



PROYECTO FINAL DE CARRERA

Desarrollo de un sistema de búsqueda
inteligente de interacciones entre
fármacos y genes en textos científicos
utilizando aprendizaje profundo

Informe Final

Alumnos: Bertinetti, Juan; Ramírez, Darién

Director: Dr. Cristian Yones

Índice general

| | |
|---|-----------|
| 1. Introducción | 1 |
| 1.1. Motivación | 1 |
| 1.2. Objetivos | 8 |
| 1.2.1. Objetivo general | 8 |
| 1.2.2. Objetivos específicos | 8 |
| 1.3. Alcance | 8 |
| 1.3.1. Requerimientos funcionales | 8 |
| 1.3.2. Requerimientos no funcionales | 9 |
| 1.4. Metodología | 10 |
| 2. Marco teórico | 13 |
| 2.1. Aprendizaje automático | 13 |
| 2.2. Aprendizaje profundo | 17 |
| 2.2.1. <i>Embeddings</i> | 19 |
| 2.2.2. Redes convolucionales | 22 |
| 2.2.3. Redes <i>Transformer</i> | 25 |
| 2.2.4. <i>Dropout</i> | 27 |
| 2.2.5. <i>Batch Normalization</i> | 27 |
| 3. Trabajo realizado: Motor de inferencia | 29 |
| 3.1. Análisis de requerimientos | 29 |
| 3.2. Obtención de datos | 30 |
| 3.2.1. Base de datos de etiquetas | 30 |
| 3.2.2. <i>Web scraping</i> | 32 |
| 3.2.3. Pre-procesamiento de los datos | 34 |
| 3.2.4. Generación de casos negativos | 36 |
| 3.2.5. Generación de secuencias de ejemplos | 39 |
| 3.3. Entrenamiento del modelo | 42 |
| 3.3.1. Recursos de hardware empleados | 45 |
| 3.3.2. Red Neuronal Convolucional (CNN) | 45 |
| 3.3.3. Red Neuronal <i>Transformer</i> | 49 |

| | |
|---|-----------|
| 4. Evaluación de los modelos neuronales | 51 |
| 4.1. Hiperparámetros comunes | 51 |
| 4.2. Acierto de los modelos | 52 |
| 4.3. Evaluación del desempeño | 54 |
| 4.4. Selección del modelo final | 60 |
| 5. Trabajo realizado: Aplicación Web | 61 |
| 5.1. Análisis de requerimientos | 61 |
| 5.2. Diseño | 63 |
| 5.2.1. Diseño del <i>front-end</i> | 66 |
| 5.2.2. Diseño del <i>back-end</i> | 67 |
| 5.2.3. Base de datos | 72 |
| 5.3. Codificación | 73 |
| 5.3.1. Submódulo de gestión y autenticación de usuarios . . . | 77 |
| 5.3.2. Submódulo de búsqueda externa | 81 |
| 5.3.3. Submódulo de interacción con el modelo neuronal . . . | 82 |
| 5.3.4. Submódulo de presentación de datos | 83 |
| 5.3.5. Submódulo de añadidos | 84 |
| 5.4. Pruebas y verificación | 91 |
| 6. Conclusiones y trabajos futuros | 97 |
| 6.1. Conclusiones | 97 |
| 6.2. Trabajos futuros | 98 |
| Referencias | 100 |

Capítulo 1

Introducción

En este informe se describe el trabajo realizado para lograr implementar un sistema de búsqueda inteligente de interacciones entre fármacos y genes en textos científicos utilizando aprendizaje profundo. Este sistema está constituido por dos grandes módulos: uno de ellos es el módulo de aplicación web, encargado de permitir al usuario realizar las búsquedas deseadas y responsable de comunicarse con el motor de búsqueda, y por otro lado el módulo neuronal, que es el motor de búsqueda propiamente dicho que tiene como tarea principal determinar las interacciones biomédicas entre fármacos y genes en las publicaciones.

1.1. Motivación

La medicina siempre fue personal hasta cierto punto: los doctores examinan cuál es la mejor manera de ayudar al paciente que se sienta frente a ellos. La edad, el estilo de vida, la salud y otros factores influyen en cómo responde el organismo a los medicamentos, pero así también lo hacen los genes. Los avances tecnológicos están haciendo posible el uso de las características más únicas -nuestro genoma- para diseñar tratamientos individualizados. El genoma es el conjunto completo de genes contenidos en los cromosomas, el mapa de nuestro ADN y el manual de instrucciones sobre cómo crear y mantener los millones de millones de células que hay en nuestro cuerpo. Dos personas comparten más de 99 % de su ADN, por lo tanto, lo que hace a la singularidad de cada una es algo menos de ese 1 % restante que, además, puede afectar la gravedad de una enfermedad y la efectividad de los tratamientos. Cada gen proporciona el plan para la producción de una proteína determinada en el cuerpo. Una proteína en particular puede tener un papel importante en el tratamiento farmacológico por una de varias razones, incluidas las siguientes:

- Influye en la descomposición del medicamento.
- Ayuda a la absorción o al transporte del medicamento.
- Es el objetivo del medicamento.
- Participa en una serie de sucesos moleculares desencadenados por el medicamento.

Los medicamentos modernos han permitido que se viva más y mejor, disminuyendo la mortalidad y aumentando la esperanza de vida promedio (Hidalgo-Vega, 2020). Sin embargo, es posible que aunque un medicamento funcione para la mayoría de las personas, no sea adecuado para todas ellas. También es posible que provoque efectos secundarios graves, pero que esto no suceda en todos los casos. Observar todas esas pequeñas diferencias puede ayudar a entender cuál es la mejor manera de tratar a un paciente para una variedad de enfermedades tales como las cardiovasculares y las pulmonares, el cáncer, la artritis, el colesterol alto, la depresión y la infección por VIH.

La medicina personalizada o de precisión consiste, principalmente, en un modelo médico que propone la personalización de las decisiones médicas, las prácticas y los tratamientos para cada paciente de forma individual teniendo como sustento la genómica personalizada (Gonzalez-Garay, 2014). Esta última consiste en la utilización de fármacos dirigidos a moléculas específicas que intervienen en la enfermedad del individuo teniendo en cuenta la información genética, clínica, ambiental y el estilo de vida del paciente. Se apoya tanto en el conocimiento de la naturaleza molecular de las enfermedades, como en la individualidad química y genética que posee cada paciente, gracias a las posibilidades técnicas aportadas por el conocimiento del genoma humano (Tobella, 2010). Cuando los investigadores comparan los genomas de personas que toman el mismo medicamento, pueden descubrir que un grupo de personas que comparten cierta variación genética también comparten una respuesta común al tratamiento, como por ejemplo:

- Un mayor riesgo de efectos secundarios.
- La necesidad de una dosis más alta para lograr un efecto terapéutico.
- La inexistencia de beneficios con el tratamiento.
- Un beneficio mayor o más probable con el tratamiento.
- La duración óptima del tratamiento.

En el marco de la medicina personalizada con frecuencia se les presentan a los médicos listas de genes mutados o alterados implicados en la enfermedad de un paciente. Para corregir los efectos de estas mutaciones, es de vital importancia conocer cómo diferentes fármacos interactúan con distintos genes y de qué forma. El genoma farmacológico se puede definir como los genes o productos genéticos que se sabe o se predice que interactúan con los fármacos, idealmente con un beneficio terapéutico para el paciente. El efecto que produce una droga sobre un gen determinado se denomina interacción fármaco-gen. A continuación se nombran algunos ejemplos de tipos de interacción:

- *inhibitor*: en las interacciones inhibidoras, el fármaco se une a un objetivo y disminuye su expresión o actividad. La mayoría de las interacciones de esta clase son inhibidores enzimáticos, que se unen a una enzima para reducir la actividad enzimática.
- *agonist*: una interacción agonista ocurre cuando un fármaco se une a un receptor objetivo y activa el receptor para producir una respuesta biológica.
- *antagonist*: una interacción antagonista ocurre cuando un fármaco bloquea o amortigua las respuestas mediadas por agonistas en lugar de provocar una respuesta biológica en sí misma al unirse a un receptor objetivo.
- *cofactor*: un cofactor es un medicamento que se requiere para la actividad biológica de una proteína objetivo.
- *binder*: una interacción aglutinante tiene drogas que se unen físicamente a su objetivo.
- *inducer*: en las interacciones inductoras, el fármaco aumenta la actividad de su enzima objetivo.
- *antibody*: una interacción de anticuerpos ocurre cuando un fármaco de anticuerpo se une específicamente a la molécula objetivo.
- *partial agonist*: en una interacción agonista parcial, un fármaco provocará una respuesta funcional de amplitud reducida en su receptor objetivo, en comparación con la respuesta provocada por un agonista completo.
- *ligand*: en las interacciones de ligando, un fármaco forma un complejo con su proteína objetivo para cumplir una función biológica.

- *product of*: estas interacciones ocurren cuando el gen objetivo produce el fármaco endógeno.
- *multitarget*: en las interacciones de múltiples objetivos, las drogas logran un efecto fisiológico a través de la interacción simultánea con múltiples objetivos genéticos.
- *potentiator*: en una interacción potenciadora, el fármaco mejora la sensibilidad del objetivo a los ligandos del objetivo.
- *modulator*: en las interacciones de este tipo, el medicamento regula o cambia la actividad de su objetivo.
- *activator*: una interacción activadora es cuando un fármaco activa una respuesta biológica de un objetivo, aunque el mecanismo por el cual lo hace puede no entenderse.
- *negative modulator*: en una interacción moduladora negativa, el fármaco regula negativamente la cantidad o actividad de su objetivo.

Existen numerosos recursos para ayudar a formar hipótesis acerca de cómo podría hacerse frente, terapéuticamente, a dichos eventos genómicos. Sin embargo, la utilización de estos recursos generalmente implica una tediosa revisión manual de la literatura, de los registros de ensayos clínicos y de las bases de conocimiento. Además, a medida que mejoran las tecnologías de secuenciación de próxima generación, la cantidad de literatura que contiene datos de variantes genómicas, como nuevas funciones o fenotipos relacionados, aumenta exponencialmente (K. Lee, Lee, y cols., 2016). Debido a que se publican numerosos artículos todos los días, es imposible hacer una curación (revisar, analizar y catalogar) manual de toda la información de la literatura. Por ejemplo, la base de datos *ClinicalTrials*¹ tiene a la fecha más de trescientos mil estudios. Si bien cuenta con un buscador, este sólo identifica los documentos que contengan los términos de búsqueda pero no arroja luz sobre las relaciones entre ellos. Esto genera grandes listas de resultados cuya revisión debe ser manual, implicando demoras en el inicio de los tratamientos.

Es impracticable hacer una curación manual de la literatura existente para generar una base de conocimientos de los eventos genómicos de interés que sea lo más completa y actualizada posible. Los expertos han hecho intentos de automatizar este esfuerzo mediante diferentes métodos que permiten capturar información relevante sobre un tema. Más concretamente, estos métodos consisten en realizar de manera automática procesamiento

¹<https://clinicaltrials.gov>

del lenguaje natural (NLP por sus siglas en inglés, *Natural Language Processing*) para determinar las interacciones. Un grupo de ellos, denominados NER (*Named Entity Recognition*) consisten en encontrar entidades biomédicas en texto, identificando nombres de mutaciones, genes, enfermedades y fármacos en el mismo. Ejemplos de estos métodos son el sistema MutationFinder (Caporaso, Baumgartner, Randolph, Cohen, y Hunter, 2007) que extrae mutaciones puntuales en textos con alta precisión basado en reglas; tmVar (Wei, Harris, Kao, y Lu, 2013), basado en campos aleatorios condicionales (CRF por sus siglas en inglés, *Conditional Random Fields*, un método probabilístico) que permite extraer una amplia gama de variaciones de secuencias tanto de proteínas, ADN y ARN de acuerdo a una nomenclatura estándar; y BEST (S. Lee y cols., 2016), una herramienta de búsqueda de entidades biomédicas con soporte para diez tipos diferentes de entidades incluyendo genes, drogas, enfermedades y mutaciones, que permite búsquedas actualizadas en tiempo real.

Sin embargo, estas herramientas sólo permiten identificar términos, pero no arrojan información sobre la relación entre ellos. Se hicieron esfuerzos basados en información de co-ocurrencia en el texto para capturar estas relaciones: PolySearch (Cheng y cols., 2008), un sistema web de búsqueda de estas relaciones que utiliza un enfoque relativamente simple, basado en un diccionario, que limita la identificación de entidades nuevas en la literatura y es incapaz de extraer información de contexto de las oraciones; EMU (Doughty y cols., 2010), un extractor de mutaciones basado en reglas y expresiones regulares que aunque cuente con una etapa de filtrado de falsos positivos, la cantidad de estos es alta; y HiPub (K. Lee, Shin, y cols., 2016) entre otros, que integra dos métodos NER complementarios para crear grafos para descubrimiento de conocimiento. No obstante, todos estos esfuerzos no tienen buen desempeño y su precisión es baja. Esto es una cuestión crítica puesto que a la hora de realizar búsquedas inteligentes es crucial que el buscador no se equivoque o lo haga en la menor medida posible; por otro lado, no logran discernir el tipo de relación entre las entidades. Para mejorar los resultados obtenidos con los métodos mencionados, se combinó la extracción automática con *crowdsourcing*². A pesar de ello, el *crowdsourcing* es todavía costoso y consume mucho tiempo comparado con métodos totalmente automatizados (Burger y cols., 2014).

Otro enfoque es el uso de clasificadores de aprendizaje automático o *machine learning*, más específicamente de aprendizaje profundo, o *deep learning*

²Colaboración abierta distribuida, que consiste en externalizar tareas dejándolas a cargo de un grupo numeroso de personas o de una comunidad, a través de una convocatoria abierta.

en inglés, que se refiere a una parte de la familia más amplia de métodos de aprendizaje automático basados en redes neuronales artificiales. En la actualidad son muy utilizados en una amplia variedad de aplicaciones por ofrecer un excelente desempeño (Goodfellow, Bengio, y Courville, 2016).

Un tipo de modelo de aprendizaje profundo muy utilizado para la tarea de NLP son las redes neuronales recurrentes (RNN, por sus siglas en inglés *Recurrent Neural Network*), más predominantemente las del tipo *long short-term memory* (LSTM) (Hochreiter y Schmidhuber, 1997). Estas redes han demostrado funcionar bien en el procesamiento del lenguaje natural y son robustas frente a una amplia gama de tareas (Yin, Kann, Yu, y Schütze, 2017). Sin embargo, su consumo de memoria es muy elevado y sus tiempos de entrenamiento son muy prolongados. Un nuevo tipo de modelos de aprendizaje profundo diseñado, también, para tratar con información secuencial como lo es el lenguaje, pero que no requiere que esta información secuencial sea procesada en orden como en las RNN, son los *Transformers* (Vaswani y cols., 2017), que gracias a esta característica permiten más paralelización y logran tiempos de entrenamiento reducidos.

Un trabajo que hizo uso de clasificadores de aprendizaje automático para extraer relaciones entre genes, mutaciones y fármacos en textos, es el realizado por K. Lee *et al.* (K. Lee y cols., 2018). En este trabajo se emplearon otro tipo de redes de aprendizaje profundo, denominadas redes neuronales convolucionales (CNN por sus siglas en inglés, *Convolutional Neural Network*), y aunque lo presentado en él sólo funciona a nivel de oraciones en lugar de artículos completos y carece de la distinción del tipo de interacción entre entidades, resulta prometedor la utilización de estas redes para capturar automáticamente las relaciones entre entidades biomédicas, específicamente fármaco-gen. La minería de texto utilizando aprendizaje profundo posee varias ventajas (Lai, Xu, Liu, y Zhao, 2015), sobre todo si se emplean redes neuronales de tipo convolucional, ya que en tareas relacionadas se obtuvieron buenos resultados (Dos Santos, Xiang, y Zhou, 2015) y pueden ser más prácticas que otros tipos de redes desde el punto de vista computacional (Johnson y Zhang, 2017). Sin embargo, ninguno de los dos tipos de redes CNN y *Transformers* se ha probado en el problema específico de la predicción de interacciones.

El Instituto de Investigación en Señales, Sistemas e Inteligencia Computacional *sinc(i)* ha realizado un trabajo preliminar sobre la determinación de la existencia de interacciones entre entidades biomédicas (genes, mutaciones y fármacos) en textos científicos utilizando aprendizaje automático (Bugnon y cols., 2020). Aunque este trabajo permite analizar artículos completos, carece de la distinción del tipo de interacción entre entidades, además de que el conjunto de datos de entrenamiento utilizado en dicho proyecto es pequeño.

Por lo tanto, los resultados obtenidos corren el riesgo de no ser representativos con respecto a volúmenes de datos de entrada mucho mayores.

Dada la problemática de que la gran cantidad de información generada por la comunidad científica es demasiada para ser procesada y clasificada manualmente, y que los buscadores tradicionales de texto sólo encuentran los trabajos que contienen los términos de búsqueda sin arrojar luz sobre las relaciones entre ellos, sumado a que los intentos de automatizar el trabajo de curación no brindan los resultados deseados, en este proyecto se presenta un sistema integral de búsqueda inteligente de interacciones entre fármacos y genes. Además, se trata de solventar los huecos donde los métodos automáticos existentes son débiles o fallan. El mismo está compuesto por un módulo de aprendizaje automático que permite inferir la existencia y el tipo de interacciones entre fármacos y genes en textos científicos. Este módulo neuronal, a su vez, es el motor de una aplicación web que brinda la posibilidad de realizar búsquedas específicas sobre estos conceptos.

Para su realización se prueban y comparan dos arquitecturas de aprendizaje profundo para el procesamiento del lenguaje natural, una red neuronal convolucional y una red *Transformer*, sobre un conjunto amplio de datos mediante distintas combinaciones de hiperparámetros, obteniendo conclusiones sobre cuál es más adecuada para esta tarea específica.

Se espera que el sistema de búsqueda aquí desarrollado permita la recuperación de información sobre las interacciones entre fármacos y genes de una manera más rápida y precisa. Como consecuencia, es deseable que se vuelva un recurso valioso para los médicos en el campo de la medicina de precisión, pudiendo llegar a contribuir en acelerar el inicio de los tratamientos en los pacientes. Además, será útil para cubrir la brecha de conocimiento que actualmente existe entre los expertos en el campo clínico y los investigadores genómicos. Los primeros están íntimamente familiarizados con los comportamientos específicos de la enfermedad y las terapias dirigidas que se utilizan en el campo, en cambio los últimos poseen la experiencia técnica para detectar eventos ocultos conocidos o potencialmente nuevos en los datos moleculares de las muestras de enfermedades en estudio. Cubriendo esta brecha se ayuda a los investigadores, tanto básicos como clínicos, a priorizar e interpretar los resultados de los estudios de genoma en el contexto de la función de los genes, los fenotipos clínicos, las decisiones de tratamiento y los resultados de los pacientes.

1.2. Objetivos

1.2.1. Objetivo general

- Desarrollar un sistema de búsqueda inteligente de interacciones entre fármacos y genes en textos científicos utilizando aprendizaje profundo.

1.2.2. Objetivos específicos

- Crear una base de datos de publicaciones científicas etiquetadas que permita el entrenamiento de modelos de aprendizaje automático.
- Desarrollar un motor de búsqueda de interacciones fármaco-gen mediante redes neuronales convolucionales (CNN).
- Desarrollar un motor de búsqueda de interacciones fármaco-gen mediante *Transformers*.
- Desarrollar una aplicación web que permita a usuarios realizar búsquedas de interacciones fármaco-gen utilizando el mejor motor de búsqueda obtenido.

1.3. Alcance

1.3.1. Requerimientos funcionales

Modelos neuronales:

- Deben ser capaces de realizar procesamiento del lenguaje natural (NLP).
- Deben lograr la clasificación de los tipos de interacciones entre fármacos y genes a partir del contenido de publicaciones médicas.
- Deben estar entrenados bajo el tipo de aprendizaje supervisado.

Aplicación web:

- El sistema desarrollado debe permitir realizar búsquedas de interacciones fármaco-gen sobre el recurso *online PubMed*³, infiriendo las interacciones con la utilización de la red entrenada.

³<https://pubmed.ncbi.nlm.nih.gov>

- Debe contar con un registro opcional de cuentas de usuario para brindar a este ciertas funciones añadidas como historial de búsquedas, búsquedas específicas guardadas y artículos favoritos. No es necesario que el usuario esté registrado para poder realizar búsquedas.
- Debe permitir al usuario realizar búsquedas ingresando el nombre de un gen y/o un fármaco.
- El usuario debe poder especificar un tipo de interacción, lo cual filtra los resultados sólo a ese tipo.
- Los resultados deben ser presentados junto a la información de los artículos de donde se obtuvieron los datos correspondientes al tipo de interacción.

1.3.2. Requerimientos no funcionales

Modelos neuronales:

- Deben clasificar los tipos de interacciones entre fármacos y genes con un porcentaje de acierto mayor al 60 %.
- El área bajo la curva ROC de las distintas clases debe ser superior al 60 % para la mayoría de clases.
- Las redes deben estar programadas con el lenguaje de programación Python.
- Se debe utilizar Keras⁴ para la confección de los modelos neuronales. Keras es una API libre de redes neuronales (desarrollo y entrenamiento) de alto nivel, escrita en Python y capaz de ejecutarse sobre TensorFlow⁵.
- Se debe utilizar la biblioteca libre de Python NumPy⁶ para el manejo matricial de los datos.

Aplicación web:

- Debe poseer una interfaz de usuario basada en un formulario web usando tecnologías web estándares.

⁴<https://keras.io>

⁵<https://www.tensorflow.org>

⁶<https://numpy.org>

- Debe ser compatible con los navegadores actuales más usados del mercado.
- Debe estar programada en el lenguaje de programación Python para concordar con el desarrollo del modelo neuronal y permitir una integración simple y fluida.
- Se debe utilizar el *framework* web libre Flask⁷.
- La base de datos de la aplicación web debe correr sobre el motor libre PostgreSQL⁸.

1.4. Metodología

En este proyecto se adoptan dos metodologías de trabajo, una para los modelos neuronales y otra para la aplicación web.

Modelos neuronales:

Se adopta una metodología específica para desarrollos de aprendizaje automático propuesta por S. Amershi *et. al* (Amershi y cols., 2019) (Figura 1.1), ya que las metodologías tradicionales, tanto las clásicas como las ágiles, no son aplicables debido a tres características fundamentales de este tipo de trabajo: el aprendizaje automático se trata completamente sobre datos; requiere habilidades muy diferentes a las encontradas típicamente en equipos de software; y es más difícil mantener una separación de módulos estricta entre componentes de aprendizaje automático que entre otros componentes de software.

La metodología adoptada se compone de 9 etapas definidas: *requerimientos del modelo*; *recolección de datos*; *limpieza de datos*; *etiquetado de datos*; *análisis de características*; *entrenamiento del modelo*; *evaluación del modelo*; *despliegue*; y *monitoreo*. La etapa de *análisis de características* no se realizará de forma directa ya que las redes de aprendizaje profundo pueden extraer características automáticamente. Las etapas de *despliegue del modelo* y *monitoreo del modelo* quedan descartadas ya que están fuera del alcance de este proyecto.

Aplicación web:

⁷<https://flask.palletsprojects.com>

⁸<https://www.postgresql.org>

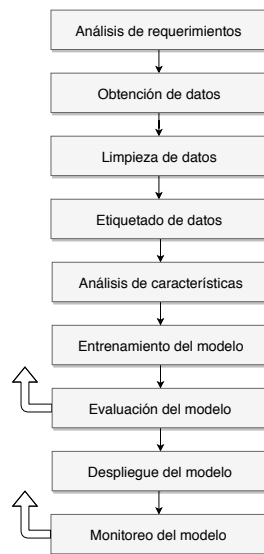


Figura 1.1: Metodología para el desarrollo de la red neuronal.

Para el desarrollo de la aplicación web se adopta la metodología del ciclo de vida en cascada. La misma se compone de etapas definidas que se llevan a cabo secuencialmente (Figura 1.2): *análisis de requerimientos*; *diseño*; *codificación*; *pruebas*; *implementación*; y *mantenimiento*. Las dos últimas etapas quedan descartadas ya que están fuera del alcance de este proyecto.

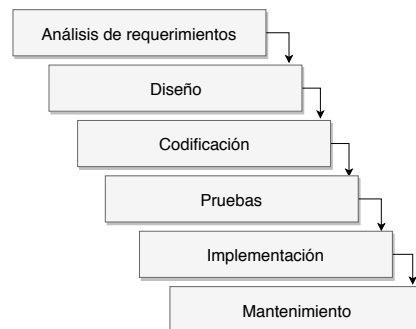


Figura 1.2: Metodología en cascada para el desarrollo de la aplicación web.

Además, cada módulo incluye inicialmente una etapa de recopilación y revisión bibliográfica relacionada con su tema en cuestión, donde se recolecta y estudia toda la información necesaria para llevar a cabo este proyecto. Esto incluye, por un lado, inteligencia computacional, específicamente relacionada con aprendizaje automático y sobre todo con redes neuronales artificiales de aprendizaje profundo, y por el otro información sobre desarrollo web.

Capítulo 2

Marco teórico

En este capítulo se presenta el marco teórico necesario para poder comprender el desarrollo del proyecto.

2.1. Aprendizaje automático

Un algoritmo de aprendizaje automático es un algoritmo que puede aprender de los datos. *Se dice que un programa de computadora aprende de la experiencia E con respecto a algunas clases de tareas T y la medida de desempeño P , si su desempeño en tareas en T , medido por P , mejora con la experiencia E* (Mitchell, 1997). El aprendizaje automático permite abordar tareas que son demasiado difíciles de resolver con programas fijos, escritos y diseñados por seres humanos. El proceso de aprendizaje en sí no es la tarea, sino el medio para lograr la capacidad de realizarla. Las tareas de aprendizaje automático, generalmente, se describen en términos de cómo el sistema debería procesar un ejemplo, donde un ejemplo es una colección de características, normalmente representada como un vector, que se han medido cuantitativamente a partir de algún objeto o evento que se quiere que el sistema procese.

Se pueden resolver muchos tipos de tareas con el aprendizaje automático, pero se enfocará en describir sólo la tarea de *clasificación*, pues es la que se utilizó en este proyecto. En la tarea de clasificación se le pide al programa de computadora que especifique a qué k categorías pertenece alguna entrada. Para resolver esta tarea, generalmente se le pide al algoritmo de aprendizaje que produzca una función $f : \mathbb{R}^n \rightarrow \{1, \dots, k\}$. Cuando $y = f(x)$, el modelo asigna una entrada descrita por el vector x a una categoría identificada por el código numérico y . Existen otras variantes de la tarea de clasificación, por ejemplo, donde f genera una distribución de probabilidad sobre las clases.

Para evaluar las habilidades de un algoritmo de aprendizaje automático,

se debe diseñar una medida cuantitativa de su rendimiento. Usualmente, esta medida de rendimiento P es específica de la tarea T que lleva a cabo el sistema. Para la tarea de clasificación a menudo se mide la precisión del modelo. La precisión es sólo la proporción de ejemplos para los cuales el modelo produce la salida correcta. También se puede obtener información equivalente midiendo la tasa de error (la proporción de ejemplos para los cuales el modelo produce una salida incorrecta). Por lo general, se está interesado en qué tan bien funciona el algoritmo de aprendizaje automático en datos que no ha visto antes, ya que esto determina qué tan bien funcionará cuando se implemente en el mundo real. Por lo tanto, se evalúan estas medidas de rendimiento utilizando un conjunto de datos de prueba que es independiente de los datos utilizados durante el proceso de aprendizaje del sistema (proceso denominado entrenamiento).

Los algoritmos de aprendizaje automático se pueden clasificar en términos generales como supervisados o no supervisados por el tipo de experiencia que se les permite tener durante el proceso de aprendizaje. Los algoritmos de aprendizaje supervisados experimentan un conjunto de datos que contiene características, pero cada ejemplo también está asociado con una etiqueta u objetivo, lo que implica observar varios ejemplos de un vector aleatorio x y un valor asociado o vector y , y luego aprender a predecir y a partir de x , generalmente estimando $p(y|x)$. Este tipo de aprendizaje es el utilizado en la tarea de clasificación. El aprendizaje no supervisado consiste en observar varios ejemplos de un vector aleatorio x e intentar aprender implícita o explícitamente la distribución de probabilidad $p(x)$, o algunas propiedades interesantes de esa distribución.

Una forma común de describir un conjunto de datos es con una matriz que contiene un ejemplo diferente en cada fila, donde cada columna de la matriz corresponde a una característica diferente. Esto implica que los datos deban poder ser descritos como vectores numéricos y ser del mismo tamaño. En el caso del aprendizaje supervisado, el ejemplo contiene una etiqueta u objetivo, es decir, que dada una matriz de observaciones de características \mathbf{X} , también se proporciona un vector de etiquetas \mathbf{y} , donde y_i proporciona la etiqueta para el i -ésimo ejemplo.

En todo algoritmo de aprendizaje automático existe un conjunto de parámetros w , los cuales son valores numéricos que controlan el comportamiento del sistema. Se puede pensar en w como un conjunto de pesos que determinan cómo cada característica afecta a la predicción: si el peso de una característica es grande en magnitud, entonces tiene un gran efecto en la predicción; si el peso es cero, entonces no tiene efecto en la predicción. Para medir el desempeño del algoritmo de aprendizaje automático se debe comparar la salida deseada (y) con la predicción obtenida (\hat{y}), mediante alguna función de

pérdida (*loss function*), la cual puede variar de acuerdo al problema. Con esta diferencia se ajustan los valores de los pesos w para reducirla. Esto se realiza utilizando el método del gradiente descendiente aplicado a dicha función.

El desafío central en el aprendizaje automático es que los algoritmos deben funcionar bien en entradas nuevas, nunca antes vistas, no sólo aquellas en las que el modelo fue entrenado, lo que se denomina capacidad de generalización. Cuando se entrena un modelo sobre un conjunto de datos de entrenamiento se calcula alguna medida de error en dicho conjunto llamado error de entrenamiento, y se busca reducirlo. Pero lo más importante es reducir el error de generalización, también llamado error de prueba, que se define como el valor esperado del error en una nueva entrada. Por lo general, se estima el error de generalización de un modelo de aprendizaje automático midiendo su rendimiento en un conjunto de ejemplos de prueba que se recopilaron por separado del conjunto de entrenamiento. Se supone que los ejemplos en cada conjunto de datos son independientes entre sí, y que el conjunto de entrenamiento y el de prueba están distribuidos de manera idéntica, respetando la misma distribución de probabilidad. De esta manera, utilizando el conjunto de entrenamiento para elegir los parámetros (pesos w) para reducir el error de entrenamiento, se toman luego muestras del conjunto de prueba donde el error de prueba esperado es mayor o igual que el valor esperado del error de entrenamiento.

Los factores que determinan qué tan bien funcionará un algoritmo de aprendizaje automático son su capacidad para hacer que el error de entrenamiento sea pequeño y su capacidad para reducir la brecha entre el error de entrenamiento y el de prueba. Estos dos factores corresponden a sortear los dos retos centrales en el aprendizaje automático: ajuste insuficiente (*underfitting*) y ajuste excesivo (*overfitting*). El *underfitting* ocurre cuando el modelo no puede obtener un valor de error suficientemente bajo en el conjunto de entrenamiento; el *overfitting*, cuando la brecha entre el error de entrenamiento y el error de prueba es demasiado grande, es decir que el modelo se sobreajusta memorizando las propiedades del conjunto de entrenamiento que no le sirven bien en el conjunto de prueba.

La mayoría de los algoritmos de aprendizaje automático tienen hiperparámetros: configuraciones que se pueden usar para controlar el comportamiento del algoritmo. Los valores de los hiperparámetros no están adaptados por el algoritmo de aprendizaje en sí (aunque se puede diseñar un procedimiento de aprendizaje anidado en el que un algoritmo de aprendizaje aprende los mejores hiperparámetros para otro algoritmo de aprendizaje) sino que se los selecciona *a priori* y se los va adaptando a través de varias sesiones de entrenamiento.

Si se ajustan los hiperparámetros del modelo de tal manera que se busca

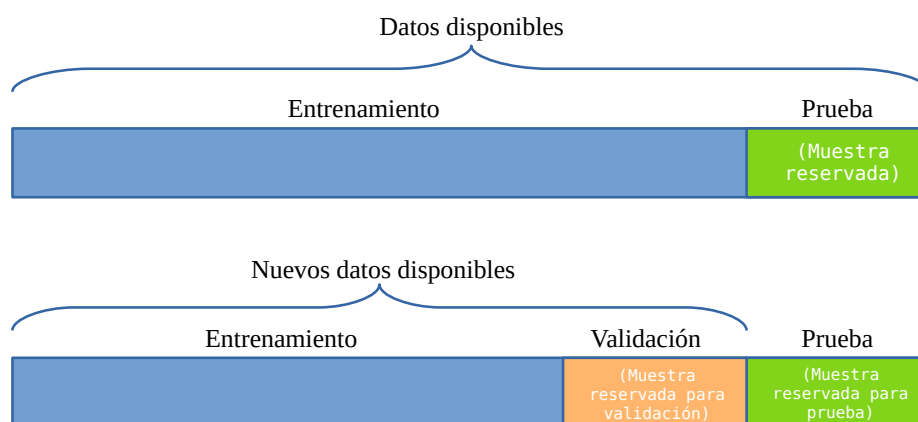


Figura 2.1: División de los datos disponibles en los distintos conjuntos.

lograr el máximo desempeño posible de este a partir de, solamente, el conjunto de datos de entrenamiento, puede dar como resultado el problema del sobreajuste. Para resolver este problema, se necesita un conjunto de ejemplos de validación que el algoritmo de entrenamiento no observe. Anteriormente, se habló de que un conjunto de datos de prueba puede usarse para estimar el error de generalización, sin embargo, es importante que los ejemplos de prueba no se usen de ninguna manera para tomar decisiones sobre el modelo. Por lo tanto, no deben usarse estos datos para ajustar los hiperparámetros. Por este motivo se construye un conjunto de validación a partir de los datos de entrenamiento que servirá para ajustarlos. Específicamente, se divide los datos de entrenamiento en dos subconjuntos disjuntos: uno se usa para aprender los parámetros (pesos) y el otro es utilizado para estimar el error de generalización durante o después del entrenamiento, lo que permite que los hiperparámetros se actualicen en consecuencia. Este subconjunto de datos que guía la selección de hiperparámetros se denomina conjunto de validación (ver Figura 2.1). Por lo general, se usa aproximadamente el 80 por ciento de los datos de entrenamiento para el entrenamiento en sí y el 20 por ciento para la validación. Dado que el conjunto de validación se utiliza para *entrenar* los hiperparámetros, el error del conjunto de validación subestimaré el error de generalización, aunque generalmente en una cantidad menor que el error de entrenamiento. Una vez completada la optimización de hiperparámetros, el error de generalización puede estimarse usando el conjunto de prueba.

Dividir el conjunto de datos en un conjunto de entrenamiento fijo y un conjunto de validación fijo puede ser problemático si resulta que el conjunto de validación es pequeño. Un pequeño conjunto de validación implica incertidumbre estadística en torno al error de validación promedio estimado, por lo

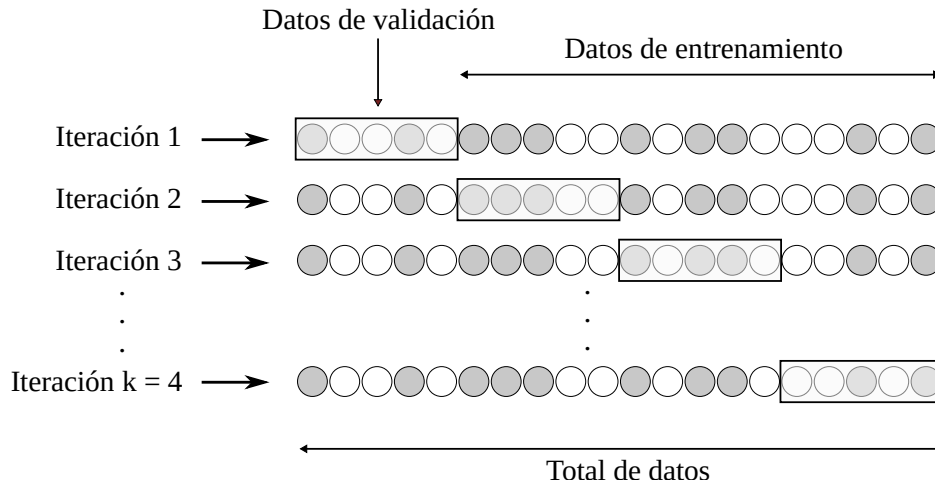


Figura 2.2: Representación del proceso de validación cruzada de K iteraciones con $K = 4$.

que es difícil ajustar ese algoritmo. Cuando el conjunto de datos tiene cientos de miles de ejemplos o más, no es un problema grave. Cuando es demasiado pequeño, los procedimientos alternativos permiten utilizar todos los ejemplos en la estimación del error medio de validación, al precio de un mayor costo computacional. Estos procedimientos se basan en la idea de repetir el entrenamiento y el cómputo de pruebas de validación en diferentes subconjuntos o divisiones elegidos al azar del conjunto de datos original. El más común de estos es el procedimiento de validación cruzada k -fold, en el que se forman particiones del conjunto de datos dividiéndolo en k subconjuntos no superpuestos de entrenamiento y validación (Figura 2.2). El error de validación se puede estimar tomando el error de validación promedio en k pruebas. El i -ésimo subconjunto k de los datos se usa como el conjunto de validación, y el resto de los datos se usa como el conjunto de entrenamiento. Una alternativa llamada k -fold estratificado consiste en realizar el procedimiento de k -fold antes mencionado pero manteniendo la proporción de las clases en los distintos conjuntos de entrenamiento y validación.

2.2. Aprendizaje profundo

Las redes neuronales profundas son redes neuronales artificiales que se caracterizan por el encadenamiento de múltiples capas, que pueden ser de distintos tipos, para realizar el aprendizaje automático. Entre los tipos de capas existen capas recurrentes, capas convolucionales, capas densas (perceptrones), entre otras.

El objetivo de estas redes es aproximar alguna función f^* . Por ejemplo, para un clasificador, $y = f^*(\mathbf{x})$ asigna una entrada \mathbf{x} a una categoría y . Un tipo muy usado para esto son las redes de propagación hacia adelante, las cuales definen una asignación $y = f(\mathbf{x}; \boldsymbol{\theta})$ y aprenden el valor de los parámetros $\boldsymbol{\theta}$ que dan como resultado la mejor aproximación de la función.

Estos modelos se denominan *de propagación hacia adelante* porque la información fluye a través de la función que se evalúa desde \mathbf{x} , a través de los cálculos intermedios utilizados para definir f , y finalmente hasta la salida y , la cual no vuelve a entrar al modelo mediante conexiones de retroalimentación. Las redes de propagación hacia adelante son la base de muchas aplicaciones comerciales importantes (Goodfellow y cols., 2016). Por ejemplo, las redes convolucionales utilizadas para el reconocimiento de objetos a partir de fotos son un tipo especializado de estas redes.

Estos modelos se denominan redes porque, normalmente, se representan componiendo muchas funciones diferentes. El modelo está asociado con un grafo acíclico dirigido que describe cómo se componen las funciones juntas. Por ejemplo, se podrían tener tres funciones f_1 , f_2 y f_3 conectadas en una cadena, para formar $f(\mathbf{x}) = f_3(f_2(f_1(\mathbf{x})))$. Estas funciones anidadas son las estructuras más utilizadas de las redes neuronales. En este caso, f_1 se llama la primera capa de la red, f_2 se llama la segunda capa, y así sucesivamente. La longitud total de la cadena da la profundidad del modelo, y de allí surge el nombre *aprendizaje profundo*. La capa final de una red se llama capa de salida. Durante el entrenamiento de la red neuronal, se trata de llevar a $f(\mathbf{x})$ a que coincida con $f^*(\mathbf{x})$. Los datos de entrenamiento brindan ejemplos ruidosos y aproximados de $f^*(\mathbf{x})$ evaluados en diferentes puntos de entrenamiento. Cada ejemplo \mathbf{x} va acompañado de una etiqueta $y \approx f^*(\mathbf{x})$. Los ejemplos de entrenamiento especifican directamente lo que debe hacer la capa de salida en cada punto \mathbf{x} (debe producir un valor cercano a y), pero no especifican directamente el comportamiento de las otras capas. El algoritmo de aprendizaje debe decidir cómo usar esas capas para producir la salida deseada, es decir, para implementar mejor una aproximación de f^* . Debido a que los datos de entrenamiento no muestran el resultado deseado para cada una de estas capas, se denominan capas ocultas.

Finalmente, estas redes se llaman neuronales porque están inspiradas por la neurociencia. Cada capa oculta de la red suele tener un valor vectorial, y su dimensión determina el tamaño del modelo. Cada elemento del vector puede interpretarse como un papel análogo a una neurona. En lugar de pensar que la capa representa una sola función de vector a vector, también se puede pensar que consta de muchas unidades que actúan en paralelo, cada una de las cuales representa una función de vector a escalar. Cada unidad se asemeja a una neurona en el sentido de que recibe información de muchas otras unidades

y calcula su propio valor de activación. Sin embargo, es mejor pensar en las redes de propagación hacia adelante como máquinas de aproximación de funciones que están diseñadas para lograr una generalización estadística, y no para modelar perfectamente el cerebro.

Las redes neuronales, generalmente, se entrenan mediante el uso de optimizadores iterativos basados en gradientes que, simplemente, llevan la función de pérdida a un valor muy bajo. Cuando se usa una red neuronal de propagación hacia adelante para aceptar una entrada \mathbf{x} y producir una salida \hat{y} , la información fluye hacia adelante por la red a través de las capas ocultas. Durante el entrenamiento, la propagación puede continuar hacia adelante hasta que produzca un costo escalar $J(\theta)$. Luego, mediante el algoritmo de propagación hacia atrás o retropropagación, se puede hacer que la información de este costo fluya hacia atrás a través de la red para calcular el gradiente. Calcular una expresión analítica para el gradiente es sencillo, pero evaluar numéricamente tal expresión puede ser computacionalmente costoso; a pesar de ello el algoritmo de retropropagación lo hace utilizando un procedimiento simple y económico. Se debe tener en cuenta que la retropropagación se refiere sólo al método para calcular el gradiente, mientras que otro algoritmo, como el gradiente descendiente estocástico, se utiliza para realizar el aprendizaje utilizando este gradiente.

2.2.1. *Embeddings*

Los *embeddings* de palabras son un tipo de representación de palabras que permiten que aquellas con un significado similar tengan una representación similar. Estas representaciones son aprendidas a partir de los textos de interés. Los *embeddings* de palabras son, de hecho, una clase de técnicas en las que las palabras individuales se representan como vectores con valores reales en un espacio vectorial predefinido. Cada palabra se asigna a un vector y los valores de este se aprenden de una manera que se asemeja a una red neuronal y, por lo tanto, la técnica a menudo se incluye en el campo del aprendizaje profundo.

La clave del enfoque es la idea de utilizar una representación distribuida densa para cada palabra; cada una de ellas está representada por un vector de valor real, a menudo de decenas o cientos de dimensiones. Esto contrasta con las miles o millones de dimensiones requeridas para representaciones de palabras dispersas, como la *one-hot encoding*, pero el número de características resulta mucho menor que el tamaño del vocabulario. La representación distribuida se aprende en función del uso de palabras, lo que permite que las palabras que se usan de manera similar tengan como resultado representaciones similares, capturando naturalmente su significado. Esto puede

contrastarse con la representación nítida pero frágil en un modelo de bolsa de palabras (*bag of words*) donde, a menos que se gestionen explícitamente, las diferentes palabras tienen diferentes representaciones, independientemente de cómo se usen. Uno de los beneficios del uso de vectores densos y de baja dimensión es computacional. La mayoría de las redes neuronales no funcionan bien con vectores dispersos de muy alta dimensión, y por otro lado, el principal beneficio de las representaciones densas es el poder de generalización: si se cree que algunas características pueden proporcionar pistas similares, vale la pena proporcionar una representación que pueda capturar estas similitudes.

Los métodos de *embeddings* de palabras aprenden una representación vectorial de valores reales para un vocabulario de tamaño fijo predefinido a partir de un corpus de texto. El proceso de aprendizaje se combina con el modelo de red neuronal en alguna tarea, como la clasificación de documentos, o es un proceso no supervisado que utiliza estadísticas de documentos. Existen tres técnicas que se pueden usar para aprender *embeddings* de palabras a partir de datos de texto:

- Capas de *embedding*:

Una capa de *embedding*, por falta de un nombre mejor, es un *embedding* de palabras que se aprende conjuntamente con un modelo de red neuronal en una tarea específica de procesamiento del lenguaje natural, como el modelado del lenguaje o la clasificación de documentos. Requiere que el texto del documento se limpie y prepare de manera que cada palabra tenga una codificación *one-hot*. El tamaño del espacio vectorial se especifica como parte del modelo, como 50, 100 o 300 dimensiones. Los vectores se inicializan con pequeños números aleatorios. La capa de *embedding* se usa como primera capa de una red neuronal y se ajusta de forma supervisada utilizando el algoritmo de retropropagación. Cuando la entrada a una red neuronal contiene características categóricas simbólicas (por ejemplo, características que toman uno de k símbolos distintos, como palabras de un vocabulario cerrado), es común asociar cada valor de característica posible (es decir, cada palabra en el vocabulario) con un vector d -dimensional para algunos d . Estos vectores se consideran parámetros del modelo y se entrenan conjuntamente con los otros parámetros. Las palabras con codificación *one-hot* se asignan a los vectores de palabras. Si se usa un modelo de perceptrón multicapa, entonces los vectores de palabras se concatenan antes de ser alimentados como entrada al modelo. Si se utiliza una red neuronal recurrente, cada palabra se puede tomar como una entrada en una secuencia. Este enfoque de aprender *embeddings* mediante una capa requiere una gran cantidad de datos de entrenamiento y puede ser lento, pero aprenderá

embeddings dirigidos tanto a los datos de texto específicos como a la tarea de NLP.

- Word2vec:

Word2vec (Mikolov, Chen, Corrado, y Dean, 2013) es un método estadístico para aprender eficientemente *embeddings* de palabras independiente de un corpus de texto. Fue desarrollado para hacer que el entrenamiento de *embeddings* basado en redes neuronales sea más eficiente y, desde entonces, se ha convertido en el estándar para el desarrollo de la inclusión de palabras previamente entrenadas (Goodfellow y cols., 2016). Se encontró que estas representaciones son sorprendentemente buenas para capturar regularidades sintácticas y semánticas en el lenguaje, y que cada relación se caracteriza por un desplazamiento de vector específico de relación. Esto permite un razonamiento orientado a vectores, basado en los desplazamientos entre palabras. Por ejemplo, la relación hombre / mujer se aprende automáticamente, y con las representaciones vectoriales inducidas, *Rey - Hombre + Mujer* da como resultado un vector muy cercano a *Reina*.

Se introdujeron dos modelos de aprendizaje diferentes que se pueden utilizar como parte del enfoque word2vec. El modelo CBOW aprende el *embedding* prediciendo la palabra actual en función de su contexto. El modelo continuo *Skip-Gram* aprende prediciendo las palabras circundantes dada una palabra actual. Ambos modelos se centran en aprender sobre palabras dado su contexto de uso local, donde el contexto se define mediante una ventana de palabras vecinas, y donde esta ventana es un parámetro configurable del modelo. El tamaño de la ventana deslizante tiene un fuerte efecto en las similitudes de vectores resultantes; las ventanas grandes tienden a producir más similitudes tópicas, mientras que las más pequeñas tienden a producir similitudes más funcionales y sintácticas. El beneficio clave del enfoque es que los *embeddings* de palabras de alta calidad se pueden aprender de manera eficiente (poco espacio y complejidad de tiempo), lo que permite aprender *embeddings* más grandes (más dimensiones) de cuerpos de texto mucho más amplios (miles de millones de palabras).

- GloVe:

GloVe (Pennington, Socher, y Manning, 2014), o algoritmo de vectores globales para representación de palabras, es una extensión del método word2vec para aprender eficientemente vectores de palabras. Las representaciones clásicas de palabras del modelo de espacio vectorial

se desarrollaron utilizando técnicas de factorización matricial como el Análisis Semántico Latente (LSA). Hacen un buen trabajo al usar estadísticas globales de texto, pero no son tan buenas como los métodos aprendidos como word2vec para capturar significado y demostrarlo en tareas como calcular analogías (el ejemplo del rey y la reina anterior). GloVe es un enfoque para unir las estadísticas globales de las técnicas de factorización matricial como LSA con el aprendizaje basado en el contexto local en word2vec. En lugar de utilizar una ventana para definir el contexto local, GloVe construye una matriz explícita de contexto de palabras o de coincidencia de palabras utilizando estadísticas en todo el corpus de texto. El resultado es un modelo de aprendizaje que puede resultar en mejores *embeddings* de palabras en general.

A la hora de utilizar *embeddings* de palabras en proyectos NLP se cuenta con dos opciones: aprender los *embeddings* o cargar *embeddings* previamente entrenados. En el primer caso se requerirá una gran cantidad de datos de texto para garantizar que se aprendan *embeddings* útiles, como millones o miles de millones de palabras. Los *embeddings* se pueden aprender de forma independiente, donde un modelo está capacitado para aprenderlos, para luego guardarlos y usarlos como parte de otro modelo para su tarea más adelante. Siendo este un buen enfoque, si se desea utilizar los mismos *embeddings* en varios modelos. También se pueden aprender de manera conjunta, como parte de un modelo grande de tareas específicas.

Es común que los investigadores posibiliten que los *embeddings* de palabras previamente entrenados estén disponibles de forma gratuita, a menudo bajo una licencia permisiva, para que puedan ser utilizados en proyectos académicos o comerciales, como es el caso de los *embeddings* de word2vec y GloVe, los cuales están disponibles para su descarga gratuita. Se dispone de dos opciones principales cuando se trata de usar *embeddings* previamente entrenados. En la primera, los *embeddings* se mantienen estáticos y se usan como un componente del modelo. Este es un enfoque adecuado si el *embedding* es apropiado para el problema y da buenos resultados. En la segunda, los *embeddings* se actualizan conjuntamente durante el entrenamiento del modelo, siendo esta una buena opción si se está buscando aprovechar al máximo el modelo e integrarlo en su tarea.

2.2.2. Redes convolucionales

Las redes neuronales convolucionales, o CNN (*Convolutional Neural Network*), son un tipo especializado de red neuronal para procesar datos que tienen una topología similar a una cuadrícula conocida. Los ejemplos inclu-

yen datos de series temporales, que pueden considerarse como una cuadrícula 1D que toma muestras a intervalos de tiempo regulares, y datos de imágenes, que pueden considerarse como una cuadrícula de píxeles 2D. El nombre *red neuronal convolucional* indica que la red emplea un tipo especializado de operación lineal matemática llamada convolución, en lugar de la multiplicación de matriz general, en al menos una de sus capas. Este tipo de redes han sido muy exitosas en aplicaciones prácticas (Goodfellow y cols., 2016).

En su forma más general, la convolución es una operación entre dos funciones de un argumento de valor real y se denota con un asterisco: $s(t) = (x * w)(t)$. La convolución se define para cualquier función para la cual se define la integral $s(t) = \int x(a)w(t-a)da$, y puede usarse para otros fines además de tomar promedios ponderados. En su versión discreta la convolución se define como: $\sum_{a=-\infty}^{\infty} x(a)w(t-a)$. En la terminología de red convolucional, el primer argumento de la convolución a menudo se denomina entrada, y el segundo argumento como núcleo (*kernel*). La salida a veces se denomina mapa de características. En las aplicaciones de aprendizaje automático, la entrada suele ser una matriz de datos multidimensional, y el núcleo suele ser una matriz multidimensional de parámetros que son adaptados por el algoritmo de aprendizaje. Cuando el núcleo es bidimensional, la convolución es denominada 2D y puede pensarse, en el contexto de imágenes, como la aplicación de un filtro a una imagen. También existe la convolución 1D, denominada temporal, que da resultado a un elemento unidimensional respetando la naturaleza temporal de los datos de entrada. Debido a que cada elemento de la entrada y el núcleo deben almacenarse explícitamente por separado, generalmente se asume que estas funciones son cero en todas partes excepto en el conjunto finito de puntos para los que almacenamos los valores. En la práctica significa que se puede implementar la suma infinita como una suma sobre un número finito de elementos de matriz.

La convolución aprovecha tres ideas importantes que pueden ayudar a mejorar un sistema de aprendizaje automático: interacciones dispersas, intercambio de parámetros y representaciones equivariantes. Además, la convolución proporciona un medio para trabajar con entradas de tamaño variable. Las redes convolucionales suelen tener interacciones dispersas (también conocido como conectividad dispersa o pesos dispersos). Esto se logra haciendo que el núcleo sea más pequeño que la entrada. Por ejemplo, al procesar una imagen, la imagen de entrada puede tener miles o millones de píxeles, pero se pueden detectar características pequeñas y significativas como bordes con núcleos que ocupan sólo decenas o cientos de píxeles. Esto significa que se necesita almacenar menos parámetros, lo que reduce los requisitos de memoria del modelo y mejora su eficiencia estadística. También significa que calcular la salida requiere menos operaciones. Estas mejoras en la eficiencia suelen ser

bastante grandes. Si hay m entradas y n salidas, entonces la multiplicación de la matriz requiere $m \times n$ parámetros, y los algoritmos utilizados en la práctica tienen un tiempo de ejecución $O(m \times n)$ (por ejemplo). Si se limita el número de conexiones que cada salida puede tener a k , entonces el enfoque escasamente conectado requiere sólo $k \times n$ parámetros y $O(k \times n)$ tiempo de ejecución. Para muchas aplicaciones prácticas, es posible obtener un buen rendimiento en la tarea de aprendizaje automático mientras se mantiene un k pequeño, varios órdenes de magnitud menor que m .

El uso compartido de parámetros se refiere al uso del mismo parámetro para más de una función en un modelo. En una red neuronal tradicional, cada elemento de la matriz de pesos se usa, exactamente, una vez al calcular la salida de una capa, multiplicándose por un elemento de la entrada y, luego, no volviéndose a visitar. En una red neuronal convolucional, cada miembro del núcleo se usa en cada posición de la entrada (excepto quizás algunos de los píxeles del límite, dependiendo de las decisiones de diseño con respecto al límite). El uso compartido de parámetros utilizado por la operación de convolución significa que, en lugar de aprender un conjunto separado de parámetros para cada ubicación, se aprende sólo un conjunto. Esto no afecta el tiempo de ejecución de la propagación directa, pero reduce aún más el almacenamiento de requisitos del modelo para k parámetros.

El uso compartido de parámetros, en el caso de la convolución, tiene la consecuencia de que la capa tenga la propiedad de equivanianza a la traslación. Decir que una función es equivariante significa que si la entrada cambia, la salida cambia del mismo modo. Al procesar datos de series temporales, significa que la convolución produce una especie de línea de tiempo que muestra cuándo aparecen diferentes características en la entrada; si se mueve un evento más tarde en la entrada, la misma representación exacta aparecerá en la salida, justo más tarde. De manera similar con las imágenes, la convolución crea un mapa 2D en donde aparecen ciertas características en la entrada. Si se mueve el objeto en la entrada, su representación se moverá la misma cantidad en la salida. Esto es conveniente cuando se sabe que alguna función de un pequeño número de píxeles vecinos es útil cuando se aplica a múltiples ubicaciones de entrada. Sin embargo, la convolución no es naturalmente equivariante a algunas otras transformaciones, como los cambios en la escala o la rotación de una imagen, por lo que otros mecanismos son necesarios para manejar este tipo de transformaciones.

Una capa típica de una red convolucional consta de tres etapas. En la primera, la capa realiza varias convoluciones en paralelo para producir un conjunto de activaciones lineales. En la segunda, cada activación lineal se ejecuta a través de una función de activación no lineal, como la función de activación lineal rectificada (ReLU). Esta etapa a veces se llama etapa del

detector. En la tercera etapa, se usa una función de agrupación (*pooling*) para modificar aún más la salida de la capa. Una función de agrupación reemplaza la salida de la red en una ubicación determinada con una estadística resumida de las salidas cercanas. Por ejemplo, la operación de agrupación máxima (*Max pooling*) informa la salida máxima dentro de un vecindario rectangular. Otras funciones de agrupamiento populares incluyen el promedio (*Average pooling*) o la norma L^2 de un vecindario rectangular, o un promedio ponderado basado en la distancia desde el píxel central.

En todos los casos, la agrupación ayuda a que la representación sea, aproximadamente, invariable a pequeñas traslaciones de la entrada. Esto significa que si se traslada la entrada en una pequeña cantidad, los valores de la mayoría de las salidas agrupadas no cambian. La invariancia a la traslación local puede ser una propiedad útil si es más importante saber si alguna característica está presente que exactamente dónde está. En otros contextos, es más importante preservar la ubicación de una característica. Debido a que la agrupación resume las respuestas en todo un vecindario, es posible usar menos unidades de agrupación que unidades detectoras, al informar estadísticas resumidas para las regiones de agrupación espaciadas k píxeles de distancia en lugar de 1 píxel de distancia. Esto mejora la eficiencia computacional de la red porque la siguiente capa tiene, aproximadamente, k veces menos entradas para procesar. Para muchas tareas, la agrupación es esencial para manejar entradas de diferentes tamaños. Por ejemplo, si se quiere clasificar imágenes de tamaño variable, la entrada a la capa de clasificación debe tener un tamaño fijo. Esto, generalmente, se logra variando el tamaño de un conjunto entre regiones de agrupación para que la capa de clasificación siempre reciba el mismo número de estadísticas de resumen, independientemente del tamaño de entrada.

2.2.3. Redes *Transformer*

Un *Transformer* (Vaswani y cols., 2017) es un modelo de transducción de secuencias basado íntegramente en *atención* que reemplaza el uso de capas recurrentes (capas cuyas salidas sirven de entrada a capas anteriores) o convolucionales (descriptas en la sección anterior), comúnmente utilizadas en arquitecturas codificador-decodificador, por un mecanismo de auto-atención multicabezal. Este modela dependencias globales entre la entrada y la salida permitiendo un alto nivel de paralelización y, por lo tanto, requiriendo mucho menos tiempo para entrenar. Un modelo de transducción, o secuencia-a-secuencia, es una red neuronal que transforma una secuencia dada de elementos, como una sucesión de palabras de una oración, en otra secuencia. En este contexto, la atención es una técnica que imita la atención cognitiva.

Realza las partes importantes de los datos de entrada y disminuye el resto con la idea de que la red debe destinar más recursos (poder computacional) a las partes pequeñas pero importantes de los datos. Qué parte de los datos es más importante que otra depende del contexto, y se aprende durante el entrenamiento mediante ejemplos.

En los modelos neuronales de transducción de secuencias con una estructura codificador-decodificador, el codificador mapea una secuencia de entrada de representaciones de símbolos (x_1, \dots, x_n) a una secuencia de representaciones continuas $\mathbf{z} = (z_1, \dots, z_n)$. Dado \mathbf{z} , el decodificador genera una secuencia de salida (y_1, \dots, y_m) de símbolos un elemento a la vez. El *Transformer* sigue esta arquitectura general, pero hace uso de capas apiladas de auto-atención evitando, completamente, el uso de recurrencias al contrario que en la mayoría de modelos de este tipo. De esta manera, se elimina la restricción de computación secuencial inherente a la recurrencia, logrando significativamente más paralelización.

Una función de atención se puede describir como el mapeo de una consulta (*query*) y un conjunto de pares clave-valor (*key-value*) a una salida, donde la consulta, las claves, los valores y la salida son todos vectores. Esta última se calcula como una suma ponderada de los valores, donde el peso asignado a cada valor se calcula mediante una función de compatibilidad de la consulta con la clave correspondiente. El *Transformer* usa la atención producto escalar afectada por un factor de escala ya que es mucho más rápida y más eficiente en el espacio en la práctica. Se puede implementar utilizando un código de multiplicación de matrices altamente optimizado.

En lugar de realizar una única función de atención es posible proyectar linealmente las consultas, claves y valores varias veces con diferentes proyecciones lineales aprendidas. En cada una de estas versiones proyectadas se realiza la función de atención en paralelo, produciendo varios valores de salida que se concatenan y, una vez más, se proyectan, lo que da como resultado los valores finales. La atención multicabezal (*Multi-Head Attention*) permite que el modelo atienda, conjuntamente, la información de diferentes subespacios de representación en diferentes posiciones.

Dado que el *Transformer* no contiene recurrencias ni convoluciones, para que el modelo haga uso del orden de la secuencia debe inyectar alguna información sobre la posición relativa o absoluta de los elementos en la secuencia. Esto se logra mediante *codificaciones posicionales* a los *embeddings* que permiten establecer la posición de cada elemento de la secuencia de entrada. Estas codificaciones, cuya dimensión es la misma de los *embeddings*, se suman a estos para obtener una representación final de la secuencia.

2.2.4. *Dropout*

El *dropout* es una forma computacionalmente económica de regularizar una red neuronal profunda (Srivastava, Hinton, Krizhevsky, Sutskever, y Salakhutdinov, 2014). Este funciona mediante la eliminación probabilística de las entradas de una capa, que pueden ser variables de entrada en la muestra de datos o activaciones de una capa anterior. En esta técnica, neuronas seleccionadas al azar se ignoran durante el entrenamiento. Esto significa que su contribución a la activación de las neuronas descendentes se elimina temporalmente en la propagación hacia adelante y las actualizaciones de peso no se aplican en la retropropagación.

A medida que una red neuronal aprende, los pesos de las neuronas se adaptan a su contexto dentro de la red, ajustándose a características específicas que brindan cierta especialización. Las neuronas vecinas se vuelven dependientes de esta especialización, que si se lleva demasiado lejos puede resultar en un modelo frágil, en exceso especializado para los datos de entrenamiento. Si algunas neuronas se *apagan* aleatoriamente durante el entrenamiento, otras neuronas tendrán que intervenir y manejar la representación requerida para hacer predicciones para las neuronas faltantes. Esto da como resultado que la red aprenda múltiples representaciones internas independientes haciendo que se vuelva menos sensible a los pesos específicos de las neuronas, que tenga una mejor capacidad de generalización y que disminuya la probabilidad que se ajuste a los datos de entrenamiento. En otras palabras, ayuda a prevenir el sobreajuste.

2.2.5. *Batch Normalization*

Las redes neuronales profundas pueden ser sensibles a los pesos aleatorios iniciales y configuración del algoritmo de aprendizaje. Una posible razón de esta dificultad es que la distribución de las entradas de las capas profundas de la red puede cambiar después de cada mini-lote cuando se actualizan los pesos. Este cambio se conoce con el nombre técnico de *cambio covariable interno*. La *batch normalization* (normalización de lote) es una técnica para entrenar redes neuronales muy profundas que estandariza las entradas a una capa para cada mini-lote. Esto tiene el efecto de estabilizar el proceso de aprendizaje y reducir, drásticamente, el número de épocas necesarias para el entrenamiento, además de proporcionar cierta regularización, lo que reduce el error de generalización (Ioffe y Szegedy, 2015).

Un modelo profundo se actualiza capa por capa hacia atrás, desde la salida hasta la entrada, utilizando una estimación de error que asume que los pesos en las capas antes de la capa actual son fijos. El gradiente indica có-

mo actualizar cada parámetro, bajo el supuesto de que las otras capas no cambian, pero en la práctica, se actualizan todas las capas en simultáneo. Debido a que todas las capas se cambian durante una actualización, el procedimiento de actualización siempre persigue un objetivo en movimiento. Por ejemplo, los pesos de una capa se actualizan dada la expectativa de que la anterior genere valores con una distribución determinada. Es probable que esta distribución cambie después de que se actualicen los pesos de la capa anterior. Esto ralentiza el entrenamiento al requerir tasas de aprendizaje más bajas y una inicialización cuidadosa de los parámetros. Además hace que sea notoriamente difícil entrenar modelos con saturación de no linealidades.

La normalización de lote es una técnica para ayudar a coordinar la actualización de múltiples capas en el modelo, y proporciona una forma elegante de reparametrizar casi cualquier red profunda. La reparametrización reduce significativamente el problema de coordinar actualizaciones en muchas capas. Esto se logra escalando la salida de la capa, específicamente estandarizando las activaciones de cada variable de entrada por mini-lote, como las activaciones de un nodo de la capa anterior. La estandarización se refiere al cambio de escala de los datos para que tengan una media de cero y una desviación estándar de uno. Estandarizar las activaciones de la capa anterior significa que las suposiciones que hace la capa posterior sobre la extensión y distribución de las entradas durante la actualización del peso no cambiarán, al menos no de manera dramática. Esto tiene el efecto de estabilizar y acelerar el proceso de entrenamiento de las redes neuronales profundas. Es importante destacar que el algoritmo de retropropagación se actualiza para operar sobre las entradas transformadas, y el error también se usa para actualizar la nueva escala y los parámetros de cambio aprendidos por el modelo. La estandarización se aplica a las entradas de la capa, es decir, las variables de entrada o la salida de la función de activación de la capa anterior. Dada la elección de la función de activación, la distribución de las entradas a la capa puede ser bastante no gaussiana. En este caso, puede resultar beneficioso estandarizar la activación sumada, antes de la función de activación en la capa anterior.

Capítulo 3

Trabajo realizado: Motor de inferencia

Como se mencionó en el Capítulo 1, el sistema de búsqueda de interacciones fármaco-gen desarrollado en este proyecto se encuentra dividido en dos grandes módulos: uno correspondiente al motor de inferencia basado en redes neuronales y, el otro, a la aplicación web que utiliza dicho motor. Cada módulo se desarrolló siguiendo la metodología correspondiente mencionadas en la Sección 1.4. En este capítulo se presenta el trabajo realizado para obtener el motor de inferencia neuronal.

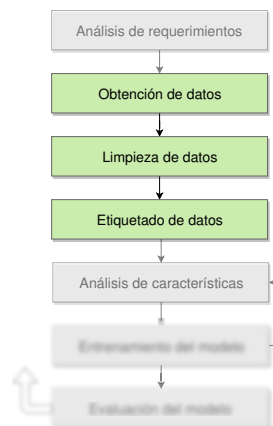
3.1. Análisis de requerimientos

Los requerimientos, tanto funcionales como no funcionales, que guiaron este desarrollo fueron especificados, anteriormente, en la Sección 1.3.



3.2. Obtención de datos

En esta sección se describe todo el trabajo necesario para obtener los datos de entrenamiento y dejarlos listos para su utilización por los modelos neuronales.



3.2.1. Base de datos de etiquetas

Para poder crear un sistema de aprendizaje automático fiable, con respecto al gran volumen de datos disponibles, se necesita un conjunto de entrenamiento etiquetado lo suficientemente extenso y representativo. El trabajo realizado por K. C. Cotto *et al.* con su base de datos de interacciones fármaco-gen *DGIdb* (Cotto y cols., 2017) es un medio útil para suplir esta parte. La misma es un recurso web que consolida la información de más de veintisiete fuentes de datos dispares que describen las interacciones entre fármacos y genes, confeccionada mediante un trabajo de curación manual experta. Esta base de datos contiene etiquetas de fármacos, genes e interacciones presentes en las publicaciones y, además, brinda los enlaces para acceder a las mismas en sus respectivos sitios web.

DGIdb se encuentra disponible para su descarga en <https://github.com/griffithlab/dgi-db>. Esta contiene un total de 44 tablas que organizan la información de genes, drogas, publicaciones e interacciones, e información adicional como moléculas bioactivas con propiedades similares a un fármaco, atributos de entidades, nivel de confianza de las fuentes, tipo de fuente, etc.

Considerando nuestro problema en cuestión, en *DGIdb* se encuentran registrados un total de 41 102 genes, 9501 drogas y 51 tipos de interacciones. Contiene un total de 36 979 publicaciones de las que se dispone el identificador de *PubMed* o PMID (*PubMed Identification*). *PubMed* es un motor de búsqueda de libre acceso, ofrecido por la Biblioteca Nacional de Medicina de los Estados Unidos, que permite consultar principalmente los contenidos de la base de datos de bibliografía médica MEDLINE, y otra variedad de revistas científicas. Esto permite acceder a resúmenes y referencias bibliográficas

de estos artículos de investigación biomédica. MEDLINE, a su vez, tiene alrededor de 4800 revistas publicadas en los Estados Unidos y en más de 70 países del mundo, reuniendo actualmente más de 30 000 000 de citas.

De esas publicaciones presentes en *DGIdb*, sólo son útiles aquellas que desprenden etiquetas completas, es decir, gen + droga + tipo de interacción + publicación fuente. Ya que para poder entrenar los modelos neuronales es necesario tener la información completa, los datos se reducen a 10 175 ejemplos que corresponden a 5115 publicaciones distintas (una publicación puede tener asociada más de una etiqueta). Si se consideran sólo estos ejemplos, el número de genes, drogas y tipo de interacciones presentes se reduce a 899, 1119 y 33, respectivamente. Los genes y drogas no tienen una única forma de llamarse, existen alias para ellos. El total de alias en las publicaciones de interés para las drogas es de 149 169 y el de genes de 12 057.

Considerando el análisis anterior se confecciona una base de datos local de etiquetas con la información pertinente para el entrenamiento de los modelos neuronales (genes y sus alias, drogas y sus alias, interacciones y publicaciones). Para esta tarea se utiliza el sistema de bases de datos relacionales de código abierto *PostgreSQL*.

Esta base de datos se construyó a partir de la siguiente consulta a *DGIdb* que reúne todas las etiquetas completas:

```
1  create or replace view vista_ifg as (  
2  select  
3      g.id as gen_id,  
4      g.name as gen,  
5      d.id as droga_id,  
6      d.name as droga,  
7      ict.id as interaccion_id,  
8      ict.type as interaccion,  
9      p.pmid as publicacion  
10 from  
11     interaction_claims ic  
12 left join gene_claims gc  
13     on ic.gene_claim_id = gc.id  
14 left join drug_claims dc  
15     on ic.drug_claim_id = dc.id  
16 left join interaction_claims_publications icp  
17     on icp.interaction_claim_id = ic.id  
18 left join publications p  
19     on icp.publication_id = p.id  
20 left join interaction_claim_types_interaction_claims  
ictic  
21     on ictic.interaction_claim_id = ic.id  
22 left join interaction_claim_types ict  
23     on ictic.interaction_claim_type_id = ict.id  
24 left join genes g
```

```

25         on gc.gene_id = g.id
26     left join drugs d
27         on dc.drug_id = d.id
28
29     where
30         g.id is not null and
31         d.id is not null and
32         ict.id is not null and
33         p.pmid is not null
34     group by
35         g.id, gen, d.id, droga, ict.id, interaccion,
publicacion
36     order by
37         gen, droga, interaccion, publicacion
38 )

```

La tabla inicial, con las interacciones entre fármacos y genes, es *interaction_claims*, la cual vincula:

- un gen de la tabla *gene_claims*
- un fármaco de la tabla *drug_claims*
- uno o varios tipos de interacción de la tabla *interaction_claim_types* mediante la tabla de unión *interaction_claim_types_interaction_claims*
- alguna publicación de la tabla *publications* mediante la tabla de unión *interaction_claims_publications*

Toda la información que se desprende de esta consulta es volcada a las tablas de la base de datos local de etiquetas que pueden observarse en la Figura 3.1.

3.2.2. *Web scraping*

A partir de esta información recolectada, se procede a la obtención de los archivos de las publicaciones mediante la aplicación de *web scraping*. El mismo es una técnica que se utiliza para extraer información de sitios web, generalmente simulando la navegación de un humano en el sitio.

En primer lugar, se deben obtener las URL de las fuentes originales de las publicaciones a partir de la página de *PubMed* del artículo en cuestión. Todos los artículos presentes en la base de datos se encuentran con su PMID, y dado este número, se deduce su URL en *PubMed* de la forma:



Figura 3.1: **Esquema de la base de datos.** Cada entrada de la tabla `interaccion_farmaco_gen` representa un ejemplo con el gen, la droga, el tipo de interacción entre ambos y la publicación de origen de esta información.

`https://pubmed.ncbi.nlm.nih.gov/{pmid}`

Luego, en este sitio, la URL del artículo en su fuente original se encuentra, si está disponible, en uno o más enlaces junto al texto *Full Text Links*. Se tuvo en cuenta este patrón, y se implementó un *scraper* sencillo, que dado un PMID, obtiene este enlace automáticamente. Cuando se ejecutó este *scraper* sobre los 5115 artículos, se obtuvieron enlaces de 4576 publicaciones, correspondientes a casi 300 fuentes distintas. Sin embargo, la gran mayoría reside en unas pocas fuentes principales, por ejemplo *ScienceDirect*, *Doi.org*, *Springer*, *Wiley*, *Nature*, *Oxford Academic*, entre otras (ver Figura 3.2).

Para las fuentes principales se implementaron, una vez analizadas específicamente, *scrapers* que permiten descargar automáticamente el archivo PDF del *paper* correspondiente. Estos scripts *parsean* el contenido HTML y analizan las etiquetas adecuadas para obtener la información de interés, en este

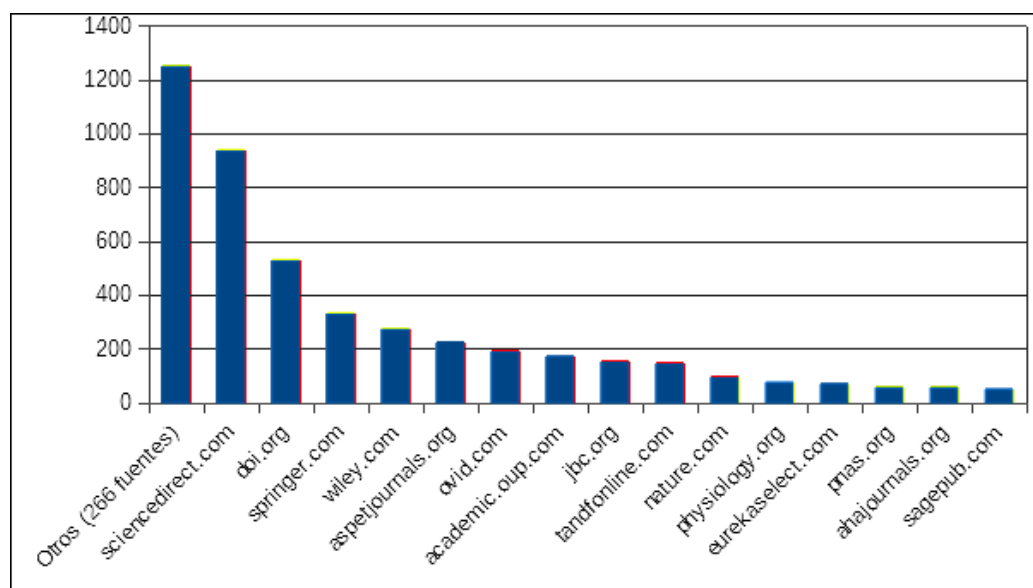


Figura 3.2: Cantidad de artículos por fuente

caso, la URL directa de cada archivo PDF. De las 4576 publicaciones con enlace disponible, se lograron descargar 4021 archivos. Esta disminución se debe, en primer a lugar, a la falta de *scraper* para las fuentes no consideradas principales. En segundo lugar, a la imposibilidad de conectarse al servidor por problemas de este o por inhabilitación del acceso al artículo, entre otras. Para suplir esta falta de archivos, se implementó, además, un *scraper* que descarga los *abstracts* (resúmenes) de los artículos junto al título y las palabras clave desde el mismo *PubMed*.

Por otro lado, se implementó también un *scraper* para el sitio web *GeneCards*¹, una base de datos que provee información exhaustiva sobre todos los genes humanos, con el objetivo de obtener más cantidad de alias para los mismos (esta información es añadida a la base de datos).

3.2.3. Pre-procesamiento de los datos

Teniendo en cuenta que los modelos neuronales trabajan con información numérica, para poder realizar NLP es necesario convertir el contenido de dichas publicaciones en un formato que estos sean capaz de procesar. Para ello en primer lugar se convierten todos los archivos PDF descargados (4021) a formato de texto plano. Se utiliza la herramienta *pdftotext*² para

¹<https://www.genecards.org>

²<https://poppler.freedesktop.org>

los archivos que soportan conversión directa, es decir, aquellos donde el texto puede extraerse, directamente, del documento. Para los casos donde esto no es posible (por ejemplo se trata de documentos escaneados), se aplica reconocimiento óptico de caracteres (OCR) mediante la herramienta Tesseract OCR³. Este proceso no pudo realizarse para ciertos archivos (presencia de caracteres no romanos, documentos protegidos, etc.), por lo tanto, se obtienen en total como resultado 3910 archivos de texto plano. En este punto, se recurre a la información adicional recolectada mediante el *scraper* de *abstracts* para suplir las publicaciones faltantes (1205), luego del proceso de descarga y conversión; de 1116 se pudo obtener *abstract* y título, y de 89 únicamente el título.

Si se parte de estos archivos de texto, el siguiente paso a realizar es una limpieza de los mismos con el fin de eliminar caracteres basura que puedan complicar la comprensión por parte de los modelos neuronales. A su vez, determinar con más precisión la ocurrencia de entidades biomédicas (es decir, encontrar genes y drogas en el texto según sus nombres o alias). Esta limpieza consiste en pasar todo el texto a minúsculas, convertir las letras griegas (bastante comunes en denominaciones de fármacos) a una representación romana (su nombre) y reemplazar las entidades HTML por sus caracteres equivalentes.

Como se mencionó anteriormente, los genes y drogas pueden poseer alias para identificarlos, además de sus nombres; en promedio cada uno posee 48 alias distintos (basado en la base de datos de etiquetas), además de que pueden contener caracteres no alfanuméricos (por ejemplo "*1-methyl-6-phenyl-1h-imidazo[4,5-b]pyridin-2-amine*"), lo que dificulta el posterior proceso de *tokenización* (el cual será explicado más adelante). Con el fin de condicionar los datos para servir de entrada a los modelos neuronales, se busca reemplazar las ocurrencias de estos alias por la entidad biomédica equivalente. Para ello, se establece una codificación única alfanumérica para cada entidad. Esta codificación es de la forma **eX**, donde **e** es **g** para los genes y **d** para las drogas, y **X** es un identificador numérico que corresponde al *primary key* de la tabla respectiva en la base de datos de etiquetas; ejemplos son *g264* para el gen *aadacl2* y *d765* para la droga *aspirin*.

En esta instancia se presentan dos dificultades. Por un lado, hay que tener en cuenta que un alias puede estar repetido para distintas entidades dificultando la determinación de a cuál realmente corresponde. Por otro lado, existen alias que se corresponden con palabras del lenguaje natural (inglés) ocasionando que una palabra de este tipo sea identificada como una entidad cuando, realmente, no lo es (por ejemplo el alias *yes* para *yes1* u *overall*

³<https://tesseract-ocr.github.io>

para *roxithromycin*). De este modo, en la búsqueda de ocurrencias se consideran estas dos situaciones realizando tres etapas diferentes de búsqueda y reemplazo, según resulte necesario, ateniéndose a las etiquetas desprendidas de cada publicación. En primer lugar, sólo se consideran aquellos alias que no son palabras del lenguaje natural y que no tienen asociado más de un gen o droga. Si esta etapa no logra satisfacer las etiquetas, en segundo lugar, se consideran además los que sí tienen asociado más de un gen o droga, reemplazándolo por todas sus posibles denominaciones. En último lugar, se consideran todos los alias. Para los reemplazos se ha optado por utilizar los nombres codificados de las entidades junto a la cadena *xxx*, ubicada antes y después de los mismos, por ejemplo *xxxd342xxx* y *xxvg261xxx*, para asegurarse de no haber ambigüedad con alguna otra palabra que pueda estar mencionada.

El siguiente ejemplo demuestra todo este proceso:

Furthermore, there are also affinity differences between pramipexole and **ropinirole**, with pramipexole more selective for the **D₃** subtype receptors within the **D₂** sub-family, than **ropinirole**. The non-ergot **rotigotine**, shows low selectivity among the **DA** receptors **D₁**, **D₂**, **D₃**, **D₄** and **D₄**, also indicating low selectivity for **DAergic**-receptor sub-families. With the exception of **D₄ receptor**, **rotigotine** exhibits a binding profile towards **DA** receptors similar to that of **DA**, with a 30-fold higher affinity for the receptors than **DA**.

furthermore, there are also affinity differences between pramipexole and **xxxd3145xxx**, with pramipexole more selective for the **d3** subtype receptors within the **d2** sub-family, than **xxxd3145xxx**. the non-ergot **xxxd3148xxx**, shows low selectivity among the **da** receptors **d1**, **d2**, **d3**, **d4** and **d5**, also indicating low selectivity for **daergic**-receptor sub-families. with the exception of **xxvg658xxx**, **xxxd3148xxx** exhibits a binding profile towards **da** receptors similar to that of **da**, with a 30-fold higher affinity for the receptors than **da**.

Obsérvese que **D₄ receptor** es un alias para el gen **DRD4**, codificado como **g658**.

3.2.4. Generación de casos negativos

Para que las redes aprendan a clasificar aquellos casos que no entran en ninguna de las clases etiquetadas, es decir, casos donde las entidades no interactúan o se desconoce si lo hacen, es necesario contar con ejemplos que

contemplan esta situación. Para ello, se genera una clase *sin interacción* que tiene en cuenta estos casos, suponiendo que si en un artículo se nombran genes y drogas pero no se relacionan mediante alguna etiqueta, entonces se puede decir que se desconoce si lo hacen. Para generar ejemplos de esta clase se consideran, por publicación, todas las etiquetas para la misma; cada una de estas etiquetas es descartada si las entidades correspondientes no pudieron ser halladas en el texto, en caso contrario, se *cruzan* las entidades de las distintas etiquetas y se las categoriza como la clase sintética *sin_interacción*. En el siguiente ejemplo se muestran las tres etiquetas de las que dispone la publicación de PMID 2871880:

```
2871880,g91,d2812,antagonist
2871880,g91,d2991,antagonist
2871880,g92,d123,antagonist
```

Al realizar la cruce entre las distintas entidades de estas etiquetas, se obtiene el siguiente conjunto:

```
2871880,g91,d2812,antagonist
2871880,g91,d2991,antagonist
2871880,g91,d123,sin_interacción
2871880,g92,d2812,sin_interacción
2871880,g92,d2991,sin_interacción
2871880,g92,d123,antagonist
```

Como no se encontraron ocurrencias en dicha publicación del gen de codificación *g91*, las etiquetas que lo contienen son descartadas, quedando finalmente las siguientes:

```
2871880,g92,d2812,sin_interacción
2871880,g92,d2991,sin_interacción
2871880,g92,d123,antagonist
```

Considerando todos los factores nombrados anteriormente que propiciaron la pérdida de etiquetas, de las 10 175 iniciales quedan 2967 (correspondientes a 2350 publicaciones distintas), a las que se añaden 1283 sintéticas más (por la razón explicada previamente), sumando en total 4250 etiquetas para el entrenamiento de los modelos neuronales distribuidas en 30 tipos diferentes de clases (interacciones). Estas se observan en el Cuadro 3.1.

| Tipo de interacción | Cantidad de ejemplos |
|-------------------------------|----------------------|
| sin_interacción | 1283 |
| inhibitor | 1048 |
| agonist | 658 |
| antagonist | 649 |
| cofactor | 161 |
| binder | 83 |
| inducer | 57 |
| antibody | 51 |
| partial agonist | 36 |
| ligand | 27 |
| product of | 26 |
| potentiator | 25 |
| multitarget | 25 |
| modulator | 22 |
| negative modulator | 20 |
| activator | 19 |
| other | 15 |
| suppressor | 9 |
| blocker | 8 |
| stimulator | 6 |
| allosteric modulator | 5 |
| unknown | 3 |
| partial antagonist | 3 |
| inverse agonist | 2 |
| inhibitory immune response | 2 |
| inhibitor, competitive | 2 |
| binding | 2 |
| positive allosteric modulator | 1 |
| neutralizer | 1 |
| adduct | 1 |

Cuadro 3.1: Cantidad de ejemplos por tipo de interacción.

3.2.5. Generación de secuencias de ejemplos

Como se mencionó en su momento, las redes neuronales sólo trabajan con números y no con palabras. Para convertir cada palabra a un vector numérico equivalente (su *embedding*), en primer lugar se convierte cada ejemplo en una secuencia numérica con el fin de determinar a qué se considera una palabra.

En este punto entra en juego el proceso de *tokenización*: dado un texto y en base a una regla, se separa el mismo en *tokens* (en este caso palabras) asignándole a cada uno un número único. De esta forma, se genera un vocabulario a partir de todas las palabras nombradas en todas las publicaciones, donde cada una posee un número de secuencia único, ordenándolas en base a su frecuencia de aparición. Es decir, una palabra que es nombrada muchas veces se encuentra antes en el vocabulario generado que una que se nombra pocas veces. Además se extiende el vocabulario con dos palabras específicas que sirven para identificar al gen y a la droga de una etiqueta en el texto dado, con el fin de distinguirlos con respecto a las demás entidades biomédicas que están presentes, pero no forman parte de la etiqueta; estos *tokens* especiales son <GEN> y <DROGA>, los cuales a su vez poseen su propio número único de secuencia. A partir de los datos de entrenamiento disponibles se obtuvo un vocabulario de 506 332 palabras, incluyendo las dos específicas anteriores, donde la palabra número 506331 corresponde a <GEN> y la 506332 a <DROGA>.

Si se parte de este vocabulario, de los textos de las publicaciones y de las etiquetas, se genera, para cada una, la secuencia correspondiente reemplazando cada palabra por su número en el vocabulario, teniendo en cuenta que se asignan los *tokens* especiales a las entidades biomédicas de la etiqueta. Para ejemplificar, supóngase que se *tokeniza* y genera un vocabulario a partir del siguiente texto:

- *In this article g1, g2 and g3 are tested against d1 and d2.*

El resultado de esto es el siguiente:

| | | | | | | | | | | | | | |
|-----|---------|-----|---------|----|----|----|----|----|----|--------|------|-------|---------|
| and | against | are | article | d1 | d2 | in | g1 | g2 | g3 | tested | this | <GEN> | <DROGA> |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

Contando con las etiquetas:

| Gen | Droga | Interacción |
|-----|-------|-------------|
| g1 | d2 | int1 |
| g3 | d1 | int2 |

se generan las siguientes secuencias para cada ejemplo respectivamente:

| | | | | | | | | | | | | |
|----|------|---------|-------|----|-----|----|-----|--------|---------|----|-----|---------|
| In | this | article | <GEN> | g2 | and | g3 | are | tested | against | d1 | and | <DROGA> |
| 7 | 12 | 4 | 13 | 9 | 1 | 10 | 3 | 11 | 2 | 6 | 1 | 14 |

| | | | | | | | | | | | | |
|----|------|---------|----|----|-----|-------|-----|--------|---------|---------|-----|----|
| In | this | article | g1 | g2 | and | <GEN> | are | tested | against | <DROGA> | and | d2 |
| 7 | 12 | 4 | 8 | 9 | 1 | 13 | 3 | 11 | 2 | 14 | 1 | 6 |

Las redes neuronales necesitan trabajar con conjuntos de datos homogéneos, es decir, los ejemplos de entrada deben poseer, todos, la misma longitud. Como se sabe, las secuencias son de longitudes variables por lo que se vuelve imprescindible redimensionarlas a una longitud fija. En base a un análisis de histograma de las posiciones de aparición de genes y drogas de interés (aquellos etiquetados) en las publicaciones (visto en Figura 3.3), se decide utilizar secuencias de longitud 7500, 10 000 y 20 000 para realizar distintas pruebas; las secuencias más cortas se completan con ceros (*zero padding* posterior) y, las más largas, se truncan.

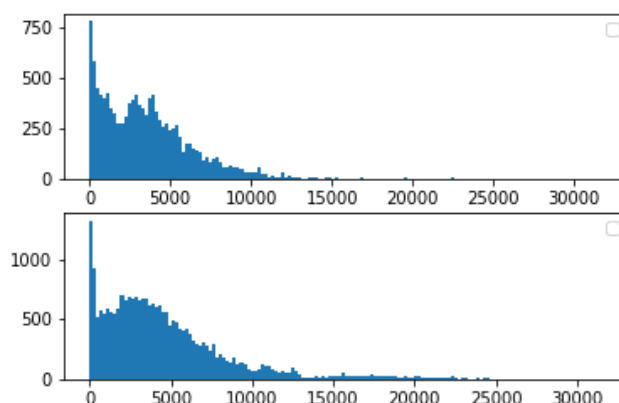


Figura 3.3: Frecuencia de aparición de entidades biomédicas (eje y) por posición en las publicaciones (eje x). Imagen superior: apariciones de genes. Imagen inferior: apariciones de drogas.

En este punto, se necesita asignar a cada *token* el vector numérico o *embedding* correspondiente (ver Subsección 2.2.1). Hay varias formas de hacer esto; en este trabajo se utilizan dos maneras para comparar: generarlos específicamente para el corpus disponible mediante el algoritmo de word2vec, lo que resulta en un conjunto de 506 329 vectores distintos, remarcando que contempla terminología específica del dominio de la medicina genómica (debido

al corpus generador); y usar un conjunto preentrenado mediante el algoritmo de GloVe⁴. Este último está basado en estadísticas agregadas globales de co-ocurrencia palabra-a-palabra, y el conjunto de datos utilizado fue entrenado a partir del contenido de Wikipedia (inglés) y el *English Gigaword Fifth Edition*, conteniendo al final un total de 400 000 palabras.

Asimismo, se deben adicionar tres vectores más; uno de ellos es el vector de *embedding* nulo (vector de ceros) que se utiliza para representar aquellas palabras que no existen en el vocabulario, mientras que los otros dos son *embeddings* especiales, únicos, utilizados para representar las entidades bi-médicas (gen y droga) etiquetadas en un ejemplo. Para asegurarse de que estos *embeddings* especiales son únicos, se extienden los demás en dos posiciones con valor cero. El vector especial de gen tiene todos elementos ceros en todas las posiciones, excepto en la penúltima que posee un 1, mientras que el de droga es similar salvo que el 1 está en la última posición. Se ilustra lo dicho en el siguiente ejemplo:

| | | | | | | | | | |
|--------------------|---------------|---|---------|---------|------|----------|----|---|---|
| * Vector 0: | (desconocida) | [| 0, | 0, | ..., | 0, | 0, | 0 |] |
| Vector 1: | <i>the</i> | [| 0.3487, | 0.5689, | ..., | -0.3278, | 0, | 0 |] |
| * Vector $n - 1$: | <GEN> | [| 0, | 0, | ..., | 0, | 1, | 0 |] |
| * Vector n : | <DROGA> | [| 0, | 0, | ..., | 0, | 0, | 1 |] |

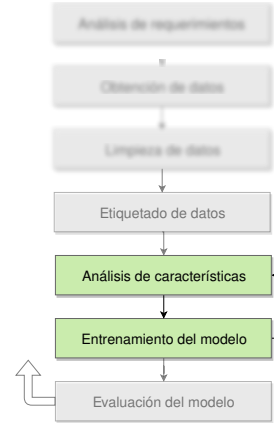
Los algoritmos de las redes neuronales son los encargados de realizar el mapeo entre los números de las secuencias que conforman cada ejemplo, con los vectores de *embeddings* correspondientes. Para ello se precisa de una matriz que permita establecer este mapeo, por lo que se generan matrices de *embeddings* (una para word2vec y otra para GloVe). Aquí, el índice de cada fila representa el número de secuencia, y la fila es el vector de *embedding*, de la misma manera como se ilustra en el ejemplo anterior. De este modo, un ejemplo de entrada para la red obtiene una representación matricial, si se consideran los vectores de word2vec, de la siguiente manera:

⁴<https://nlp.stanford.edu/projects/glove>

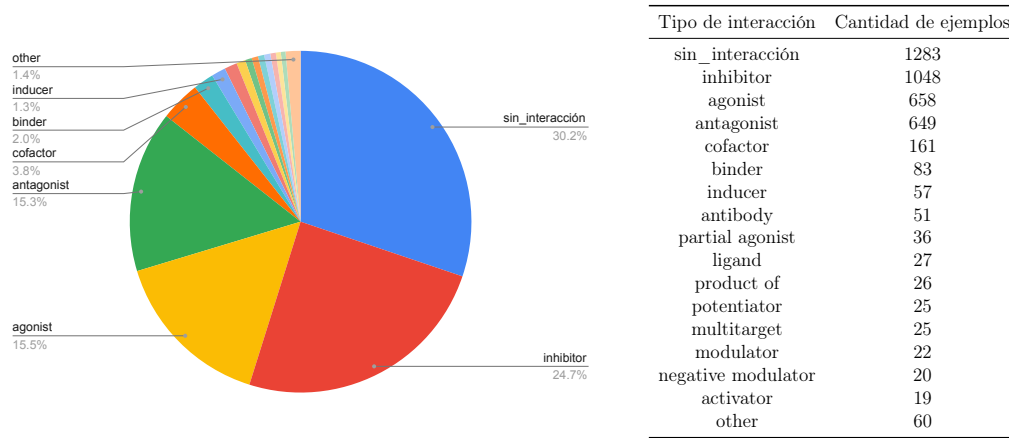
| Ejemplo | Ocurrencias | Secuencia | <i>Embedding</i> | | | | | | |
|-----------------|-----------------|-----------|------------------|---|---------|---------|-----|---------|-----------|
| It | it | 72 | → | [| -0.0091 | 0.0053 | ... | -0.0072 | 0.0 0.0] |
| has | has | 74 | → | [| 0.0073 | -0.0086 | ... | -0.006 | 0.0 0.0] |
| been | been | 70 | → | [| -0.0057 | -0.0019 | ... | -0.0072 | 0.0 0.0] |
| reported | reported | 160 | → | [| 0.0004 | 0.0085 | ... | -0.0007 | 0.0 0.0] |
| that | that | 17 | → | [| 0.0067 | 0.0056 | ... | 0.0 | 0.0 0.0] |
| xxxd3014xxx | <DROGA> | 506331 | → | [| 0.0 | 0.0 | ... | 0.0 | 0.0 1.0] |
| treatment | treatment | 58 | → | [| -0.0097 | -0.0069 | ... | -0.0004 | 0.0 0.0] |
| induced | induced | 82 | → | [| 0.0037 | 0.0027 | ... | 0.0047 | 0.0 0.0] |
| a | a | 5 | → | [| -0.0053 | -0.0088 | ... | 0.0069 | 0.0 0.0] |
| more | more | 129 | → | [| -0.0082 | -0.0002 | ... | -0.0019 | 0.0 0.0] |
| rapid | rapid | 811 | → | [| 0.009 | 0.0026 | ... | 0.0082 | 0.0 0.0] |
| progression | progression | 995 | → | [| -0.0001 | 0.0075 | ... | -0.0091 | 0.0 0.0] |
| of | of | 2 | → | [| -0.005 | 0.0049 | ... | -0.0026 | 0.0 0.0] |
| atherosclerotic | atherosclerotic | 7785 | → | [| -0.0084 | 0.0065 | ... | -0.0081 | 0.0 0.0] |
| lesions | lesions | 1794 | → | [| -0.0013 | -0.0024 | ... | -0.009 | 0.0 0.0] |
| in | in | 4 | → | [| -0.008 | -0.0008 | ... | 0.0094 | 0.0 0.0] |
| xxvg177xxx | <GEN> | 506330 | → | [| 0.0 | 0.0 | ... | 0.0 | 1.0 0.0] |

3.3. Entrenamiento del modelo

A continuación, se explica el proceso de entrenamiento de los modelos neuronales, y se detalla cómo se configura la salida de los mismos para predecir entre las distintas clases disponibles. Como se mencionó en su momento, los tipos de redes neuronales utilizados pueden extraer características automáticamente, por lo que esa etapa se incluye con el entrenamiento como un todo.



Como se especificó anteriormente, existe un total de 30 clases; sin embargo muchas poseen muy pocos ejemplos, lo cual no resulta útil para el entrenamiento de los modelos. Es por ello, que se opta por agrupar los tipos de interacción con poca cantidad de ejemplos (menor a 10) en la clase **other**, quedando un total de 17 clases finales, como se muestra a continuación:

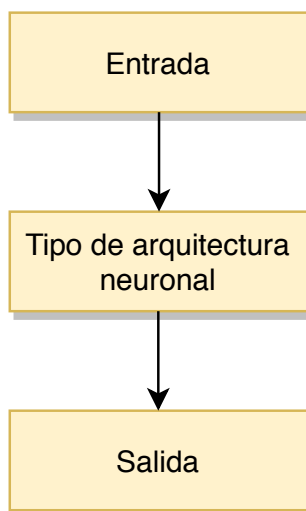


Se debe aclarar que el tipo de interacción **other** indica que existe una interacción de algún tipo entre un gen y una droga, mientras que la clase **sin_interacción** indica que entre las entidades biomédicas no hay una interacción conocida (al menos al momento de la confección del modelo).

Antes de comenzar a entrenar los modelos, se separan las etiquetas en dos conjuntos, uno para entrenamiento y otro para prueba con una relación 80 % (3400) - 20 % (850) respectivamente y de manera estratificada. Esto se refiere a que, en ambos conjuntos, se mantiene la proporción de ejemplos de cada clase. El conjunto de prueba es reservado para realizar la evaluación final de los modelos (proceso detallado en el Capítulo 4).

Para el entrenamiento se realiza validación cruzada utilizando un algoritmo de *k-folding* estratificado con tres particiones. Esto divide al conjunto de entrenamiento en dos partes para cada partición, una para entrenar y la otra para validar. Como la cantidad de clases no se encuentra balanceada, es decir que hay distinta cantidad de ejemplos para cada clase, se crea un vector de pesos para el entrenamiento, para que las clases con mayor cantidad de ejemplos pesen menos que las clases con menor cantidad de ellos.

Ambas redes tienen la siguiente estructura genérica:



El bloque de entrada representa las entradas como se detalló en la Sección 3.2. El segundo bloque representa el tipo de arquitectura de red neuronal utilizado, en este caso CNN o *Transformer*. La salida es un vector de longitud N, donde N es la cantidad de clases (17). Constituye una distribución de probabilidades de las interacciones (es decir que los valores de sus elementos están entre 0 y 1, y que la suma de todos ellos es igual a 1), donde el elemento máximo es interpretado como la clase inferida. Como el aprendizaje es supervisado, se necesita poder comparar la salida devuelta por la red con la salida deseada (el tipo de interacción correcto); es por ello que se aplica un *one hot encoding* de las distintas clases, quedando así vectores de longitud N que tienen 1 en la posición que representa la interacción correspondiente y ceros en el resto. Lo dicho puede observarse en el siguiente ejemplo:

[illegible]

En el siguiente ejemplo se muestra una salida calculada por la red, donde al considerar el máximo elemento se corresponde con la salida deseada:

| Salida calculada | | | | | | | | | | | | | | | | |
|---------------------------------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|--------|-------|
| 0.055 | 0.041 | 0.033 | 0.016 | 0.098 | 0.079 | 0.072 | 0.096 | 0.063 | 0.070 | 0.014 | 0.002 | 0.067 | 0.049 | 0.052 | 0.112 | 0.080 |
| Salida interpretada (máximo elemento) | | | | | | | | | | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ↓ 1 | 0 |
| Salida deseada | | | | | | | | | | | | | | | | |

A continuación se detallan las arquitecturas de los modelos neuronales y el trabajo de selección de los distintos hiperparámetros para cada una, pero antes se especifica el hardware donde se ejecuta todo el proceso.

3.3.1. Recursos de hardware empleados

Se utilizó el servicio *Google Colaboratory*⁵ (llamado normalmente Colab) para realizar el entrenamiento y evaluación de los modelos. El mismo es un servicio gratuito que permite al usuario escribir y ejecutar código Python arbitrario en el navegador, en forma de *notebook* de *Jupyter*⁶. A su vez, brinda acceso gratuito a recursos computacionales, incluidas GPU, lo que lo hace ideal para proyectos de aprendizaje automático y análisis de datos. Los recursos de Colab no son ilimitados, existe una restricción temporal donde cada sesión tiene una vida útil máxima, de hasta 12 horas. Luego, se debe esperar cierta cantidad de tiempo extra (por lo general de 24 horas) para disponer nuevamente de recursos (la vida útil máxima y el tiempo de espera de inactividad pueden variar con el tiempo o en función del uso dado). Sin embargo, el tiempo necesario para entrenar los modelos de este proyecto se mantuvo dentro del margen de disponibilidad de la plataforma. En cada sesión de Colab, los recursos disponibles pueden variar; cada sesión contó, en promedio, con 32 GB de memoria RAM, y distintos tipos de GPU, entre algunas de las disponibles: NVIDIA Tesla P4 de 8 GB de memoria, NVIDIA T4 Tensor Core GPU de 16 GB, NVIDIA Tesla P100 de 16 GB y NVIDIA Tesla K80 de 24 GB.

3.3.2. Red Neuronal Convolutiva (CNN)

En la Figura 3.4 se muestra la arquitectura usada para el modelo CNN. Esta consta de una capa de entrada, la cual es la encargada de tomar las secuencias de ejemplos, alimentando a la siguiente capa (capa de *embedding*). Esta última transforma los datos de entrada en la representación matricial procesable por la red mediante la matriz de *embeddings* establecida (word2vec o GloVe). Luego se utilizan capas de convolución y *pooling*, específicas de este tipo de redes. Como los datos con los que se trabaja son de naturaleza secuencial, el tipo de convolución apropiado para el procesamiento es la convolución 1D o temporal, y el *pooling* es el que toma el máximo global. Existen tantas

⁵<https://colab.research.google.com>

⁶<https://jupyter.org>



Figura 3.4: Arquitectura del modelo CNN.

capaz de convolución y *pooling* en paralelo como tamaños de *kernel* definidos; en este trabajo se utilizan 5 tamaños distintos: 1, 3, 5, 7 y 9, interpretándose como la cantidad de palabras tomadas en conjunto para analizar. Lo obtenido de estas capas paralelas se concatena, luego, en una nueva capa que sirve de entrada a un MLP que cuenta con una capa oculta y una de salida. Cabe aclarar, además, que la arquitectura definida toma en cuenta las estrategias de *dropout* y *batch normalization*, ayudando a prevenir el sobreentrenamiento la primera, y acelerando el proceso de entrenamiento la segunda, pudiendo incluso mejorar el desempeño.

Los hiperparámetros finales del modelo fueron escogidos luego de un proceso continuo de prueba y error, específicamente de entrenamiento y validación con diferentes valores. Se intentó, en primer lugar, que los resultados cumplan los requerimientos establecidos (Sección 1.3) y, a la vez, se trató de maximizar el desempeño del modelo. Como consideración, la selección de estos hiperparámetros se encuentra, además, limitada por los recursos disponibles y por la selección de longitudes de las secuencias de ejemplos para realizar las pruebas (Figura 3.3). Esto implica que sólo se pueda trabajar con ciertos tamaños de dimensión de los vectores de *embedding*. GloVe ofrece sus vectores de *embeddings* preentrenados con longitudes de 50, 100, 200 y 300 elementos, pero por lo dicho anteriormente se utilizan sólo las longitudes de 50 y 100. Para que los resultados sean comparables se generan, por lo tanto, vectores de word2vec de estas mismas longitudes. Por esta limitación de

recursos, el diseño de la arquitectura se encuentra, además, acotado a cinco tamaños distintos de *kernels* (1, 3, 5, 7 y 9), habiendo para cada uno de ellos 100 filtros distintos inicializados aleatoriamente.

Hay que tener en cuenta que los vectores de *embeddings* pueden utilizarse tal cual fueron definidos, o pueden ir adaptándose durante el entrenamiento del modelo, lo que puede llegar a lograr una mejora en el desempeño del mismo. Por este motivo, se realizan pruebas con ambas configuraciones para comparar los resultados obtenidos.

La capa de salida del perceptrón multicapa posee 17 neuronas, una para cada clase, cuya activación es la función *softmax* que se encarga de representar la salida como una distribución de probabilidades, tal como se mencionó en la Sección 3.3. Para definir la cantidad de neuronas en la capa oculta del mismo, se realiza una optimización de este valor utilizando una búsqueda en grilla (*grid search*), en este caso de una dimensión al ser un único parámetro (la grilla no es más que una recta). Este método es, simplemente, una búsqueda exhaustiva en un subconjunto definido manualmente del espacio de valores del parámetro. Aquí se parte de un subconjunto inicial, dentro del rango de 17 y 512 neuronas, refinando la cantidad mediante pruebas pequeñas de entrenamiento y clasificación. Se concluye que, el valor que mejor parece comportarse es de 51 neuronas (correspondiente al triple de la cantidad de clases). La función de activación para esta capa es *relu* (Rectified Linear Unit, definida como $f(x) = \max(0, x)$), la cual es utilizada, también, como función de activación para todas las capas de *batch normalization*. Finalmente, la función utilizada para calcular el error al haber más de dos clases es *categorical crossentropy* y el optimizador (algoritmo de cálculo del gradiente descendiente) es *adam*. Este es computacionalmente eficiente, tiene poco consumo de memoria y es útil para problemas grandes, en términos de datos y parámetros (Kingma y Ba, 2014).

De las pruebas anteriores, también se concluye que un número de 40 épocas es suficiente para realizar el entrenamiento, ya que el error de validación no se reduce más pasada esta cantidad. Además, se implementa el uso de *callbacks* de corte temprano y de reducción de velocidad de aprendizaje. El primero se encarga de finalizar el entrenamiento (para una partición de la validación cruzada) cuando el error de validación no disminuye luego de un número arbitrario de épocas (4 en este trabajo), mientras que el segundo, dada la misma condición (pero luego de 2 épocas), se encarga de reducir la velocidad de aprendizaje con el fin de ajustar finamente los pesos del modelo.

En el Cuadro 3.2 se resumen los datos presentados en esta sección.

| | |
|---|---------------------------------|
| Total de ejemplos | 4250 |
| Total de ejemplos para entrenamiento | 3400 (80 %) |
| Total de ejemplos para prueba | 850 (20 %) |
| Cantidad de particiones (<i>k-folding</i>) | 3 |
| Cantidad de ejemplos para entrenar por partición | 2266 |
| Cantidad de ejemplos para validar por partición | 1134 |
| Longitudes de los ejemplos | 7500, 10000, 20000 |
| Vectores de <i>embedding</i> | GloVe, word2vec |
| Longitudes de los vectores de <i>embedding</i> | 52, 102 |
| Tamaños de <i>kernels</i> | 1, 3, 5, 7, 9 |
| Cantidad de <i>kernels</i> por cada tamaño | 100 |
| Tipo de convolución | convolución 1D |
| Neuronas en la capa de salida (clases) | 17 |
| Activación de la capa de salida | <i>softmax</i> |
| Cantidad de capas ocultas | 1 |
| Neuronas en capa oculta | 51 |
| Activación de la capa oculta | <i>relu</i> |
| Activación de las capas de <i>batch normalization</i> | <i>relu</i> |
| Porcentaje de <i>dropout</i> | 40 % |
| Optimizador | <i>adam</i> |
| Velocidad de aprendizaje | 0.001 |
| Función de error | <i>categorical_crossentropy</i> |
| Métrica | <i>categorical_accuracy</i> |
| Cantidad de épocas | 40 |
| Dimensión del <i>batch</i> | 32 |

Cuadro 3.2: Resumen CNN.

3.3.3. Red Neuronal *Transformer*

En la Figura 3.5 se muestra la arquitectura usada para el modelo *Transformer*. De manera similar a la anterior, consta de una capa de entrada que toma las secuencias de ejemplos. Las siguientes capas son específicas de este tipo de arquitectura, tal como se explicó en la Subsección 2.2.3: *TokenAndPositionEmbedding* y *TransformerBlock*. Lo obtenido luego pasa por una capa de *pooling* que toma el promedio global, finalizando en un MLP sin capas ocultas con una capa de salida, igual a la de la arquitectura anterior (con 17 neuronas).

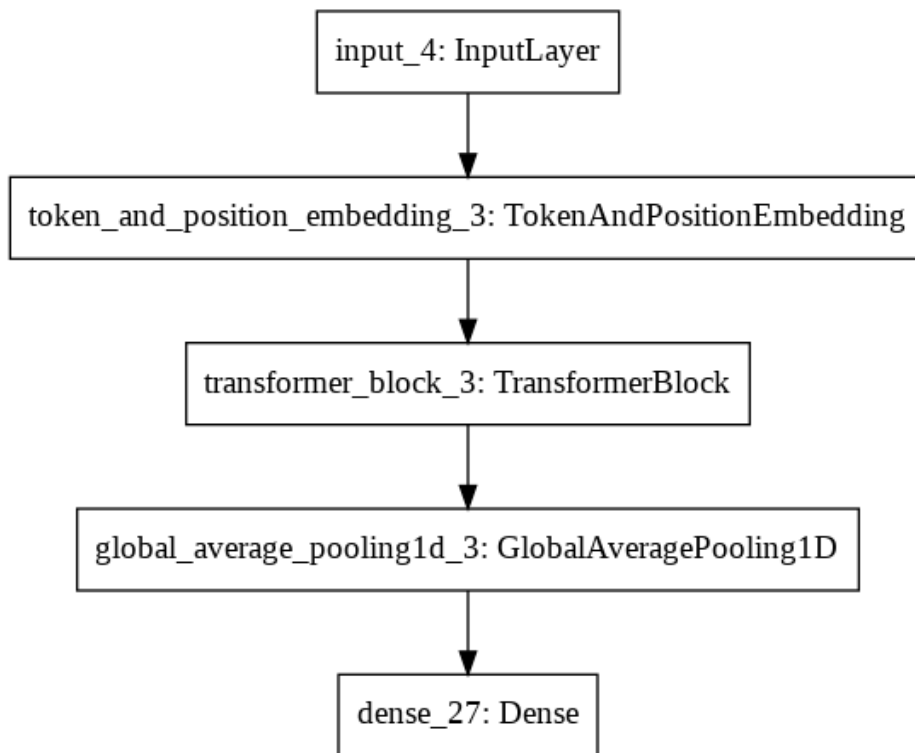


Figura 3.5: Arquitectura del modelo *Transformer*.

Teniendo en cuenta que este tipo de redes son exhaustivas en cuanto al uso de memoria, para respetar la longitud de los ejemplos y vectores de *embeddings* del caso anterior, la configuración de sus hiperparámetros se ve más limitada que con las redes CNN. Los hiperparámetros específicos definidos son el número de cabezas de atención, que se encuentra limitado a 1, y la dimensión de la red interna de propagación hacia adelante que posee el *Transformer* (*ff_dim*). Este valor se establece en el número recomendado del doble del tamaño de los vectores *embeddings*. Se debe tener en cuenta que

| | |
|--|---------------------------------|
| Total de ejemplos | 4250 |
| Total de ejemplos para entrenamiento | 3400 (80 %) |
| Total de ejemplos para prueba | 850 (20 %) |
| Cantidad de particiones (<i>k-folding</i>) | 3 |
| Cantidad de ejemplos para entrenar por partición | 2266 |
| Cantidad de ejemplos para validar por partición | 1134 |
| Longitudes de los ejemplos | 7500, 10000, 20000 |
| Vectores de <i>embedding</i> | <i>uniform</i> |
| Longitudes de los vectores de <i>embedding</i> | 52, 102 |
| Número de cabezas de atención | 1 |
| <i>ff_dim</i> | 104, 204 |
| Neuronas en la capa de salida (clases) | 17 |
| Activación de la capa de salida | <i>softmax</i> |
| Optimizador | <i>adam</i> |
| Velocidad de aprendizaje | 0.001 |
| Función de error | <i>categorical_crossentropy</i> |
| Métrica | <i>categorical_accuracy</i> |
| Cantidad de épocas | 40 |
| Dimensión del <i>batch</i> | 1 |

Cuadro 3.3: Resumen del modelo *Transformer*.

estas redes no utilizan *embeddings* preentrenados (en su definición original que es la utilizada en este proyecto), sino que los mismos son inicializados de manera aleatoria con distribución uniforme y son adaptados durante el entrenamiento.

El Cuadro 3.3 resume los datos usados para este modelo.

Capítulo 4

Evaluación de los modelos neuronales

En este capítulo se presentan los resultados obtenidos en los entrenamientos de los distintos modelos neuronales y su evaluación.



4.1. Hiperparámetros comunes

En los cuadros 4.1 y 4.2 se detallan los hiperparámetros comunes a todos los entrenamientos realizados para los modelos CNN y *Transformers* respectivamente. La selección de estos fue influenciada por un gran número de pruebas, en las cuales se intentó obtener los mejores porcentajes de acierto en validación, llevando al límite el hardware disponible (referirse a la Subsección 3.3.1). Una vez fijados estos parámetros, se procedió a realizar entrenamientos variando la longitud de los ejemplos, la longitud y tipo de vectores de *embeddings*, y adaptando o no estos últimos durante el entrenamiento.

| | |
|--|---------------------------------|
| Cantidad de particiones para validación cruzada (<i>folds</i>) | 3 |
| Cantidad de épocas | 40 |
| Procentaje de <i>dropout</i> | 40 % |
| Dimensiones de <i>kernels</i> | 1, 3, 5, 7, 9 |
| Cantidad de <i>kernels</i> por dimensión | 100 |
| Cantidad de capas ocultas | 1 |
| Cantidad de neuronas en la capa oculta | 51 |
| Función de activación en la capa oculta | <i>relu</i> |
| Cantidad de neuronas en la capa de salida | 17 |
| Función de activación en la capa de salida | <i>softmax</i> |
| Optimizador | <i>adam</i> |
| Función de error | <i>categorical_crossentropy</i> |
| Métrica | <i>categorical_accuracy</i> |
| Dimensión del <i>batch</i> | 32 |

Cuadro 4.1: Conjunto de hiperparámetros comunes a los entrenamientos de los modelos CNN.

| | |
|--|---------------------------------|
| Cantidad de particiones para validación cruzada (<i>folds</i>) | 3 |
| Cantidad de épocas | 40 |
| Procentaje de <i>dropout</i> | 40 % |
| Número de cabezales de atención | 1 |
| <i>ff_dim</i> | 104 y 204 |
| Cantidad de neuronas en la capa de salida | 17 |
| Función de activación en la capa de salida | <i>softmax</i> |
| Optimizador | <i>adam</i> |
| Función de error | <i>categorical_crossentropy</i> |
| Métrica | <i>categorical_accuracy</i> |
| Dimensión del <i>batch</i> | 1 |

Cuadro 4.2: Conjunto de hiperparámetros comunes a los entrenamientos de los modelos *Transformers*.

4.2. Acierto de los modelos

Los resultados mostrados en la Figura 4.1 corresponden a los porcentajes de clasificaciones correctas de los distintos modelos. Se utilizaron los conjuntos de hiperparámetros de los cuadros 4.1 y 4.2, variando la longitud de los ejemplos (7500, 10 000 y 20 000 palabras), la dimensión de los vectores de *embeddings* (52 y 102), el tipo de *embedding* (word2vec, GloVe y *random uniform*) y si los *embeddings* son o no adaptados durante el entrenamiento (*true* o *false*).

Si se comparan los promedios de aciertos en el conjunto de datos de prueba (promediando la última columna) se obtiene que las CNN son inferiores

| Arquitectura | Tipo de embedding | Embeddings adaptados durante el entrenamiento | Dimensión de los ejemplos | Dimensión de los embeddings | Acierto en el entrenamiento | | Acierto en la validación | | Acierto en la prueba |
|--------------|-------------------|---|---------------------------|-----------------------------|-----------------------------|--------|--------------------------|--------|----------------------|
| | | | | | Media | Desvío | Media | Desvío | |
| CNN | word2vec | FALSE | 7500 | 52 | 97.07% | 0.09% | 74.97% | 1.38% | 75.29% |
| | | | | 102 | 97.01% | 0.29% | 75.06% | 1.83% | 75.76% |
| | | | 10000 | 52 | 96.97% | 0.18% | 76.44% | 2.15% | 74.71% |
| | | | | 102 | 96.90% | 0.31% | 76.24% | 1.16% | 76.82% |
| | | | 20000 | 52 | 96.88% | 0.42% | 75.62% | 1.65% | 76.24% |
| | | | | 102 | 96.96% | 0.08% | 75.41% | 0.53% | 76.47% |
| | | TRUE | 7500 | 52 | 96.88% | 0.29% | 75.47% | 0.33% | 76.12% |
| | | | | 102 | 96.81% | 0.25% | 75.76% | 0.72% | 76.59% |
| | | | 10000 | 52 | 97.06% | 0.22% | 75.09% | 1.61% | 77.06% |
| | | | | 102 | 96.90% | 0.23% | 75.79% | 0.75% | 76.24% |
| | | | 20000 | 52 | 97.01% | 0.33% | 75.62% | 1.41% | 76.35% |
| | | | | 102 | 96.75% | 0.18% | 76.76% | 1.65% | 75.29% |
| | GloVe | FALSE | 7500 | 52 | 83.34% | 0.91% | 67.09% | 2.85% | 66.71% |
| | | | | 102 | 85.07% | 0.81% | 69.44% | 1.18% | 71.06% |
| | | | 10000 | 52 | 79.82% | 0.05% | 64.74% | 2.14% | 68.24% |
| | | | | 102 | 85.57% | 1.61% | 68.12% | 0.20% | 66.59% |
| | | | 20000 | 52 | 83.99% | 1.17% | 68.71% | 0.85% | 66.94% |
| | | | | 102 | 89.99% | 1.03% | 71.47% | 1.98% | 72.71% |
| | | TRUE | 7500 | 52 | 75.88% | 1.97% | 63.26% | 2.79% | 63.76% |
| | | | | 102 | 84.84% | 2.31% | 69.41% | 0.50% | 69.06% |
| | | | 10000 | 52 | 76.94% | 1.60% | 64.12% | 2.53% | 63.76% |
| | | | | 102 | 85.13% | 4.20% | 68.15% | 1.54% | 67.76% |
| | | | 20000 | 52 | 84.87% | 1.46% | 67.94% | 0.67% | 68.59% |
| | | | | 102 | 92.16% | 0.60% | 71.82% | 2.34% | 74.35% |
| Transformer | random uniform | TRUE | 7500 | 52 | 96.60% | 0.49% | 76.56% | 1.06% | 76.00% |
| | | | | 102 | 96.56% | 0.27% | 76.41% | 1.47% | 78.24% |
| | | | 10000 | 52 | 96.09% | 0.27% | 76.18% | 2.15% | 76.71% |
| | | | | 102 | 96.26% | 0.60% | 75.97% | 0.56% | 75.88% |
| | | | 20000 | 52 | 96.40% | 0.34% | 74.59% | 0.47% | 73.41% |
| | | | | 102 | 96.12% | 0.43% | 73.32% | 3.18% | 76.47% |

Figura 4.1: Porcentajes de acierto de los modelos.

a los *Transformers* con un acierto del 72.19 %, contra uno del 76.12 % respectivamente. Sin embargo, al comparar los tipos de *embeddings* utilizados por las CNN se tiene que word2vec acierta en 76.08 % mientras que GloVe lo hace en 68.29 %, por esta razón se descarta este último tipo en las siguientes comparaciones. En lo que respecta a los modelos CNN con *embeddings* word2vec se observa que, al no adaptarlos durante el entrenamiento, se obtiene un 75.88 % mientras que si se adaptan se obtiene un porcentaje mayor de 76.28 %. En relación a las longitudes de los ejemplos utilizados para el entrenamiento considerando modelos CNN con *embeddings* word2vec actualizados durante el entrenamiento, se tienen unos porcentajes de acierto promedio de 76.36 %, 76.65 % y 75.82 % para longitudes de 7500, 10 000 y 20 000 palabras; mientras que para los *Transformers* estos son 77.12 %, 76.30 % y 74.94 % respectivamente. De aquí, se puede destacar que adaptar los vectores de *embeddings* durante el entrenamiento permite la utilización de ejemplos de longitud menor manteniendo buenos resultados. Por otro lado, si se comparan las longitudes de *embeddings* utilizadas, 52 y 102, se observa que existe una tendencia que, a mayor dimensión de estos, se llega a mejores resultados (Figura 4.2).

4.3. Evaluación del desempeño

En esta sección se presentan los valores obtenidos para el área bajo la curva ROC (AUC por sus siglas en inglés, *area under curve*), evaluada sobre los tres tipos de *embeddings* comparados (word2vec, GloVe y *random uniform* para *Transformers*) utilizados por los modelos, para cada una de las 17 clases. Es decir, en base a las mediciones obtenidas de todos los modelos que usan el mismo tipo de *embedding* y para cada una de las clases. Estos son: 12 modelos que utilizan *embeddings* word2vec con arquitectura CNN, 12 modelos que utilizan *embeddings* GloVe con arquitectura CNN y 6 modelos que utilizan *embeddings random uniform* con arquitectura *Transformer* (véase la Figura 4.1).

Una curva ROC (*Receiver Operating Characteristics*, curva de característica operativa del receptor) es un gráfico que muestra el rendimiento de un modelo de clasificación en todos los umbrales de clasificación, permitiendo visualizar, organizar y seleccionar clasificadores en base a su desempeño (Fawcett, 2004). Esta curva representa dos parámetros: tasa de verdaderos positivos (TPR, *True Positive Rate*) y tasa de falsos positivos (FPR, *False Positive Rate*). Siendo TP, TN, FP y FN los verdaderos positivos, verdaderos negativos, falsos positivos y falsos negativos respectivamente (ver Figura 4.3), la tasa de verdaderos positivos (también conocida como *sensibilidad*) se de-

| | | | |
|------------------------------|--------|--------------|--------|
| CNN (word2vec, GloVe) | 72.19% | CNN word2vec | 76.08% |
| | | CNN GloVe | 68.29% |
| Transformer (Random uniform) | 76.12% | | |

| | | | |
|------------------------------|--------|---------------------------|--------|
| CNN word2vec FALSE | 75.88% | CNN word2vec FALSE 7500 | 75.53% |
| | | CNN word2vec FALSE 10000 | 75.77% |
| | | CNN word2vec FALSE 20000 | 76.36% |
| | | CNN word2vec FALSE 52 | 75.41% |
| | | CNN word2vec FALSE 102 | 76.35% |
| CNN word2vec TRUE | 76.28% | CNN word2vec TRUE 7500 | 76.36% |
| | | CNN word2vec TRUE 10000 | 76.65% |
| | | CNN word2vec TRUE 20000 | 75.82% |
| | | CNN word2vec TRUE 52 | 76.51% |
| | | CNN word2vec TRUE 102 | 76.04% |
| CNN GloVe FALSE | 68.71% | CNN GloVe FALSE 7500 | 68.89% |
| | | CNN GloVe FALSE 10000 | 67.42% |
| | | CNN GloVe FALSE 20000 | 69.83% |
| | | CNN GloVe FALSE 52 | 67.30% |
| | | CNN GloVe FALSE 102 | 70.12% |
| CNN GloVe TRUE | 67.88% | CNN GloVe TRUE 7500 | 66.41% |
| | | CNN GloVe TRUE 10000 | 65.76% |
| | | CNN GloVe TRUE 20000 | 71.47% |
| | | CNN GloVe TRUE 52 | 65.37% |
| | | CNN GloVe TRUE 102 | 70.39% |
| Transformer (Random uniform) | 76.12% | Transformer RU TRUE 7500 | 77.12% |
| | | Transformer RU TRUE 10000 | 76.30% |
| | | Transformer RU TRUE 20000 | 74.94% |
| | | Transformer RU TRUE 52 | 75.37% |
| | | Transformer RU TRUE 102 | 76.86% |

Figura 4.2: Aciertos promedio de los modelos.

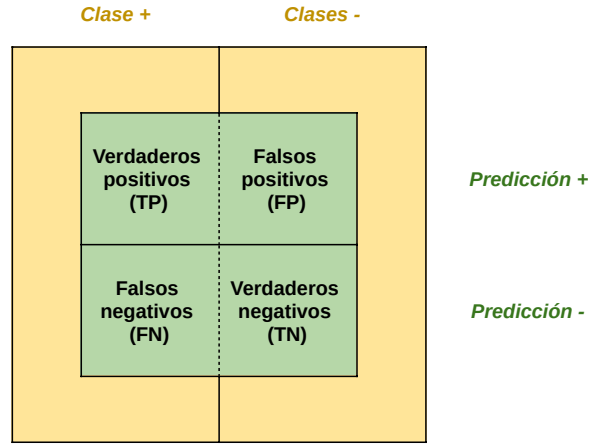


Figura 4.3: Matriz de confusión.

fine como:

$$TPR = \frac{TP}{TP + FN}$$

La tasa de falsos positivos se define de la siguiente manera:

$$FPR = \frac{FP}{FP + TN}$$

Una curva ROC representa TPR frente a FPR en diferentes umbrales de clasificación, pero es una representación bidimensional del desempeño de un clasificador. Para comparar clasificadores se puede reducir esta información a un único valor escalar, siendo un método común calcular el área bajo esta curva (AUC). Esto significa que el AUC mide toda el área bidimensional por debajo de la curva ROC completa de (0,0) a (1,1), y proporciona una medición agregada del rendimiento en todos los umbrales de clasificación posibles. Una forma de interpretar el AUC es como la probabilidad de que el modelo clasifique un ejemplo positivo aleatorio más alto (es decir que devuelva una mayor probabilidad) que un ejemplo negativo aleatorio. El AUC oscila en valor entre 0 y 1, teniendo un modelo cuyas predicciones son un 100 % incorrectas un AUC de 0.0, y otro cuyas predicciones son un 100 % correctas un AUC de 1.0 (mientras que lanzar una moneda al azar tiene un AUC de 0.5). Esta métrica se usa a menudo en la práctica cuando se requiere una medida general de la capacidad de predicción de un modelo (Fawcett, 2004).

Como dicha métrica se define para clasificaciones binarias, a la hora de evaluarla para las 17 clases de este proyecto, se considera a la clase de interés como la categoría positiva y al resto de las clases como la categoría negativa. Hay muchas formas de calcular esta área bajo la curva y existen algoritmos eficientes para hacerlo; en este proyecto se usó la función `auc` del módulo `metrics` de la biblioteca `sklearn`¹, que recibe como parámetros la TPR y la FPR.

Toda esta información obtenida se presenta en forma de diagrama de caja y bigotes, que puede observarse en la Figura 4.4. En la Figura 4.5 se muestra el esquema de colores utilizado para distinguir los diferentes tipos de *embeddings*.

Los valores AUC pueden interpretarse de la siguiente manera, según los siguientes intervalos (se detallan además las clases que caen en estos intervalos según el valor medio de cada caja de la Figura 4.4, junto con el tipo de *embedding* asociado):

- [0,5]: Es como lanzar una moneda. *activator* (word2vec y GloVe).
- [0,5,0,6): Prueba mala. *binder* (GloVe), *inducer* (GloVe), *product of* (GloVe), *modulator* (Transformer), *activator* (Transformer), *negative modulator* (Transformer) y *other* (word2vec y GloVe).
- [0,6,0,75): Prueba regular. *antagonist* (GloVe), *binder* (word2vec y Transformer), *inducer* (word2vec y Transformer), *antibody* (Transformer), *partial agonist* (todos los casos), *ligand* (GloVe), *product of* (word2vec y Transformer), *potentiator* (todos los casos), *modulator* (word2vec y GloVe), *negative modulator* (word2vec y GloVe) y *other* (Transformer).
- [0,75,0,9): Prueba buena. *inhibitor*, *agonist* (todos los casos), *antagonist* (word2vec y Transformer), *cofactor* (word2vec y Transformer), *ligand* (word2vec y Transformer) y *multitarget* (GloVe).
- [0,9,0,97): Prueba muy buena. *cofactor* (GloVe), *antibody* (word2vec) y *multitarget* (word2vec y Transformer).
- [0,97,1): Prueba excelente. *sin_interacción* (todos los casos) y *antibody* (GloVe).

¹<https://scikit-learn.org>

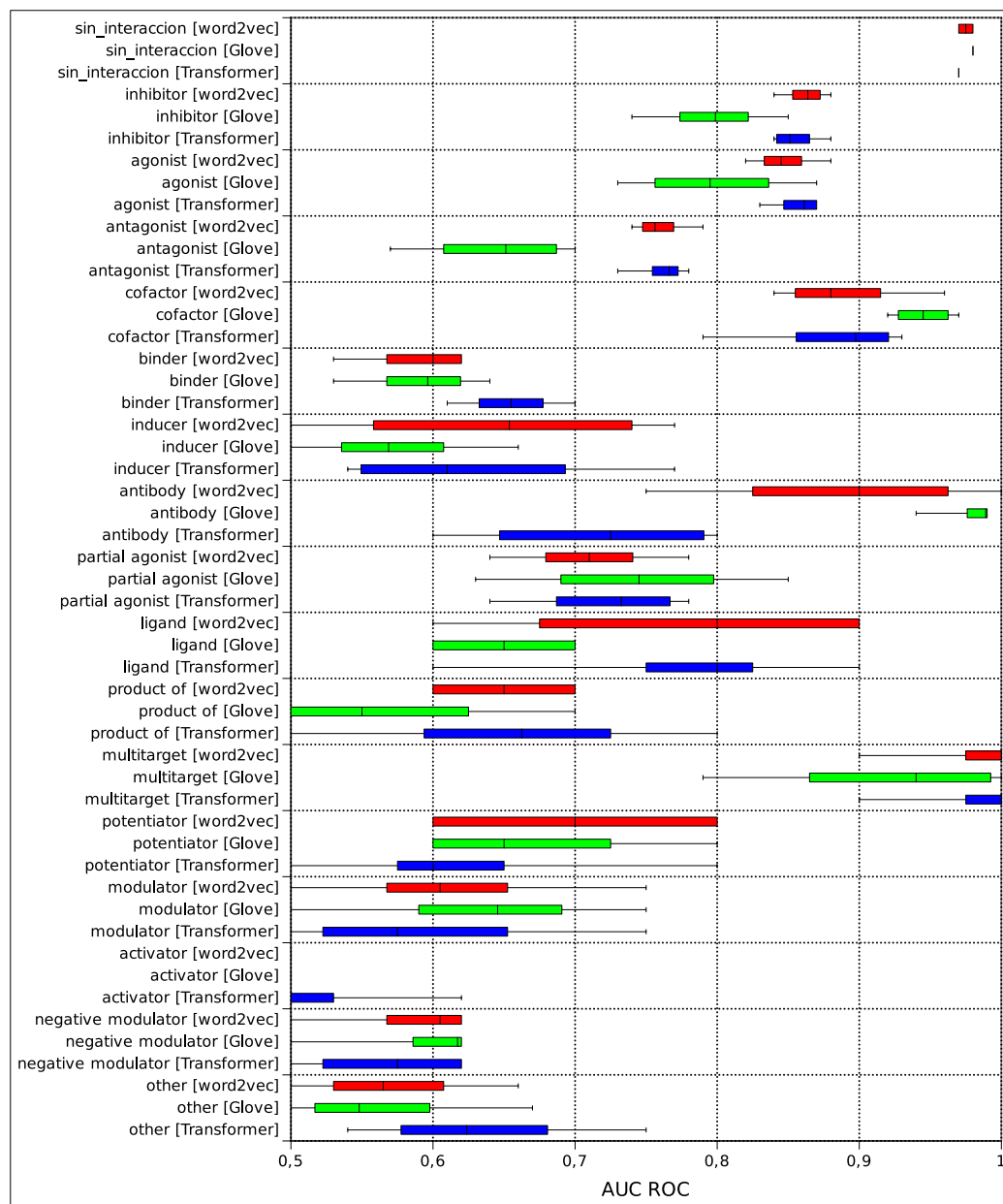


Figura 4.4: Diagrama de caja y bigotes para el área bajo la curva ROC (AUC).

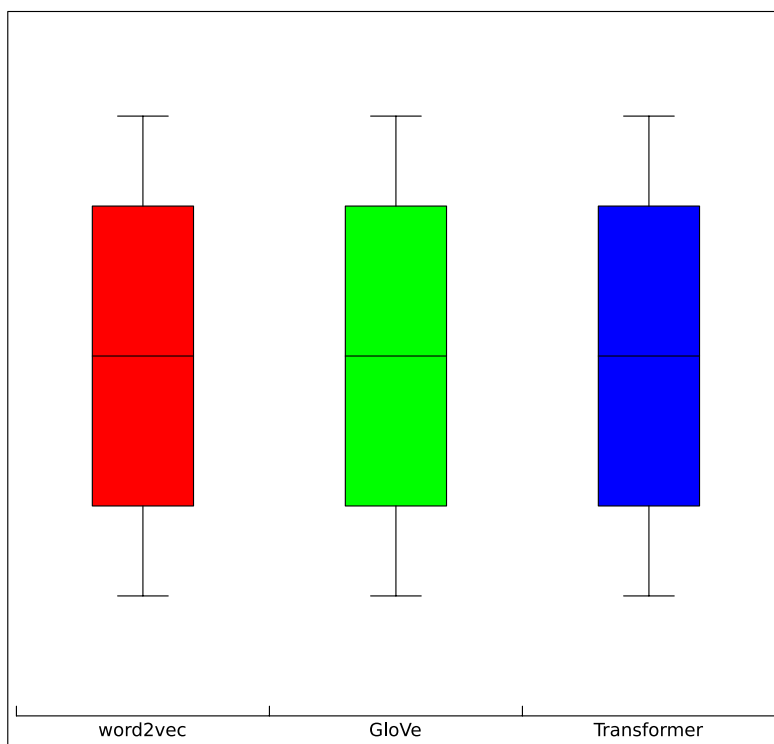


Figura 4.5: Leyenda de colores del diagrama de caja y bigotes. Rojo representa *embeddings* word2vec, verde representa *embeddings* GloVe y azul representa *embeddings random uniform* utilizados por los *Transformers*.

Se puede observar que la clasificación no es buena con ningún modelo para la clase *activator*. En cambio, las clases *binder*, *inducer*, *product of*, *modulator*, *negative modulator* y *other* tienen problemas de clasificación con algunos modelos mientras que, para otros, dicha clasificación supera el 60 % de AUC. Por otro lado, los modelos superan, siempre, el 60 % de AUC para las clases *inhibitor*, *agonist*, *antagonist*, *cofactor*, *antibody*, *partial agonist*, *ligand*, *multitarget*, *potentiator* y *sin_interacción*. El tipo de *embedding* GloVe logra ubicar sólo 12 clases por encima del 60 %, de las cuales sólo 6 se encuadran dentro de prueba buena o superior (*inhibitor*, *agonist*, *cofactor*, *antibody*, *multitarget* y *sin_interacción*); *Transformer* supera el 60 % con 14 clases con 7 dentro de prueba buena o superior (*inhibitor*, *agonist*, *antagonist*, *cofactor*, *ligand*, *multitarget* y *sin_interacción*); y word2vec excede el 60 % con 15 clases, de las cuales 8 caen en prueba buena o superior (las mismas 7 que *Transformer* más *antibody*).

Es fácil observar que la clase con mayor cantidad de ejemplos disponibles en el entrenamiento (*sin_interacción*, generados a partir de las etiquetas exis-

tentes como se describió en la Subsección 3.2.4) tiene un excelente desempeño, mientras que los peores resultados los arrojan las clases con menor cantidad de ejemplos (por ejemplo *negative modulator* y *activator*). La clase *other* es, incluso, más difícil de clasificar a pesar de tener un poco más de ejemplos (referirse a la Sección 3.3). Esto puede deberse a que es una mezcla de otras varias clases donde, quizás, los ejemplos correspondientes para cada una son muy distintos como para que se pueda aprender algo que los represente a todos por igual.

4.4. Selección del modelo final

Tras el análisis de los resultados, tanto de los valores AUC como de los porcentajes de acierto, se han descartado las redes CNN con *embeddings* de GloVe como opción de modelo final debido a su desempeño menor.

Las CNN con *embeddings* de word2vec y los *Transformers* han mostrado tener mejor desempeño, pero entre estos las primeras son, a nivel general, un poco mejores. Es de destacar que los *Transformers* han dado unos resultados muy buenos a pesar de haberse usado una arquitectura muy simple debido a las limitaciones de hardware (dado que entrenar este tipo de modelos requiere muchos recursos de memoria), que por el lado de las CNN es algo más elaborada. Debido a su desempeño un poco menor también se descartan los *Transformers*.

Dentro de las CNN con *embeddings* word2vec, el promedio de acierto es mayor cuando los *embeddings* se actualizan durante el entrenamiento. Siguiendo dentro de esta rama, no existen grandes diferencias entre las distintas longitudes de ejemplos, pero es de notar que el promedio de aciertos es un poco más alto para los *embeddings* con la menor longitud (52). Esto es importante puesto que los modelos con *embeddings* de longitud 102 tienen un tamaño de 623.3 MB mientras que los modelos con *embeddings* de longitud 52 tienen un tamaño de 318 MB (aproximadamente la mitad). Esta diferencia de tamaño hace que la carga y procesamiento del modelo por parte de la aplicación web requiera menor tiempo y recursos. Es por todo lo dicho, que el **modelo final seleccionado** es la CNN con *embeddings* word2vec de longitud 52 actualizados durante el entrenamiento y ejemplos de longitud 10 000.

Cabe aclarar que, el porcentaje de acierto (como se observa en la Figura 4.1), supera el 60 %, al igual que los valores para la métrica AUC (como se observa en la Figura 4.4) superan el 60 % para la mayoría de clases, cumpliendo con los requerimientos impuestos para este proyecto.

Capítulo 5

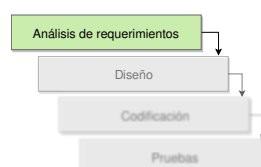
Trabajo realizado: Aplicación Web

En este capítulo, se presenta el trabajo realizado correspondiente al segundo gran módulo del sistema de búsqueda de interacciones fármaco-gen desarrollado: la aplicación web. Un esquema del flujo de uso de la misma integrada con el motor de inferencia neuronal se observa en la Figura 5.1.

5.1. Análisis de requerimientos

Los requerimientos para la aplicación web, tanto funcionales como no funcionales, fueron especificados oportunamente en la Sección 1.3.

Los mismos son planteados a partir de un conjunto de casos de uso esenciales definidos (ver Figura 5.2, detallados en los cuadros 5.1, 5.2 y 5.3) que los posibles usuarios e interesados identificados, conformados por médicos del campo de la medicina de precisión e investigadores genómicos, esperan realizar con la aplicación. Teniendo esto en cuenta, los requerimientos surgen de expandir y desarrollar estos casos de usos, agregando funcionalidad adicional que podría ser útil a los interesados, y validando los mismos con una serie de entrevistas informales realizadas a un pequeño grupo de posibles usuarios médicos que se logró reunir.



| | |
|----------------------|--|
| Actor | Usuario |
| <i>Identificador</i> | A1 |
| <i>Descripción</i> | Médico y/o investigador interesado en realizar búsquedas de interacciones genómicas. |

| | |
|------------------------|--|
| <i>Características</i> | Es el usuario que va a hacer uso de la aplicación, puede ser tanto un médico del campo de la medicina de precisión o un investigador genómico, que son los tipos de usuario a los que va dirigido el sistema. No se hace distinción entre los mismos ya que el resultado de las búsquedas (interacciones entre genes y/o fármacos de interés) les es útil a ambos. |
|------------------------|--|

Cuadro 5.1: Ficha textual descriptiva del actor *Usuario*.

| | |
|----------------------|--|
| Caso de uso | Realizar búsquedas de interacciones entre genes y/o fármacos |
| <i>Identificador</i> | CU1 |
| <i>Actores</i> | A1 |
| <i>Tipo</i> | esencial |
| <i>Precondición</i> | La aplicación se encuentra a la espera del ingreso de términos de búsqueda. |
| <i>Postcondición</i> | La aplicación queda en modo de visualización de resultados según los términos de búsqueda especificados. |

| Curso normal | Alternativas |
|---|---|
| 1) El usuario accede a la aplicación a través de la URL de la misma. | |
| 2) El usuario ingresa sus términos de búsqueda, tanto un gen y/o una droga. | |
| 3) La aplicación lista los resultados encontrados. | 3.1) La aplicación no encuentra ningún resultado, informando al usuario de esto. 3.2) El usuario tiene la posibilidad de realizar otra búsqueda. |

Cuadro 5.2: Ficha textual descriptiva del caso de uso *Realizar búsquedas de interacciones entre genes y/o fármacos*.

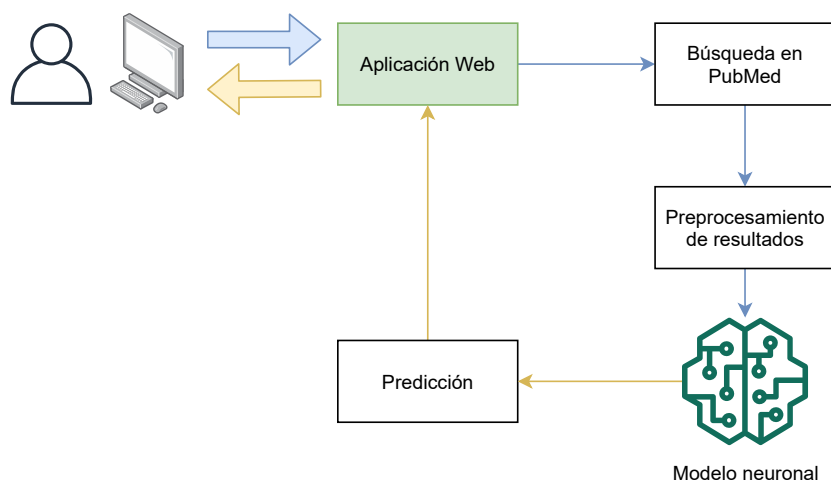


Figura 5.1: Flujo que sigue la aplicación a partir de una búsqueda de un usuario: la petición del mismo es recibida por la aplicación, esta busca y descarga artículos médicos externamente en *PubMed*, preprocesa estos artículos y con ello alimenta al modelo neuronal, el cual infiere el tipo de interacción entre los términos dados por el usuario. Esta predicción luego la aplicación la presenta en una página de resultados al usuario.

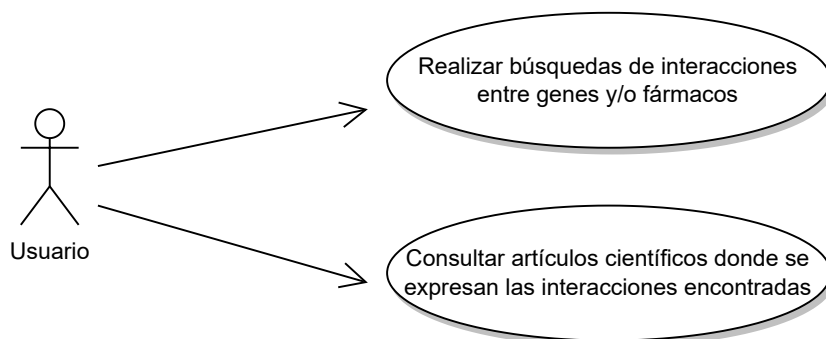


Figura 5.2: Diagrama de casos de uso esenciales.

5.2. Diseño

El diseño de la aplicación web se divide teniendo en cuenta los dos bloques de separación funcional utilizados, comúnmente, en el desarrollo web: *front-end* y *back-end*.



| | |
|----------------------|---|
| Caso de uso | Consultar artículos científicos donde se expresan las interacciones encontradas |
| <i>Identificador</i> | CU2 |
| <i>Actores</i> | A1 |
| <i>Tipo</i> | esencial |
| <i>Precondición</i> | La aplicación se encuentra en modo de visualización de resultados según los términos de búsqueda especificados. |

| Curso normal | Alternativas |
|---|---------------------|
| 1) El usuario selecciona un resultado de la lista de resultados. | |
| 2) La aplicación brinda al usuario la posibilidad de acceder a la fuente externa del mismo. | |
| 3) El usuario accede a la fuente. | |

Cuadro 5.3: Ficha textual descriptiva del caso de uso *Consultar artículos científicos donde se expresan las interacciones encontradas*.

Estos términos se refieren a la separación de intereses entre la capa de presentación y la capa de acceso a datos en una arquitectura multicapa. Esta es una arquitectura cliente-servidor en la cual las funciones de presentación, procesamiento y administración de datos se encuentran físicamente separadas. En el modelo cliente-servidor, el cliente es usualmente considerado el *front-end* y el servidor es, comúnmente, considerado el *back-end*, aun en los casos en que, un poco de trabajo de presentación, es realizado en el servidor.

Con respecto a la aplicación aquí desarrollada, el *back-end* es el encargado de procesar las consultas del usuario utilizando el modelo neuronal entrenado, además de proveer la funcionalidad extra, mediante submódulos. Para esto, se vale de una base de datos para la persistencia de la información. Cada submódulo cuenta con su parte de *front-end*. El bloque de *front-end* recolecta información del usuario y muestra los resultados. En la Figura 5.3 se detalla el diseño de la arquitectura que tiene la aplicación web en base a lo dicho. En la Figura 5.4 se brinda un diseño arquitectónico que da cuenta de los componentes del sistema, de sus interfaces y de sus interrelaciones a nivel físico.

En lo que respecta a la confección de las páginas servidas al usuario a partir de sus acciones, se decide utilizar un esquema mixto donde parte de la información es renderizada en el lado del servidor (carga completa de toda la vista), y donde cierta información adicional es solicitada al servidor

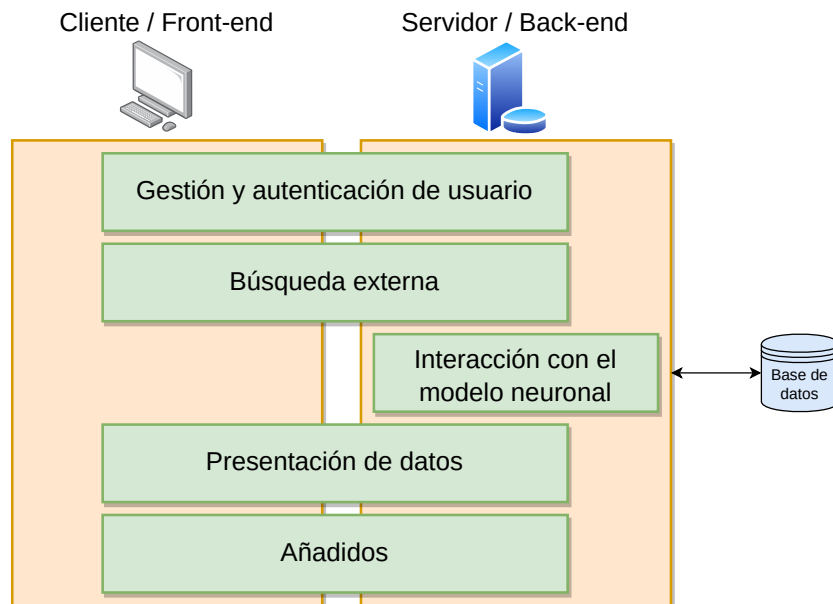


Figura 5.3: Arquitectura de la aplicación web con sus submódulos.

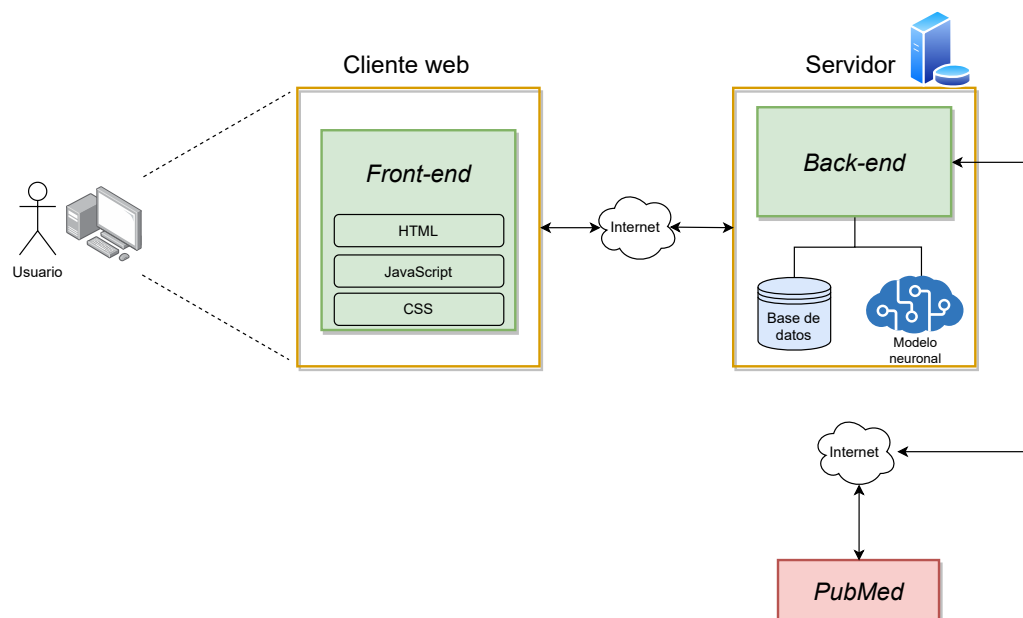


Figura 5.4: Arquitectura física del sistema.

asíncronamente desde el lado del cliente. Esta última, se carga en segundo plano sin interferir con la visualización ni el comportamiento de la página, mediante peticiones AJAX (*Asynchronous JavaScript and XML*). Esto implica la programación de scripts que se ejecutan en el cliente, es decir, en el navegador de los usuarios, mientras se mantiene la comunicación asíncrona con el servidor en segundo plano, lo que permite realizar cambios sobre las páginas sin necesidad de recargarlas, mejorando la interactividad, velocidad y usabilidad de la aplicación.

5.2.1. Diseño del *front-end*

Con diseño del *front-end* se refiere al diseño de las distintas vistas que posee la aplicación, es decir, a la interfaz de usuario de la misma. Para la creación de estas vistas o pantallas se tienen en cuenta las siguientes consideraciones:

- Se sigue un diseño limpio y sencillo, minimalista, al estilo de muchas aplicaciones web modernas. Esta idea se respeta para todas las vistas.
- El esquema de colores es principalmente blanco y negro (letras negras sobre fondo blanco), con agregados de otros colores donde se desea llamar la atención.
- La pantalla principal muestra el logo de la aplicación centrado, junto a los campos de búsqueda para que el usuario ingrese sus términos de interés. Es el punto de entrada de la funcionalidad principal.
- Cuando un usuario está registrado y con sesión iniciada, se muestra en todo momento un botón para acceder al menú de usuario. Este se despliega como panel lateral al presionar el botón, listando las opciones específicas para usuarios con sesión iniciada.
- En la parte superior, contenidos en una barra de navegación implícita, se ubican los distintos botones de acción disponibles, como por ejemplo Iniciar sesión, menú de usuario, guardar búsqueda, etc.; además de información sobre la pantalla actual.
- Las pantallas que corresponden a listados guardados (historial de búsqueda, búsquedas guardadas, publicaciones favoritas) presentan esta información como ítems cliqueables (enlaces adonde corresponda) en forma de única columna (una sola cadena de texto con toda la información necesaria, y no en forma de tabla o grilla).

- Los listados que permiten eliminación de elementos individuales poseen, al lado de cada ítem, un botón de eliminar. Además, se agrega debajo de todo el contenido un botón para eliminar todos los elementos de una única vez.
- Los resultados de las búsquedas son presentados en un listado que permite diferenciarlos claramente, donde se indica el título de cada artículo, su PMID (identificador en *PubMed*), y las interacciones que el modelo neuronal infirió, junto al porcentaje de confianza con que clasificó el tipo de interacción.
- Cuando se ingresa un solo tipo de entidad (un gen o una droga) y el sistema debe inferir interacciones con las demás entidades nombradas en el texto, en los resultados se agrupan para cada artículo las distintas inferencias realizadas.

5.2.2. Diseño del *back-end*

Como se exhibió anteriormente, se decide dividir toda la lógica referida a este bloque en varios submódulos, que en su conjunto brindan toda la funcionalidad de la aplicación. Se debe tener en cuenta que algunos de ellos tienen su parte de *front-end* asociada. Además, se sigue un diseño orientado a objetos permitiendo agilizar el desarrollo del software, gracias principalmente a sus características de abstracción, encapsulamiento y modularidad, facilitando su mantenimiento y extensión futura (Coad y Yourdon, 1991).

En la Figura 5.5 se presenta el diagrama de clases completo de toda la aplicación (se debe tener en cuenta que en el mismo se ignoran los métodos *getters* y *setters* de los atributos para mayor legibilidad). A continuación, se detalla la función que cumple cada submódulo junto a las clases correspondientes a cada uno, mediante un subdiagrama donde se sobresaltan las clases importantes para ese submódulo y se abrevian las demás (Object Management Group (OMG), 2003).

Gestión y autenticación de usuarios

Es el submódulo encargado de administrar las cuentas de usuario. Provee la funcionalidad para el registro de nuevas cuentas y la autenticación de cuentas existentes, mediante objetos de la clase **Usuario** (Figura 5.6).

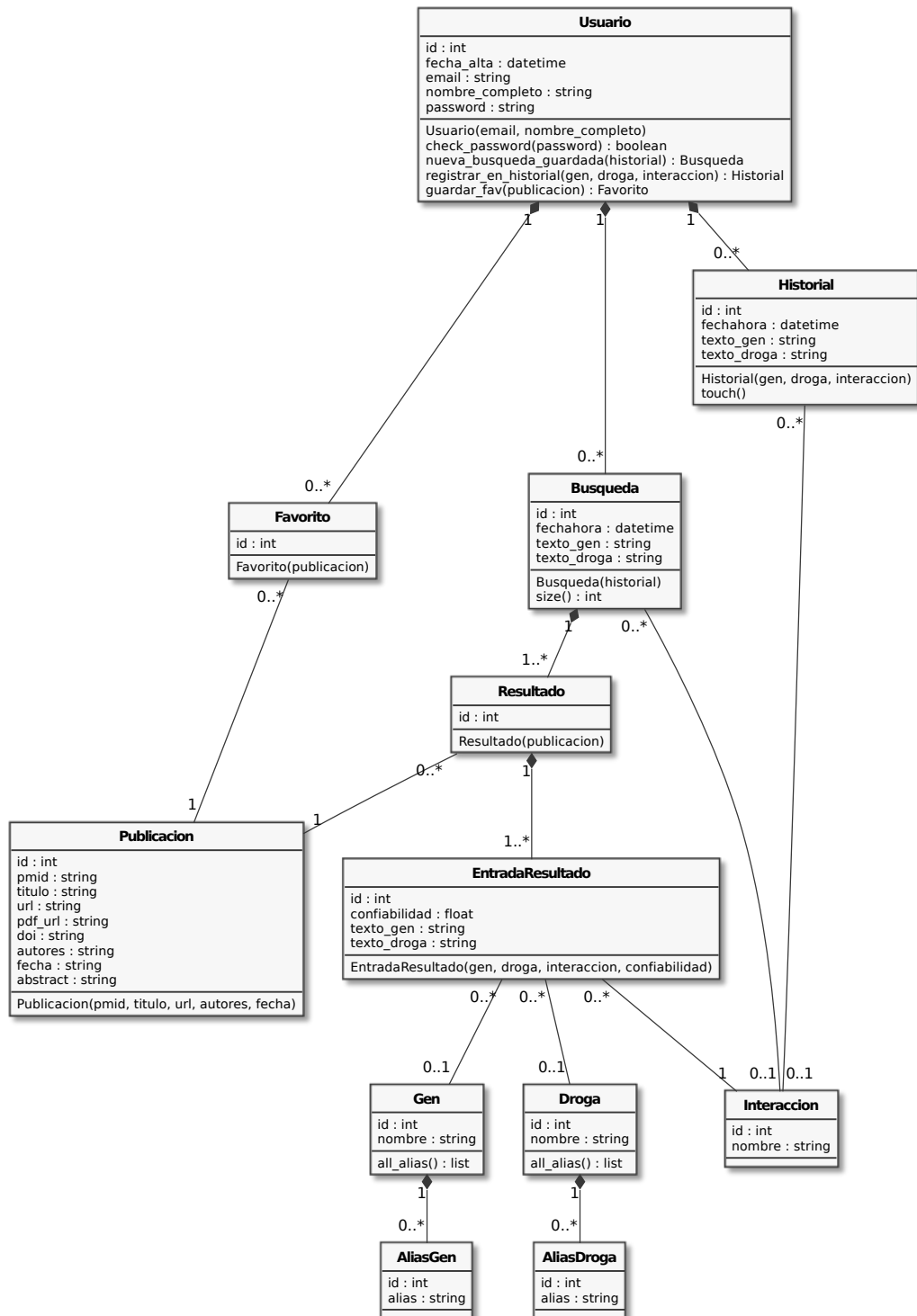


Figura 5.5: Diagrama de clases completo de la aplicación web.

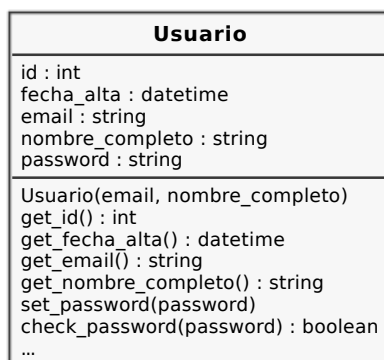


Figura 5.6: Clase **Usuario** como es usada por el submódulo de gestión y autenticación de usuarios.

Búsqueda externa

Este submódulo se encarga de consultar al recurso *online PubMed*, obteniendo distintos artículos científicos como resultado de la búsqueda deseada por el usuario. De la lista de resultados obtenida, descarga los artículos posibles mediante los distintos *scrapers* desarrollados (descarga limitada según la disponibilidad gratuita de los *papers* correspondientes). Todos los documentos que se van descargando son guardados en un caché para evitar descargarlos varias veces.

Para este submódulo se define la clase **Publicacion** (Figura 5.7) con la cual se guardan los distintos atributos de los artículos obtenidos.

Interacción con el modelo neuronal

Este submódulo es el encargado de preparar los datos y alimentar el modelo neuronal a partir de los documentos descargados por el submódulo anterior. Aquí se definen las clases **Gen**, **AliasGen**, **Droga**, **AliasDroga** e **Interaccion** (Figura 5.8) con el fin de tener una tipificación de las entidades biomédicas, conocidas por el modelo neuronal entrenado. De esta manera, si el usuario busca términos conocidos, automáticamente se tienen en cuenta los alias correspondientes para encontrar las ocurrencias de la entidad biomédica en los textos de los artículos.

Presentación de datos

Este submódulo se encarga de armar la presentación de los resultados del submódulo anterior, de manera estructurada, para poder renderizarse en el *front-end*. Para ello define la clase **Resultado**, que se asocia a una

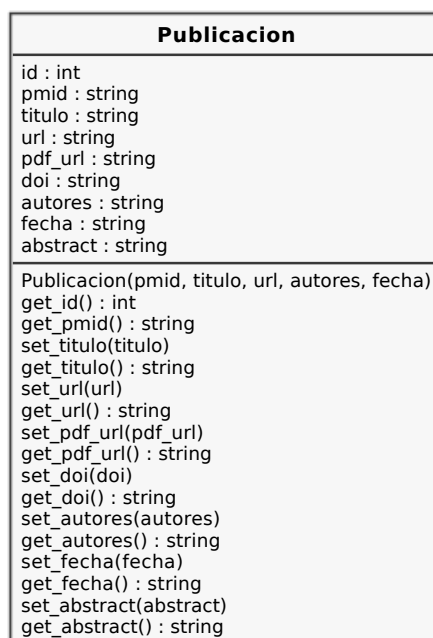


Figura 5.7: Clase Publicacion.

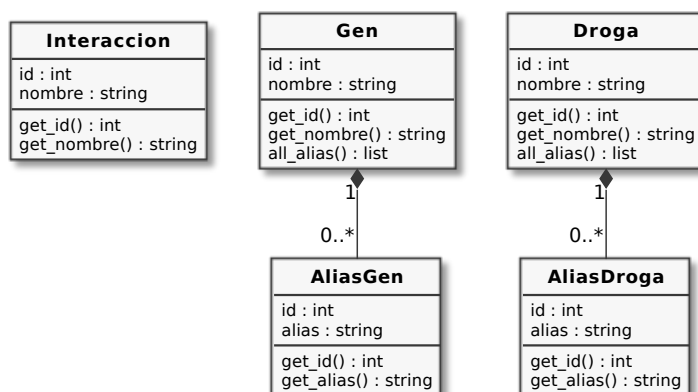


Figura 5.8: Clases de las entidades biomédicas.

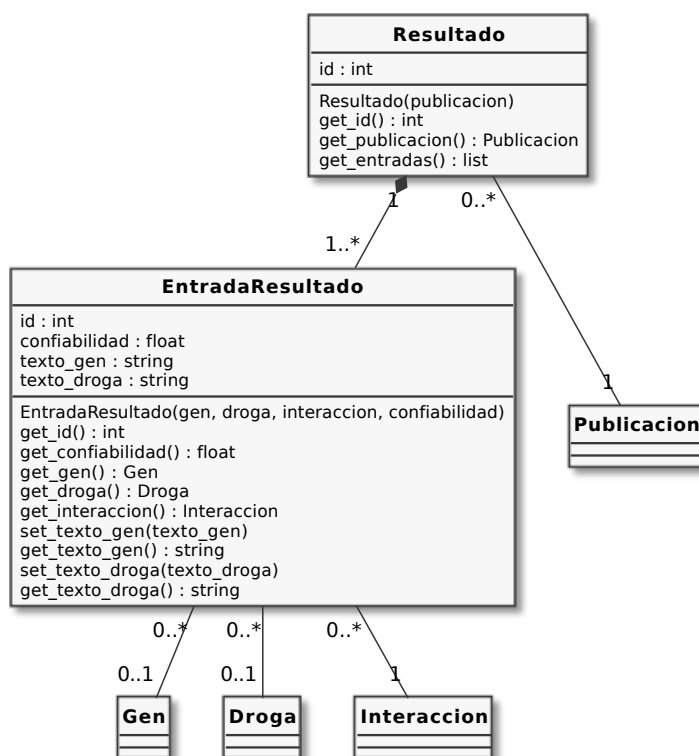


Figura 5.9: Clases para presentar los resultados.

Publicacion, y la cual tiene una colección de **EntradaResultado**, clase que lleva la información de qué gen interactúa con qué droga, mediante qué tipo de interacción y con cuánto porcentaje de confiabilidad se realizó dicha predicción (Figura 5.9). Como la búsqueda de términos es libre, es decir que el usuario puede ingresar nombres de genes y/o drogas que no se incluyen en la base de datos, la clase **EntradaResultado** posee atributos para guardar esta información (`texto_gen` y `texto_droga`) en el caso de no poder relacionarse con un **Gen** y/o **Droga**.

Añadidos

Este submódulo brinda toda la funcionalidad extra de la aplicación web para usuarios registrados: el historial de búsquedas, búsquedas específicas guardadas y artículos favoritos. Cada petición que realiza el usuario, desde el formulario principal de búsqueda, queda registrada en su historial de búsquedas (como una instancia de **Historial**), con toda la información necesaria para poder repetirla cuando desee. A su vez, una página de resultados obtenida también puede ser guardada para revisitarla en cualquier momento

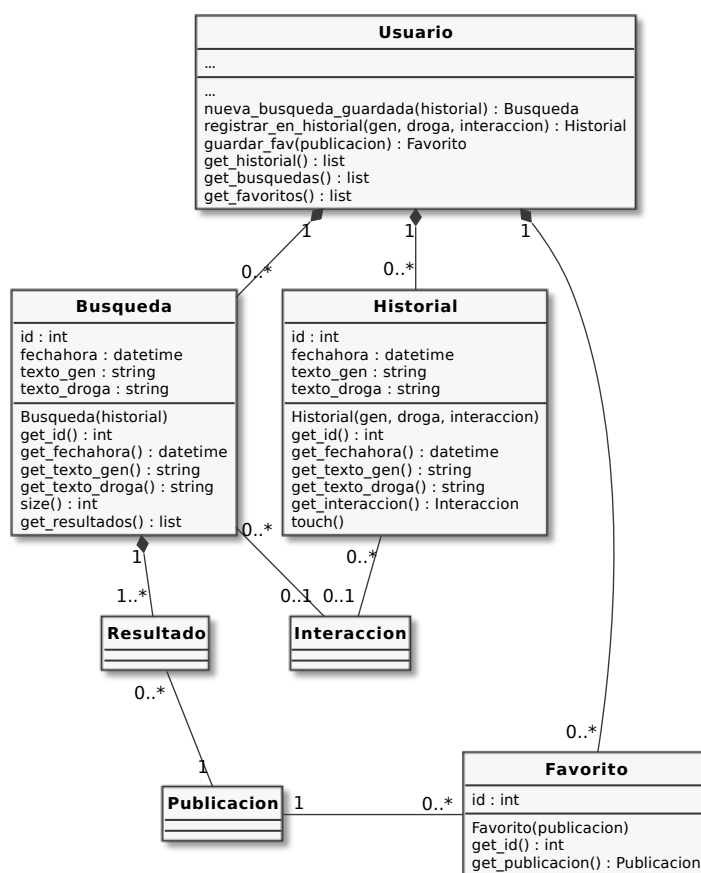


Figura 5.10: Clases que intervienen en el submódulo de los añadidos.

posterior, sin necesidad de repetir todo el proceso de búsqueda externa asociado; para esto se define la clase *Busqueda*. Por último, los artículos que el usuario encuentra de su interés puede marcarlos para tener un acceso rápido a ellos, es decir, a la página de *PubMed* de los mismos, definiéndose para esto la clase *Favorito*. En la Figura 5.10 puede observarse cómo se diagraman los añadidos.

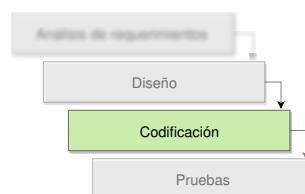
5.2.3. Base de datos

La base de datos que utiliza la aplicación web sirve como capa de persistencia para las clases definidas previamente, donde cada una posee una tabla asociada siguiendo un modelo relacional. De esta manera, es muy similar y comparte la mayoría de tablas con la base de datos armada anteriormente de etiquetas (ver Subsección 3.2.1): tablas de las entidades biomédicas

(`gen`, `droga`, `gen_alias`, `droga_alias`), de interacciones y de publicaciones. La única tabla que no posee es `interaccion_farmaco_gen` ya que esta guarda las etiquetas en sí y es exclusiva de la base de datos de etiquetas. Adicionalmente, posee varias tablas extras para poder proveer la funcionalidad específica de la aplicación, como son `usuario`, `historial`, `favorito` y las tablas para permitir guardar una búsqueda con sus resultados: `busqueda`, `resultado` y `entrada_resultado`. Un diagrama del esquema completo se observa en la Figura 5.11.

5.3. Codificación

Para la codificación de la aplicación web se hace uso del *framework* libre Flask. Es considerado un *microframework* ya que sólo incluye los componentes básicos necesarios para crear una aplicación y, la funcionalidad adicional requerida, se puede añadir mediante *plugins*.



Debido a esto, Flask es muy liviano. Provee simplicidad y flexibilidad, lo que lo hace ideal para aplicaciones tanto pequeñas como medianas (de la Guardia, 2016). Además, posee una excelente documentación¹, tiene una amplia comunidad *online*², es de código abierto y su desarrollo es activo (según su repositorio de código³).

El esquema de funcionamiento de la aplicación web es sencillo: el usuario hace una petición a una URL, la aplicación la procesa, y devuelve una página o *vista* presentando toda la información que solicitó el usuario. Flask permite, fácilmente, separar la lógica asociada al procesamiento de una petición y la presentación en sí de la información, simplificando la división entre *back-end* y *front-end*. Por un lado, permite definir *rutras* (*endpoints*), que son las distintas URL que la aplicación implementa, y mapearlas a una función de vista (*view function*) que contiene la lógica que se ejecuta cuando el usuario realiza una petición a dicha URL. Por otro lado, el formato gráfico y representación puramente visual pueden ser guardados de manera genérica e independiente del contenido en forma de *plantillas*, que son archivos HTML. Estos se completan, luego, con la información adecuada, generando la respuesta final que

¹<https://flask.palletsprojects.com>

²<https://stackoverflow.com/questions/tagged/flask>

³<https://github.com/pallets/flask>

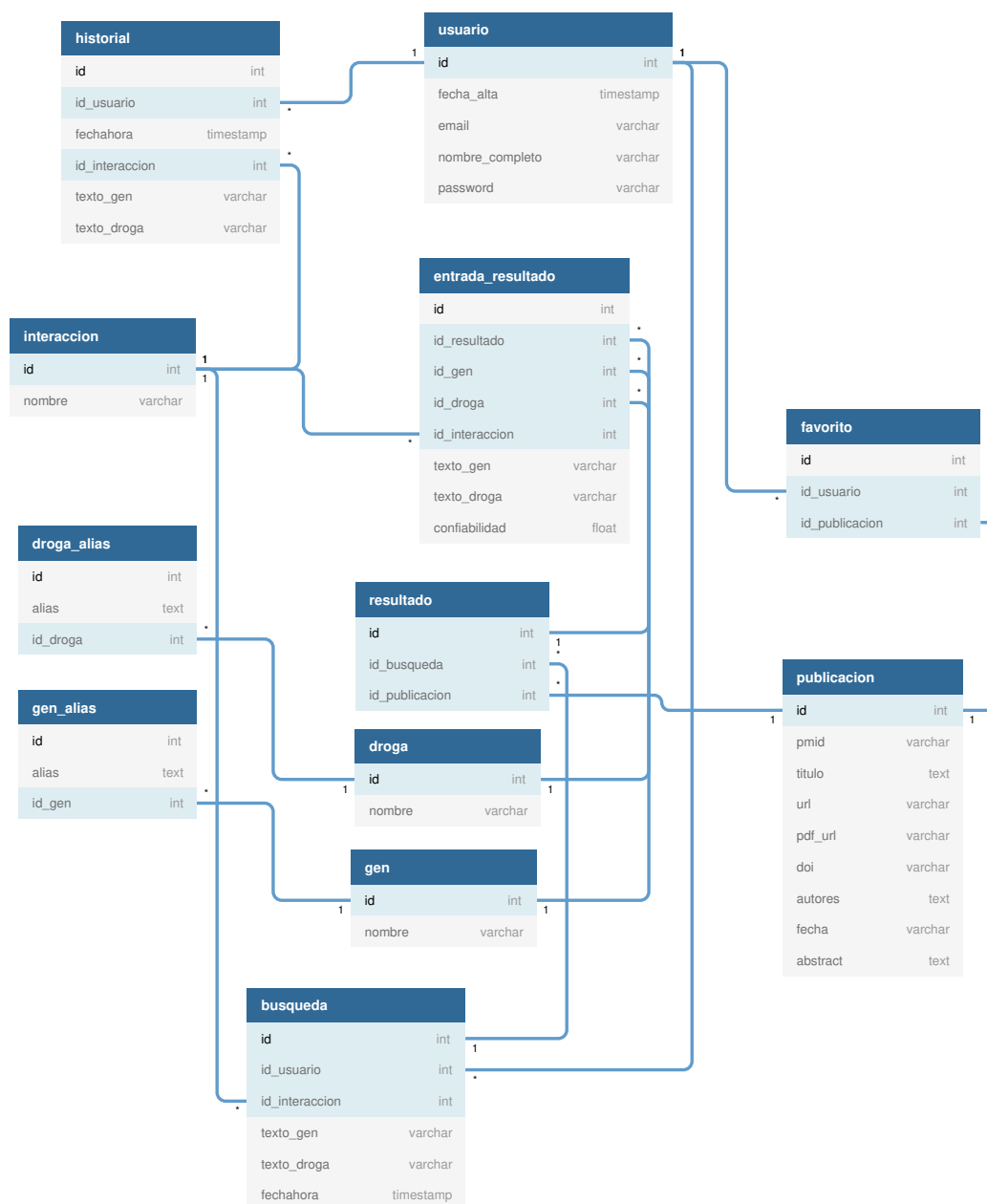


Figura 5.11: Diagrama de la base de datos de la aplicación web.

obtiene el usuario. Para esto, Flask hace uso del motor de plantillas Jinja⁴.

Flask posee muchas opciones de configuración, con valores por defecto adecuados y varias convenciones a seguir que permiten iniciar el desarrollo de forma rápida y que son suficientes para la mayoría de aplicaciones. Por ejemplo, las plantillas y archivos estáticos son ubicados en subdirectorios específicos ya predeterminados, y aunque permite cambiarlos, no es necesario. La aplicación web aquí desarrollada sigue estas convenciones y es estructurada en forma de *package* (paquete) de Python. En este lenguaje, un directorio que incluya un archivo llamado `__init__.py` es considerado un paquete y puede ser importado por programas externos. Todo el código incluido en el archivo `__init__.py` se ejecuta cuando el paquete es importado, inicializando el mismo y definiendo qué símbolos se exponen al mundo exterior.

La organización base del paquete de la aplicación web se estructura de la siguiente manera:

```
app/  
  static/  
  templates/  
  __init__.py  
  config.py  
  models.py  
  routes.py
```

Como se mencionó anteriormente, el subdirectorio `static` es para ubicar los archivos estáticos de la aplicación (imágenes, hojas de estilo, scripts del lado del cliente, etc.) y `templates` para las plantillas. En el archivo `config.py` se definen variables de configuración específicas de la aplicación, como son por ejemplo la URL del servidor de base de datos, la ruta al archivo con el modelo de la red neuronal a utilizar, o la *secret key* para encriptar las *cookies*⁵ de sesión (funcionalidad ya provista por Flask). El archivo `routes.py` define las rutas de los distintos *endpoints* que expone la aplicación, y el archivo `models.py` los modelos de las distintas clases de entidades mapeadas a la base de datos. Con respecto a esto, para hacer uso de la base de datos se utiliza la herramienta SQLAlchemy⁶ (a través de la extensión Flask-SQLAlchemy⁷)

⁴<https://palletsprojects.com/p/jinja>

⁵Pequeña pieza de información enviada por un sitio web y almacenada en el navegador del usuario, el cual la devuelve al servidor sin modificar, reflejando así un estado (memoria de eventos anteriores) en las transacciones HTTP, que de otra manera serían independientes de ese estado.

⁶<https://www.sqlalchemy.org>

⁷<https://flask-sqlalchemy.palletsprojects.com>

que es un ORM u *Object Relational Mapper* (mapeador objeto-relacional). El mismo permite que una aplicación administre una base de datos relacional a través de entidades de alto nivel como clases, objetos y métodos, en lugar de tablas y sentencias SQL, traduciendo de manera automática las operaciones de alto nivel a los comandos SQL correspondientes. Además, hace sencilla la configuración de la conexión al servidor de base de datos, en este caso un servidor de PostgreSQL.

Una aplicación en Flask, con conexión a una base de datos, a través de Flask-SQLAlchemy y uso de plantillas para devolver la información, requiere como mínimo las siguientes líneas, definidas en el archivo `__init__.py`:

```
1  from flask import Flask, render_template
2  from flask_sqlalchemy import SQLAlchemy
3
4  app = Flask(__name__)
5  db = SQLAlchemy(app)
6
7  @app.route("/")
8  def index():
9      return render_template("plantilla.html",
10                           data="Bienvenidos")
```

A partir de este archivo inicial se implementan los distintos submódulos (definiendo sus propias rutas, modelos y lógica correspondiente).

Con respecto al *front-end*, se codifican, por un lado, las plantillas utilizando la sintaxis de Jinja y lenguaje de marcado HTML 5 (que es la versión última y actual del estándar recomendado para presentar contenido en la *World Wide Web*), junto a las hojas de estilo CSS (*Cascading Style Sheets*); y por otro lado, los scripts del lado del cliente que permiten la interacción asíncrona y contenido dinámico, los cuales se escriben en el lenguaje JavaScript. Este, también, es un estándar para el desarrollo web y una de las tecnologías principales de la WWW, soportado por todos los navegadores actuales del mercado (Flanagan, 2020).

Para brindarle estilo a las distintas pantallas de la aplicación se hace uso del *framework CSS* Bulma⁸, ya que provee componentes visuales atractivos listos para usar de una manera fácil y simple. Es muy liviano y consiste en un único archivo `.css` de, aproximadamente, 200 KB, con la posibilidad de reducir este tamaño personalizando los componentes incluidos, gracias a ser modular; de esta manera se puede reducir al mínimo con sólo lo necesario. Además de su sencillez y modularidad, Bulma es altamente personalizable y responsivo, esto es, permite que todas las páginas se visualicen de manera correcta en una amplia variedad de dispositivos y tamaños de pantalla.

⁸<https://bulma.io>

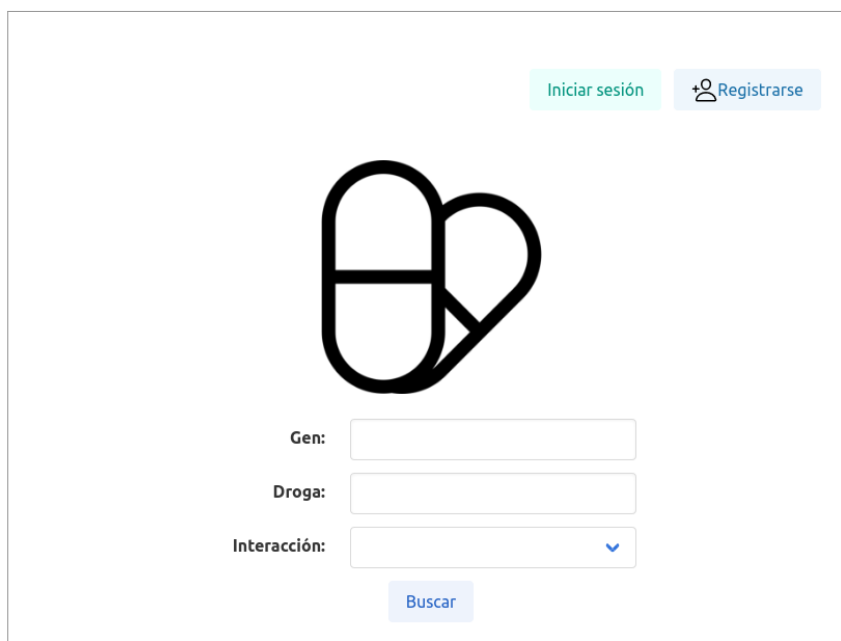
The image shows a web application interface. At the top right, there are two buttons: 'Iniciar sesión' (light green) and 'Registrarse' (light blue with a user icon). In the center is a large black logo consisting of a stylized heart shape with a horizontal bar and a vertical line. Below the logo are three input fields: 'Gen:' (text), 'Droga:' (text), and 'Interacción:' (dropdown menu with a blue arrow). Below these fields is a blue 'Buscar' button.

Figura 5.12: Pantalla principal de la aplicación web.

En la Figura 5.12 se presenta la pantalla principal de la aplicación, siendo la primera que ve el usuario al utilizar la misma. La implementación de los distintos submódulos y sus vistas asociadas, se detalla en las siguientes subsecciones.

5.3.1. Submódulo de gestión y autenticación de usuarios

Para este submódulo se crea, inicialmente, el modelo para la entidad **Usuario**. El registro de cuentas y su información se lleva como filas en la tabla asociada. De allí se recupera un usuario existente, a partir de su dirección de correo electrónico, en el momento de la autenticación y se valida la contraseña ingresada, rechazando la acción en caso de que no pase el chequeo.

En una autenticación válida, para mantener la sesión del usuario activa a través de las distintas peticiones, se utiliza la extensión Flask-Login⁹. Esta administra el estado de inicio de sesión a través de *cookies* de manera automática, de tal forma que un usuario que inicia sesión en la aplicación puede navegar a distintas páginas de la misma, mientras esta “recuerda” al usuario. Además, esta extensión provee la funcionalidad de *mantener sesión abierta*, es decir, permite que un usuario mantenga su sesión activa en la aplicación, incluso después de cerrar la ventana del navegador.

⁹<https://flask-login.readthedocs.io>

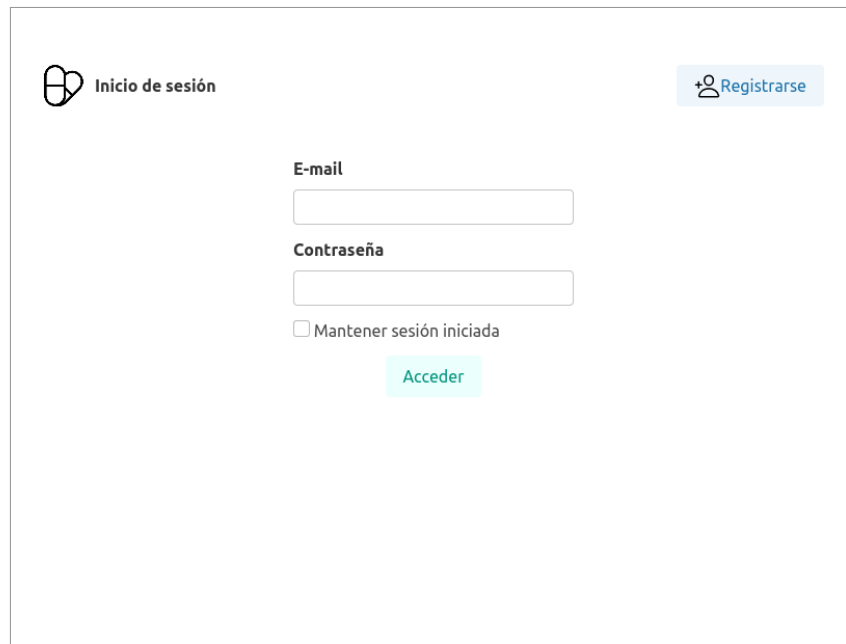


Figura 5.13: Pantalla de inicio de sesión.

Para este submódulo, se implementan las siguientes rutas:

- (GET/POST) `/login`

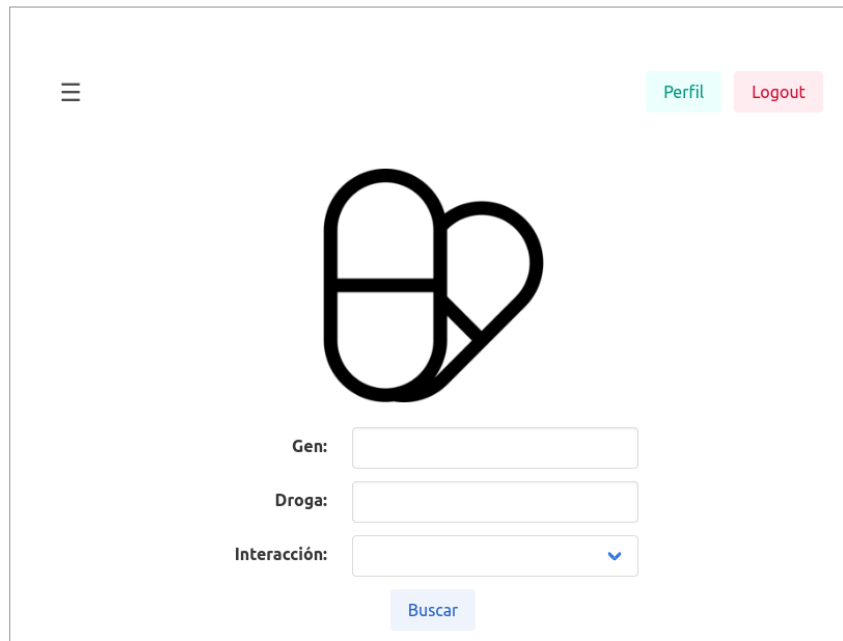
Esta ruta acepta dos métodos HTTP distintos: GET y POST. Si se solicita con GET devuelve el formulario para ingresar las credenciales (Figura 5.13). Llamado con POST recibe los datos de ese formulario y realiza la validación de los mismos. En el caso de credenciales correctas, se responde con una redirección a la ruta principal (`/index`) que devuelve la página principal renderizada con los datos de una sesión activa (Figura 5.14).

- (GET) `/logout`

Cierra la sesión del usuario.

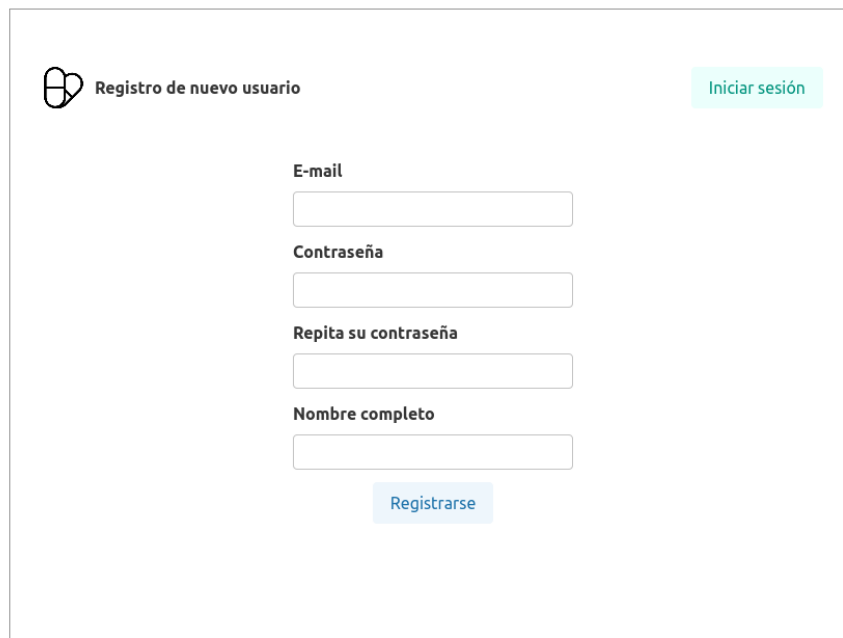
- (GET/POST) `/signup`

Esta ruta también acepta los métodos GET y POST. Llamada con GET devuelve el formulario de registro de nuevas cuentas (Figura 5.15), y con POST, procesa la información de este formulario para crear un usuario nuevo, validando que no exista ya otro con la misma dirección de e-mail, y persistiéndolo en la base de datos.



The screenshot shows a user profile page. In the top left corner, there is a hamburger menu icon. In the top right corner, there are two buttons: 'Perfil' (green) and 'Logout' (red). The main content area features a large, stylized black icon of a pill and a heart. Below this icon, there are three input fields: 'Gen:', 'Droga:', and 'Interacción:'. The 'Interacción:' field has a dropdown arrow on its right side. At the bottom center, there is a blue button labeled 'Buscar'.

Figura 5.14: Pantalla principal con sesión de usuario iniciada.



The screenshot shows a registration form titled 'Registro de nuevo usuario' with a pill and heart icon. In the top right corner, there is a green button labeled 'Iniciar sesión'. The form contains four input fields: 'E-mail', 'Contraseña', 'Repita su contraseña', and 'Nombre completo'. At the bottom center, there is a blue button labeled 'Registrarse'.

Figura 5.15: Formulario de registro de cuentas nuevas.

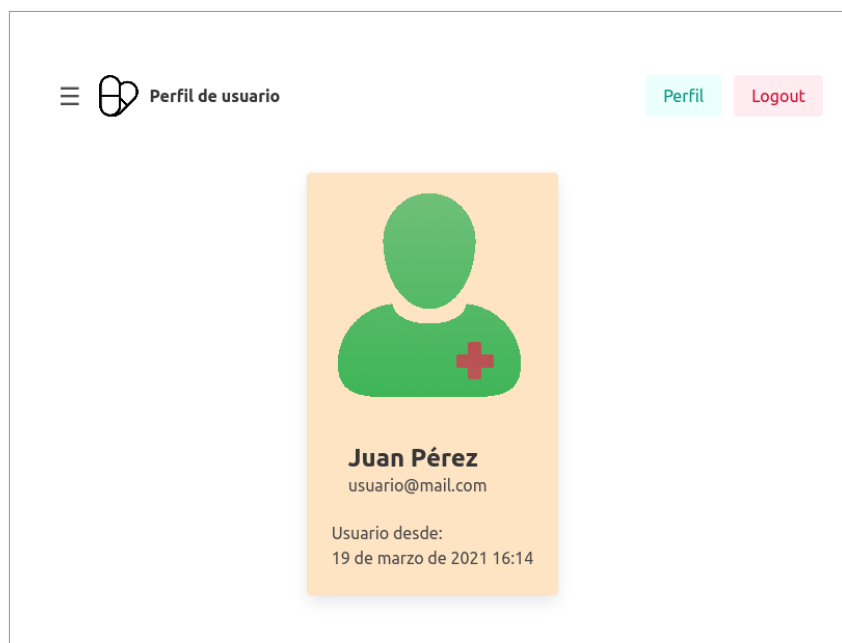


Figura 5.16: Información de perfil del usuario.

- (GET) /profile

Desde esta ruta, se obtiene una pantalla con información sobre la cuenta del usuario actualmente con sesión iniciada (nombre, e-mail, avatar, fecha de alta, etc.; ver Figura 5.16).

Para simplificar la comunicación desde el *front-end*, se hace uso de la extensión Flask-WTF¹⁰, la cual es un *wrapper*¹¹ a la librería WTForms¹². Esta extensión permite hacer uso de clases Python para representar formularios web, las cuales definen los campos del mismo como variables de clase. Cuando un formulario es completado y enviado al servidor, estas clases brindan, a través de sus variables, la información recibida. Además, esta extensión agrega protección contra ataques de tipo *Cross-Site Request Forgery* o CSRF (falsificación de petición en sitios cruzados), que es un tipo de ataque a un sitio web en el que comandos no autorizados son transmitidos por un usuario en el cual el sitio web confía (Siddiqui y Verma, 2011). Los formularios que define este submódulo son `RegistrationForm` para el registro, y `LoginForm` para el inicio de sesión.

¹⁰<https://flask-wtf.readthedocs.io>

¹¹Pequeña capa de código que traduce la interfaz existente de una librería en una interfaz compatible.

¹²<https://wtforms.readthedocs.io>

5.3.2. Submódulo de búsqueda externa

Este submódulo es el encargado de obtener los distintos artículos científicos que, luego, son analizados por el motor de inferencia neuronal. Para este submódulo se crea el modelo de la entidad **Publicacion**, y el módulo de Python llamado **pubmed_extract**, que reúne los métodos implementados.

En primer lugar, a partir de los términos que ingresa el usuario (gen, droga o ambos) se consulta en *PubMed* una búsqueda con esta información. Como *PubMed* no ofrece un servicio de búsqueda para ser consumido por aplicaciones en forma de API, se debe simular una búsqueda por parte de un usuario en el sitio web y luego parsear la/s página/s de resultados. Para ello, se crea la función **search** dentro del módulo **pubmed_extract**, que realiza una petición HTTP a la URL pública de *PubMed* de búsquedas. En ella, pasa como parámetro una cadena de la forma “gen droga” para que busque la aparición de ambos términos, y la opción de filtrar por artículos que dispongan de la descarga gratuita del documento del mismo. Como respuesta a esta petición, se obtiene una página de resultados que lista los artículos que según *PubMed* coinciden con la búsqueda. Se crea, entonces, una clase de utilidad **PubMedResultsParser** que, dado el contenido de esta página, extrae los PMID de los artículos y los deja disponibles en una lista (por ejemplo: ["29489145", "28710245", "29217012", "16716"]).

En una segunda etapa, dada esta lista de PMID, se crean instancias de la clase **Publicacion** para ir registrando toda la información asociada a cada artículo. De esta forma, si múltiples búsquedas devuelven un mismo PMID, la información necesaria del mismo se obtiene, directamente, desde la base de datos de la aplicación. Para esto, se extraen los datos disponibles de cada artículo desde su página específica de *PubMed*, mediante la función **get_pmid_info** implementada en el módulo **pubmed_extract**, que hace uso del *scraper* **PubMedParser** creado. Este extrae datos como título, autores, fecha, resumen, etc., y sobre todo la URL desde la cual se puede descargar el documento completo en PDF. El siguiente paso es descargar estos archivos con el texto completo de los artículos. Si la URL obtenida anteriormente se corresponde a un sitio para el cual se programó un *scraper* (ver Subsección 3.2.2), se utiliza el mismo para realizar la descarga del archivo. Este último, se almacena en un directorio caché para no volver a descargarlo nuevamente. Tanto este proceso de descarga como también el proceso anterior de obtener un objeto **Publicacion** a partir de un PMID, se implementaron para ejecutarse de forma paralela y acelerar los mismos mediante múltiples hilos, ya que el cuello de botella de la aplicación se encuentra en este submódulo al ser *I/O bound*. En otras palabras, el tiempo que le toma en completar sus tareas está determinado, principalmente, por la espera a que las operaciones

de entrada/salida terminen (específicamente las diversas conexiones de red intervinientes y la escritura en disco de los archivos descargados).

Este submódulo define el formulario `SearchForm` (renderizado en la pantalla principal como se observaba en la Figura 5.12) con el cual el usuario envía la información sobre qué gen y/o droga desea buscar, así como la interacción por la que desea filtrar los resultados, a través de una petición POST a la ruta `/search`.

5.3.3. Submódulo de interacción con el modelo neuronal

Este submódulo toma los documentos descargados por el submódulo anterior y los procesa para poder alimentar al modelo neuronal y obtener una inferencia del mismo. En una primera etapa de preprocesado, los archivos PDF descargados son convertidos a texto plano realizándoles, a la vez, una limpieza, de la misma forma en que se explicó en Subsección 3.2.3. Este contenido preprocesado también es guardado en un directorio caché como archivos de texto, para evitar la repetición de esta tarea si un artículo está presente en resultados de búsquedas distintas.

Luego, en este texto, de manera análoga a como se detalló en Subsección 3.2.5, se realiza la búsqueda de ocurrencias de los términos de interés del usuario (utilizando también los alias disponibles, reemplazando las apariciones por un identificador único de la forma `<GEN>` para genes o `<DROGA>` para drogas) y se transforma el resultado en secuencias numéricas que son las entradas de la red neuronal.

Según qué criterio de búsqueda ingresa el usuario, el comportamiento para buscar las ocurrencias en los textos y armar las secuencias de entrada es el siguiente:

- Ingresa sólo un gen: se buscan las ocurrencias de este gen junto con sus alias, y de todas las drogas conocidas por la aplicación (sólo sus nombres), para que la red infiera la interacción del gen con todas las drogas mencionadas en el texto.
- Ingresa sólo una droga: se buscan las ocurrencias de esta droga junto con sus alias, y de todos los genes conocidos por la aplicación (sólo sus nombres), para que la red infiera la interacción de la droga con todos los genes mencionados en el texto.
- Ingresa un gen y una droga: se buscan las ocurrencias de este gen y esta droga, junto con sus alias, para que la red infiera la interacción entre ellos dos.

Para que quede claro, se presenta un ejemplo ilustrativo. Suponiendo que el usuario ingresa para buscar el término **gen: gen1**, y en la base de datos de la aplicación existen las drogas **droga1**, **droga2** y **droga3**, para cada publicación a analizar devuelta por el submódulo anterior, se realiza lo siguiente: si **gen1** es un gen existente en la base de datos, se toma su nombre y el de sus alias y se buscan en el texto sus apariciones, reemplazándolos por el identificador **<GEN>**. De esta manera, se indica en el texto cuál es el gen de interés para que lo reconozca la red neuronal. Puede ser que el término que ingresa el usuario no sea un gen conocido por la aplicación y, en ese caso, se utiliza el término tal cual fue ingresado para buscarlo y reemplazarlo en el texto. Luego, se toman las drogas conocidas por la aplicación; en primer lugar, se busca en el texto la aparición de **droga1** y se reemplaza por el identificador **<DROGA>**, indicando de esta manera cuál es la droga de interés para que la reconozca la red neuronal. El resultado de estos reemplazos se convierte a secuencia numérica y constituye una entrada para la red. Se procede a realizar lo mismo para **droga2**, obteniendo otra entrada y, luego, para **droga3**, obteniendo una tercera entrada. Es así como se alimenta a la red para que infiera las interacciones entre 3 pares de entidades: **gen1** y **droga1**, **gen1** y **droga2** y **gen1** y **droga3**. En los demás casos, en que el usuario puede ingresar los criterios de búsqueda, el proceso es equivalente.

Si se tiene la representación como secuencia numérica de las entradas, se llama al método **predict** del modelo de la red neuronal con las mismas, lo que devuelve para cada una un vector de 17 elementos flotantes que indican con qué porcentaje, cada tipo de interacción, tiene probabilidades de ser la existente. De este vector se escoge el elemento máximo y, de esta manera, el índice de este elemento indica la interacción ganadora. Como se mencionó en los requerimientos, si el usuario ingresa un tipo de interacción para filtrar los resultados y esta no coincide con la interacción inferida, entonces esa entrada es descartada.

5.3.4. Submódulo de presentación de datos

Este submódulo estructura la información devuelta por la red neuronal para facilitar su presentación en pantalla y, a la vez, su posterior persistencia mediante el submódulo siguiente. Como se mencionó en el apartado Diseño (5.2.2), se define la clase **Resultado**, asociada a una **Publicacion**, y con una colección de **EntradaResultado** con la información de la/s predicción/es inferida/s entre genes y drogas. Dada esta estructura, la confección de la página devuelta al usuario se realiza de manera sencilla, como se observa en las figuras 5.17 y 5.18. A su vez, se define una estructura de datos **Resumen**, que se utiliza para contar cuántos resultados se obtuvieron para los distintos

tipos de interacción, con el fin de poder graficar, luego, esta información.

Para este submódulo se implementa la ruta `/get_results` mediante la cual el *front-end* solicita, a través de peticiones GET asíncronas, los datos obtenidos para una búsqueda: se devuelve, en primer lugar, un conjunto inicial de resultados, permitiendo con el botón de “Obtener más resultados” solicitar otra tanda y adjuntarla debajo del listado ya mostrado. De esta manera, no hay necesidad de recargar la página, hasta que no se encuentren más datos para devolver.

5.3.5. Submódulo de añadidos

Este submódulo brinda la funcionalidad extra de la aplicación web para las cuentas de usuarios registrados. El acceso a esta funcionalidad se habilita solo cuando hay un usuario con sesión iniciada, mediante el menú desplegado al presionar el “botón hamburguesa” (ícono formado por tres barras apiladas) disponible al iniciar sesión (como se observaba en la Figura 5.14 en la esquina superior izquierda de la pantalla). Dicho menú se expone en la Figura 5.19.

Todas las rutas implementadas en este módulo verifican que son llamadas por un usuario válido, rechazando las acciones cuando son invocadas sin ninguna sesión activa. Para esto, se utiliza el *decorator*¹³ `@login_required` de Flask-Login en las funciones de vista que, automáticamente, devuelve una redirección a la página de *login* cuando no pasa la verificación. Además, Flask-Login deja disponible para la función el objeto `current_user` que representa al usuario actual. Un ejemplo de su uso es el siguiente:

```
1 @app.route("/hist/")
2 @login_required
3 def historial():
4     return render_template("hist.html",
5                             hist=list(current_user.historial))
```

Historial de búsquedas

Este añadido registra las distintas búsquedas que realiza el usuario y le permite listarlas y repetirlas. Para ello, se crea el modelo de la entidad **Historial**, con el cual se persiste en la tabla asociada la información de cada búsqueda: el usuario que la realizó, la fecha y hora en que lo hizo, y los términos que ingresó. La creación de cada entrada en el historial del usuario se integra en la llamada a la ruta `/search` del submódulo de búsqueda externa

¹³Función que permite “envolver” a otra función con el fin de extender su comportamiento sin modificarla permanentemente.

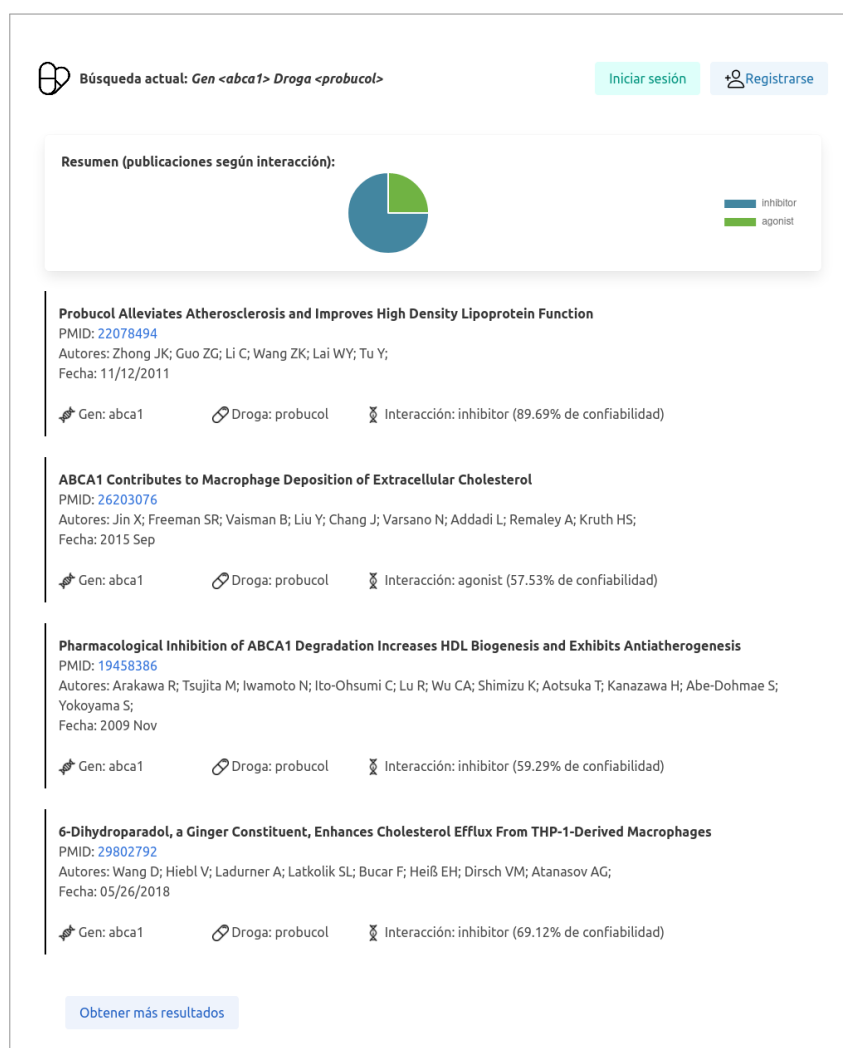


Figura 5.17: Presentación de los resultados de búsqueda.

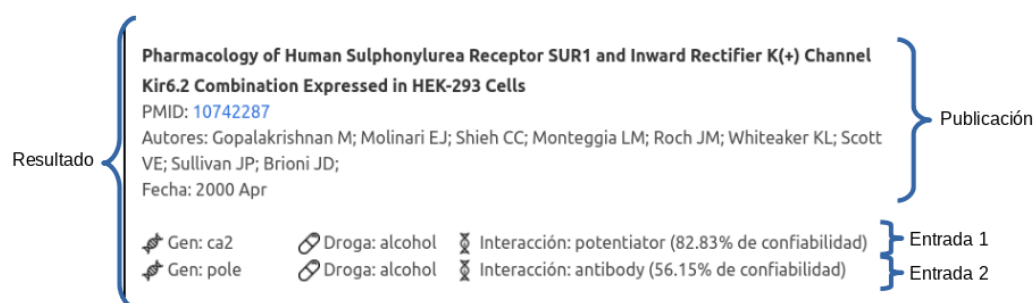


Figura 5.18: Estructura de un resultado.

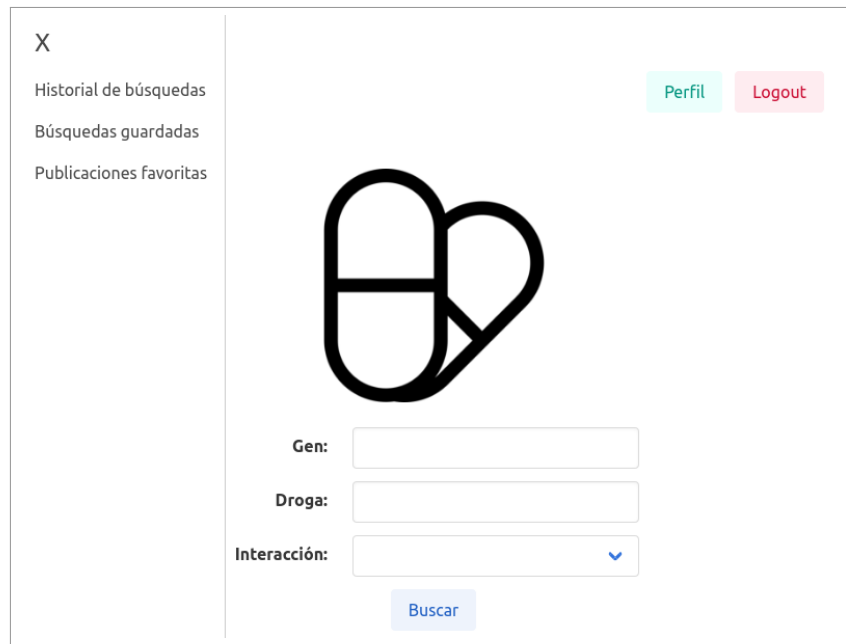


Figura 5.19: Menú con funcionalidad añadida de la aplicación para usuarios registrados.

y, además, se definen las siguientes rutas propias de este submódulo para administrar el historial desde el *front-end*:

- (GET) `/hist`

Esta ruta devuelve el historial completo guardado del usuario, dando la posibilidad de repetir alguna búsqueda si se cliquea sobre un elemento del listado (Figura 5.20).

- (GET) `/hist/<int:id>/`

Esta ruta repite la búsqueda asociada a la entrada con el ID dado del historial del usuario. Es la ruta que se llama al hacer clic en un elemento de la pantalla mencionada anteriormente.

- (GET) `/hist/clean`

Limpia el historial del usuario y devuelve una redirección a la misma página, la cual aparecerá, ahora, sin elementos. Es la ruta a la que apunta el botón de “Limpiar” que se observa en la Figura 5.20.

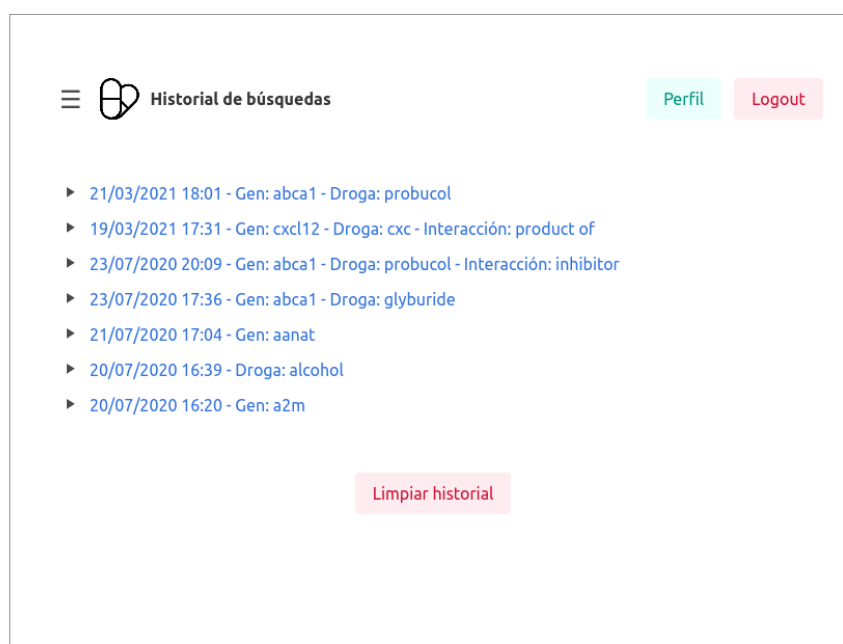


Figura 5.20: Pantalla con el historial de búsquedas del usuario.

Búsquedas específicas guardadas

Esta funcionalidad le permite al usuario “tomar una imagen” de una búsqueda, es decir, guardar la misma junto a sus resultados para poder verla tal como es en un momento posterior, sin tener que repetir todo el proceso de búsqueda externa e inferencia asociado. Para esto se define el modelo de la entidad **Busqueda**, con la cual se encapsula una colección de **Resultado** (definido en la Subsección 5.3.4) para persistirlos grupalmente. Esta funcionalidad se habilita mediante un botón específico que se muestra en una página de resultados cuando el usuario tiene sesión iniciada en la aplicación (ver Figura 5.21). Luego, pueden verse estas búsquedas guardadas desde una página específica (Figura 5.22), donde el usuario puede navegarlas o eliminarlas.

Las rutas propias para brindar esta funcionalidad añadida son las siguientes:

- (POST) `/save`

Esta ruta está implementada para ser llamada de forma asíncrona, devolviendo, simplemente, un código HTTP 200 OK, de tal forma que no se requiera un refresco de la pantalla completa al guardar una búsqueda. La información que guarda es la de la búsqueda actual, que se mantiene en la sesión HTTP.



Figura 5.21: Un usuario registrado puede guardar una página de resultados mediante el botón en forma de lupa tenue habilitado (figura superior). Cuando el botón posee un tilde verde significa que la búsqueda actual está guardada (figura inferior), y permite eliminarla al presionarlo.



Figura 5.22: Pantalla con las búsquedas guardadas del usuario.

- (POST) `/unsave`

Esta ruta también espera una llamada asíncrona. Elimina la búsqueda actual guardada. Esta ruta junto a la anterior son las que se invocan con los botones de la Figura 5.21.

- (GET) `/search/all`

Esta ruta devuelve la página donde el usuario puede ver y administrar sus búsquedas guardadas (Figura 5.22).

- (GET) `/search/<int:id>/`

Devuelve la página donde se mostrará la búsqueda guardada con el ID dado. Esta página renderizará los resultados de la misma utilizando la ruta siguiente.

- (GET) `/get_results/<int:id>`

Devuelve los resultados de la búsqueda guardada con el ID dado.

- (GET) `/unsave/<int:id>/`

Elimina la búsqueda guardada con el ID dado y devuelve una redirección a la misma página, la cual se renderizará ahora sin ese elemento. Es la ruta que se llama al presionar los botones en forma de cruz cuadrada al lado de cada elemento que se ven en la Figura 5.22.

- (GET) `/search/clean`

Elimina todas las búsquedas guardadas del usuario y devuelve una redirección a la misma página, la cual aparecerá ahora sin elementos. Es la ruta a la que apunta el botón de “Eliminar todas” que se observa en la Figura 5.22.

Artículos favoritos

La funcionalidad de artículos favoritos permite que el usuario marque y tenga a mano las publicaciones que considera importantes. Para esto se define el modelo de la entidad **Favorito**, que relaciona un usuario con una publicación. Al momento de devolver la página de resultados de una búsqueda, si hay un usuario con sesión iniciada, se habilita para cada publicación listada un botón que permite marcarla como favorita, así como también para desmarcarla, como se observa en la Figura 5.23. Todas las publicaciones marcadas como favoritas pueden ser accedidas luego en una pantalla específica con dicho listado (ver Figura 5.24). Allí, también se pueden eliminar las

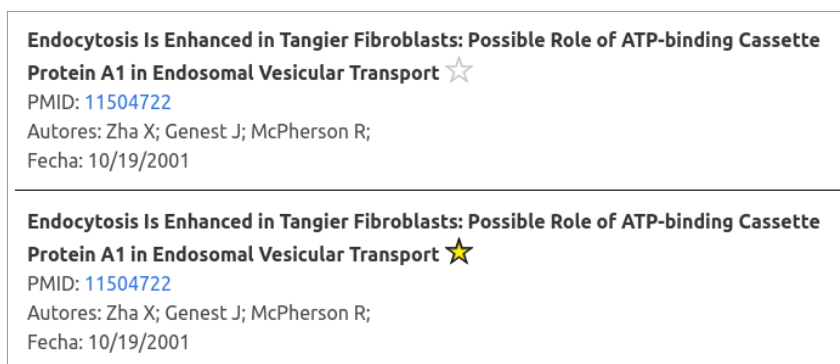


Figura 5.23: Un usuario registrado puede, a partir de una página de resultados, marcar una publicación como favorita presionando el botón en forma de estrella vacía (figura superior). La publicación favorita se mostrará entonces como una estrella llena (imagen inferior) que permite, a su vez, desmarcarla como favorita.

que ya no se desean presionando el botón en forma de cruz recuadrada que acompaña cada elemento.

Las rutas que se definen para completar esta funcionalidad añadida son las siguientes:

- (POST/DELETE) /fav

Esta ruta está implementada para ser llamada de forma asíncrona, devolviendo simplemente un código HTTP 200 OK, para poder integrarse con los botones mostrados en la Figura 5.23, de tal forma que no se requiera un refresco de la pantalla completa al marcar/desmarcar una publicación favorita. Recibe en un JSON¹⁴ (*JavaScript Object Notation*) el ID de la publicación cliqueada. Invocada mediante el método HTTP POST guarda el favorito, y mediante el método HTTP DELETE lo elimina.

- (GET) /favs

Esta ruta devuelve la página donde el usuario puede acceder y administrar sus favoritos (Figura 5.24).

- (GET) /unfav/<int:id_fav>/

Elimina el favorito dado por el ID `id_fav` devolviendo una redirección a la misma página, la cual se renderizará ahora sin ese elemento.

¹⁴Formato de texto sencillo para el intercambio de datos.

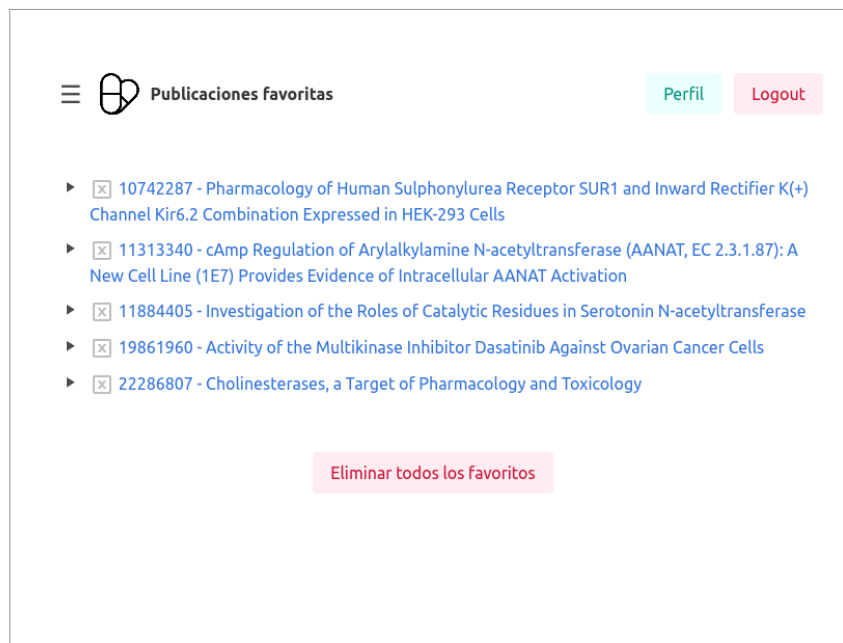


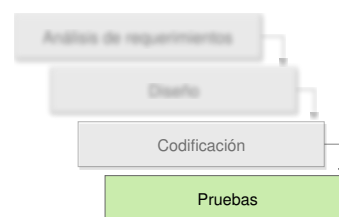
Figura 5.24: Pantalla con listado de publicaciones favoritas del usuario.

■ (GET) /favs/clean

Elimina todos los favoritos del usuario y devuelve una redirección a la misma página, la cual aparecerá ahora sin elementos. Es la ruta a la que apunta el botón de “Eliminar todos” que se observa en la Figura 5.24.

5.4. Pruebas y verificación

Para esta etapa, se realiza una prueba manual *end-to-end* con varios escenarios para garantizar el correcto funcionamiento de la aplicación como producto terminado. A su vez, verifica la integración de todos los componentes que interactúan (*front-end*, *back-end*, base de datos, modelo neuronal, interacción con sistema externo *PubMed*) y que el comportamiento de la misma sea el adecuado y cumpla con los requerimientos especificados. Las pruebas *end-to-end* (E2E) son una técnica que *testea* el producto de software entero, desde el inicio hasta el fin, para asegurar que



el flujo de la aplicación se comporte como se espera, así como también que todas las piezas integradas trabajen juntas como deben. El objetivo principal de las pruebas E2E es *testear* desde el punto de vista de la experiencia del usuario final mediante escenarios reales.

En el cuadro 5.4 se detalla el plan de estas pruebas con los distintos escenarios planteados y el resultado esperado y obtenido tras validar cada uno de ellos. A partir del N° 6 corresponden a escenarios con una sesión de usuario iniciada.

| N° | Escenario | Resultado esperado | Resultado obtenido |
|----|---|--|--|
| 1 | Realizar una búsqueda ingresando sólo un gen. | Obtener una página de resultados con artículos que nombran ese gen, y las distintas inferencias de las interacciones del mismo con las distintas drogas mencionadas en cada artículo. | Se obtiene correctamente la página esperada. |
| 2 | Realizar una búsqueda ingresando sólo una droga. | Obtener una página de resultados con artículos que nombran esa droga, y las distintas inferencias de las interacciones de la misma con los distintos genes mencionados en cada artículo. | Se obtiene correctamente la página esperada. |
| 3 | Realizar una búsqueda ingresando un gen y una droga. | Obtener una página de resultados con artículos que nombran esos términos y la inferencia de la interacción entre ellos en cada artículo. | Se obtiene correctamente la página esperada. |
| 4 | Realizar una búsqueda arbitraria especificando un tipo de interacción para filtrar. | Obtener una página de resultados con artículos donde se infirió el tipo de interacción dado. | Se obtiene correctamente la página esperada. |

| | | | |
|----|---|---|--|
| 5 | Registrar un nuevo usuario e iniciar sesión con el mismo. | Registro correcto del usuario, inicio de sesión exitoso con el mismo, y funcionalidad adicional habilitada. | Se inicia correctamente la sesión y las funciones adicionales son habilitadas apropiadamente. |
| 6 | Navegar hacia la página del perfil del usuario. | Página con la información del usuario ingresada al registrarse. | La información del usuario es mostrada correctamente. |
| 7 | Realizar una búsqueda arbitraria y verificar que la misma aparezca en el historial. | La búsqueda debe aparecer listada en el historial de búsquedas del usuario. Se debe poder repetir la búsqueda al hacerle clic. | La búsqueda realizada aparece listada correctamente. Se puede repetir exitosamente la misma desde el ítem del listado. |
| 8 | Eliminar una entrada del historial y limpiar el historial. | La entrada eliminada no debe aparecer más en el historial. Al limpiar el historial completo, el mismo debe quedar vacío. | Se elimina una entrada en particular y se eliminan todas juntas de manera correcta. |
| 9 | Realizar una búsqueda arbitraria y guardar la misma. | En el listado de búsquedas guardadas debe aparecer listada y debe cargarse correctamente con sus resultados si se cliquea en el ítem. | La búsqueda guardada aparece listada y carga correctamente sus resultados. |
| 10 | Desde la página de resultados de una búsqueda guardada, eliminar la misma. | En el listado de búsquedas guardadas no debe listarse más. | Al navegar al listado con todas las guardadas, no está más presente tal como corresponde. |
| 11 | Desde el listado de búsquedas guardadas eliminar una en particular y eliminar todas juntas. | La búsqueda guardada no debe aparecer más. Al eliminar todas juntas, el listado debe quedar vacío. | Se elimina una en particular y se eliminan todas juntas de manera correcta. |

| | | | |
|----|---|--|--|
| 12 | Realizar una búsqueda arbitraria y marcar como favorita una publicación de los resultados. | En el listado de favoritos debe aparecer lista-da correctamente, y su enlace dirigir a su página de <i>PubMed</i> asociada. | Se marca la publicación como favorita correctamente y al cliquearla se navega a su página de <i>PubMed</i> . |
| 13 | Desde una página de resultados con una publicación favorita, desmarcar la misma. | En el listado con todos los favoritos no debe listarse más. | Al navegar al listado con todos los favoritos no está más presente tal como corresponde. |
| 14 | Desde el listado de artículos favoritos eliminar uno en particular y eliminar todos juntos. | El favorito eliminado no debe aparecer más. Al eliminar todos juntos, el listado debe quedar vacío. | Se elimina uno en particular y se eliminan todos juntos de manera correcta. |
| 15 | Cerrar la sesión del usuario. | Mostrarse la página principal de búsqueda con los botones para registrarse e iniciar sesión, sin las funciones añadidas propias para un usuario. | Se cierra la sesión del usuario correctamente con el botón correspondiente. |

Cuadro 5.4: Pruebas E2E.

Con los resultados anteriores se puede verificar que se cumplen con los requerimientos funcionales especificados para la aplicación web (Sección 1.3):

- Los escenarios 1 al 4 verifican los requerimientos de que “debe permitir realizar búsquedas de interacciones fármaco-gen sobre el recurso online *PubMed*, infiriendo las interacciones con la utilización de la red entrenada” y “los resultados deben ser presentados junto a la información de los artículos de donde se obtuvieron los datos correspondientes al tipo de interacción”.
- Los escenarios 5 a 15 verifican el requerimiento de que “debe contar con un registro opcional de cuentas de usuario para brindarle a este ciertas funciones añadidas como historial de búsquedas, búsquedas específicas guardadas y artículos favoritos”.
- El requerimiento “debe permitir al usuario realizar búsquedas ingresando el nombre de un gen y/o un fármaco” es verificado por los escenarios

1, 2 y 3.

- El requerimiento “el usuario debe poder especificar un tipo de interacción, lo cual filtra los resultados sólo a ese tipo” es verificado por el escenario 4.

En cuanto a los requerimientos no funcionales, también se satisfacen los mismos:

- Teniendo en cuenta el diseño de todas las vistas de *front-end* y lo detallado en secciones anteriores de este capítulo, las mismas se programaron utilizando HTML 5, CSS y JavaScript, por lo que se verifica que “posee una interfaz de usuario basada en un formulario web usando tecnologías web estándares”.
- Se probó el uso de la aplicación en varios navegadores web actuales (Mozilla Firefox, Google Chrome, Microsoft Edge, Opera, Brave y Vivaldi), funcionando correctamente en ellos, por lo que se verifica que “debe ser compatible con los navegadores actuales más usados del mercado”.
- Como se mencionó en la Sección 5.3, se programó en el lenguaje Python utilizando el *framework* Flask y el motor de base de datos PostgreSQL.

Por otro lado, se cuenta además con una suite de *tests* automáticos desarrollados para los *scrapers* disponibles y para los parseadores de *PubMed* (`PubMedParser` y `PubMedResultsParser`). Estos verifican que puedan extraer la información de los sitios correspondientes de manera correcta, permitiendo detectar cuando algún sitio cambia su estructura y requiera adaptar su *scraper* para que siga funcionando.

Los mismos se programan en forma de *tests unitarios* mediante el módulo `unittest` de Python. Un *test* unitario prueba un componente particular mediante una serie de *asserts* o afirmaciones las cuales chequea que sean ciertas. Si al menos una afirmación no se cumple, el *test* falla. Por ejemplo, un *test* de `PubMedParser` estaría conformado de la siguiente manera:

- Dato de entrada: 28710245 (PMID)
- Salida: objeto `parser` no nulo de tipo `PubMedParser`.
- Asserts:

| Atributo | Valor comprobado |
|-----------------------------------|--|
| <code>parser.pmid</code> | "28710245" |
| <code>parser.doi</code> | "10.1136/bcr-2017-221182" |
| <code>parser.titulo</code> | "Abciximab-induced Acute Profound Thrombocytopenia Postpercutaneous Coronary Intervention" |
| <code>parser.autores</code> | "Golden T;Ghazala S;Wadeea R;Junna S;" |
| <code>parser.fecha</code> | "07/14/2017" |
| <code>parser.url</code> | "https://www.ncbi.nlm.nih.gov/pmc/articles/28710245/" |
| <code>len(parser.abstract)</code> | 939 |

Si alguno de los *asserts* anteriores no se cumple, quiere decir que hubo algún cambio en la estructura de la página fuente, por lo que se requiere actualizar el *scraper* correspondiente.

Capítulo 6

Conclusiones y trabajos futuros

En este capítulo se presentan las conclusiones de este proyecto junto a posibles trabajos futuros que permitan su mejora y expansión.

6.1. Conclusiones

Tras haber concretado este proyecto, y coincidiendo con la bibliografía citada en el Capítulo 1, se puede afirmar que las redes de aprendizaje profundo, específicamente, las redes neuronales convolucionales y *Transformers*, son útiles para realizar procesamiento del lenguaje natural (NLP) dando buenos resultados en esta tarea. Sin embargo, se han observado una serie de problemas importantes que influyeron en el desempeño de los modelos neuronales. Uno de ellos fue la gran presencia de dispersión en los ejemplos de entrenamiento, es decir, que la longitud (7500, 10 000 y 20 000 palabras) de los ejemplos supera, enormemente, a la cantidad de los mismos (3400 para el entrenamiento), cuando lo ideal sería que la cantidad de ejemplos supere a la longitud de los mismos. Recuérdese que estas longitudes de ejemplos eran necesarias para ser representativas del contenido de las publicaciones, permitiendo que los modelos extraigan conclusiones del texto completo de las mismas; de ser menor, se hubiera perdido información importante para la clasificación. Otro de los problemas está relacionado con el desbalance en la cantidad de ejemplos por clase, teniendo algunas de ellas muy pocos. Aun cuando estas clases de pocos ejemplos fueron agrupadas en el conjunto *other*, varias de las 17 clases resultantes siguieron conteniendo una cantidad pequeña de ejemplos, por lo que la clasificación para ellas no fue muy efectiva. Como contraparte, las clases con mayor cantidad de ejemplos resultaron bien clasificadas. Cabe destacar que no todas las clases de pocos ejemplos tuvieron resultados regulares en la clasificación, sino que, por el contrario,

dieron muy buenos resultados. Esto merece un análisis más profundo de la naturaleza de los datos de entrenamiento, los cuales se confeccionaron confiando plenamente en la información otorgada por *DGIdb* y el procesamiento de los mismos descrito en el Capítulo 3. Una estrategia que se podría utilizar sería la reducción en la cantidad de clases de entrenamiento. Esto es, realizar un agrupamiento mayor, con el cual se obtendrían mejores resultados en la clasificación, a costa de una menor calidad del producto final al ser capaz de clasificar menor cantidad de tipos de interacción.

Por su parte, la base de datos de publicaciones etiquetadas que se creó a partir de la información extraída de *DGIdb* y el *web scraping* puede ser utilizada para futuros proyectos, relacionados o no a este, donde se requieran entrenar otros modelos.

En cuanto a la aplicación web, la misma cumple con los requerimientos expuestos en su momento, constituyendo junto al modelo neuronal finalmente seleccionado, una herramienta útil para profesionales de la salud dentro del campo de la medicina de precisión.

De esta manera se puede afirmar, en este punto, que el objetivo general del proyecto, que es desarrollar un sistema de búsqueda inteligente de interacciones entre fármacos y genes en textos científicos utilizando aprendizaje profundo, ha sido alcanzado, cumpliendo, a la vez, con los objetivos específicos del mismo.

6.2. Trabajos futuros

Aunque el preprocesamiento de los datos es una tarea tediosa, el enfoque utilizado en este proyecto es simple ya que es, básicamente, una búsqueda y reemplazo de ocurrencias, con cierta heurística implementada y conjuntamente con una limpieza de los documentos. Mejorar este procesamiento en cuanto a calidad (obteniendo finalmente archivos de texto perfectamente coherentes, sin caracteres basura, y con alguna estrategia de búsqueda de entidades más robusta) y en cuanto a optimización (reducir lo más posible el tiempo necesario de procesamiento) puede ayudar a aumentar el volumen de ejemplos de entrenamiento y la calidad de los mismos, mejorando posiblemente el desempeño de los modelos neuronales. Al mismo tiempo, lograría que los tiempos de búsqueda de la aplicación web se reduzcan y se mejore la experiencia de usuario.

Las redes neuronales utilizadas fueron confeccionadas y entrenadas como parte del desarrollo del proyecto, sin embargo, recientemente un modelo preentrenado basado en *Transformers* desarrollado por Google denominado BERT (*Bidirectional Encoder Representations from Transformers*, o Re-

presentación de Codificador Bidireccional de *Transformers*) (Devlin, Chang, Lee, y Toutanova, 2019) ha estado demostrando dar los mejores resultados en lo que a modelos de aprendizaje profundo para NLP respecta. Es por ello que, sería interesante su utilización en el conjunto de datos de entrenamiento de este proyecto, pudiendo arrojar mejoras notables en los resultados de clasificación.

Las posibles mejoras que se puedan obtener implementando lo descrito en los párrafos anteriores, pueden dar lugar a una evaluación más rigurosa de los modelos entrenados, mediante la incorporación de otras medidas de desempeño como son *precisión* (p), *sensibilidad* (s^+), F_1Score , *Matthew Correlation Coefficients* (MCC) y media geométrica (G_m), utilizadas, comúnmente, para evaluar modelos de aprendizaje automático.

En cuanto a la aplicación web, como trabajo futuro se puede agregar funcionalidad adicional que brinde más utilidad a los usuarios, como por ejemplo alertas (cuando haya artículos nuevos relacionados con sus términos de interés publicados en *PubMed*, para estar actualizado con novedades de manera automática). Además, se puede trabajar en reducir el tiempo de espera mientras la aplicación obtiene resultados para mostrar al usuario. Finalmente, se puede publicar la misma *online* para uso gratuito por cualquier persona de Internet, sin embargo, no fue realizado ya que no formaba parte de los alcances del proyecto.

Referencias

- Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., ... Zimmermann, T. (2019, May). Software engineering for machine learning: A case study. En *International Conference on Software Engineering (ICSE 2019) - Software Engineering in Practice track*. IEEE Computer Society.
- Bugnon, L., Yones, C., Raad, J., Gerard, M., Rubiolo, M., Merino, G., ... Stegmayer, G. (2020). DL4papers: a deep learning model for interpretation of scientific articles. *Bioinformatics*, 36(11), 3499-3506.
- Burger, J. D., Doughty, E., Khare, R., Wei, C.-H., Mishra, R., Aberdeen, J., ... Hirschman, L. (2014, 09). Hybrid curation of gene-mutation relations combining automated extraction and crowdsourcing. *Database*, 2014. doi: 10.1093/database/bau094
- Caporaso, J. G., Baumgartner, J., William A., Randolph, D. A., Cohen, K. B., y Hunter, L. (2007, 05). MutationFinder: a high-performance system for extracting point mutation mentions from text. *Bioinformatics*, 23(14), 1862-1865. doi: 10.1093/bioinformatics/btm235
- Cheng, D., Knox, C., Young, N., Stothard, P., Damaraju, S., y Wishart, D. S. (2008, 05). PolySearch: a web-based text mining system for extracting relationships between human diseases, genes, mutations, drugs and metabolites. *Nucleic Acids Research*, 36(suppl_2), W399-W405. doi: 10.1093/nar/gkn296
- Coad, P., y Yourdon, E. (1991). *Object-oriented Design*. Yourdon Press.
- Cotto, K. C., Wagner, A. H., Feng, Y.-Y., Kiwala, S., Coffman, A. C., Spies, G., ... Griffith, M. (2017, 11). DGIdb 3.0: a redesign and expansion of the drug-gene interaction database. *Nucleic Acids Research*, 46(D1), D1068-D1073. doi: 10.1093/nar/gkx1143
- de la Guardia, C. (2016). *Python Web Frameworks*. O'Reilly Media.
- Devlin, J., Chang, M.-W., Lee, K., y Toutanova, K. (2019). *BERT: Pre-training of deep bidirectional transformers for language understanding*.
- Dos Santos, C., Xiang, B., y Zhou, B. (2015, 04). Classifying relations by ranking with convolutional neural networks. *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing, 1*. doi: 10.3115/v1/P15-1061
- Doughty, E., Kertesz-Farkas, A., Bodenreider, O., Thompson, G., Adadey, A., Peterson, T., y Kann, M. G. (2010, 12). Toward an automatic method for extracting cancer- and other disease-related point mutations from the biomedical literature. *Bioinformatics*, 27(3), 408-415. doi: 10.1093/bioinformatics/btq667

- Fawcett, T. (2004, 01). ROC graphs: Notes and practical considerations for researchers. *Machine Learning*, 31, 1-38.
- Flanagan, D. (2020). *JavaScript: The definitive guide: Master the world's most-used programming language* (7.^a ed.). O'Reilly Media.
- Gonzalez-Garay, M. L. (2014). The road from next-generation sequencing to personalized medicine. *Personalized Medicine*, 11(5), 523-544. doi: 10.2217/pme.14.34
- Goodfellow, I., Bengio, Y., y Courville, A. (2016). *Deep Learning*. MIT Press.
- Hidalgo-Vega, A. (2020). *El valor del medicamento desde una perspectiva social en Argentina y en países de su entorno*. Madrid, España: Fundación Weber. doi: 10.37666/L10-2020
- Hochreiter, S., y Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), 1735-1780. doi: 10.1162/neco.1997.9.8.1735
- Ioffe, S., y Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. En *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37* (p. 448-456). JMLR.org.
- Johnson, R., y Zhang, T. (2017, 01). Deep pyramid convolutional neural networks for text categorization. En *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics* (p. 562-570). doi: 10.18653/v1/P17-1052
- Kingma, D. P., y Ba, J. (2014). *Adam: A method for stochastic optimization*.
- Lai, S., Xu, L., Liu, K., y Zhao, J. (2015). Recurrent convolutional neural networks for text classification. En *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (pp. 2267-2273). AAAI Press.
- Lee, K., Kim, B., Choi, Y., Kim, S., Shin, W., Lee, S., ... Kang, J. (2018, 01). Deep learning of mutation-gene-drug relations from the literature. *BMC Bioinformatics*, 19, 21. doi: 10.1186/s12859-018-2029-1
- Lee, K., Lee, S., Park, S., Kim, S., Kim, S., Choi, K., ... Kang, J. (2016, 04). BRONCO: Biomedical entity Relation ONcology COrpus for extracting gene-variant-disease-drug relations. *Database*, 2016. doi: 10.1093/database/baw043
- Lee, K., Shin, W., Kim, B., Lee, S., Choi, Y., Kim, S., ... Kang, J. (2016, 08). HiPub: translating PubMed and PMC texts to networks for knowledge discovery. *Bioinformatics*, 32(18), 2886-2888. doi: 10.1093/bioinformatics/btw511
- Lee, S., Kim, D., Lee, K., Choi, J., Kim, S., Jeon, M., ... Kang, J. (2016, 10). BEST: Next-Generation Biomedical Entity Search Tool for Knowledge Discovery from Biomedical Literature. *PLOS ONE*, 11(10), 1-16. doi:

- 10.1371/journal.pone.0164680
- Mikolov, T., Chen, K., Corrado, G., y Dean, J. (2013). *Efficient estimation of word representations in vector space*.
- Mitchell, T. M. (1997). *Machine Learning* (1.^a ed.). USA: McGraw-Hill, Inc.
- Object Management Group (OMG). (2003). *OMG Unified Modeling Language Specification Version 1.5*.
- Pennington, J., Socher, R., y Manning, C. D. (2014). GloVe: Global vectors for word representation. En *Empirical Methods in Natural Language Processing (EMNLP)* (pp. 1532–1543).
- Siddiqui, M., y Verma, D. (2011, 05). Cross Site Request Forgery: A common web application weakness. En *2011 IEEE 3rd International Conference on Communication Software and Networks* (p. 538-543). doi: 10.1109/ICCSN.2011.6014783
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., y Salakhutdinov, R. (2014, enero). Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1), 1929–1958.
- Tobella, J. (2010). *Medicina personalizada posgenómica. Conceptos prácticos para clínicos*. Elsevier Masson.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. En *Proceedings of the 31st International Conference on Neural Information Processing Systems* (p. 6000–6010). Red Hook, NY, USA: Curran Associates Inc.
- Wei, C.-H., Harris, B. R., Kao, H.-Y., y Lu, Z. (2013, 04). tmVar: a text mining approach for extracting sequence variants in biomedical literature. *Bioinformatics*, 29(11), 1433-1439. doi: 10.1093/bioinformatics/btt156
- Yin, W., Kann, K., Yu, M., y Schütze, H. (2017). Comparative study of CNN and RNN for natural language processing. *CoRR*, abs/1702.01923.