distribuidos basados en hipermedia. REST es independiente de cualquier protocolo subyacente y no está necesariamente unido a HTTP. Sin embargo, en las implementaciones más comunes de REST se usa HTTP como protocolo de aplicación, y proyecto se centra en el diseño de la API para HTTP.

Una de las principales ventajas de REST sobre HTTP es que usa estándares abiertos y no vincula la implementación de la API o de las aplicaciones cliente con ninguna implementación específica. Por ejemplo, se podría escribir un servicio web de REST en ASP.NET, y las aplicaciones cliente pueden usar cualquier lenguaje o conjunto de herramientas que puedan generar solicitudes HTTP y analizar respuestas HTTP (Microsoft. 2018) [22].

4.4.2 Estructura de un proyecto en Flask-restPlus

En términos de organización de proyectos, se ha hecho uso de una estructura funcional, es decir organizar los elementos de la aplicación por lo que hacen. En una estructura funcional, las plantillas se agrupan en un directorio, los archivos estáticos en otro y las vistas en un tercero (Obinna, G. 2018) [43]. En el directorio del proyecto, existe una carpeta *app,* dentro de ella se encuentra dos carpetas llamadas *main* y *test,* que implementan la lógica del servicio y la de las pruebas, respectivamente. La estructura del directorio es la siguiente:

```
.
├── app
│   ├── __init__.py
│   ├── main
│   │   └── __init__.py
│   └── test
│       └── __init__.py
└── requirements.txt
```

Figura 4.25: Estructura del proyecto base con Flask-Restplus.

Fuente: (Obinna, G. 2018) [43].

En el directorio *main* se encuentra el fichero *config.py*, este fichero se asegura de brindarnos tres instancias diferentes para nuestra aplicación: de prueba, de desarrollo y

de producción, asegurándose de cambiar los recursos necesarios para cada una de ellas. Para que esto sea posible, utiliza el patrón de diseño *applicaction factory* este patrón se encarga de crear múltiples instancias de una misma aplicación con recursos y dependencias diferentes para cada una, también se suele utilizar para cambiar entre versiones de una aplicación. El fichero *config.py* de este servicio sería el siguiente:

```python
import os

basedir = os.path.abspath(os.path.dirname(__file__))

class Config:
    SECRET_KEY = os.getenv('SECRET_KEY', 'my_precious_secret_key')
    DEBUG = False


class DevelopmentConfig(Config):
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = 'sqlite:///' + os.path.join(basedir, 'flask_File_main.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False


class TestingConfig(Config):
    DEBUG = True
    TESTING = True
    SQLALCHEMY_DATABASE_URI = 'sqlite:///' + os.path.join(basedir, 'flask_File_test.db')
    PRESERVE_CONTEXT_ON_EXCEPTION = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False


class ProductionConfig(Config):
    DEBUG = False
    SQLALCHEMY_DATABASE_URI = 'sqlite:///' + os.path.join(basedir, 'flask_File_test.db')
    PRESERVE_CONTEXT_ON_EXCEPTION = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False

config_by_name = dict(
    dev=DevelopmentConfig,
    test=TestingConfig,
    prod=ProductionConfig
)

key = Config.SECRET_KEY
```

Figura 4.26: Configuración del servidor.

Bajo el directorio raíz del proyecto se tiene que crear un fichero llamado *manage.py* que se encarga en implementar la iniciación del servidor, así como también el tipo de instancia que se quiere ejecutar. Se ha creado a su vez un script en bash llamado *setup.py* para que la ejecución del servicio sea lo más sencilla posible.

4.4.3 Buscador de datos personales

La idea de este trabajo es lograr tener un servicio web que proporcione la capacidad de extracción de datos personales, tanto nombres y apellidos que hacen referencia a una persona como DNI, en formato *json* o *csv*, así como también disponer de mecanismos de reducción de riesgo de identificación. Con ese objetivo en mente se ha dispuesto la siguiente API web:

- ***/search/version:*** Devuelve mediante GET la versión actual del software.
- ***/search/file/encode:*** Se encarga de recoger ficheros mediante POST y de devolver otra versión de ellos con los datos personales anonimizados. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento.
- ***/search/file/disintegration:*** Se encarga de recoger ficheros mediante POST y de devolver otra versión de ellos con los datos personales transformados mediante un proceso de disgregación. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento.
- ***/search/file/obfuscation:*** Se encarga de recoger ficheros mediante POST y de devolver otra versión de ellos con los datos personales transformados mediante un proceso de ofuscación. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento.
- ***/search/file/extract-data/json-file:*** Se encarga de recoger ficheros mediante POST y de devolver un fichero *json* con los datos personales encontrados. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento.
- ***/search/file/extract-data/json:*** Se encarga de recoger ficheros mediante POST y de devolver un objeto *json* con los datos personales encontrados. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento.
- ***/search/file/extract-data/csv:*** Se encarga de recoger ficheros mediante POST y de devolver un fichero *csv* con los datos personales. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento.
- ***/search/file/operation-web:*** Se encarga mediante GET de recoger una URL y un tipo de operación de transformación o extracción de datos, y de devolver un archivo *json* o *csv* con los datos o el archivo *html* con las entidades transformadas. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento.
- ***/search/file/tagger-html:*** El último *endpoint* se encarga de recoger ficheros *html* mediante POST y de devolverlos, pero marcados con los datos etiquetados. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento. Este recurso está para tener una representación visual de la bondad del sistema en la localización de los datos.

4.5 Arquitectura y Diseño de Software

Dado que el objetivo final de este proyecto es disponer de un servicio de anonimización y extracción de datos, es necesario diseñar el software marcando una jerarquía y organización de los diferentes componentes que integran la aplicación web, de tal manera que si se llega a poner en producción o se quiere avanzar y modificar algunos de sus aspectos se puede hacer relativamente sencillo y cumpliendo con los

principios del buen diseño del software. Dentro de la arquitectura también cabe destacar el tipo de paradigma utilizado que en este caso ha sido un paradigma orientado a objetos.

4.5.1 Model-Controller-Service

El *Model-Controller-Service*(MCS) es un patrón arquitectónico de software heredero del clásico *Model-View-Controller*. En el MVC se separa los datos y la lógica de negocio de una aplicación de su representación y el módulo encargado de gestionar los eventos y las comunicaciones (Obinna, G. 2018) [43]. Para ello MVC propone la construcción de tres componentes distintos que son el *modelo*, la *vista* y el *controlador*, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario. Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento. El MVC actualmente es uno de los patrones más usados sobre todo en web y en interfaces gráficas.

El MCS intenta llevar el concepto del MVC al dominio de las API REST, para ello se plantea la división del código en tres partes:

- **Modelo:** Se utiliza para administrar la información y la lógica de la aplicación, a su vez es común implementar el modelo haciendo uso del patrón *Active Record* ampliamente utilizado en muchos framework como Laravel (PHP) o Spring (Java). Con este enfoque se pretende envolver las tablas que componen una base de datos en una clase de tal forma que los métodos de esta sean los propios de la base de datos, permitiendo a las clases interactuar con la base de datos mediante la interfaz proporcionada por la clase envoltorio. Cabe decir que *Active Record* es considerado un anti-patrón, debido a que genera acoplamiento entre la lógica de la aplicación y la base de datos haciendo que las pruebas unitarias sean difíciles de implementar, además de que está pensado para bases de datos relacionales haciendo complejo su implementación en bases de datos no relacionales, no obstante se ha elegido utilizar este patrón porque para un modelo de datos pequeño y que sea relacional como es el caso, el acoplamiento es mínimo y aporta comodidad.
- **Controlador:** Es el encargado de manejar todas las solicitudes HTTP entrantes en la aplicación y usar las operaciones del servicio para poder resolverlas adecuadamente.
- **Servicio:** Esta parte actúa por un lado de capa o interfaz entre el controlador y el modelo, de esa forma dada una petición HTTP al servidor el controlador usará las operaciones suministradas por el servicio para interactuar con el modelo. Por otro lado, en la parte de servicio también se especifican todas aquellas utilidades propias o ajenas al servidor que ayuden a cumplir con las peticiones. Por ejemplo, si se dispone de un servicio web que interactúe con la API de Google, los métodos y las clases necesarias para su implementación estarían especificadas como parte de la capa de servicios. Es aquí donde se implementa toda la lógica necesaria para la extracción, la anonimización y el manejo de los datos en este proyecto.

Asimismo, como estamos en un entorno *Python* existen diversas utilidades comunes a varias capas de la aplicación por ello se ha añadido una cuarta parte donde se componen de utilidades básicas para el proyecto.

4.5.2 Estructura del servicio web

En la sección 4.4.3 se presentó como se estructura un proyecto *Flask-resplus* de manera simple, en esta se explica la organización de ficheros y clases dentro del servidor, bajo los estándares de una estructura funcional:

Archivo/Carpeta	Propósito
storageFiles/	Este directorio se usa para almacenar cualquier documento que se sube al servicio.
migrations/	Este directorio contiene tanto los scripts necesarios para realizar la migración de la base de datos como la información relativa a esta.
app/	Este directorio contiene todos los directorios y ficheros necesarios para la implementación del servicio
app/main	Este directorio contiene el código dividido en tres carpetas *model*, *controller* y *service*, además de las bases de datos tanto para test como para producción y desarrollo.
app/test	Este directorio contiene el código necesario para realizar las pruebas unitarias y de rendimiento.
.vscode/	Este directorio contiene los archivos necesarios para que visual studio sirva de editor al proyecto
.gitignore	Este archivo indica a Git que archivos (o patrones) deberían ignorarse a la hora de hacer confirmaciones (commits)
.gitattributes	Este archivo contiene los atributos para los nombres de ruta de los ficheros de un proyecto en git.
requirements.txt	Este archivo contiene las dependencias del proyecto.
setup.sh	El script de inicialización en un sistema Linux.
spanish_names	La base de datos de nombre.
updateDB.sh	El script de actualización de migraciones para la base de datos, en un sistema Linux.
cleanDB.py	El script en *Python* para limpiar el proyecto de los documentos subidos según el registro de la base de datos.
manage.py	Este archivo es el administrador de la ejecución del servicio.

Tabla 4.1: Estructura general del directorio del servidor

4.5.3 Base de Datos y Modelo

En esta sección se muestra la base de datos generada mediante los métodos ofrecidos por la librería *Flask-SQLAlchemy*. Como se puede ver en la figura 4.27 se utiliza la interfaz para crear las tablas y los atributos necesarios.

```
from .. import db, flask_bcrypt

class FileLog(db.Model):
    __tablename__ = "fileLog"

    id  = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name  = db.Column(db.String(255), unique=True, nullable=False)
    folder = db.Column(db.String(255), nullable=False)
    isDelete = db.Column(db.Boolean, nullable=False)
    filetype = db.Column(db.String(255), nullable=False)
    publicId = db.Column(db.String(100), unique=True,nullable=False)

    def __repr__(self):
        return '<file %r>' % self.name
```

Figura 4.27: Código para generar la base de datos.

Dado la naturaleza del servicio web la capa de datos es muy sencilla: se trata de una única tabla para marcar el registro de los ficheros guardados bajo un mismo directorio y que dispondrán de los siguientes atributos:

- **Id:** El identificador de registros, visible únicamente para la base de datos.
- **PublicId:** Almacena el id visible para la aplicación.
- **Folder:** Almacena la carpeta en el servidor donde se ha guardado el fichero
- **Name:** Almacena el nombre del fichero subido al servidor.
- **FileType:** Almacena el tipo del fichero, es decir, la extensión.
- **IsDelete:** Almacena un indicador booleano para saber si el fichero se puede borrar o no.

En la figura 4.27 se expone el código utilizado para generar la base de datos, como es común con *Active Record* toda clase que se cree para ser envoltorio de la capa de datos debe de heredar de una superclase llamada modelo, la librería ofrece los métodos *db.Column* que actúan como columnas y de los atributos necesarios para su traducción SQL. Para la traducción la librería usa un mecanismo de migración de datos, de tal manera antes de tener la base de datos se deberá exportar el esquema y aplicar los cambios a la base de datos y al gestor correspondiente, tanto la primera vez que se inicie la base de datos como cuando se quiere realizar cualquier cambio. Dentro del proyecto se ha preparado un script con los dos comandos necesarios que se pueden ver en la siguiente figura.

```
python3 manage.py db migrate --message 'new schema'
python3 manage.py db upgrade
```

Figura 4.28: Comandos necesarios para la migración de la BD.

4.5.4 Servicio

La capa de servicios actúa de interfaz entre el controlador y el modelo, por tanto, su función principal es definir aquellas funciones que interactúen con el modelo, de tal forma que el controlador pueda registrar los diferentes documentos en la base de datos y actualizar el estado de los registros. En la figura 4.29 está el código que configura la capa de servicios.

```python
from app.main               import db
from app.main.model.fileLog import FileLog

import uuid

def saveLog(data: dict):
    publicId = str(uuid.uuid4())
    log = FileLog(
        publicId = publicId,
        name     = data['name'],
        folder   = data['folder'],
        isDelete = data['isdelete'],
        filetype = data['filetype']
    )
    commit(log)
    return publicId


def getAllLog():
    return FileLog.query.all()


def getByPublicId(id: str):
    return FileLog.query.filter_by(publicId=id).first()


def getFileToDelete():
    return FileLog.query.filter_by(isDelete=True).all()


def updateDelete(public_id: str, boolean: bool):
    log          = FileLog.query.filter_by(publicId=public_id).first()
    log.isDelete = boolean
    db.session.commit()


def commit(log):
    db.session.add(log)
    db.session.commit()
```

Figura 4.29: Contenido del fichero LogService.py, capa de servicio.

Las operaciones que se definen en la capa de servicios son las siguientes:

- ***saveLog:*** recibe información sobre el documento y lo registra en el modelo devolviendo el identificar del mismo.
- ***getAllLog:*** devuelve todos los registros.
- ***getByPublicId:*** devuelve un registro en específico gracias el identificador recibido.
- ***getFileToDelete:*** Obtiene todos los registros de documentos que se tienen que borrar tanto de la base de datos como del servidor.
- ***updateDelete:*** Actualiza el estado de un registro para que se borre o para que se mantenga.

- ***commit:*** Hace permanente la operación en la base de datos, se usa para cuando se inserta un nuevo registro.

Dentro de esta capa también se ha implementado toda la lógica necesaria para la detección y el procesamiento de entidades textuales, que se explicó en las secciones 4.1 y 4.2 de este capítulo.

4.5.5 Controlador

El controlador es una de las partes más relevantes, en él se define la puerta de entrada o comunicación de la aplicación web con el exterior mediante el protocolo HTTP. *Flask-restplus* dispone de una clase abstracta llamada *Resourse* en donde se puede heredar una serie de métodos para definir las operaciones HTTP, en este servicio se utiliza POST en la mayoría de las peticiones por lo cual la implementación es bastante simple. Asimismo, se dispone de un decorador *@api* que se usa para dar nombre al *endpoint* correspondiente y añadir información sobre las operaciones que realiza. Los *endpoints* ya fueron presentados en la sección 4.4.3, en esta parte se expondrá el algoritmo que siguen para procesar una petición y el código correspondiente.

Para evaluar una petición HTTP el servidor usa la clase *RequesteEvaluator* que está definida en las utilidades, esta clase se encarga de recibir una petición y de evaluar que se cumple las reglas para considerarla válida. Para que se procese la petición se debe de cumplir una serie de restricciones:

1. Dentro de la petición POST se debe de enviar un documento adjunto.
2. El archivo de tener una extensión válida: *docx*, *txt*, *pdf*, *html*, *csv* y archivos *Excel*.
3. El tipo de dato personal a buscar debe ser válido: nombres, carnés de identidad o ambos.

Fuera de dichas restricciones el servidor enviará un código de estado 400 (Mozilla. 2020) [24], indicando que existe un error en la petición. La clase devuelve un diccionario con tres elementos, por un lado, el nombre del fichero, el resultado de la evaluación y el error asociado a la petición si existe. Todas las peticiones pasan por los mismos pasos para ser resueltas, que serían las siguientes:

1. Evaluar la petición.
2. Si existe un error se devuelve un código 400 (Mozilla. 2020) [24].
3. Si no, se registra el fichero adjunto si lo hubiese y se modifica el nombre del fichero antes de guardarlo en el servidor.
4. Se crea un objeto *documentHandler* con la información del documento y el tipo de operación a aplicar.
5. Si se crean nuevos documentos se registran.
6. Se devuelve el identificador del nuevo recurso creado.

El *endpoint /search/file/download* recoge un identificador, si existe un documento con un identificador público idéntico procede a enviar el documento y a marcar este como enviado.

4.5.6. Frontend

El objetivo al que se quiere llegar con este software es disponer de un *backend* y un *frontend* independientes, por lo que se puede utilizar cualquier tecnología en cualquiera de ellos. En este caso el *backend* utiliza una API REST creada con *Python* usando el microframework *Flask* con la librería *Flask-restplus*, pero para el *frontend* se puede utilizar cualquier otro framework como podrían ser *Angular*, *React*, *php* o incluso crear una aplicación móvil con *Android* o *IOS*, al estar totalmente desacoplado del servicio web mientras se utilice la API de este.

Como no es un requisito indispensable en este TFG el desarrollo de la interfaz, hemos escogido utilizar *Flask* junto a tecnologías como *Bootstrap, SCSS y jQuery* por su sencillez y porque se puede obtener resultados satisfactorios relativamente rápido usando este *stack* de programación. Se ha montado un servidor *Flask* muy sencillo con la estructura que se muestra en la tabla 4.2.

Archivo/Carpeta	**Propósito**
.vscode/	Este directorio contiene los archivos necesarios para que visual studio sirva de editor del proyecto
demos/	Este directorio se usa para almacenar los ejemplos de documentos para testear el software
static/	Este directorio los archivos estáticos del *frontend*, como son: archivos JavaScript, archivos SCSS y las fuentes para las imágenes
templates/	Este directorio contiene todos los archivos *html*
run.py	Script para iniciar el servidor

Tabla 4.2: Estructura del *frontend* de la aplicación

En las siguientes figuras se muestra cómo se quedó el prototipo de la interfaz, hemos trabajado también en realizar una interfaz que se adapte bien en dispositivos móviles. En un entorno en producción es muy probable que se quiera emplear otras tecnologías más orientadas al desarrollo de interfaces gráficas en web como *Angular* o *React*.

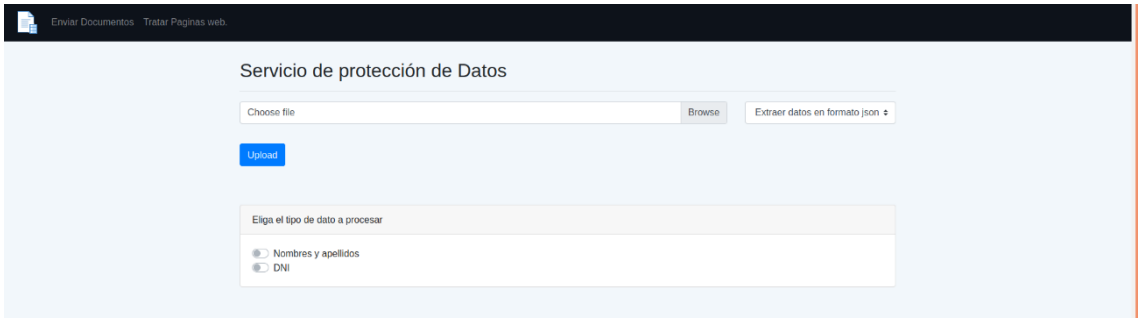


Figura 4.30:Ejemplo de la interfaz en pantallas grandes.

Figura 4.31: Ejemplo de la interfaz en dispositivos móviles.

4.6 Pruebas y Despliegue

A la hora del desarrollo se ha planteado un enfoque ágil centrándose en las pruebas automatizadas. Se ha llevado a cabo un diseño *test-frist* por el cual la primera fase del desarrollo fue escribir las pruebas y luego escribir las clases y los algoritmos necesarios. *Flask-restPlus* dispone de un módulo dedicado al *testing* llamado *flask_testing* donde el conjunto de herramientas y métodos se agrupan en la clase *TestCase*: su función principal es garantizar una base de datos y un servidor para hacer pruebas unitarias del código aparte de los típicos métodos *assert* de estas. Cualquier clase que herede de *TestCase* será una clase de prueba y por tanto cuando se ejecuten las pruebas se lanzarán todas las clases que hereden de esta, también será necesario la librería *unittest* para poder ejecutar las pruebas en la máquina. Como se puede apreciar con *Python-Flask* las pruebas unitarias son muy sencillas de hacer, y nos proporciona capacidad para diseñar el software de manera segura permitiendo cualquier tipo de refactorización o cambio teniendo todo un sistema de respaldo para garantizar el buen comportamiento del servicio.

Como se vio en la tabla 4.1 de la sección 4.5.2, las pruebas se encuentran dentro del fichero *app/test*, en la siguiente tabla se exponen cada uno de los archivos que se van a ejecutar cuando se lancen las pruebas y su función.

Archivo	**Función**
base.py	Crea la base de datos y el servidor de pruebas.
fileVariables.py	Conjunto de constantes para realizar las pruebas
testConfig.py	Testea las diferentes configuraciones del servidor: pruebas, desarrollo y despliegue.
testDocumentHandler.py	Testea los diferentes métodos de los manejadores de documentos.
testFileLog.py	Testea las operaciones a la base de datos del servicio.
testFileUtils.py	Testea las utilidades del archivo *fileUtils.py*

testLanguageBuilder.py	Testea las operaciones de la clase *LanguageBuilder*
testPerfomance.py	Testea el rendimiento de los modelos y técnica de búsqueda de entidades, así como del servicio.
testPersonalDataRules.py	Testea las operaciones de la clase *PersonalDataRules*
testPersonalDataSearch.py	Testea las operaciones de la clase *PersonalDataSearch*
testPicker.py	Testea las operaciones de la clase auxiliar para analizar tablas en algunos documentos.

Tabla 4.3: Explicación de las pruebas unitarias.

A la hora de desplegar el servicio en una máquina, lo primero que se debe tener instalado es *Python* con la versión 3.6 junto con todas las dependencias, que se encuentran en el archivo *requirements.txt*, estas pueden ser instaladas con *pip*, *conda* o cualquier otro gestor de paquetes *Python* tanto en un entorno virtual como en la propia máquina. Para ejecutar las pruebas se utiliza el comando *Python3 manage.py test*, para iniciar el servicio se usa el comando *Python3 manage.py run*. Estos comandos son independientes del sistema operativo, pero como la mayoría de los servicios web se encuentran en un entorno Unix, y este ha sido el principal entorno donde hemos desarrollado el proyecto, se ha creado el script *setup.sh* que sirve para ejecutar tanto las pruebas como para poner en marcha el servicio, así como también hacer limpieza de la base de datos.

Hay que tener en cuenta que al ser esto un prototipo y un trabajo académico se ha configurado el proyecto para que el servicio utilice la red interna de la máquina donde se ejecute, estableciendo su *endpoint* en el *localhost.* Si se va a pasar a producción habría que primeramente obtener un dominio y un *endpoint* público y cambiar este en el código indicando su nueva dirección o utilizar alguna aplicación como *ngrok* o *gunicorn*.

Capítulo 5 Experimentos y Resultados

En este capítulo se describen los diferentes experimentos realizados en el proyecto, por un lado, se va a probar la eficacia de las técnicas utilizadas con respecto a la anonimización, ofuscación o disgregación de datos en diferentes documentos con el objetivo de conocer la bondad del software en su tarea principal y más útil, por otro lado también es interesante conocer el tiempo que garantiza el servicio en buscar las entidades de un documento y modificar el mismo, esto es esencial dado que el objetivo es tener un sistema en tiempo real. Además, se ha realizado un estudio de los falsos positivos y negativos para conocer en que contextos, oraciones, frases o tipos de documentos suelen ocurrir y para tener una información más fiable a la hora de proponer mejoras o líneas futuras. En suma, este capítulo busca contestar a dos preguntas fundamentales: ¿Cómo de buenas son las técnicas planteadas?, y ¿Cuánto tarda el sistema en cumplir su función?

5.1 Pruebas de rendimiento

Estas se centran en medir la eficacia de las técnicas de reconocimiento de entidades nombradas aplicadas únicamente a nombres y apellidos, decidimos hacerlo así debido a que un carné de identidad es una entidad formal con una forma de escribirse muy concreta que la expresión regular recoge adecuadamente, y como no es una técnica en el que haya algún modelo estadístico o de aprendizaje no tiene mucho sentido realizar experimentos y sacar métricas. Con respecto a la métrica, se ha utilizado una bastante clásica en procesamiento del lenguaje natural, la *F-score* [2], también es muy común en algoritmos de búsqueda, recuperación de información o clasificación de documentos. Esta métrica mide la precisión que tiene una prueba, se calcula como la media armónica entre la precisión y la exhaustividad, estas dos medidas se encargan, utilizando los verdaderos positivos, los falsos positivos y los falsos negativos, de determinar del conjunto de entidades cuántas se ajustan al término buscado, en este caso nombres y apellidos, y cuantas de dichas entidades son realmente lo que se quiere extraer. En otras palabras, la precisión mide la cantidad de información que es relevante para la consulta, y la exhaustividad mide la fracción de información que se ha recuperado con éxito como se indica en (Brownlee, J. 2020) [15]. Usando las ecuaciones (5.1), (5.2) y (5.3) se calcula el valor de estas métricas.

$$F_1 = \frac{P - R}{(P + R)} \quad (5.1)$$

$$P_{precisión} = \frac{verdaderos\ positivos}{verdaderos\ positivos + falsos\ positvos} \quad (5.2)$$

$$R_{exhaustividad} = \frac{verdaderos\ positivos}{verdaderos\ positivos + falsos\ negativos} \quad (5.3)$$

Como conjunto de datos de prueba para datos no estructurados hemos seleccionado: 10 cartas o textos formales relacionadas con temas jurídicos y

administrativos y 10 páginas webs diferentes formadas por blogs o artículos periodísticos, y como ejemplo de datos estructurados: 10 hojas de cálculo que representan distintos conjuntos de datos. Los experimentos consisten en comparar las dos formas de extracción de entidades que se han descrito en las secciones anteriores para las cartas y las páginas web, como siempre se va a usar el modelo *es_core_news_md* las pruebas consisten en variar los valores heurísticos y medir su impacto en el rendimiento. Con respecto a las tablas las pruebas estarán en la misma línea, pero variando sus heurísticas correspondientes. Todos los valores extraídos están representados como porcentajes con una precisión máxima de tres decimales.

5.1.1 Datos no estructurados

Prueba 1: Sin un diccionario de respaldo

En esta primera prueba se evalúan las técnicas sin usar una base de conocimiento, simplemente se mide la bondad de la red convolucional de *SpaCy* y el sistema basado en reglas.

	NER con *SpaCy*		**NER basado en reglas**	
Métricas	**Texto**	**Web**	**Texto**	**Web**
P	72,093	38,316	41,666	12,57
R	93,939	87,128	90,909	85,148
F1	81,578	53,225	57,142	21,906

Tabla 5.1:: Resultados sin un diccionario de respaldo.

Como se puede ver en la tabla 5.1 se han probado las técnicas para las cartas y para las páginas web, en este caso la red neuronal supera al sistema en reglas en los dos tipos de documentos logrando un 81,6% y un 53,2% frente a un 57,1% y un 21,9%, aunque es importante destacar que en términos de exhaustividad ninguno de los dos baja del 80% obteniendo para cualquier documento un buen resultado pero siendo muy bajo en la precisión. El parámetro principal en la precisión son los falsos positivos, por tanto, podemos concluir que sin una técnica que verifique si una entidad está mal o bien etiquetada, tanto para un enfoque basado en reglas como para uno basado en aprendizaje se van a incurrir en una explosión de falsos positivos que es más palpable en los documentos web, pues su lenguaje e información es más general que en los textos formales que hemos seleccionado.

Prueba 2: Con un diccionario de respaldo y un umbral de coincidencia del 25%

En este caso existe un diccionario con un gran conjunto de nombres y apellidos almacenados, y al menos el 25% de las palabras que forman una entidad deben de estar recogidas en el diccionario.

	NER con *SpaCy*		NER basado en reglas	
Métricas	**Texto**	**Web**	**Texto**	**Web**
P	81,578	77,744	65,217	34,503
R	93,939	86,468	90,909	84,323
F1	87,323	81,875	75,949	48,969

Tabla 5.2: Resultados para datos no estructurados con un umbral de coincidencia del 25%.

Gracias al diccionario gran parte de los falsos positivos se ven corregidos y ha ayudado a la red neuronal a aumentar su precisión por encima del 75%, pero en caso del sistema basado en reglas todavía su precisión no mejora para alcanzar a la red. Con respecto a la exhaustividad vemos que se mantiene muy similar a la prueba anterior, pero en el caso de la web ha disminuido ligeramente.

Prueba 3: Con un diccionario de respaldo y un umbral de coincidencia del 50%

	NER con *SpaCy*		NER basado en reglas	
Métricas	**Texto**	**Web**	**Texto**	**Web**
P	83,783	81,299	66,666	37,381
R	93,939	84,653	90,909	84,323
F1	88,571	82,942	76,923	51,799

Tabla 5.3: Resultados para datos no estructurados con un umbral de coincidencia del 50%.

Se aprecia que aumentando el umbral de coincidencia mejora la precisión, pero baja la exhaustividad ligeramente, aunque no es apreciable en textos si lo es en web, y esto es debido a que en las páginas web la información suele ser más general y se puede encontrar una mayor variedad de información de contexto que puede afectar a los sistemas de búsqueda. No obstante, vemos que con la red para textos se ha llegado a un 88,57% y a un 83% en web, también la mejora del sistema de reglas es relevante, pero a pesar de aumentar el valor heurístico sigue habiendo una gran cantidad de falsos positivos.

Prueba 4: Con un diccionario de respaldo y un umbral de coincidencia del 75%

	NER con *SpaCy*		NER basado en reglas	
Métricas	**Texto**	**Web**	**Texto**	**Web**
P	86,111	83,606	72,5	42,972
R	93,939	75,742	87,87	78,712
F1	89,855	79,48	79,452	55,594

Tabla 5.4: Resultados para datos no estructurados con un umbral de coincidencia del 75%.

La precisión mejora cuanto más se parezca el contenido de la calidad del sistema a la base de conocimiento, no obstante, para un umbral de coincidencia del 75% la exhaustividad se ve disminuida en los documentos webs. En el caso de los textos aumentando el valor coincidencia aumenta la precisión y esta mejora ligeramente la media armónica del 88,57% al 89,85% para la red neuronal. En términos generales la métrica *F-score* mejora significativamente en el sistema de reglas, pero esto es debido a que cada vez que se tenga más en cuenta el diccionario, si la técnica genera un gran número de falsos positivos, el impacto de este sobre el resultado también será mayor.

Prueba 5: Con un diccionario de respaldo y un umbral de coincidencia del 90%

	NER con *SpaCy*		**NER basado en reglas**	
Métricas	**Texto**	**Web**	**Texto**	**Web**
P	88,571	83,455	74,358	43,169
R	93,939	74,917	87,87	78,217
F1	91,176	78,956	80,55	55,633

Tabla 5.5: Resultados para datos no estructurados con un umbral de coincidencia del 90%.

Para textos en un dominio del lenguaje concreto y formal como son los textos usados, el diccionario con este umbral de coincidencia funciona muy bien afectando positivamente tanto a la exhaustividad como a la precisión para ambas técnicas, pero en textos más generales con un dominio de la lengua irregular la exhaustividad se puede ver afectada con este valor, siendo este fenómeno más significativo en el sistema basado en aprendizaje que en el de reglas, no obstante vemos como el primero supera al segundo por un amplio margen.

Visión General

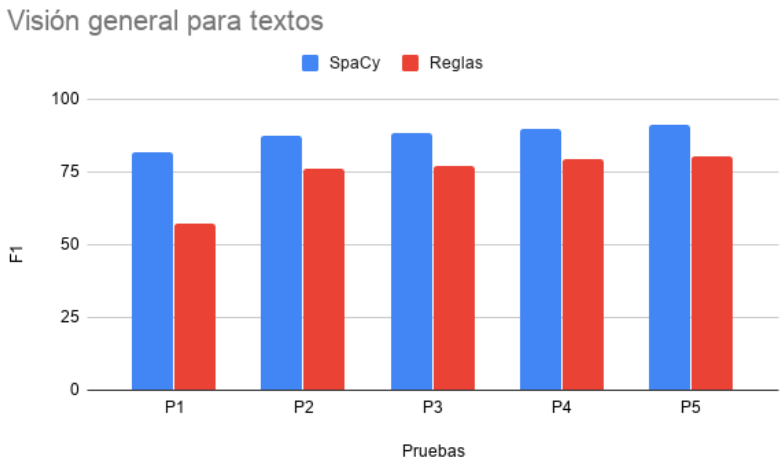


Figura 5.1: Gráfica que muestra una comparativa con todas las pruebas hechas con datos no estructurados sacados de cartas.

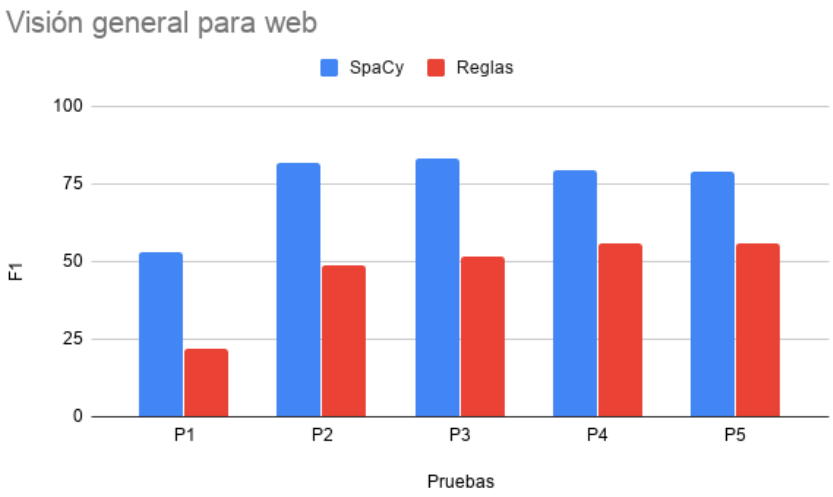


Figura 5.2: Gráfica que muestra una comparativa con todas las pruebas hechas con datos no estructurados sacados de la web.

En las figuras 5.1 y 5.2 se aprecia la relación de las pruebas con ambas técnicas teniendo en consideración la métrica *F-score*, lo primero a tener en cuenta es que la búsqueda de entidades abstractas, en este caso de nombres y apellidos, mediante una aproximación en reglas se queda por atrás en comparación a utilizar el modelo entrenado que nos ofrece *SpaCy*, sin embargo, en términos de exhaustividad las dos técnicas son muy similares, el problema está en la precisión. Como se vio en las pruebas utilizando reglas, a pesar de ser un sistema híbrido cuyo análisis morfosintáctico se realiza mediante redes neuronales, estas degeneran en muchos falsos positivos que incluso con un diccionario tan amplio como el utilizado no se supera del todo está carencia, siendo este fenómeno mucho más visible en webs donde el lenguaje es más coloquial y genérico que en textos formales.

Otra característica interesante es la influencia que tiene el dominio del lenguaje al que las técnicas se van a enfrentar sobre el resultado. Por un lado, para los textos formales la inferencia de ambas técnicas es mejor sin necesidad de un diccionario; vemos como en la primera prueba la red de *SpaCy* llegó a lograr estar por encima del 75% y el sistema de reglas casi llega a un 60%, al empezar a utilizar la base de conocimiento se fueron obteniendo mejores resultados conforme se obligaba a las entidades a parecerse más a la información almacenada en el diccionario de nombres y apellidos, hasta lograr con la salida de la red de *SpaCy* un 91,17% y mediante reglas un 80,5%. Por otro lado, si utilizamos textos con un lenguaje mucho más general, en mayor cantidad y en donde la información se nos puede presentar de varias maneras diferentes como es el caso de la web, la utilidad del diccionario es mucho más visible, sin él las técnicas son muy pobres debido a la precisión y a la cantidad de falsos positivos que generan: la red de *SpaCy* tan solo era capaz de lograr un 53,2% y con reglas se quedaba en un 21,9%, si se usa el diccionario la precisión mejora bastante y lleva al sistema a estar por encima del 80% con la red y a lograr con reglas a superar un 50%. Sin embargo, no ocurre como con las cartas

o textos formales donde aumentando el umbral de coincidencia mejoran los resultados, si se restringe mucho la salida de los modelos a parecerse demasiado al diccionario puede llegar a que una bajada de la exhaustividad, debido a que hay nombres y apellidos que se componen de palabras que no están en la base de datos utilizada, con lo que hay que controlar el umbral de coincidencia para que no afecte negativamente al conjunto de entidades extraídas.

5.1.2 Datos estructurados

Para el tratamiento de las tablas no se usa ningún modelo estadístico ni clasificador, sin embargo, se utilizan una serie de heurísticas muy concretas. El objetivo de estas pruebas es conocer el valor o el rango de valores más adecuados para lograr los mejores resultados posibles para resolver el problema. Entre las pruebas utilizadas se encuentran varios conjuntos de datos con nombres de personas, de empresas, organizaciones, lugares, calles o direcciones, datos numéricos, alfanuméricos …. Se sigue usando la métrica *F-score* junto a las mediciones de precisión y exhaustividad con las mismas ecuaciones para su cálculo. Como para este tipo de datos se utiliza un modelo de lenguaje para buscar similitudes en las cabeceras de las tablas, si no coinciden con la lista de sinónimos alguna columna que sí almacene nombres, toda esta se convertirá en falsos negativos, dentro de todas estas tablas no hemos encontrado esta incidencia. Se ha usado la siguiente lista de palabras: persona, nombre, apellido, alumno, personal, sujeto y candidato. Además, para todas las pruebas se ha dejado el umbral de coincidencia con el diccionario al 50%.

Métricas	**Prueba 1**	**Prueba 2**	**Prueba 3**	**Prueba 4**	**Prueba 5**	**Prueba 6**
P	58,57	61,61	74,91	74,91	93,76	100
R	100	100	100	100	100	100
F1	73,87	76,24	85,66	85,66	96,78	100

Tabla 5.6: Resultados de las métricas de las pruebas realizadas con datos no estructurados.

Prueba 1: Únicamente utilizando un diccionario, sin ningún otro tipo de heurística.

En la tabla 5.6 se puede ver que para esta prueba se logra tan solo un 58,57 de precisión, siendo los falsos positivos el principal problema. Tan solo utilizando un diccionario y comprobando cada celda con él, vemos que es una estrategia que genera muchos falsos positivos, aunque en términos de exhaustividad sea muy buena.

Prueba 2: Obligando a un 25% de las celdas de una columna que coincidan con el diccionario.

En este caso se contabilizan para cada columna cuántas celdas coinciden con el diccionario y si superan un determinado umbral, en este caso el 25%, se elige la columna

entera. Con este enfoque se logra aumentar ligeramente la precisión hasta llevarla a un 61,6%, llevando a la media armónica a un 76,24%.

Prueba 3: Obligando a un 50% de las celdas de una columna que coincidan con el diccionario.

Aumenta ligeramente los resultados logrando un 75% en precisión y sobre un 85% en la media *F-score*.

Prueba 4: Obligando a un 75% de las celdas de una columna que coincidan con el diccionario y buscando similitudes con las cabeceras de las columnas y una lista de palabras siendo este porcentaje de similitud mayor al 50%.

Vemos en la tabla 5.6 que no tiene ningún impacto significativo en los resultados con respecto a la prueba anterior el haber usado una búsqueda de sinónimos de las cabeceras con un valor del 50%.

Prueba 5: Obligando a un 75% de las celdas de una columna que coincidan con el diccionario y buscando similitudes con las cabeceras de las columnas y una lista de palabras siendo este porcentaje de similitud mayor al 75%.

Aquí la mejora es muy significativa se logra una precisión del 93,76% y una media con la exhaustividad del 96.78%.

Prueba 6: Obligando a un 80% de las celdas de una columna que coincidan con el diccionario y buscando similitudes con las cabeceras de las columnas y una lista de palabras siendo este porcentaje de similitud mayor al 75%.

Aquí los resultados se adaptan a la perfección a los datos, así pues, para este conjunto de datos de pruebas se logra un 100% y para otros similares también debería obtener resultados igual de buenos o muy parecidos.

Visión General

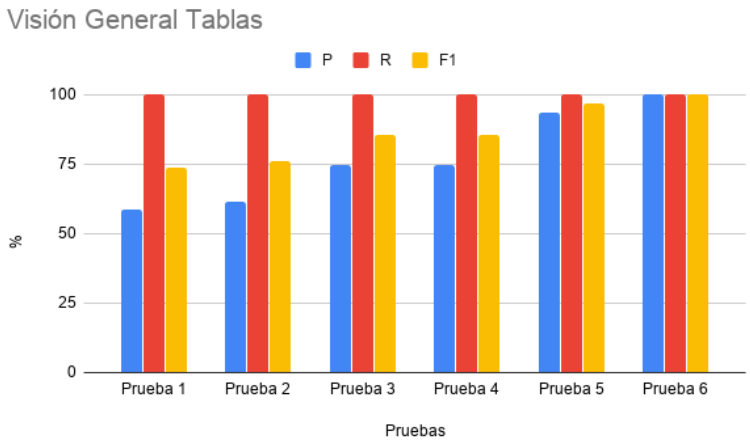


Figura 5.3: Gráfica que muestra una comparativa con todas las pruebas hechas con datos estructurados.

Es evidente que todas las heurísticas utilizadas son buenas y con el valor adecuado pueden obtener muy buenos resultados, vemos que filtrar las columnas por sus cabeceras se traduce en una corrección significativa de los falsos positivos, y que con una técnica de diccionario bastante simple para datos estructurados que presentan una forma muy común con una cabecera, se garantiza la anonimización, ofuscación o disgregación de las entidades en un alto margen.

5.2 Pruebas de tiempo

El objetivo de estas pruebas es medir la efectividad temporal del servicio y poder tener una evidencia de que se trata de un sistema en tiempo real, y con una garantía de que bajo un tiempo decente se puede obtener resultados para cualquier tipo de documento válido. Las pruebas de tiempo consisten en anonimizar, con la función más simple, cien documentos haciendo cien peticiones a la API, es decir una petición por documento, obteniendo el tiempo medido en segundos, que tardó en resolver la petición y calculando: la suma total, la media y la desviación estándar de las peticiones, esto se repite para cada tipo de documento. Al tratarse de un servicio web los resultados se verían afectados teniendo en consideración los protocolos de red y el ancho de banda que el usuario disponga, hemos decidido obviar esta parte y probar el servicio desde el *localhost* de la máquina con el que se hizo las pruebas para así tener una medida que refleje el comportamiento del servicio sin tener en cuenta el tiempo de envió de datos y de respuesta.

En las tablas 5.7 y 5.8 se muestran los datos obtenidos del servicio con las dos técnicas utilizadas: con la red de *SpaCy* y con reglas. Vemos que para textos en ambas técnicas se obtienen tiempos relativamente bajos con una media y desviación buenas. El tipo de documento para los textos que más tiempo consume es el *pdf*, pero esto es natural debido a que las librerías para modificar y leer *pdf* en *Python* tardan en procesar estos ficheros, en el caso de la web las muestras de ejemplo con las que se hicieron las pruebas contenían bastante más información en comparación a los textos utilizados, además que para procesar documentos webs hay que tener en consideración otros aspectos que hacen a los algoritmos ligeramente ineficientes en comparación con ficheros que contienen simplemente texto. Se puede apreciar también cómo usando reglas se consigue reducir a la mitad en tiempo para todos los documentos, demostrando así que el mayor coste computacional viene dado por el modelo de clasificación para detectar entidades de *SpaCy*, sin embargo, es con el que se obtiene mejores resultados y será más rápido que utilizar otros modelos de aprendizaje como podrían ser campos condicionados aleatorios o redes LSTM.

	Txt	Pdf	Docx	Hojas de cálculo	Web
Total	11,42 seg	16,66 seg	11,45 seg	6,16 seg	302,71 seg
media	0,11 seg	0,17 seg	0,11 seg	0,06 seg	3,03 seg
std	0,06	0,05	0,04	0,02	1,4

Tabla 5.7: Mediciones de tiempo en segundos con la red convolucional de *SpaCy*, usando 100 documentos con formatos distintos.

	Txt	Pdf	Docx	Hojas de cálculo	Web
Total	7,29 seg	12,9 seg	7,584 seg	5,053 seg	159,55 seg
media	0,074 seg	0,129 seg	0,076 seg	0,05 seg	1,59 seg
std	0,024	0,0312	0,02	0,0066	0,789

Tabla 5.8: Mediciones de tiempo en segundos con el sistema basado en reglas, usando 100 documentos con formatos distintos.

5.3 Estudio sobre los falsos negativos y positivos

En esta sección se han analizado los fallos que presentan las técnicas de detección de entidades con el conjunto de pruebas, se han centrados los esfuerzos sobre todo en datos no estructurados puesto que para las tablas no se realiza ningún proceso de inferencia mediante el contexto, y la relación de falsos positivos y negativos ya se ha explicado en secciones anteriores. Lo que se pretende con esto es centrar las bases de futuras hipótesis para la mejora o modificación de las técnicas planteadas. Para hacer este estudio se seleccionó un umbral de coincidencia con el diccionario del 50% porque es el más equilibrado tanto para textos como para páginas web.

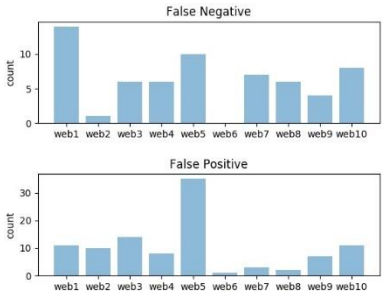


Figura 5.5: relación de falsos positivos y falsos negativos en web con la red convolucional de SpaCy.

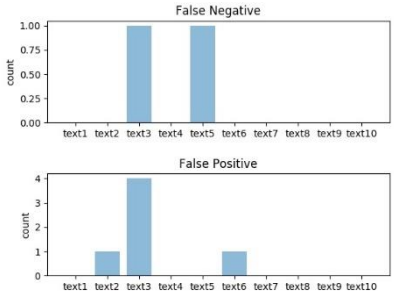


Figura 5.4:relación de falsos positivos y falsos negativos en textos para la red convolucional de SpaCy.

Centrándonos en la red convolucional de *SpaCy* en la figura 5.4 y 5.5 se muestran una distribución de falsos negativos y falsos positivos con respecto a las pruebas de textos y webs realizadas.

En textos existen pocos falsos negativos, se centran en el texto 3 y el texto 5, estos dos son cartas formales para ceder una custodia y una carta formal entre dos dignatarios, falla en dos nombres: uno compuesto "María de Guadalupe Sánchez Vargas" y en "Urbina Reciba", se ha comprobado si las palabras de estos nombres están en el diccionario; en el caso del segundo "Reciba" no lo está, por lo que es probable que se haya descartado ese nombre por no estar lo suficientemente representado en la base de conocimiento, en el caso del primero todas las palabras que componen al nombre están en el diccionario por lo que el modelo no ha conseguido extraerlo como entidad, la frase donde aparece es la siguiente:

"Con la presente y ante el notario público No 4997 del Estado de México, la Sra. María de Guadalupe Sánchez Vargas cede la custodia total de su hermana mayor Lucia Antonia Sánchez Vargas de 77 años de edad, esto debido a que por razones de salud no puede continuar con su atención ni hacerse cargo de sus bienes."

Si analizamos la oración puede deberse a que en la posición donde se manifiesta el nombre no se disponga del suficiente contexto a pesar de que actúe como sujeto, o que como está acompañado de la expresión "Sra." y sea un nombre compuesto confunda al clasificador, si observamos los falsos positivos de la figura 5.5 vemos que el texto 3 es donde más abundan y "María" es uno de ellos, por lo que el hecho de ser un nombre compuesto ha confundido al modelo. En relación únicamente a los falsos positivos, los más relevante son sustantivos en mayúsculas que actúan como sujetos acompañados de verbos, estos pasan el diccionario porque están compuestos por palabras que también son nombres, o palabras que acompañan a un nombre pero que no forman parte de él, un ejemplo:

"Causa por la cual cede todas las obligaciones y cuidados a su sobrino-nieto Jesús Fernández Fuentes a quien le otorgará una prima económica mensual de 1,500.00 (mil quinientos pesos) para su atención y sustento cubriendo el resto de la atención y manutención el Sr Jesús Fernández Fuentes que es igualmente familiar y obligado de la Sra. Lucía Antonia Sánchez Vargas."

En esta oración "Sr Jesús Fernández Fuentes" es seleccionado como un nombre, el apóstrofe "Sr" no tiene suficiente información para que el clasificador consiga separarlo de la entidad, pero en la misma oración con la expresión "Sra. Lucía Antonia Sánchez Vargas" si consigue separar el "Sra." del nombre, esto ocurre porque el punto da la información necesaria para separarlo del nombre.

En el caso de la web, al ser más extensos los documentos de pruebas, el análisis se centra más en aspectos generales. En la figura 5.5, la web 1 es la que presenta con mucha diferencia la mayor cantidad de falsos negativos, esta web es una página en Wikipedia que habla sobre la "unión de centro democrático", se caracteriza por tener mucha información sin contextos con nombres compuestos y repartidos en listas y tablas, es normal que a falta de un contexto muchos nombres se pierdan, o el clasificador y los algoritmos no logren localizar todas las entidades, en el resto de ejemplos los principales

problemas están: en nombres compuestos con caracteres como “-” o “y”, en nombres que no tienen suficiente contexto para poder ser inferidos posicionados en enlaces o en listas, y algunos que no llegan a cumplir el requisito del umbral con la base de datos. En relación a los falsos positivos la que más destaca es la web 5, otra página en Wikipedia, en este caso trata de “José Juan Rodríguez Fernández” un matemático español del siglo 19, donde la mayoría de nombres son compuestos y aunque la información sea fácil de trata y en su mayoría viene a acompañada de un contexto, muchos de estos nombres no son clasificados como tales de manera completa, así mismo en otros ejemplos se manifiesta de manera paulatina un gran número de falsos positivos producidos por sustantivos en mayúsculas normalmente acompañados con verbos.

En el caso del sistema basado en reglas, como se utilizan patrones para buscar los nombres los fallos que pueden ocurrir son más abundantes, sobre todo en falsos positivos, puesto que esta aproximación se basa en generalidades para buscar las entidades que otras expresiones también cumplen, y muchas de ellas están compuestas por palabras recogidas en la base de datos de nombres y apellidos al ser esta de un gran tamaño. En la figura 5.7 el principal problema para los falsos negativos son los nombres completos o muy largos, los cuales no encajan del todo bien con el patrón de búsqueda, por ejemplo, el texto 2 trata de una carta de custodia para dos hijos, una de las frases es la siguiente:

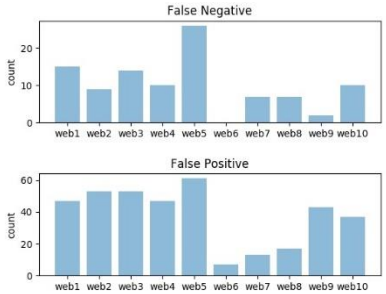


Figura 5.6:relación de falsos positivos y falsos negativos en web para el sistema de reglas.

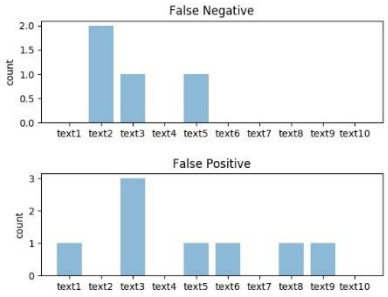


Figura 5.7: relación de falsos positivos y falsos negativos en textos para el sistema de reglas.

“Quien suscribe María Antonieta Mendieta Castro, con la presente solicito la custodia permanente de mis menores hijos de nombres Antonio y Javier de apellidos Gonzales Mendieta quienes se encuentran bajo la custodia de su padre desde que se consumó el divorcio hace ya tres años.”

En esta frase los patrones cuadran con “Antonio” pero dejan a “Javier” a causa de que viene a acompañado de la proposición “y”. En relación con los falsos positivos suelen deberse a las mismas razones que con la técnica anterior: nombres muy largos o compuestos y sustantivos en mayúsculas acompañados de verbos, esto se cumple para los textos y las páginas webs, siendo la distribución de falsos positivos en la figura 5.6 más homogénea. En la figura 5.6 también se puede ver cómo es la web 5 la que más falsos negativos se obtiene, debido a las mismas razones que con el anterior método, la cantidad de nombres compuestos.

En conclusión, las dos técnicas generan fallos similares, los cuales son: nombres compuestos o demasiado largos, el valor del umbral de coincidencia puede aumentar el número de falsos negativos, sustantivos con mayúsculas y que actúan como nombres, nombres acompañados con expresiones extrañas y la falta de información de contexto.

Capítulo 6 Conclusiones

Con este trabajo se ha logrado cumplir con los objetivos inicialmente planteados. Tal y como se ha especificado al comienzo de esta memoria, en primer lugar, se ha llevado a cabo un estudio sobre las técnicas de procesamiento del lenguaje natural orientadas a la identificación y búsqueda de entidades nombradas, y sobre algunas arquitecturas o sistemas para la reducción de identificación de datos que existen actualmente. Una vez obtenidos los conocimientos necesarios, se han utilizado una serie de técnicas y metodologías para la búsqueda de entidades en textos, con el objetivo de diseñar un sistema inteligente híbrido para la extracción y reducción de riesgo de identificación de nombres, apellidos y carnés de identidad, usando una combinación de técnicas basadas en ingeniería del conocimiento (técnicas de diccionario, búsqueda mediante patrones, expresiones regulares y heurísticas simples) y modelos de aprendizaje (redes neuronales convolucionales entrenados mediante aprendizaje multitarea), asimismo se ha trabajado sobre las librerías y técnicas que permiten leer desde *Python* diferentes documentos, ya sean con formato(*Word*, *Pdf*, *Excel*), texto plano o incluso páginas web. Todo esto está integrado en un servicio web cuya interacción puede producirse mediante una API o mediante una propuesta de *frontend* que hemos diseñado.

Se ha probado y testeado la fiabilidad de las técnicas utilizadas para diferentes tipos de documentos, en el caso de los carnés de identidad al tratarse de una expresión regular se garantiza al completo su reducción de riesgo para cualquier tipo de documento, en caso de los nombres y apellidos la técnica con la que hemos obtenido mejor resultado es mediante los modelos de aprendizaje que nos ofrece *SpaCy* con un umbral de coincidencia con el diccionario de nombres del 50%, logrando una reducción de riesgo de identificación del 88,6% en textos y del 82,94% en páginas web, en el caso de las hojas de cálculo y las tablas se garantiza la reducción usando los valores heurísticos adecuados. Asimismo, se ha probado la fiabilidad del servicio web obteniendo resultados que demuestran que se trata de un sistema en tiempo real para cualquier documento válido.

6.1 Valoración personal

He elegido este trabajo precisamente porque me permitía aplicar conceptos del desarrollo de sistemas inteligentes y de ingeniería del software aplicados al procesamiento del lenguaje natural, todo un campo que no se trabaja demasiado en el Grado. También porque me permitía acercarme a conceptos propios de la investigación en inteligencia artificial y al desarrollo de una herramienta útil que, a pesar de que se trata de un prototipo, las ideas y los conceptos que plantea son perfectamente válidas para llevar a un desarrollo comercial. Otras de las razones que me llevó a abordar este tema, es que creo firmemente que la tecnología y el avance científico no solo debe ser una herramienta más de la sociedad, sino también garantía de derechos y de seguridad para las personas.

Tras la finalización de este trabajo me siento capacitado para trabajar en proyectos utilizando conocimientos sobre inteligencia artificial, procesamiento del lenguaje natural, en grandes partes de sus ámbitos, y también en temas relacionados con la protección de

datos personales. Me siento afortunado de haber recibido asesoramiento de *Acoran* y de haber recibido el visto bueno de una empresa que se dedica día a día a trabajar sobre este tema. Por último, me gustaría añadir que los conocimientos aprendidos en este TFG me van a servir para lo que venga después de esto, durante mi carrera profesional.

6.2 Trabajos Futuros

Entre los posibles trabajos que pueden derivar de este proyecto, se encuentra el estudio y la mejora de las técnicas de aprendizaje utilizadas, se podrían probar otras arquitecturas o redes neuronales como las LSTM o el reentrenamiento de los modelos utilizados, siendo esto particularmente complicado porque no se dispone de demasiados conjuntos de datos útiles para la búsqueda de entidades nombradas en castellano. Otra línea sería probar otros tipos de técnicas de diccionario, como podrían ser diccionarios inducidos. Actualmente los datos personales que son tratados son nombres, apellidos y DNI, no obstante, las técnicas utilizadas podrían servir para buscar más tipos de datos como números de teléfonos, cuentas bancarias, nombres de ciudades…

Con respecto al software, una de las líneas destinadas a mejorar el servicio web es el empleo de otros frameworks que estén orientados a un desarrollo en producción, como podrían ser *Spring* o *Symfony* en el *backend* y *React* o *Vue* en el *frontend*, asimismo se podría extender la cantidad de tipos de documentos que el servicio puede tratar como podrían ser archivos *.doc*, u otros formatos típicos de *LibreOffice*. Otro de los conceptos más interesantes sería mejorar la integración del sistema con la web, actualmente se necesita pasar una url o un documento *html* para tratar páginas web, pero se podría refactorizar la api para que se lanzará frente a un navegador y se tratasen las páginas sin necesidad de que el usuario interactúe con la api directamente, el ejemplo más directo sería lo que hace *Google* con su traductor que permite cambiar el idioma de una página web directamente desde el navegador.

Bibliografía

[1] Bird, S., Klein, E. & Loper, E. (2009). Natural Language Processing with *Python*. Segunda edición. O'Reilly Media Inc.

URL: *http://www.nltk.org/book_1ed/*

[2] Hassan, F., Domingo-Ferrer,J. & Soria-Comas J. (2018).Anonimización de datos no estructurados a través del reconocimiento de entidades nominadas. pp. 1-5.

[3] Hasanuzzaman, M., Saha, S. & Ekbal, A. (2010). Feature Subset Selection Using Genetic Algorithm for Named Entity Recognition. pp. 1-10.

[4] Hochreiter, S. & Schmidhuber, J. (1997). Long Short-Term Memory. pp. 6-9.

[5] Lafferty, J., McCallum, A. & Pereira, F. (2001). Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. pp. 1-10

[6] Shure, L. (2017, septiembre 21). Math with *Word*s – *Word* Embeddings with MATLAB and Text Analytics Toolbox:

URL: *https://blogs.mathworks.com/loren/2017/09/21/math-with-Words-Word-embeddings-with-matlab-and-text-analytics-toolbox/*

[7] Canny, S. (2013). *Python*-docx:

URL: *https://Python-docx.readthedocs.io/en/latest/*

[8] McKinney, W. (2017). *Python* for data analysis: data wrangling with Pandas, NumPy, and I*Python.* Segunda edición. O'Reilly Media Inc.

[9] Shinyama, Y, Guglielmetti, P & Marsman, P. (2019). Welcome to *pdf*miner.six's documentation!

URL: *https://pdfminersix.readthedocs.io/en/latest/*

[10] Ariga, A. (2019). tabula-py: Read tables in a *PDF* into DataFrame.

URL: *https://tabula-py.readthedocs.io/en/latest/*

[11] Peréz-Porto, J. & Gardey, A. (2019). Definición de *PDF*.

URL: *https://definicion.de/pdf/*

[12] Tauber, J. (2019). *Pdf*-redactor.

URL: *https://github.com/JoshData/pdf-redactor*

[13] Maupin, P., Gandouet, B., Wu, H., Kluyver, T., Devenish, N., Horn, B., Mazulo, P., Brinkmann, M., Steffensen, J. L., Townshend, D., Wah, J., McKie, J. X., Betts, E. (2018). *pdf*rw 0.4.

URL: *https://github.com/pmaupin/pdfrw*

[14] Explosion AI. (2016-2020). *SpaCy* 101: Everything you need to know.

URL: *https://SpaCy.io/usage/SpaCy-101*

[15] Brownlee, J. (2020, enero de 3). How to Calculate Precision, Recall, and F-Measure for Imbalanced Classification.

URL:*https://machinelearningmastery.com/precision-recall-and-f-measure-for-imbalanced-classification/*

[16] Olah, C. (2015, Agosto 17). Understanding LSTM Networks:

URL: *https://colah.github.io/posts/2015-08-Understanding-LSTMs/*

[17] McCormick, C. (2016, abril 19). *Word*2Vec Tutorial - The Skip-Gram Model.

URL:*http://mccormickml.com/2016/04/19/Word2vec-tutorial-the-skip-gram-model/*

[18] Guerra-Artal, C. Algoritmo de aprendizaje del perceptrón:

URL:*https://nbviewer.jupyter.org/url/cayetanoguerra.github.io/ia/nbpy/learning-perceptron.ipynb*

[19] Vieira, S., Pinaya, W. & Mechelli, A. (2017). Using deep learning to investigate the neuroimaging correlates of psychiatric and neurological disorders: Methods and applications. Neuroscience & Biobehavioral Reviews. pp. 2-19

[20] Instituto nacional de estadística. (2018). Apellidos y nombres más frecuentes.

URL:*https://www.ine.es/dyngs/INEbase/es/operacion.htm?c=Estadistica_C&cid=1254736177009&menu=ultiDatos&idp=1254734710990*

[21] Saha, S. (2018, diciembre 15), A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way:

URL: *https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53*

[22] Microsoft. (2018, enero 12). Diseño de API web.

URL:*https://docs.microsoft.com/es-es/azure/architecture/best-practices/api-design*

[23] Singer-Vine, J., Jain, S., Fenton, J., Weweler, Y., Jain, S., Wilk, J., Shorstein, D., Cheung, K., Lannon, B., Moran, O. (2020). *PDF*Plumber v0.5.21.

URL: *https://github.com/jsvine/pdfplumber*

[24] Mozilla. (2005-2020). Códigos de estado de respuesta HTTP.

URL: *https://developer.mozilla.org/es/docs/Web/HTTP/Status*

[25] Jurafsky, D. & Martin, J. H. (2019). Speech and Language Processing (3rd ed. draft). pp. 94-324.

URL: *https://web.stanford.edu/~jurafsky/slp3/*

[26] Wong, W., Martinez, D. & Cavedon, L. (2009). Extraction of Named Entities from Tables in Gene Mutation Literature. pp. 1-9.

[27] Yildiz, B., Kaiser, K. & Miksch, S. (2005). *pdf*2table: A Method to Extract Table Information from *PDF* Files. pp. 3-13.

[28] Russell, S. J. & Norvig, P. (2004). Inteligencia Artificial: un enfoque moderno, number 04; Q335, R8y 2004.

[29] Lample, G., Ballesteros, M., Subramanian, S., Kawakami K. & Dyer C. (2016). Neural Architectures for Named Entity Recognition. pp 2-7.

[30] Agencia Española De Protección De Datos. (2016). Orientaciones y garantías en los procedimientos de anonimización de datos personales.

[31] España, Real Decreto 1553/2005, de 23 de diciembre, Artículo 11. Boletín Oficial del Estado, 23 de diciembre de 2005:

URL:*http://www.interior.gob.es/web/servicios-al-ciudadano/normativa/reales-decretos/real-decreto-1553-2005-de-23-de-diciembre#art11*

[32] Mandal, J. & Mukhopadhyay, S. (2020). Proceedings of the Global AI Congress 2019. 10.1007/978-981-15-2188-1.

[33] Kiss, T. & Strunk, J. (2006). Unsupervised Multilingual Sentence Boundary Detection. pp 1-41.

[34] Russ, C. (2007). Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, *Python*, Ruby, ...)

URL: *https://swtch.com/~rsc/regexp/regexp1.html*

[35] Arevalo, M., Carreras, X., Márquez, L., Martí, M. A., Padró, L. & Simón, J. (2002). A Proposal for Wide-Coverage Spanish Named Entity Recognition. Procesamiento del Lenguaje Natural. pp 63-80.

[36] The jQuery Foundation. (2020). What is jQuery?

URL: *https://jquery.com/*

[37] W3C. (2020). World Wide Web Consortium.

URL: *https://www.w3.org/*

[38] Richardson, L. (2004-2020). Beautiful Soup Documentation.

URL: *https://www.crummy.com/software/BeautifulSoup/bs4/doc/*

[39] Nurminen, A. (2013). Algorithmic Extraction Of Data In Tables In *Pdf* Documents, pp. 18-44

[40] Vico, H. (2013). Arquitectura de Referencia para Anonimizar Documentos, pp. 8-23

[41] España. Ley Orgánica 3/2018, de 5 de diciembre, de Protección de Datos Personales y garantía de los derechos digitales. Boletín Oficial del Estado, 5 de diciembre de 2018.

URL: *https://www.boe.es/eli/es/lo/2018/12/05/3*

[42] ONU: Asamblea General, *Declaración Universal de Derechos Humanos*, 10 diciembre 1948, 217 A (III).

URL: *https://www.un.org/es/universal-declaration-human-rights/*

[43] Obinna, G. (2018). How to structure a Flask-RESTPlus web service for production builds.

URL:*https://www.freecodecamp.org/news/structuring-a-flask-restplus-web-service-for-production-builds-c2ec676de563/*

ANEXO

ANEXO I: Manual de Usuario

Ejecución

Para ejecutar el proyecto se necesita tener instalado *Python*3.6, debido a que algunas dependencias no van bien con las recientes actualizaciones de *Python* al menos en el momento del desarrollo de este software.

Instalar Dependencias

Ya sea en un entorno virtual o no, lo primero que se tiene que hacer es instalar las dependencias del archivo *requirements.txt*. Se puede hacer con *pip* o cualquier otro gestor de paquetes.

```
git clone https://github.com/miguel-kjh/NameSearcher-WebService
cd NameSearcher-WebService/backend/
pip install -r requirements.txt
```

Ejecutar el servicio para Windows, Mac o Linux

Por defecto se abre en el puerto 5000, pero esta opción se puede modificar en el fichero *manage.py*

```
Python backend/manage.py run  # Ejecuta el servicio
Python backend/manage.py test # Ejecuta las pruebas
Python backend/cleanDB.py     # Limpia la BD
```

Ejecutar el *frontend* para Windows, Mac o Linux

Por defecto se abre en el puerto 5001, pero esta opción se puede modificar en el fichero *run.py*

```
Python frontend/run.py # Ejecuta el frontend
```

Migrar la base de datos

```
Python manage.py db migrate --message 'new schema'
Python manage.py db upgrade
```

Usar *gunicorn* para abrir el servicio en producción

Los comandos anteriores sirven para ejecutar y desplegar tanto el *backend* como el *frontend* desde el servidor que nos ofrece *Flask*, sin embargo, este servidor no está optimizado para un entorno en producción. Por contra se recomiendo utilizar *Gunicorn*. *Gunicorn* es una herramienta que nos ofrece un entorno en producción para servidores web en *Python*, además de que ya viene incluido en las dependencias del proyecto.

```
gunicorn --bind ip_de_la_máquina:5000 manage:app --timeout 300 &
gunicorn --bind ip_de_la_máquina:5001 run:app &
```

API REST

- ***/search/version:*** Devuelve mediante GET la versión actual del software.

- ***/search/file/encode:*** Se encarga de recoger ficheros mediante POST y de devolver otra versión de ellos con los datos personales anonimizados. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento.
- ***/search/file/disintegration:*** Se encarga de recoger ficheros mediante POST y de devolver otra versión de ellos con los datos personales transformados mediante un proceso de disgregación. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento.
- ***/search/file/obfuscation:*** Se encarga de recoger ficheros mediante POST y de devolver otra versión de ellos con los datos personales transformados mediante un proceso de ofuscación. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento.
- ***/search/file/extract-data/json-file:*** Se encarga de recoger ficheros mediante POST y de devolver un fichero *json* con los datos personales encontrados. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento.
- ***/search/file/extract-data/json:*** Se encarga de recoger ficheros mediante POST y de devolver un objeto *json* con los datos personales encontrados. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento.
- ***/search/file/extract-data/csv:*** Se encarga de recoger ficheros mediante POST y de devolver un fichero *csv* con los datos personales. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento.
- ***/search/file/operation-web:*** Se encarga mediante GET de recoger una URL y un tipo de operación de transformación o extracción de datos, y de devolver un archivo *json* o *csv* con los datos o el archivo *html* con las entidades transformadas. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento.
- ***/search/file/tagger-html:*** El último *endpoint* se encarga de recoger ficheros *html* mediante POST y de devolverlos, pero marcados con los datos etiquetados. Recibe mediante la cadena de consulta el tipo de dato que tiene que localizar en el documento. Este recurso está para tener una representación visual de la bondad del sistema en la localización de los datos.

Guía de Usuario

El *frontend* dispone de un apartado donde se le muestra información al usuario sobre el servicio y la ley de protección de datos.

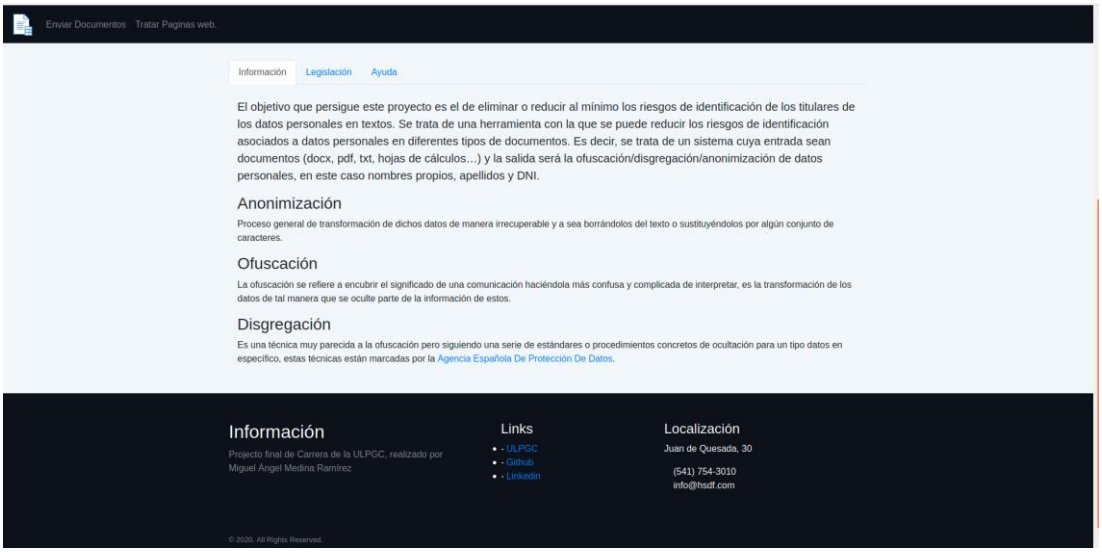


Figura I: Información sobre el servicio y el proyecto

Se permitirá enviar tanto documentos como direcciones a páginas web eligiendo que tipo de datos quiere procesar, toda operación devolverá el id correspondiente del nuevo recurso creado, por lo tanto, una vez finalizada la transferencia el usuario deberá darle al botón de descargar el fichero generado.

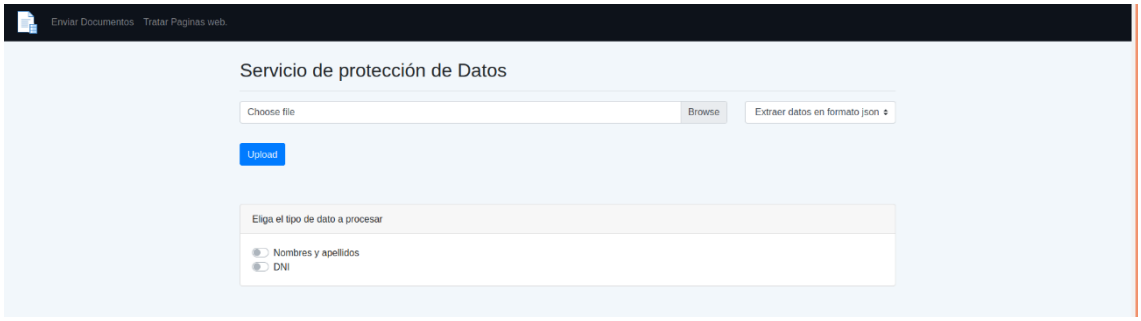


Figura II: Enviar documentos

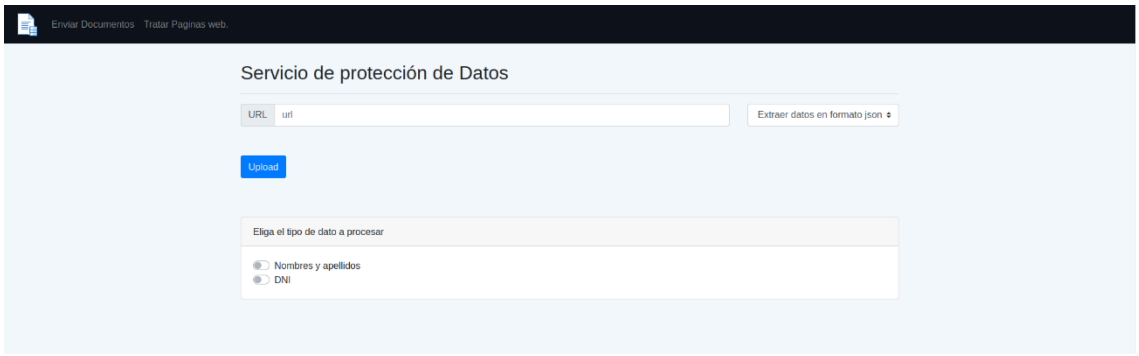


Figura III: Enviar una dirección URL

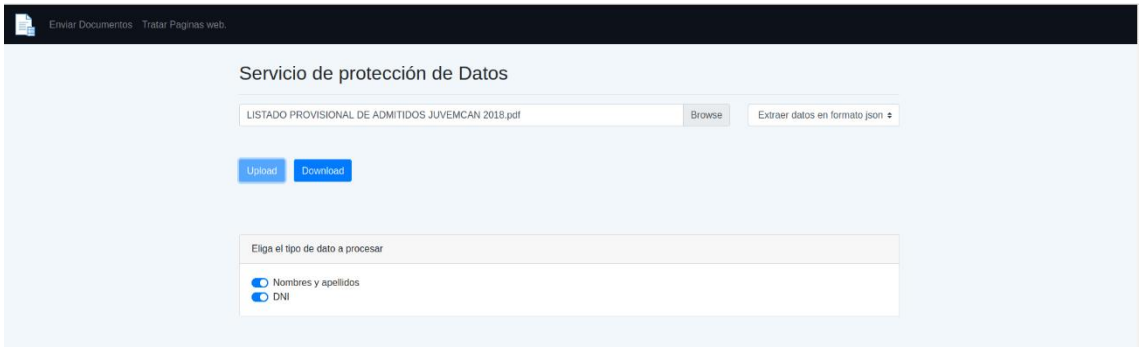


Figura IV: Descargar resultados de peticiones

ANEXO II: Repositorio

Todo el código que se ha desarrollado en este proyecto está en el siguiente repositorio, en donde el *readme* del mismo actúa como manual de usuario también:

https://github.com/miguel-kjh/NameSearcher-WebService

ANEXO III: Diagrama de clases

El diagrama UML está colgado en un recurso de *Google drive* debido a su tamaño, se puede acceder a él mediante el siguiente enlace.

https://drive.google.com/file/d/1uWWVXXgOkb5RTkjAZb-71n1zYxyBja4M/view?usp=sharing