



CARRERA DE ESPECIALIZACIÓN EN INTELIGENCIA ARTIFICIAL

MEMORIA DEL TRABAJO FINAL

Clasificación de reclamos de usuario

Autor:
Ing. Lucas Rivela

Director:
Dr. Lic. Rodrigo Cárdenas (FIUBA)

Jurados:
Nombre del jurado 1 (pertenencia)
Nombre del jurado 2 (pertenencia)
Nombre del jurado 3 (pertenencia)

*Este trabajo fue realizado en la Ciudad Autónoma de Buenos Aires,
entre mayo de 2023 y septiembre de 2023.*

Resumen

La presente memoria describe el diseño e implementación de un sistema de clasificación de reclamos desarrollado para Ualá. La solución permite optimizar el tiempo que lleva el proceso de clasificación mediante la asignación automática de categorías de primer y segundo nivel; y por otro lado, reducir la cantidad de personas necesarias para esta tarea.

Para poder realizar este trabajo se aplicaron conceptos de bases de datos, procesamiento del lenguaje natural y aprendizaje profundo para realizar la extracción de los datos, el procesamiento del texto, el entrenamiento de los modelos de IA y el despliegue de los mismos en un ambiente de desarrollo.

Agradecimientos

A mi familia y amigos por apoyarme durante la realización de esta carrera.

A mis compañeros y profesores por el acompañamiento.

A mi director, Dr. Lic. Rodrigo Cárdenas por orientarme y guiarme en la realización de este trabajo.

Índice general

Resumen	I
1. Introducción general	1
1.1. Introducción	1
1.1.1. Qué es la atención al cliente	1
1.1.2. Organización de un equipo de atención al cliente	1
1.1.3. El proceso de atención al cliente	2
1.2. Motivación	3
1.3. Estado del arte	4
1.3.1. Word Embeddings	4
1.3.2. Transformers	4
1.4. Objetivos y alcance	5
1.4.1. Objetivos	5
1.4.2. Alcance	6
2. Introducción específica	7
2.1. Requerimientos	7
2.2. Preprocesamiento del texto	8
2.2.1. Natural Language Toolkit	8
Segmentación	8
Remoción de palabras vacías	8
Derivación	9
2.2.2. Scikit-Learn	9
Normalización de etiquetas	9
Vectorizador Term Frequency - Inverse Document Frequency	9
2.3. Modelos de inteligencia artificial utilizados	10
2.3.1. Complement Naive Bayes	10
2.3.2. Representaciones de codificador bidireccional de transformadores	11
2.4. Herramientas de software utilizadas	13
2.4.1. Python	13
2.4.2. GitHub	13
2.4.3. BigQuery	13
2.4.4. Vertex AI workbench	13
2.4.5. Docker	13
2.4.6. Artifact registry	14
2.4.7. Cloud Composer y Apache Airflow	14
3. Diseño e implementación	15
3.1. Arquitectura del sistema	15
3.2. Extracción y preprocesamiento del texto	16
3.2.1. Extracción de datos de BigQuery	17

3.2.2.	<i>Pipeline</i> de preprocesamiento	18
	Codificación para CNB	19
	Codificación para BERT	19
3.3.	Entrenamiento de los modelos	20
3.3.1.	Análisis exploratorio	20
3.3.2.	Integración de los modelos	21
3.3.3.	Entrenamiento de los modelos base	21
	Modelos base en validación cruzada	22
3.3.4.	Entrenamiento de BERT	22
	Aprendizaje por transferencia	23
	Entorno de entrenamiento	24
	Puntos de control	24
	Parada temprana	25
	Variaciones en el entrenamiento	26
	Técnicas para el desbalance de clases	26
	Optimizadores	27
3.4.	Empaquetamiento del código y artefactos	28
3.5.	Desarrollo del <i>pipeline</i> de predicción	30
3.5.1.	Creación y ejecución del flujo de trabajo	30
3.5.2.	Parametrización de los contenedores	32
4.	Ensayos y resultados	35
4.1.	Métrica de evaluación de los modelos	35
4.2.	Resultados de desempeño en pruebas	37
4.2.1.	Resultados para clasificador L1	37
4.2.2.	Resultados para clasificador L2 A	37
4.2.3.	Resultados para clasificador L2 B	38
4.2.4.	Resultados para clasificador L2 C	38
4.2.5.	Resultados para clasificador L2 D	39
4.2.6.	Resultados para clasificador L2 E	39
4.2.7.	Resultados para clasificador L2 F	39
4.2.8.	Resultados para clasificador L2 G	40
4.2.9.	Resultados para clasificador L2 H	40
4.2.10.	Resultados para clasificador L2 I	41
4.2.11.	Resultados para clasificador L2 J	41
4.2.12.	Resultados para clasificador L2 K	41
4.2.13.	Resultados para clasificador L2 L	42
4.2.14.	Resultados para clasificador L2 M	42
4.2.15.	Resultados para clasificador L2 N	43
4.3.	Simulación en ambiente de desarrollo	43
5.	Conclusiones	45
5.1.	Conclusiones generales	45
5.2.	Próximos pasos	45
	Bibliografía	47

Índice de figuras

1.1. Organigrama de un área de atención al cliente ¹	2
1.2. Evolución de las redes en cantidad de parámetros. ²	5
1.3. Diagrama general de la solución.	6
2.1. Ejemplo de codificación realizado sobre una variable ³	9
2.2. Ejemplo de funcionamiento del mecanismo de atención propia ⁴ . . .	12
2.3. Ejemplo de entrada de datos a BERT con MLM y NSP combinados ⁵ . .	12
3.1. Diagrama de la arquitectura de alto nivel.	15
3.2. Cantidad de categorías L2 para cada L1.	17
3.3. Diagrama del procesamiento del texto crudo.	18
3.4. Diagrama de codificación para CNB.	19
3.5. Diagrama de codificación para BERT.	19
3.6. Ejemplo de segmentación <i>WordPiece</i> ⁶	20
3.7. Proporción de casos.	20
3.8. Longitud de los mensajes.	21
3.9. Diagrama general de los modelos de IA.	21
3.10. Esquema de entrenamiento de BERT para clasificación de texto. . .	23
3.11. Técnica de extracción de características.	23
3.12. Técnica de ajuste fino.	23
3.13. Funciones de pérdida a través de las épocas.	25
3.14. Función de pérdida y exactitud para Adam (azul) y LAMB (rojo) para MNIST ⁷	28
3.15. Interacción de los distintos componentes.	29
3.16. Menú de GitHub para ejecutar el flujo de trabajo.	29
3.17. Vista detallada de un Dockerfile.	30
3.18. Interacción de los componentes del <i>pipeline</i>	31
3.19. Vista en forma de grafo del DAG desde el menú de Airflow.	31
3.20. Estados del ciclo de vida de un Pod de Kubernetes ⁸	32
3.21. Diagrama de parametrización de un contenedor.	33
4.1. Ejemplo de matriz de confusión multiclase genérica ⁹	35
4.2. Resultados obtenidos para los modelos clasificadores L1.	37
4.3. Resultados obtenidos para los modelos clasificadores L2 A.	38
4.4. Resultados obtenidos para los modelos clasificadores L2 B.	38
4.5. Resultados obtenidos para los modelos clasificadores L2 C.	38
4.6. Resultados obtenidos para los modelos clasificadores L2 D.	39
4.7. Resultados obtenidos para los modelos clasificadores L2 E.	39
4.8. Resultados obtenidos para los modelos clasificadores L2 F.	40
4.9. Resultados obtenidos para los modelos clasificadores L2 G.	40
4.10. Resultados obtenidos para los modelos clasificadores L2 H.	40
4.11. Resultados obtenidos para los modelos clasificadores L2 I.	41
4.12. Resultados obtenidos para los modelos clasificadores L2 J.	41

4.13. Resultados obtenidos para los modelos clasificadores L2 K.	42
4.14. Resultados obtenidos para los modelos clasificadores L2 L.	42

Índice de tablas

3.1. Especificaciones técnicas Vertex	24
3.2. Configuración ModelCheckpoint	25
3.3. Configuración EarlyStopping	26

Capítulo 1

Introducción general

En este capítulo se realiza una introducción al funcionamiento de un área de atención al cliente. Además, se menciona el estado del arte de los sistemas de procesamiento del lenguaje natural, y por último se explican los objetivos y alcances del presente trabajo.

1.1. Introducción

En esta sección se introduce un área típica de atención al cliente en una empresa.

1.1.1. Qué es la atención al cliente

El área de atención al cliente se encarga de dar soporte al consumidor y tiene como objetivo resolver sus problemas [1].

A menudo se confunde el área de atención con el área de servicio al cliente. La principal diferencia radica en que el servicio es una función proactiva con la intención de anticiparse a las necesidades inmediatas y a largo plazo del cliente. La atención al cliente es una función reactiva que busca resolver los problemas que el cliente manifestó [2].

A continuación se listan las características principales del proceso [2]:

1. Inicio y duración: inicia cuando un cliente se pone en contacto con la empresa. Demora el tiempo que sea necesario hasta brindar una solución.
2. Objetivo: solucionar problemas que surjan del funcionamiento del producto o condiciones del servicio.
3. Actitud: reactiva.
4. Interacción: canales específicos. Por ejemplo, telefónicos, *email* y *chat*.
5. Participantes: cliente y representante del centro de atención. Raras veces entran en juego otros trabajadores de la empresa.

1.1.2. Organización de un equipo de atención al cliente

A continuación se enumeran los principales roles que se pueden encontrar en el área de atención al cliente:

1. Gerente de atención al cliente: tiene bajo su responsabilidad el cumplimiento de metas estratégicas. Gestionan el volumen de casos entrantes y comunican las tendencias a otros departamentos. [3]

2. Coordinador de atención al cliente: es quien está en el día a día en contacto con los agentes. Apoya la resolución de problemas y escala aquellos que requieren validaciones o decisiones que no estén a su alcance [4].
3. Analista de atención al cliente: canaliza las quejas, reclamos y sugerencias. Se encarga de proveer soporte a los usuarios. [4]

En algunas organizaciones, dependiendo de su tamaño, se pueden encontrar más o menos mandos intermedios entre el gerente y el coordinador. Una variante puede ser tener varios coordinadores respondiendo a un supervisor. Este supervisor puede estar a cargo de la atención de reclamos para ciertos productos de la empresa que están relacionados.

En la figura 1.1 se puede ver la principal tendencia que hay en cuanto a organización de equipos. La idea es que cada equipo sea especialista en un conjunto de casos.



FIGURA 1.1. Organigrama de un área de atención al cliente¹.

1.1.3. El proceso de atención al cliente

El proceso de atención al cliente consiste en una serie de pasos que se realizan para atender los reclamos y/o consultas que recibe la empresa [5].

A continuación se describen los pasos principales de un proceso de atención típico [6].

- Contacto y captura de la demanda del cliente: en esta etapa se recibe el mensaje del cliente por cualquiera de los canales establecidos y se procede a registrarlo en el sistema.
- Análisis y clasificación: se analiza la información recibida y se evalúa si es suficiente para proceder a la clasificación del caso. De lo contrario se vuelve a contactar al cliente para obtener más detalles.

¹Imagen tomada de <https://blog.hubspot.es/service/que-es-atencion-al-cliente>

- Resolución: en esta etapa se busca dar una respuesta a la consulta o reclamo del cliente. En algunos casos puede involucrar varios equipos o personas, por lo que su duración es variable.
- Cierre: en esta etapa se presenta la solución al cliente y se le comunican los próximos pasos si los hubiera.

1.2. Motivación

En la actualidad, resulta sumamente sencillo para el cliente optar por la competencia si la empresa no logra satisfacer sus expectativas. Una mala CX (*Customer Experience*) puede generar un efecto de bola de nieve, ya que los usuarios que hayan tenido una mala experiencia, son mas propensos a comunicarlo con sus conocidos haciendo que la empresa no sólo pierda un cliente sino que además, se le dificulte expandir su mercado [7].

Según informes de *CX Trends* [8][9][10] y de Esteban Kolsky [11]:

- El 70 % de los consumidores gastará más en una empresa que ofrezca una buena CX.
- El 50 % de los clientes se pasaría a la competencia después de haber tenido tan solo una mala experiencia. Este valor sube a un 80 % si se les pregunta si hubieran tenido dos o más malas experiencias.
- El 72 % de los clientes compartiría una experiencia positiva con 6 o más personas.
- El 13 % de los clientes compartiría su experiencia con 15 o más personas si no está satisfecho.
- Sólo el 3,85 % de los clientes insatisfechos se lo comunican a la empresa.

Estos números muestran que la ausencia de quejas no es un signo de satisfacción. Por el contrario, es probable que los clientes hayan abandonado la empresa y estén compartiendo su insatisfacción con otras personas. También muestran la importancia de mantenerlos satisfechos, ya que son propensos a divulgarlo.

Más aún, entre las principales razones de una mala atención al cliente se encuentran tener tiempos de espera demasiado largos [12] y tener muchas derivaciones internas [13].

De lo anterior se infiere que resulta muy importante que los procesos del área se realicen de forma eficiente. La utilización de modelos de IA para automatizar el proceso de clasificación de reclamos y consultas, ayuda a la empresa a responder a sus clientes con tiempos de respuesta menores y permite disponer de más analistas en la etapa de resolución y cierre. De esta manera, se genera una imagen positiva de la empresa que facilita la fidelización de clientes existentes y aumenta la incorporación de los nuevos.

1.3. Estado del arte

El PNL (Procesamiento del lenguaje natural) es un subcampo de la inteligencia artificial que busca enseñar a un programa informático a comprender, interpretar y generar texto en lenguaje humano. A principios del siglo XXI, el PNL experimentó un crecimiento significativo gracias a la aplicación de algoritmos de aprendizaje profundo y la disponibilidad de grandes corpus de texto [14].

1.3.1. Word Embeddings

En 2003 se entrena la primer red neuronal orientada al lenguaje utilizando vectores para representar palabras. Llamarían a este proceso “aprender una representación distribuida para cada palabra” [15].

En 2008 se introduce el concepto de *Word Embeddings* como una herramienta potente para tareas de PNL. Se distinguirían de sus antecesores mencionando que su objetivo era predecir la relevancia de una palabra dada la parte previa y posterior de la oración, a diferencia del trabajo previo que buscaba predecir la probabilidad de una palabra dada la parte previa de una oración.

En 2013, publican un artículo dónde detallan las arquitecturas *CBOW* y *Skip-Gram*. Además liberan el primer modelo pre-entrenado llamado *Word2Vec* (basado en *Skip-Gram*) popularizando el uso de los *Word Embeddings* [16].

Posteriormente, en 2014 publicarían *GloVe* como otro método de generación de *Word Embeddings* que utiliza una probabilidad de co-ocurrencia [17]. Con esta técnica, si dos palabras co-existen muchas veces, ambas palabras tienen una probabilidad alta de tener un mismo significado.

Los *Word Embeddings* se convirtieron en la herramienta principal dentro del PNL. Capturan el significado de una palabra y la traducen a una representación numérica que puede ser usada como entrada para las redes neuronales.

1.3.2. Transformers

El avance del aprendizaje profundo, permitió a los investigadores desarrollar arquitecturas de redes neuronales más avanzadas y eficientes.

Sin embargo, el verdadero hito no fue hasta 2017, cuando se publica la utilización de un mecanismo de atención para desarrollar una nueva arquitectura que se llamaría *Transformers* [18]. Las redes más usadas hoy en día están basadas en esta mejora.

Básicamente, en los *Transformers* se reemplazan las capas recurrentes que se venían utilizando hasta ese momento por “capas de atención” que codifican cada palabra en función del resto de la frase, permitiendo de esta forma, introducir el contexto en la representación matemática del texto. Por este motivo, sus *Embeddings* generados son denominados *Embeddings* contextuales.

Otra de las innovaciones introducidas es el uso de *Embeddings* posicionales. Con ellos se logra una mayor paralelización, ya que no es más necesario pasar una palabra a la vez. Agregando un valor secuencial con cada palabra, hacen posible pasarle a la red todas las palabras en simultáneo.

Estas redes fueron evolucionando en tamaño y complejidad conforme transcurría el tiempo. En la figura 1.2 se puede ver la evolución de la cantidad de parámetros de estas redes a lo largo de los años.

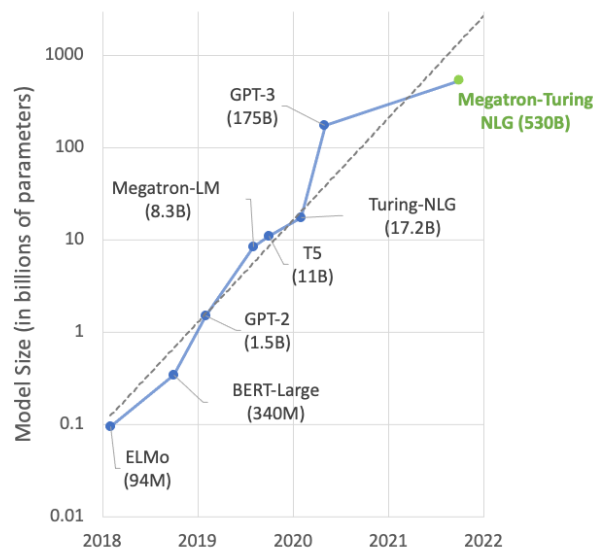


FIGURA 1.2. Evolución de las redes en cantidad de parámetros.²

Los modelos basados en *Transformers* se convirtieron en el modelo por defecto para tareas como traducción, clasificación de texto y resumen de textos, por citar algunos ejemplos.

En el área de atención al cliente, los principales usos de estas redes son [19]:

- *Chatbots*: ayudan a responder consultas de forma inmediata evitando filas de espera.
- Análisis de sentimiento: para realizar análisis sobre comentarios y quejas de los clientes.
- Redireccionamiento de tickets: eliminan cuellos de botella en la parte de clasificación al redirigir el ticket directamente al área correspondiente.

1.4. Objetivos y alcance

1.4.1. Objetivos

El propósito de este trabajo fue el desarrollo de modelos de inteligencia artificial (IA) para mejorar y agilizar la atención al cliente. Para lograr este objetivo es necesario realizar la clasificación automática de los reclamos de usuario en categorías de primer y segundo nivel.

En la figura 1.3 se presenta un diagrama de alto nivel de la solución. Se observa que en primera instancia, los reclamos se reciben por correo y chat, y luego son registrados en Salesforce, el sistema de CRM (*Customer Relationship Management*)

²Imagen tomada de <https://developer.nvidia.com/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/>

que utiliza el área de atención al cliente [20]. Estos datos se replican en BigQuery a través de un proceso orquestado por Apache Airflow [21] [22].

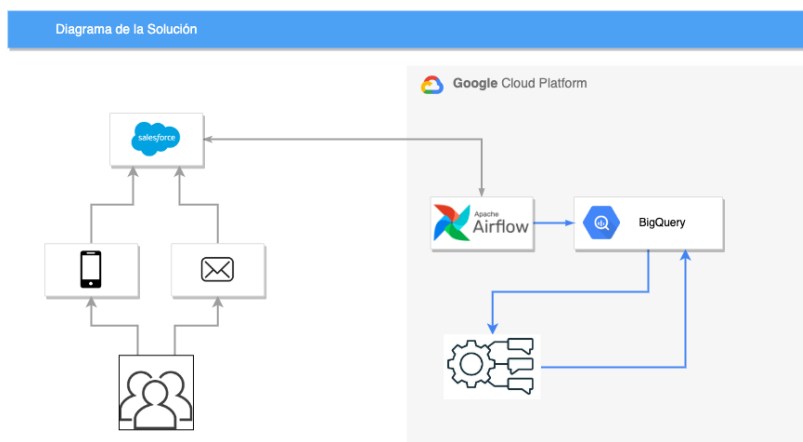


FIGURA 1.3. Diagrama general de la solución.

Los modelos de IA clasifican los casos que están en BigQuery y guardan la categoría de primer y segundo nivel en una tabla que luego el usuario podrá utilizar.

1.4.2. Alcance

El alcance de este proyecto estuvo orientado a desarrollar un prototipo de solución de software que incluyó las siguientes actividades:

- Obtención de los datos: se corresponde con el análisis de las fuentes de datos disponibles, tanto para entrenar los modelos de IA como para la parte de predicción de nuevos casos.
- Análisis exploratorio de los datos: se corresponde con las actividades necesarias para generar nuevos *insights*, que sirvieron para guiar el desarrollo de los modelos.
- Modelado: se corresponde con la generación de variables a partir de los datos disponibles.
- Entrenamiento: se corresponde con el entrenamiento de los modelos a partir de las variables obtenidas. También incluye la selección del mejor modelo para cada clasificación.
- Despliegue: se corresponde con el diseño de la infraestructura para ejecutar los modelos de IA y su despliegue en un ambiente no productivo.
- Documentación: se corresponde con los documentos de soporte que explican los procesos de modelado, entrenamiento y despliegue.

El alcance de este trabajo no cubre:

- La adaptación de los modelos a nuevas categorías inexistentes en los años 2021 y 2022.
- El despliegue de los modelos en un ambiente productivo.
- El soporte de la infraestructura desplegada en el ambiente no productivo.

Capítulo 2

Introducción específica

En este capítulo se enumeran los requisitos que la solución debe cumplir y luego se describen principalmente las herramientas utilizadas durante el desarrollo para el entrenamiento y despliegue de los modelos para satisfacerlos.

2.1. Requerimientos

En esta sección se detallan los requerimientos funcionales y las restricciones de implementación del trabajo.

1. Requerimientos funcionales

- a) El sistema debe poder detectar la categoría de un reclamo escrito en lenguaje natural.
- b) El sistema debe poder detectar la categoría de una consulta escrita en lenguaje natural.
- c) El usuario debe poder utilizar los resultados de la clasificación desde una base de datos.
- d) El proceso debe ser capaz de interpretar errores de ortografía.
- e) El proceso debe ser capaz de adaptarse a distinta cantidad de palabras en el mensaje.
- f) La solución debe ejecutarse en forma *batch*, corriendo diariamente y tomando los casos del día anterior.

2. Requerimientos no funcionales

- a) El sistema debe estar desarrollado en lenguaje Python.
- b) El código debe ser versionado con Git.
- c) La solución debe estar desplegada sobre infraestructura de Google Cloud Platform.
- d) La salida de los modelos debe ser almacenada en BigQuery.
- e) El proceso debe ser ejecutado a través del orquestador Apache Airflow.

3. Requerimientos de testing

- a) Se deben generar métricas de desempeño de los modelos con el dataset de entrenamiento y de prueba.

4. Requerimientos de documentación

- a) Se debe confeccionar un documento con el diseño de la arquitectura de alto nivel.
- b) Se debe confeccionar un documento con el diseño de los modelos de IA.
- c) Se debe confeccionar un documento que especifique los datos que consumen los modelos y su origen.

2.2. Preprocesamiento del texto

En esta sección se introducen las herramientas utilizadas para realizar la limpieza y preparación del texto que sirvió tanto para la parte de entrenamiento como para la parte de predicción en el despliegue.

2.2.1. Natural Language Toolkit

NLTK (*Natural Language Toolkit*) es una librería de Python para preprocesar texto. Es una de las herramientas principales del mercado y entre otras cosas permite:

- Realizar la “tokenización” o segmentación del texto.
- Remover *stopwords* o palabras vacías.
- Aplicar *stemming* sobre las palabras.
- Realizar “lematización” sobre las palabras.
- Etiquetar cada término en una oración según sea sustantivo, verbo, adjetivo, preposición, etc.

A continuación se profundizará en las funcionalidades de esta librería que se usaron para este trabajo.

Segmentación

La segmentación consiste en separar un documento en términos individuales. Estos términos se llaman *tokens* y es la unidad mínima de análisis de texto. Dependiendo de la tarea en cuestión, un *token* puede ser una palabra, una sílaba o incluso un solo carácter.

Esta etapa es importante porque sirve como base para las etapas posteriores del preprocesamiento del texto.

Remoción de palabras vacías

Las *stopwords* o palabras vacías son palabras que son muy comunes en el idioma español y están destinadas a aparecer en todas las oraciones, sin importar el tema al que hagan referencia [23], por ejemplo: “en”, “este”, “el”, “las”.

Eliminar palabras comunes permite eliminar palabras con poco valor discriminativo entre textos (o categorías de reclamos). Además reduce el volumen de datos a procesar.

Derivación

El *stemming* o derivación o poda consiste en recortar las palabras para reducirlas a una base común. Es decir, devuelve el tallo de una palabra, que no necesariamente es igual a la raíz morfológica de la palabra. Por ejemplo, para la palabra “comienza” su derivación será “comienz” y para “peces” será “pec”.

La idea detrás de esta técnica es tratar de agrupar las palabras similares, reduciendo la cantidad de palabras únicas en un *dataset*.

2.2.2. Scikit-Learn

Scikit-Learn es una librería de Python para *Machine Learning* que cuenta con una variedad de algoritmos y modelos para clasificación y regresión. Además cuenta con un conjunto de herramientas de extracción y generación de variables, tanto para datos numéricos como para texto.


A continuación se dará una explicación de los módulos usados de esta librería.

Normalización de etiquetas

La normalización de etiquetas o *label encoding* consiste en codificar variables categóricas en valores numéricos que irán entre 0 y $n - 1$ siendo n la cantidad de categorías. Es decir, asigna un número único a cada categoría. De esta forma, al entrenar los modelos de IA se puede pasar la variable *target* en formato numérico.

Otra utilidad muy importante de la versión que trae Scikit-Learn es la posibilidad de realizar la transformación inversa. Esto resulta muy útil para determinar el texto de la variable numérica que predice el modelo.

En la figura 2.1 se puede ver un ejemplo de la transformación.



State (Nominal Scale)	State (Label Encoding)
Maharashtra	3
Tamil Nadu	4
Delhi	0
Karnataka	2
Gujarat	1
Uttar Pradesh	5

FIGURA 2.1. Ejemplo de codificación realizado sobre una variable¹.

Vectorizador Term Frequency - Inverse Document Frequency

La vectorización es una técnica de PNL que convierte una secuencia de *tokens* (obtenidos previamente en la etapa de segmentación) a un vector numérico.

Hay distintas maneras de realizar este proceso, la elegida para este trabajo es la que se conoce como TF-IDF (*Term Frequency - Inverse Document Frequency*). Su objetivo es reflejar cuán importante es una palabra respecto al documento. *Nota:*

¹Imagen tomada de <https://www.mygreatlearning.com/blog/label-encoding-in-python/>

Para este trabajo, la palabra documento representa un caso de reclamo o consulta de un cliente.

Tiene dos partes: el cálculo de TF y el cálculo de IDF.

TF calcula para cada documento del *dataset*, la cantidad de veces que un *token* aparece en él. Es decir, calcula la frecuencia de un *token* en cada documento. Luego, ese número se divide por la cantidad de palabras que tenía ese documento. La fórmula para calcularla es la siguiente [24]

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}} \quad (2.1)$$

IDF calcula la proporción de documentos del *dataset* que poseen el *token*. Es decir, divide la cantidad total de documentos sobre la cantidad de documentos con ese *token*, y luego a ese resultado le aplica el logaritmo. Los términos raros tendrán un puntaje más alto. La siguiente ecuación muestra cómo se calcula:

$$idf(w) = \log\left(\frac{N}{df_i}\right) \quad (2.2)$$

Finalmente, una vez obtenidos el TF y el IDF, lo que se hace es multiplicar ambos términos para cada *token* en todos los documentos:

$$w_{i,j} = tf_{i,j} \times idf(w) \quad (2.3)$$

2.3. Modelos de inteligencia artificial utilizados

En esta sección se presentan los modelos de IA utilizados.

2.3.1. Complement Naive Bayes

Los algoritmos de *Naive Bayes* o Bayes ingenuo son muy utilizados para tareas de clasificación por su velocidad de cómputo. Son modelos probabilísticos que deben su nombre a la probabilidad condicional y el teorema de Bayes. Se les denomina ingenuos porque suponen una independencia entre variables, que muchas veces no es real [25].

Para este trabajo se seleccionó particularmente el modelo CNB (*Complement Naive Bayes*), que es una adaptación del MNB (*Multinomial Naive Bayes*), pero que se adecúa mejor al desbalanceo entre clases. Por lo general, CNB supera a MNB en tareas de clasificación de texto [26].

Lo que hace el algoritmo es calcular, para cada clase, la probabilidad de que un documento no le pertenezca. Luego, la clase del documento será la clase con menor probabilidad, ya que aquí se quiere minimizar la probabilidad de no pertenecer.

La fórmula de CNB es la siguiente [27]:

$$l(t) = \arg \min \sum_i t_i w_{ci} \quad (2.4)$$

donde t_i es la cantidad de veces que aparece el *token* i en el documento y:

$$w_{ci} = \frac{\log \hat{\theta}_{ci}}{\sum_k |\log \hat{\theta}_{ck}|} \quad (2.5)$$

siendo:

$$\hat{\theta}_{ci} = \frac{\alpha_i + \sum_{j: y_j \neq c} d_{ij}}{\alpha + \sum_{j: y_j \neq c} \sum_k d_{kj}} \quad (2.6)$$

donde las sumatorias son sobre todos los documentos j que no sean de la clase c , d_{ij} es la cantidad de veces que aparece el *token* en el documento j , α_i es un hiperparámetro de suavizado (por lo general equivale a 1) y $\alpha = \sum_i \alpha_i$.

2.3.2. Representaciones de codificador bidireccional de transformadores

BERT (*Bidirectional Encoder Representations from Transformers*) es un modelo *open source* de PNL desarrollado por Google en el año 2018. El modelo original fue entrenado con texto de Wikipedia y el *dataset* BookCorpus de Google.

Inicialmente, dos modelos de BERT fueron presentados:

- *base*: con 12 capas, 12 cabezas de atención y 110 millones de parámetros.
- *large*: con 24 capas, 16 cabezas de atención y 340 millones de parámetros.

Cada capa de BERT es un *Transformer Encoder*. Cuando se habla de, por ejemplo, 12 capas, son 12 de esos codificadores apilados uno encima de otro. Los datos de entrada son consumidos por la primer capa, y van pasando por cada una hasta llegar a la última. Como salida, habrá un *embedding* donde cada posición del *token* tendrá un vector de 768 posiciones para BERT *base* y 1024 para BERT *large* [28].

La innovación introducida por BERT es aplicar dos técnicas en la etapa de entrenamiento: MLM (*Masked Language Model*) y NSP (*Next Sentence Prediction*).

MLM consiste en dar a BERT una secuencia para que optimice sus pesos internos y produzca la misma secuencia. Sin embargo, previamente a dar a BERT la secuencia, se enmascaran ciertos *tokens* para que BERT los tenga que “adivinar”. La implicancia de esta técnica es que BERT termina aprendiendo del contexto de la oración para predecirlos.

Para lograr esto, BERT se apoya en un mecanismo de atención propia o *self-attention* posibilitado por los *Transformers* bidireccionales en el núcleo del diseño de BERT. Esto es importante ya que el significado de una palabra puede cambiar a medida que se desarrolla una oración. Cuantas más palabras haya en cada oración o frase, más ambigua se vuelve la palabra en cuestión. BERT se encarga de esta ambigüedad leyendo una oración bidireccionalmente, teniendo en cuenta el efecto de todas las palabras en la oración y no sólo las que la preceden [29].

En la figura 2.2 se puede ver cómo para una palabra, el modelo de IA hace una ponderación de la importancia del resto de las palabras en esa oración.

²Imagen tomada de <https://towardsdatascience.com/understand-self-attention-in-bert-intuitively-cd480cbff30b>

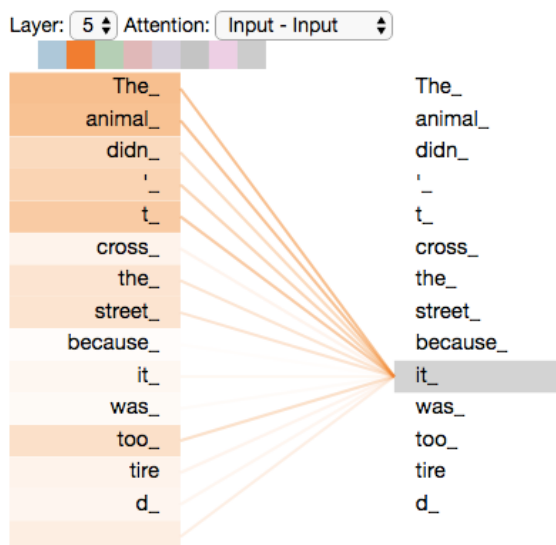


FIGURA 2.2. Ejemplo de funcionamiento del mecanismo de atención propia².

NSP, por otro lado, es otro mecanismo en donde el modelo recibe oraciones de pares como entrada y aprende a predecir si la segunda oración en el par es subsecuente de la primera. Durante el entrenamiento, la mitad de los datos son pares de oraciones subsecuentes y en la otra mitad la segunda oración es seleccionada de manera aleatoria.

Para saber distinguir cuándo comienza la primer oración y cuándo la segunda, la entrada es procesada de la siguiente forma:

1. Un *token* “[CLS]” se inserta al comienzo de la primer oración y un *token* “[SEP]” es insertado al final de las dos oraciones.
2. Se agrega un *embedding* que indica para cada *token* si corresponde con la oración A o B.
3. Se agrega un *embedding* posicional que indica para cada *token* su posición en la secuencia.

En la figura 2.3 se puede ver cómo se combinan estas dos técnicas cuando se entrena BERT.

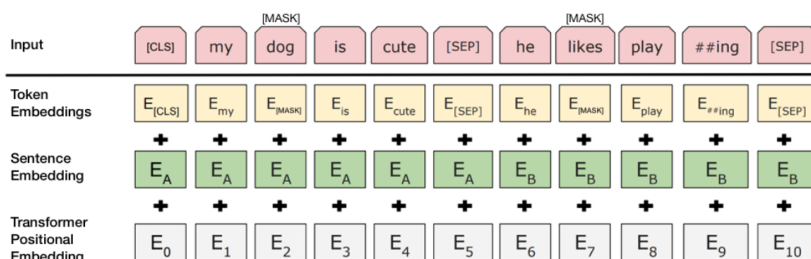


FIGURA 2.3. Ejemplo de entrada de datos a BERT con MLM y NSP combinados³.

³Imagen tomada de <https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270>

En la primera fila se observa cómo fueron separadas las dos oraciones. Luego en el *embedding* amarillo, se observa cómo algunos *tokens* son enmascarados para que luego la red los tenga que predecir. En el tercera, en verde, se observa la asignación de cada *token* a la oración A o B y por último, se observa un *embedding* con el orden de cada *token*.

2.4. Herramientas de software utilizadas

En esta sección se presentan las herramientas que se utilizaron tanto para la fase de entrenamiento como para el despliegue de los modelos.

2.4.1. Python

Python es un lenguaje de programación utilizado principalmente para desarrollo de aplicaciones y *machine learning*. Entre sus características se destaca que es orientado a objetos, multiplataforma e interpretado y su tipado es dinámico y fuerte [30].

2.4.2. GitHub

GitHub es un servicio web donde los usuarios pueden alojar repositorios con Git como sistema de control de versiones. Este sistema permite gestionar y rastrear cambios en el código a través del tiempo, ayudando a los equipos de desarrolladores a trabajar en proyectos de manera colaborativa [31].

Por otro lado, tiene una plataforma de integración y despliegue continuo llamada GitHub Actions. Esta plataforma permite crear y configurar la ejecución de flujos de trabajo o *workflows* cuando un evento sucede en el repositorio [32].

2.4.3. BigQuery

BigQuery es un servicio *serverless* de *Data Warehouse* de GCP (Google Cloud Platform). Los datos se guardan en colecciones de tablas, que a su vez, se agrupan en *datasets*. Cada columna de las tablas se almacena por separado, por eso se dice que BigQuery es una base de datos columnar [21].

El acceso a los datos se hace a través del lenguaje SQL. Los recursos que se necesitan para ejecutar una consulta se calculan dinámicamente en base a sus características y también en cómo esté configurada la tabla.

2.4.4. Vertex AI workbench

Vertex es un entorno de desarrollo especializado para ciencia de datos. Permite disponibilizar un *Jupyter notebook* que se integra con BigQuery y Google Cloud Storage para la lectura de datos y con GitHub para la sincronización de código. Además posee instancias con distintas configuraciones de CPU y GPU [33].

2.4.5. Docker

Docker es una herramienta que empaqueta *software* en unidades llamadas contenedores, que incluyen todas las dependencias para que el programa se ejecute:

ejecutables, archivos de configuración, bibliotecas, etc. Funciona de manera similar a una máquina virtual, solo que además de virtualizar el hardware también virtualiza el sistema operativo [34].

La principal ventaja que otorga el uso de Docker es que asegura que el contenedor se pueda ejecutar de manera fiable en cualquier plataforma compatible, como Kubernetes.

2.4.6. Artifact registry

Artifact Registry permite almacenar y administrar imágenes de Docker. Posibilita armar un *pipeline* de integración y despliegue continuo al permitir cargar una imagen de Docker para que pueda ser consumida desde las distintas plataformas de orquestación de contenedores [35].

2.4.7. Cloud Composer y Apache Airflow

Cloud Composer es un servicio de organización de flujos de trabajo administrado por Google. Está basado en el proyecto *open source* Apache Airflow [22].

Un flujo de trabajo representa una serie de tareas para trabajar con datos. Estos flujos son creados mediante grafos acíclicos dirigidos o DAGs (*Directed Acyclic Graphs*). Los DAGs se crean con código en Python que especifica su estructura.

Capítulo 3

Diseño e implementación

En este capítulo se describe cómo se han utilizado las herramientas mencionadas en el capítulo 2 y sus integraciones. Se presenta la arquitectura de alto nivel completa y luego se describe detalladamente desde el consumo de datos hasta la salida del proceso y su almacenamiento.

3.1. Arquitectura del sistema

En la figura 3.1 se puede observar la arquitectura de alto nivel que integra las herramientas utilizadas en la fase de entrenamiento y evaluación de los modelos de IA con las herramientas utilizadas para el despliegue.

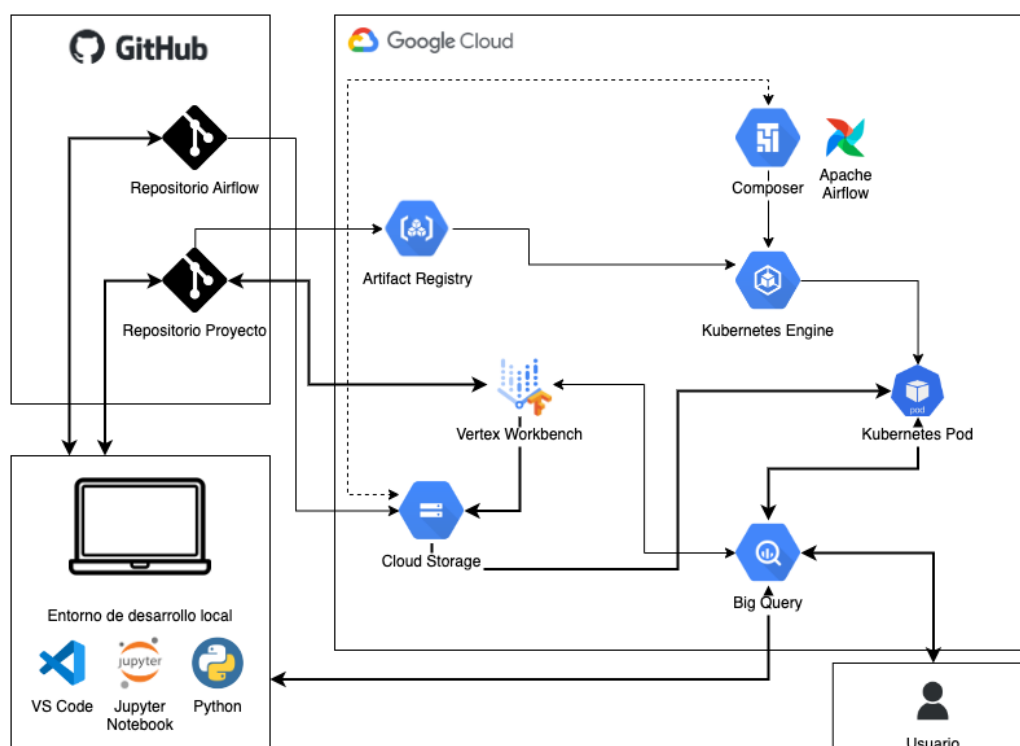


FIGURA 3.1. Diagrama de la arquitectura de alto nivel.

Como se puede ver en el cuadro inferior izquierdo, la mayor parte del desarrollo se realizó en una computadora local, que contaba con un intérprete de Python y Visual Studio Code con Jupyter para la escritura de código. Estos archivos se sincronizaban contra un repositorio de GitHub.

Los datos para realizar el entrenamiento de los modelos de IA fueron tomados del *data warehouse* BigQuery utilizando consultas SQL y bibliotecas de Google para Python para manejar su ejecución, como se puede ver en la flecha de la parte inferior que conecta BigQuery con el entorno de desarrollo.

Estos datos son llevados a BigQuery por un proceso ajeno a este desarrollo, que se encarga de sincronizar la base de Salesforce hacia el *data warehouse* diariamente. Particularmente, hay una tabla que contiene los reclamos y consultas de los usuarios, con su identificador de caso, fecha de creación y estado.

En las ocasiones en las que fue necesario un mayor poder de cómputo a través del uso de una placa de vídeo dedicada, se utilizó una instancia de Vertex AI Workbench en la nube de Google. Por lo general, estas situaciones se dieron al momento de realizar distintas pruebas con las redes neuronales BERT. Los modelos eran luego exportados al almacenamiento de la nube llamado Cloud Storage en formato "Pickle" o "HDF5" (*Hierarchical Data Format 5*). Esto se puede notar por la flecha que conecta la figura de Cloud Storage con la de Vertex Workbench.

En la parte superior izquierda se encuentran los repositorios creados en GitHub. El repositorio del proyecto cuenta con un Workflow de GitHub Action que se corre manualmente y genera una imagen de Docker, incluyendo bibliotecas y librerías necesarias, archivos de código y de configuración. Esta imagen de Docker es enviada al Artifact Registry de la nube de Google y almacenada allí.

Asimismo, se trabajó en otro repositorio que está destinado exclusivamente a la administración de los flujos de trabajo de Apache Airflow que tiene el área de *machine learning*. Las actividades realizadas incluyeron agregar un archivo de Python con el código del DAG para administrar la ejecución de la predicción.

El repositorio de Airflow cuenta con un Workflow de GitHub Action que sincroniza el código de los DAGs contra un Cloud Storage que está conectado a la instancia del orquestador. Airflow periódicamente controla si hay un flujo de trabajo nuevo y cuando lo detecta, lo carga.

En la parte superior derecha del diagrama se puede ver como al momento de ejecutar el proceso, Airflow utiliza un operador dentro del DAG para conectarse con la plataforma de Kubernetes e instanciar un Pod. Este Pod ejecuta los archivos de código del contenedor que se empaquetaron en la imagen de Docker cargada previamente en el Artifact Registry.

Al finalizar, se guardan los resultados de las predicciones en una tabla en BigQuery y el Pod es eliminado de Kubernetes. Los datos quedan disponibles en la base para que cualquier usuario con los suficientes permisos pueda consumirlos ejecutando una consulta.

3.2. Extracción y preprocesamiento del texto

En esta sección se describen en detalle los pasos realizados para obtener los datos crudos y aplicarles un preprocesamiento para poder usarlos de entrada en el posterior entrenamiento de los modelos de IA.

3.2.1. Extracción de datos de BigQuery

Inicialmente se realizó un relevamiento de las tablas existentes, y en particular se encontró una tabla que contenía información de los reclamos y consultas.

Las columnas de interés de esa tabla fueron:

- El identificador del caso.
- El número del caso.
- La categoría L2 (de segundo nivel).
- La categoría L3 (de tercer nivel).
- El estado del caso.
- La descripción: el contenido del mensaje que escribió el usuario.
- La fecha de creación.
- La fecha de cierre.

Para la fase de entrenamiento, se limitó la extracción de casos a los que tenían fecha de creación entre enero de 2021 y diciembre de 2022. Además, su estado debía ser “Resuelto” o “Cerrado”. Sin embargo, por la poca cantidad de datos que había para algunos clasificadores L2, fue necesario realizar consultas adicionales para tomar casos de 2023. Una vez obtenidos los datos, se guardaron en archivos “Parquet” para su posterior uso con la librería Pandas para Python.

Por otro lado, el área de atención al cliente maneja un archivo CSV (*comma-separated values*) con la jerarquía de casos L1, L2 y L3. Es decir, para cada L1, qué casos L2 le corresponden, y para cada uno de esos L2, qué casos L3 le corresponden. Las categorías L3 fueron ignoradas para este trabajo, por lo que se hizo foco en las primeras dos.

En total, se encontraron 15 categorías L1 y 122 categorías L2. Luego de una reunión con el área de atención al cliente para un mejor entendimiento de cada una, se descartó una categoría especial. Luego de este refinamiento quedaron 14 L1 y 117 L2.

En la figura 3.2 se puede observar para cada categoría L1 cuántas subcategorías L2 contiene.

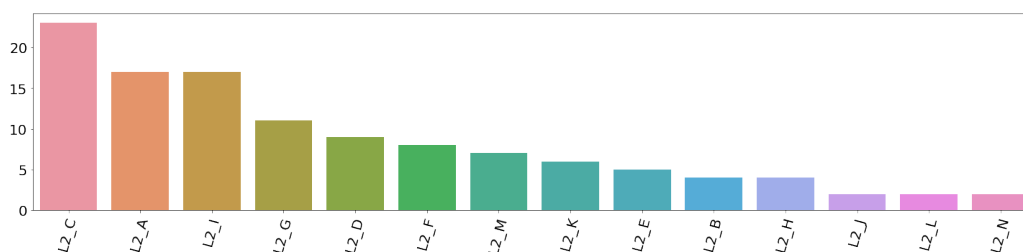


FIGURA 3.2. Cantidad de categorías L2 para cada L1.

Para trabajar en la fase de preprocesamiento, se combinó el archivo de los casos con el archivo que contenía la jerarquía de categorías, y se obtuvo la L1 de cada reclamo y consulta.

3.2.2. Pipeline de preprocesamiento

El preprocesamiento es una de las tareas más importantes en PNL. Los datos en crudo pueden contener información que no es relevante para la tarea en cuestión, que se debe eliminar. Por otro lado, muchos modelos de IA requieren un formateo previo de los datos para poder consumirlos.

En la figura 3.3 se detallan los pasos realizados para “limpiar” el texto.

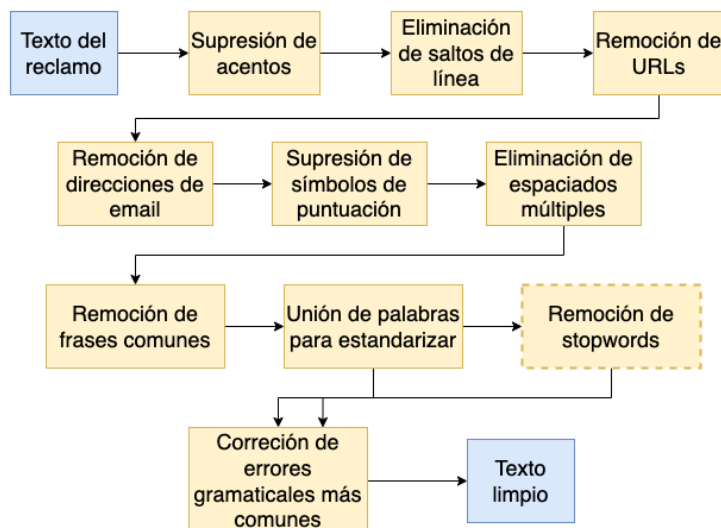


FIGURA 3.3. Diagrama del procesamiento del texto crudo.

El proceso comienza con el texto original del reclamo tal cual quedó guardado en la base. Luego se le aplican las siguientes acciones:

1. Se suprimen los acentos de las palabras (si los hubiera) utilizando una expresión regular.
2. Se eliminan los saltos de línea.
3. Se sacan URLs en el mensaje en caso de que hubiese hipervínculos.
4. Se sacan direcciones de *email*. Estos por lo general aparecen cuando el reclamo se hace a través de ese medio.
5. Se suprimen signos de puntuación utilizando otra expresión regular.
6. Se eliminan espacios múltiples para normalizar en un espacio entre dos palabras.
7. Se remueven frases que se repiten en muchos reclamos: algunos casos quedan registrados en la base con la respuesta por parte de la empresa. Estos mensajes tienen una plantilla de base con frases que luego se repiten entre casos.
8. Se unen palabras con un significado específico: por ejemplo “pedidos ya” se deja como “pedidosya” porque hace referencia a una empresa.
9. Para el caso de los modelos de CNB entrenados, se aplica un paso extra para eliminar *stopwords*.
10. Por último se realizó un análisis con los datos de entrenamiento para ver los errores gramaticales más comunes utilizando la librería PyEnchant para

Python. De esto se obtuvo un listado con los errores más comunes y un mapeo a su versión correcta. En este paso se aplica la corrección sobre ese listado de palabras.

Una vez obtenido el texto limpio, se procede a pasarlo a un vector numérico para que sea interpretado por los modelos de IA. Se distinguen dos formas de hacerlo, dependiendo de si se usa el modelo de CNB o el de BERT.

Codificación para CNB

En la figura 3.4 se explica el proceso para obtener los vectores numéricos para entrenar el modelo de CNB. Al texto limpio se le aplica un paso de segmentación para obtener los *tokens* que luego serán vectorizados utilizando el método TF-IDF. Por otro lado, a la variable objetivo que también está en formato de texto, se le aplica el proceso de *label encoding* para obtener una variable numérica.

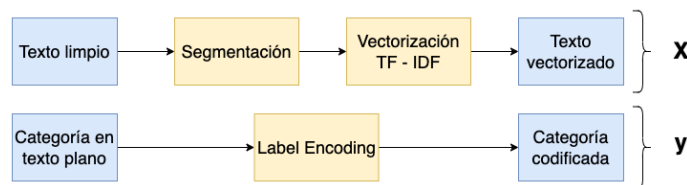


FIGURA 3.4. Diagrama de codificación para CNB.

Codificación para BERT

En la figura 3.5 se puede ver el proceso para el caso de BERT. Se puede notar que es un poco más largo que el de CNB.

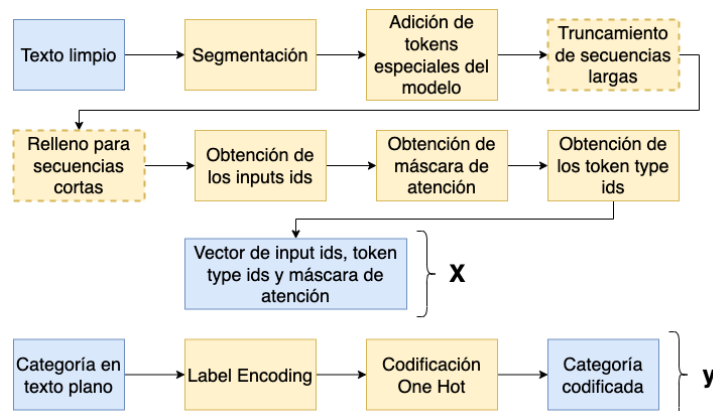


FIGURA 3.5. Diagrama de codificación para BERT.

El algoritmo de segmentación de BERT se llama *WordPiece* y tiene la particularidad de que no sólo segmenta por palabras sino que también lo hace por subpalabras y por caracteres. Comienza segmentando por todos los caracteres del vocabulario con el que fue entrenado y luego los va uniéndolos en pares.

Utiliza la siguiente fórmula para computar los puntajes de los pares de *tokens* a unir:

$$puntuaje = \frac{\text{frecuencia del par}}{\text{frecuencia del primer elemento} \times \text{frecuencia del segundo elemento}} \quad (3.1)$$

Luego une los pares con mayor puntaje y re-calcula para todos los *tokens* resultantes. Este proceso es repetido hasta alcanzar un umbral para el puntaje o hasta alcanzar un límite de *tokens*.

El vocabulario del modelo pre-entrenado de BERT que se utilizó está compuesto por 977 *tokens* reservados del estilo “[MASK]” o “[PAD]” por ejemplo y luego por *tokens* de caracteres individuales, subpalabras y palabras. [36]

En la figura 3.6 se puede ver un ejemplo de cómo es segmentada una oración.

Original: BETO es clave para el desarrollo del NLP en América Latina.
 Tokenizado: ['BE', '##TO', 'es', 'clave', 'para', 'el', 'desarrollo', 'del', 'NL', '##P', 'en', 'América', 'Latina',
 IDs: [13065, 6524, 1058, 5714, 1110, 1040, 1927, 1072, 15938, 30966, 1036, 4109, 9062, 1009]

FIGURA 3.6. Ejemplo de segmentación *WordPiece*¹.

Luego de hacer la segmentación de cada frase, se hace un truncamiento de las secuencias largas a un máximo preestablecido y se realiza un relleno a las secuencias cortas con un *token* especial hasta ocupar ese mismo máximo.

Los siguientes pasos son obtener los *input ids* que son los identificadores de los *tokens*, obtener la máscara de atención que marca qué *tokens* la red debe mirar y por último obtener los *token type ids* que marcan a qué secuencia pertenece cada *token* (recordar que BERT se entrena de a pares de secuencia).

Por otro lado, la codificación de la variable objetivo es igual que en CNB solo que a la variable numérica obtenida se le aplica una codificación *one hot* adicional.

3.3. Entrenamiento de los modelos

En esta sección se describe en detalle los análisis exploratorios realizados sobre los datos, cómo fue concebida la solución de los modelos de IA y las técnicas utilizadas para entrenar los modelos.

3.3.1. Análisis exploratorio

En la figura 3.7 se observa la proporción de casos de cada categoría L1 sobre el total. Se puede ver que hay una que tiene casi el 50 %.

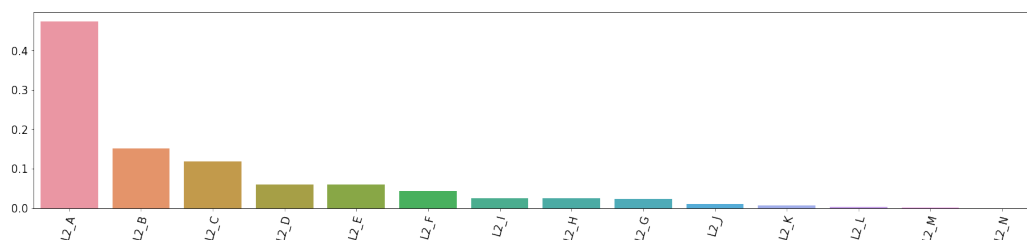


FIGURA 3.7. Proporción de casos.

¹Imagen tomada de <https://espejel.substack.com/p/beto-bert-01-importacion-y-tokenizing>

También se realizó un análisis de las longitudes de los textos. En promedio, la cantidad de palabras en los mensajes fue de 78 previo a aplicar preprocesamiento, 71 luego de aplicar una parte (se dejan las *stopwords*) y 44 cuando se aplicó el preprocesamiento completo.

En la figura 3.8 se puede observar la longitud promedio para cada categoría. En cada barra se puede ver la longitud del texto sin preprocesamiento, con preprocesamiento dejando *stopwords* y también removiéndolas. Hay dos categorías cuya longitud promedio es particularmente elevada.

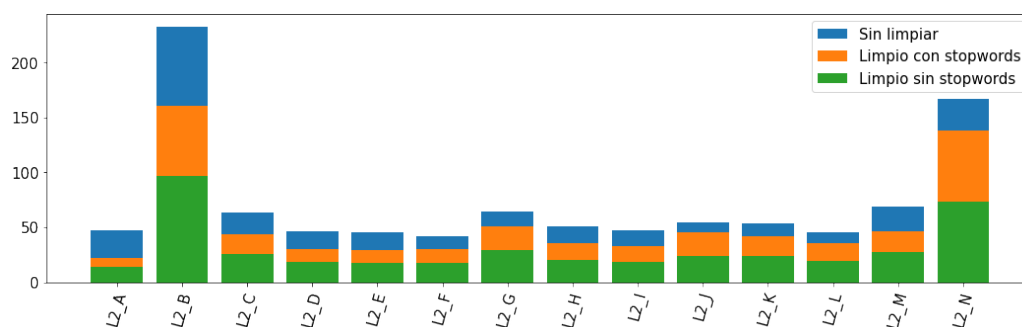


FIGURA 3.8. Longitud de los mensajes.

3.3.2. Integración de los modelos

A continuación se presenta la figura 3.9 donde se puede ver cómo fue concebida la arquitectura de la solución de los modelos de IA.

Se tiene un clasificador que en primera instancia hace la predicción de las categorías de primer nivel, y luego se tienen clasificadores para cada subcategoría de segundo nivel. Esto significa que en total se entrenaron 15 modelos: el general y 14 de subcategorías.

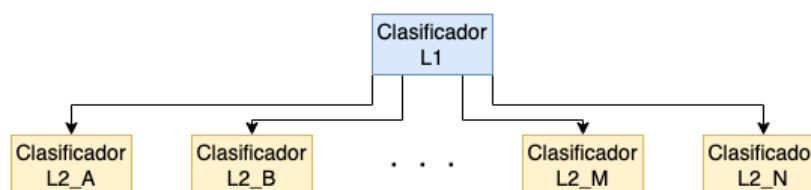


FIGURA 3.9. Diagrama general de los modelos de IA.

Este diseño presenta una serie de ventajas, entre las que principalmente se destaca su diseño modular. Esto es importante para la etapa de soporte a futuro, ya que si surge una nueva categoría o se quiere mejorar una parte de la clasificación, sólo es necesario trabajar en el modelo afectado dejando los demás intactos.

Además, al hacer que cada clasificador tenga que realizar una tarea más específica, su *head* de clasificación también queda más simple.

3.3.3. Entrenamiento de los modelos base

Un modelo base o *baseline* es un modelo simple que actúa como referencia y sirve para tener con qué comparar cuando se entrenan modelos más complejos.

Las ventajas de utilizar un modelo base son las siguientes [37]:

- Se entrenan rápidamente y sin recursos especializados.
- Permiten tener un mejor entendimiento en etapas tempranas.
- Sirven para encontrar errores y poner a prueba suposiciones.
- Sirven para medir el progreso de los modelos más complejos.

Como se mencionó en el capítulo 2, para este trabajo el *baseline* elegido fue CNB de la librería Scikit-Learn. La clase provista se llama *ComplementNB* y provee un método *fit* para realizar el entrenamiento y un método *predict* para realizar una predicción sobre datos que se pasan como parámetro.

El entorno donde se realizó el entrenamiento fue una computadora local.

Modelos base en validación cruzada

Hubo algunas categorías en particular que cuando se quiso entrenar el clasificador L2, contaban con muy pocos datos incluso tomando del año 2023. Estas categorías fueron:

- L2_M.
- L2_N.

La primera contaba con un total de 200 casos y cuando se incluyeron los de 2023 se llegó a tener 267. Para este caso, lo que se hizo fue obtener una métrica en validación cruzada usando la clase *RepeatedStratifiedKFold* de Scikit-Learn con 5 *folds* y 5 *repeats*. Luego se procedió a entrenar el modelo CNB final utilizando el set de datos completo.

La segunda contaba originalmente con sólo 2 casos y cuando se incluyeron los de 2023 se llegó a 8. Este resultó ser un caso muy extremo, donde se utilizó otra técnica de validación cruzada con la clase *LeaveOneOut* de Scikit-Learn. Lo que hace esta técnica es ir apartando un dato por *fold* y entrenar con el resto. Una vez tomada la métrica, se entrenó el CNB final con todos los datos.

3.3.4. Entrenamiento de BERT

Como los textos del dataset estaban escritos en español, se utilizó la red BETO. Este modelo es un modelo BERT entrenado con el dataset *spanish unannotated corpora* que cuenta con casi tres millardos de palabras. Está disponible dentro de la librería Transformers de HuggingFace para Python, tanto en Pytorch como en Tensorflow. Para este trabajo se utilizó esta última [38].

Por otro lado, estos modelos pre-entrenados deben ser adaptados para realizar tareas como clasificación de texto. En la figura 3.10 se muestra un esquema de entrenamiento de BETO.

Se puede notar que sobre el modelo pre-entrenado de BETO se agrega: un *head* de clasificación compuesto por perceptrones multicapa combinados con técnicas de regularización como *dropout* y por último *softmax* como función de activación para realizar la tarea de clasificación per se.

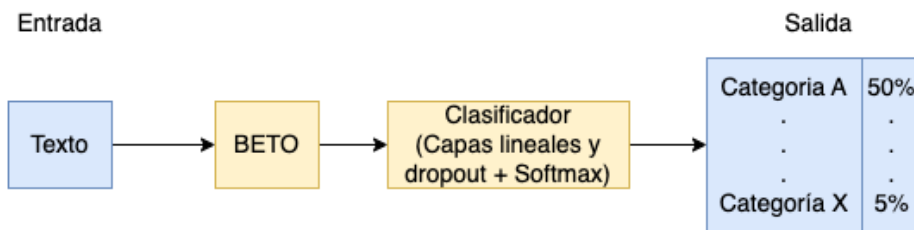


FIGURA 3.10. Esquema de entrenamiento de BERT para clasificación de texto.

Aprendizaje por transferencia

El aprendizaje por transferencia consiste en entrenar un modelo en un conjunto de datos a gran escala y luego usar ese modelo previamente entrenado para llevar a cabo el aprendizaje para otra tarea posterior (la tarea objetivo) [39].

Se distinguen tres métodos para transferir el aprendizaje:

- Extracción de características: se usan las capas pre-entrenadas para extraer características o *features* de los datos. Los pesos de las capas inferiores no se actualizan durante el *backpropagation*.
- Ajuste fino: se ajustan todos los parámetros del modelo.
- Extracción de capas: se extraen sólo las capas necesarias para la tarea en cuestión.

Para este trabajo, el aprendizaje por transferencia consistió en utilizar el modelo pre-entrenado de BERTO y luego ajustarlo para la tarea de clasificación de reclamos.

En primera instancia se utilizó el método de extracción de características que consistió en congelar las capas de BERTO y dejar libre el *head* de clasificación para entrenar sus pesos. Para esto, se dejó el *learning rate* default de Tensorflow.

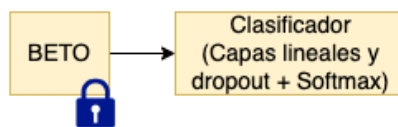


FIGURA 3.11. Técnica de extracción de características.

Luego, se realizó un ajuste fino, dejando libres todas las capas del modelo para ajustar todos sus pesos, esta vez con un *learning rate* más bajo.

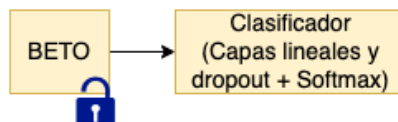


FIGURA 3.12. Técnica de ajuste fino.

La técnica de extracción de capas no fue utilizada en este trabajo.

Entorno de entrenamiento

Como se menciona en el capítulo 2, para entrenar a BETO se utilizó un entorno de trabajo en la nube llamado Vertex AI Workbench. Se diferencian dos tipos de instancias:

- Los *notebooks* administrados: son entornos administrados por Google.
- Los *notebooks* administrados por el usuario: permiten una mayor personalización y son ideales cuando se necesita mayor control sobre el entorno.

La instancia utilizada fue la segunda: las administradas por el usuario. El hardware con el que contaba se explica en la tabla 3.1:

TABLA 3.1. Especificaciones técnicas del entorno de entrenamiento.

Componente	Descripción
Versión del ambiente	M107
Tipo de máquina	n1-standard-8
Versión de CPU	Intel Xeon E5
Nro. de núcleos virtuales	8 vCPUs
Memoria RAM	30 GB
Nro. de placas de video	1
Versión de placa de video	NVIDIA T4
Memoria de placa de video	16 GB GDDR6
Almacenamiento	1 HDD x 200 GB

Por otro lado, se decidió utilizar el entorno TensorFlow Enterprise 2.11 como imagen virtual de la instancia, lo que permitió tener todos los *drivers* de la placa de video pre-instalados y además contaba con el sistema de paquetes Conda con las librerías de TensorFlow para Python también pre-instaladas. Tener esa configuración cubierta por la nube de Google resultó en una gran ventaja porque evitó tener que analizar la compatibilidad entre el hardware y el software.

Puntos de control

Para evitar pérdida de información accidental en la etapa de entrenamiento se utilizó una clase de TensorFlow que se llama ModelCheckpoint.

ModelCheckpoint es un *callback* que, dependiendo de cómo se configure, puede ejecutarse al finalizar una época o cada cierta cantidad de lotes o *batches*. Su función principal es guardar los pesos de la red y el estado del optimizador en almacenamiento permanente para no perder esa información en caso de que se apague la instancia o surja algún error. Se pueden configurar criterios de guardado como, por ejemplo, sólo si mejoró una métrica o la función de pérdida.

La configuración usada se explica en la tabla 3.2:

TABLA 3.2. Configuración del ModelCheckpoint.

Parámetro	Valor
Nombre del archivo	Nombre del archivo en extensión h5
Métrica a monitorear	Pérdida para lote de entrenamiento
Guardar sólo el mejor	Sí
Modo de la métrica	Minimizar
Frecuencia de guardado	En cada época

Parada temprana

Otro de los mecanismos de prevención de sobreajuste o *overfitting* utilizados fue la parada temprana o *early stopping*. La idea detrás de esta técnica es detener el entrenamiento cuando el modelo empieza a disminuir su capacidad de generalización.

Esto se puede notar en la imagen 3.13 donde la función de pérdida para el lote de entrenamiento continúa disminuyendo pero para el lote de validación no.

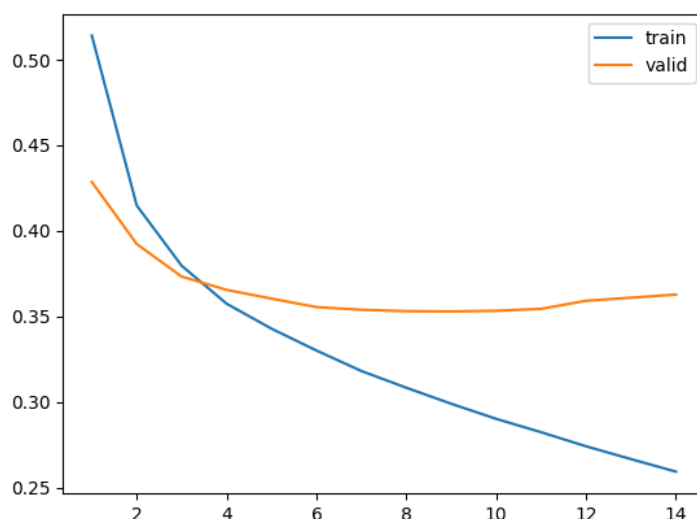


FIGURA 3.13. Funciones de pérdida a través de las épocas.

La clase de TensorFlow utilizada para esto fue EarlyStopping. También es un *callback* pero este lo que hace es ir monitoreando una métrica que se pasa como parámetro y luego de un par de épocas en las que no mejora o empeora, detiene el entrenamiento.

La configuración utilizada se puede ver en la tabla 3.3:

TABLA 3.3. Configuración de EarlyStopping.

Parámetro	Valor
Métrica a monitorear	Pérdida para lote de validación
Delta mínimo	0
Paciencia	5 épocas
Verboso	Sí
Modo	Automático
Restaurar mejores pesos	Sí
Comienzo	Época 0

Variaciones en el entrenamiento

Al momento de realizar los distintos entrenamientos de BETO, se analizaron qué variaciones se podían realizar en sus etapas desde el preprocesamiento hasta el entrenamiento per se, con el objetivo de intentar maximizar el desempeño del modelo.

Una de esas variaciones fue dejar o remover las *stopwords*. Esto se puede ver en la imagen 3.3 donde el paso de “Remoción de *stopwords*” aparece en líneas punteadas. Por la forma en que los modelos de BERT aprenden del contexto, en todos los resultados el desempeño mejoró al dejarlas.

Otra variación que se intentó probar fue cambiar el largo de la secuencia. Los modelos de BERT admiten hasta un máximo de 512 *tokens*. El desafío aquí es que a mayor largo de secuencia, habrá mayor consumo de memoria, y la placa de video utilizada estaba limitada en 16 GB. Otra consecuencia es que al consumir más memoria, obliga a reducir el *batch size*.

Para la primer parte del entrenamiento, donde las capas inferiores estaban bloqueadas, se utilizó un largo de secuencia de 128 con el tamaño del *batch* en 128 muestras.

Para la segunda parte del entrenamiento, donde se realizó un ajuste fino a todas las capas de la red, se utilizó también un largo de secuencia de 128 pero en este caso hubo que reducir el tamaño del *batch* a 24 muestras.

Cuando recién se estaba comenzando con los entrenamientos, se probó utilizar un largo de secuencia de 512 y con un *batch* de 8, pero estas pruebas fueron desestimadas porque no había mejoras significativas en el desempeño y el entrenamiento demandaba mucho tiempo.

Técnicas para el desbalance de clases

La distribución de categorías presentaba un desbalance por la naturaleza misma de los reclamos. Hay reclamos con mayor probabilidad de ocurrencia que otros, ya sea porque una funcionalidad se utiliza más, porque tiene mayor propensión a fallar o simplemente porque genera más consultas por parte de los usuarios.

Entre los métodos disponibles para tratar el desbalance había [40] [41]:

- Reagrupar categorías minoritarias en una sola.

- Sobremuestreo aleatorio: consiste en repetir casos de las clases minoritarias de forma aleatoria.
- Submuestreo aleatorio: consiste en eliminar casos de las clases mayoritarias de forma aleatoria.
- *Text augmentation*: consiste en realizar manipulaciones sobre los textos originales como, por ejemplo, reemplazar palabras por sus sinónimos de forma aleatoria, para generar nuevos casos de las clases minoritarias.
- *Hidden space augmentation*: consiste en generar nuevos vectores en el espacio de representación de BERT reemplazando partes pequeñas de un vector con otras de otro vector de la misma clase.
- Pesos para las clases: consiste en modificar la función de pérdida de la red por medio del uso de un vector con pesos ponderados para cada clase, dándole mayor importancia a las clases minoritarias.

La primera opción, la de reagrupar categorías, fue descartada porque el área de atención al cliente tiene categorías de reclamo pre-definidas que no se pueden modificar.

La opción de sobremuestreo podía generar sobreajuste por lo que fue descartada. Por otro lado, la opción de submuestreo también fue descartada porque implicaba pérdida de información importante para los reclamos que tienen mayor probabilidad de ocurrencia, y que en definitiva son los que suelen tratar más a menudo en el área.

Las opciones de *augmentation* no llegaron a probarse por su complejidad y tiempo adicional que llevaría en el desarrollo. Sin embargo, pueden ser una gran alternativa para probar en caso de que se quiera seguir evolucionando el modelo a futuro.

Para este trabajo, se decidió probar *class weights* o pesos para las clases utilizando la clase `class_weight` de Scikit-Learn para generar el vector de pesos y luego pasándolo como parámetro a TensorFlow en la etapa de entrenamiento.

Optimizadores

Inicialmente, en los entrenamientos realizados se utilizó el optimizador Adam, que es el que viene por defecto en la librería de TensorFlow. De hecho, cuando fue lanzado BERT, sus autores mencionan que [42]:

- Utilizaron el optimizador Adam con un *batch size* de 256, un *learning rate* de $1e-4$ y 40 épocas para el pre-entrenamiento.
- Utilizaron el optimizador Adam con un *batch size* de 16 o 32, un *learning rate* de $5e-5$, $3e-5$ o $2e-5$ y un número de épocas de entre 2 y 4 para realizar el ajuste fino.

Sin embargo, diversos artículos mencionan que utilizar el optimizador LAMB (*Layer-wise adaptive moments optimizer for batch training*) tiene un mejor desempeño en pruebas, una mejor capacidad de generalización y además permite entrenar con *batches* de mayor tamaño y *learning rates* más grandes, por lo que también se decidió probar con este optimizador para los entrenamientos [43] [44].

El algoritmo está basado en Adam, por lo que es muy similar a este. La principal diferencia es que calcula una relación conocida como “factor de confianza” que sirve para normalizar los gradientes y evitar divergencias. Esta normalización es doble: (i) normaliza por dimensión con respecto a la raíz cuadrada del segundo momento utilizado en Adam y (ii) normaliza por capas.

En la figura 3.14 se pueden observar los valores de la función de pérdida y el nivel exactitud en pruebas para Adam (en azul) y LAMB (en rojo) a través del tiempo en el conjunto de datos MNIST.

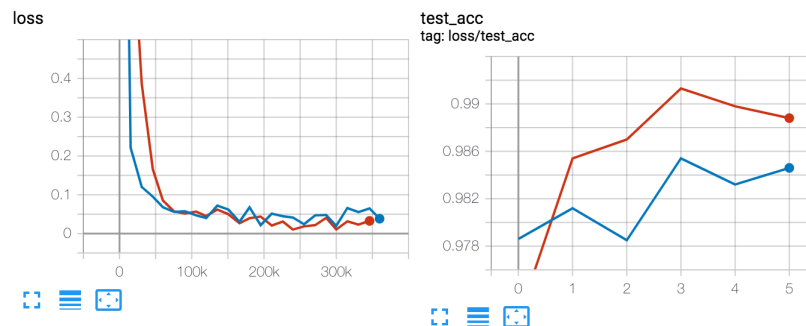


FIGURA 3.14. Función de pérdida y exactitud para Adam (azul) y LAMB (rojo) para MNIST².

3.4. Empaquetamiento del código y artefactos

Para empaquetar el código se utilizó la herramienta Docker en conjunto con un GitHub Workflow desarrollado en el mismo repositorio. Los pasos realizados fueron los siguientes:

1. Escribir los archivos de código que hacen la predicción.
2. Escribir un archivo con las librerías de Python necesarias.
3. Escribir el GitHub Workflow para generar una imagen de Docker y cargarla al Artifact Registry.
4. Cargar las credenciales de la nube de Google como *secret* en el repositorio.
5. Escribir el Dockerfile.
6. Ejecutar el Workflow y verificar la creación de la imagen en el Artifact Registry.

A continuación, en la figura 3.15 se puede ver la integración de los distintos componentes para generar una imagen de Docker.

En el repositorio de GitHub se encuentran los archivos de código y el Dockerfile. Además dentro del mismo repositorio se encuentra un archivo “.yaml” que describe un GitHub Workflow. Este archivo contiene una secuencia de pasos (los GitHub Actions) que deben ejecutarse para generar una imagen de Docker. Estos Actions son ejecutados por un GitHub Runner, que son máquinas virtuales hospedadas por la plataforma. Entre otras cosas, se ejecutan Actions para clonar

²Imagen tomada de <https://towardsdatascience.com/an-intuitive-understanding-of-the-lamb-optimizer-46f8c0ae4866>

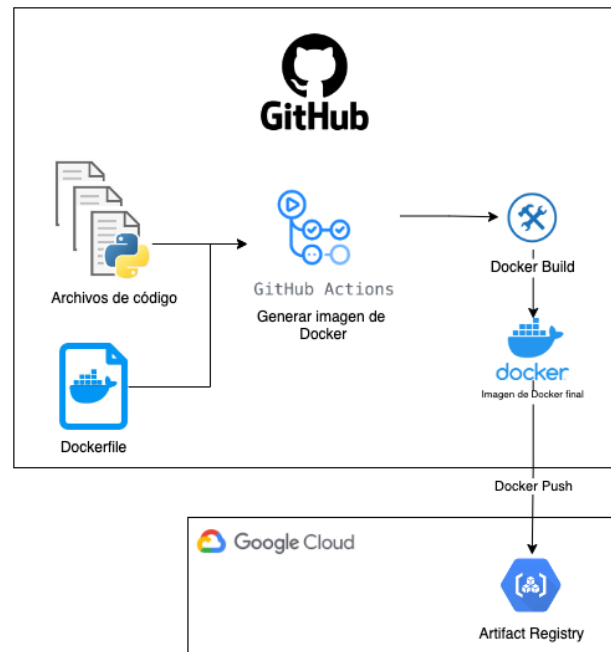


FIGURA 3.15. Interacción de los distintos componentes.

el código en el Runner, autenticarlo contra la nube de Google, configurar Docker para conectarse al Artifact Registry y comenzar el proceso de construcción.

En la figura 3.16 puede verse el menú de GitHub desde donde ejecutar el flujo manual para generar la imagen de Docker.

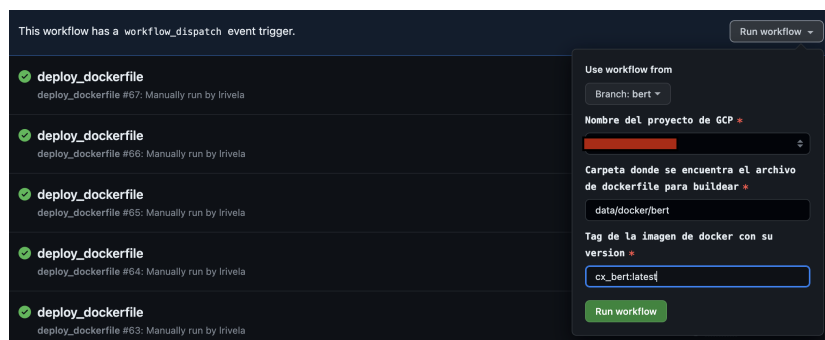


FIGURA 3.16. Menú de GitHub para ejecutar el flujo de trabajo.

El Dockerfile es un archivo que incluye instrucciones para crear una imagen de Docker. En la figura 3.17 puede verse su estructura en detalle.

Este archivo debe comenzar con una sentencia *"FROM"* que indica la imagen padre desde la cual se va a crear la imagen. Para este trabajo se utilizó una imagen de la empresa Nvidia que está basada en el sistema operativo Ubuntu y que incluye todos los *drivers* necesarios para utilizar una placa de video. La imagen se encuentra alojada en un repositorio público llamado Docker Hub.

Con el comando *"WORKDIR"* se configura el directorio de trabajo dentro de la imagen de Docker y luego con el comando *"COPY"* se le agregan los archivos Python y SQL del repositorio.

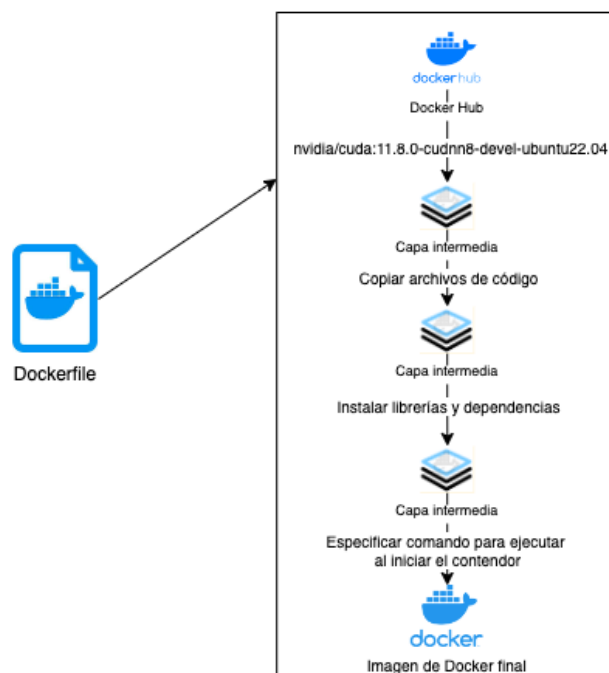


FIGURA 3.17. Vista detallada de un Dockerfile.

Con la ayuda del comando “*RUN*” se instalan paquetes de Linux, el intérprete de Python y sus dependencias. Por último se especifica el comando a ejecutar para iniciar el contenedor de Docker con la sentencia “*CMD*”.

3.5. Desarrollo del *pipeline* de predicción

En esta sección se describe cómo se ejecuta el código en la nube de Google para generar las predicciones del modelo y su posterior almacenamiento en una base de datos.

3.5.1. Creación y ejecución del flujo de trabajo

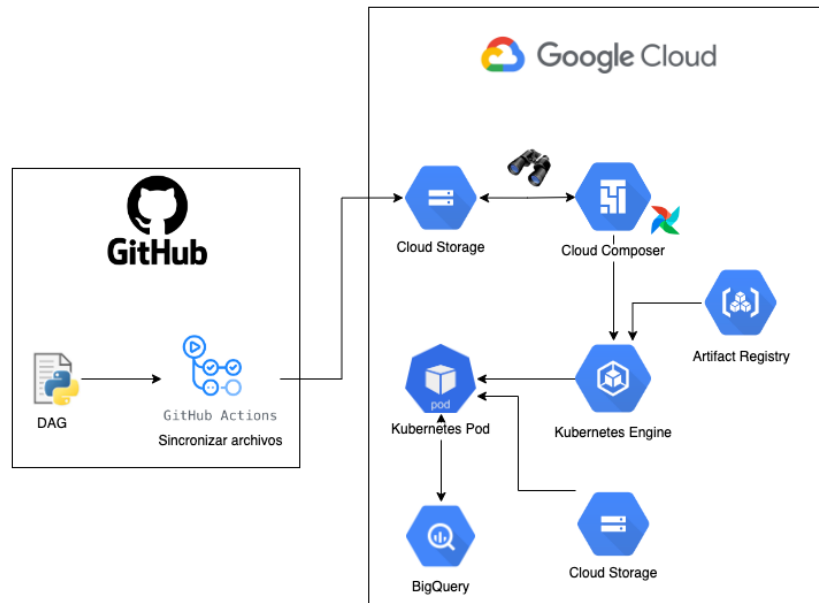
La herramienta seleccionada para dirigir la ejecución de código fue Apache Airflow, que en su versión administrada por la nube de Google se llama Cloud Composer.

En la figura 3.18 puede verse cómo es el flujo para cargar un DAG a Airflow, desde su creación en el repositorio de código hasta su ejecución en un Pod de Kubernetes.

El proceso comienza generando un DAG en el repositorio dedicado para Airflow, que tiene un GitHub Workflow que sincroniza el archivo contra un contenedor de Cloud Storage llamado *bucket*.

El Cloud Composer está continuamente monitoreando si se cargaron nuevos archivos al *bucket* que tiene configurado, y cuando así sucede, los carga en la base de Airflow. Luego procede a mostrarlo en su portal *web*.

Un DAG puede ser programado para ejecutarse a intervalos regulares de tiempo o también puede ser ejecutado manualmente. Cuando se ejecuta, se comienzan a

FIGURA 3.18. Interacción de los componentes del *pipeline*.

ejecutar distintos operadores en su flujo de trabajo. En la figura 3.19 puede verse la estructura del DAG desarrollado.

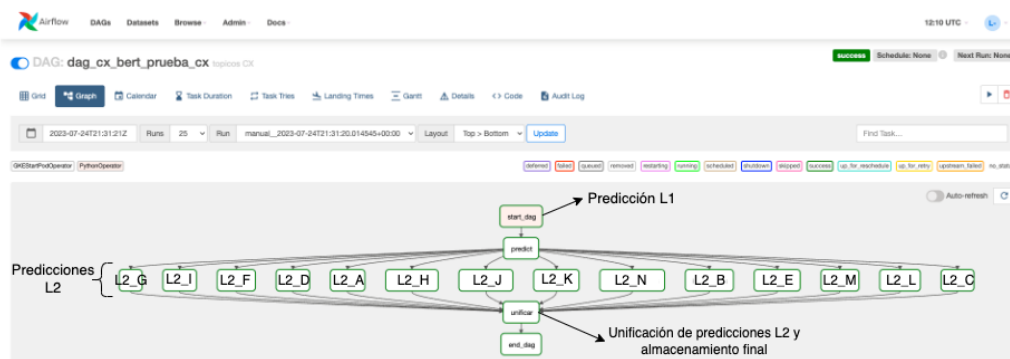


FIGURA 3.19. Vista en forma de grafo del DAG desde el menú de Airflow.

Todos los operadores que componen este DAG son del tipo `GKEStartPodOperator`. Este operador hace que Airflow se conecte con el motor de Kubernetes Engine que se pase como parámetro e instancie un Pod. Las configuraciones más importantes que se utilizaron fueron:

- El nombre del *cluster* de GKE donde se va a instanciar el Pod.
- La localización del *cluster* de GKE.
- La imagen de Docker que se generó en el Artifact Registry con el código.
- Variables de entorno que se le van a pasar al Pod: qué archivo de Python ejecutar, las tablas donde guardar información temporal, los *datasets* de las tablas, el *storage* donde están los archivos con los pesos de los modelos, etc.
- Los recursos del pod (CPU, RAM, Cantidad de placas de video).

- El nodo de Kubernetes donde asignar la ejecución del Pod para asegurar que tenga placa de video.

Al final del DAG se define el ruteo entre los operadores para definir cómo se conectan y cómo son las reglas de ejecución. Para este trabajo, como todo se encuentra en un ambiente de desarrollo, la ejecución de los operadores es secuencial (va de a uno por vez).

Los estados por los que pasa un Pod pueden verse en la figura 3.20.

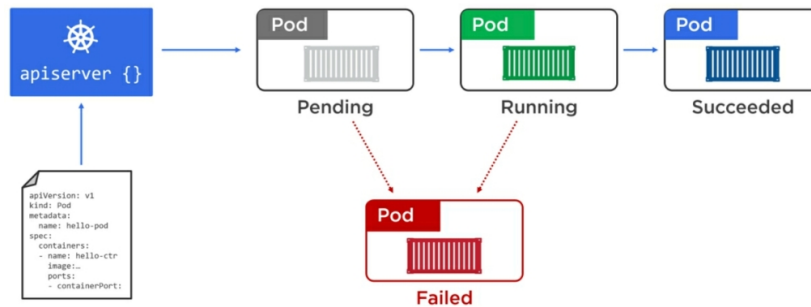


FIGURA 3.20. Estados del ciclo de vida de un Pod de Kubernetes³.

A continuación se explica cada uno de ellos [45]:

- *Pending*: Cuando el Pod ha sido aceptado por el *cluster* de Kubernetes, pero los contenedores aún no están listos. Por ejemplo, la imagen de Docker aún se está descargando.
- *Running*: Cuando el Pod ha sido asignado a un nodo de Kubernetes y tiene su contenedor corriendo.
- *Succeeded*: Cuando el contenedor del Pod ha terminado exitosamente. Por ejemplo, finalizó la ejecución del código de Python.
- *Failed*: Cuando el contenedor termina con un error. Por ejemplo, un error de programación en el código o un error en la configuración de Kubernetes.

3.5.2. Parametrización de los contenedores

En el momento de su creación, el contenedor recibe por medio del `GKEStartPodOperator` unas variables de entorno que le permiten saber para qué categoría está clasificando y ajustar su comportamiento en base a eso.

En la figura 3.21 se puede ver en detalle este comportamiento:

El contenedor recibe el nombre del archivo de Python que debe ejecutar:

- El de clasificación L1.
- El de clasificación L2 con BERT.
- El de clasificación L2 con CNB.
- El de unificación de las tablas temporales y generación de la tabla de predicción.

³Imagen tomada de <https://sunitc.dev/2020/12/26/kubernetes-pods/>

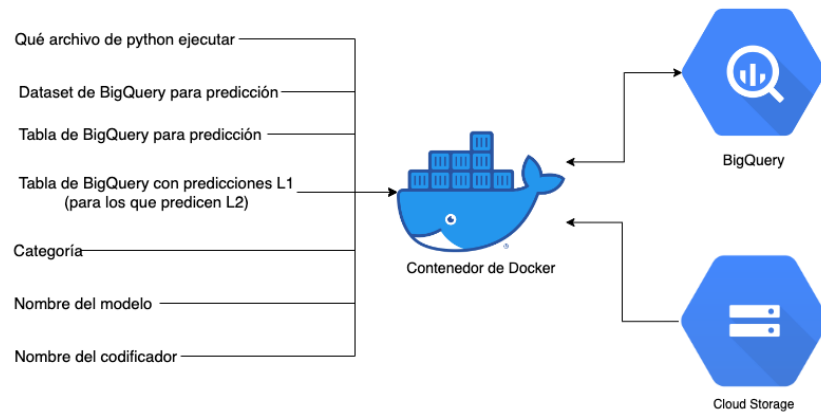


FIGURA 3.21. Diagrama de parametrización de un contenedor.

También recibe:

- El nombre de la tabla donde almacenar la predicción y si es el dataset temporal o es el definitivo (para el paso de unificación).
- La tabla con las predicciones temporales de L1 (para los pasos de clasificación L2).
- El nombre del modelo y el codificador para descargarlos desde el almacenamiento (para los pasos de predicción).

Capítulo 4

Ensayos y resultados

En este capítulo se presenta y describe la métrica elegida para evaluar el desempeño de los modelos, se muestran los resultados de desempeño de cada uno de los modelos entrenados y por último, se explican las simulaciones realizadas en un ambiente de desarrollo para probar la ejecución del flujo de trabajo de Airflow.

4.1. Métrica de evaluación de los modelos

Para evaluar un sistema de clasificación automático, en general se suele usar una matriz de confusión. Esta matriz es una tabla de doble entrada, donde cada columna muestra el número de predicciones de cada clase y cada fila muestra el número real de instancias de cada clase. En la figura 4.1 puede observarse una para clasificación multiclase:

MATRIZ DE CONFUSIÓN - $MC(i, j)$ de $N \times N$

CLASE I						CLASE N-I					
	Predicted = 1	Predicted = 2	Predicted = ...	Predicted = N-1	Predicted = N		Predicted = 1	Predicted = 2	Predicted = ...	Predicted = N-1	Predicted = N
Real = 1	TP	FN	FN	FN	FN	Real = 1	TN	TN	TN	FP	TN
Real = 2	FP	TN	TN	TN	TN	Real = 2	TN	TN	TN	FP	TN
Real = ...	FP	TN	...	TN	TN	Real = ...	TN	TN	...	FP	TN
Real = N-1	FP	TN	TN	TN	TN	Real = N-1	FN	FN	FN	TP	FN
Real = N	FP	TN	TN	TN	TN	Real = N	TN	TN	TN	FP	TN

FIGURA 4.1. Ejemplo de matriz de confusión multiclase genérica¹.

Para entender esta matriz, es necesario presentar cuatro conceptos que vienen asociados:

- Verdaderos positivo (VP o TP): la cantidad de positivos clasificados como positivos por el modelo.
- Falso negativo (FN): la cantidad de positivos clasificados como negativos por el modelo.

¹Imagen tomada de <https://wbarriosb.medium.com/calculando-la-precisi%C3%B3n-en-un-modelo-de-clasificaci%C3%B3n-multiclase-224d96f52043>

- Verdadero negativo (VN o TN): la cantidad de negativos clasificados como negativos por el modelo.
- Falso positivo (FP): la cantidad de negativos clasificados como positivos por el modelo.

Presentar los resultados en esta matriz permite calcular distintas métricas para evaluar características del sistema de clasificación. Por ejemplo:

La precisión, que está relacionada con la capacidad del modelo de no predecir un positivo cuando en realidad es un negativo:

$$\text{precisión}_k = \frac{MC(k, k)}{\sum_{i=1, j=k}^N MC(i, j)} \quad (4.1)$$

La exhaustividad, que está relacionada con la capacidad del modelo de detectar todos los positivos:

$$\text{exhaustividad}_k = \frac{MC(k, k)}{\sum_{j=1, i=k}^N MC(i, j)} \quad (4.2)$$

Para este trabajo, la métrica seleccionada para evaluar el desempeño de los modelos fue el puntaje F1 o *F1-Score*. Lo que hace esta métrica es calcular la media armónica entre la precisión y la exhaustividad (*recall*) del modelo. Su fórmula es la siguiente:

$$F1_k = 2 \times \frac{\text{precisión}_k \times \text{exhaustividad}_k}{\text{precisión}_k + \text{exhaustividad}_k} \quad (4.3)$$

De esta forma, se estaría obteniendo el F1 para cada clase. Para calcular un F1 global del sistema de clasificación, existen tres tipos de estrategia [46]

- *Macro average*: Consiste en calcular la media aritmética de todos los F1 calculados para cada clase. Esta estrategia le da a todas las clases la misma importancia en el cálculo del F1 global.
- *Micro average*: Consiste en calcular el F1 considerando el número total de TP, FP y FN, en lugar de hacerlo por cada clase. Esta estrategia calcula la proporción de las observaciones clasificadas correctamente sobre las observaciones totales.
- *Weighted average*: Consiste en calcular la media aritmética de todos los F1 calculados para cada clase, pero ponderando el soporte de cada clase (la cantidad de observaciones). Esta estrategia le da más importancia a las clases con más observaciones para el cálculo del F1 global.

Para *datasets* balanceados, es decir, cuando tienen una cantidad de observaciones parecida para cada una de las clases, se suele usar el *micro average*. No es el caso de este trabajo, donde por la naturaleza de los reclamos, hubo categorías que presentaron más observaciones que otras.

Dentro de las dos opciones restantes, se decidió por utilizar el *weighted F1*, ya que para medir el desempeño del modelo, se pretende dar una mayor importancia a las clases cuyos reclamos aparecen más a menudo.

La fórmula se muestra a continuación:

$$F1_{global} = \sum_{k=1}^N F1_k \times \text{soporte}_k \quad (4.4)$$

4.2. Resultados de desempeño en pruebas

En esta sección se presentan los resultados obtenidos para todos los modelos entrenados de cada categoría.

4.2.1. Resultados para clasificador L1

En la figura 4.2 se pueden observar los distintos entrenamientos realizados para el clasificador L1 y su puntaje obtenido:

Modelo	Entrenamiento	Class weights	Stopwords	Nro. de épocas	Optimizador	F1
CNB_1	-	-	-	-	-	0,8193
CNB_2	-	-	-	-	-	0,8604
BETO_1	Grueso	No	Sí	23 (early stopping)	LAMB	0,7324
BETO_1	Fino	No	Sí	14 (early stopping)	LAMB	0,8924
BETO_2	Grueso	No	Sí	9 (early stopping)	LAMB	0,4717
BETO_3	Grueso	Sí	Sí	19 (early stopping)	LAMB	0,7014

FIGURA 4.2. Resultados obtenidos para los modelos clasificadores L1.

A continuación se hacen algunas aclaraciones respecto de los modelos:

- Modelo CNB_1: los reclamos pertenecientes a la categoría especial que se menciona en la subsección 3.2.1 fueron eliminados.
- Modelo CNB_2: los reclamos pertenecientes a la categoría especial fueron redireccionados a otras categorías L1.
- Modelo BETO_1: los reclamos pertenecientes a la categoría especial fueron eliminados.
- Modelo BETO_2: los reclamos pertenecientes a la categoría especial fueron redireccionados a otras categorías L1.
- Modelo BETO_3: los reclamos pertenecientes a la categoría especial fueron eliminados.

4.2.2. Resultados para clasificador L2 A

En la figura 4.3 se pueden observar los distintos entrenamientos realizados para el clasificador L2 A y su puntaje obtenido:

Modelo	Entrenamiento	Class weights	Stopwords	Nro. de épocas	Optimizador	F1
CNB_1	-	-	-	-	-	0,7373
BETO_1	Grueso	No	No	10	ADAM	0,6891
BETO_1	Fino	No	No	13 (early stopping)	ADAM	0,7841
BETO_2	Grueso	Sí	No	19 (early stopping)	ADAM	0,6066
BETO_2	Fino	Sí	No	28 (early stopping)	ADAM	0,7196
BETO_3	Grueso	No	Sí	8 (early stopping)	ADAM	0,6993
BETO_3	Fino	No	Sí	12 (early stopping)	ADAM	0,7854
BETO_4	Grueso	No	Sí	10 (early stopping)	LAMB	0,7038
BETO_4	Fino	No	Sí	13 (early stopping)	LAMB	0,7849

FIGURA 4.3. Resultados obtenidos para los modelos clasificadores L2 A.

Aclaración: el modelo BETO_4 cuenta con un *head* de clasificación de dos capas.

4.2.3. Resultados para clasificador L2 B

En la figura 4.4 se muestran los entrenamientos realizados para el clasificador L2 B y sus resultados:

Modelo	Entrenamiento	Class weights	Stopwords	Nro. de épocas	Optimizador	F1
CNB_1	-	-	-	-	-	0,9593
BETO_1	Grueso	No	No	10	ADAM	0,9781
BETO_1	Fino	No	No	10	ADAM	0,9934
BETO_2	Grueso	Sí	No	22 (early stopping)	ADAM	0,9605
BETO_2	Fino	Sí	No	10	ADAM	0,9844
BETO_3	Grueso	No	Sí	22 (early stopping)	ADAM	0,9819
BETO_3	Fino	No	Sí	8 (early stopping)	ADAM	0,9948

FIGURA 4.4. Resultados obtenidos para los modelos clasificadores L2 B.

4.2.4. Resultados para clasificador L2 C

En la figura 4.5 se presentan los resultados para los entrenamientos del clasificador L2 C:

Modelo	Entrenamiento	Class weights	Stopwords	Nro. de épocas	Optimizador	F1
CNB_1	-	-	-	-	-	0,7116
BETO_1	Grueso	No	No	20	ADAM	0,53
BETO_1	Fino	No	No	18 (early stopping)	ADAM	0,7821
BETO_2	Grueso	No	Sí	20	ADAM	0,5404
BETO_2	Fino	No	Sí	18 (early stopping)	ADAM	0,7992
BETO_3	Grueso	Sí	Sí	19 (early stopping)	ADAM	0,4572
BETO_3	Fino	Sí	Sí	30	ADAM	0,7582
BETO_4	Grueso	No	Sí	20	LAMB	0,5823
BETO_4	Fino	No	Sí	15 (early stopping)	LAMB	0,7924

FIGURA 4.5. Resultados obtenidos para los modelos clasificadores L2 C.

Aclaración: los modelos entrenados con LAMB cuentan con un *head* de clasificación de dos capas.

4.2.5. Resultados para clasificador L2 D

A continuación se muestra la figura 4.6 con los resultados para los entrenamientos del clasificador L2 D.

Modelo	Entrenamiento	Class weights	Stopwords	Nro. de épocas	Optimizador	F1
CNB_1	-	-	-	-	-	0,7129
BETO_1	Grueso	No	No	20	ADAM	0,6433
BETO_1	Fino	No	No	15 (early stopping)	ADAM	0,7767
BETO_2	Grueso	No	Sí	30	LAMB	0,6317
BETO_2	Fino	No	Sí	13 (early stopping)	LAMB	0,7834
BETO_3	Grueso	Sí	Sí	30	LAMB	0,6367
BETO_3	Fino	Sí	Sí	14 (early stopping)	LAMB	0,7577

FIGURA 4.6. Resultados obtenidos para los modelos clasificadores L2 D.

Aclaración: los modelos entrenados con LAMB cuentan con un *head* de clasificación de dos capas.

4.2.6. Resultados para clasificador L2 E

En la figura 4.7 se pueden observar los distintos entrenamientos realizados para el clasificador L2 E y su puntaje obtenido:

Modelo	Entrenamiento	Class weights	Stopwords	Nro. de épocas	Optimizador	F1
CNB_1	-	-	-	-	-	0,6257
BETO_1	Grueso	No	Sí	25	ADAM	0,5723
BETO_1	Fino	No	Sí	7 (early stopping)	LAMB	0,6787
BETO_2	Grueso	No	Sí	50	LAMB	0,6126
BETO_2	Fino	No	Sí	12 (early stopping)	LAMB	0,6885
BETO_3	Grueso	Sí	Sí	18 (early stopping)	LAMB	0,5451
BETO_3	Fino	Sí	Sí	16 (early stopping)	LAMB	0,6864

FIGURA 4.7. Resultados obtenidos para los modelos clasificadores L2 E.

A continuación se hacen algunas aclaraciones:

- Los modelos CNB_1 y BETO_1 fueron entrenados únicamente con el dataset que tenía datos hasta 2022. Algunas categorías no tenían datos hasta esa fecha, por lo que los modelos fueron entrenados con datos faltantes para esas categorías.
- Los modelos BETO_2 y BETO_3 fueron entrenados con el dataset que tenía datos hasta 2022 pero incluyendo datos de 2023 para las categorías que no tenían datos.

4.2.7. Resultados para clasificador L2 F

En la figura 4.8 se muestran los entrenamientos realizados para el clasificador L2 F y sus resultados:

Modelo	Entrenamiento	Class weights	Stopwords	Nro. de épocas	Optimizador	F1
Baseline	-	-	-	-	-	0,8155
BETO_1	Grueso	No	No	20	ADAM	0,7573
BETO_2	Grueso	No	Sí	22 (early stopping)	ADAM	0,7648
BETO_2	Fino	No	Sí	16 (early stopping)	ADAM	0,8891
BETO_3	Grueso	Si	Sí	19 (early stopping)	ADAM	0,6267
BETO_3	Fino	Si	Sí	30 (early stopping)	ADAM	0,8487

FIGURA 4.8. Resultados obtenidos para los modelos clasificadores L2 F.

4.2.8. Resultados para clasificador L2 G

En la figura 4.9 se presentan los resultados para los entrenamientos del clasificador L2 G:

Modelo	Entrenamiento	Class weights	Stopwords	Nro. de épocas	Optimizador	F1
CNB_1	-	-	-	-	-	0,7131
BETO_1	Grueso	No	No	20	ADAM	0,5601
BETO_1	Fino	No	No	19 (early stopping)	ADAM	0,7394
BETO_2	Grueso	No	Sí	25	ADAM	0,5482
BETO_2	Fino	No	Sí	7 (early stopping)	LAMB	0,7503
BETO_3	Grueso	No	Sí	33 (early stopping)	LAMB	0,5782
BETO_3	Fino	No	Sí	5 (early stopping)	LAMB	0,7573
BETO_4	Grueso	Sí	Sí	23 (early stopping)	LAMB	0,4861
BETO_4	Fino	Sí	Sí	22 (early stopping)	LAMB	0,7001

FIGURA 4.9. Resultados obtenidos para los modelos clasificadores L2 G.

Aclaración: los modelos BETO_3 y BETO_4 fueron entrenados con un *head* de clasificación de dos capas.

4.2.9. Resultados para clasificador L2 H

A continuación se muestra la figura 4.10 con los resultados para los entrenamientos del clasificador L2 H.

Modelo	Entrenamiento	Class weights	Stopwords	Nro. de épocas	Optimizador	F1
CNB_1	-	-	-	-	-	0,6890
BETO_1	Grueso	No	No	20	ADAM	0,6753
BETO_1	Fino	No	No	14 (early stopping)	ADAM	0,7377
BETO_2	Grueso	No	Sí	22 (early stopping)	ADAM	0,6840
BETO_2	Fino	No	Sí	14 (early stopping)	ADAM	0,7702
BETO_2	Fino	No	Sí	5 (early stopping)	LAMB	0,7674
BETO_3	Grueso	Sí	Sí	24 (early stopping)	ADAM	0,5794
BETO_3	Fino	Sí	Sí	20 (early stopping)	ADAM	0,7006
BETO_4	Grueso	No	Sí	25 (early stopping)	LAMB	0,6810
BETO_4	Fino	No	Sí	7 (early stopping)	LAMB	0,7615

FIGURA 4.10. Resultados obtenidos para los modelos clasificadores L2 H.

Aclaración: el modelo BETO_4 fue entrenado con un *head* de clasificación de dos capas.

4.2.10. Resultados para clasificador L2 I

En la figura 4.11 se presentan los resultados para los entrenamientos del clasificador L2 I.

Modelo	Entrenamiento	Class weights	Stopwords	Nro. de épocas	Optimizador	F1
CNB_1	-	-	-	-	-	0,8567
CNB_2	-	-	-	-	-	0,8619
BETO_1	Grueso	No	Sí	25	ADAM	0,7957
BETO_1	Fino	No	Sí	16 (early stopping)	ADAM	0,8887
BETO_2	Grueso	No	Sí	30	LAMB	0,7604
BETO_2	Fino	No	Sí	16 (early stopping)	LAMB	0,8840
BETO_3	Grueso	No	Sí	30	LAMB	0,8268
BETO_3	Fino	No	Sí	11 (early stopping)	LAMB	0,8967
BETO_4	Grueso	Sí	Sí	30	LAMB	0,6386

FIGURA 4.11. Resultados obtenidos para los modelos clasificadores L2 I.

Se hacen algunas aclaraciones:

- Tanto en el dataset que tenía datos hasta 2022 como el que incluía datos de 2023 carecían de datos para algunas categorías.
- Todos los modelos entrenados con LAMB contaban con un *head* de clasificación de dos capas.
- El modelo BETO_1 fue entrenado exclusivamente con el dataset original.
- El modelo BETO_2 fue entrenado con el dataset original y se incluyeron datos de 2023 para las categorías sin datos.
- Los modelos BETO_3 y BETO_4 fueron entrenados con datos hasta 2023 inclusive para todas las categorías.

4.2.11. Resultados para clasificador L2 J

A continuación se muestra la figura 4.12 con los resultados para los entrenamientos del clasificador L2 J.

Modelo	Entrenamiento	Class weights	Stopwords	Nro. de épocas	Optimizador	F1
CNB_1	-	-	-	-	-	0,8617
BETO_1	Grueso	No	No	25	ADAM	0,7192
BETO_1	Fino	No	No	14 (early stopping)	ADAM	0,8555
BETO_2	Grueso	No	Sí	21 (early stopping)	ADAM	0,7375
BETO_2	Fino	No	Sí	19 (early stopping)	ADAM	0,8620
BETO_3	Grueso	Sí	Sí	25	ADAM	0,7228
BETO_3	Fino	Sí	Sí	19 (early stopping)	ADAM	0,8580

FIGURA 4.12. Resultados obtenidos para los modelos clasificadores L2 J.

4.2.12. Resultados para clasificador L2 K

En la figura 4.13 se presentan los resultados para los entrenamientos del clasificador L2 K.

Modelo	Entrenamiento	Class weights	Stopwords	Nro. de épocas	Optimizador	F1
CNB_1	-	-	-	-	-	0,9128
BETO_1	Grueso	No	No	25	ADAM	0,8956
BETO_1	Fino	No	No	9 (early stopping)	ADAM	0,9100
BETO_2	Grueso	No	Sí	25	ADAM	0,8855
BETO_2	Fino	No	Sí	14 (early stopping)	ADAM	0,9210
BETO_3	Grueso	Sí	Sí	25	ADAM	0,7967
BETO_3	Fino	Sí	Sí	30	ADAM	0,9023

FIGURA 4.13. Resultados obtenidos para los modelos clasificadores L2 K.

4.2.13. Resultados para clasificador L2 L

En la figura 4.14 se presentan los resultados para los entrenamientos del clasificador L2 L.

Modelo	Entrenamiento	Class weights	Stopwords	Nro. de épocas	Optimizador	F1
CNB_1	-	-	-	-	-	0,6854
CNB_2	-	-	-	-	-	0,6814
BETO_1	Grueso	No	No	25	ADAM	0,6300
BETO_2	Grueso	No	Sí	19 (early stopping)	ADAM	0,6852
BETO_2	Fino	No	Sí	17 (early stopping)	ADAM	0,7684
BETO_2	Fino	No	Sí	5 (early stopping)	LAMB	0,7643
BETO_3	Grueso	No	Sí	21 (early stopping)	ADAM	0,7040
BETO_3	Fino	No	Sí	16 (early stopping)	ADAM	0,7664
BETO_4	Grueso	No	Sí	14 (early stopping)	ADAM	0,7143
BETO_4	Fino	No	Sí	25 (early stopping)	ADAM	0,7361
BETO_5	Grueso	No	Sí	17 (early stopping)	LAMB	0,7248
BETO_5	Fino	No	Sí	5 (early stopping)	LAMB	0,7806
BETO_6	Grueso	Sí	Sí	16 (early stopping)	LAMB	0,7088
BETO_6	Fino	Sí	Sí	6 (early stopping)	LAMB	0,7805

FIGURA 4.14. Resultados obtenidos para los modelos clasificadores L2 L.

Se hacen algunas aclaraciones de los modelos:

- Los modelos CNB_1, BETO_1 y BETO_2 fueron entrenados con el dataset original que tenía datos hasta 2022 inclusive.
- Los modelos CNB_2, BETO_3, BETO_5 y BETO_6 fueron entrenados con el dataset que también incluía datos de 2023.
- Para el entrenamiento del modelo BETO_4 se utilizó el dataset original que tenía datos hasta 2022 y además se incluyeron datos de 2023 para la clase minoritaria.
- Para los modelos BETO_5 y BETO_6 se utilizó un *head* de clasificación de dos capas.

4.2.14. Resultados para clasificador L2 M

Como se mencionó en la sección 3.3.3, el entrenamiento del modelo L2 M fue particular porque se contaban con 267 muestras incluyendo los casos de 2023. Se decidió obtener una métrica con validación cruzada para tener una medida de desempeño y luego entrenar un modelo con todos los datos. En este caso el F1 obtenido fue 0,5568.

4.2.15. Resultados para clasificador L2 N

Además, en la sección 3.3.3 también se hace mención al modelo L2 N, que fue un caso extremo donde sólo había 8 muestras incluyendo los datos de 2023. Aquí se utilizó la técnica de *leave one out* y el F1 obtenido fue 0,7500.

4.3. Simulación en ambiente de desarrollo

Capítulo 5

Conclusiones

5.1. Conclusiones generales

La idea de esta sección es resaltar cuáles son los principales aportes del trabajo realizado y cómo se podría continuar. Debe ser especialmente breve y concisa. Es buena idea usar un listado para enumerar los logros obtenidos.

Algunas preguntas que pueden servir para completar este capítulo:

- ¿Cuál es el grado de cumplimiento de los requerimientos?
- ¿Cuán fielmente se pudo seguir la planificación original (cronograma incluido)?
- ¿Se manifestó algunos de los riesgos identificados en la planificación? ¿Fue efectivo el plan de mitigación? ¿Se debió aplicar alguna otra acción no contemplada previamente?
- Si se debieron hacer modificaciones a lo planificado ¿Cuáles fueron las causas y los efectos?
- ¿Qué técnicas resultaron útiles para el desarrollo del proyecto y cuáles no tanto?

5.2. Próximos pasos

Acá se indica cómo se podría continuar el trabajo más adelante.

Bibliografía

- [1] Tiendanube. *Atención al cliente: qué es y claves para mejorar tu servicio*. <https://www.tiendanube.com/blog/que-es-servicio-atencion-cliente/>. Mar. de 2023. (Visitado 01-07-2023).
- [2] HubSpot. *¿En qué se diferencian el servicio al cliente y la atención al cliente?* <https://blog.hubspot.es/service/diferencia-servicio-cliente-y-atencion-cliente>. Ene. de 2023. (Visitado 01-07-2023).
- [3] HubSpot. *¿Qué hace un gerente de servicio al cliente?* <https://blog.hubspot.es/service/gerente-servicio-cliente>. Ene. de 2023. (Visitado 01-07-2023).
- [4] Zendesk. *Manual para un departamento de atención al cliente exitoso*. <https://www.zendesk.com.mx/blog/manual-de-funciones-de-servicio-al-cliente/>. Abr. de 2023. (Visitado 01-07-2023).
- [5] HubSpot. *Qué es el proceso de atención al cliente y cuáles son sus fases clave*. <https://blog.hubspot.es/service/proceso-atencion-cliente>. Abr. de 2023. (Visitado 01-07-2023).
- [6] Zendesk. *Entienda lo que son las fases del proceso de atención al cliente y en que puedes ayudar tener ese proceso interno en una empresa*. <https://www.zendesk.com.mx/blog/fases-del-proceso-de-atencion-al-cliente/>. Jun. de 2020. (Visitado 01-07-2023).
- [7] Smart Tribune. *The Importance of Customer Experience*. <https://blog.smart-tribune.com/en/importance-of-customer-experience>. Sep. de 2021. (Visitado 01-07-2023).
- [8] Zendesk. *CX Trends 2023*. <https://cxtrends.zendesk.com/mx>. Feb. de 2023. (Visitado 01-07-2023).
- [9] HubSpot. *Qué es la atención al cliente, elementos clave e importancia*. <https://blog.hubspot.es/service/que-es-atencion-al-cliente>. Abr. de 2023. (Visitado 01-07-2023).
- [10] Zendesk. *5 examples of bad customer service (and how to be great instead)*. <https://www.zendesk.com/blog/what-is-bad-customer-service/>. Mayo de 2023. (Visitado 01-07-2023).
- [11] SuperOffice. *32 CUSTOMER EXPERIENCE STATISTICS YOU NEED TO KNOW FOR 2023*. <https://www.superoffice.com/blog/customer-experience-statistics/>. Feb. de 2023. (Visitado 01-07-2023).
- [12] HubSpot. *8 ejemplos de mal servicio al cliente (y cómo evitarlos)*. <https://blog.hubspot.es/service/mal-servicio-cliente>. Ene. de 2023. (Visitado 02-07-2023).
- [13] Zendesk. *8 problemas comunes en servicio al cliente y cómo resolverlos*. <https://www.zendesk.com.mx/blog/problemas-comunes-con-clientes/>. Feb. de 2021. (Visitado 02-07-2023).
- [14] Piperlab. *Cómo hemos llegado hasta ChatGPT y el resto de LLMs*. <https://piperlab.es/2023/03/29/como-hemos-llegado-hasta-chatgpt-y-el-resto-de-llms/>. Mar. de 2023. (Visitado 02-07-2023).

- [15] Yoshua Bengio, Réjean Ducharme y Pascal Vincent. «A Neural Probabilistic Language Model». En: (2003).
- [16] Tomas Mikolov y col. «Efficient Estimation of Word Representations in Vector Space». En: (2013).
- [17] Jeffrey Pennington, Richard Socher y Christopher D. Manning. «GloVe: Global Vectors for Word Representation». En: (2014).
- [18] Ashish Vaswani y col. «Attention Is All You Need». En: (2017).
- [19] TechTarget. *5 examples of effective NLP in customer service*. <https://www.techtarget.com/searchenterpriseai/feature/5-examples-of-effective-NLP-in-customer-service>. Feb. de 2021. (Visitado 03-07-2023).
- [20] Salesforce. *¿Qué es la CRM?* <https://www.salesforce.com/es/learning-center/crm/what-is-crm/?d=cta-resources-1-ungated-crm>. (Visitado 29-07-2023).
- [21] Google Cloud. *Almacén de datos en la nube para impulsar la innovación basada en datos*. <https://cloud.google.com/bigquery?hl=es>. (Visitado 29-07-2023).
- [22] Google Cloud. *Cloud Composer*. https://cloud.google.com/composer?hl=es_419. (Visitado 29-07-2023).
- [23] Tacos de datos. *Introducción al análisis de texto*. <https://old.tacosdedatos.com/analisis-texto>. Ago. de 2020. (Visitado 07-07-2023).
- [24] freeCodeCamp. *How to process textual data using TF-IDF in Python*. <https://www.freecodecamp.org/news/how-to-process-textual-data-using-tf-idf-in-python-cd2bbc0a94a3/>. Jun. de 2018. (Visitado 08-07-2023).
- [25] freeCodeCamp. *Cómo funcionan los clasificadores Naive Bayes: con ejemplos de código de Python*. <https://www.freecodecamp.org/espanol/news/como-funcionan-los-clasificadores-naive-bayes-con-ejemplos-de-codigo-de-python/>. Abr. de 2021. (Visitado 08-07-2023).
- [26] Jason Rennie y col. «Tackling the Poor Assumptions of Naive Bayes Text Classifiers». En: (2003).
- [27] Scikit-Learn. *Complement Naive Bayes*. https://scikit-learn.org/stable/modules/naive_bayes.html. (Visitado 08-07-2023).
- [28] Analytics Vidhya. *Why and how to use BERT for NLP Text Classification?* <https://www.analyticsvidhya.com/blog/2021/06/why-and-how-to-use-bert-for-nlp-text-classification/>. Jun. de 2021. (Visitado 11-07-2023).
- [29] TechTarget. *BERT language model*. <https://www.techtarget.com/searchenterpriseai/definition/BERT-language-model>. (Visitado 10-07-2023).
- [30] Amazon Web Services. *¿Qué es Python?* <https://aws.amazon.com/es/what-is/python/>. (Visitado 29-07-2023).
- [31] Hostinger. *Qué es GitHub y cómo empezar a usarlo*. <https://www.hostinger.com.ar/tutoriales/que-es-github>. Ene. de 2023. (Visitado 29-07-2023).
- [32] GitHub. *Entender las GitHub Actions*. <https://docs.github.com/es/actions/learn-github-actions/understanding-github-actions>. (Visitado 29-07-2023).
- [33] Google Cloud. *Vertex AI Workbench*. <https://cloud.google.com/vertex-ai-workbench?hl=es-419>. (Visitado 29-07-2023).
- [34] Amazon Web Services. *¿Qué es Docker?* <https://aws.amazon.com/es/docker/>. (Visitado 29-07-2023).

- [35] Google Cloud. *Artifact Registry*.
<https://cloud.google.com/artifact-registry?hl=es-419>. (Visitado 29-07-2023).
- [36] Omar Espejel. *BETO (BERT) 01 : Importación y tokenizing*.
<https://espejel.substack.com/p/beto-bert-01-importacion-y-tokenizing>.
(Visitado 15-07-2023).
- [37] Openlayer. *Baseline models demystified: a practical guide*.
<https://www.openlayer.com/blog/post/baseline-models>. Jun. de 2023.
(Visitado 18-07-2023).
- [38] HuggingFace. *BETO: Spanish BERT*.
<https://huggingface.co/dccuchile/bert-base-spanish-wwm-uncased>.
(Visitado 30-07-2023).
- [39] EuropeanValley. *Aprendizaje por transferencia: NLP*. <https://www.europeanvalley.es/noticias/aprendizaje-por-transferencia-nlp/>. Ago. de 2020.
(Visitado 19-07-2023).
- [40] Sophie Henning y col. «A Survey of Methods for Addressing Class Imbalance in Deep-Learning Based Natural Language Processing». En: (2023).
- [41] Towards DataScience. *Dealing with Imbalanced Data in TensorFlow: Class Weights*. <https://towardsdatascience.com/dealing-with-imbalanced-data-in-tensorflow-class-weights-60f876911f99>. Jun. de 2021. (Visitado 22-07-2023).
- [42] Jacob Devlin y col. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». En: (2019).
- [43] Yang You y col. «Large Batch Optimization for Deep Learning: Training BERT in 76 minutes». En: (2020).
- [44] Towards DataScience. *An intuitive understanding of the LAMB optimizer*.
<https://towardsdatascience.com/an-intuitive-understanding-of-the-lamb-optimizer-46f8c0ae4866>. Mayo de 2019. (Visitado 22-07-2023).
- [45] Kubernetes. *Pod Lifecycle*.
<https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/>.
(Visitado 26-07-2023).
- [46] Amir Masoud Sefidian. *Understanding Micro, Macro, and Weighted Averages for Scikit-Learn metrics in multi-class classification with example*.
<http://iamirmasoud.com/2022/06/19/understanding-micro-macro-and-weighted-averages-for-scikit-learn-metrics-in-multi-class-classification-with-example/>. (Visitado 29-07-2023).