Sistema de clasificación de historias clínicas con deep learning

Trabajo de fin de grado

David Pérez Asensio
Doble grado en Ingeniería Informática y Matemáticas

Directores
Alberto Díaz Esteban
Antonio Fernando García Sevilla

8 de junio, 2018

Departamento de Ingeniería de Software e Inteligencia Artifcial
Facultad de Informática
Universidad Complutense de Madrid (UCM)

A mis padres.

Resumen

Las historias clínicas electrónicas contienen extensa información de gran utilidad en ensayos clínicos a gran escala y en el diagnóstico de pacientes. En este trabajo abordamos el problema de asignar códigos ICD-9 tomando como entrada notas médicas escritas por personal sanitario. Para ello utilizamos el reciente dataset MIMIC-III, construyendo un sistema que facilita el diseño, la implementación y la evaluación de multitud de clasificadores *multilabel* fundamentados en regresión logística, redes neuronales *feedforward* y redes neuronales recurrentes. Nuestros resultados son similares a los obtenidos por otros investigadores. Los modelos basados en *deep learning* resuelven con éxito la tarea de clasificación para las 10 etiquetas ICD-9 más frecuentes.

Palabras clave: clasificación *multilabel*, MIMIC, códigos ICD-9, regresión logística, redes neuronales, redes neuronales recurrentes.

Abstract

Electronic health records contain extensive information which is very useful in large scale clinical studies and patient diagnosing. In the present work we attempt the problem of auto assigning ICD-9 codes based on free-text medical notes. To this end we use the relatively new MIMIC-III dataset to build a framework that enables the design, implementation and evaluation of a variety of multilabel classifiers based on logistic regression, feedforward neural networks and recurrent neural networks. Our results align with those obtained by other researchers, with our deep learning models solving the top 10 label ICD-9 classification task moderately well.

Keywords: multilabel classification, MIMIC, ICD-9 codes, logistic regression, neural networks, recurrent neural networks.

Índice general

1. Introducción **5**

2. Trabajo relacionado **8**
2.1. Dataset 8
2.2. Preprocesamiento 12
2.3. Trabajo previo 15
2.4. Aspectos éticos y legales en la investigación con sujetos humanos 15

3. Modelos **18**
3.1. Regresión logística 18
3.2. Redes neuronales *feedforward* 24
3.2.1. Función de coste 25
3.2.2. Funciones de activación 29
3.3. Redes neuronales recurrentes 33

4. Implementación y metodología práctica **38**
4.1. Generación de vocabulario 40
4.2. Generación de vectores *bag of words* 41
4.3. Diseño y evaluación de los clasificadores 42
4.3.1. Regresión logística 42
4.3.2. Red neuronal *feedforward* 43
4.3.3. Redes neuronales recurrentes 44
4.4. Otros desarrollos 45
4.5. Entorno de ejecución 48

5. Resultados **50**
5.1. Métricas 50
5.2. Análisis cualitativo y cuantitativo 51

6. Conclusiones y trabajo futuro **58**

Capítulo 1

Introducción

En este trabajo abordamos el problema de diagnosticar pacientes en base a texto en formato libre escrito por médicos y otro personal sanitario. En concreto, la entrada es un conjunto de notas médicas sobre un paciente y la salida es un conjunto de enfermedades. El trabajo consiste en investigar, implementar y evaluar distintos clasificadores que asignen a cada historial clínico un conjunto de códigos, llamados códigos ICD-9, que codifican enfermedades, afecciones y lesiones [26]. Para el desarrollo de los clasificadores se exploran distintas técnicas basadas en procesamiento de lenguaje natural (NLP) y aprendizaje automático (*machine learning*). El objetivo final del trabajo es hacer una comparativa del rendimiento de los clasificadores, contrastando los resultados de los que hasta la fecha se consideran estado del arte con los resultados obtenidos por otros investigadores que han abordado este problema y problemas similares (véase el apartado 2.3). Estos clasificadores están basados en *deep learning*. Asimismo, el trabajo pretende también exponer los fundamentos teóricos de las técnicas y los algoritmos empleados, intentando justificar, en la medida de lo posible, por qué funcionan y cuánto se adecúan a la resolución de este problema en particular.

Implementaremos 3 modelos:

- regresión logística (*logistic regression*),
- redes neuronales prealimentadas (*feedforward neural networks*), y
- redes neuronales recurrentes (*recurrent neural networks*).

El primero es un modelo estadístico ampliamente usado y el segundo goza de una gran popularidad para multitud de problemas de aprendizaje automático, en particular para problemas de clasificación. Las redes neuronales recurrentes y sus variantes son redes neuronales especialmente útiles para tratar de manera flexible y dinámica datos temporales. Los fundamentos de los tres modelos se explican en el capítulo 3.

Los modelos toman como entrada vectores reales $\boldsymbol{x} \in \mathbb{R}^D$ y producen como salida vectores binarios $\boldsymbol{y} \in \{0,1\}^K$, donde K es el número de clases. Para nuestro problema, la entrada es un historial médico $\boldsymbol{x} \in \mathbb{R}^D$ y la salida es un vector $\boldsymbol{y} \in \{0,1\}^K$, donde K es el número de enfermedades y la entrada k-ésima del vector $\boldsymbol{y}$ indica si el paciente padece la enfermedad k. Esto se conoce como codificación *multi-hot*. Se trata de un problema de clasifición *multiclass multilabel*, pues hay más de dos clases ($K > 2$) y cada ejemplo puede asignarse a más de una clase. Nosotros abordaremos el problema para $K = 10$, tratando de implementar clasificadores que sean capaces de predecir las 10 enfermedades más comunes de nuestro dataset. El número D se conoce como dimensión del espacio de características (*features*). De alguna manera, tendremos que convertir el texto de un historial médico a un vector de números reales que codifique, en la medida de lo posible, toda la información que podamos. Este proceso se conoce como extracción de características (*feature extraction*) y es de vital importancia: si la entrada de nuestros algoritmos modela mal el dominio de nuestro problema, obtendremos malos resultados, por muy buenos que sean nuestros algoritmos de clasificación. Nosotros usaremos una representación muy sencilla para la entrada, conocida como *bag of words model*, ampliamente usado en procesamiento de lenguaje natural.

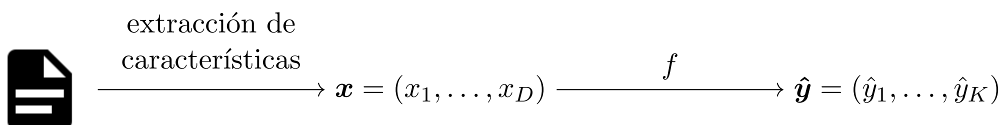


Figura 1.1: Diagrama representando el proceso de clasificación. Dado un texto, el proceso de extracción de características lo convierte a un vector real de tamaño fijo $\boldsymbol{x}$. El modelo, una vez entrenado, toma como entrada este vector y lo clasifica usando la función aprendida f. El resultado es un vector $\hat{\boldsymbol{y}}$ de enfermedades en codificación *multi-hot*.

En cuanto a cómo aprenden los modelos, los tres son modelos de aprendizaje automático *supervisado*. Esto significa que los modelos aprenden una función que mapea una entrada a una salida basándose en parejas de entrada-salida que llamamos *ejemplos* de entrenamiento (*training set*). Los algoritmos procesan el *training set* e infieren una función que "etiqueta" la entrada. El objetivo deseado es que la función sea capaz de determinar las etiquetas de una entrada jamás vista anteriormente, se decir, que generalice razonablemente bien a partir del conjunto de datos de entrenamiento. [72] Para entrenar el algoritmo, dispondremos de un dataset de notas médicas y diagnósticos (véase el apartado 2.1). De este dataset obtendremos un *vocabulario*, que nos permitirá construir nuestros vectores de entrada. Explicamos esto en detalle en el apartado 2.2. A continuación daremos como

entrada a nuestro algoritmo los datos de entrenamiento: las parejas de vectores de entrada junto con sus respectivas etiquetas de enfermedades. Tras la etapa de entrenamiento, el algoritmo aprende la función de clasificación. Finalmente, evaluamos la eficiencia del clasificador construido dando como entrada a nuestro algoritmo un conjunto de datos nuevo jamás visto anteriormente (*test set*), y calculamos métricas que nos permitan comparar su rendimiento con otros clasificadores. Detallamos el proceso de implementación y entrenamiento de los clasificadores en el capítulo 4. Analizamos los resultados obtenidos en el capítulo 5, y concluimos y proponemos vías de trabajo futuro en el capítulo 6.

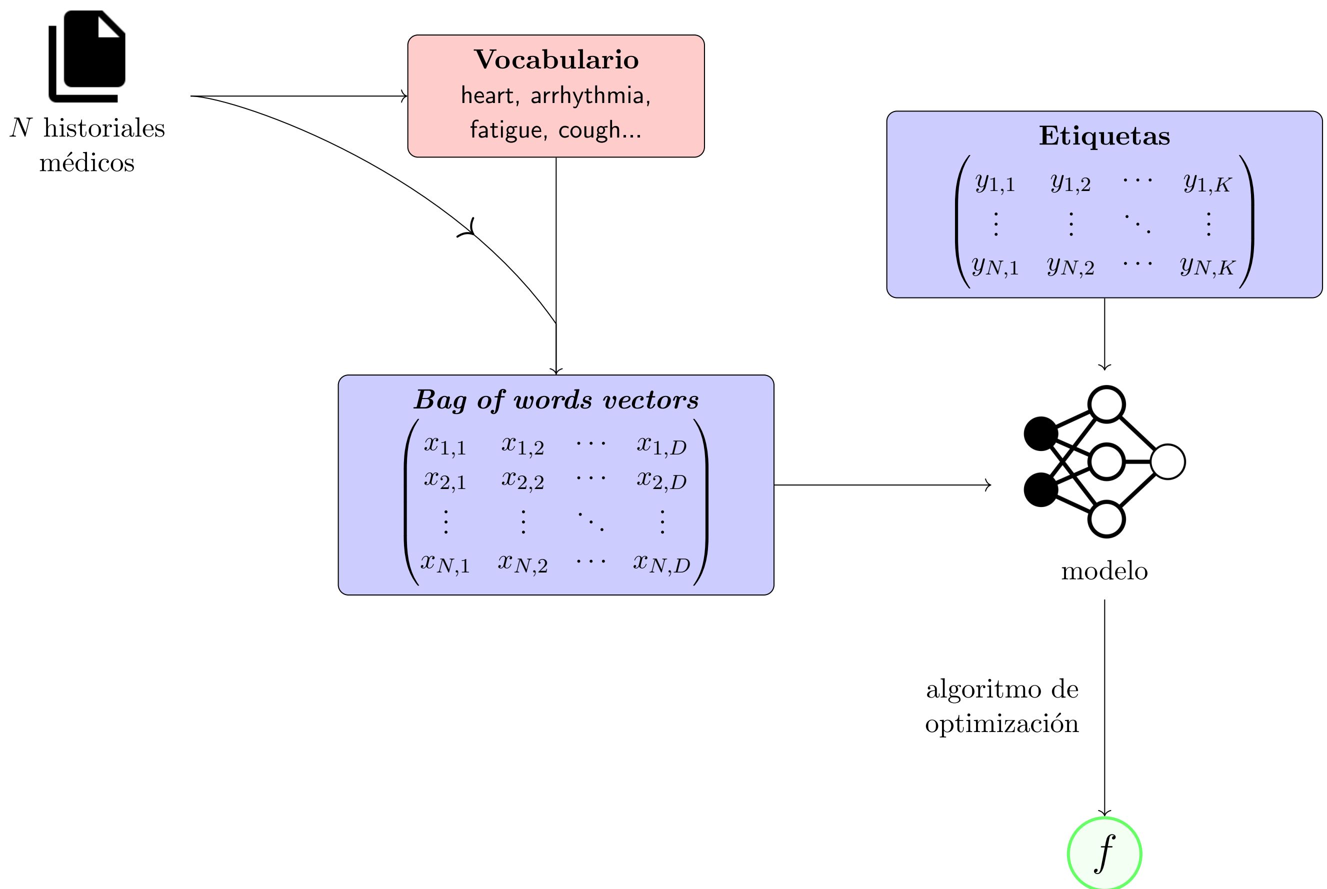


Figura 1.2: Diagrama representando el proceso de entrenamiento. Dado un corpus de historiales médicos, se escoge un subconjunto de palabras representativas que conforman el vocabulario. Con ayuda de éste, los historiales médicos se convierten a vectores *bag of words*. Un algoritmo de optimización entrena el modelo usando los vectores de entrada y sus respectivas etiquetas, produciendo una función de clasificación.

Capítulo 2

Trabajo relacionado

En este capítulo introducimos el dataset que usamos en el trabajo, así como las técnicas que empleamos para generar el vocabulario y la entrada de nuestros modelos. También describimos brevemente el trabajo previo realizado por otros investigadores que han abordado el problema introducido y otros similares, y concluimos con algunos comentarios acerca de aspectos éticos y legales en la investigación con sujetos humanos.

2.1. Dataset

Los registros médicos electrónicos (*electronich health records, EHR*) contienen notas detalladas escritas por personal sanitario sobre la salud física y mental de los pacientes, análisis de los resultados de laboratorio, tratamientos y más. Esta información es valiosa para mejorar el cuidado médico. [38]

Nosotros obtendremos nuestros datos de MIMIC, "un dataset abierto desarrollado por el MIT Lab for Computational Physiology, abarcando datos médicos anonimizados asociados a unos 40000 pacientes de cuidados intensivos. Incluye datos demográficos, signos vitales, resultados de laboratorio, medicaciones y más". [34] En particular, utilizaremos la última versión (publicada en diciembre de 2015), MIMIC-III [14]. Los datos provienen del Beth Israel Deaconess Medical Center en Boston, Massachusetts, y son accesibles a investigadores bajo un acuerdo de uso de datos. La accesibilidad del dataset ha mejorado la realización de ensayos clínicos y la reproducidibilidad de estudios a un nivel que antes no era posible.

El dataset puede descargarse en formato CSV e importarse en una base de datos. La base de datos entera ocupa unos 80 GiB. La tabla más importante para nosotros es la de las notas médicas (`note_events`) que contiene

2083180 notas de 46146 pacientes. [1] Una nota médica de ejemplo se muestra en la cuadro 2.1. Las notas están asociadas a la visita del paciente al hospital, la fecha en la que se tomó y el identificador del facultativo que la registró en el sistema. [39] Al tratarse de registros médicos de una UCI, es probable que enfermedades más debilitantes y enfermedades en etapas avanzadas estén sobrerrepresentadas en el dataset. Los pacientes están diagnosticados con 14 enfermedades en promedio (véase la figura 2.1).

Nota médica (extracto):

*Patient is a 84 year old male with daily ETOH use, tobacco use, hypothyroidism, h/o throat cancer and dementia who presented to [**Hospital1 3356**] last Saturday ([**7-24**]) after he was found to have a diffuse rash, fever to 102. Per patient's son, on the day PTA he started complaining of back pain which improved with 2 advil. The next day his daughter found him on the floor at his house with rash on his neck and back. Of note, the patient's son reports that he has had progressive dysphagia, poor PO intake and weight loss over the last year. At [**Hospital1 632**] he was also noted to have ARF, transaminitis, UTI. The patient's rash was noted to involve his entire body, sparing the mucous membranes. Per report, it exfoliated without bullae. Dermatology was involved, biopsy was done and showed leukocyto clastic vasculitis that was presumed to be [**12-21**] advil. Patient was initially treated for RMSF with CTX/Levo/Doxy but the this was d/c'd and he treated with a prednisone taper (PO) and supportive care, almost like a burn victim. Rash improved over the course. On Tuesday the patient had afib with RVR, requiring a dilt drip and was transferred to the ICU. Reverted back to normal sinus rhythm.* `[...]`

Etiquetas:

5849: Acute Kidney Failure
51881: Acute respiratory failure
5990: Urinary tract infection

Tabla 2.1: Nota de ejemplo del dataset, junto con las etiquetas ICD-9 asignadas al paciente pertenecientes a la tabla 2.2. El texto *[** **]* denota información anonimizada.

El dataset contiene 6984 etiquetas ICD-9 únicas que se distribuyen muy sesgadamente (véase la figura 2.2): los 105 códigos más frecuentes cubren el 50 % de todas las apariciones de las etiquetas, 1503 etiquetas aparecen una única vez en todo el dataset y 3111 etiquetas tienen menos de 5 ejemplos.

[1] El repositorio de código que acompaña a este trabajo contiene multitud de scripts de exploración del dataset de donde se han elaborado estos estadísticos. Consúltese el capítulo 4.

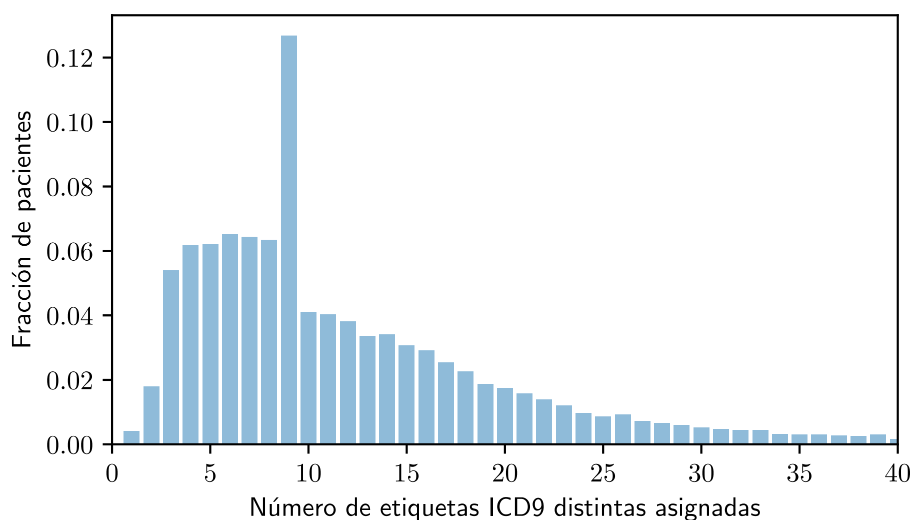


Figura 2.1: Distribución del número de etiquetas asignadas por paciente.

Un sistema de aprendizaje automático supervisado precisa de una gran cantidad de ejemplos etiquetados para aprender una función de clasificación que generalice adecuadamente. Por consiguiente, no podemos entrenar nuestros clasificadores sobre todas las etiquetas ICD-9. Nos limitamos a las 10 etiquetas más frecuentes, como muchos otros en la literatura ([38, 25]). Éstas se listan en la tabla 2.2.

Etiqueta ICD-9	**Número de pacientes**	**Fracción de pacientes**
4019: Hypertension	20703	0.445
4280: Congestive heart failure	13111	0.282
42731: Atrial fibrillation	12891	0.277
41401: Coronary artherosclerosis	12429	0.267
5849: Acute kidney failure	9119	0.196
25000: Diabetes mellitus, type II	9058	0.195
2724: Hyperlipidemia	8690	0.187
51881: Acute respiratory failure	7497	0.161
5990: Urinary tract infection	6555	0.141
53081: Esophageal reflux	6326	0.136

Tabla 2.2: Distribución de las 10 etiquetas ICD-9 más frecuentes.

Preprocesamos el dataset de la siguiente manera. Primero eliminamos todos los pacientes que no tengan una de estas etiquetas, quedando 32063 pacientes. De éstos, 31865 tienen al menos una nota médica escrita. A continuación filtramos todas las etiquetas ICD-9 de estos pacientes que no estén

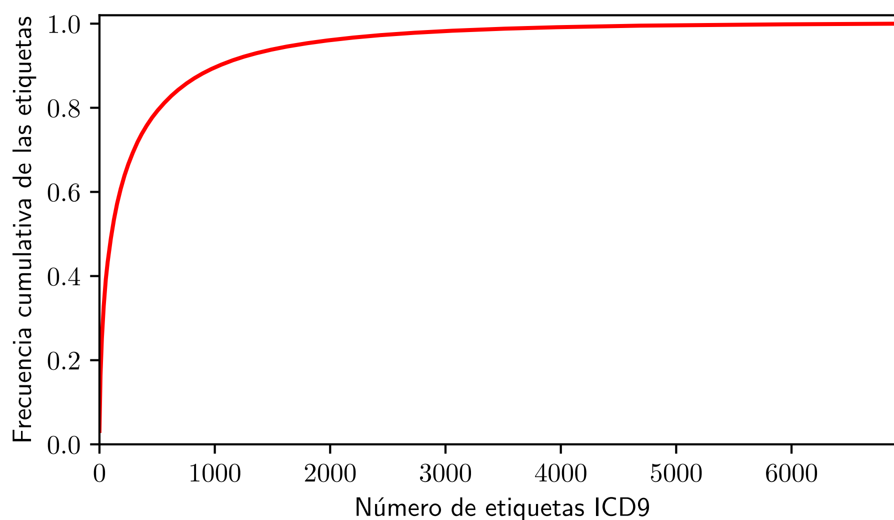


Figura 2.2: Distribución cumulativa de la frecuencia de aparición de etiquetas en MIMIC-III.

en la tabla 2.2. Estos pacientes tienen un total de 1522558 notas médicas asignadas. Sin embargo, la distribución del número de notas escritas por paciente no es centrada, como puede observarse en la figura 2.3.

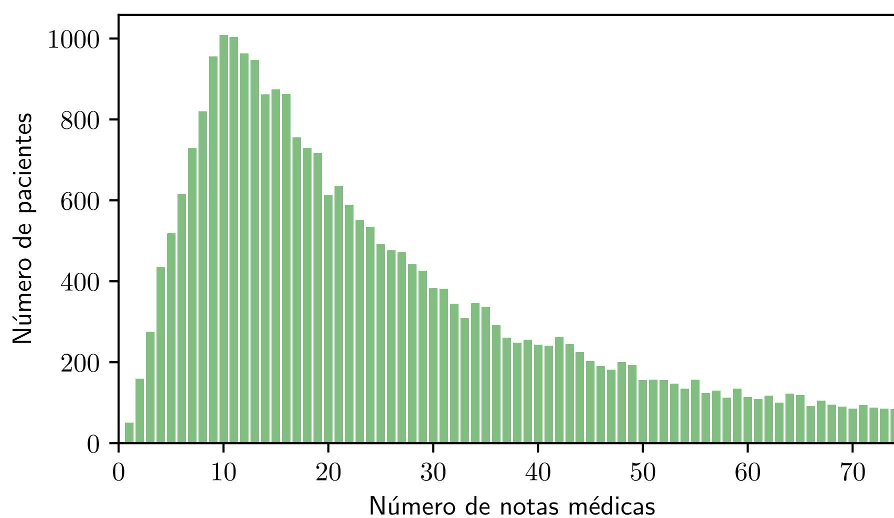


Figura 2.3: Distribución del número de notas médicas asociadas a pacientes que tienen asignadas al menos una etiqueta ICD-9 de la tabla 2.2.

Filtramos entonces aún más el dataset, elaborando una lista de a lo sumo

20 notas por paciente ordenadas cronológicamente. Esto arroja un resultado final de 524755 notas. Estos documentos son de naturaleza dispar: informes de radiología, análisis de electrocardiogramas, partes de enfermería... La longitud de las notas es muy variada, con una distribución peculiar (véase la figura 2.4). Esto podría suponer un problema si la entrada de los modelos dependiera directamente de la longitud de la nota. Veremos como solventamos esto en el apartado 2.2 definiendo una noción de relevancia de una palabra en un texto que tiene en consideración la longitud del texto.

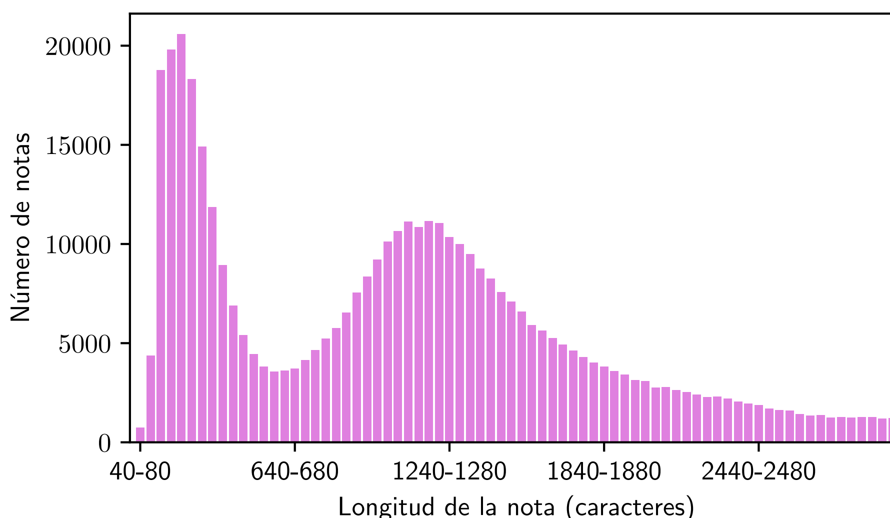


Figura 2.4: Distribución de la longitud de las notas médicas asociadas a los pacientes del dataset filtrado. Cada barra agrupa notas cuya longitud no difiere en más de 40 caracteres.

Particionamos *los pacientes* (y por tanto el subconjunto de notas asociadas) en tres *splits*, con una proporción 50-25-25: *training set*, *validation set* y *test set*. Usamos el *training set* para entrenar nuestros modelos, el *validation set* para comprobar periódicamente que no cometemos *overfitting* y para optimizar los hiperparámetros, y el *test set* para evaluar el rendimiento de las funciones de clasificación aprendidas. La división la hacemos secuencialmente por el identificador único del paciente.

2.2. Preprocesamiento

Comenzamos pues la tarea de construcción de nuestros clasificadores, y lo primero que hemos de hacer es preprocesar el dataset, ahora que tenemos una idea general de sus contenidos, para poder trabajar con él más fácilmente. Hacemos caso de las recomendaciones en [20, cap. 11] y fijamos lo primero

la manera en la que vamos a evaluar el rendimiento de los clasificadores que implementemos (véase el apartado 5.1). Todas las decisiones de diseño persiguen mejorar estas métricas.

En esta sección describimos cómo preprocesamos el dataset para convertir las notas médicas a la representación interna en vectores que mencionamos en el primer capítulo, conocidos como vectores *bag of words*. Lo haremos a nivel teórico, siendo la explicación aplicable a cualquier corpus lingüístico y dejando los detalles concretos y de implementación para el capítulo 4.

Supongamos que tenemos un conjunto de documentos electrónicos y que queremos construir un sistema de búsqueda y recuperación de información de texto con un funcionamiento similar a un buscador web. Es decir, ante una consulta en forma de cadena, hemos de seleccionar aquellos documentos que sean más *relevantes*. Una primera aproximación podría consistir en eliminar todos los documentos que no contengan una de las palabras que componen la consulta, y devolver los restantes. Pero ante una consulta genérica y un corpus grande la cantidad de documentos devueltos sería enorme. Además, esta solución no permite distinguir entre cuáles de ellos son más relevantes.

Supongamos que hemos procesado el conjunto de documentos $\mathcal{D} = \{d_n\}_{n=1}^{N}$, obteniendo el conjunto de términos que aparecen en todos los documentos $\mathcal{T} = \{t_i\}_{i=1}^{T}$. Una manera de tener una noción de la relevancia de una palabra en un documento es calcular la frecuencia de aparición de la palabra en el documento. Sin embargo, los documentos que son más largos contendrán con mayor probabilidad cualquier palabra, por lo que conviene normalizar la frecuencia de aparición por la longitud del documento. Esta medida se conoce como *term-frequency*, es decir

$$\mathrm{tf}(t, d_n) = \frac{f_{t,d_n}}{|d_n|}, \tag{2.1}$$

donde f_{t,d_n} es el número de veces que aparece el término t en el documento d_n y $|d_n|$ denota la longitud (en palabras) del documento d_n.

No obstante, las palabras que más aparecen en un documento no son casi nunca las que más información transmiten del documento, desde un punto de vista semántico. Esto es porque en cualquier lengua las palabras más frecuentes son siempre palabras vacías (*stop words*) como artículos, pronombres, preposiciones... Para tener en cuenta cuánta información aporta una palabra en un documento, puede calcularse si el término es común o raro en todos los documentos, es decir, calcular una expresión inversamente proporcional a la frecuencia de aparición de un término en el corpus:

$$\frac{N}{\big|\{d_n \in \mathcal{D} \mid t \in d_n\}\big|}. \tag{2.2}$$

Obsérvese que la expresión 2.2 no está definida si t es un término que no aparece en el corpus. Por ello se suele sumar 1 al denominador. Se define la frecuencia inversa de documento (*inverse document frequency*) de un término como el logaritmo de la expresión resultante:

$$\mathrm{idf}(t, d_n) = \log\left(\frac{N}{1 + \left|\{d_n \in \mathcal{D} \mid t \in d_n\}\right|}\right). \tag{2.3}$$

tf-idf (*term frequency — inverse document frequency*) es una métrica que define una noción de cuán relevante es una palabra en un documento concreto, teniendo en cuenta la frecuencia de aparición del término en el documento y la cantidad de información que aporta en ese documento, comparado con el resto del corpus. Es simplemente el producto de las métricas (2.1) y (2.3), es decir:

$$\begin{aligned}\text{tf-idf}(t, d_n) &= \mathrm{tf}(t, d_n) \times \mathrm{idf}(t, d_n) \\ &= \frac{f_{t,d_n}}{|d_n|} \log\left(\frac{N}{1 + \left|\{d_i \in \mathcal{D} \mid t \in d_i\}\right|}\right).\end{aligned} \tag{2.4}$$

Para seleccionar un subconjunto de términos representativos de un corpus, podemos calcular las métricas $\text{tf-idf}(t_i, d_n)$ para todo $i \in \{1, \ldots, T\}, n \in \{1, \ldots, N\}$, y "resumir" la relevancia de cada término en el corpus como, por ejemplo,

$$\mathrm{relevancia}(t_i, \mathcal{D}) = \sum_{n=1}^{N} \text{tf-idf}(t_i, d_n).$$

El conjunto de términos escogidos de esta manera se denominará *vocabulario* del corpus.

Una vez calculado el vocabulario, queda por determinar cómo vamos a representar nuestros documentos en forma de vectores numéricos haciendo uso de él. Existen muchas alternativas. Nostros usaremos una representación muy sencilla, conocida como *bag of words model*, en la que cada documento es asignado un vector de números enteros de longitud igual a la longitud del vocabulario. Cada entrada del vector es el número de veces que aparece un término del vocabulario en ese documento concreto. Es decir, un texto se representa como una "bolsa" (un multiconjunto) de las palabras que lo componen. La representación *bag of words* es una representación extremadamente simplificada, pues no tiene en cuenta el orden de las palabras en un documento — sólo el conteo de las palabras importa. Esta carencia puede paliarse extendiendo la representación al conteo de n-gramas. [69].

2.3. Trabajo previo

El problema de asignar códigos ICD-9 automáticamente en base a registros médicos se ha abordado durante décadas. Larkey y Croft diseñaron clasificadores para resúmenes de alta en 1996. [54] El problema de asignar códigos ICD-9 a reportes de radiología se abordó en 2007. [19] El mayor cuerpo de trabajo previo se ha centrado en diseñar sistemas basados en reglas y técnicas tradicionales de *machine learning* como máquinas vectores soporte jerárquicas y regresión logística. [44] La mayoría de estos modelos han usado representaciones basadas en el modelo *bag of words*. En varios casos los sistemas basados en reglas exhiben mejor rendimiento. [19] Esto no debería ser muy sorprendente, ya que los sistemas basados en reglas se asemejan más a cómo los médicos diagnostican a los pacientes.

Las aproximaciones de extremo a extremo (*end-to-end*) han ganado tracción en los últimos años. Es decir, se apuesta cada vez más por sistemas basados en el aprendizaje automático, en donde no se precisa de reglas diseñadas por expertos con conocimientos médicos avanzados. Estos sistemas se basan en algoritmos de aprendizaje para modelar la distribución subyacente de los datos y suelen escalar mejor y funcionar con casos más generales. [25] En particular, los sistemas basados en *deep learning* son cada vez más populares. Sistemas de diagnóstico automático como DoctorAI de Choi et al. (2015) han cosechado éxitos apostando por las redes neuronales recurrentes. [8] Prakash et al. (2017) usan redes neuronales con memoria para utilizar una base de conocimientos médicos. [50]

En definitiva, se trata de un área de investigación muy amplia y muy activa, y existen muchas vías de trabajo abiertas y prometedoras. Nuestro trabajo se basa en gran parte en el de Nigam (2016); [38] replicamos sus modelos y realizamos numerosos experimentos sobre ellos. Nigam no publicó el código de sus modelos, así que este trabajo implementa los clasificadores desde cero (véase el capítulo 4). A lo largo del desarrollo del presente trabajo, en el último año, se han publicado dos trabajos muy exhaustivos que también se fundamentan en el de Nigam, por Huang et al. (feb. 2018) [25] y Ayyar et al. (abril 2018). [55]

2.4. Aspectos éticos y legales en la investigación con sujetos humanos

MIMIC-III es accesible a cualquiera bajo un acuerdo de uso de los datos. Para obtener acceso al dataset hay que realizar una formación sobre ética en la investigación. El curso es online y se llama "Data or Specimens Only Research", elaborado por CITI Program (*Collaborative Institutional Training*

Initiative). [9]. Tras realizar el curso y superar las pruebas, hay que realizar una solicitud en PhysioNet [45] (donde se aloja el dataset) aportando la acreditación del curso y explicando el proyecto de investigación en el cual se va a usar MIMIC. La solicitud es revisada manualmente, y, si es concedida, se obtienen unas credenciales para descargar la base de datos de un servidor.

Eventos históricos y abusos relacionados con la experimentación con personas han motivado el desarrollo de principios éticos y regulaciones que gobiernan la investigación con sujetos humanos. La preocupación en el mundo moderno sobre la experimentación con humanos se intensificó como resultado de los Juicios de Núremberg tras el final de la Segunda Guerra Mundial, en los que 23 médicos del régimen Nazi fueron acusados de crímenes contra la humanidad por sus experimentos en los campos de concentración. De allí surgió el Código de ética médica de Núremberg (1947), en el que se exponen diez directrices que regulan los "experimentos médicos permisibles", entre las que destacan el requisito esencial de consentimiento voluntario y que los beneficios de la investigación han de exceder los riesgos. De especial importancia es también el Informe Belmont (*Belmont Report*), un informe redactado por el Departamento de Salud, Eduación y Bienestar de los Estados Unidos en 1978 que expone principios éticos que guían la elaboración de leyes y la conducta de los investigadores que trabajan con sujetos humanos. [66] Entre estos principios están el respeto a las personas, la beneficiencia (la filosofía de "no hacer daño" a los sujetos humanos a la vez que se maximizan los beneficios para el proyecto de investigación) y la justicia (distribuir los riesgos y los beneficios justamente entre todos los participantes).

Hoy en día los investigadores pueden realizar importantes avances en medicina sin siquiera tener una interacción humana con los sujetos, gracias a la investigación basada en registros (*records-based research*). Los mayores riesgos asociados a la investigación con estos datos es la invasión de la *privacidad* y posibles infracciones de *confidencialidad*. Los riesgos de privacidad están relacionados mayoritariamente con los métodos con los que se ha obtenido información de los sujetos, que evidentemente son muy bajos en los estudios en los que el participante ha dado su consentimiento para que se use su información personal. Por otra parte, la confidencialidad se refiere a la manera en la que esta información personal es manejada, una vez que ya ha sido obtenida. Es decir, ¿cómo usar y almacenar la información de una manera adecuada y consistente con la forma en la que se adquirió? Leyes y numerosos comités que las revisan se encargan de dilucidar estas cuestiones, y a menudo las respuestas y la manera de actuar no son evidentes. Por ejemplo, las consecuencias son claras si información médica de un paciente sobre su drogadicción se filtra a alguien encargado de cumplir la ley, pero las normas son más ambiguas en aŕeas relativamente nuevas como la investigación genética. Supongamos que un investigador se encarga de analizar el

genoma humano para identificar genes que favorecen la aparición de un tipo de cáncer, o una enfermedad neurológica severa. ¿Debería el investigador informar a los sujetos portadores de estos genes de que son más propensos a contraer cáncer? ¿O advertir sólo a aquellos que tienen la intención de tener hijos?

Anonimizar los datos (esto es, eliminar todo tipo de información identificativa de una persona) es una manera de reducir el riesgo de que los investigadores cometan una infracción de confidencialidad. Por ello, la base de datos de MIMIC está anonimizada (como se ve en la tabla 2.1), de acuerdo con los estándares del *Health Insurance Portability and Accountability Act (HIPAA)*. Esto conlleva una limpieza estructuada de los datos que consiste en omitir los nombres de los pacientes, números de teléfono, direcciones y nombres de hospitales. Las fechas también se han desplazado un *offset* fijo por cada paciente en el futuro, de manera que se preservan los intervalos entre los eventos registrados en el dataset. Todas las fechas ocurren entre el año 2100 y el 2200. [14] Todo este trabajo lleva mucho tiempo y esfuerzo y es propenso a errores si es llevado a cabo manualmente por expertos. Existe software, como el programa `deid`, que facilita esta ardua tarea. [37] Curiosamente, el problema de anonimizar registros médicos electrónicos se ha abordado también mediante *deep learning*. En el trabajo de Dernoncourt et al. se entrenan redes neuronales recurrentes con el dataset de MIMIC, consiguiéndose excelentes resultados. [11]

Capítulo 3

Modelos

En este capítulo describimos los fundamentos teóricos de los modelos que implementaremos para abordar el problema descrito, desde un punto de vista matemático. Nos centramos exclusivamente en los aspectos pertinentes a un problema de clasificación *multilabel*, en el marco introducido en el capítulo 1.

3.1. Regresión logística

En esta sección introducimos un modelo de clasificación probabilístico conocido como *regresión logística* (*logistic regression*), siguiendo la línea de exposición en [6, cap. 4].

Supongamos que queremos clasificar datos $\boldsymbol{x} \in \mathbb{R}^D$ que pertenecen a una de dos [1] clases $\mathcal{C}_1, \mathcal{C}_2$. Una manera de hacerlo es intentar estimar las distribuciones a posteriori $p(\mathcal{C}_k \mid \boldsymbol{x}),\ k = 1, 2$. Como $\mathcal{C}_1$ y $\mathcal{C}_2$ particionan el espacio muestral, tenemos que $p(\mathcal{C}_2 \mid \boldsymbol{x}) = 1 - p(\mathcal{C}_1 \mid \boldsymbol{x})$, por lo que podemos centrarnos en estimar sólo $p(\mathcal{C}_1 \mid \boldsymbol{x})$, por ejemplo. Por el teorema de Bayes tenemos que

$$
\begin{aligned}
p(\mathcal{C}_1 \mid \boldsymbol{x}) &= \frac{p(\boldsymbol{x} \mid \mathcal{C}_1)p(\mathcal{C}_1)}{p(\boldsymbol{x} \mid \mathcal{C}_1)p(\mathcal{C}_1) + p(\boldsymbol{x} \mid \mathcal{C}_2)p(\mathcal{C}_2)} \\
&= \frac{1}{1 + e^{-z(\boldsymbol{x})}} = \sigma(z(\boldsymbol{x})),
\end{aligned} \tag{3.1}
$$

donde $z(\boldsymbol{x}) = \ln \dfrac{p(\boldsymbol{x} \mid \mathcal{C}_1)p(\mathcal{C}_1)}{p(\boldsymbol{x} \mid \mathcal{C}_2)p(\mathcal{C}_2)}$ y σ es la función sigmoide (véase el apar-

[1] El modelo de regresión logística no se circunscribe únicamente a dos clases; es fácilmente extendible al caso multiclase, llamándose en la literatura en este caso regresión logística *multinomial* (véase [6, cap. 4.3.4]). Sin embargo, el modelo es más costoso de entrenar y no arroja resultados mucho mejores que usar K clasificadores binarios que indican cada uno si un ejemplo pertenece a una clase o a las $K - 1$ restantes (*One-vs-Rest logistic regression [47]*).

tado 3.2.2). Muchos modelos probabilísticos surgen de modelar las probabilidades condicionadas $p(\boldsymbol{x} \mid \mathcal{C}_k)$, así como los priores $p(\mathcal{C}_k)$, y usarlos para determinar la expresión (3.1). Si los datos de entrada $\boldsymbol{x} \in \mathbb{R}^D$ son continuos, una suposición razonable puede ser la de asumir que las funciones de densidad condicionadas de los datos de cada clase se distribuyen según una gaussiana multivariante,

$$p(\boldsymbol{x} \mid \mathcal{C}_k) = \frac{1}{(2\pi)^{D/2}|\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \mu_k)^T \Sigma_k^{-1} (\boldsymbol{x} - \mu_k)\right), \tag{3.2}$$

donde Σ_k es la matriz de covarianza [2] de los datos pertenecientes a la clase k y $\mu_k = \frac{1}{N_k} \sum_{\boldsymbol{x} \in \mathcal{C}_k} \boldsymbol{x}$ es su media. Si asumimos que ambas clases tienen la misma matriz de covarianza Σ, sustituyendo obtenemos que

$$\begin{aligned}
z(\boldsymbol{x}) &= \ln \left(\frac{p(\mathcal{C}_1) \exp\left(-\frac{1}{2}(\boldsymbol{x} - \mu_1)^T \Sigma^{-1} (\boldsymbol{x} - \mu_1)\right)}{p(\mathcal{C}_2) \exp\left(-\frac{1}{2}(\boldsymbol{x} - \mu_2)^T \Sigma^{-1} (\boldsymbol{x} - \mu_2)\right)} \right) \\
&= \ln \left(\frac{p(\mathcal{C}_1)}{p(\mathcal{C}_2)} \right) - \frac{1}{2}\Big(- \mu_1^T \Sigma^{-1} \boldsymbol{x} + \mu_2^T \Sigma^{-1} \boldsymbol{x} - \boldsymbol{x}^T \Sigma^{-1} \mu_1 + \boldsymbol{x}^T \Sigma^{-1} \mu_2 + \\
&\qquad + \mu_1^T \Sigma^{-1} \mu_1 - \mu_2^T \Sigma^{-1} \mu_2 \Big) \\
&= \ln \left(\frac{p(\mathcal{C}_1)}{p(\mathcal{C}_2)} \right) + \frac{1}{2} \mu_2^T \Sigma^{-1} \mu_2 - \frac{1}{2} \mu_1^T \Sigma^{-1} \mu_1 + \left((\mu_1 - \mu_2)^T \Sigma^{-1} \right) \boldsymbol{x}.
\end{aligned}$$

Por tanto, podemos reescribir la ecuación (3.1) como

$$p(\mathcal{C}_1 \mid \boldsymbol{x}) = \sigma(\boldsymbol{w}^T \boldsymbol{x} + w_0), \tag{3.3}$$

donde hemos definido

$$\boldsymbol{w} = \Sigma^{-1}(\mu_1 - \mu_2), \tag{3.4}$$

$$w_0 = -\frac{1}{2} \mu_1^T \Sigma^{-1} \mu_1 + \frac{1}{2} \mu_2^T \Sigma^{-1} \mu_2 + \ln \left(\frac{p(\mathcal{C}_1)}{p(\mathcal{C}_2)} \right). \tag{3.5}$$

Observamos que los términos cuadráticos en $\boldsymbol{x}$ en la función de densidad de la normal (3.2) desaparecen debido a la suposición de considerar una matriz de covarianza común a ambas clases, quedando una función *afín* de $\boldsymbol{x}$ en el argumento de la sigmoide. Podemos hacer que el argumento de la

[2] Asumimos que el argumento de la función exponencial es una forma cuadrática no degenerada (es decir, es simétrica definida positiva), por lo que la matriz es invertible. Lo más probable es que la estimación de Σ_k para un conjunto de datos de tamaño N_k produzca una matriz invertible si $N_k \gg D$ y los datos están en posición general (son linealmente independientes).

sigmoide sea *lineal* en $\boldsymbol{x}$ haciendo $\tilde{\boldsymbol{x}} = (1, \boldsymbol{x})^T, \tilde{\boldsymbol{w}} = (w_0, \boldsymbol{w})^T$, "embebiendo" así los datos en una dimensión superior.

$$p(\mathcal{C}_1 \mid \tilde{\boldsymbol{x}}) = \sigma(\tilde{\boldsymbol{w}}^T \tilde{\boldsymbol{x}})$$

La ecuación (3.3) por sí sola define un *modelo lineal generalizado* (*generalized linear model* [67]); suponiendo distintas distribuciones para $p(\boldsymbol{x} \mid \mathcal{C}_k)$ determinamos el aspecto que tienen $\boldsymbol{w}$ y w_0 y obtenemos distintos modelos.

Para construir un clasificador con uno de estos modelos, basta con especificar un valor de corte $\rho \in (0, 1)$ (*cutoff probability*) de manera que el clasificador etiqueta un dato $\boldsymbol{x_0}$ como perteneciente a la clase $\mathcal{C}_1$ si

$$p(\mathcal{C}_1 \mid \boldsymbol{x}) \geq \rho,$$

clasificándolo en $\mathcal{C}_2$ en caso contrario. Lo más habitual es escoger $\rho = 0{,}5$.

A pesar de la no linealidad de la función sigmoide, el separador de un clasificador construido así es lineal. En efecto, el conjunto de puntos en la frontera de decisión es $\{\boldsymbol{x} \in \mathbb{R}^D \mid p(\mathcal{C}_1 \mid \boldsymbol{x}) = \rho\}$, o equivalentemente, los $\boldsymbol{x}$ tal que

$$\frac{1}{1 + e^{-(\boldsymbol{w}^T \boldsymbol{x} + w_0)}} = \rho.$$

Multiplicando ambos miembros por $1 + e^{-(\boldsymbol{w}^T \boldsymbol{x} + w_0)}$, reordenando y tomando logaritmos obtenemos la ecuación de un hiperplano (véase la figura 3.1).

$$\ln(1 - \rho) = \ln(\rho) - (\boldsymbol{w}^T \boldsymbol{x} + w_0) \iff \boldsymbol{w}^T \boldsymbol{x} + w_0 + \ln\left(\frac{1 - \rho}{\rho}\right) = 0.$$

Supongamos que tenemos un conjunto de datos que hemos muestreado y queremos clasificarlos en las dos clases usando un modelo lineal de la forma (3.3) con las ecuaciones (3.4) y (3.5). Podemos estimar los parámetros del modelo usando el método de estimación de máxima verosimilitud. Si los datos tienen dimensión D, tendremos $2D$ parámetros para las medias y $D(D+1)/2$ para la matriz (común) de covarianza. Junto con el parámetro para la prior $p(\mathcal{C}_1)$, tendremos un total de $\dfrac{D(D+5)}{2} + 1$ parámetros, expresión que crece cuadráticamente en D. Esto es un ejemplo de un modelo probabilístico *generativo*, porque una vez identificados Σ, μ_k y $p(\mathcal{C}_1)$ tendremos una descripción completa de la distribución de probabilidad subyacente de los datos, y podremos generar nuevos datos (datos sintéticos) si así lo deseáramos. Otra aproximación consiste en usar directamente la forma funcional del modelo lineal generalizado,

$$p(\mathcal{C}_1 \mid \boldsymbol{x}) = \sigma(\boldsymbol{w}^T \boldsymbol{x}), \tag{3.6}$$

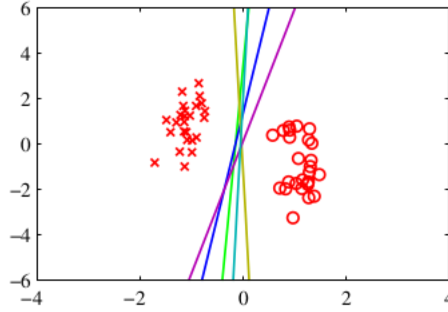


Figura 3.1: Un dataset comprendido por puntos en $\mathbb{R}^2$ pertenecientes a dos clases linealmente separables. También se muestran 5 hiperplanos separadores, de los infinitos que existen. Regresión logística, en principio, podría encontrar cualquiera de ellos. Imagen tomada de [6, cap. 10.6.2].

y tratar de estimar los pesos $\boldsymbol{w}$ convenientemente, que es una forma de entrenamiento meramente *discriminatoria*. Una ventaja de los modelos discriminatorios frente a los generativos es que típicamente habrá menos parámetros a determinar (aquí la ventaja es clara si D es grande, ya que el número de parámetros crece linealmente con D). También puede que predigan mejor las clases de los datos, al no estar suponiendo nada acerca de la forma de las distribuciones $p(\mathcal{C}_k \mid \boldsymbol{x})$.

El modelo definido por la ecuación (3.6) se conoce en el campo de la estadística como *regresión logística*, y es uno de los modelos más usados para problemas de clasificación binaria. El nombre del modelo es desafortunado, pues no es un modelo de regresión. [3] A continuación veremos cómo encontrar sus parámetros mediante el método de estimación de máxima verosimilitud.

Supongamos que el conjunto de datos $\mathcal{D} = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_N\} \subseteq \mathbb{R}^D$ está etiquetado con las etiquetas binarias $\mathcal{T} = \{t_1, \dots, t_N\} \subseteq \{0, 1\}$, donde $t_n = 0$ indica que la muestra n-ésima pertenece a la clase $\mathcal{C}_2$, y $t_n = 1$ indica que pertenece a la clase $\mathcal{C}_1$. Entonces, la función de verosimilitud es

$$p(\boldsymbol{t} \mid \boldsymbol{w}) = \prod_{n=1}^{N} p(\mathcal{C}_1 \mid \boldsymbol{x}_n)^{t_n} (1 - p(\mathcal{C}_1 \mid \boldsymbol{x}_n))^{(1-t_n)},$$

donde $\boldsymbol{t} = (t_1, \dots, t_N)^T$. El método de estimación de máxima verosimilitud define una función de error $E(\boldsymbol{w})$ a minimizar, tomando el menos

[3] El nombre proviene de una interpretación estadística en la que se argumenta que se hace regresión *de probabilidades*. [24] No obstante, el modelo se emplea con una regla de decisión, convirtiéndolo así en un clasificador.

logaritmo de esta expresión.

$$E(\boldsymbol{w}) = -\ln(p(\boldsymbol{t} \mid \boldsymbol{w})) = -\sum_{n=1}^{N} t_n \ln y_n + (1 - t_n) \ln(1 - y_n), \tag{3.7}$$

donde $y_n = \sigma(\boldsymbol{w}^T x_n)$. En el apartado 3.2.1 daremos una interpretración enriquecedora de la expresión (3.7) basada en teoría de la información.

Principalmente debido a la no linealidad de la función sigmoide, el problema de minimización (3.7) no cuenta con una solución en forma cerrada. [4] No obstante, la función $E(\boldsymbol{w})$ es estrictamente convexa y por tanto, si tiene un mínimo local, éste es un mínimo global único. [5] Podemos usar un algoritmo de optimización basado en descenso del gradiente para encontrarlo. Tenemos que, para un vector cualquiera $\boldsymbol{a} \in \mathbb{R}^D$,

$$E(\boldsymbol{w} + \boldsymbol{\varepsilon a}) = -\sum_{n=1}^{N} t_n \ln(\sigma((\boldsymbol{w} + \varepsilon \boldsymbol{a})^T x_n)) + (1 - t_n) \ln(1 - \sigma((\boldsymbol{w} + \varepsilon \boldsymbol{a})^T x_n).$$

Vamos a calcular la derivada direccional de la ecuación (3.7) en la dirección del vector $\boldsymbol{a}$. Para ello hacemos uso de la igualdad

$$\frac{\mathrm{d}}{\mathrm{d}\varepsilon} E(\boldsymbol{w} + \varepsilon \boldsymbol{a})\Big|_{\varepsilon=0} = \nabla_{\boldsymbol{a}} E(\boldsymbol{w}).$$

Entonces,

[4] En contraposición con, por ejemplo, otros métodos de clasificación, como clasificación por el método de mínimos cuadrados, o métodos de regresión como regresión lineal. [6, cap. 3.1.1]

[5] Asumiendo que el espacio de búsqueda de los parámetros $\boldsymbol{w}$ es un conjunto convexo.

$$
\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}\varepsilon}E(\boldsymbol{w}+\varepsilon\boldsymbol{a}) = &-\sum_{n=1}^{N} \frac{t_n \sigma\left((\boldsymbol{w}+\varepsilon\boldsymbol{a})^T\boldsymbol{x}_n\right)\left(1-\sigma\left((\boldsymbol{w}+\varepsilon\boldsymbol{a})^T\boldsymbol{x}_n\right)\right)\boldsymbol{a}^T x_n}{\sigma\left((\boldsymbol{w}+\varepsilon\boldsymbol{a})^T\boldsymbol{x}_n\right)} \\
&+ (1-t_n)\frac{-\sigma\left((\boldsymbol{w}+\varepsilon\boldsymbol{a})^T\boldsymbol{x}_n\right)\left(1-\sigma\left((\boldsymbol{w}+\varepsilon\boldsymbol{a})^T\boldsymbol{x}_n\right)\right)\boldsymbol{a}^T x_n}{1-\sigma\left((\boldsymbol{w}+\varepsilon\boldsymbol{a})^T\boldsymbol{x}_n\right)} \\
= &-\sum_{n=1}^{N} t_n\left(1-\sigma\left((\boldsymbol{w}+\varepsilon\boldsymbol{a})^T\boldsymbol{x}_n\right)\right)\boldsymbol{a}^T x_n \\
&+ (1-t_n)\left(-\sigma\left((\boldsymbol{w}+\varepsilon\boldsymbol{a})^T\boldsymbol{x}_n\right)\boldsymbol{a}^T x_n\right) \\
= &-\sum_{n=1}^{N} t_n\boldsymbol{a}^T\boldsymbol{x}_n - \sigma\left((\boldsymbol{w}+\varepsilon\boldsymbol{a})^T\boldsymbol{x}_n\right)\boldsymbol{a}^T x_n \\
= &\sum_{n=1}^{N}\left(\sigma\left((\boldsymbol{w}+\varepsilon\boldsymbol{a})^T\boldsymbol{x}_n\right) - t_n\right)\boldsymbol{a}^T\boldsymbol{x}_n,
\end{aligned}
$$

donde en la primera igualdad hemos hecho uso de la regla de la cadena y de la ecuación (3.17) en la página 32. Particularizando en $\varepsilon = 0$, tenemos que

$$
\nabla_{\boldsymbol{a}}E(\boldsymbol{w}) = \sum_{n=1}^{N}\left(\sigma\left(\boldsymbol{w}^T\boldsymbol{x}_n\right) - t_n\right)\boldsymbol{a}^T\boldsymbol{x}_n = \left\langle \sum_{n=1}^{N}\left(\sigma\left(\boldsymbol{w}^T\boldsymbol{x}_n\right) - t_n\right)\boldsymbol{x}_n, \boldsymbol{a}\right\rangle.
$$

Por tanto concluimos

$$
\nabla E(\boldsymbol{w}) = \sum_{n=1}^{N}\left(\sigma\left(\boldsymbol{w}^T\boldsymbol{x}_n\right) - t_n\right)\boldsymbol{x}_n = \sum_{n=1}^{N}(y_n - t_n)\boldsymbol{x}_n. \tag{3.8}
$$

Como el gradiente apunta en la dirección de máximo ascenso, un algoritmo de descenso de gradiente tratará de actualizar los pesos en la dirección de $-\nabla E(\boldsymbol{w})$. Lo que dice la ecuación (3.8) es que la dirección en la que desciende el coste $E(\boldsymbol{w})$ asociado a una muestra $\boldsymbol{x}_n$ es proporcional a $\boldsymbol{x}_n$ con factor $t_n - y_n$, que mide precisamente cómo de lejos está la salida del clasificador y_n de su verdadera etiqueta t_n.

Se pueden usar muchos algoritmos de optimización basados en descenso de gradiente para encontrar los pesos $\boldsymbol{w}$ que minimicen la función de coste. Un algoritmo eficiente muy popular para minimizar funciones de coste asociadas a modelos lineales generalizados es *iteratively reweighted least squares*

(IRLS) [3]. Nosotros usaremos LIBLINEAR [16], un optimizador que es considerado hoy en día estado del arte para regresión logística, muy eficiente para grandes conjuntos de datos *sparse*, y que además está incluido en la popular librería de aprendizaje automático `scikit-learn` [43, 7].

3.2. Redes neuronales *feedforward*

Desde un punto de vista matemático, una red neuronal prealimentada completamente conectada es un perceptrón multicapa, [64, 20] definido por una función $\boldsymbol{y}\colon \mathbb{R}^D \to \mathbb{R}^K$ que es una composición de aplicaciones

$$\mathbb{R}^{D=K_0} \xrightarrow{\boldsymbol{a}^1} \mathbb{R}^{K_1} \xrightarrow{\boldsymbol{a}^2} \mathbb{R}^{K_2} \xrightarrow{\boldsymbol{a}^3} \cdots \xrightarrow{\boldsymbol{a}^{L-1}} \mathbb{R}^{K_{L-1}} \xrightarrow{\boldsymbol{a}^L} \mathbb{R}^{K_L=K}.$$

$$\boldsymbol{y}$$

Las funciones $\boldsymbol{a}^l$ son a su vez una composición

$$\mathbb{R}^{K_{l-1}} \xrightarrow{\boldsymbol{z}^l} \mathbb{R}^{K_l} \xrightarrow{\boldsymbol{h}^l} \mathbb{R}^{K_l}$$

$$\boldsymbol{a}^l$$

donde la aplicación $\boldsymbol{z}^l$ es una aplicación afín

$$\begin{aligned} \boldsymbol{z}^l\colon \mathbb{R}^{K_{l-1}} &\to \mathbb{R}^l \\ \boldsymbol{a} &\mapsto W^l\boldsymbol{a} + \boldsymbol{b}^l, \end{aligned}$$

siendo $W^l \in \mathbb{R}^{K_l \times K_{l-1}}$ y $\boldsymbol{b}^l$ la matriz de pesos y el sesgo de la capa l-ésima de la red, respectivamente. La función $\boldsymbol{h}^l$ es una función real que actúa componente a componente y se denomina función de activación (véase el apartado 3.2.2). El número total de capas de la red es L. La función $\boldsymbol{y}$ (que llamamos f en el contexto de la figura 1.1) está determinada por las matrices $\{W^l\}_{l=1}^{l=L}$ y los sesgos $\{\boldsymbol{b}^l\}_{l=1}^{l=L}$ de cada capa, que conforman los parámetros de la red. Hay un total de $\sum_{l=1}^{L} K_l K_{l-1} + K_l$ parámetros, que listaremos en un vector $\boldsymbol{w}$ para referirnos a ellos. La salida de la red depende del valor de los parámetros y de la entrada, de manera que escribiremos $\boldsymbol{y}(\boldsymbol{x}, \boldsymbol{w})$ para denotarla. Los parámetros son "aprendidos" por un algoritmo de aprendizaje. Lo más habitual es definir una *función de coste* o error $E(\boldsymbol{y}, \hat{\boldsymbol{y}})$ que, dado un ejemplo $\boldsymbol{x}$, mida cómo de dísimiles son la salida de la red $\hat{\boldsymbol{y}} = \boldsymbol{y}(\boldsymbol{x}, \boldsymbol{w})$ y la etiqueta verdadera $\boldsymbol{y}$. El algoritmo de optimización tratará de minimizar este coste. Nos quedan entonces dos aspectos a determinar para poder entrenar una red neuronal:

1. qué función de coste definir, y

2. qué algoritmo de optimización usar.

El algoritmo de optimización que usaremos será *descenso de gradiente estocástico*, también conocido como *SGD* por sus siglas en inglés. Básicamente, lo que hace este algoritmo es calcular eficientemente el gradiente $\nabla_{\boldsymbol{w}} E$, y actualizar los parámetros $\boldsymbol{w}$ en la dirección de máximo descenso de la función de coste, que viene dado por $-\nabla_{\boldsymbol{w}} E$. Se trata de un algoritmo genérico que vale para una amplia variedad de problemas de optimización, por lo que referimos al lector a otras fuentes para más información ([20, cap. 8], [53]). Nos centramos entonces en la primera cuestión, la de cómo definir una función de coste adecuada para nuestro problema particular, dando una interpretación de la misma basada en teoría de la información.

3.2.1. Función de coste

La función de coste más usada para entrenar redes neuronales es la *entropía cruzada*. A continuación repasamos los conceptos de entropía, entropía relativa y entropía cruzada, y qué aspecto tiene la expresión de esta última para un problema de clasificación *multilabel*. Sea X una variable aleatoria discreta definida en un espacio de probabilidad Ω que toma valores en $\mathbb{R}^D$. Denotamos por $p(\boldsymbol{x})$ la probabilidad de que X tome el valor $\boldsymbol{x}$, es decir, $p(\boldsymbol{x}) = p(\{\omega \in \Omega \mid X(\omega) = \boldsymbol{x}\})$. Decimos que p es una función de masa de probabilidad de la variable aleatoria discreta X. Definimos la *entropía* de X como la esperanza de la distribución de la variable aleatoria $-\log_2 p(X)$,

$$\mathrm{H}[X] := \mathrm{E}[-\log_2 p(X)] = -\sum_{\boldsymbol{x}} p(\boldsymbol{x}) \log_2 p(\boldsymbol{x}),$$

donde en la definición ha de entenderse que si $p(\boldsymbol{x}) = 0$ entonces $p(\boldsymbol{x}) \log_2 p(\boldsymbol{x}) = 0$.[6] La entropía mide el valor esperado de la cantidad de información recibida que proporciona X ante una observación. En efecto, la función

$$\begin{aligned} h(p(\boldsymbol{x}))\colon [0,1] &\to \mathbb{R} \cup \{\infty\} \\ p(\boldsymbol{x}) &\mapsto \begin{cases} \infty & \text{si } p(\boldsymbol{x}) = 0 \\ -\log_2 p(\boldsymbol{x}) & \text{si } p(\boldsymbol{x}) \neq 0 \end{cases} \end{aligned} \tag{3.9}$$

toma valores altos o bajos ante eventos de probabilidad alta o baja, respectivamente. Podría decirse que h (función conocida como *self-information* o *surprisal* de una variable aleatoria [70]) cuantifica la "sorpresa" que sentimos al conocer que la variable aleatoria toma cierto valor. Por tanto, la

[6] Dado que $\lim_{\boldsymbol{x} \to 0^+} \boldsymbol{x} \log_2 \boldsymbol{x} = 0$.

entropía cuantifica la sorpresa media. Se mide en *shannons* o bits en el contexto de teoría de la información (cuando se usa el logaritmo en base 2); en *machine learning* se usa más comúnmente el logaritmo neperiano y se mide en *nats*. Además de la continuidad, la función h cumple otra propiedad deseable: ante variables aleatorias independientes X e Y, la información recibida al observar $(\boldsymbol{x}, \boldsymbol{y})$ es la suma de la información recibida al observar $\boldsymbol{x}$ y la información recibida al observar $\boldsymbol{y}$, es decir,[7]

$$h(p(\boldsymbol{x}, \boldsymbol{y})) = h(p(\boldsymbol{x})p(\boldsymbol{y})) = h(p(\boldsymbol{x})) + h(p(\boldsymbol{y})).$$

En nuestro problema, interpretaremos las salidas de nuestras redes neuronales desde un enfoque probabilístico. En general, el vector producido en la última capa tomará valores en $\mathbb{R}^K$, donde K es el número de clases. Usaremos la función de activación sigmoide (véase el apartado 3.2.2) para transformar este vector al cubo unidad $[0, 1]^K$, de manera que podemos interpretar nuestra salida $\boldsymbol{y}(\boldsymbol{x}, \boldsymbol{w})$ como un vector cuya entrada k-ésima es una distribución de probabilidad que mide cuán probable es que el ejemplo $\boldsymbol{x}$ pertenezca a la clase k, cuando la red neuronal está configurada con los pesos $\boldsymbol{w}$.

En general, en muchos problemas de *machine learning* desconocemos una distribución de probabilidad $p(\boldsymbol{x})$ de una variable aleatoria discreta, que aproximamos, mediante algún procedimiento, por una distribución de probabilidad $q(\boldsymbol{x})$. Se define la *entropía relativa* o *divergencia de Kullback-Leibler* entre las distribuciones $p(\boldsymbol{x})$ y $q(\boldsymbol{x})$ como

$$\begin{aligned} D_{\mathrm{KL}}(p \mid\mid q) &:= -\sum_{\boldsymbol{x}} p(\boldsymbol{x}) \log_2 q(\boldsymbol{x}) - \left(-\sum_{\boldsymbol{x}} p(\boldsymbol{x}) \log_2 p(\boldsymbol{x}) \right) \\ &= -\sum_{\boldsymbol{x}} p(\boldsymbol{x}) \log_2 \left(\frac{q(\boldsymbol{x})}{p(\boldsymbol{x})} \right). \end{aligned} \tag{3.10}$$

El siguiente teorema nos permite interpretar la divergencia de Kullback-Leibler como una medida de la disimilitud[8] entre dos distribuciones de probabilidad $p(\boldsymbol{x})$ y $q(\boldsymbol{x})$.

Teorema 3.2.1 (Desigualdad de Gibbs [68])**.** *Sea X una variable aleatoria discreta y sean p y q dos funciones de masa de probabilidad de X. Entonces $D_{\mathrm{KL}}(p \mid\mid q) \geq 0$, dándose la igualdad si y sólo si $p = q$.*

[7] Puede demostrarse que si h es una función continua que cumple esta propiedad, ha de ser de la forma $h(p(\boldsymbol{x})) = C \log_b p(\boldsymbol{x})$. Si además exigimos que $h(p(\boldsymbol{x})) \geq 0$ para toda observación $\boldsymbol{x}$, entonces $C < 0$. La definición dada en 3.9 corresponde a pedir $h(\frac{1}{2}) = 1$.

[8] Uno podría verse tentado a pensar en la divergencia de Kullback-Leibler como la "distancia" entre dos distribuciones de probabilidad. Lejos de ser una métrica, la divergencia de Kullback-Leibler ni siquiera es una función simétrica, es decir, $D_{\mathrm{KL}}(p \mid\mid q) \neq D_{\mathrm{KL}}(q \mid\mid p)$ en general. Tampoco satisface la desigualdad triangular.

Demostración ([12, cap. 2, problema 46]). Basta con demostrar que

$$\mathrm{H}_p[X] \leq -\sum_{\boldsymbol{x}} p(\boldsymbol{x}) \log_2 q(\boldsymbol{x}).$$

En la demostración usaremos la desigualdad $\ln \alpha \leq \alpha - 1$ (para ver por qué esta desigualdad es cierta, escríbase $\ln \alpha = \int_1^\alpha \beta^{-1}\, \mathrm{d}\beta < \int_1^\alpha \mathrm{d}\beta = \alpha - 1$ para $\alpha > 1$, y $\ln \alpha = -\int_\alpha^1 \beta^{-1}\, \mathrm{d}\beta < -\int_\alpha^1 \mathrm{d}\beta = \alpha - 1$ para $0 < \alpha < 1$). Tenemos que

$$\begin{aligned} -\sum_{\boldsymbol{x}} p(\boldsymbol{x}) \ln p(\boldsymbol{x}) + \sum_{\boldsymbol{x}} p(\boldsymbol{x}) \ln q(\boldsymbol{x}) &= \sum_{\boldsymbol{x}} p(\boldsymbol{x}) \ln \left(\frac{q(\boldsymbol{x})}{p(\boldsymbol{x})} \right) \\ &\leq \sum_{\boldsymbol{x}} p(\boldsymbol{x}) \left(\frac{q(\boldsymbol{x})}{p(\boldsymbol{x})} - 1 \right) = 0, \end{aligned}$$

dándose la igualdad si y sólo si $p = q$. Por tanto,

$$-\sum_{\boldsymbol{x}} p(\boldsymbol{x}) \ln p(\boldsymbol{x}) \leq -\sum_{\boldsymbol{x}} p(\boldsymbol{x}) \ln q(\boldsymbol{x}).$$

Como $\ln p(\boldsymbol{x}) = \log_2 p(\boldsymbol{x}) \ln 2$, obtenemos el resultado dividiendo la desigualdad anterior por $\ln 2$. □

Nótese que como corolario del teorema anterior, obtenemos como caso particular la desigualdad

$$\mathrm{H}_p[X] \leq \log_2 n \tag{3.11}$$

si hacemos que el soporte de X tenga tamaño finito n y hacemos que q produzca una distribución equiprobable, es decir, $q(\boldsymbol{x}_i) = \frac{1}{n}$ para todo i. Esto concuerda con nuestra intuición acerca del significado de la entropía de X. En efecto, $\mathrm{H}[X]$ es muy cercana a 0 cuando X es "casi determinista" (por ejemplo cuando, $p(\boldsymbol{x}_i)$ es 0 en todos los $\boldsymbol{x}_i$ salvo en un punto donde la probabilidad es 1), ya que la información que obtenemos cuando observamos un valor es nula. Por el contrario, cuando la distribución de p es uniforme se alcanza la cota (3.11).

En consecuencia, si queremos aproximar una distribución desconocida $p(\boldsymbol{x})$ mediante una distribución $q(\boldsymbol{x})$, trataremos de minimizar la divergencia de Kullback-Leibler. Nótese que el segundo término de la definición dada en la ecuación (3.10) es la entropía de p, que no depende de q. Por tanto podemos minimizar equivalentemente la cantidad

$$H(p,q) = \mathrm{E}_p[-\log_2 q(\boldsymbol{x})] = -\sum_{\boldsymbol{x}} p(\boldsymbol{x}) \log_2 q(\boldsymbol{x}) = D_{\mathrm{KL}}(p \,||\, q) + \mathrm{H}_p[X],$$

que se llama *entropía cruzada* (*cross-entropy*) entre las distribuciones de probabilidad p y q.

Supongamos que tenemos N ejemplos $\mathcal{D} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\} \subseteq \mathbb{R}^D$ etiquetados con etiquetas $\mathcal{T} = \{\boldsymbol{t}_1, \ldots, \boldsymbol{t}_N\} \subseteq \{0,1\}^K$ en una codificación *multi-hot*, y que nuestra red neuronal produce un vector $\boldsymbol{y}(\boldsymbol{x}, \boldsymbol{w}) \in \mathbb{R}^K$ en el que cada componente da la probabilidad de que el ejemplo esté etiquetado con una etiqueta. Si asumimos que las etiquetas son independientes, [9] la verosimilitud de que, dado un ejemplo $\boldsymbol{x}_i$ y unos pesos $\boldsymbol{w}$, la red neuronal lo etiquete con el subconjunto de las K etiquetas $\boldsymbol{t}_i$ es

$$q(\boldsymbol{t}_i \mid \boldsymbol{x}_i, \boldsymbol{w}) = \prod_{k=1}^{K} y_k(\boldsymbol{x}_i, \boldsymbol{w})^{t_{ik}} (1 - y_k(\boldsymbol{x}_i, \boldsymbol{w}))^{(1-t_{ik})}.$$

Nótese que como $t_{ik} \in \{0,1\}$, la verosimilitud $q(\boldsymbol{t}_i \mid \boldsymbol{x}_i, \boldsymbol{w})$ es una distribución de Bernoulli. Si suponemos además que los datos $(\mathcal{D}, \mathcal{T})$ son independientes, tendremos que

$$q(\mathcal{T} \mid \mathcal{D}, \boldsymbol{w}) = \prod_{n=1}^{N} q(\boldsymbol{t}_n \mid \boldsymbol{x}_n, \boldsymbol{w}) = \prod_{n=1}^{N} \prod_{k=1}^{K} y_k(\boldsymbol{x}_n, \boldsymbol{w})^{t_{nk}} (1 - y_k(\boldsymbol{x}_n, \boldsymbol{w}))^{(1-t_{nk})}.$$

Nuestro objetivo es maximizar esta expresión. Equivalentemente, podemos convertir los productos en sumas y el problema de maximización en un problema de minimización si tomamos el menos logaritmo de la función de verosimilitud, dando lugar a la función

$$E(\boldsymbol{w}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \ln y_{nk} + (1 - t_{nk}) \ln(1 - y_{nk}), \qquad (3.12)$$

donde $y_{nk} = y_k(\boldsymbol{x}_n, \boldsymbol{w})$, t_{nk} codifica si el ejemplo $\boldsymbol{x}_n$ está etiquetado con la etiqueta k-ésima, y los logaritmos los tomamos en base e por ser más común en la literatura en este contexto [20, cap 3.13]. Como podemos ver en la ecuación (3.12), el menos logaritmo de la función de verosimilitud resulta ser precisamente la entropía cruzada entre la distribución que aproxima la red neuronal y la distribución verdadera de los datos.

Decimos que $E(\boldsymbol{w})$ es nuestra función de coste, pues estima el error que estamos cometiendo al aproximar la distribución de probabilidad verdadera de los datos con la producida por nuestra red neuronal al configurarla con

[9] Esta suposición es incorrecta para nuestro problema. Por ejemplo, el 66 % de los pacientes de MIMIC que tienen hiperlipidemia (etiqueta 2724) tienen también hipertensión (etiqueta 4019). La agregación de diversos factores de riesgo de carácter cardiometabólico, relacionados con estilos de vida comportamentales, es lo que se conoce como Síndrome Metabólico [40], entre los que se encuentran la hipertesión arterial, la dislipemia aterogénica, la obesidad y la diabetes.

los pesos $\boldsymbol{w}$. Nuestro objetivo será el de ajustar los pesos $\boldsymbol{w}$ para minimizar todo lo que podamos la función de coste.

En la práctica, el cálculo de la función de coste usando directamente la expresión (3.12) es bastante costosa y numéricamente inestable. Para simplificar la notación, supongamos que el escalar t codifica una etiqueta del ejemplo $\boldsymbol{x}_i$, y que la red neuronal produce como salida el escalar $z \in \mathbb{R}$ (antes de pasarlo por la función sigmoide) en la neurona asociada a esa etiqueta. Entonces el error cometido en 3.12 para este ejemplo y esta etiqueta es

$$\begin{aligned}
-t\ln(\sigma(z)) - (1-t)\ln(1-\sigma(z)) &= -t\ln\left(\frac{1}{1+e^{-z}}\right) - (1-t)\ln\left(\frac{e^{-z}}{1+e^{-z}}\right) \\
&= t\ln(1+e^{-z}) - (1-t)(\ln(e^{-z}) - \ln(1+e^{-z})) \\
&= t\ln(1+e^{-z}) + (1-t)(z + \ln(1+e^{-z})) \\
&= (1-t)z + \ln(1+e^{-z}) \\
&= z - tz + \ln(1+e^{-z}), \qquad (3.13)
\end{aligned}$$

que es una expresión computacionalmente más eficiente pero propensa a desbordamiento de la función exponencial para $z < 0$. También podemos reformular la expresión como

$$\begin{aligned}
z - tz + \ln(1+e^{-z}) &= \ln(e^{z}) - tz + \ln(1+e^{-z}) \\
&= -tz + \ln(1+e^{z}), \qquad (3.14)
\end{aligned}$$

que ahora puede desbordar para $z > 0$. Por consiguiente, para garantizar la estabilidad y evitar el desbordamiento, las implementaciones de la entropía cruzada en los frameworks de *deep learning* [4, 62] utilizan

$$\text{máx}(z, 0) - tz + \ln(1 + e^{-|z|}),$$

que es equivalente a (3.13) o (3.14) según z sea positivo o negativo, respectivamente.

3.2.2. Funciones de activación

Dedicamos este apartado a hablar acerca de una decisión de diseño particular a las redes neuronales que normalmente no está en otros modelos paramétricos de *machine learning* que se entrenan mediante algoritmos de optimización basados en descenso del gradiente. Se trata de escoger las funciones de activación de la red neuronal, que hasta ahora hemos denotado por h, ubicándolas al final de cada capa de la red. En general, se llama función de activación a una función real cuya entrada es el resultado de una transformación afín $\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$. La función de activación opera entonces sobre el vector $\boldsymbol{z}$ componente a componente.

El diseño de funciones de activación (*activation functions* o *hidden units*) es un área de investigación muy activa y, en principio, es imposible predecir de antemano qué función de activación dará mejores resultados. [20, cap. 6.3]. En la práctica suelen probarse algunas y quedarse con las que mejores resultados producen mediante un proceso de prueba y error. No obstante, en los últimos años la comunidad se ha decantado por escoger la función ReLU y la tangente hiperbólica como las opciones "por defecto" para redes neuronales feedforward y redes neuronales recurrentes, respectivamente.

ReLU **y variantes**

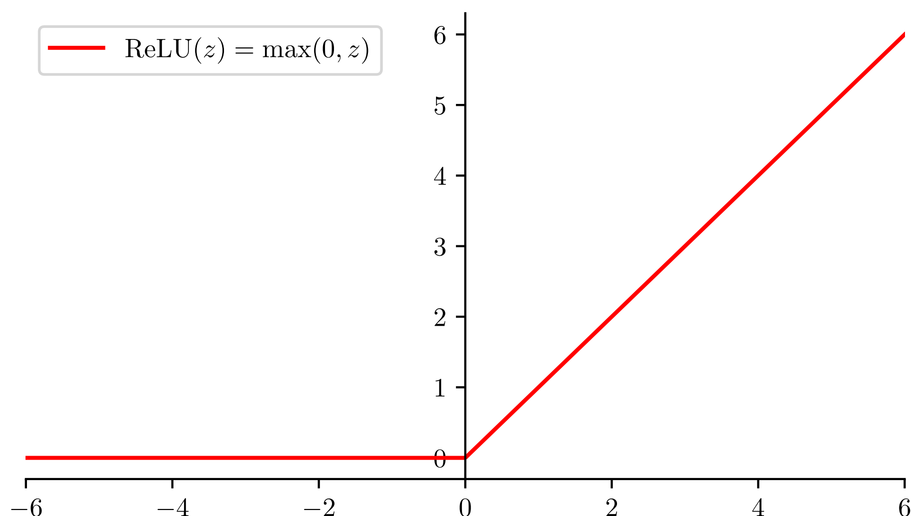


Figura 3.2: Gráfica de la función ReLU.

La función ReLU (*Rectified linear unit*) viene dada por la ecuación

$$\mathrm{ReLU}(z) = \max(0, z). \tag{3.15}$$

La función ReLU es la función de activación recomendada para la mayoría de las redes neuronales feedforward. [20, cap. 6]. Como puede verse en la figura 3.2, se trata de una transformación lineal a trozos, lo que preserva muchas de las propiedades que hacen que sea tan fácil optimizar modelos lineales usando algoritmos basados en descenso del gradiente. Esta afirmación puede parecer rara, ya que la función no es diferenciable en $z = 0$. No obstante, en la práctica esto no supone un problema: las implementaciones en las librerías de *deep learning* suelen usar el valor de las derivadas laterales para funciones de activación que no son diferenciables en un número finito de puntos. Esto se puede justificar heurísticamente si tenemos en cuenta que el

cómputo del gradiente es propenso a errores numéricos en un ordenador. Esto es, cuando en un problema necesitamos evaluar $h(0)$ para alguna función de activación h, es *extremadamente improbable* que el valor real subyacente sea verdaderamente 0. Más bien queremos evaluar $h(\varepsilon)$, para un ε pequeño que ha sido redondeado por la máquina a 0.

Según Ian Goodfellow et al. [20, cap 6.6], los dos mayores avances algorítmicos en el entrenamiento de redes neuronales han sido el uso de la entropía cruzada como función de coste en vez del error cuadrático medio y el reemplazamiento de las funciones sigmoide por funciones de rectificación. De hecho, la función ReLU se introdujo en las primeras redes neuronales [17] pero cayó en desuso a lo largo de los 80 y 90 en favor de la logística sigmoide. No fue hasta el año 2009 (Jarret et al., [27]) cuando se observó que "usar una función de rectificación es el factor más importante para mejorar el rendimiento de sistemas de reconocimiento", siendo incluso más importante que aprender los pesos de las capas ocultas. [10]

Función logística sigmoide

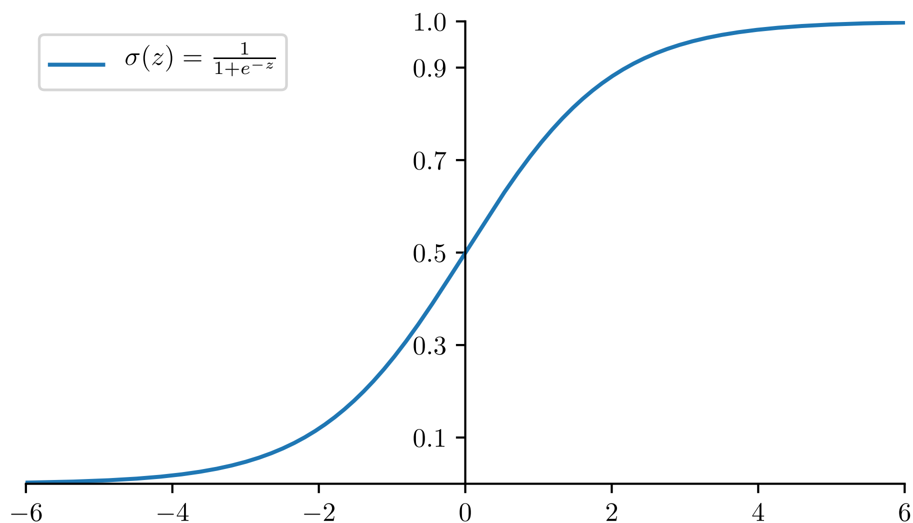


Figura 3.3: Gráfica de la función logística sigmoide. Su rango es $(0, 1)$. La función se satura en las colas cuando el argumento es muy negativo o muy positivo: se vuelve plana y no es sensible ante cambios pequeños en la entrada.

[10]Las funciones de rectificación propagan el gradiente hacia atrás mejor que las funciones que saturan, pues en estas últimas el gradiente es nulo en las zonas de saturación.

La función logística sigmoide, también llamada función logística estándar, y a la que a lo largo de esta memoria hemos llamado a veces como simplemente, función sigmoide [11], tiene por ecuación

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{e^z+1}. \tag{3.16}$$

Con anterioridad a la introducción de la función ReLU, las funciones de activación más usadas en redes neuronales eran la función logística sigmoide y la tangente hiperbólica, principalmente por creerse que su comportamiento se asemejaba más al modo en que una neurona humana se activa. Estas dos funciones están íntimamente relacionadas, [12] al darse la identidad

$$\tanh z = 2\sigma(2z) - 1.$$

La función sigmoide verifica $\sigma(-a) = 1 - \sigma(a)$. Su derivada se puede escribir convenientemente en términos de ella misma,

$$\frac{\mathrm{d}\sigma}{\mathrm{d}a} = \sigma(a)(1-\sigma(a)). \tag{3.17}$$

[11] Matemáticamente, una función sigmoide es cualquier función que tenga una forma característica de S [71]. Algunos textos [64, cap. 3] definen como *función sigmoidea* a una función continua $\sigma\colon \mathbb{R} \to \mathbb{R}$ que verifique $\lim_{z\to\infty} \sigma(z) = 1, \lim_{z\to-\infty} \sigma(z) = 0$. De este modo, la función logística estándar y la tangente hiperbólica se consideran ambas funciones sigmoides. No obstante, no es difícil encontrar literatura en el campo de *machine learning* en la que se llame a la función logística estándar simplemente como función sigmoide.

[12] De hecho, puede demostrarse que dada dada una red neuronal con funciones de activación sigmoides en las capas ocultas, existe una red neuronal equivalente (es decir, que produce la misma función) con el mismo número de capas y el mismo número de neuronas por capa tal que las funciones de activación son la tangente hiperbólica. [64, cáp. 3]

Tangente hiperbólica

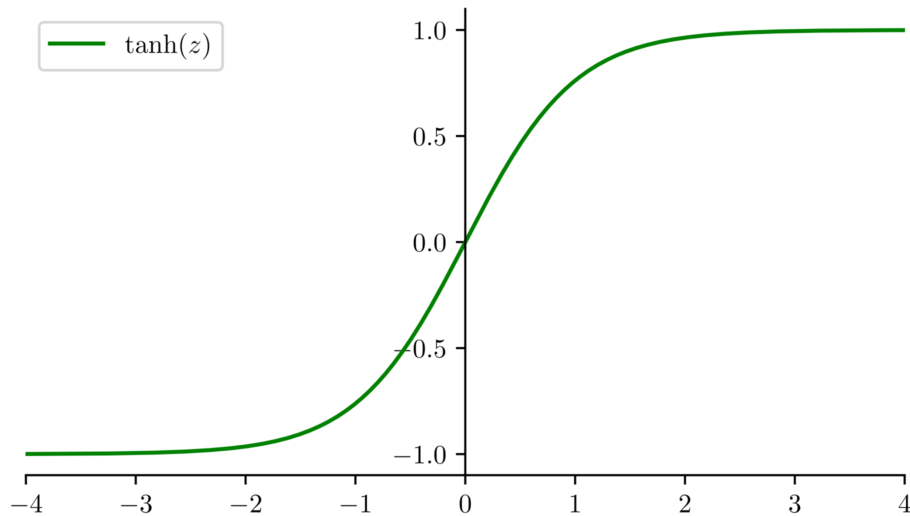


Figura 3.4: Gráfica de la tangente hiperbólica.

La función tangente hiperbólica se define como

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^{2z} - 1}{e^{2z} + 1}.$$

Tiene asíntotas horizontales cuando $z \to \pm\infty$ y la pendiente de la función en $z = 0$ vale 1, de forma que en un entorno de $z = 0$ se aproxima a la recta $y = z$.

La tangente hiperbólica es la función de activación más común para las capas ocultas de las redes neuronales recurrentes, como veremos en el apartado 3.3.

3.3. Redes neuronales recurrentes

Las redes neuronales recurrentes *(recurrent neural networks, RNN)* son una familia de redes neuronales para procesar datos secuenciales. Una de las limitaciones más evidentes de las redes neuronales *feedforward* (y también de las convolucionales) es su poca flexibilidad. [29] Más concretamente, las redes neuronales *feedforward*:

- toman como entrada un vector de tamaño fijo,
- producen como salida un vector de tamaño fijo, y

- convierten la entrada en la salida en un número fijo de operaciones (tanto el número de neuronas por capa como el número de capas no cambian).

En contraposición, las redes recurrentes permiten operar sobre secuencias de vectores: secuencias en la entrada, en la salida, o en ambas. Más aún, el número de pasos computacionales para transformar la entrada en la salida puede ser fijo, variable, o incluso un hiperparámetro a ser aprendido por el modelo. Toda esta flexibilidad permite modelar y aprender funciones fácilmente para todo tipo de problemas. [13]

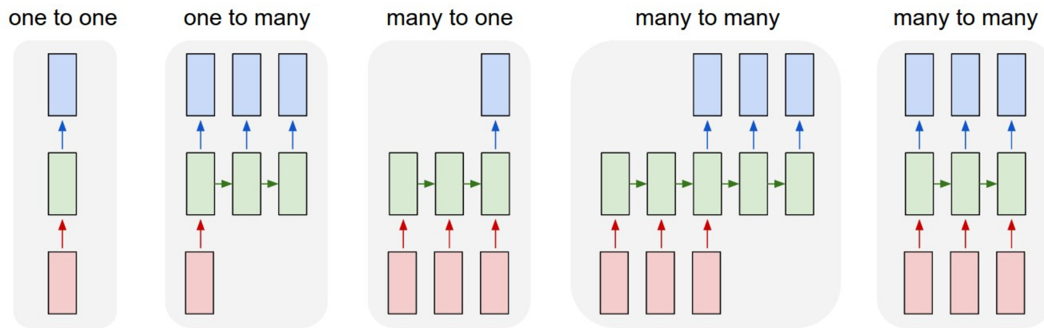


Figura 3.5: Las redes neuronales recurrentes son más flexibles en el tamaño de la entrada y la salida. Imagen tomada de [29].

Como las redes recurrentes consumen secuencias de datos, se prestan fácilmente a construir modelos en los que la temporalidad de los datos de entrada es un factor a tener en cuenta en el aprendizaje. En todo momento la red mantiene un *estado* interno conteniendo la información aprendida hasta el momento, que irá evolucionando según consume la entrada. Nosotros usaremos, al igual que en [38, 25], una red neuronal recurrente muy sencilla, conocida como red neuronal Elman. [30] Su funcionamiento está representado en la figura 3.6. La red toma como entrada, uno a uno, un conjunto de vectores $\boldsymbol{x}^{(t)}$, con $t \in 1, \ldots, \tau$ indexando la secuencia temporal. El vector de entrada se combina con el estado anterior $\boldsymbol{h}^{(t-1)}$ para producir un nuevo estado $\boldsymbol{h}^{(t)}$ y una salida $\boldsymbol{y}^{(t)}$, de la siguiente manera:

$$\boldsymbol{h}^{(t)} = \tanh\left(W_{ih}\boldsymbol{x}^{(t)} + W_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h\right) \tag{3.18}$$

$$\boldsymbol{y}^{(t)} = \tanh\left(W_{ho}\boldsymbol{h}^{(t)} + \boldsymbol{b}_{ho}\right), \tag{3.19}$$

donde W_{ih}, W_{hh} y W_{ho} son matrices y la función tangente hiperbólica se usa como función de activación, como es habitual en las redes recurrentes. (véase el apartado 3.2.2).

[13] De hecho, se conoce que las redes neuronales recurrentes son Turing completas en el sentido de que pueden simular programas arbitrarios (con los pesos apropiados). [57]

Los parámetros de la red a ser aprendidos son las matrices de pesos $W_{ih} \in \mathbb{R}^{H\times D}, W_{hh} \in \mathbb{R}^{H\times H}, W_{ho} \in \mathbb{R}^{K\times H}$ y los correspondientes sesgos $\boldsymbol{b}_h \in \mathbb{R}^H, \boldsymbol{b}_{ho} \in \mathbb{R}^K$ que definen la parte afín de las transformaciones de las ecuaciones (3.18) y (3.19). Son parámetros *compartidos* por todas las etapas de procesamiento de la secuencia de entrada. El tamaño H de la capa oculta es un hiperparámetro.

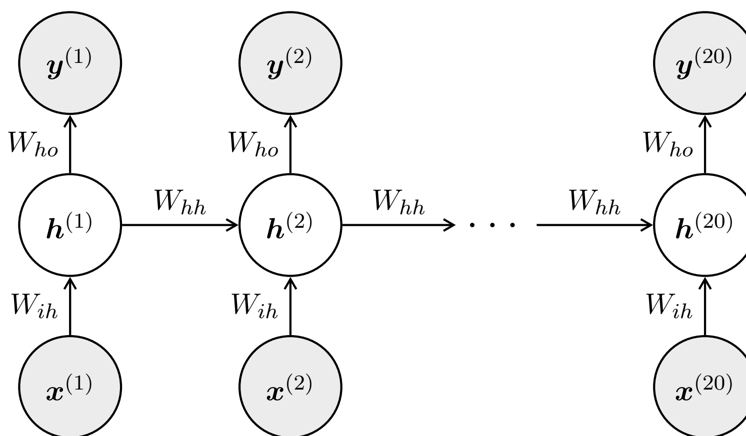


Figura 3.6: Red neuronal recurrente Elman de 20 pasos temporales.

Las redes neuronales recurrentes componen la misma función varias veces, una vez en cada paso temporal. El problema es que, empíricamente, los gradientes propagados a lo largo de varias etapas tienden a desaparecer. Si consideramos una red recurrente muy sencilla sin entrada ni función de activación, la ecuación que describe la evolución del estado es:

$$\boldsymbol{h}^{(t)} = W\boldsymbol{h}^{(t-1)}$$

Esta relación de recurrencia puede simplificarse a

$$\boldsymbol{h}^{(t)} = W^t\boldsymbol{h}^{(0)}$$

Si W es diagonalizable como $W = Q^T\Lambda Q$, siendo Q ortogonal, la ecuación anterior se escribe

$$\boldsymbol{h}^{(t)} = Q^T\Lambda^t Q\boldsymbol{h}^{(0)}$$

y el problema es más aparente: si los autovalores son todos menores que cero, elevarlos a t provocará que el resultado tienda a $\mathbf{0}$. Cualquier componente de $\boldsymbol{h}^{(0)}$ que no esté alineado con el mayor autovector de W desaparecerá eventualmente. Este problema se traslada al cálculo del gradiente de la función que computa la red, haciendo que los gradientes desaparezcan tras

varias etapas temporales y por tanto provocando que la red no pueda actualizar los parámetros durante el proceso de entrenamiento. Este problema se conoce como el problema del desvanecimiento del gradiente (*vanishing gradient problem*), y fue descubierto para las RNN por primera vez en [5]. [14] El problema de desvanecimiento del gradiente impide que las redes recurrentes sean capaces de "aprender" dependencias temporales a largo plazo de, más o menos, 10 o 20 pasos temporales.

Existen soluciones para paliar este problema. En la actualidad los modelos secuenciales más efectivos en las aplicaciones se llaman *gated RNNs*. Aquí están incluidos la red de "memoria a larga y corto plazo" (*long short-term memory, LSTM*) y la *gated recurrent unit, GRU*. Las *gated RNNs* se basan en la idea de crear "caminos" a lo largo del tiempo que tienen derivadas que no desvanecen, ni explotan. [20, cap. 10.10] Para ello modifican el grafo de la figura 3.6 y establecen una recurrencia interna que controla el estado, además de la recurrencia externa ya presente invariante en el tiempo que teníamos antes que transformaba un estado en el siguiente. El estado, la entrada y la salida se agrupan colectivamente en una "celda" de la LSTM, y las celdas están conectadas recurrentemente. La estructura de una celda se muestra en la figura 3.7. La celda cuenta adicionalmente con neuronas que actúan como "puertas", que dependiendo de su entrada, dejan fluir los valores o los cortan por una parte de la red. La puerta de entrada (*input gate*) controla si el valor de la entrada $\boldsymbol{x}^{(t)}$ entra o no en la celda. Análogamente, la puerta de "olvido" o "desaprendizaje" (*forget gate*) controla el bucle que define la recurrencia interna de la celda, y la puerta de salida controla si deja pasar la salida. Las tres puertas son sigmoidales, por lo que sus valores están en el rango $(0, 1)$. De esta manera la red tiene mecanismos para "borrar todo lo aprendido hasta el momento" o acumular la información de la entrada con lo ya aprendido. Naturalmente, todas estas interacciones dentro de la celda y entre las celdas se describen matemáticamente mediante ecuaciones que se usan en el proceso de aprendizaje. Las ecuaciones para las LSTM son varias y no son tan sencillas como las ecuaciones (3.18) y (3.19) que describen el comportamiento de las redes recurrentes normales o *vanilla*. Explicar todas las ecuaciones queda fuera del alcance de este documento; referimos al lector a las fuentes [20, cap. 10.10] y [74, 41] para un tratamiento riguroso e intuitivo de las mismas.

¿Qué partes de la arquitectura de una red LSTM son realmente importantes para aprender dependencias temporales a largo plazo? ¿Existen otras arquitecturas? Lo que hemos descrito es una red LSTM "normal", pero existen muchas variantes y el modelo de la figura 3.7 puede complicarse tanto como uno quiera. Una variación de la LSTM de reciente popularidad

[14] El problema de desvanecimiento del gradiente no sucede tan a menudo en las redes neuronales normales *profundas*. Esto es porque en este caso, las matrices de pesos en cada capa *son distintas*. Una correcta inicialización de los pesos puede emplearse para que el producto de todas ellas tenga una determinada distribución. Véase [20, cap. 10.7].

es la *gated recurrent unit (GRU)*, que es un modelo simplificado de una red LSTM. El cambio más importante es que las puertas de *input* y *forget* están ahora acopladas: cuando la puerta de olvido se abre también lo hace la de entrada. Por tanto, cuando el estado interno se borra la información de la entrada se usa para calcular uno nuevo inmediatamente. Solamente por el mero interés de saber si es comparable el rendimiento de las LSTM y las GRU para nuestro problema, implementamos y probamos ambas redes (véase el capítulo 4).

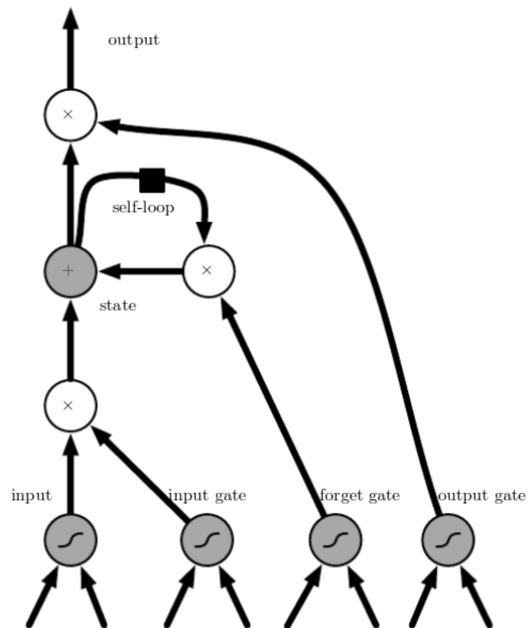


Figura 3.7: Diagrama de un bloques con una de las celdas recurrentes de una red LSTM. Imagen tomada de [20, cap.10.10].

Capítulo 4

Implementación y metodología práctica

A pesar de la gran cantidad de teoría y trabajo relacionado que se ha investigado para abordar el problema planteado, la mayor parte del trabajo ha sido de carácter práctico. En este capítulo describimos la implementación del proyecto que acompaña a este documento, que se puede encontrar en el siguiente repositorio de código.

`https://github.com/david-perez/tfg-code`

Asimismo, detallamos cómo hemos usado las técnicas de preprocesamiento y los modelos, expuestos en el apartado 2.2 y el capítulo 3 a nivel teórico, respectivamente.

La tabla 4.1 resume brevemente la función principal de los directorios y los archivos más importantes que componen el proyecto. Usamos Python 3 por la gran cantidad de librerías de *machine learning* disponibles en este lenguaje. Las dependencias del proyecto se encuentran en el archivo `requirements.txt`, para ser instaladas fácilmente mediante un gestor como pip. [46] [1] El archivo `readme.md` contiene instrucciones de instalación y uso del proyecto.

[1] Debido a la gran cantidad de dependencias del proyecto, recomendamos hacer uso de alguna herramienta de gestión de entornos como virtualenv [65] o conda [10].

Archivo / Directorio	Funcionalidad
sql_setup/	Contiene los archivos de creación de tablas y vistas materializadas para montar la base de datos que utiliza el proyecto.
explore_dataset/	Contiene archivos de exploración del dataset y genera estadísticos descriptivos, como los expuestos en el apartado 2.1.
logs/	Los logs, además de mostrarse por consola, se serializan en esta carpeta.
tensorboard_logs/	Logs generados por la herramienta de visualización TensorBoard.
readme.md	Instrucciones de instalación y uso del proyecto.
requirements.txt	Listado de dependencias para ser instaladas con un gestor de dependencias.
database.ini	Archivo de configuración de la conexión a la base de datos.
database_manager.py	Clase que implementa una API para interactuar con la base de datos. Todas las consultas y modificaciones de las tablas realizadas por todos los archivos del proyecto se centralizan aquí. Usamos la librería psycopg2 para interactuar con PostgreSQL. [48]
logging_utils.py	Métodos comunes para logging.
spacy_analyzer.py	Analizador de spaCy personalizado para tokenizar textos.
vocabulary_generator.py	Generador de vocabulario.
bag_of_words_generator.py	Generador de vectores *bag of words*.
logistic_regression.py	Entrenamiento de clasificadores basados en regresión logística.
feed_forward_nn.py	Entrenamiento de clasificadores basados en redes neuronales *feedforward*.
rnn.py	Entrenamiento de clasificadores basados en modelos recurrentes.
rnn_model.py	Modelo recurrente personalizable para definir redes neuronales recurrentes básicas, LSTMs y GRUs.
evaluate_classifier.py	Evaluador de clasificadores. Calcula métricas.

Tabla 4.1: Funcionalidad de los archivos y directorios más importantes que componen el proyecto.

A continuación detallamos cómo hemos implementado las principales fases para construir un clasificador, que son:

- generación del vocabulario,
- generación de los *bag of words*,
- entrenamiento de los modelos, y
- evaluación de los clasificadores,

que habíamos representado en las figuras 1.1 y 1.2 de las páginas 6 y 7, respectivamente.

4.1. Generación de vocabulario

La generación del vocabulario se realiza ejecutando `vocabulary_generator.py`, que se conecta a la base de datos y procesa el corpus lingüístico asociado a las notas médicas de los pacientes del *training set.* [2] Para ello el script hace uso de la librería spaCy, [60, 23] una librería open-source para procesamiento de lenguaje natural que cuenta con herramientas útiles como tokenizadores, taggers y parsers (entre otras) [3] que se pueden encadenar para producir, a partir de textos, objetos llamados *documentos* (véase la figura figura 4.1). Nosotros construimos un analizador de spaCy personalizado (`spacy_analyzer.py`) para segmentar el texto en tokens. Tras la segmentación, realizamos en orden las siguientes tareas:

1. eliminamos las palabras vacías (*stop words*),
2. eliminamos los tokens con un único carácter,
3. eliminamos los tokens que contengan algún caracter númerico,
4. eliminamos separadores en blanco y,
5. eliminamos los tokens que contengan algún signo de puntuación.

A continuación construimos una matriz término-documento, en donde la entrada (i, j) denota la frecuencia de aparición del término j en el documento i. Usando esta matriz, descartamos todos los términos que no aparezcan en al menos 3 documentos, y calculamos las medidas tf-idf (véase el apartado 2.2) de cada término en cada documento, construyendo así otra matriz. Sumamos

[2] Es importante el hecho de que el vocabulario se genera únicamente con las notas médicas del *training set*, pues queremos evaluar el rendimiento de los clasificadores sobre el *test set* cómo si las notas del *test set* fueran *completamente nuevas*, quizá provenientes de otro dataset distinto a MIMIC.

[3] Algunas de estas herramientas han sido desarrolladas usando, a su vez, modelos basados en *deep learning.* spaCy cuenta con numerosos modelos, que pueden consultarse en [36]. Los hay entrenados con corpus en distintos idiomas y provenientes de varias fuentes (artículos, blogs, comentarios...). Nosotros usamos el modelo por defecto en inglés, `en_core_web_sm`.

las columnas de esta matriz, obteniendo así un vector con la suma de los valores tf-idf para cada término, que nos indica, en cierta medida, cómo de relevante es un término en todo el corpus (véase la ecuación (2.4) en la página 14). El vocabulario final se selecciona como los primeros 40000 términos tras ordenarlo descendientemente. Este vocabulario se serializa en formato JSON en una tabla nueva de la base de datos.

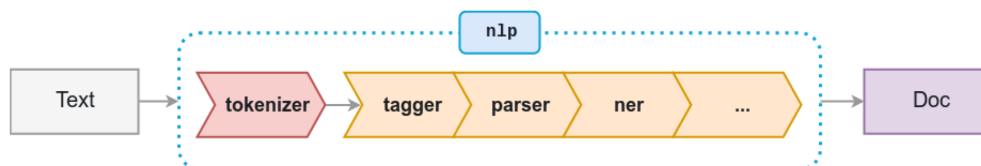


Figura 4.1: Pipeline de procesamiento por defecto de spaCy. Imagen tomada de [31].

4.2. Generación de vectores *bag of words*

Respecto a la generación de los *bag of words*, ésta se lleva a cabo en `bag_of_words_generator.py`. El script toma como entrada un vocabulario y produce como salida un conjunto de vectores *bag of words* para cada paciente (o para cada nota médica de cada paciente, en el caso de las RNN). Los vectores están asociados exclusivamente a uno de los tres tipos de datasets: *training set*, *validation set* o *test set*, indicándose el dataset asociado como argumento de entrada. Para los modelos de regresión logística y redes neuronales *feedforward*, el script construye, para cada paciente, un vector con la frecuencia de aparición de cada término del vocabulario en *todas sus notas*. Este vector se promedia por el número de notas médicas escritas para este paciente. Para los modelos basados en redes recurrentes, el script calcula, para cada paciente, un conjunto de vectores con la frecuencia de aparición de cada término del vocabulario en *cada una de sus notas*. En ambos casos, los vectores de salida se serializan en formato binario (mediante la librería pickle [1]) en una tabla nueva de la base de datos. Es importante la forma en la que se almacenan estos datos, pues cada vector tiene D entradas con números enteros, donde D es la longitud del vocabulario. En total habría que almacenar DN números para un determinado corpus, siendo N el número de documentos, lo cual es inviable para un vocabulario de 40000 palabras. Afortunadamente, la mayoría de las entradas de los vectores son nulas, dando lugar a una representación *sparse* (dispersa, poco densa). Para almacenar vectores y matrices *sparse*, sólo hace falta guardar las entradas no nulas junto con los índices que describen la entrada. El formato en el que esta información se almacena en memoria es de crucial importancia, pues la disposición en memoria hará más o menos eficientes cálculos como la

suma de vectores o la multipicación de una matriz por un vector. Nosotros usaremos un formato muy sencillo para guardar matrices, llamado formato COO (*coordinate format*, [56]). Consiste en almacenar la tripleta de arrays (`row`, `col`, `data`), tal que `data[i]` es el dato almacenado en la entrada de la matriz que tiene por índices (`row[i]`, `col[i]`). Sólo usamos este formato para almacenar los vectores en la base de datos y para construir las matrices término-documento en la generación de vocabulario. Cuando entrenamos nuestros modelos, cargamos todos los vectores a memoria y los convertimos al formato habitual (almacenando explícitamente los ceros de las entradas nulas también) para mejorar la eficiencia de los cálculos. [4] Si no caben todos en memoria (ya sea RAM o la memoria de la GPU), como es el caso para los vectores de las RNN, cargamos los vectores en *batches* (bloques) y entrenamos los modelos iterando por los *batches*.

4.3. Diseño y evaluación de los clasificadores

4.3.1. Regresión logística

Para el modelo de regresión logística, entrenamos diez clasificadores (uno por cada enfermedad de las 10 enfermedades más comunes en MIMIC, tal y como se indica en el apartado 2.1). La información relativa al algoritmo de entrenamiento puede consultarse en la documentación de la librería que hemos usado ([33, 58]). Cada clasificador toma como entrada un vector *bag of words* asociado al historial médico de un paciente y predice una probabilidad que indica cuán probable es que ese paciente padezca la enfermedad. Asignamos a un paciente el código ICD-9 de la enfermedad si la probabilidad es mayor que 0.5 (*cutoff probability*). Esto está implementado construyendo dos matrices: una matriz X_{train} de dimensiones $n \times 40000$ y una matriz Y_{train} de dimensiones $n \times 10$, donde n es el número de pacientes del *training set*. [5] La matriz X_{train} tiene por filas los vectores *bag of words* asociados a cada paciente, y la matriz Y_{train} es una matriz binaria cuya entrada (i, j) denota si el paciente i está diagnosticado según MIMIC con la enfermedad j. Una vez entrenado el clasificador, evaluamos su eficiencia pasándole una matriz X_{test}. El clasificador produce una matriz $\hat{Y}_{\text{test}}$, que comparamos con la matriz verdadera de etiquetas Y_{test}.

[4] Existen frameworks de *deep learning* que comienzan a dar soporte experimental para cálculos con matrices y vectores almacenados en algún formato *sparse*. Veáse por ejemplo [63].

[5] En el modelo de regresión logística, no usamos el *validation set*, pues no buscamos mejorar la eficiencia del modelo modificando los hiperparámetros del algoritmo de optimización. Los hiperparámetros son fijos; usamos los de por defecto de la librería sklearn.

4.3.2. Red neuronal *feedforward*

Para los modelos basados en redes neuronales usamos PyTorch [51, 42], un potente framework de *deep learning* que se caracteriza por su flexibilidad, lo que otorga a los investigadores una gran facilidad para experimentar con distintas arquitecturas. [15]

La red neuronal feedforward está implementada en `feed_forward_nn.py`. El script toma como entrada el nombre de tres tablas de la base de datos (las producidas por el generador de *bag of words*) que contienen los *bag of words* asociados al *training set*, *validation set* y *test set*. Lo primero que hace es cargar el contenido de los vectores en memoria y definir la arquitectura de una red, que se puede modificar muy fácilmente (véase el fragmento de código 4.1). En los experimentos, probamos con una multitud de redes, con de 2 a 4 capas ocultas y de 100 a 1000 neuronas en cada capa oculta. Usamos funciones de activación ReLU en las capas ocultas y la función sigmoide en la última capa, que tiene 10 neuronas (véase el apartado 3.2.2). Cada neurona de la última capa está asociada a una enfermedad. Asignamos un vector de entrada $\boldsymbol{x} \in \mathbb{R}^{40000}$ a una enfermedad si la probabilidad predicha por su neurona es mayor que 0.5. Los vectores de entrada son los *bag of words* asociados a cada paciente *sin promediar* por el número de notas médicas, tal y como se hace en [38]. La función de coste a optimizar es la entropía cruzada y el algoritmo de optimización es descenso de gradiente estocástico. Algunos hiperparámetros asociados son la tasa de aprendizaje (*learning rate*) y el decaimiento de los pesos (*weight decay*), con los que experimentamos dentro de un umbral. Para el cálculo eficiente de los gradientes, PyTorch implementa el algoritmo de retropropagación [53] sobre el grafo computacional que define la red.

Frágmento de código 4.1: Definición de una red neuronal *feedforward* de 3 capas ocultas con el paquete `Sequential` de PyTorch. Las capas tienen `H1`, `H2` y `H3` neuronas cada una, respectivamente. La red consume una entrada de dimensión `D_in` y produce una salida de dimensión `D_out`.

```
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H1),
    torch.nn.ReLU(),
    torch.nn.Linear(H1, H2),
    torch.nn.ReLU(),
    torch.nn.Linear(H2, H3),
    torch.nn.ReLU(),
    torch.nn.Linear(H3, D_out),
)
```

4.3.3. Redes neuronales recurrentes

En cuanto a las redes recurrentes, implementamos una red neuronal recurrente Elman tal y como se describe en el apartado 3.3. La definición de la red se realiza en el archivo `rnn_model.py`, que permite definir la red como una red recurrente *vanilla* (según las ecuaciones (3.18) y (3.19)), una red LSTM o una red GRU. Experimentamos también con el número de neuronas en la capa oculta. La definición de la red se importa en `rnn.py`, que funciona análogamente al script `feed_forward_nn.py`. Toma como entrada las tablas asociadas al *training set*, *validation set* y *test set*, que contiene los *bag of words* de cada nota médica de cada paciente. A diferencia del resto de modelos, en las RNN no podemos cargar todos los vectores en memoria debido a la gran cantidad de datos. Lo que hacemos es dividir el dataset en bloques (*chunks*) y entrenamos cada bloque iterativamente un número fijo de *epochs*. Sería más correcto entrenar la red en cada *epoch* con todos los *chunks* para evitar que el modelo memorice los pesos de alguna parte del dataset en particular y por ende se produzca un posible *overfitting*. Este comportamiento se observa durante el entrenamiento de la red aunque creemos que no influye en los resultados.

El orden en el que una red recurrente procesa los vectores de entrada es muy importante. En nuestro caso, lo que haremos será procesar las notas médicas en el orden en el que fueron escritas por el personal sanitario. De esta manera la red es capaz de aprender los síntomas del paciente, así como los resultados de las pruebas, en el mismo orden en el que lo hicieron los médicos antes de llegar a un diagnóstico.

En lo que sigue, referimos al lector a la figura 3.6 de la página 35. Los vectores $\boldsymbol{x}^{(t)}$ son, como antes, *bag of words* de notas médicas, y por tanto están en $\mathbb{R}^D$, donde D es la longitud del vocabulario, es decir, 40000. No obstante, esta vez cada paciente no tendrá asignado un único *bag of words* con la suma de las frecuencias de las palabras en todas sus notas médicas, sino que tendrá asignado un conjunto de vectores $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}$, donde cada $\boldsymbol{x}^{(t)}$ es un *bag of words* de una única nota médica, y las notas han sido ordenadas cronológicamente. Como detallamos en el apartado 2.2, tenemos a lo sumo 20 notas médicas por cada paciente en nuestros datasets de entrenamiento y de test (es decir, $\tau = 20$). Si un paciente tiene menos de 20 notas, rellenamos el resto de vectores con ceros (*padding*).

La salida de la red, los vectores $\boldsymbol{y}^{(t)}$, son, como antes, probabilidades no normalizadas o *logits* de pertenencia a las clases de cada enfermedad, y por tanto están en $\mathbb{R}^K$. Los vectores $\boldsymbol{h}^{(t)}$ componen el estado de la red recurrente, y pertenecen a $\mathbb{R}^H$. El estado $\boldsymbol{h}^{(t)}$ depende de todos los estados y entradas anteriores, y representa por tanto de alguna manera lo que la red ha aprendido tras consumir las t primeras notas médicas. En todo instante de tiempo la red produce un vector $\boldsymbol{y}^{(t)}$ que puede interpretarse como un "diagnóstico" o asignación de códigos ICD-9 *preliminar*. Estas salidas par-

ciales pueden usarse para observar cómo evoluciona la salida a medida que se va consumiendo la entrada, o simplemente para depurar la implementación de la red. Nosotros sólo usaremos la salida correspondiente al último elemento de la secuencia, $\boldsymbol{y}^{(\tau)}$, pasándola por la función sigmoide para obtener probabilidades. Como en el resto de casos, el clasificador final lo construimos asignando la etiqueta (enfermedad) k a toda la secuencia de entrada (es decir, al paciente) si la entrada k-ésima del vector resultante tiene una probabilidad mayor que 0.5.

4.4. Otros desarrollos

Dedicamos este apartado a comentar algunas herramientas útiles usadas transversalmente en todo el proyecto que hemos desarrollado para acelerar el ritmo de experimentación.

En primer lugar, todos los scripts ejecutables por consola poseen una interfaz que describe los argumentos y *flags* que el programa toma como entrada. Por ejemplo, algunos scripts como `bag_of_words_generator.py` (véase la figura 4.2) toman un flag `--toy_set`. Una ejecución de un programa con este flag hace que se procesen por defecto solamente 700 notas del dataset indicado. Esto es útil cuando se hacen muchos cambios en una implementación, pues permite ejecutar el programa por completo muy rápidamente y comprobar si se produce alguna excepción u otro comportamiento inesperado. Otro flag importante con el que cuentan los modelos de *deep learning* es `--no_gpu`. Por defecto, los modelos se ejecutan en la GPU del sistema, ya que las capacidades de procesamiento en paralelo de grandes cantidades de datos es mucho más rápida en las tarjetas gráficas. [28] PyTorch cuenta con implementaciones de operaciones sobre tensores muy rápidas escritas en CUDA para ello, y los tensores se almacenan en regiones de memoria de la GPU. No obstante, sólo GPUs modernas (con soporte para versiones actualizadas del runtime CUDA) pueden funcionar con PyTorch. La inclusión de este flag ejecuta todo el modelo en la CPU del sistema.

```
File  Edit  View  Search  Terminal  Help
 david@aerios  ~/code/tfg-code-clean/tfg-code   master ?  python bag_of_word
s_generator.py --help
usage: bag_of_words_generator.py [-h] [--toy_set [TOY_SET]] [--test_set]
                                 [--validation_set] [--top100_labels]
                                 [--for_rnn]
                                 vocabulary_experiment

Generate bag of words vectors for each patient using the patient's medical
notes and a provided vocabulary. The bag of words are stored in a database
table.

positional arguments:
  vocabulary_experiment
                        the experiment id from the row in the database where
                        the vocabulary to be used is stored

optional arguments:
  -h, --help            show this help message and exit
  --toy_set [TOY_SET]   how many rows to fetch from the corpus table
  --test_set            fetch the notes from the test table
  --validation_set      fetch the notes from the validation table
  --top100_labels
  --for_rnn
 david@aerios  ~/code/tfg-code-clean/tfg-code   master ? 
```

Figura 4.2: Interfaz por consola de uno de los scripts del proyecto.

En segundo lugar, cada vez que ejecutamos un programa, el log, además de mostrarse por la salida estándar, se serializa en un archivo de texto que se guarda en el directorio `logs/`. Esto facilita la depuración de los programas, pues los modelos tardan mucho en ejecutarse y permite que su ejecución se pueda examinar posteriormente. También permite comparar los resultados con otras versiones del mismo modelo o con implementaciones idénticas ejecutadas en distintas máquinas.

```
davidp00@holstein: ~
File  Edit  View  Search  Terminal  Help
2018-06-07 19:56:35,815 rnn          INFO     [train]  F1-score = 0.29725, Precision = 0.48264
, Recall = 0.24256, Subset-accuracy = 0.06784, Jaccard index = 0.22466
2018-06-07 19:56:35,821 rnn          INFO     [val]  F1-score = 0.24254, Precision = 0.42735,
Recall = 0.19152, Subset-accuracy = 0.05528, Jaccard index = 0.18214
2018-06-07 19:56:35,958 rnn          INFO     Chunk= 17/20, Epoch = 3/3, Loss train = 0.60907,
 Loss val = 0.62779
2018-06-07 19:56:35,966 rnn          INFO     [train]  F1-score = 0.29599, Precision = 0.48610
, Recall = 0.23994, Subset-accuracy = 0.06658, Jaccard index = 0.22311
2018-06-07 19:56:35,972 rnn          INFO     [val]  F1-score = 0.24166, Precision = 0.42504,
Recall = 0.19089, Subset-accuracy = 0.05528, Jaccard index = 0.18160
2018-06-07 19:56:35,974 rnn          INFO     Building train tensors...
2018-06-07 19:56:35,975 rnn          INFO     GPU detected: TITAN X (Pascal)
2018-06-07 19:56:35,976 rnn          INFO     Running on GPU
2018-06-07 19:57:55,987 rnn          INFO     [bw_train_top10_rnn_4]   Patients: 796, Features
: 40000
2018-06-07 19:57:55,987 rnn          INFO     [bw_train_top10_rnn_4]   Bag of words vectors lo
aded
2018-06-07 19:57:59,252 rnn          INFO     [bw_train_top10_rnn_4]   X tensor built
2018-06-07 19:57:59,253 rnn          INFO     [bw_train_top10_rnn_4]   Y tensor built
2018-06-07 19:57:59,253 rnn          INFO     train tensors built
2018-06-07 19:57:59,302 rnn          INFO     Building validation tensors...
2018-06-07 19:57:59,302 rnn          INFO     GPU detected: TITAN X (Pascal)
2018-06-07 19:57:59,303 rnn          INFO     Running on GPU
2018-06-07 19:58:21,784 rnn          INFO     [bw_val_top10_rnn_5]   Patients: 398, Features:
40000
2018-06-07 19:58:21,785 rnn          INFO     [bw_val_top10_rnn_5]   Bag of words vectors load
ed
2018-06-07 19:58:23,414 rnn          INFO     [bw_val_top10_rnn_5]   X tensor built
2018-06-07 19:58:23,414 rnn          INFO     [bw_val_top10_rnn_5]   Y tensor built
2018-06-07 19:58:23,414 rnn          INFO     validation tensors built
```

Figura 4.3: Log mostrando el proceso de entrenamiento de un clasificador.

Asimismo, cada vez que se ejecuta un programa se crea un "experimento" en la base de datos asociado a la ejecución del programa. Un experimento guarda información útil como los timestamps de inicio y fin de ejecución, el nombre del archivo del log o la configuración con la que se ha instanciado un modelo en el caso de estar entrenando un clasificador. Esto es útil para comprobar que los modelos acaban correctamente y tardan un tiempo razonable en ejecutarse. Los resultados de ejecución finales para todos los datasets también se almacenan en la tabla para su posterior análisis. Al correr muchos experimentos, esto permite consultar con facilidad cuáles han obtenido mejores resultados e inferir conclusiones observando los hiperparámetros que los definen. Un experimento está unívocamente determinado por un identificador `experiment_id` que se asigna cuando el experimento se crea. Estos identificadores permiten "rastrear" cómo se construye un clasificador que depende de muchos experimentos. Por ejemplo, la ejecución de una red neuronal es un experimento, que toma como entrada vectores generados por otro experimento, que a su vez depende de un experimento de generación de vocabulario. Podemos experimentar en cada una de estas etapas, aislando un único parámetro de configuración de uno de estos experimentos y observando cómo cambios en el parámetro influyen en los resultados finales.

Finalmente, para entrenar los modelos neuronales hemos usado mucho TensorBoard, una herramienta de visualización de metricas relacionadas con el proceso de entrenamiento. [61] Esta herramiente permite ver, en tiempo real, los valores que va tomando la función de coste mientras el modelo va aprendiendo, así como cualquier otro estadístico que se compute mientras se entrena la red neuronal. Los gráficos interactivos de la interfaz web facilitan la experimentación, al ser posible comparar métricas de distintas series en el tiempo, incluso provenientes de modelos distintos (véase la figura 4.4). TensorBoard es una herramienta diseñada para ser usada por TensorFlow, un framework de *deep learning* desarrollado por Google. [2] PyTorch no dispone aún de un sistema de visualización parecido, por lo que usamos una librería *open source* para convertir los datos generados por nuestros programas al formato que entiende TensorBoard, antes de mandarlos al servidor de TensorBoard para que los procese. [32]

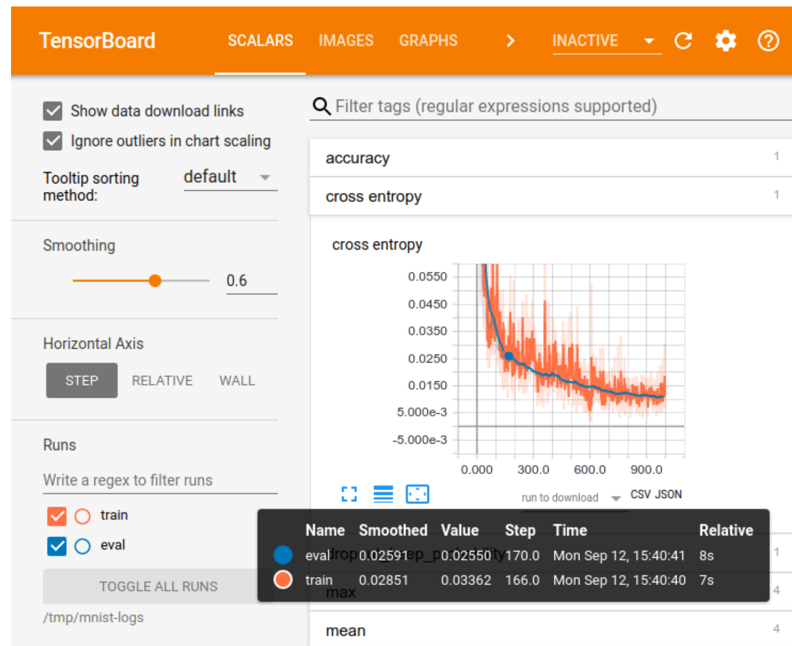


Figura 4.4: Captura de pantalla de la interfaz web de Tensorboard. Imagen tomada de [61].

4.5. Entorno de ejecución

Todos los modelos, salvo los basados en redes neuronales recurrentes, fueron ejecutados en dos máquinas distintas: un ordenador de sobremesa de alta gama y un servidor con mayor memoria y capacidad de procesamiento. El primero ejecutó todo en la CPU, mientras que el servidor cuenta con una tarjeta gráfica de altas prestaciones donada por el *GPU Grant Program* de NVIDIA, [21] conseguida gracias al trabajo [22]. Las redes neuronales recurrentes se ejecutaron únicamente en el servidor, debido a la mayor memoria y el mayor tiempo de ejecución que requieren. El modelo de regresión logística se ejecutó únicamente en la CPU de ambas máquinas. La principal razón por la cual hemos ejecutado el código en dos máquinas es comprobar que se obtienen resultados similares, [6] y comprobar la diferencia de rendimiento de los modelos en una GPU y una CPU.

[6]Las resultados en ambas máquinas no son *idénticos* debido a la intrínseca aleatoriedad del algoritmo de descenso de gradiente estocástico, que ordena los datos de entrenamiento aleatoriamente en cada *epoch* antes de pasárselos a la red neuronal.

Ordenador de sobremesa	**Servidor**
▪ CPU: Intel Core i7 4770k. ▪ Memoria: 8 GiB DDR3 1866 MHz. ▪ Almacenamiento: Disco duro 1 TiB 7200 RPM. ▪ Sistema operativo: Ubuntu 17.10 (Linux kernel 4.13).	▪ CPU: Intel Core i7 920. ▪ Memoria: 16 GiB. ▪ GPU: NVIDIA Titan X (arquitectura Pascal), 3584 CUDA cores, 12 GiB. ▪ Almacenamiento: Discos duros, 2.5 TiB. ▪ Sistema operativo: Debian GNU/Linux 9.4 (Linux kernel 4.9).

Tabla 4.2: Prestaciones de las dos máquinas en las que se han ejecutado los programas.

En cuanto a la base de datos en la que se importó MIMIC, usamos dos servidores PostgreSQL [49], uno instalado en cada máquina. Los servidores se instalaron mediante Docker, [13] una herramienta para realizar virtualización a nivel de sistema operativo que crea "contenedores" en donde se ejecutan las aplicaciones aisladamente. [73] La comunidad de investigadores y desarrolladores en torno a MIMIC cuenta con un repositorio de código, `mimic-code`, [35] en donde los contribuyentes comparten código útil para instalar, administrar y realizar experimentos con la base de datos. El repositorio cuenta con una imagen de Docker para PostgreSQL, junto con scripts de instalación y manejo de la base de datos MIMIC, que se han utilizado como punto de partida para montar la infraestructura de este proyecto.

Capítulo 5

Resultados

5.1. Métricas

Nosotros usaremos las siguientes *sample-based metrics*, que calculan métricas tradicionales en problemas de clasificación *por cada muestra* y promedian los resultados [75, 59]:

$$\text{Precision}(h) = \frac{1}{n}\sum_{i=1}^{n} \frac{\mid Y_i \cap h(x_i) \mid}{\mid h(x_i) \mid} \qquad \text{Recall}(h) = \frac{1}{n}\sum_{i=1}^{n} \frac{\mid Y_i \cap h(x_i) \mid}{\mid Y_i \mid}$$

$$F_1(h) = \frac{1}{n}\sum_{i=1}^{n} \frac{2 \mid Y_i \cap h(x_i) \mid}{\mid Y_i \mid + \mid h(x_i) \mid} = \frac{1}{n}\sum_{i=1}^{n} \frac{2\frac{|Y_i \cap h(x_i)|}{|h(x_i)|}\frac{|Y_i \cap h(x_i)|}{|Y_i|}}{\frac{|Y_i \cap h(x_i)|}{|h(x_i)|} + \frac{|Y_i \cap h(x_i)|}{|Y_i|}}$$

donde n = número de muestras

$h(x_i)$ = conjunto de etiquetas predichas por el clasificador h para la muestra x_i

Y_i = conjunto de etiquetas verdaderas asociadas a la muestra x_i.

Nótese que en el caso de que sea $h(x_i) = \emptyset$ o $Y_i = \emptyset$ algunas de las métricas no están definidas. En tal caso se define la métrica correspondiente asociada a la muestra x_i como 0. Nótese también que $F_1(h) \neq \frac{2\,\text{Precision}(h)\text{Recall}(h)}{\text{Precision}(h)+\text{Recall}(h)}$ como sucedería si hiciéramos una *micro-average* o si estuvieramos ante un problema de clasificación binaria. En efecto, aquí la métrica $F_1(h)$ es la *media harmónica* entre la precisión y el recall del clasificador *por cada muestra*, promediando los resultados, en consonancia con la definición de Precision(h) y Recall(h), como está indicado arriba. La fórmula general es en realidad

$$F_\beta(h) = \frac{1}{n}\sum_{i=1}^{n} \frac{(1+\beta^2)\text{Precision}(h)_i \text{Recall}(h)_i}{(\beta^2 \text{Precision}(h)_i) + \text{Recall}(h)_i},$$

donde Precision$(h)_i$, Recall$(h)_i$ denotan la precisión y el recall del clasificador asociados a la muestra i-ésima, y β es un número real positivo que

controla cuanta más importancia se le da al recall frente a la precisión, y se usa en sistemas de recuperación de información a gran escala en los que el rendimiento está más ligado a la precisión o al recall. [1]

Un estadístico usado también en problemas de clasificación es la exactitud, que mide la fracción de ejemplos correctamente clasificados. Para un problema *multilabel* esto se traduce en la exactitud de subconjuntos (*subset accuracy*), que es una medida muy dura dada la dificultad de estos problemas.

$$\text{Subset-accuracy}(h) = \frac{1}{n}\sum_{i=1}^{n}\left[h(x_i) = Y_i\right]$$

Una métrica menos exigente es el índice o coeficiente de Jaccard, que mide el grado de similitud entre dos conjuntos [52]. Obsérvese que en el caso de que los conjuntos sean siempre unipuntuales — es decir, estamos ante un problema de clasificación binaria o multiclase — el índice de Jaccard es equivalente a la exactitud.

$$\text{Jaccard-index}(h) = \frac{1}{n}\sum_{i=1}^{n}\frac{\mid Y_i \cap h(x_i) \mid}{\mid Y_i \cup h(x_i) \mid}$$

5.2. Análisis cualitativo y cuantitativo

Los resultados obtenidos por los clasificadores se listan en la tabla 5.1 y están representados en la figura 5.2. En los modelos neuronales, los valores recogidos representan los mejores resultados tras muchos experimentos, en los que se ha probado con distintas configuraciones y valores para los hiperparámetros. En general, consideramos que un clasificador tiene mejor rendimiento que otro si las métricas para el *training set* y el *test set* son similares, y las métricas que "resumen" la precisión y el recall, así como la similitud de subconjuntos, son mayores. Es decir, damos especial importancia a las métricas F_1 y al índice de Jaccard.

Reproducimos también en la tabla 5.2 los resultados de Nigam en su trabajo de 2016, [38] que aborda el mismo problema y cuyos modelos se asemejan más a los nuestros.

En primer lugar, comentamos el rendimiento del clasificador basado en el modelo de regresión logística, que hemos tomado como línea de base (*ba-*

[1] Por ejemplo, en un sistema de detección de spam en email, es mucho peor clasificar un correo como spam cuando en realidad no lo es, que clasificar un correo como no spam cuando en realidad es un correo no deseado. Por tanto, para este sistema uno trata de reducir lo máximo posible el número de *falsos positivos.* Esto se hace poniendo enfásis en aumentar la precisión del clasificador *spam* y en aumentar el recall del clasificador *no spam.*

Modelo	Training set					Test set				
	Prec.	Rec.	F_1	Jacc.	Sub. acc.	Prec.	Rec.	F_1	Jacc.	Sub. acc.
Regresión logística	0.999	0.999	0.980	0.950	0.920	0.369	0.203	0.243	0.174	0.024
Red neuronal feedforward	0.706	0.564	0.595	0.499	0.186	0.634	0.593	0.572	0.461	0.117
RNN básica	0.801	0.352	0.410	0.499	0.222	0.550	0.250	0.410	0.462	0.131
RNN LSTM	0.850	0.555	0.550	0.510	0.231	0.790	0.450	0.441	0.460	0.133
RNN GRU	0.620	0.299	0.355	0.150	0.071	0.550	0.250	0.299	0.110	0.042

Tabla 5.1: Métricas obtenidas por los clasificadores en el *training set* y en el *test set*. Los valores para el *validation set* se excluye, pero son similares a los del *test set*. Los valores señalados en azul son los más altos para la correspondiente métrica.

Modelo	Training set			Test set		
	Prec.	Rec.	F_1	Prec.	Rec.	F_1
Regresión logística	0.9447	0.8718	0.8939	0.4381	0.2678	0.3000
Red neuronal feedforward	0.6629	0.3058	0.3925	0.6671	0.3062	0.3937
RNN básica	0.7932	0.3143	0.4286	0.5406	0.3610	0.4035
RNN LSTM	0.8310	0.3224	0.4469	0.7488	0.3199	0.4168
RNN GRU	0.8500	0.3074	0.4333	0.8505	0.3005	0.4203

Tabla 5.2: Métricas obtenidas por los clasificadores de Nigam [38] en el *training set* y en el *test set*. Los valores señalados en azul son los más altos para la correspondiente métrica.

seline model) para su comparación con los modelos neuronales. Al tratarse de un separador lineal muy sencillo, obtiene los peores resultados en todas las métricas, como esperábamos. Lo más evidente es el brutal *overfitting* que padece el modelo, acertando todas las etiquetas en un 92 % de los ejemplos del conjunto de datos de entrenamiento, mientras que en el *test set* acierta un 2 %. No obstante, los resultados obtenidos en el *test set* para el resto de métricas son decentes y mejores de lo esperados, dada la simplicidad del modelo. El sobreajuste que se produce es debido a una "memorización" de los datos: el modelo tiene tanta capacidad que el algoritmo de optimización es capaz de encontrar unos pesos que hacen que el *training set* se clasifique correctamente, pero en realidad el clasificador no ha "aprendido" nada, como pone de manifiesto el rendimiento sobre el *test set*. Es decir, al existir una gran cantidad de parámetros (recuérdese que los vectores de entrada viven en $\mathbb{R}^{40000}$), es posible encontrar un hiperplano separador que discrimina los datos linealmente, pero en realidad esta solución no refleja la"inteligencia" que requiere la naturaleza y la dificultad del problema, y que en teoría tendría un médico. Lo más probable es, que al tratarse de un modelo estadístico, el clasificador esté aprendiendo exclusivamente las frecuencias de aparición de los términos médicos en los documentos. Aprender las frecuencias de las palabras es de hecho una primera aproximación bas-

tante razonable para clasificar documentos, y Nigam constata en su trabajo que algunos modelos recurrentes hacen esto también en cierta medida.

Desde un punto de vista matemático, e independientemente del problema subyacente que quiere resolver, Bishop hace notar que el modelo de regresión logística optimizado mediante el método de estimación de máxima verosimilitud es propenso a exhibir *overfitting* severo. [6, cap. 4.3.2]. Esto suele ocurrir frecuentemente cuando el dataset es linealmente separable (como en el caso de la figura 3.1), ya que suele encontrarse una solución de los parámetros $\boldsymbol{w}$ cuya magnitud tiende a infinito. Esto hace que la función sigmoide se vuelva "infinitamente empinada" en el espacio transformado, asemejándose a una función de Heaviside (véase la figura 5.1), de manera que cada punto de entrenamiento de cada clase k es asignada una probabilidad posterior $p(\mathcal{C}_k \mid \boldsymbol{x}) = 1$. Este problema puede resolverse de varias maneras. Una de las más sencillas es modificar la función de coste (ecuación (3.7) de la página 22) para que tenga una noción de la magnitud de los pesos $\boldsymbol{w}$, penalizando los de mayor norma. En general, dada una función de coste $E(\boldsymbol{w})$, esta puede ser modificada de la siguiente manera

$$\tilde{E}(\boldsymbol{w}) = E(\boldsymbol{w}) + \frac{1}{2}\lambda\|w\|^2,$$

donde $\lambda > 0$ es un coeficiente de regularización que controla el decaimiento de los pesos. Los algoritmos de optimización basados en descenso de gradiente favorecerán ahora soluciones que minimicen la norma de $\boldsymbol{w}$, al estar ahora el gradiente dado por la expresión

$$\nabla\tilde{E}(\boldsymbol{w}) = \nabla E(\boldsymbol{w}) + \lambda w.$$

Esta técnica para evitar el *overfitting* se conoce como *regularización* L_2, y es solamente una de las "técnicas de regularización" que se emplean para solventar este problema.

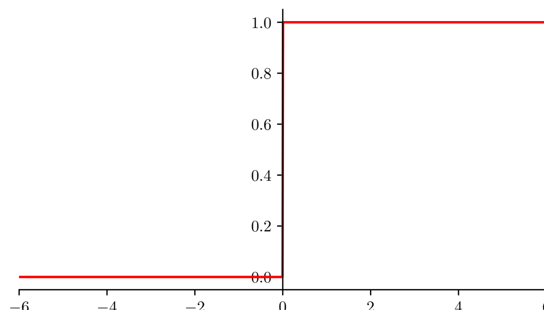


Figura 5.1: Función escalón de Heaviside. Compárese con la figura 3.3 de la página 31.

En cuanto a los resultados de la red neuronal, observamos una notable mejora en las métricas de recall y F_1 respecto al clasificador de Nigam. Nuestra red neuronal tiene 3 capas ocultas de 1000 neuronas cada una, mientras que la de Nigam tiene 2 capas ocultas de 300 y 100 neuronas, respectivamente. Usamos la función ReLU como función de activación en las dos capas ocultas, y la función sigmoide en la capa de salida. Puede que el mejor rendimiento de nuestro modelo se vea reflejado en esta diferencia de arquitectura. Con la arquitectura de Nigam nosotros obtuvimos resultados malos tanto en el *training set* como en el *test set*. Es muy interesante observar la evolución de las distintas métricas durante el entrenamiento de la red, que se muestra en la figura 5.3. Observése como en el inicio las métricas fluctúan hasta que, presumiblemente, el algoritmo de descenso de gradiente estocástico encuentra un camino por el que se minimiza el coste. Los valores entonces se estabilizan y la red comienza a aprender los pesos. Al final del proceso de entrenamiento, todas los gráficos de las métricas esbozan el inicio de una asíntota: la función de coste es más o menos plana para los parámetros aprendidos. La figura también muestra la evolución de la entropía cruzada para el *validation set*, que hemos usado para ajustar los hiperparámetros. Obsérvese que más o menos es una traslación de la curva de la entropía cruzada para el *training set*, quedando siempre por encima de esta última. Esto es una buena indicación de que el modelo está bien implementado y aprendiendo correctamente, pues el espacio entre estas dos curvas da una estimación del “error de generalización” que cometerá nuestro modelo cuando evaluemos la función de clasificación aprendida sobre el *test set*. Este espacio se va haciendo cada vez más grande al final del entrenamiento. Una técnica de regularización que hemos usado es la parada prematura (*early stopping*) del proceso de entrenamiento: cuando la entropía cruzada del *validation set* empieza a ser constante pero la entropía cruzada del *training set* sigue disminuyendo, es importante parar el proceso de entrenamiento para evitar el *overfitting*. De lo contrario, el riesgo de aumentar el error de generalización aumenta a medida que el espacio entre ambas curvas es mayor. Por ello se suele parar el entrenamiento cuando la función de coste para el *validation set* no ha mejorado en una ventana de tamaño fijo. [20, cap. 7.8]

Respecto a las modelos recurrentes, los resultados son extremadamente similares a los de Nigam, a excepción de la RNN GRU, que para Nigam se comporta como la RNN LSTM mientras que nuestra implementación tiene un rendimiento menor a las otras dos redes recurrentes. Tanto Nigam como nosotros obtenemos los mejores resultados con 100 neuronas en la capa oculta. No obstante, nuestra RNN básica tiene mejor rendimiento que la de Nigam en todas las métricas, en el *training set* y en el *test set*. Parece díficil determinar si un modelo recurrente tiene mejor rendimiento que una red neuronal *feedforward*, pues las métricas son muy dispares en el primer caso mientras que en el segundo éstas están más equilibradas. La RNN LSTM goza de una clara mayor precisión, pero esto no parece contribuir en las

métricas más importantes de F_1 y el índice de Jaccard. En resumen, no parece que los modelos recurrentes estén “aprendiendo” algo significativamente distinto a la red neuronal normal, ni que hagan uso del orden de las notas médicas en la secuencia temporal, algo que preveíamos que sería muy difícil. Es decir, que el rendimiento es comparable al de las redes neuronales normales, pero el coste computacional de entrenamiento es mucho mayor.

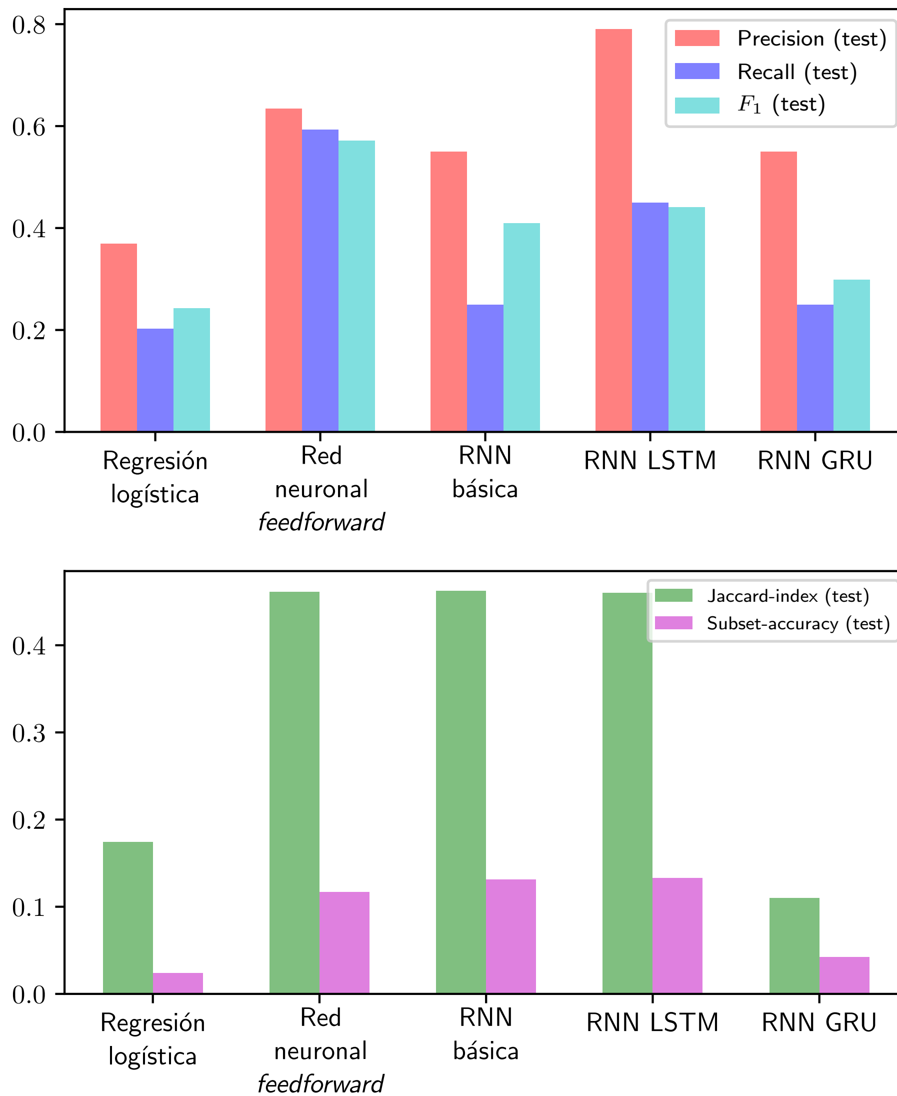
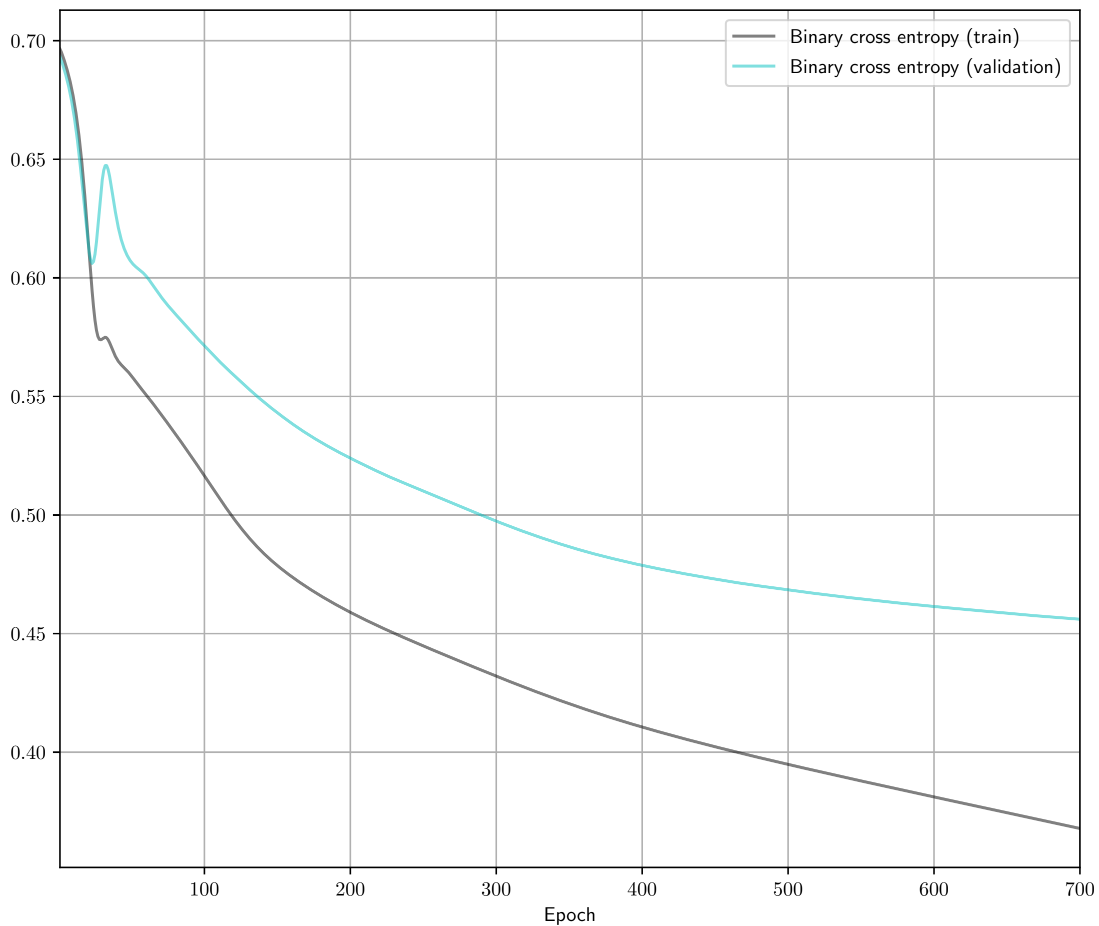


Figura 5.2: Gráficos con las métricas de la tabla 5.1 correspondientes al *test set*, agrupadas por clasificador. Arriba, las métricas de precisión, recall y F_1. Abajo, las métricas relacionadas con similitud de conjuntos: el índice de Jaccard y la exactitud de subconjuntos.



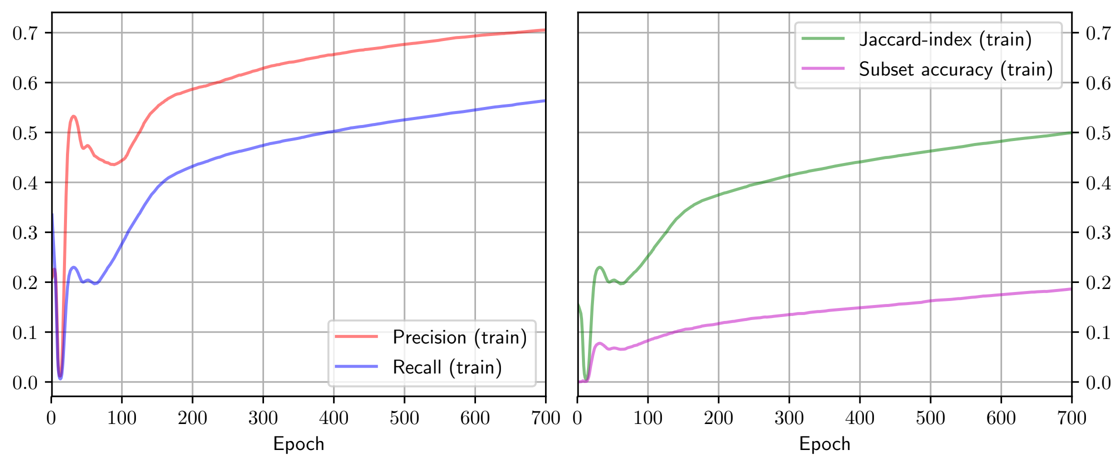


Figura 5.3: Evolución de algunas métricas durante el proceso de entrenamiento (700 epochs) de nuestra red neuronal feedforward.

Capítulo 6

Conclusiones y trabajo futuro

En este trabajo hemos implementado varios clasificadores para abordar el problema de asignación automática de códigos ICD-9 en base a notas médicas. Muchos investigadores han desarrollado clasificadores basados en técnicas tradicionales de *machine learning*, pero menos estudios se han llevado a cabo con modelos basados en *deep learning* y sobre el dataset MIMIC-III, debido a ser relativamente nuevo. Es por ello que hemos profundizado en el trabajo de Nigam, [38] y hemos implementado un modelo básico de regresión logística y varios modelos basados en redes neuronales, cuyos resultados esperamos que sean de gran utilidad como referencia para futuros investigadores. De nuevo los modelos basados en *deep learning* se muestran vencedores, obteniéndose resultados similares a los de la literatura existente. No obstante, esperábamos mejores resultados de los modelos recurrentes, que ponen de manifiesto que no se obtiene tanta ventaja al aprender dependencias temporales a largo plazo en las notas médicas. Esto también es constatado por otros. [38, 25].

Respecto a si algún clasificador ha sido exitoso abordando este problema, la respuesta radica en cómo se utilizaría en la práctica.

Si se espera implementar con él una inteligencia artificial que diagnostique automáticamente a pacientes en base a observaciones realizadas por personal sanitario, aún estamos muy lejos de conseguir tal objetivo. A pesar de que las métricas son bastante buenas, hay que tener en cuenta que sólo hemos abordado el problema para las 10 etiquetas ICD-9 más frecuentes en el dataset, y que los médicos son capaces de diagnosticar las enfermedades comunes excepcionalmente. Hay que tener también en consideración que el clasificador necesita notas escritas por profesionales sanitarios, y que a menudo quienes las escriben lo hacen con varios diágnosticos parciales en mente, si bien no están ya completamente decididos.

El clasificador sí podría ser utilizado en hospitales como una ayuda al personal administrativo para rellenar registros médicos más rápidamente. En varios países los códigos ICD-9 se registran en la historia clínica de un paciente por motivos administrativos, como por ejemplo emitir facturas a las compañías de seguros médicos. Esta etiquetación de historias se realiza manualmente, buscando las enfermedades en listados para obtener su código. Este proceso es muy largo y tedioso en la práctica. Un programa basado en un clasificador de los implementados en el presente trabajo podría "sugerir" las etiquetas automáticamente analizando la historia del paciente, si determina que éste padece alguna enfermedad común para la cual ha sido entrenado en detectar.

En cuanto a cómo obtener mejores resultados para este problema, creemos que existen dos grandes vías de trabajo futuro claramente diferenciadas:

1. mejorar la representación de las notas médicas como entrada de los modelos, y
2. emplear modelos más avanzados.

Respecto a la segunda vía de mejora, se podrían utilizar redes neuronales más sofisticadas, como redes recurrentes profundas con LSTMs o GRUs que capturen mejor las dependencias temporales entre las notas. También se podrían utilizar redes neuronales recursivas, en las que cada bloque de la red se encargara de procesar distintos grupos de notas relacionadas. Esto podría capturar mejor la idea de que en la vida real un diagnóstico se lleva a cabo teniendo en cuenta distintos aspectos o "componentes" de la enfermedad, y que todos estos trozos se combinan, no necesariamente secuencialmente, para emitir un diagnóstico final. Con un dataset suficientemente grande, esta idea podría llevarse al extremo. Por ejemplo, se podrían entrenar varias redes neuronales, cada una especializada en detectar una enfermedad concreta o un grupo de enfermedades relacionadas (cardiológicas, neurológicas, inmunológicas...), y que otro programa tomara como entrada la salida de estas redes y emitiera un juicio final, teniendo en cuenta toda la información. Otra manera de sofisticar los modelos podría ser la de tener una base de conocimientos médica estática, como un listado de las enfermedades y su sintomatología. La red utilizaría los datos de la nota y además de aprender del conjunto de notas de entrenamiento, consultaría la base de conocimientos para ayudarse y confirmar el diagnóstico. Este enfoque es el empleado por el reciente trabajo de Prakash et al., que utiliza "redes neuronales con memoria condensada" (*condensed memory neural networks, C-MemNNs*) sobre las notas de MIMIC-III, usando como base de conocimiento texto en bruto proveniente de artículos de enfermedades de la Wikipedia. [50] Esta aproximación al problema ya ha cosechado muy buenos resultados.

Sin embargo, creemos que una mejora sustancial del rendimiento de los modelos se puede lograr más fácilmente por la primera vía de mejora propuesta. Como subrayamos en el capítulo 1, los algoritmos sofisticados funcionan bien únicamente cuando la entrada a los modelos es buena, es decir, cuando modela bien el dominio de nuestro problema. Realizar diagnósticos es una tarea increíblemente complicada, y creemos firmemente que una mera representación de las notas como vectores que miden la frecuencia de aparición de las palabras no es suficiente. En primer lugar, el vocabulario escogido podría ser mejorado preprocesando más el dataset. Inspeccionándolo más en detalle, uno se da cuenta de que los textos clínicos son muy idiosincrásicos: contienen numerosos tecnicismos, abreviaturas, errores de ortografía y frases hechas que nuestra representación no tiene en consideración. Algunos ejemplos se listan en la tabla 6.1. Idear representaciones de la entrada que tengan en cuenta estas cosas ayudaría a la red a "entender" mejor el contenido de las notas: expresiones como "lóbulo frontal medial derecho" deberían interpretarse como una única unidad semántica y no deberían ser segmentadas por palabras. Abandonar una representación *bag of words* por una más sofisticada también es una alternativa prometedora. Los vectores *bag of words* no capturan suficiente información semántica de un documento. Técnicas de modelado de lenguaje como *word embeddings* podrían intentarse, asignando vectores "próximos" en un espacio vectorial a palabras semánticamente similares. Estas representaciones pueden generarse a su vez con modelos de *deep learning*, entrenándose redes sobre vocabulario médico especializado proveniente de algún lexicón establecido. Esta aproximación es implementada por Huang et al. en su trabajo, [25] en donde se utiliza `word2vec`. [18]

Faltas de ortografía		**Expresiones comunes**
zaroxalyn	hypercholesteremia	central line
zaroxlyn	hypercholesterinemia	subcortical white matter
zaroxolyn	hypercholestermia	ulcerative colitis
zaroxylin	hypercholesteroemia	small cell carcinoma
zaroxyln	hypercholesterolaemia	
zaroxylyn	hypercholesterolinemia	
	hypercholestolemia	
	hypercholestremia	
	hypercholestreolemia	
	hypercholestrolemia	
	hypercholeterolemia	

Tabla 6.1: Algunas faltas de ortografía y "frases hechas" que aparecen frecuentemente en MIMIC-III. Tabla tomada de [38].

Los hospitales generan cada día una cantidad ingente de datos que se almacena en registros médicos. La acumulación de toda esta información valiosa ha posibilitado la realización de estudios a escalas masivas que han mejorado los cuidados y tratamientos médicos. Gracias al *deep learning*, estamos comenzando a diseñar inteligencias artificiales que están revolucionando campos como la bioinformática, la genómica y la radiología. Un futuro en el que el diágnostico asistido por ordenador esté a la orden del día no parece tan lejano.

Bibliografía

Todos los enlaces fueron visitados por última vez el 8 de junio de 2018.

[1] *12.1. pickle — Python object serialization — Python 3.6.5 documentation.* URL: https://docs.python.org/3/library/pickle.html.

[2] Martín Abadi y col. «TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems». En: *CoRR* abs/1603.04467 (2016). arXiv: 1603.04467. URL: http://arxiv.org/abs/1603.04467.

[3] Rubin Donald B. «Iteratively Reweighted Least Squares». En: *Encyclopedia of Statistical Sciences.* Kotz y Johnson (eds.). New York: Wiley, 1983, págs. 272-275.

[4] *BCEWithLogitsLoss.* URL: https://pytorch.org/docs/stable/nn.html?highlight=bcewithlogits#bcewithlogitsloss.

[5] Y. Bengio, P. Frasconi y P. Simard. «The problem of learning long-term dependencies in recurrent networks». En: *IEEE International Conference on Neural Networks.* 1993, 1183-1188 vol.3. DOI: 10.1109/ICNN.1993.298725.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[7] Lars Buitinck y col. «API design for machine learning software: experiences from the scikit-learn project». En: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning.* 2013, págs. 108-122.

[8] Edward Choi y col. «Doctor AI: Predicting Clinical Events via Recurrent Neural Networks». En: *Proceedings of the 1st Machine Learning for Healthcare Conference.* Ed. por Finale Doshi-Velez y col. Vol. 56. Proceedings of Machine Learning Research. Children's Hospital LA, Los Angeles, CA, USA: PMLR, 18–19 Aug de 2016, págs. 301-318. URL: http://proceedings.mlr.press/v56/Choi16.html.

[9] *CITI Program – Collaborative Institutional Training Initiative.* URL: https://about.citiprogram.org/en/homepage/.

[10] *Conda.* URL: https://conda.io/docs/.

[11] Franck Dernoncourt y col. «De-identification of Patient Notes with Recurrent Neural Networks». En: *CoRR* abs/1606.03475 (2016). arXiv: 1606.03475. URL: http://arxiv.org/abs/1606.03475.

[12] John N. Tsitsiklis Dimitri P. Bertsekas. *Introduction to probability.* Second edition. Athena Scientific, 2008.

[13] *Docker - Build, Ship, and Run Any App, Anywhere.* URL: https://www.docker.com/.

[14] Alistair Edward William Johnson y col. «MIMIC-III, a freely accessible critical care database». En: 3 (mayo de 2016), pág. 160035.

[15] Stanford University School of Engineering. *Lecture 8 — Deep Learning Software.* URL: https://www.youtube.com/watch?v=6SlgtELqOWc&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv.

[16] Rong-En Fan y col. «LIBLINEAR: A Library for Large Linear Classification». En: *J. Mach. Learn. Res.* 9 (jun. de 2008), págs. 1871-1874. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1390681.1442794.

[17] Kunihiko Fukushima. «Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position». En: *Biological Cybernetics* 36.4 (abr. de 1980), págs. 193-202. ISSN: 1432-0770. DOI: 10.1007/BF00344251. URL: https://doi.org/10.1007/BF00344251.

[18] Y. Goldberg y O. Levy. «word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method». En: *ArXiv e-prints* (feb. de 2014). arXiv: 1402.3722 [cs.CL].

[19] Ira Goldstein, Anna Arzrumtsyan y Ozlem Uzuner. «Three Approaches to Automatic Assignment of ICD9-CM Codes to Radiology Reports». En: 2007 (feb. de 2007), págs. 279-83.

[20] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning.* http://www.deeplearningbook.org. MIT Press, 2016.

[21] *GPU Grant Program — NVIDIA Developer.* URL: https://developer.nvidia.com/academic_gpu_seeding.

[22] Ángel Javier Alonso Hernández. *Deep learning aplicado al resumen de texto.* Trabajo de Fin de Grado en Ingeniería Informática y Matemáticas (Universidad Complutense, Facultad de Informática, curso 2016/2017). 2017. URL: http://eprints.ucm.es/44573/.

[23] Matthew Honnibal e Ines Montani. «spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing». En: *To appear* (2017).

[24] Sycorax (https://stats.stackexchange.com/users/22311/sycorax). *Why isn't Logistic Regression called Logistic Classification?* Cross Validated. URL:https://stats.stackexchange.com/q/127044 (version: 2017-06-29). eprint: `https://stats.stackexchange.com/q/127044`. URL: `https://stats.stackexchange.com/q/127044`.

[25] Jinmiao Huang, Cesar Osorio y Luke Wicent Sy. «An Empirical Evaluation of Deep Learning for ICD-9 Code Assignment using MIMIC-III Clinical Notes». En: *CoRR* abs/1802.02311 (2018). arXiv: `1802.02311`. URL: `http://arxiv.org/abs/1802.02311`.

[26] *International Classification of Diseases,Ninth Revision (ICD-9).* URL: `https://www.cdc.gov/nchs/icd/icd9.htm`.

[27] Kevin Jarrett y col. «What is the best multi-stage architecture for object recognition?» En: *Computer Vision, 2009 IEEE 12th International Conference on.* IEEE. 2009, págs. 2146-2153.

[28] Justin Johnson. *jcjohnson/cnn-benchmarks: Benchmarks for popular CNN models.* URL: `https://github.com/jcjohnson/cnn-benchmarks`.

[29] Andrej Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks.* Mayo de 2015. URL: `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`.

[30] Elman Jeffrey L. «Finding Structure in Time». En: *Cognitive Science* 14.2 (), págs. 179-211. DOI: `10.1207/s15516709cog1402_1`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1207/s15516709cog1402_1`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1402_1`.

[31] *Language Processing Pipelines · spaCy Usage Documentation.* URL: `https://spacy.io/usage/processing-pipelines`.

[32] *lanpa/tensorboard-pytorch: tensorboard for pytorch (and chainer, mxnet, numpy, ...)* URL: `https://github.com/lanpa/tensorboard-pytorch`.

[33] *Logistic regression.* URL: `http://scikit-learn.org/stable/modules/linear_model.html#logistic-regression`.

[34] *MIMIC Critical Care Database.* URL: `https://mimic.physionet.org/`.

[35] *MIT-LCP/mimic-code: MIMIC Code Repository: Code shared by the research community for the MIMIC-III database.* URL: `https://github.com/MIT-LCP/mimic-code`.

[36] *Models & Languages.* URL: `https://spacy.io/usage/models`.

[37] Ishna Neamatullah y col. «Automated de-identification of free-text medical records». En: *BMC Medical Informatics and Decision Making* 8.1 (jul. de 2008), pág. 32. ISSN: 1472-6947. DOI: 10.1186/1472-6947-8-32. URL: https://doi.org/10.1186/1472-6947-8-32.

[38] Priyanka Nigam. *Applying Deep Learning to ICD-9 Multi-label Classification from Medical Records*. 2016. URL: https://cs224d.stanford.edu/reports/priyanka.pdf.

[39] *NOTEEVENTS*. URL: https://mimic.physionet.org/mimictables/noteevents/.

[40] Eiji Oda. «Metabolic syndrome: its history, mechanisms, and limitations». En: *Acta Diabetologica* 49.2 (abr. de 2012), págs. 89-95. ISSN: 1432-5233. DOI: 10.1007/s00592-011-0309-6. URL: https://doi.org/10.1007/s00592-011-0309-6.

[41] Christopher Olah. *Understanding LSTM Networks*. Ago. de 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[42] Adam Paszke y col. «Automatic differentiation in PyTorch». En: *NIPS-W*. 2017.

[43] F. Pedregosa y col. «Scikit-learn: Machine Learning in Python». En: *Journal of Machine Learning Research* 12 (2011), págs. 2825-2830.

[44] Adler Perotte y col. «Diagnosis code assignment: models and evaluation metrics». En: *Journal of the American Medical Informatics Association* 21.2 (2014), págs. 231-237. DOI: 10.1136/amiajnl-2013-002159. eprint: /oup/backfile/content_public/journal/jamia/21/2/10.1136_amiajnl-2013-002159/2/21-2-231.pdf. URL: http://dx.doi.org/10.1136/amiajnl-2013-002159.

[45] *PhysioNet*. URL: https://physionet.org/.

[46] *pip — pip 10.0.1 documentation*. URL: https://pip.pypa.io/en/stable/.

[47] *Plot multinomial and One-vs-Rest Logistic Regression*. URL: http://scikit-learn.org/stable/auto_examples/linear_model/plot_logistic_multinomial.html#sphx-glr-auto-examples-linear-model-plot-logistic-multinomial-py.

[48] *PostgreSQL + Python — Psycopg*. URL: http://initd.org/psycopg/.

[49] *PostgreSQL: The world's most advanced open source database*. URL: https://www.postgresql.org.

[50] Aaditya Prakash y col. «Condensed Memory Networks for Clinical Diagnostic Inferencing». En: (dic. de 2016).

[51] *PyTorch*. URL: https://pytorch.org/.

[52] Raimundo Real y J Vargas. «The Probabilistic Basis of Jaccard's Index of Similarity». En: 45 (sep. de 1996), págs. 380-385.

[53] David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams. «Neurocomputing: Foundations of Research». En: ed. por James A. Anderson y Edward Rosenfeld. Cambridge, MA, USA: MIT Press, 1988. Cap. Learning Representations by Back-propagating Errors, págs. 696-699. ISBN: 0-262-01097-6. URL: http://dl.acm.org/citation.cfm?id=65669.104451.

[54] Leah S. Larkey y W Bruce Croft. «Automatic Assignment of ICD9 Codes To Discharge Summaries». En: (feb. de 1996).

[55] Oliver Bear Don't Walk IV Sandeep Ayyar. *Tagging Patient Notes With ICD-9 Codes*. Abr. de 2018. URL: https://web.stanford.edu/class/cs224n/reports/2744196.pdf.

[56] *scipy.sparse.coo_matrix — SciPy v1.1.0 Reference Guide*. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html.

[57] Hava T. Siegelmann. «Computation Beyond the Turing Limit». En: *Science* 268.5210 (1995), págs. 545-548. ISSN: 0036-8075. DOI: 10.1126/science.268.5210.545. eprint: http://science.sciencemag.org/content/268/5210/545.full.pdf. URL: http://science.sciencemag.org/content/268/5210/545.

[58] *sklearn.linear_model.LogisticRegression*. URL: http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html.

[59] *sklearn.metrics.f1_score*. URL: http://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html#sklearn.metrics.f1_score.

[60] *spaCy: Industrial-Strength Natural Language Processing*. URL: https://spacy.io/.

[61] *TensorBoard: Visualizing Learning — TensorFlow*. URL: https://www.tensorflow.org/programmers_guide/summaries_and_tensorboard.

[62] *tf.nn.sigmoid_cross_entropy_with_logits*. URL: https://www.tensorflow.org/api_docs/python/tf/nn/sigmoid_cross_entropy_with_logits.

[63] *torch.sparse — PyTorch master documentation*. URL: https://pytorch.org/docs/0.3.1/sparse.html.

[64] Antonio Valdés. *Apuntes de geometría computacional*. Sin publicar.

[65] *Virtualenv — virtualenv 16.0.0 documentation*. URL: https://virtualenv.pypa.io/en/stable/.

[66] Wikipedia contributors. *Belmont Report — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Belmont_Report&oldid=812781401.

[67] Wikipedia contributors. *Generalized linear model — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Generalized_linear_model&oldid=840151351.

[68] Wikipedia contributors. *Gibbs' inequality — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Gibbs%27_inequality&oldid=837811586.

[69] Wikipedia contributors. *n-gram — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=N-gram&oldid=835900923.

[70] Wikipedia contributors. *Self-information — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Self-information&oldid=839581743.

[71] Wikipedia contributors. *Sigmoid function — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Sigmoid_function&oldid=838359073.

[72] Wikipedia contributors. *Supervised learning — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/wiki/Supervised_learning.

[73] Wikipedia contributors. *Virtualización a nivel de sistema operativo — Wikipedia, The Free Encyclopedia.* https://es.wikipedia.org/w/index.php?title=Virtualizaci%C3%B3n_a_nivel_de_sistema_operativo&oldid=108103336.

[74] Shi Yan. *Understanding LSTM and its diagrams.* Mar. de 2016. URL: https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714.

[75] M. L. Zhang y Z. H. Zhou. «A Review on Multi-Label Learning Algorithms». En: *IEEE Transactions on Knowledge and Data Engineering* 26.8 (ago. de 2014), págs. 1819-1837. ISSN: 1041-4347. DOI: 10.1109/TKDE.2013.39.