UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

PROTOTIPO DE CLASIFICADOR MULTICLASE PARA RELATOS MÉDICOS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

RODRIGO ADRIÁN FUENTES ZÚÑIGA

PROFESOR GUÍA:
JORGE PÉREZ ROJAS

MIEMBROS DE LA COMISIÓN:
BÁRBARA POBLETE LABRA
SANDRA DE LA FUENTE GONZALEZ

SANTIAGO DE CHILE
2021

Resumen

En Chile, todas las entidades empleadoras del país deben estar afiliadas a un organismo administrador del Seguro Social contra Riesgos de Accidentes del Trabajo y Enfermedades Profesionales. La ACHS es uno de estos organismos.

Cuando un trabajador de una entidad afiliada a la ACHS sufre un accidente laboral, recurre a una de las sedes de la ACHS donde recibe atención médica. En este proceso, se generan relatos y textos escritos por distintos trabajadores de la ACHS. Además, el siniestro ocurrido al trabajador debe ser clasificado en distintas categorías.

Actualmente, la clasificación de los textos es realizada manualmente por trabajadores de la ACHS. Debido a que existen alrededor de 3000 categorías, existe un porcentaje no menor de textos que son clasificados erróneamente.

En este proyecto implementamos un prototipo de clasificador de textos médicos que esperamos en el futuro sirva como una ayuda a la toma de decisiones para los trabajadores de la ACHS que deben clasificar estos textos. Esperamos que el prototipo implementado sirva como una base para implementar un clasificador que disminuya el porcentaje de errores cometidos, además de unificar los criterios de las personas encargadas de clasificar los textos.

En los últimos años se han realizado grandes avances en el campo del procesamiento de lenguaje natural (PLN). En este proyecto utilizamos dos arquitecturas de *Deep Learning* para implementar clasificadores de relatos médicos. La primera es BiLSTM, que utilizamos como un baseline para nuestro modelo final basado en BERT. También implementamos otro baseline basado en un modelo más simple llamado *Naive Bayes*.

De los resultados concluimos que BERT es la mejor alternativa para realizar esta tarea, obteniendo los resultados más equilibrados. También observamos que al disminuir el tamaño del universo de categorías en las que puede ser clasificado un relato, las métricas reportadas aumentan su valor significativamente. Debido a esto, un trabajo importante a realizar en el futuro es disminuir la cantidad de categorías en las que un relato puede ser clasificado. Una forma de llevar esto a cabo es ordenarlas jerárquicamente, y utilizar los clasificadores implementados en este proyecto para obtener una clasificación general de los relatos.

A todos aquellos que me acompañaron
en algún momento de mi carrera

Tabla de Contenido

Índice de Ilustraciones

Capítulo 1

Introducción

El presente documento describe la implementación de un clasificador de relatos médicos generados en la Asociación Chilena de Seguridad (ACHS). En el presente capítulo describimos la situación actual y la motivación para realizar este trabajo de título. En el Capítulo 2 presentamos los antecedentes generales y el estado del arte de las herramientas y técnicas utilizadas. En el Capítulo 3 describimos la implementación llevada a cabo. Finalmente en los Capítulos 4, 5 y 6, presentamos los resultados obtenidos, realizamos un análisis de estos, y entregamos las conclusiones del trabajo realizado.

1.1. Motivación

La ACHS es una organización privada sin fines de lucro que participa en el sistema de seguridad social de Chile. Por ley, todas las entidades empleadoras del país deben estar afiliadas a un organismo administrador del Seguro Social contra Riesgos de Accidentes del Trabajo y Enfermedades Profesionales [1]. La ACHS es uno de estos organismos y entrega tres tipos de prestaciones: prevención de accidentes laborales, prestaciones económicas y prestaciones de salud.

El presente trabajo de título se concentra en las prestaciones de salud recién mencionadas. Cuando un trabajador de una entidad afiliada a la ACHS sufre un accidente laboral, debe recurrir a una de sus sedes de atención médica a reportar dicho accidente. Este reporte tiene un protocolo establecido en el cual se generan relatos y textos médicos que luego deben ser clasificados en una o más categorías.

Nuestro objetivo final es implementar un clasificador de textos que en el futuro le permita a la ACHS validar el proceso de clasificación de relatos de manera automática. Esto tendría un impacto positivo para los médicos que deben clasificar los relatos, al servirles como un sistema de ayuda a la toma de decisiones, disminuyendo los errores de diagnóstico y ayudando así a mantener una base de datos más limpia.

1.2. Situación Actual

En la actualidad, el proceso de diagnóstico se hace de forma manual por trabajadores de la ACHS. El proceso parte cuando la persona accidentada acude a la sede de atención médica y un recepcionista registra el relato entregado por la persona sobre su accidente. Posteriormente, la persona es atendida por un médico, que le realiza un examen físico y anota su diagnóstico, junto con una o varias categorías estandarizadas en las cuales él cree que entra el caso. Actualmente existen alrededor de 3000 categorías distintas.

Cuando una persona sufre un accidente y se convierte en paciente de la ACHS, puede tener varias visitas a un centro de atención a lo largo de su tratamiento. Cada uno de estos encuentros tiene asociado un documento que lo registra. Actualmente existen alrededor de 800 mil documentos que relatan encuentros con pacientes referentes a alrededor de 400 mil accidentes distintos.

El principal problema de la forma en la que se lleva a cabo el proceso actualmente es la susceptibilidad que tiene ante el error humano. Al ser categorías estandarizadas a través de un código, es fácil confundir un código con otro. También, al ser una decisión humana, está sujeta al criterio de cada médico tratante. Los 800 mil relatos nombrados previamente están escritos por más de mil médicos distintos. Todas estas personas tienen criterios y experiencias distintas. Muchas veces ocurre que médicos poco experimentados categorizan un accidente de una forma, y otro con más experiencia lo hace de otra.

Creemos que incorporar un modelo computacional al proceso disminuirá la cantidad de errores que se cometen al categorizar relatos, permitiendo que la clasificación de los accidentes sea más rigurosa de lo que es actualmente.

1.3. Resultados generales

Para apoyar la toma de decisiones en los procesos de clasificación de la ACHS, en un principio propusimos implementar un prototipo de clasificador multiclase para relatos médicos de trabajadores accidentados. Para llevar a cabo esta implementación, utilizamos redes neuronales profundas basadas en BERT, y las comparamos con dos baselines implementados, uno consistente en BiLSTM y otro más simple llamado *Naive Bayes*. Esperamos que los clasificadores implementados sirvan como una base para implementar un clasificador utilizable en producción que ayude a disminuir los errores de diagnóstico y mantenga una clasificación más rigurosa de los textos médicos de la ACHS.

Para llevar a cabo lo que nos propusimos en un principio, realizamos los siguientes pasos intermedios:

1. Definimos una métrica de evaluación para los clasificadores que esperamos refleje efectivamente el impacto que tendrán estos para la ACHS. Si bien se pueden utilizar varias métricas, nos quedamos con los macro y micro *averages* de la precisión y el *recall* obtenidos por los modelos, además del coeficiente de Jaccard para comparar un modelo con otro. Más adelante explicamos en detalle estas métricas.

2. Definimos *baselines*, fáciles de replicar, con los cuáles comparar el prototipo. Los *baselines* implementados consisten en *Naive Bayes* y BiLSTM.

3. Generamos modelos de redes neuronales profundas basadas en BERT para llevar a cabo la tarea propuesta.

4. Validamos el prototipo implementado utilizando las métricas nombradas anteriormente y comparando sus resultados con los de los baselines implementados.

Capítulo 2

Antecedentes Generales

2.1. Machine Learning

Machine Learning es un área de la inteligencia artificial que permite a los sistemas funcionar aprendiendo directamente desde los datos en lugar de las instrucciones explícitas del programador.

Para aprender desde los datos, se diseñan modelos consistentes en estructuras y algoritmos que interactúan entre sí y con los datos para llevar a cabo alguna tarea definida. Estas estructuras y algoritmos están definidos por múltiples parámetros que deben ser ajustados para llevar a cabo la tarea para la que fue implementado el modelo.

Para poder ajustar sus parámetros, pasamos a los modelos por un proceso que llamamos entrenamiento. Durante dicho proceso, los modelos leen los datos, realizan una tarea, y dependiendo de que tan correctamente la hayan llevado a cabo, ajustamos sus parámetros utilizando algoritmos diseñados para que la próxima vez que el modelo se enfrente a la misma tarea, pueda realizarla más correctamente. Más adelante explicamos más en detalle que significa que un modelo realice una tarea correctamente, pero por ahora podemos decir que el proceso de entrenamiento tiene como objetivo que el modelo se equivoque lo menos posible al llevar a cabo una tarea.

Luego de la etapa de entrenamiento, se espera que el modelo sea capaz de llevar a cabo la tarea para la que fue entrenado. Por ejemplo, si queremos entrenar un modelo para que clasifique imágenes en distintas categorías lo que haremos será, en la etapa de entrenamiento, pasarle al modelo un conjunto de imágenes y la categoría a la que pertenecen. Luego, esperaríamos que al entregarle una imagen con la que el modelo no se ha encontrado, sea capaz de devolver como output su categoría correspondiente.

2.1.1. Aprendizaje Supervisado y Aprendizaje no Supervisado

Machine Learning se divide en dos grandes áreas, el aprendizaje supervisado y el aprendizaje no supervisado. Estos conceptos se refieren a cómo utilizaremos los datos que tenemos en la etapa de entrenamiento.

Si estamos realizando aprendizaje no supervisado, esperamos que el modelo, en la etapa de entrenamiento, lea datos que no están etiquetados y sea capaz de encontrar una estructura o patrón dentro de ellos.

Por otro lado, si estamos realizando aprendizaje supervisado para que un modelo pueda realizar una tarea, debemos mostrarle ejemplos en los que dicha tarea ya está resuelta, para que luego el modelo pueda llevarla a cabo solo. Al hacer esto decimos que estamos entregándole datos etiquetados al modelo.

Por ejemplo, en el presente trabajo de título entrenamos modelos para clasificar relatos médicos en varias categorías. Para hacer esto entrenamos los modelos pasándoles miles de ejemplos consistentes en texto escrito, acompañado por el listado de categorías al que el relato pertenece. Luego de hacer esto evaluamos si es que los modelos son capaces de clasificar relatos no vistos en la etapa de entrenamiento.

2.2. Problema de Clasificación Multiclase

Uno de los problemas que más se busca resolver con técnicas de *Machine Learning* es el de clasificación. En su versión más básica el problema trata de, dada una entidad, asignarle una categoría. Si la cantidad de categorías posibles para una entidad son dos, decimos que es un problema de clasificación binaria. Por ejemplo, clasificar emails en deseados e indeseados.

Cuando en un problema de clasificación existen más de dos categorías posibles, decimos que es un problema de clasificación multiclase. Por ejemplo, de un conjunto de autores, decidir cuál escribió un relato.

Finalmente, si en un problema de clasificación, las entidades pueden tener asignada más de una categoría a la vez, decimos que es un problema de clasificación multietiqueta. Por ejemplo, de un conjunto de objetos, decidir cuales objetos se encuentran en una imagen.

En el caso del presente trabajo, el problema de clasificación que buscamos resolver es multiclase multietiqueta. Puesto que para un relato existen más de dos categorías posibles a las que puede pertenecer, y además puede pertenecer a varias categorías a la vez.

2.3. Naive Bayes

Naive Bayes es un algoritmo de clasificación binaria simple que puede ser muy rápido en comparación a otros algoritmos de clasificación más avanzados. Trabaja utilizando el teorema de Bayes de probabilidades para predecir la clase de datos desconocidos. En el presente trabajo de título lo utilizamos como un *baseline* para luego comparar sus resultados con los demás algoritmos que implementamos.

Se le llama *Naive* o ingenuo en español, debido a que para funcionar, el algoritmo asume que la presencia de alguna característica o *feature* en los datos no está relacionada con la presencia de ninguna otra característica. Por ejemplo, si consideramos que una manzana es roja, redonda y con diámetro de unos 10cm, supondremos que cada una de estas características aportan independientemente a la probabilidad de que una fruta sea una manzana, a pesar de que en la realidad la presencia de esas características dependen unas de las otras. [2]

El teorema de Bayes expresa la probabilidad condicional de un evento A dado un evento B, en función de la probabilidad del evento B dado A y la probabilidad de que ocurra solo A. La expresión es la siguiente:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

En nuestro caso, al tratarse de un problema de clasificación, lo que nos interesa es saber la probabilidad de que un ejemplo pertenezca a una clase, dado que posee cierta característica.

Para entrenar un algoritmo de clasificación *Naive Bayes*, lo que se hace es entregarle un conjunto de ejemplos, a su vez consistentes en un conjunto de características, junto a un booleano que es verdadero si el ejemplo pertenece a la categoría que estamos prediciendo. Lo que hará el modelo en esta etapa es calcular la distribución de probabilidad de que un ejemplo pertenezca a la categoría para cada característica. Luego, cuando el clasificador vea un ejemplo nuevo, calculará utilizando el teorema de Bayes la probabilidad de, dado un conjunto de características, pertenecer o no a la categoría correspondiente.

Es importante notar que este clasificador solo puede decidir si un ejemplo pertenece a una categoría o no. En nuestro caso, al tener un problema multiclase multietiqueta, lo que queremos hacer es asignarle una o varias categorías a un ejemplo. Para lograr esto, lo que hicimos fue implementar un clasificador *Naive Bayes* para cada una de las posibles categorías, luego decimos que un ejemplo pertenece a las categorías cuyos clasificadores predijeron que dicho ejemplo sí pertenecía.

2.4. Deep Learning

Deep Learning es una rama del área de *Machine Learning* que basa su funcionamiento en estructuras llamadas redes neuronales profundas. Una red neuronal es un modelo vagamente inspirado en las redes neuronales biológicas [3] que tiene unidades de cómputo, llamadas neuronas artificiales, distribuidas en capas. Una capa contiene un número fijo de neuronas y se conecta con la siguiente para enviarle información. Le llamamos arquitectura a la forma en la que se organizan las neuronas, capas y conexiones en una red neuronal.

2.4.1. Redes Neuronales

Una red neuronal es una estructura consistente en un conjunto de unidades de cómputo a las que llamamos neuronas artificiales o perceptrones. Un perceptrón intenta simular el comportamiento de una neurona biológica.

Un perceptrón recibe como input un vector de dimensión finita. El resultado entregado consiste en la aplicación de una función a la suma ponderada de los valores del vector. A la función aplicada le llamamos función de activación (Figura 2.1). A los valores por los que ponderamos al vector de entrada les llamamos parámetros, y son modificados en la etapa de entrenamiento de la red.

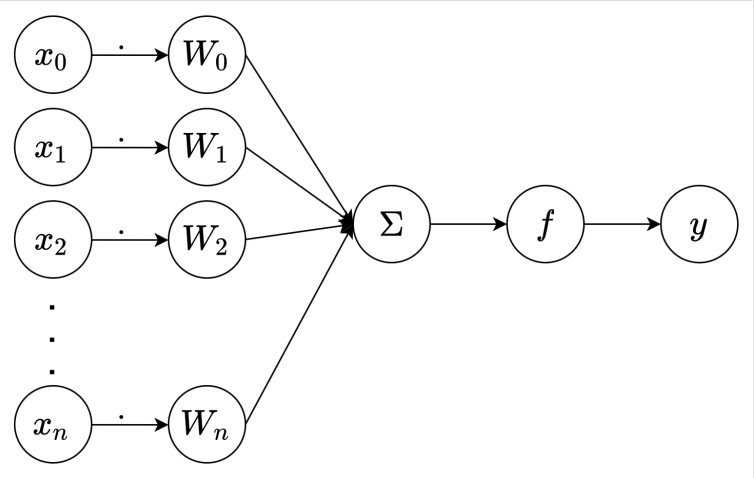


Figura 2.1: Perceptrón: Representamos con W el vector de parámetros y con f la función de activación.

Si el input X es el vector $X = (x_0, x_1, x_2, ..., x_n)$ y W el vector de parámetros $W = (W_0, W_1, W_2, ..., W_n)$ decimos que el output y se ve de la forma:

$$y = f(X^t \cdot W)$$

Para construir una red neuronal se organizan los perceptrones en capas. Una capa es un conjunto finito de perceptrones. Una de las arquitecturas más básicas de redes neuronales consiste en pasarle todos los outputs de una capa como input a la siguiente. A esta arquitectura le llamamos *feed forward* (Figura 2.2).

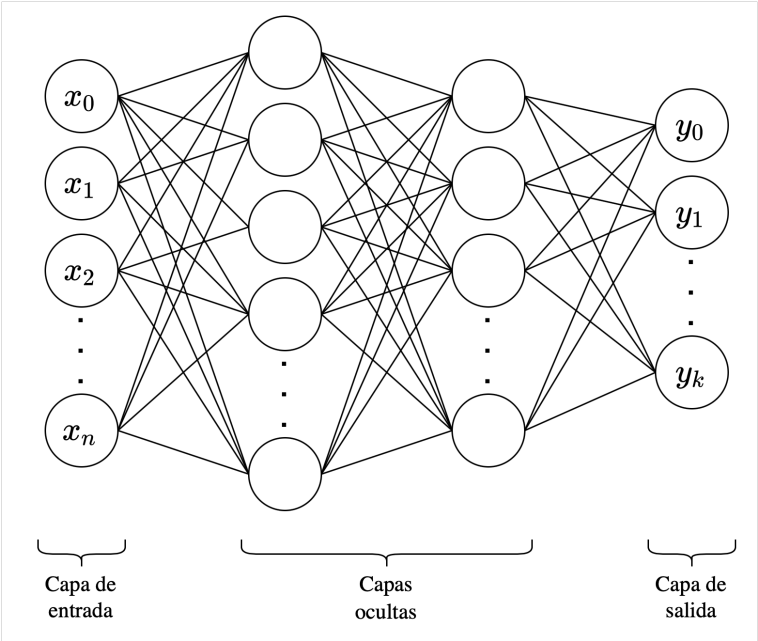


Figura 2.2: Ejemplo de red *feed forward*. La red recibe un input de tamaño n y entrega un output de tamaño k.

Ahora podemos decir que si el input de una capa X es el vector $X = (x_0, x_1, x_2, ..., x_n)$ y W una matriz de dimensiones (n, j) donde j es la cantidad de perceptrones de la capa, y cada columna de W representa el vector de parámetros de un perceptrón distinto, el output de la capa es un vector y de la forma:

$$y = f(X^t \cdot W + b)$$

Notemos que esta vez agregamos un vector b. Este es de tamaño j, y le permite a la red agregar un valor escalar a cada output de los perceptrones. Este vector también es ajustado en la etapa de entrenamiento.

El objetivo de una red neuronal es aproximar una función f^*. Para hacer esto, la red define una función $y = f(x; \theta)$ y aprende los valores de los parámetros θ durante la etapa de entrenamiento para obtener la mejor aproximación posible. [4]

La idea es que cada capa de la red a su vez aproxima una función. Por ejemplo, si tenemos tres capas, cada una aproxima a una función $f^{(1)}, f^{(2)}$ y $f^{(3)}$ respectivamente. Luego, la red aproximará la función $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. A medida que se agreguen capas, y en consecuencia aumente el largo de esta cadena, decimos que la red neuronal se vuelve cada vez más profunda. A la primera capa de la red, que recibe el input, le llamamos capa de entrada. A la última capa de la red que entrega el output final, le llamamos capa de salida. A las demás capas intermedias les llamamos capas ocultas.

En la etapa de entrenamiento, intentamos ayudar a la red neuronal a aproximar lo más cercanamente posible a f^*. Para hacer esto le entregamos ejemplos x acompañados de una etiqueta $y \approx f^*(x)$. Los datos de entrenamiento especifican directamente que output debe entregar la red en su capa de salida. El modelo debe decidir como usar las capas ocultas para lograr entregar este output. [4]

Se dice que las redes neuronales profundas aprenden a representar el input que se les entrega de manera incremental a través de sus capas, empezando por representaciones poco abstractas hasta llegar a representaciones con un alto nivel de abstracción [5]. Un ejemplo concreto es como un modelo de *Deep Learning* que aprende a representar texto, primero aprende a representar letras, luego palabras y finalmente estructuras de más alto nivel como oraciones.

2.4.2. Deep Learning versus Machine Learning

Una ventaja de las técnicas de *Deep Learning* que ayuda a entender por qué preferimos utilizarlas por sobre métodos de *Machine Learning* tradicional, es que su efectividad escala junto a la cantidad de datos que se posee. La evidencia empírica en el uso de arquitecturas de *Deep Learning* ha mostrado que mientras más datos de entrenamiento se posea, la eficiencia en general mejora. Algoritmos más antiguos de *Machine Learning* suelen llegar a un máximo de efectividad, y luego, por mucho que se consigan más datos, la efectividad no aumentará significativamente. A cambio, las redes neuronales profundas requieren una capacidad de cómputo más elevada. Por esto las GPUs (Unidad de procesamiento gráfico, o *Graphic Processing Unit* en inglés) se han vuelto una pieza esencial a la hora de ejecutar cualquier algoritmo de Deep Learning [6].

En el área de *Machine Learning* tradicional, las características importantes del input deben ser identificadas por un experto en el dominio para disminuir la complejidad de los datos y permitirle al algoritmo trabajar. Esto no ocurre en el área de *Deep Learning*, donde los modelos pueden aprender qué características son las más importantes. En nuestro caso, si quisiéramos utilizar *Machine Learning* tradicional, probablemente necesitaríamos conocedores de medicina y lingüística que pudieran identificar en qué partes de los relatos médicos habría que fijarse para sacar conclusiones.

2.4.3. Función de pérdida

La función de pérdida es una función que, en la etapa de entrenamiento, penaliza a la red por cometer errores en sus predicciones. Con esta penalización, la red es capaz de adecuar sus parámetros para que en el futuro la penalización disminuya.

Una función de pérdida recibe como input el output predicho por la red para un ejemplo, y el output que sabemos correcto para ese ejemplo desde nuestros datos de entrenamiento. En base a esto, la función de perdida entrega como output un valor real que representa la penalización. La noción es que mientras más alto sea dicho valor, "más incorrecta" fue la predicción de la red.

2.4.3.1. Entropía cruzada

Entropía cruzada es la función de pérdida utilizada cuando estamos clasificando ejemplos en una sola clase. En estos casos lo que hacemos es interpretar el output de la red como una distribución de probabilidad para todas las clases.

Por ejemplo, si tenemos un problema de clasificación de 3 clases A, B y C. Nuestro output podría verse de la forma $[0.1, 0.6, 0.3]$. Esto lo interpretaríamos como que existe un $60\,\%$ de probabilidades de que el ejemplo pertenezca a la clase B. Por otro lado tendríamos la distribución real, supongamos que efectivamente el ejemplo pertenece a la clase B, en este caso la distribución de probabilidad real se verá de la forma $[0.0, 1.0, 0.0]$.

Para calcular la entropía cruzada se utiliza la siguiente fórmula:

$$H(p, q) = -\sum_{x} p(x) ln(q(x))$$

Donde p es la distribución de probabilidad real y q la predicha por el algoritmo.

En el caso del ejemplo anterior, el valor de la entropía cruzada sería:

$$H = -(0.0 \cdot ln(0.1) + 1.0 \cdot ln(0.6) + 0.0 \cdot ln(0.3)) = 0.51$$

Este valor lo interpretamos como qué tan incorrecta está la predicción hecha por el modelo. En la etapa de entrenamiento se espera que un modelo sea capaz de adecuar sus parámetros para que esta función llegue a ser lo más cercana a 0 posible.

2.4.3.2. Entropía cruzada binaria

Utilizamos la entropía cruzada cuando los ejemplos pueden pertenecer solo a una clase a la vez. En nuestro caso, en el que cada ejemplo puede pertenecer a varias clases, utilizamos la entropía cruzada binaria.

La diferencia entre la entropía cruzada normal y la binaria, es que la segunda, en lugar de tratar el output como una distribución de probabilidad para todas las clases, trata cada uno de valores del vector de output como una distribución de probabilidad binaria para una de las clases.

Por ejemplo, si tenemos un problema de clasificación multiclase multietiqueta con 3 clases posibles A, B y C, podríamos tener un vector de output $[0.6, 0.9, 0.1]$. En este caso, este vector lo interpretamos como que existe un 60 % de probabilidades de que el ejemplo pertenezca a la clase A, un 90 % de que pertenezca a la clase B y un 10 % de que pertenezca a la clase C.

En este caso, primero calculamos un vector de entropías de la forma $H = [h_A, h_B, h_C]$ donde h_i con $i \in A, B, C$ se ve de la forma:

$$h_i(p, q) = -\Big(p(x) \cdot ln(q(x)) + (1 - p(x)) \cdot ln(1 - q(x))\Big)$$

Una vez se tiene el vector H, los valores se pueden agregar de distintas formas dependiendo de la implementación. Dos formas comunes son promediarlos y sumarlos.

En el caso del ejemplo, supongamos que el ejemplo pertenece realmente a las clases B y C, su vector de distribución binario real se vería como $[0.0, 1.0, 1.0]$. Luego, el vector H se vería de la forma:

$$H = [-ln(1 - 0.6),\ -ln(0.9),\ -ln(0.1)] = [0.9,\ 0.1,\ 2.3]$$

Este vector nos indica, para cada clase, que tan lejos estuvo la predicción del valor real, y esperamos que el modelo que estemos entrenando sea capaz de adecuar sus parámetros para que estos valores se acerquen lo más posible a 0.

2.4.4. Etapa de entrenamiento

La etapa de entrenamiento es aquella en la que el modelo lee los datos que le entregamos y adecúa sus parámetros para aprender características de estos.

El entrenamiento se lleva a cabo iterativamente. Primero el modelo lee los datos que le entregamos y realiza una predicción determinada para la tarea que lo estemos entrenando. Con la respuesta real para cada uno de los ejemplos, calculamos la función de pérdida como explicamos en la sección anterior. Finalmente el modelo, a través de un proceso llamado descenso de gradiente, adecúa sus parámetros para disminuir el valor de pérdida en futuras predicciones.

Algo que se hace para que el modelo pueda efectivamente aprender de los datos es presentarle los datos varias veces. Al ciclo en el que el modelo lee todos los datos con los que

será entrenado le llamamos época. Se espera que mientras más épocas un modelo entrene, más características aprenda de los datos. En la Sección 2.5 explicamos por qué querríamos limitar las épocas que entrena un modelo a un número no tan elevado.

2.4.5. Redes Neuronales Recurrentes

Anteriormente presentamos a grandes rasgos el funcionamiento de las redes neuronales *feed forward*, dijimos que en esta arquitectura, el output de una capa se pasa como input a la siguiente y así, luego de pasar por todas las capas la red entrega su output.

Una de las principales desventajas de las redes *feed forward* es la baja capacidad de trabajar con inputs de tamaño variable. En su lugar, el input de las redes *feed forward* está definido por el tamaño de su capa de entrada, y este no es variable.

Esta baja capacidad de tratar con inputs de largo variable nos impide, por ejemplo, procesar de manera eficiente secuencias de texto, que pueden tener 2 o 100 palabras. Intentos pueden realizarse de truncar o agregar *padding* a las secuencias para que tengan un largo fijo, y pasar esta secuencia modificada por una red *feed forward* pero los resultados no serán los mejores.

Las redes recurrentes [7] son un tipo de red neuronal profunda utilizadas para procesar secuencias. Dichas secuencias contienen elementos de un mismo tipo y pueden ser de largo variable. Dos ejemplos típicos de secuencias son los videos, que se estructuran como secuencias de imágenes, y el texto que se estructura como secuencia de palabras.

Para su funcionamiento, las redes neuronales recurrentes poseen ciclos en su computación, en los que las capas de la red se realimentan a si mismas, a diferencia de las redes *feed forward*. Esto permite que la información procesada con un elemento de la secuencia tenga incidencia en el procesamiento del siguiente elemento. Para nosotros como humanos tiene sentido que, al procesar texto, la información que obtuve en la palabra anterior me sirva para obtener información de la palabra actual.

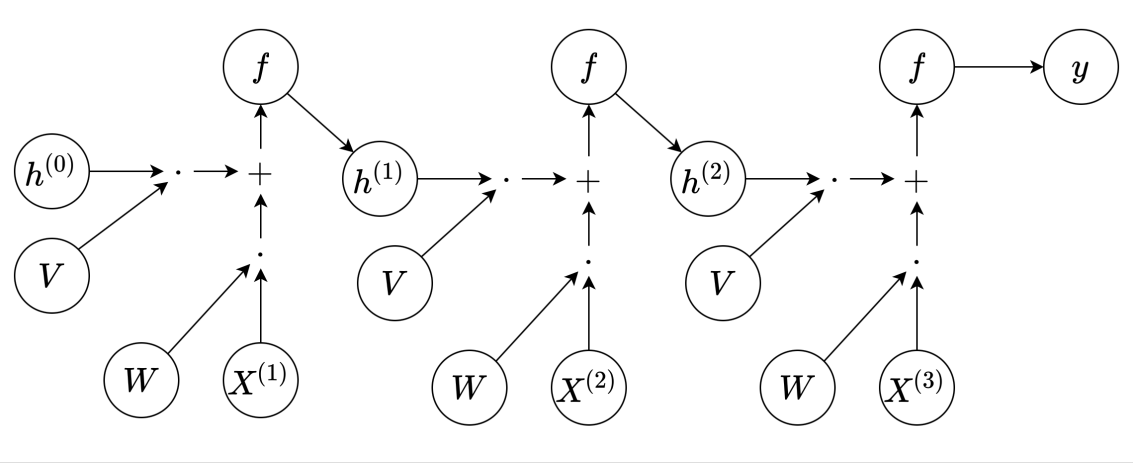


Figura 2.3: Funcionamiento de una red neuronal recurrente

Esta retroalimentación se hace utilizando el output de una capa como input de si misma

en el siguiente paso de la computación. Como se muestra en la Figura 2.3, h es el output de procesar cada elemento de la secuencia de input, este output es entregado como input a la misma capa cuando se procesa el siguiente elemento en la secuencia. Notar que W y V son siempre el mismo, por lo que estos parámetros ponderarán todos los elementos de la secuencia y se adecuarán en la etapa de entrenamiento de acuerdo a esto.

En general, el output de una capa recurrente es:

$$h^{(i)} = f(X^{(i)} \cdot W + h^{(i-1)} \cdot V + b)$$

Aquí nuevamente hemos agregado el vector b que le permite a la red agregar un valor escalar al output.

2.4.6. Dependencias de largo plazo

Como explicamos anteriormente, las redes neuronales recurrentes abordan el problema que tienen las redes tradicionales de no poder procesar secuencias de largo variable de manera efectiva. El funcionamiento de las capas de una red neuronal recurrente se basa en retroalimentarse a si mismas luego de procesar cada elemento de la secuencia de input. Esta retroalimentación causa que el resultado de procesar la información para un elemento de la secuencia, pueda influir en el procesamiento de elementos a procesar posteriormente.

El problema que tienen las redes neuronales recurrentes es que si bien son capaces de "enviar información" hacia adelante en el procesamiento de la secuencia, no tienen como manejar lo que llamamos dependencias de largo plazo (*long-term dependencies*). Con esto nos referimos a cuando el procesamiento de un elemento de la secuencia requiere utilizar información vista en un elemento que se proceso hace varios pasos.

Si estamos intentando predecir la siguiente palabra en un texto, podríamos tener los siguientes dos ejemplos:

1. "El pasto es de color"
2. "He vivido toda mi vida en Francia. Entré a la universidad de los 18 años y me gradué con 24. Me gusta leer y escuchar música. El idioma que más manejo es"

En el primer ejemplo, esperaríamos que para una red neuronal recurrente sea sencillo aprender que muy probablemente la siguiente palabra sea "verde", debido a que toda la información necesaria para inferir esto se encuentra pocas palabras atrás, en este caso decimos que existe una dependencia de corto plazo. Sin embargo, en el segundo ejemplo, la red necesitaría de alguna forma tener la información del país de origen de la persona, encontrado en la primera oración, al estar procesando la última, en este caso decimos que existe una dependencia de largo plazo. Las redes neuronales recurrentes tienen una gran dificultad para manejar estas dependencias [8, 9].

2.4.7. LSTM

La arquitectura LSTM [10] (Long short-term memory) es un tipo de red neuronal recurrente diseñada para poder manejar dependencias de largo plazo.

Para poder manejar las dependencias de largo plazo, la red LSTM mantiene el esqueleto de las redes neuronales recurrentes tradicionales en el que un módulo de la red se realimenta a si mismo. Sin embargo, en las redes LSTM, en lugar de ser una sola capa la que se realimenta a si misma, es una estructura más compleja compuesta de varias capas, a esta estructura le llamamos celda LSTM.

Una celda LSTM recibe tres elementos como input y arroja dos como output, dos de los inputs son precisamente los dos outputs del procesamiento del elemento anterior. A continuación listamos los tres inputs recibidos por una celda LSTM:

- Estado de celda (*Cell state*): Información que la red guarda a lo largo del procesamiento de la secuencia. En cada iteración del procesamiento se puede modificar este estado acorde a lo que la red aprenda como necesario. Se puede interpretar como una memoria en la que la red guarda información para el futuro.
- Estado oculto (*Hidden state*): Resultado de haber procesado el elemento anterior de la secuencia.
- Elemento actual de la secuencia: Elemento de la secuencia a procesar en la iteración actual.

Los outputs de una celda LSTM son los siguientes:

- Nuevo estado de celda: Información guardada actualizada según necesidades de la red.
- Nuevo estado oculto: Resultado de haber procesado el elemento actual de la secuencia.

El procesamiento de un elemento por parte de una celda LSTM se puede separar en 3 etapas:

1. La celda decide cuáles partes del estado de celda borrará y cuáles no.
2. La celda decide qué información nueva de la leída en el input guardará en su estado de celda.
3. En función del estado de celda actualizado, el estado oculto anterior y el elemento de la secuencia leído actualmente, la celda calcula su nuevo estado oculto.

Cada estado oculto calculado por la celda LSTM se entrega finalmente como output. Este estado oculto tiene un tamaño definido al implementar la arquitectura. Dicho tamaño puede ser distinto al tamaño de los elementos del input.

Finalmente, una red LSTM entrega como output una secuencia del mismo tamaño que la secuencia de entrada. Cada elemento de esta secuencia de output es el estado oculto calculado al procesar el elemento análogo del input.

En ciertas tareas, como por ejemplo clasificar la secuencia de input, lo que hacemos es agregar los elementos de la secuencia de output de la red LSTM para obtener un solo elemento como output. Agregaciones típicas son calcular el promedio o el máximo de todos los elementos, de esta forma, independiente del largo de la secuencia de input, siempre obtenemos un elemento del tamaño del estado oculto como output. Este elemento lo interpretamos como una representación de la secuencia de input, y luego podemos, por ejemplo, entregárselo a una red neuronal *feed forward* para realizar una clasificación sobre la secuencia de input.

2.4.7.1. BiLSTM

La arquitectura BiLSTM (*Bidirectional Long short-term memory*) es un tipo de red neuronal recurrente utilizada, al igual que las redes LSTM, para procesar secuencias de largo variable sin tener que enfrentarse al problema de las dependencias de largo plazo.

La estructura de las redes BiLSTM consiste en dos redes LSTM que procesan la secuencia de input a la vez. Una de las redes LSTM utilizadas procesa el input en una dirección, mientras que la otra lo procesa en la dirección opuesta.

Como explicamos anteriormente, una red LSTM entrega como output una secuencia del mismo largo que la secuencia original. En consecuencia, una red BiLSTM entrega como output una secuencia de dos veces el largo de la secuencia original.

Al igual que con las redes LSTM, en tareas como clasificar la secuencia de entrada, nos interesa obtener una representación de la secuencia de entrada para utilizarla luego en otro proceso. En estos casos podemos agregar la secuencia de output de la red BiLSTM de la misma forma que para la LSTM y entregarle la representación final a una red *feed forward* para realizar la clasificación.

2.4.8. Transformers y Atención

Como explicamos anteriormente, las redes neuronales recurrentes tienen gran dificultad para enfrentarse a las dependencias de largo plazo en una secuencia. Además de las redes LSTM, existe un proceso llamado atención neuronal [11, 12] que se puede utilizar para enfrentar este problema.

A través del proceso de atención neuronal, le permitimos a una red neuronal recurrente aprender, durante el entrenamiento, qué partes del texto son más importantes para llevar a cabo la tarea para la que fue implementado. De esta forma, una red neuronal puede aprender a no leer ciertas partes del input si no le son de utilidad.

Aquí entran en juego los transformers [13]. Estos son un tipo de red neuronal recurrente que, utilizando solamente mecanismos de atención neuronal, deciden en cada paso del procesamiento de la secuencia cómo afecta el resto del input a la parte leída actualmente. De esta forma, los transformers no utilizan los mecanismos de retroalimentación que hemos explicado anteriormente y leen toda la secuencia a la vez, aprendiendo que partes de esta son más

importantes. Al leer toda la secuencia de input a la vez, los transformers no necesitan leer la secuencia de izquierda a derecha o derecha a izquierda como una red neuronal recurrente que no utiliza solamente mecanismos de atención, esto hace que sean más eficientes en tiempos de computación.

2.4.9. BERT

BERT [14] es una arquitectura de *Deep Learning* que utiliza el mecanismo de atención de los *Transformers* para generar un modelo del lenguaje (*language model*). Un modelo del lenguaje es, sin entrar en mayor detalle, una distribución de probabilidad sobre secuencias de palabras, lo que permite estimar qué tan probable es que una secuencia de palabras aparezca en una oración. Los modelos del lenguaje se usan altamente en tareas que requieren entregar otra secuencia de texto como output, como *Machine Translation* o *Question Answering*.

Luego de generar el modelo en un proceso llamado *pre-training*, BERT puede pasar por un proceso llamado *fine-tunning*, consistente en otro proceso de entrenamiento para realizar tareas específicas.

Como input, BERT puede recibir uno o dos fragmentos de texto, dependiendo de la tarea que se quiera resolver. Además, al input es necesario agregarle dos palabras adicionales. La primera palabra a agregar es "<CLS>", esta se agrega antes del primer fragmento. La segunda palabra a agregar es "<SEP>", esta se agrega al final del primer fragmento. Si es que la tarea a resolver recibe como input solo un fragmento de texto, "<SEP>" representa el final del input. Si en cambio recibe dos fragmentos, "<SEP>" representa el paso de un fragmento al segundo.

En el caso de este proyecto en el que buscamos clasificar fragmentos de texto, nuestro input consiste en un solo fragmento, al cuál le agregamos ambas palabras especiales al comienzo y al final para procesarlo.

De manera similar a una red neuronal LSTM, BERT retorna como output una secuencia del mismo tamaño que la secuencia de input, incluyendo las dos palabras adicionales agregadas. Cada elemento de la secuencia representa el estado oculto generado al procesar el elemento análogo del input. Dependiendo de la tarea para la que se esté entrenando el modelo, la secuencia de output se puede utilizar de distintas formas. Para este proyecto nos interesa clasificar, y lo que hacemos es utilizar el estado oculto generado al procesar la palabra agregada "<CLS>", a esta palabra le llamamos *token* de clasificación (*classification token*) e interpretamos el estado oculto generado desde su procesamiento como una representación del fragmento de texto. Este estado oculto se lo entregamos luego a una red neuronal *feed forward* para realizar la tarea de clasificación, de la misma manera que lo hacíamos con la red LSTM.

Al momento de su presentación en 2018, BERT alcanzó resultados estado del arte en múltiples tareas de procesamiento de lenguaje natural (NLP) incluyendo *Question Answering* y *Natural Language Inference*.

2.4.9.1. BETO

La implementación de BERT presentada en 2018 fue pre entrenada con textos en inglés, y por lo tanto las tareas para las que puede ser utilizado reciben como input texto en dicho idioma.

En el caso de este proyecto, necesitamos clasificar textos escritos en español. Para esto, utilizamos un modelo basado en BERT, llamado BETO [15], pre entrenado con textos en español.

2.5. Conjuntos de Entrenamiento, Test y Validación

Una pregunta que es importante responder es cómo utilizamos los datos que tenemos a la hora de implementar un modelo de *Machine Learning*. En principio podríamos creer que lo mejor es simplemente utilizar todos los datos que poseemos para entrenar nuestro modelo. Sin embargo, esta no es una alternativa viable para que nuestro modelo funcione efectivamente.

Siempre que implementamos un modelo de *Machine Learning* para resolver un problema, lo que nos interesa es que nuestro modelo generalice lo mejor posible. Con generalizar nos referimos a que el modelo funcione bien para datos que nunca vio en su etapa de entrenamiento.

Si utilizáramos todos los datos que tenemos en el entrenamiento, no tendríamos como evaluar el desempeño del modelo para datos desconocidos, ya que efectivamente no tendríamos ningún dato desconocido para el modelo. Es por esto que elegimos un subconjunto de los datos para no entregárselos al modelo en la etapa de entrenamiento. A este subconjunto le llamamos conjunto de *test*.

Así, nos quedamos con dos conjuntos, uno de entrenamiento que serán los datos que le entregamos al modelo en la etapa de entrenamiento, y otro de *test*, que serán los datos que el modelo no verá en esta etapa.

Lo que hacemos es que una vez que tenemos nuestro modelo entrenado, evaluamos su desempeño utilizando los datos de *test*. Con esto podemos simular predicciones para datos nunca antes vistos por el modelo.

Para que la evaluación con los datos de *test* sea lo suficientemente buena, es importante que estos datos sean representativos del universo de los posibles ejemplos. Para esto, es bueno elegir el conjunto de *test* como una muestra aleatoria de los datos. Existen técnicas más sofisticadas para elegir buenos conjuntos de *test*, especialmente cuando, por ejemplo, la distribución de clases está muy desbalanceada.

Además de lo anterior, surge otro problema. Si nuestro modelo tiene la capacidad suficiente, será capaz de "aprenderse de memoria" los datos de entrenamiento si es que es entrenado lo suficiente con estos. Con esto nos referimos a que el modelo aprenderá características demasiado específicas al conjunto de entrenamiento que en realidad no generalizan correctamente para el resto de los datos. A esto le llamamos sobreajuste u *overfitting*.

Para evitar el sobreajuste lo que hacemos es generar otro subconjunto de los datos. A este nuevo subconjunto le llamamos conjunto de validación. Lo que hacemos con este conjunto es que durante la etapa de entrenamiento, calculamos la función de pérdida para estos datos, pero no adecuamos los parámetros del modelo para disminuir este valor.

Lo que esperaríamos que ocurra al tener un conjunto de entrenamiento y uno de validación, es que al principio del entrenamiento, las pérdidas de los conjuntos de entrenamiento y validación fuesen bajando, puesto que el modelo se está adecuando para aprender las características de los datos desde el conjunto de entrenamiento, pero que en alguna época del entrenamiento, la pérdida de validación comience a subir mientras la de entrenamiento siga bajando. Cuando ocurra esto decimos que el modelo comenzó a sobreajustarse, puesto que se adecuó demasiado a los datos de entrenamiento y ya no puede generalizar tan bien como antes.

Luego de encontrar la época en la que el modelo comienza a sobreajustarse, elegimos como modelo final aquel que fue entrenado hasta dicha época. Finalmente, evaluamos el desempeño de este modelo escogido utilizando los datos de *test*.

La mayoría de los datos suelen estar en el conjunto de entrenamiento, luego una parte menor están en los conjuntos de validación y *test*. Para este proyecto trabajamos con una partición de 70 % datos de entrenamiento, 10 % datos de validación y 20 % datos de *test*.

2.6. Regularización y Dropout

La regularización es una técnica utilizada sobre redes neuronales para evitar que estas se sobreajusten. Existen varios tipos de regularización, pero en este proyecto utilizamos una llamada Dropout.

Dropout [16] es una forma de regularización en la que se "apagan" u omiten neuronas de manera aleatoria en una red neuronal durante el proceso de entrenamiento.

A través de la omisión de neuronas de manera aleatoria en la etapa de entrenamiento, se espera que la red neuronal tenga una mayor dificultad para sobreajustar sus parámetros para los datos de entrenamiento al no tener disponible toda su capacidad a diferencia de cuando se esté utilizando con otros datos, en los que las neuronas no se omitirán.

2.7. Métricas de Evaluación

Previamente hemos hablado de evaluar el desempeño de los modelos que implementamos pero hasta ahora no hemos explicado como hacer esto.

Para evaluar que tan bien o mal funciona nuestro modelo existen distintas métricas que reflejan distintas nociones de correctitud. Dependiendo del problema que estemos resolviendo, buscaremos optimizar una u otra métrica de desempeño.

Cuando se trata de problemas de clasificación binaria, lo que hacemos es llamarle a una clase la clase positiva y a la otra la negativa. Por ejemplo, si estamos clasificando *emails* entre deseado y no deseado, podemos decir que la clase de los deseados son los positivos.

Luego, si realizamos predicciones sobre un conjunto de datos utilizando un modelo implementado tendremos 4 tipos de predicciones:

1. Verdaderos positivos: Aquellos ejemplos que el modelo predijo como positivos y efectivamente lo eran.
2. Falsos negativos: Aquellos ejemplos que el modelo predijo como negativos y eran positivos.
3. Falsos positivos: Aquellos ejemplos que el modelo predijo como positivos y eran negativos.
4. Verdaderos negativos: Aquellos ejemplos que el modelo predijo como negativos y efectivamente lo eran.

Estos resultados los podemos visualizar en una matriz a la que llamamos matriz de confusión.

		Predicción	
		Positivos	Negativos
Real	Positivos	Verdaderos Positivos (VP)	Falsos Negativos (FN)
	Negativos	Falsos Positivos (FP)	Verdaderos Negativos (VN)

Figura 2.4: Matriz de confusión

Con estas nociones, podemos definir distintas métricas para un conjunto de predicciones.

2.7.1. Accuracy

Accuracy mide cuantos ejemplos el modelo predijo correctamente del total de ejemplos. Se calcula como:

$$Accuracy = \frac{VP + VN}{VP + VN + FP + FN}$$

No es una buena métrica para utilizar cuando hay una disparidad significativa entre la cantidad de positivos y negativos en los datos. Por ejemplo, si hay 90 positivos y 10 negativos, el clasificador podría predecir que todos los ejemplos son positivos y obtendría un *Accuracy* de 0.9.

2.7.2. Precisión

La precisión mide cuantos de los ejemplos que el clasificador predijo como positivos realmente lo eran. Se calcula como:

$$Precision = \frac{VP}{VP + FP}$$

2.7.3. Recall

El *recall* mide cuantos de los positivos reales fueron predichos correctamente por el modelo. Se calcula como:

$$Recall = \frac{VP}{VP + FN}$$

Para evaluar el desempeño del modelo, solemos fijarnos en la precisión y el *recall* que este alcanza a la vez. Por lo general, es difícil obtener puntajes altos en ambas métricas en conjunto puesto que para aumentar la precisión el modelo suele disminuir su *recall* y viceversa. Ante esto, dependiendo del problema podemos priorizar una métrica por sobre otra y optimizar esta.

2.7.4. El Problema de la Clasificación Multietiqueta

Hasta ahora hemos hablado de métricas para clasificadores que predicen una de dos clases, positivo o negativo. Cuando tenemos un clasificador multietiqueta esto se vuelve más complicado debido a que no podemos calcular una sola precisión o un solo *recall* para el modelo.

Es difícil encontrar un valor numérico que refleje realmente que tan bueno es el desempeño de un clasificador multietiqueta.

El problema de clasificación multietiqueta se puede tratar como varios problemas de clasificación binaria en paralelo, uno por cada etiqueta. Ante esto, podemos calcular, dado un conjunto de predicciones, precisión y *recall* para cada etiqueta. Si el ejemplo pertenece a una clase en particular, lo consideramos positivo para esa etiqueta, si no, lo consideramos negativo.

2.7.5. Macro/Micro Average por etiqueta

Para entender como combinar las distintas métricas de las clases en un solo valor consideremos el siguiente ejemplo de una matriz de confusión para varias clases:

		Predicción		
		Clase A	Clase B	Clase C
Real	Clase A	4	2	2
	Clase B	2	2	1
	Clase C	1	1	6
	Precisión	0.57	0.40	0.66
	Recall	0.50	0.40	0.75

Figura 2.5: Matriz de confusión de ejemplo para un problema multiclase

En la matriz de confusión de la Figura 2.5 hemos agregado en cada columna la precisión y el *recall* por cada clase.

Lo que usaremos para obtener un valor agregado para la precisión y el *recall* será el *macro* y el *micro average* de cada métrica.

El *macro average* se calcula como el promedio simple de las métricas de cada clase. En el caso del ejemplo:

$$Precision_{macro} = \frac{0.57 + 0.40 + 0.75}{3} = 0.57$$

$$Recall_{macro} = \frac{0.50 + 0.40 + 0.66}{3} = 0.52$$

Por otra parte, para calcular el *micro average* calculamos las métricas para todas las etiquetas juntas. En el caso de la precisión, consideramos como verdaderos positivos todas las predicciones que sean verdaderos positivos para alguna etiqueta, es decir, la diagonal de la matriz de confusión. A su vez, consideramos como falsos positivos todas las predicciones que sean falsos positivos para alguna etiqueta, en este caso, todas las predicciones que no están en la diagonal son falsos positivos para alguna etiqueta.

Análogamente a lo anterior, para calcular el *micro average* del *recall* consideramos nuevamente los verdaderos positivos como todas aquellas predicciones que sean verdaderos positivos para alguna etiqueta. A su vez, los falsos negativos son todas las predicciones incorrectas, puesto que cada una es un falso negativo para alguna etiqueta.

Utilizando lo anterior, tenemos que para el ejemplo de la Figura 2.5, los valores de *micro average* de precisión y *recall* son los siguientes:

$$Precision_{micro} = \frac{12}{12 + 3 + 3 + 3} = 0.57$$

$$Recall_{micro} = \frac{12}{12 + 3 + 3 + 3} = 0.57$$

2.7.6. Coeficiente de Jaccard

La última métrica que veremos para evaluar el desempeño de los modelos multietiqueta que implementamos es el Coeficiente de Jaccard. Esta métrica mide el grado de similitud entre dos conjuntos.

Dados dos conjuntos A y B, el coeficiente de Jaccard se calcula como:

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

El coeficiente de Jaccard también es conocido como *intersection over union* y mide que tan similar es la intersección de dos conjuntos con la unión de estos.

Si dos conjuntos son iguales, la intersección y unión de estos también lo será y por lo tanto el coeficiente de Jaccard será 1. Si por el contrario dos conjuntos son completamente distintos y no comparten ningún elemento, la intersección será vacía y por lo tanto el coeficiente de Jaccard será 0.

Esto lo aplicamos en nuestro caso para comparar que tan similar es el conjunto de las clases reales de un ejemplo con las clases predichas por los modelos implementados.

Capítulo 3

Desarrollo

3.1. Datos

Luego de haber acordado llevar a cabo el proyecto con la ACHS, tuvimos que acordar como tendríamos acceso a los datos con los que trabajaríamos. Dichos datos consisten en relatos médicos asociados a un accidente que le ocurrió a un trabajador. Muchos de los relatos contienen información delicada respecto al trabajador que sufrió el accidente, esta información puede ser el nombre, el rut, el email, etc. Para resguardar la privacidad de los pacientes de la ACHS, fue necesario firmar un acuerdo de confidencialidad con dicha entidad.

El acceso a los datos finalmente se nos dio a través de servidores propios de la ACHS, de los cuales no tuvimos autorización en un principio para descargar los datos. Estos servidores funcionan a través de Databricks, un servicio de Microsoft que permite trabajar con clusters y notebooks similares a Jupyter.

En un principio, tuvimos que llevar a cabo el proyecto en las máquinas provistas por la ACHS. Para esto, fue necesario pedirle a la contraparte que nos diera acceso a GPUs, hardware necesario para llevar a cabo los experimentos planteados. Las GPUs a las que tuvimos acceso en un principio no eran de la gama que esperábamos, por lo que los experimentos demoraron más tiempo del presupuestado. Luego de negociar con la ACHS, obtuvimos autorización para descargar los datos y utilizar servidores propios donde tenemos acceso a GPUs de gama más alta.

Antes de realizar los experimentos propuestos con clasificadores, realizamos un preprocesamiento de los datos. Estos nos fueron entregados en un archivo de texto plano, sin un ordenamiento claro, y fue necesario procesarlos para que pudieran ser entregados a los clasificadores luego implementados.

3.2. Preprocesamiento de los datos

Los datos que nos fueron entregados venían en un archivo de texto plano. Este poseía en la primera fila los nombres de las 10 columnas que representan un dato, separados por el caracter ";".

El resto del archivo pretendía mantener este formato, sin embargo, se encontraban errores varios, como saltos de linea arbitrarios o el caracter ";" inserto en el texto, lo que sería después un problema a la hora de leer los datos. En total el archivo de texto sin ningún procesamiento poseía alrededor de 12 millones de líneas, sin embargo, este número no nos decía mucho, puesto que no significaba que hubiera esa cantidad de relatos.

Para procesar y ordenar los datos, utilizamos Python. Los procesos que realizamos sobre los datos se describen a continuación:

1. Dejar todo lo que perteneciera a la misma tupla en una misma línea.
2. Reemplazar espacios en blanco múltiples seguidos por uno solo.
3. Remover tabulaciones.
4. Remover comillas repetidas del estilo ""*<Oración>*"" y reemplazarlas por una sola.
5. Remover caracteres ";" que no fueran una separación de dos columnas.

Luego, los datos quedaron listos para ser manipulados y analizados. Para esto, utilizamos la librería Pandas de Python, que nos permitió cargar los datos a un Dataframe y realizar un análisis exploratorio de estos.

3.3. Análisis previo de los datos

Teniendo los datos listos para ser trabajados, observamos que hay 857.731 tuplas, representando a 404.756 accidentes distintos, con 10 columnas cada una. Estas son descritas a continuación:

1. Siniestro: Identificador del evento al cual corresponde un relato. No es único, pues para un mismo evento pueden haber varios encuentros con un médico.
2. Id_Tipo_Documento: Identificador del tipo de documento al que refiere la tupla. Algunos ejemplos son MEDHCHOSP, que representa un ingreso hospitalario, o MEDCONTROL, que representa un control de chequeo con el paciente. En total son 11 valores distintos.
3. Fecha_Registro_PMD: Fecha en la que un médico generó un registro en el documento.
4. Fecha_Creado_Ndoc: Fecha de creación del documento.
5. Fecha_Modificado_Ndoc: Fecha de última modificación del documento.
6. Fecha_Modificado_Texto: Fecha de última modificación del texto.
7. Responsable_Texto: Identificador de persona responsable de haber escrito el texto. En total son 1.052 valores distintos.
8. Anamnesis: Datos que se obtienen del relato del paciente.

9. Examen_Fisico: Descripción de resultados de examen físico realizado al paciente.

10. Indicacion_Medico: Indicación entregada por el médico al paciente en la sesión.

Las columnas que más nos interesan son Siniestro, que identifica al accidente ocurrido y las 3 últimas: Anamnesis, Examen_Fisico e Indicacion_Medico, que contienen el texto que nos interesa a la hora de clasificar.

Para tener una idea más clara de los datos con los que trabajamos, realizamos el cálculo de la cantidad de veces que aparece cada palabra en los datos, considerando las 3 columnas de texto mencionadas previamente y sin contar stopwords. En total se utilizan alrededor de 500 mil palabras distintas. En la Figura (3.1) se muestran los resultados obtenidos.

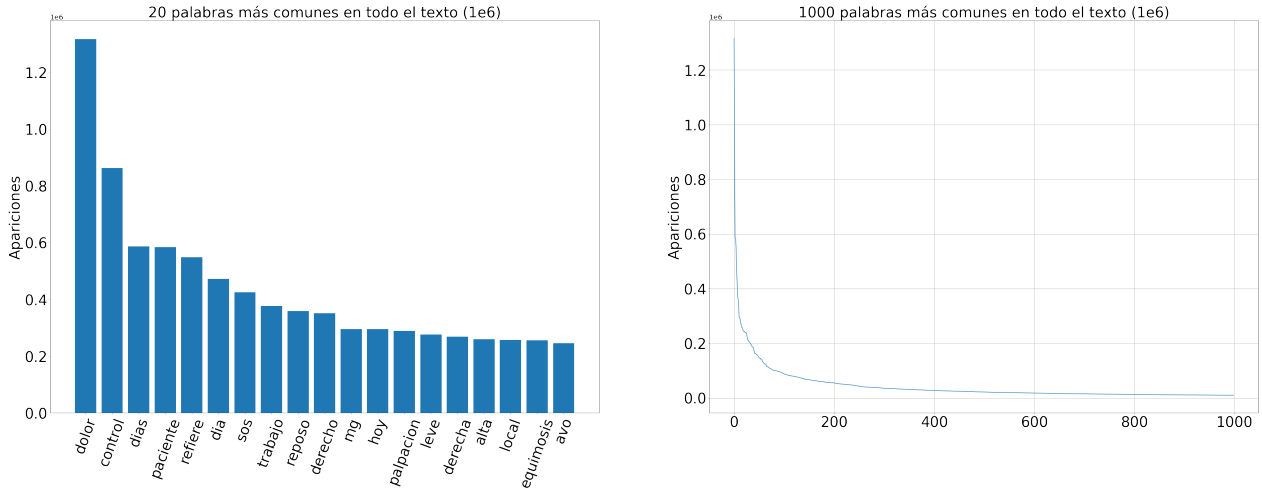


Figura 3.1: Palabras más comunes en los datos. A la izquierda las 19 más comunes, a la derecha la distribución de las 1000 más comunes

Con respecto a la distribución, los resultados se aproximan a lo que nos dice la ley de Zipf, ley empírica que describe la distribución que tienen las palabras en un idioma. Según esta, la distribución se debería aproximar a $\frac{1}{n^a}$, donde n es la posición que ocupan las palabras en orden decreciente de apariciones y a un exponente, por lo general, ligeramente superior a 1.

Como es de esperarse, la mayoría de las palabras que se encuentran entre las 19 más utilizadas tienen relación estricta con un contexto médico, tales como “dolor”, “paciente”, “reposo” o “equimosis”.

Haciendo este análisis también nos encontramos con que hay muchas palabras escritas incorrectamente. Esto no es raro considerando el contexto en el que se tipean estos relatos.

3.4. Etiquetas

Una vez tuvimos acceso a las etiquetas que corresponden a las distintas categorías en las que un relato médico puede ser clasificado, realizamos una caracterización de estas para entenderlas mejor.

En total existen alrededor de 3000 etiquetas distintas. Estas poseen un código y un texto asociado. El texto es el nombre del diagnóstico, por ejemplo "CONTUSION DE RODILLA MODERADA" o "QUEMADURA DE MANO Y MUÑECA SEGUNDO GRADO". El código es un identificador númerico único, por ejemplo 7786 o 6690.

Las palabras más comunes entre las etiquetas se muestran en la nube de palabras de la Figura 3.2.

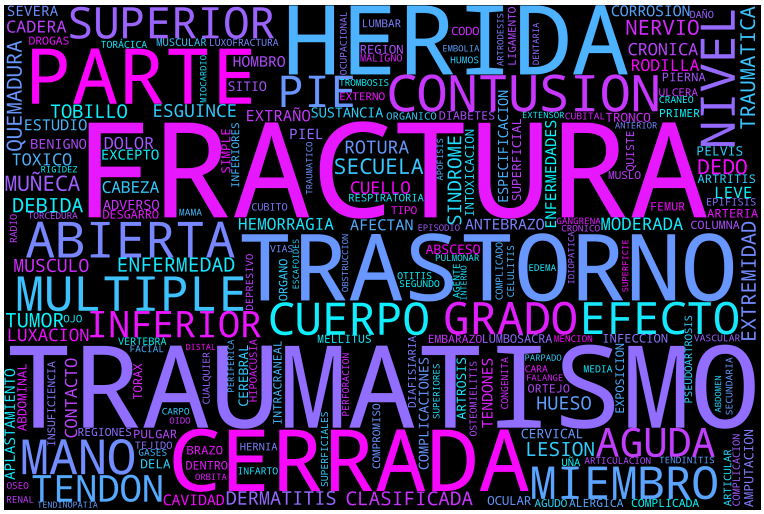


Figura 3.2: Nube de palabras para las etiquetas.

También quisimos verificar cuantos relatos asociados tiene cada etiqueta. En la Figura 3.3 se muestra este resultado para todas las etiquetas. En la Figura 3.4 se muestra el mismo resultado para las primeras 100 etiquetas.

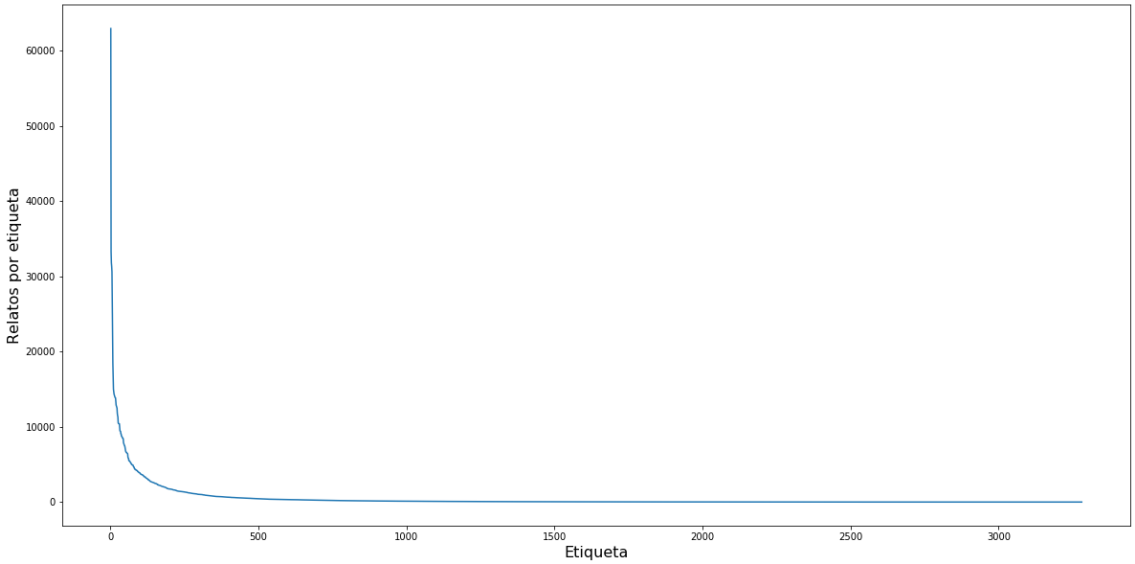


Figura 3.3: Cantidad de relatos asociados para cada etiqueta. Se muestran todas las etiquetas.

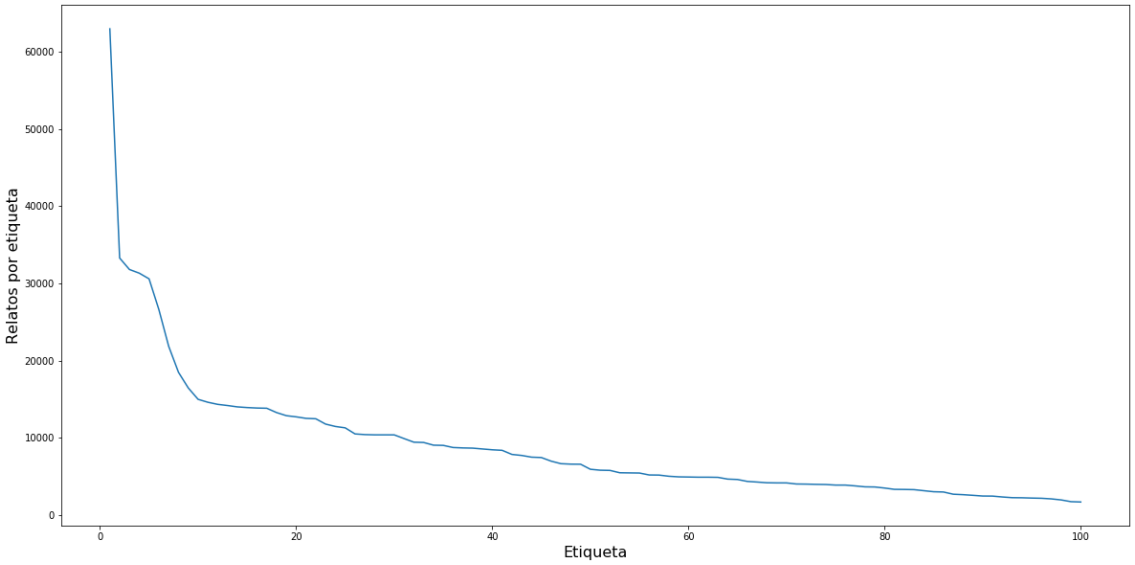


Figura 3.4: Cantidad de relatos asociados para cada etiqueta. Se muestran solo las 100 etiquetas más comunes

Como se puede observar. Llegando a las 100 etiquetas la cantidad de relatos asociados a cada etiqueta ya es cercana a 0. Debido a esto, y a que implementar clasificadores para demasiadas etiquetas es difícil, decidimos implementar clasificadores para las 50 etiquetas más comunes. De esta forma, la cantidad de relatos está alrededor de 460 mil y la etiqueta con menos relatos asociados tiene alrededor de 10000.

3.5. Baselines

A continuación explicamos los dos *baselines* implementados en el proyecto. El primero es un clasificador que utiliza Naive Bayes. El segundo es un clasificador basado en la arquitectura BiLSTM.

3.5.1. Naive Bayes

Para implementar un clasificador multiclase multietiqueta C utilizando el método de Naive Bayes lo que hicimos fue implementar un clasificador binario para cada una de las etiquetas. A cada uno de estos clasificadores lo llamaremos C_i con $i \in [0, 50]$.

Cada clasificador binario C_i recibe como input una representación del relato a la que llamamos *bag of words*. Esta representación mantiene registro de cuales palabras de todas las utilizadas en los datos son utilizadas en el relato representado.

Para explicar el funcionamiento de *bag of words* utilizaremos un ejemplo. Supongamos que nuestros datos contienen en total solo 3 relatos:

1. "paciente siente dolor de cabeza"
2. "dolor de muelas"
3. "dolor de cabeza"

En este caso, las palabras utilizadas en los datos son 6: [paciente, siente, dolor, de, cabeza, muelas]. En consecuencia, cada relato será representado por un vector de 6 valores binarios, en donde cada índice es asociado a una palabra y el vector tendrá un 1 en esa posición si la palabra se encuentra en el relato, de la siguiente forma:

Relato	paciente	siente	dolor	de	cabeza	muelas
1	1	1	1	1	1	0
2	0	0	1	1	0	1
3	0	0	1	1	1	0

Luego, el vector representante de cada relato será:

1. $[1, 1, 1, 1, 1, 0]$
2. $[0, 0, 1, 1, 0, 1]$
3. $[0, 0, 1, 1, 1, 0]$

Notemos que este modelo no tiene consideración por el orden de las palabras y solo mantiene registro de si la palabra se encuentra en alguna parte del relato.

Como mostramos en la Figura 3.1, la gran mayoría de palabras que se encuentran en los relatos aparecen muy pocas veces. Si consideráramos todas y cada una de las palabras

presentes, agregaríamos muchas dimensiones a los vectores representantes de relatos en las que solo unos pocos tendrían valor 1, esta información no es muy valiosa para el modelo y por lo tanto restringimos la frecuencia mínima que debe tener una palabra en los relatos para ser representada en los vectores de *bag of words*. Este valor lo variamos para cada experimento.

Para clasificar un relato, C le entrega su representación en *bag of words* a cada uno de los clasificadores C_i. Estos clasificadores entregarán una respuesta booleana cada uno. Finalmente, la respuesta de C será un arreglo de tamaño 50 donde cada índice i tiene la respuesta del clasificador C_i.

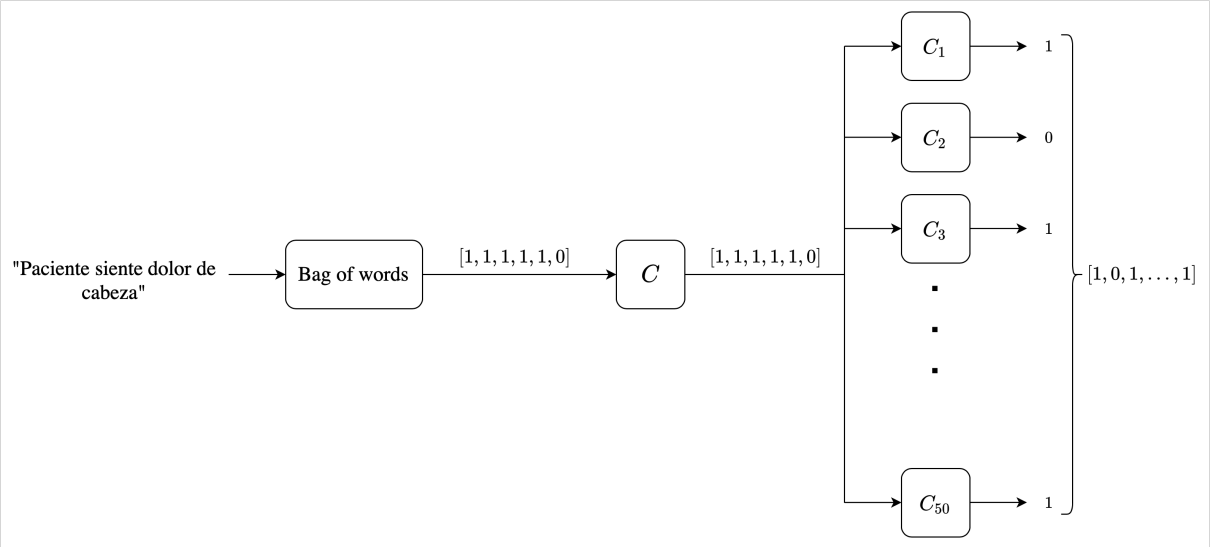


Figura 3.5: Funcionamiento de clasificador multiclase multietiqueta utilizando Naive Bayes

Luego de entrenar el modelo con los datos de entrenamiento, utilizamos el conjunto de *test* para medir el coeficiente de Jaccard, además de los macro y micro *averages* de precisión y *recall* respecto a las respuestas esperadas.

3.5.2. BiLSTM

El segundo *baseline* implementado para el proyecto consiste en una red neuronal basada en la arquitectura BiLSTM.

La primera capa de la red implementada consiste en una capa de embeddings. Los embeddings son una manera de representar texto en un espacio vectorial. A cada palabra del vocabulario utilizado en nuestro texto se le asocia un vector con una dimensionalidad definida en la implementación, en nuestro caso, utilizamos vectores de dimensión 300. La capa de embeddings de la red neuronal recibe como input una secuencia de palabras correspondiente a un relato médico y entrega como output una secuencia de vectores consistentes en los embeddings para cada palabra encontrada en el relato.

La secuencia de output de la capa de embeddings se entrega a una primera capa BiLSTM. Esta capa recibe como input una secuencia de vectores y entrega una nueva secuencia de dos veces el tamaño del input, como explicamos en la Sección 2.4.7.1. La dimensión de los

vectores de output es variable y la modificamos según el experimento.

Una vez que tenemos el output de la primera capa BiLSTM, este se puede enetregar a una segunda capa de la misma arquitectura. Análogamente, el output de esta se puede entregar a una tercera capa, y así sucesivamente. La cantidad de capas BiLSTM presentes en la red neuronal es variable y se define en cada experimento.

La secuencia de vectores entregada por la última capa BiLSTM de la red neuronal, es agregada a través de una función de máximo. Esta función calcula el máximo, punto a punto, de todos los vectores y finalmente entrega un vector de la misma dimensión que cada uno de los elementos de la secuencia.

El vector entregado por la función de máximo, que interpretamos como una representación del relato que estamos procesando, lo pasamos por una red neuronal *feed forward* con una cantidad de capas variable que finalmente, entrega un vector de dimensión 50, consistente en la cantidad de categorías posibles para un relato. Cada valor de este vector lo interpretamos como la probabilidad de que el relato pertenezca a cierta categoría.

Finalmente, calculamos la entropía cruzada binaria utilizando el vector de output de la red y el vector de la respuesta esperada que tenemos de los datos, para luego ejecutar el algoritmo que ajustará los parámetros de la red neuronal de acuerdo a que tan correcta haya sido su predicción.

Luego de entrenar el modelo con los datos de entrenamiento, utilizamos el conjunto de *test* para medir el coeficiente de Jaccard, además de los macro y micro *averages* de precisión y *recall* respecto a las respuestas esperadas.

3.6. BERT

Luego de haber implementado los dos *baselines* para el proyecto, implementamos el modelo que finalmente buscábamos comparar con dichos *baselines*. Este modelo consiste en una red neuronal basada en BETO, la arquitectura basada en BERT pre entrenada con textos en español.

Antes de procesar las secuencias de texto, como explicamos previamente les agregamos las palabras “<CLS>” y “<SEP>”, necesarias para ser procesadas por la arquitectura utilizada.

De manera similar al *baseline* implementado con BiLSTM, la primera capa de esta red neuronal consiste en una capa de embeddings en donde se construye una secuencia de vectores que representan a la secuencia de palabras de entrada. Esta secuencia luego pasa por 12 capas donde, utilizando mecanismos de atención, se procesa la secuencia.

Como explicamos previamente en la Sección 2.4.9, el output de la arquitectura BERT consiste en una secuencia de vectores, donde cada uno representa a la palabra respectiva en la misma posición de la secuencia de input. Para poder clasificar el relato, utilizamos el vector que representa a la primera palabra “<CLS>”,

El vector que representa a la palabra “<CLS>” lo interpretamos como una representación de la secuencia de input y se lo entregamos a una red neuronal *feed forward*. De aquí en

adelante los pasos son los mismos que utilizamos en la red BiLSTM. La red *feed forward* utilizada entrega como output un vector de dimensión 50, donde cada valor entre 0 y 1 representa la probabilidad de que el relato procesado pertenezca a una categoría determinada. Con este vector de output y la distribución de probabilidad real que tenemos de los datos, calculamos la entropía cruzada binaria para luego ejecutar el algoritmo que ajustará los parámetros de la red neuronal de acuerdo a que tan correcta haya sido la predicción de la red.

Luego de entrenar el modelo con los datos de entrenamiento, utilizamos el conjunto de *test* para medir el coeficiente de Jaccard, además de los macro y micro *averages* de precisión y *recall* respecto a las respuestas esperadas.

3.7. Experimentos

Previamente hemos hablado de valores que son variables y se definen para cada experimento en el momento previo a ejecutarlos. Ejemplos de estos valores son la cantidad de capas BiLSTM a utilizar o la dimensión de output de cada una de estas capas. A estos valores les llamamos hiperparámetros.

A continuación mostramos los experimentos realizados y los hiperparámetros definidos para cada uno.

3.7.1. Naive Bayes

En el caso de nuestro primer *baseline* correspondiente a Naive Bayes, el único hiperparámetro utilizado es la frecuencia mínima que le exigimos tener a una palabra para poder ser representada en los vectores de bag of words, a este valor le llamamos min_freq. Los experimentos que realizamos son los siguientes:

Experimento	min_freq
1	2
2	3
3	5

3.7.2. BiLSTM

El segundo baseline implementado posee varios hiperparámetros. A continuación listamos cada uno con el nombre que le asociamos y una breve descripción:

- batch_size: Número de ejemplos que recibe la red a la vez.
- embedding_dim: Dimensión de los vectores que representan a cada palabra en la capa de embeddings.
- hidden_dim: Dimensión de los vectores de la secuencia de output de las capas BiLSTM.
- lstm_layers: Cantidad de capas BiLSTM a utilizar.
- lr: Learning rate. Este hiperparámetro define que tan agresivo es el ajuste de los parámetros de la red luego de calcular la función de pérdida.
- ff_hidden_layers: Feed forward hidden layers. Define cuantas capas tendrá la red *feed forward* a la que se le entrega el vector representante de la secuencia luego de pasar por las capas BiLSTM.
- ff_hidden_dim: Feed forward hidden dimension. Define la dimensión de output de las capas escondidas de la red *feed forward*.
- min_freq: Define que frecuencia debe tener una palabra en los relatos para ser utilizada. Las palabras que tienen una frecuencia menor son eliminadas de los relatos.
- dropout: Probabilidad de omitir neuronas a la salida de las capas BiLSTM y a la salida de la red neuronal *feed forward*.
- embedding_dropout: Probabilidad de omitir neuronas a la salida de la capa de embeddings.

Los experimentos que realizamos con la red BiLSTM y sus hiperparámetros correspondientes son los siguientes:

Experimento	batch_size	embedding_dim	hidden_dim	lstm_layers	lr	ff_hidden_layers	ff_hidden_dim	min_freq	dropout	embedding_dropout
1	128	300	128	2	0.01	0	150	5	0.0	0.0
2	128	300	128	2	0.01	0	150	5	0.5	0.2
3	128	300	128	2	0.01	2	150	5	0.0	0.0
4	128	300	128	2	0.01	2	150	5	0.5	0.2

3.7.3. BERT

En el caso de la red neuronal basada en BERT, mantenemos la estructura de la arquitectura, sin variar el número de capas o las dimensiones de los outputs. Los hiperparámetros a definir para esta red neuronal son los siguientes:

- batch_size: Número de ejemplos que recibe la red a la vez.
- lr: Learning rate. Este hiperparámetro define que tan agresivo es el ajuste de los parámetros de la red luego de calcular la función de pérdida.

Los experimentos realizados con la red neuronal basada en BERT son los siguientes:

Experimento	batch_size	lr
1	8	1e-5
2	8	2e-5
3	8	3e-5

3.7.4. Experimentos extra

Además de haber realizado los experimentos que explicamos anteriormente, utilizando la red BiLSTM y la basada en BERT hicimos experimentos utilizando las 12 y las 25 etiquetas más comunes en lugar de las 50. El objetivo de estos experimentos es evaluar la efectividad del modelo al tener que clasificar en menos etiquetas, además de eliminar etiquetas que no son tan comunes.

A continuación los hiperparámetros utilizados para estos experimentos.

BiLSTM:

Experimento	batch_size	embedding_dim	hidden_dim	lstm_layers	lr	ff_hidden_layers	ff_hidden_dim	min_freq	dropout	embedding_dropout
12 labels	128	300	128	2	0.01	0	150	5	0.0	0.0
25 labels	128	300	128	2	0.01	0	150	5	0.0	0.0

BERT:

Experimento	batch_size	lr
12 labels	8	2e-5
25 labels	8	2e-5

Capítulo 4

Resultados

A continuación presentamos los resultados de los experimentos realizados para cada modelo implementado.

Para cada experimento presentamos la precisión y el *recall* obtenido por cada etiqueta. Luego mostramos el macro y micro *average* para la precisión y el *recall* de todas las etiquetas. Finalmente, presentamos el coeficiente de Jaccard obtenido en el experimento.

4.1. Naive Bayes

4.1.1. Experimento 1

Etiqueta	Precisión	Recall	Etiqueta	Precisión	Recall
0	0.277	0.789	25	0.185	0.500
1	0.686	0.840	26	0.231	0.480
2	0.239	0.562	27	0.221	0.813
3	0.184	0.924	28	0.199	0.773
4	0.164	0.314	29	0.159	0.748
5	0.205	0.792	30	0.200	0.469
6	0.182	0.760	31	0.273	0.778
7	0.323	0.909	32	0.157	0.562
8	0.197	0.856	33	0.192	0.740
9	0.157	0.436	34	0.129	0.477
10	0.137	0.393	35	0.161	0.664
11	0.145	0.393	36	0.148	0.182
12	0.343	0.596	37	0.219	0.450
13	0.244	0.600	38	0.127	0.820
14	0.164	0.506	39	0.094	0.801
15	0.441	0.617	40	0.185	0.943
16	0.137	0.825	41	0.362	0.963
17	0.111	0.760	42	0.141	0.908
18	0.139	0.817	43	0.244	0.403
19	0.316	0.947	44	0.111	0.818
20	0.636	0.979	45	0.125	0.850
21	0.890	0.975	46	0.155	0.791
22	0.436	0.820	47	0.222	0.901
23	0.590	0.712	48	0.214	0.919
24	0.210	0.855	49	0.400	0.949

	Precisión	Recall
micro avg	0.223	0.777
macro avg	0.248	0.714
Jaccard score	0.270	

4.1.2. Experimento 2

Etiqueta	Precisión	Recall	Etiqueta	Precisión	Recall
0	0.261	0.805	25	0.182	0.518
1	0.686	0.841	26	0.156	0.660
2	0.240	0.583	27	0.183	0.834
3	0.174	0.936	28	0.165	0.805
4	0.114	0.620	29	0.110	0.819
5	0.182	0.850	30	0.143	0.641
6	0.156	0.818	31	0.194	0.822
7	0.293	0.914	32	0.100	0.744
8	0.188	0.865	33	0.132	0.817
9	0.157	0.462	34	0.085	0.688
10	0.135	0.421	35	0.119	0.795
11	0.149	0.435	36	0.089	0.468
12	0.340	0.609	37	0.155	0.673
13	0.241	0.620	38	0.108	0.874
14	0.163	0.532	39	0.086	0.867
15	0.439	0.625	40	0.176	0.955
16	0.126	0.848	41	0.354	0.966
17	0.088	0.816	42	0.131	0.936
18	0.112	0.864	43	0.155	0.650
19	0.313	0.957	44	0.097	0.842
20	0.624	0.982	45	0.108	0.878
21	0.847	0.979	46	0.113	0.867
22	0.235	0.893	47	0.186	0.920
23	0.515	0.732	48	0.200	0.928
24	0.206	0.869	49	0.386	0.953

	Precisión	Recall
micro avg	0.192	0.811
macro avg	0.218	0.776
Jaccard score	0.255	

4.1.3. Experimento 3

Etiqueta	Precisión	Recall	Etiqueta	Precisión	Recall
0	0.248	0.819	25	0.530	0.268
1	0.685	0.841	26	0.105	0.753
2	0.240	0.598	27	0.157	0.846
3	0.168	0.943	28	0.143	0.835
4	0.083	0.758	29	0.083	0.856
5	0.163	0.894	30	0.095	0.767
6	0.135	0.850	31	0.144	0.849
7	0.273	0.919	32	0.067	0.827
8	0.182	0.869	33	0.099	0.860
9	0.157	0.477	34	0.060	0.794
10	0.134	0.445	35	0.092	0.857
11	0.145	0.449	36	0.054	0.694
12	0.336	0.620	37	0.103	0.793
13	0.238	0.634	38	0.097	0.904
14	0.162	0.548	39	0.080	0.909
15	0.435	0.633	40	0.171	0.964
16	0.120	0.864	41	0.350	0.968
17	0.075	0.850	42	0.124	0.947
18	0.095	0.885	43	0.094	0.793
19	0.308	0.960	44	0.088	0.853
20	0.615	0.982	45	0.098	0.888
21	0.800	0.980	46	0.088	0.897
22	0.133	0.923	47	0.165	0.927
23	0.428	0.756	48	0.191	0.933
24	0.203	0.882	49	0.376	0.954

	Precisión	Recall
micro avg	0.167	0.831
macro avg	0.197	0.812
Jaccard score	0.243	

4.2. BiLSTM

4.2.1. Experimento 1

Etiqueta	Precisión	Recall	Etiqueta	Precisión	Recall
0	0.671	0.593	25	0.553	0.185
1	0.775	0.855	26	0.533	0.424
2	0.645	0.485	27	0.557	0.553
3	0.639	0.363	28	0.471	0.437
4	0.519	0.398	29	0.727	0.494
5	0.690	0.303	30	0.589	0.290
6	0.638	0.537	31	0.609	0.623
7	0.618	0.749	32	0.564	0.373
8	0.640	0.577	33	0.511	0.618
9	0.590	0.427	34	0.542	0.329
10	0.483	0.134	35	0.430	0.520
11	0.349	0.144	36	0.510	0.508
12	0.607	0.619	37	0.545	0.379
13	0.470	0.562	38	0.512	0.454
14	0.399	0.519	39	0.647	0.264
15	0.731	0.746	40	0.733	0.483
16	0.444	0.334	41	0.592	0.873
17	0.547	0.416	42	0.687	0.650
18	0.451	0.579	43	0.565	0.447
19	0.407	0.773	44	0.578	0.526
20	0.726	0.893	45	0.525	0.491
21	0.971	0.955	46	0.468	0.507
22	0.948	0.858	47	0.676	0.711
23	0.943	0.603	48	0.630	0.766
24	0.593	0.315	49	0.794	0.876

	Precisión	Recall
micro avg	0.627	0.599
macro avg	0.601	0.530
Jaccard score	0.447	

4.2.2. Experimento 2

Etiqueta	Precisión	Recall	Etiqueta	Precisión	Recall
0	0.594	0.648	25	0.000	0.000
1	0.677	0.875	26	0.553	0.154
2	0.672	0.251	27	0.445	0.644
3	0.413	0.252	28	0.379	0.230
4	0.000	0.000	29	0.595	0.479
5	0.747	0.206	30	0.337	0.175
6	0.555	0.514	31	0.536	0.603
7	0.449	0.893	32	0.327	0.379
8	0.608	0.507	33	0.504	0.521
9	0.701	0.251	34	0.361	0.205
10	0.277	0.173	35	0.373	0.496
11	0.279	0.193	36	0.682	0.243
12	0.489	0.720	37	0.357	0.402
13	0.398	0.598	38	0.417	0.510
14	0.308	0.044	39	0.417	0.010
15	0.501	0.890	40	0.750	0.469
16	0.387	0.159	41	0.530	0.891
17	0.351	0.395	42	0.560	0.697
18	0.422	0.432	43	0.000	0.000
19	0.318	0.948	44	0.379	0.635
20	0.634	0.980	45	0.386	0.519
21	0.937	0.955	46	0.346	0.417
22	0.959	0.354	47	0.532	0.762
23	0.949	0.590	48	0.513	0.782
24	0.275	0.670	49	0.596	0.922

	Precisión	Recall
micro avg	0.511	0.580
macro avg	0.476	0.473
Jaccard score	0.368	

4.2.3. Experimento 3

Etiqueta	Precisión	Recall	Etiqueta	Precisión	Recall
0	0.684	0.554	25	0.514	0.165
1	0.727	0.865	26	0.581	0.373
2	0.674	0.444	27	0.574	0.535
3	0.637	0.319	28	0.467	0.392
4	0.451	0.495	29	0.578	0.559
5	0.709	0.232	30	0.396	0.456
6	0.700	0.478	31	0.628	0.540
7	0.590	0.786	32	0.515	0.360
8	0.693	0.512	33	0.548	0.547
9	0.565	0.415	34	0.450	0.445
10	0.349	0.176	35	0.600	0.221
11	0.318	0.085	36	0.508	0.470
12	0.572	0.675	37	0.568	0.349
13	0.567	0.384	38	0.494	0.491
14	0.389	0.351	39	0.479	0.365
15	0.720	0.739	40	0.794	0.413
16	0.377	0.361	41	0.631	0.835
17	0.618	0.285	42	0.629	0.666
18	0.437	0.561	43	0.659	0.377
19	0.479	0.686	44	0.571	0.509
20	0.720	0.900	45	0.553	0.383
21	0.947	0.953	46	0.452	0.530
22	0.923	0.842	47	0.687	0.679
23	0.898	0.603	48	0.708	0.702
24	0.412	0.473	49	0.741	0.898

	Precisión	Recall
micro avg	0.621	0.578
macro avg	0.590	0.509
Jaccard score	0.431	

4.2.4. Experimento 4

Etiqueta	Precisión	Recall	Etiqueta	Precisión	Recall
0	0.488	0.679	25	0.229	0.091
1	0.714	0.809	26	0.000	0.000
2	0.352	0.069	27	0.447	0.541
3	0.302	0.210	28	0.258	0.544
4	0.000	0.000	29	0.448	0.479
5	0.288	0.498	30	0.230	0.361
6	0.258	0.166	31	0.532	0.541
7	0.456	0.853	32	0.265	0.521
8	0.586	0.423	33	0.458	0.520
9	0.445	0.288	34	0.000	0.000
10	0.250	0.167	35	0.415	0.324
11	0.288	0.155	36	0.510	0.273
12	0.442	0.683	37	0.341	0.312
13	0.337	0.627	38	0.392	0.428
14	0.000	0.000	39	0.000	0.000
15	0.611	0.750	40	0.246	0.738
16	0.321	0.209	41	0.515	0.870
17	0.307	0.409	42	0.558	0.593
18	0.332	0.553	43	0.273	0.460
19	0.317	0.931	44	0.310	0.721
20	0.654	0.967	45	0.334	0.610
21	0.937	0.948	46	0.309	0.535
22	0.918	0.844	47	0.528	0.765
23	0.890	0.598	48	0.508	0.761
24	0.256	0.742	49	0.632	0.892

	Precisión	Recall
micro avg	0.453	0.576
macro avg	0.390	0.489
Jaccard score	0.349	

4.2.5. Experimento 25_labels

Etiqueta	Precisión	Recall
0	0.636	0.655
1	0.811	0.880
2	0.450	0.537
3	0.584	0.668
4	0.643	0.777
5	0.683	0.653
6	0.545	0.473
7	0.681	0.744
8	0.732	0.846
9	0.559	0.451
10	0.590	0.425
11	0.504	0.624
12	0.975	0.967
13	0.964	0.963
14	0.891	0.655
15	0.658	0.565
16	0.446	0.612
17	0.709	0.468
18	0.560	0.410
19	0.647	0.587
20	0.688	0.863
21	0.590	0.514
22	0.564	0.662
23	0.692	0.748
24	0.786	0.893

	Precisión	Recall
micro avg	0.688	0.716
macro avg	0.663	0.666
Jaccard score	0.552	

4.2.6. Experimento 12_labels

Etiqueta	Precisión	Recall
0	0.893	0.895
1	0.858	0.835
2	0.735	0.680
3	0.753	0.740
4	0.899	0.786
5	0.639	0.498
6	0.801	0.606
7	0.746	0.707
8	0.947	0.943
9	0.744	0.640
10	0.781	0.688
11	0.845	0.869

	Precisión	Recall
micro avg	0.825	0.775
macro avg	0.803	0.741
Jaccard score	0.673	

4.3. BERT

4.3.1. Experimento 1

Etiqueta	Precisión	Recall	Etiqueta	Precisión	Recall
0	0.666	0.706	25	0.610	0.307
1	0.789	0.881	26	0.558	0.602
2	0.609	0.572	27	0.620	0.607
3	0.603	0.559	28	0.524	0.484
4	0.519	0.607	29	0.709	0.628
5	0.795	0.391	30	0.514	0.473
6	0.628	0.665	31	0.645	0.669
7	0.671	0.811	32	0.514	0.553
8	0.656	0.671	33	0.720	0.538
9	0.633	0.461	34	0.633	0.408
10	0.362	0.374	35	0.591	0.491
11	0.400	0.264	36	0.651	0.539
12	0.605	0.713	37	0.628	0.488
13	0.564	0.521	38	0.499	0.652
14	0.346	0.687	39	0.514	0.488
15	0.687	0.804	40	0.569	0.687
16	0.522	0.451	41	0.737	0.748
17	0.565	0.539	42	0.710	0.738
18	0.586	0.575	43	0.676	0.520
19	0.627	0.658	44	0.609	0.612
20	0.773	0.917	45	0.575	0.604
21	0.968	0.944	46	0.586	0.610
22	0.660	0.906	47	0.736	0.757
23	0.819	0.430	48	0.706	0.796
24	0.557	0.490	49	0.825	0.901

	Precisión	Recall
micro avg	0.653	0.664
macro avg	0.625	0.610
Jaccard score	0.500	

4.3.2. Experimento 2

Etiqueta	Precisión	Recall	Etiqueta	Precisión	Recall
0	0.707	0.616	25	0.573	0.239
1	0.699	0.902	26	0.707	0.362
2	0.509	0.609	27	0.691	0.335
3	0.433	0.664	28	0.360	0.647
4	0.546	0.535	29	0.591	0.731
5	0.473	0.651	30	0.698	0.312
6	0.639	0.655	31	0.591	0.703
7	0.716	0.745	32	0.602	0.458
8	0.742	0.584	33	0.536	0.675
9	0.616	0.469	34	0.675	0.363
10	0.437	0.265	35	0.508	0.569
11	0.370	0.349	36	0.628	0.506
12	0.425	0.844	37	0.720	0.356
13	0.326	0.762	38	0.556	0.538
14	0.449	0.509	39	0.698	0.112
15	0.692	0.812	40	0.574	0.695
16	0.527	0.439	41	0.621	0.837
17	0.619	0.426	42	0.679	0.729
18	0.520	0.615	43	0.688	0.528
19	0.778	0.344	44	0.688	0.450
20	0.716	0.927	45	0.429	0.741
21	0.973	0.915	46	0.714	0.463
22	0.945	0.887	47	0.657	0.816
23	0.898	0.622	48	0.816	0.669
24	0.624	0.356	49	0.773	0.897

	Precisión	Recall
micro avg	0.603	0.655
macro avg	0.623	0.585
Jaccard score	0.468	

4.3.3. Experimento 3

Etiqueta	Precisión	Recall	Etiqueta	Precisión	Recall
0	0.704	0.596	25	0.550	0.319
1	0.740	0.921	26	0.698	0.440
2	0.585	0.585	27	0.639	0.584
3	0.592	0.578	28	0.505	0.513
4	0.484	0.579	29	0.605	0.712
5	0.555	0.516	30	0.506	0.421
6	0.682	0.605	31	0.565	0.719
7	0.643	0.829	32	0.612	0.422
8	0.749	0.570	33	0.563	0.671
9	0.631	0.474	34	0.542	0.497
10	0.425	0.279	35	0.505	0.611
11	0.485	0.146	36	0.539	0.633
12	0.615	0.685	37	0.605	0.408
13	0.463	0.664	38	0.496	0.634
14	0.497	0.410	39	0.719	0.272
15	0.783	0.737	40	0.586	0.685
16	0.491	0.472	41	0.685	0.811
17	0.644	0.387	42	0.710	0.731
18	0.562	0.578	43	0.659	0.531
19	0.643	0.531	44	0.624	0.541
20	0.710	0.965	45	0.477	0.677
21	0.973	0.912	46	0.644	0.521
22	0.977	0.858	47	0.774	0.676
23	0.750	0.583	48	0.818	0.698
24	0.534	0.475	49	0.795	0.876

	Precisión	Recall
micro avg	0.651	0.649
macro avg	0.627	0.591
Jaccard score	0.488	

4.3.4. Experimento 25_labels

Etiqueta	Precisión	Recall
0	0.710	0.653
1	0.793	0.900
2	0.742	0.425
3	0.647	0.687
4	0.671	0.832
5	0.744	0.667
6	0.542	0.595
7	0.726	0.738
8	0.874	0.772
9	0.421	0.712
10	0.577	0.589
11	0.551	0.659
12	0.974	0.969
13	0.973	0.975
14	0.938	0.468
15	0.716	0.590
16	0.434	0.726
17	0.712	0.603
18	0.639	0.456
19	0.402	0.858
20	0.742	0.788
21	0.578	0.591
22	0.650	0.658
23	0.709	0.795
24	0.850	0.866

	Precisión	Recall
micro avg	0.684	0.747
macro avg	0.692	0.703
Jaccard score	0.578	

4.3.5. Experimento 12_labels

Etiqueta	Precisión	Recall
0	0.864	0.908
1	0.835	0.877
2	0.809	0.660
3	0.805	0.660
4	0.897	0.791
5	0.661	0.571
6	0.782	0.664
7	0.797	0.718
8	0.977	0.867
9	0.767	0.640
10	0.811	0.712
11	0.761	0.920

	Precisión	Recall
micro avg	0.825	0.779
macro avg	0.814	0.749
Jaccard score	0.673	

Capítulo 5

Análisis de Resultados

5.1. Comparación de resultados

A continuación mostramos una comparación de todas las métricas para cada experimento con 50 etiquetas:

Arquitectura	Experimento	Micro avg - Precisión	Macro avg - Precisión	Micro avg - Recall	Macro avg - Recall	Jaccard Score
Naive Bayes	1	0.223	0.248	0.777	0.714	0.270
	2	0.192	0.218	0.811	0.776	0.255
	3	0.167	0.197	**0.831**	**0.812**	0.243
BiLSTM	1	0.627	0.601	0.599	0.530	0.447
	2	0.511	0.476	0.580	0.473	0.368
	3	0.621	0.590	0.578	0.509	0.431
	4	0.453	0.390	0.576	0.489	0.349
BERT	1	**0.653**	0.625	0.664	0.610	**0.500**
	2	0.603	0.623	0.655	0.585	0.468
	3	0.651	**0.627**	0.649	0.591	0.488

A continuación mostramos una comparación de los resultados de los experimentos realizados con menos de 50 etiquetas:

Arquitectura	Experimento	Micro avg - Precisión	Macro avg - Precisión	Micro avg - Recall	Macro avg - Recall	Jaccard Score
BiLSTM	25 labels	0.688	0.663	0.716	0.666	0.552
	12 labels	**0.825**	0.803	0.775	0.741	**0.673**
BERT	25 labels	0.684	0.692	0.747	0.703	0.578
	12 labels	**0.825**	**0.814**	**0.779**	**0.749**	**0.673**

5.2. Observaciones

Como se puede observar a lo largo de los resultados obtenidos en cada experimento, para aumentar su precisión, los modelos disminuyen su *recall*, y viceversa, esto causa que si eligiéramos maximizar una de estas métricas para elegir un modelo, obtendríamos un modelo que tendrá puntajes muy bajos en la otra métrica. Debido a esto, para comparar los modelos y concluir a cuál le fue mejor, nos basamos en el coeficiente de Jaccard.

Si bien el baseline *Naive Bayes* tiene un *recall* elevado, su precisión y su coeficiente de Jaccard son los más bajos de todos los experimentos. Podemos interpretar esto como que los modelos de *Naive Bayes* predicen que la gran mayoría de los relatos pertenecen a la categoría correspondiente, haciendo esto, el valor del *recall* aumenta mientras que el de la precisión baja, ya que si bien el modelo está prediciendo la mayoría de los positivos reales correctamente, también está prediciendo como positivos una cantidad elevada de ejemplos que son negativos.

El baseline basado en BiLSTM mejora el coeficiente de Jaccard por sobre Naive Bayes de manera significativa, aumentando en alrededor de 20 puntos. Además, los valores de precisión y *recall* reportados son mucho más equilibrados, ya que ninguna de las dos métricas es demasiado baja.

Algo que notamos en BiLSTM de comparar los resultados del experimento 1 con los del 2, y los del 3 con los del 4, es que el *dropout* no mejoró los resultados del modelo como era esperado.

Por otro lado, el mejor experimento utilizando BERT alcanzó un coeficiente de Jaccard de 0.5, superando por 6 puntos al mejor experimento de BiLSTM y siendo el mejor coeficiente de Jaccard obtenido en los experimentos con 50 etiquetas.

Un ejercicio interesante fue haber realizado experimentos con menos etiquetas. Tanto con BiLSTM como con BERT, los resultados mejoraron considerablemente al disminuir la cantidad de categorías posibles. Ambas arquitecturas reportaron un coeficiente de Jaccard de 0.673 utilizando 12 etiquetas, siendo el valor más alto de todos los experimentos. Además, ambos modelos obtuvieron valores de precisión y *recall* superiores a 0.65. Esto se debe a que cuando disminuimos la cantidad de etiquetas utilizadas, removimos las que menos ejemplos asociados tenían. Estas etiquetas con menos ejemplos son más difíciles de aprender a clasificar para los modelos y por esto, al removerlas, las métricas mejoran considerablemente.

5.3. Matrices de confusión

En las figuras 5.1 y 5.2 presentamos matrices de confusión para los experimentos con las 12 categorías más comunes utilizando BiLSTM y BERT respectivamente. Estas matrices fueron calculadas utilizando el conjunto de test, por lo que son datos desconocidos para los modelos. En el recuadro superior izquierdo de cada matriz se muestran los verdaderos negativos (VN), en el inferior derecho los verdaderos positivos (VP), en el superior derecho los falsos positivos (FP) y en el inferior izquierdo los falsos negativos (FN).

En las matrices de confusión se observa lo desequilibradas que son las categorías. En las 12 categorías más comunes presentes se tienen más de 40000 negativos reales, mientras que la mayoría de las categorías poseen alrededor de 2000 positivos reales. Las excepciones son las etiquetas 1, 9 y 12 que tienen más de 5000 ejemplos asociados, e incluso estas excepciones solo tienen un 10 % de positivos reales en sus ejemplos.

Debido a que en las 12 categorías más comunes ya existen algunas que no tienen más del 10 % de positivos reales, es de esperarse que al tomar más categorías el desempeño del clasificador vaya disminuyendo cada vez más, debido a que estamos incluyendo categorías con muy pocos ejemplos asociados.

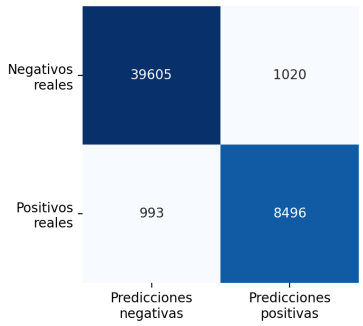
Al igual que como se vio reflejado en las métricas reportadas para los experimentos con 12 categorías, en las matrices de confusión de ambos modelos no existen grandes diferencias entre las predicciones hechas por BiLSTM y aquellas hechas por BERT. En general, la distribución de positivos y negativos para cada etiqueta están bastante similares.

Las etiquetas que más suelen confundir ambos modelos son:

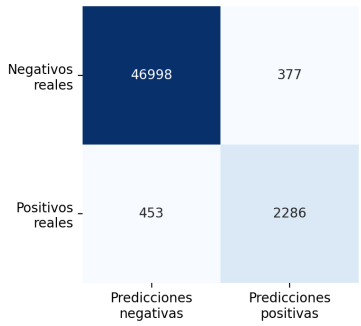
- Etiquetas 3, 4, 7 y 10: En estas etiquetas alrededor del 30 % de los positivos reales son clasificados como negativos.
- Etiqueta 6: En esta etiqueta alrededor del 40 % de los positivos reales son clasificados como negativos por BERT. Mientras que alrededor del 50 % de los positivos reales son clasificados como negativos por BiLSTM.

Por último, las etiquetas que menos suelen confundir ambos clasificadores son:

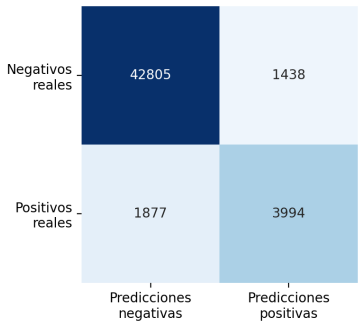
- Etiqueta 1: El 9 % de los positivos reales son clasificados como negativos.
- Etiqueta 2: El 12 % de los positivos reales son clasificados como negativos. Esta es la etiqueta con el tercer menor porcentaje de confusión y solo tiene 2700 ejemplos asociados.
- Etiqueta 9: El 13 % de los positivos reales son clasificados como negativos.
- Etiqueta 12: El 8 % de los positivos reales son clasificados como negativos.



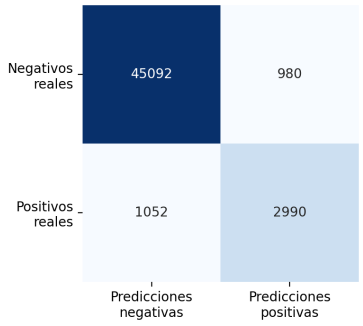
(a) Etiqueta 1



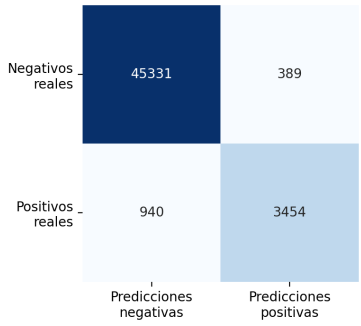
(b) Etiqueta 2



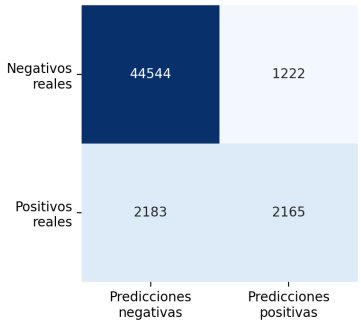
(c) Etiqueta 3



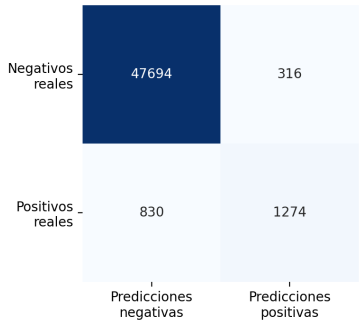
(d) Etiqueta 4



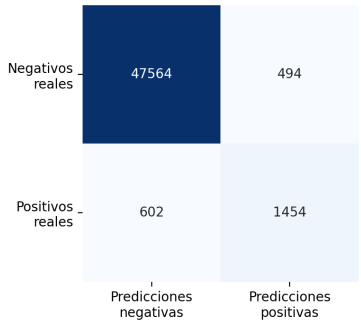
(e) Etiqueta 5



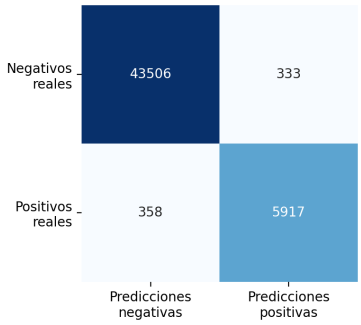
(f) Etiqueta 6



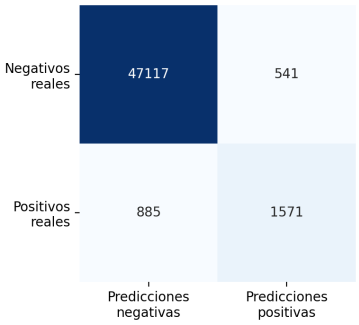
(g) Etiqueta 7



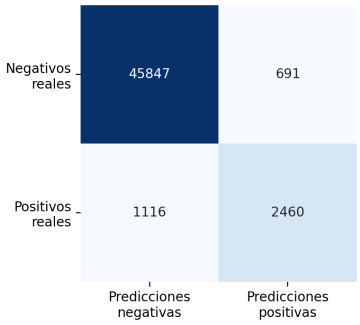
(h) Etiqueta 8



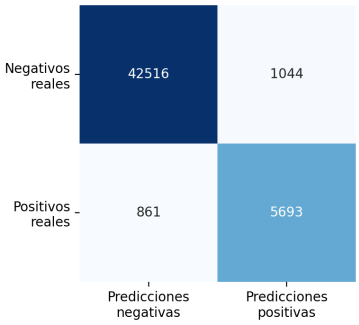
(i) Etiqueta 9



(j) Etiqueta 10

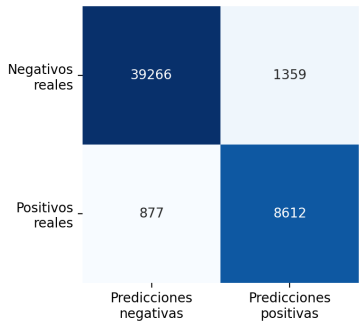


(k) Etiqueta 11

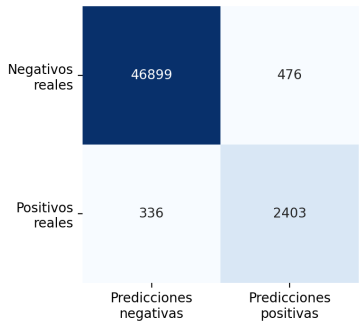


(l) Etiqueta 12

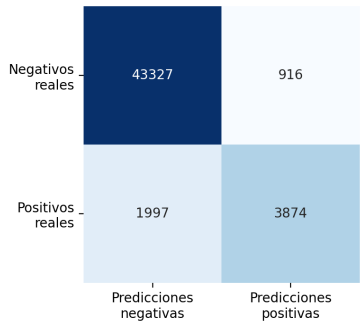
Figura 5.1: BiLSTM: Matrices de confusión para 12 categorías



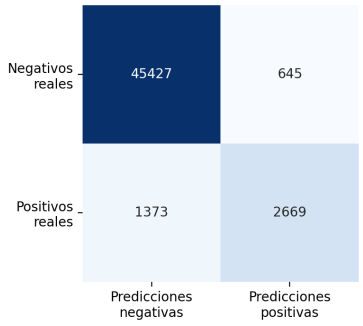
(a) Etiqueta 1



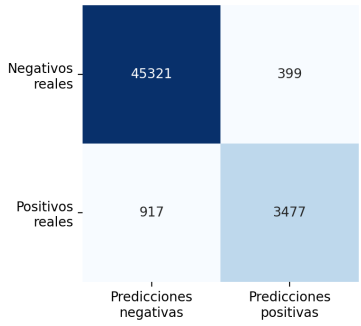
(b) Etiqueta 2



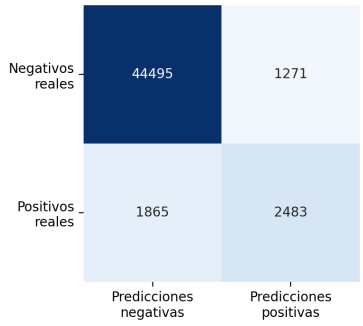
(c) Etiqueta 3



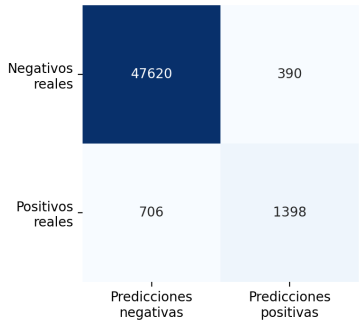
(d) Etiqueta 4



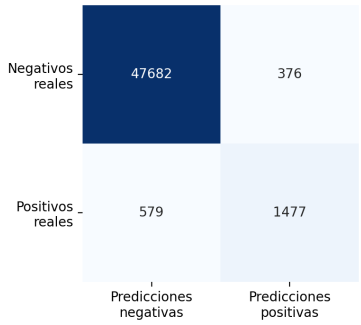
(e) Etiqueta 5



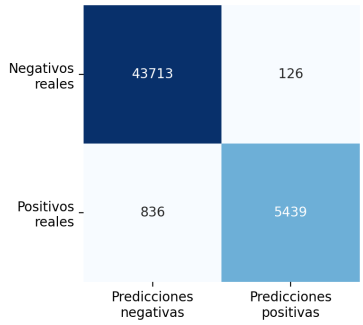
(f) Etiqueta 6



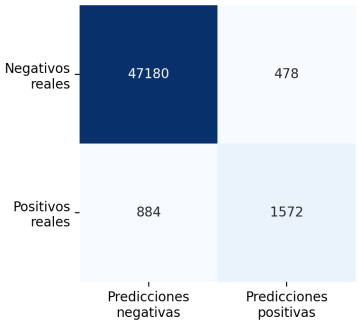
(g) Etiqueta 7



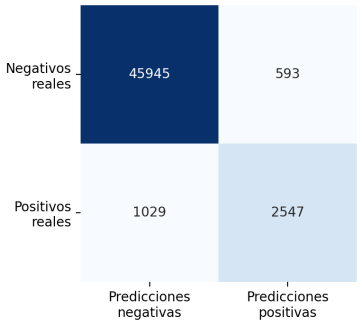
(h) Etiqueta 8



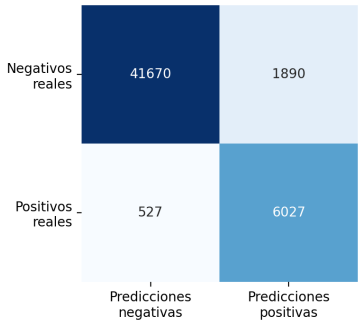
(i) Etiqueta 9



(j) Etiqueta 10



(k) Etiqueta 11



(l) Etiqueta 12

Figura 5.2: BERT: Matrices de confusión para 12 categorías

Capítulo 6

Conclusiones

Realizamos este proyecto con el objetivo de construir un prototipo de clasificador de relatos médicos que en el futuro sea capaz de apoyar el flujo de toma de decisiones en la ACHS. La idea es tener un clasificador que, dado un texto consistente en el diagnóstico de un trabajador accidentado, pueda asignarle 0 o más categorías de un conjunto de estas.

La cantidad de categorías posibles que posee la ACHS para sus diagnósticos está alrededor de las 3000. Este número es muy elevado para un clasificador que utiliza las técnicas que usamos en este proyecto y por esto no nos fue posible utilizar todas las categorías disponibles. En su lugar, lo que hicimos fue utilizar los relatos que pertenecen a las 50 categorías más comunes. Además, también realizamos experimentos extras utilizando las 25 y las 12 categorías más comunes, para evaluar el desempeño del clasificador en estas situaciones.

Debido a que los datos poseen información delicada de los pacientes atendidos (como nombres, mails o teléfonos), durante gran parte del proyecto estos debieron ser mantenidos en servidores provistos por la ACHS.

El tener que mantener los datos en servidores de la ACHS fue nuestra principal limitante a la hora de desarrollar el proyecto, debido a que en un principio no tuvimos acceso a GPUs que son necesarias para realizar experimentos con arquitecturas de *Deep Learning*. Además, una vez que tuvimos acceso a GPUs, estas no eran de la gama que esperábamos, por lo que los experimentos tomaron más tiempo del que teníamos presupuestado.

Ya en la última etapa del proyecto, después de negociar con la ACHS, logramos obtener aprobación para descargar los datos de sus servidores y utilizar servidores propios donde pudimos ejecutar los experimentos mucho más ágilmente debido a la gama de las GPUs que tenemos disponibles.

De los resultados obtenidos, notamos que BERT obtuvo los mejores resultados al utilizar las 50 etiquetas más comunes, obteniendo los mejores puntajes de precisión y coeficiente de Jaccard, y si bien no reportó mejor *recall* que Naive Bayes, este último no reportó valores de precisión o coeficiente de Jaccard que hagan que valga la pena utilizarlo, haciendo que BERT sea el modelo más equilibrado de los tres.

Algo interesante ocurre al disminuir la cantidad de etiquetas utilizadas. Al utilizar las 12 etiquetas más comunes, los puntajes de BiLSTM y BERT son muy cercanos, e incluso iguales en micro *average* de precisión y coeficiente de Jaccard. Si se hiciera un trabajo posterior de disminuir la cantidad de etiquetas, quizás no habría tanta diferencia entre utilizar ambos modelos.

Los prototipos obtenidos en este proyecto son una buena base para seguir trabajando en la construcción de un clasificador que pueda ser utilizado en los flujos de la ACHS en el futuro. Como se ve en los resultados, variar los hiperparámetros utilizados en los modelos no sube significativamente los puntajes obtenidos. Sin embargo, al disminuir la cantidad de categorías, todas las métricas reportan valores significativamente superiores. Es por esto que el principal trabajo a realizar si se busca tener modelos que puedan ser utilizados en producción es disminuir el universo de categorías en las que puede ser clasificado un relato.

Algo a notar es que las categorías, como las recibimos, no tienen ninguna agrupación o jerarquía entre ellas. Posiblemente existen varias categorías que son subcategorías de otras. Con ayuda de conocedores del dominio se podría realizar un ordenamiento y agrupación de estas categorías jerárquicamente, para luego utilizar los modelos implementados en este proyecto para clasificar los relatos de la forma más general posible. Posteriormente, utilizando otros modelos, o incluso los mismos, se podrían obtener clasificaciones más específicas, dada la clasificación general ya obtenida.

Bibliografía

[1] *Ley 16.744*, Consultado el 17 de Diciembre, 2020. https://www.previsionsocial.gob.cl/sps/seguridad-social/sst/ley-16-744/.

[2] *6 Easy Steps to Learn Naive Bayes Algorithm with codes in Python and R*, Consultado el 26 de Diciembre, 2020. https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/.

[3] M. van Gerven and S. M. Bohte, "Editorial: Artificial neural networks as models of neural information processing," *Frontiers Comput. Neurosci.*, vol. 11, p. 114, 2017.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[5] H. Liang, X. Sun, Y. Sun, and Y. Gao, "Text feature extraction based on deep learning: a review," *EURASIP J. Wireless Comm. and Networking*, vol. 2017, p. 211, 2017.

[6] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," in *7th International Conference on Cloud Computing and Big Data, CCBD 2016, Macau, China, November 16-18, 2016*, pp. 99–104, IEEE Computer Society, 2016.

[7] A. Sherstinsky, "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network," *CoRR*, vol. abs/1808.03314, 2018.

[8] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies," in *A Field Guide to Dynamical Recurrent Neural Networks* (S. C. Kremer and J. F. Kolen, eds.), IEEE Press, 2001.

[9] Y. Bengio, P. Frasconi, and P. Y. Simard, "The problem of learning long-term dependencies in recurrent networks," in *Proceedings of International Conference on Neural Networks (ICNN'88), San Francisco, CA, USA, March 28 - April 1, 1993*, pp. 1183–1188, IEEE, 1993.

[10] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[11] Y. Kim, C. Denton, L. Hoang, and A. M. Rush, "Structured attention networks," *CoRR*, vol. abs/1702.00887, 2017.

[12] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2015.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA* (I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, eds.), pp. 5998–6008, 2017.

[14] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)* (J. Burstein, C. Doran, and T. Solorio, eds.), pp. 4171–4186, Association for Computational Linguistics, 2019.

[15] J. Cañete, G. Chaperon, R. Fuentes, J.-H. Ho, H. Kang, and J. Pérez, "Spanish pre-trained bert model and evaluation data," in *PML4DC at ICLR 2020*, 2020.

[16] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.