



UNIVERSITAT OBERTA DE CATALUNYA (UOC)

MÁSTER UNIVERSITARIO EN CIENCIA DE DATOS (*Data Science*)

TRABAJO FINAL DE MÁSTER

ÁREA: PLN

Modelización de temas de llamadas en tiempo real Borrador

Autor: Manuel E. Gómez Montero

Tutora UOC: Ana Valdivia Garcia

Tutor TE: Antonio Fernández Gallardo

Profesor: Jordi Casas

Madrid, 14 de diciembre de 2019

Resumen

Un call-center es el área de una empresa el cuál se encarga de recibir y transmitir llamadas desde o hacia clientes, socios comerciales u otras compañías externas. Debido a la gran cantidad de información que se transfiere en estos centros, resulta una tarea esencial optimizar el tiempo de respuesta para así reaccionar en tiempo real a las peticiones de los clientes y mejorar la percepción que estos tienen sobre la compañía.

Una manera de mejorar el rendimiento es detectar el tema de las llamadas mediante técnicas de *machine learning* dando la posibilidad a la empresa de reaccionar en tiempo real, en función de la temática que se este tratando en cada momento.

El sistema que se presenta en el documento nos permite, a partir de la transcripción de las llamadas al *call-center* de Telefónica España, descubrir en tiempo real la temática de las mismas. Esta modelización de *topics* se ha realizado utilizando métodos de Procesamiento de Lenguaje Natural y aprendizaje profundo. El sistema realiza la clasificación de las nuevas llamadas en tiempo real, permitiendo a los usuarios visualizar la evolución en la temática de las mismas y generar alertas en base a anomalías.

TODO Es un borrador volver al resumen una vez acabado el proyecto.

Palabras clave: “natural language processing”, “sentiment analysis”, “real time”, “call center”, “topic modeling”, “deep learning”

Índice general

Abstract	I
Índice	III
Listado de Figuras	VII
I Introducción: objetivos, estado del arte y arquitectura global	1
1. Introducción	3
1.1. Descripción general de la propuesta	3
1.2. Motivación	4
1.3. Objetivos	4
1.4. Tareas y planificación	5
1.5. Estructura del documento	7
2. Estado del Arte	9
2.1. Procesamiento de lenguaje natural	9
2.1.1. Historia	9
2.1.2. Aplicaciones	10
2.1.3. Modelización de temas	12
2.2. Deep Learning y aplicación al PLN	12
2.2.1. Aprendizaje supervisado	13
2.2.2. Deep Learning	13
2.2.3. Representación de palabras en PLN	14
2.2.4. Arquitecturas especializadas	15
2.3. <i>BigData</i> y <i>Fast Data</i>	23
2.3.1. Evolución: del <i>Big Data</i> al <i>Fast Data</i>	23
2.3.2. Arquitecturas <i>RealTime</i>	25

2.4. Trabajos anteriores	28
3. Arquitectura y tecnologías	31
3.1. Modelado	31
3.2. Explotación	32
3.3. Mantenimiento e Integración Continua	33
3.4. Tecnologías	34
3.4.1. Modelado	34
3.4.2. Explotación	35
3.4.3. Capa Servicio	36
3.4.4. Integración y Despliegue Continuo	36
II Modelado: datos, modelos y optimizaciones	39
4. Conjunto de datos	41
4.1. Análisis del conjunto de datos	42
4.1.1. Imports	42
4.1.2. Carga de datos	43
4.1.3. Análisis preliminar	48
4.2. Preprocesado y Representación de transcripciones	58
4.2.1. Word2vec	58
4.2.2. Doc2vec	59
4.3. Evolución del conjunto de datos	59
4.3.1. Las llamadas	60
4.3.2. Las etiquetas	64
5. Aprendizaje No Supervisado	67
6. Aprendizaje supervisado	69
6.1. Arquitecturas	69
6.1.1. Red neuronal convolucional	70
6.1.2. Red neuronal recurrente	72
6.1.3. Red neuronal “híbrida” convolucional/recurrente	73
6.1.4. MLP	75
6.1.5. Combinación de modelos	77
6.2. Evaluación y optimización	78
6.2.1. Modelos <i>CNN</i> binarios	80

6.2.2.	Modelos <i>RNN</i> Binarios	80
6.2.3.	Modelos totalmente conectados	82
6.2.4.	Modelos multiclasé	83
6.3.	Modelo Mínimo Viable	84
III	Explotación: procesamiento, visualización y alarmados	87
7.	Explotación	89
7.1.	Requisitos del sistema productivo	90
7.2.	Arquitectura del sistema	90
7.3.	Microservicios	92
7.3.1.	Tecnología	92
7.3.2.	Microservicios	93
7.3.3.	<i>Tokenizer</i>	94
7.3.4.	<i>Sequencer</i>	96
7.3.5.	<i>tf-BajaFactura</i>	97
7.3.6.	Predictor	97
7.4.	Monitorización de Microservicios	98
7.4.1.	Servicios <i>Kafka Streams</i>	99
7.4.2.	Servicios <i>Tensorflow Serving</i>	99
7.5.	Inyector: simulador de tiempo real	100
8.	Capa de servicio	101
8.1.	Carga y modelo	101
8.2.	Visualizaciones	101
8.2.1.	Monitorización	101
8.2.2.	Sistema	101
8.3.	Alarmado	101
9.	Despliegue en contenedores	103
9.1.	Servicios <i>Kafka Streams</i>	105
9.2.	Servicios <i>Tensorflow Serving</i>	109
9.3.	Monitorización	113
9.3.1.	<i>Logstash</i>	113
9.3.2.	<i>Metricbeat</i>	116
9.4.	S2I	119
9.4.1.	<i>Kafka Streams</i>	120

9.4.2. <i>Tensorflow Serving</i>	121
IV Conclusiones: mantenimiento y futuros trabajos	123
10.DevOps	125
10.1. Introducción DevOps	125
10.2. Degradación del modelo	125
10.3. Integración y despliegue continuos	126
Bibliografía	126

Índice de figuras

1.1.	Diagrama de Gantt	5
1.2.	Fases del modelo CRISP-DM	7
2.1.	Ejemplo de arquitectura MLP. Fuente [29]	14
2.2.	Arquitecturas CBOW y Skip-gram. Fuente [23]	15
2.3.	Ejemplo de una convolución de dos dimensiones. Fuente [2]	16
2.4.	Ejemplo de aplicación de <i>zero padding</i> para mantener la dimensionalidad. Fuente [2]	17
2.5.	Ejemplo de aplicación de convolución por pasos para reducir la dimensionalidad. Fuente [2]	17
2.6.	Ejemplo de aplicación de <i>max-pooling</i> . Fuente [2]	18
2.7.	Comparación capa tradicional totalmente conectada con capa convolucional. Fuente [15]	19
2.8.	Ejemplo RNN. Fuente [28]	20
2.9.	Ejemplo RNN “desenrollada”. Fuente [28]	21
2.10.	Arquitectura celdas LSTM y GRU. Fuente [27]	22
2.11.	Evolución del <i>Big Data</i> . Fuente [10]	24
2.12.	Teorema CAP. Fuente [26]	26
2.13.	Arquitectura Lambda definida por Nathan Marz. Fuente [10]	27
2.14.	Arquitectura Kappa definida por Jay Kreps. Fuente [10]	27
3.1.	Arquitectura Kappa	33
4.1.	Distribución de llamadas por monitorizaciones	49
4.2.	Nube de palabras a partir de transcripciones monitorizadas	50
4.3.	Nube de palabras para la categoría consulta de las monitorizaciones	51
4.4.	Nube de palabras para la categoría contratar de las monitorizaciones	51
4.5.	Nube de palabras para la categoría informacion de las monitorizaciones	52
4.6.	Nube de palabras para la categoría queja de las monitorizaciones	52

4.7.	Nube de palabras para la categoría trámite de las monitorizaciones	52
4.8.	Distribución de llamadas por <i>IVR</i>	54
4.9.	Nube de palabras a partir de transcripciones con <i>IVR</i>	54
4.10.	Nube de palabras de la categoría avería de <i>IVR</i>	55
4.11.	Nube de palabras de la categoría baja de <i>IVR</i>	55
4.12.	Nube de palabras de la categoría comercial de <i>IVR</i>	56
4.13.	Nube de palabras de la categoría factura de <i>IVR</i>	56
4.14.	Nube de palabras de la categoría “no reconocido” de <i>IVR</i>	56
4.15.	Nube de palabras de la categoría reclamación de <i>IVR</i>	57
4.16.	Nube de palabras de la categoría resto de <i>IVR</i>	57
4.17.	Primeros datos: <i>wordcloud</i> inicial	61
4.18.	Primeros datos: <i>wordcloud</i> filtrado	63
6.1.	Arquitectura CNN	70
6.2.	Arquitectura RNN	72
6.3.	Arquitectura “Híbrida” CNN/RNN	74
6.4.	Arquitectura con capas totalmente conectadas	76
6.5.	Arquitectura con capas totalmente conectadas	77
6.6.	Optimización de modelos binarios con redes convolucionales	81
6.7.	Optimización de modelos binarios con redes recurrentes	82
6.8.	Optimización de modelos binarios con <i>MLP</i>	82
6.9.	Optimización de modelos multiclase	83
6.10.	Matriz de confusión datos balanceados	84
6.11.	Matriz de confusión datos no balanceados	85
7.1.	Capa de <i>streaming</i>	89
7.2.	Arquitectura microservicios	94
9.1.	configuración de despliegue y recursos usados en OCP	104

Parte I

Introducción: objetivos, estado del arte y
arquitectura global

Capítulo 1

Introducción

Este primer capítulo del trabajo tiene como objetivo presentar, a grandes rasgos, la propuesta (sección 1.1), los objetivos que pretendemos lograr (sección 1.3), la motivación que nos ha llevado a abordar este proyecto (sección 1.2) y un repaso a las tareas que serán necesarias para la ejecución del mismo (sección 1.4).

Por último, dedicaremos una sección que describa brevemente los diferentes apartados de los que constará el documento y el objetivo de cada uno (sección 1.5).

1.1. Descripción general de la propuesta

En los últimos años, la explosión ingente en la generación de datos y el avance en las capacidades tecnológicas que nos permiten recolectar, almacenar y procesar los datos generados; han provocado que empecemos a abordar el estudio de otro tipo de datos no estructurados que antes no se podían analizar como imágenes, textos, audios, etc. Como resultado, diferentes áreas del conocimiento (Procesamiento del Lenguaje Natural, Análisis de Imágenes) han experimentado un creciente interés tanto en la comunidad científica como en el mundo de los negocios.

Dentro de los datos no estructurados, una de las fuentes de información con mayor potencial en todas las grandes empresas que prestan servicio al público general, son las llamadas que los clientes realizan a su *call-center*, ya que nos permiten obtener una idea de la percepción que los clientes tienen de nuestra empresa y de sus preocupaciones en cada momento.

La propuesta que pretendemos abordar en este trabajo consiste en extraer la temática de estas llamadas en el momento en el que son capturadas. Aunque actualmente esta captura se hace periódicamente pretendemos construir una solución que nos permita el tratamiento de las mismas en tiempo real o streaming, y de esta manera mejorar el rendimiento de estos centros.

Esta extracción en tiempo real nos permitirá conocer cómo evolucionan los temas que tratan nuestros clientes cuando llaman a nuestro *call-center* para así poder reaccionar inmediatamente

ante una preocupación concreta.

1.2. Motivación

La motivación que nos ha llevado a acometer un proyecto de esta naturaleza viene originada por diferentes factores que están ligados tanto al negocio como a las capacidades técnicas disponibles en la empresa.

Por un lado, la capacidad de obtener la temática de las llamadas en tiempo real se presenta como una oportunidad de mejorar la operatividad de un *call-center* y por ende la satisfacción de los clientes, permitiéndonos entenderlos mejor y así reaccionar de una manera ágil a sus necesidades reales.

Desde el punto de vista técnico, también es el momento ideal para emprender este proyecto debido tanto a la disponibilidad periódica de transcripciones de las llamadas, que nos permiten ahorrarnos el paso de realizar un *Speech 2 Text* para obtener nuestro conjunto de datos; como al aumento de capacidades técnicas en la empresa que nos permitirán tanto entrenar nuestros modelos, como poder tratar y explotar los datos en tiempo real.

1.3. Objetivos

En este apartado definiremos los objetivos que se pretenden conseguir con este proyecto. Estos objetivos deben ser *SMART*, es decir:

- *Specific*: Deben plantearse de una forma detallada y concreta.
- *Measurable*: Deben poder medirse con facilidad.
- *Achievable*: Deben ser objetivos realistas.
- *Relevant*: Tienen que ser relevantes para la empresa y ofrecernos un beneficio claro.
- *Timely*: Estos objetivos tienen que tener un tiempo establecido.

El objetivo general es optimizar el proceso de atención de llamadas en el call-center mediante técnicas de Procesamiento del Lenguaje Natural y Aprendizaje Profundo. Concretamente, los objetivos específicos que se pretenden conseguir con este proyecto son:

- **Construir un modelo que nos permita extraer la temática de las llamadas** a partir de su transcripción a texto. Este objetivo debemos alcanzarlo en la fase de modelado y podremos medir su éxito atendiendo al porcentaje de llamadas que podamos clasificar correctamente en un proceso de test. Se trata del objetivo principal del proyecto.

- Desarrollar un mecanismo que nos permita **extraer esta temática para nuevas llamadas en tiempo real**. De este modo tendremos un sistema vigente cuando la frecuencia en la recepción de las llamadas aumente. Este objetivo se deberá alcanzar en la fase de productivización.
- Disponer de una **visualización en tiempo quasi real** para que pueda visualizarse la evolución de las temáticas a lo largo del tiempo. Este objetivo se deberá alcanzar en la fase de productivización.
- Proporcionar un **sistema de alertado** que nos permita detectar anomalías en el número de llamadas que se reciben de un determinado tema. Este objetivo se deberá alcanzar en la fase de productivización.

En las conclusiones de este proyecto se evaluará el éxito o fracaso del mismo en función del grado de cumplimiento de estos objetivos.

1.4. Tareas y planificación

El proyecto se llevará a cabo desde el 16 de Septiembre hasta el 20 de Febrero. Para poder abordar la ejecución del mismo se han extraído las siguientes tareas principales:

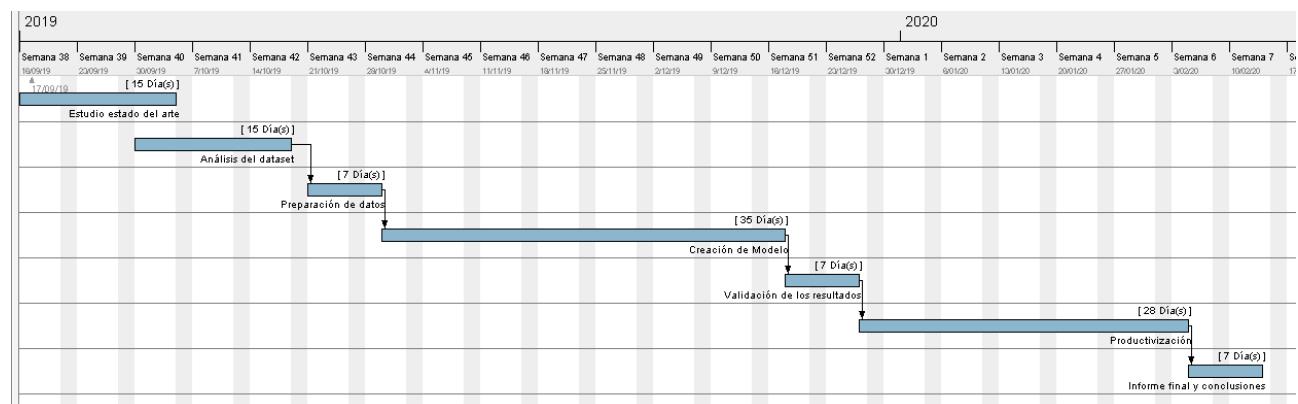


Figura 1.1: Diagrama de Gantt

- **Estudio estado del arte:** En esta fase se realizará una prospección para conocer el estado del arte en todos los puntos relacionados con el proyecto: Procesamiento del Lenguaje Natural, tecnologías de tratamiento de datos en tiempo real y *Big Data*.
- **Análisis del dataset:** El propósito de esta tarea es entender el *dataset* y estudiar las posibilidades del mismo.

- **Preparación del dataset:** Una vez realizado el estudio del *dataset* es necesario realizar labores de limpieza y transformación de los datos de modo que estos datos sean válidos para nuestro objetivo.
- **Creación del modelo:** En esta fase se procederá a la creación de un modelo capaz de obtener los temas de los que habla una determinada llamada. Este modelo será el *core* de nuestro proyecto.
- **Validación de los resultados:** Una vez entrenado el modelo será necesario validar los resultados obtenidos para poder evaluar la bondad de nuestro modelo.
- **Productivización:** El trabajo no acaba con la creación de un buen modelo que nos permita extraer los temas de nuestras llamadas. Este modelo tendrá que ser puesto en producción y permitir al usuario final extraer los temas de las llamadas en tiempo real y darle la opción de crear alarmas basadas en la variación del número de eventos (llamadas) de un determinado tema.
- **Informe final y conclusiones:** Por último, una vez llevado a producción nuestro modelo, se realizará un informe final donde, entre otros puntos, se evaluarán los resultados obtenidos y se extraerán conclusiones y pasos futuros.

Estas fases están basadas en el estándar ***CRISP-DM*** ([7]), añadiendo una última tarea para nuestro informe final, CRISP-DM nos proporciona una descripción del ciclo de vida de los proyectos de minería de datos de un modo bastante similar al que se aplica en los modelos de ciclo de vida de desarrollo *software*.

En la Figura 1.2 se observa el diseño de este modelo y cómo representa el ciclo de vida de un proyecto de minería de datos. En la imagen podemos ver en primer lugar un círculo exterior que refleja la naturaleza cíclica de los proyectos de minería de datos, además vemos cómo la secuencia de tareas no es rígida, pudiendo saltar hacia adelante o atrás entre tareas. En la gráfica se representan mediante flechas las dependencias más importantes y usuales entre tareas.

En nuestro desarrollo usaremos este modelo, aunque en el diagrama de la Figura 1.1 aparezca una secuencia de tareas más rígida, será usual, por ejemplo, el salto recíproco entre las fases de preparación de los datos y creación del modelo.

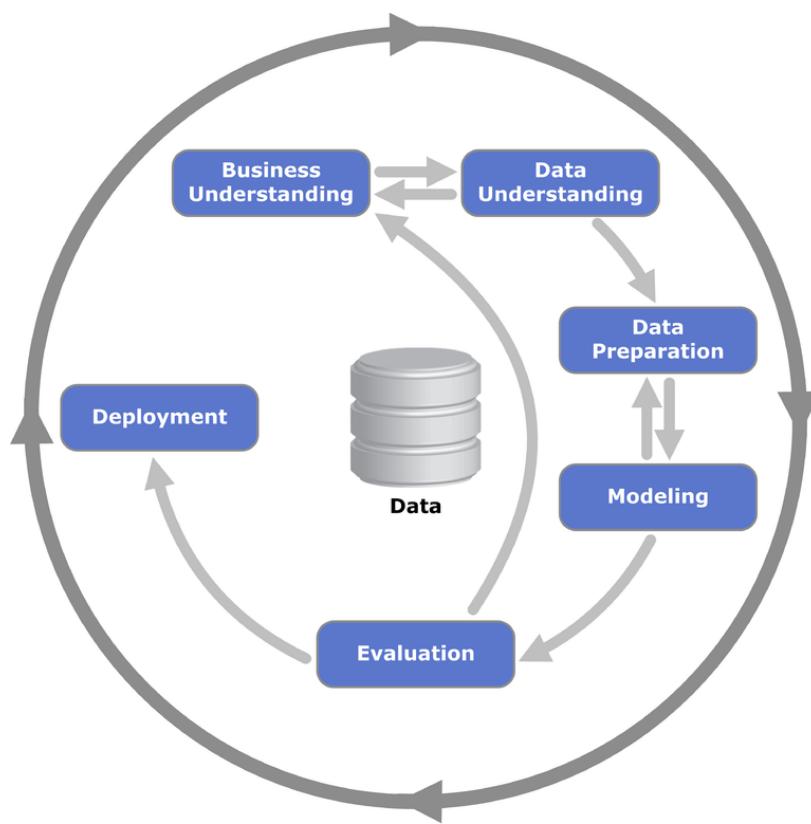


Figura 1.2: Fases del modelo CRISP-DM

1.5. Estructura del documento

TODO Hacer repaso breve de los apartados del documento final.

Capítulo 2

Estado del Arte

El objetivo de este apartado es hacer un recorrido por el estado del arte relacionado con el proyecto, este recorrido lo enfocaremos desde tres puntos de vista diferentes:

- **Procesamiento del Lenguaje Natural:** En la sección 2.1 nos centraremos en el procesamiento del lenguaje natural y su evolución a lo largo del tiempo.
- **Deep Learning y aplicación al Procesamiento del Lenguaje Natural:** En la sección 2.2 pondremos foco en el *Deep Learning*, sus ventajas y cómo se están aplicando estos métodos al procesamiento del lenguaje natural.
- **Big Data y Fast Data:** Por último, en la sección 2.3, haremos un repaso a la evolución del *Big Data* y cómo la tendencia actual es realizar el procesamiento en tiempo real mediante *Fast Data*.

Por último, una vez analizados los diferentes puntos de vista, en la sección 2.4 enumeraremos trabajos anteriores relacionados con nuestro proyecto. Estos trabajos nos serán de utilidad para justificar la realización de nuestro proyecto y su viabilidad.

2.1. Procesamiento de lenguaje natural

2.1.1. Historia

Para hablar de los orígenes del Procesamiento del Lenguaje Natural (a partir de ahora se usarán indistintamente las siglas PLN) tal y como lo conocemos, tendríamos que remontarnos a los años 50, concretamente al artículo “*Computing Machinery and Intelligence*” escrito por Alan Turing [34]. En este artículo aparece el PLN dentro del campo de la inteligencia artificial y se presenta por primera vez el conocido “Test de Turing”. Este test convirtió la pregunta abstracta

de “¿Son capaces de pensar las máquinas?” en un juego llamado: “*The Imitation Game*”. El juego propuesto inicialmente, de forma muy resumida, consiste en ver si una persona (interrogador) interrogando a dos personas (un hombre y una mujer), era capaz de descubrir el sexo de cada una; la modificación del mismo sustituye las dos personas de distinto sexo por una persona y una máquina y el interrogador debe ser capaz de descubrir si las preguntas están siendo respondidas por un humano o una máquina. En el caso de que no sepa discernir, la computadora gana la partida. Podemos encontrar más información al respecto en el libro [35].

A partir de los avances de Turing y hasta los años 80 el crecimiento en el campo del PLN se produjo principalmente con la creación de complejos sistemas basados en reglas escritas a mano. Fue en esta década cuando empezamos a vivir la incorporación de algoritmos de *Machine Learning* enfocados al procesamiento del lenguaje natural. Este hecho se vio motivado principalmente por el increíble avance en la capacidad de cómputo, ya predicho por la ley de Moore, y por la aplicación de teorías ya existentes como los trabajos de Chomsky.

Desde el comienzo de la aplicación de modelos de *Machine Learning*, y de nuevo motivados por el crecimiento de la capacidad computacional de los sistemas actuales, se ha pasado de utilizar árboles de decisión, que creaban de manera automática reglas similares a las que se venían creando manualmente, a los modelos de *deep learning* que están en auge en la última década.

2.1.2. Aplicaciones

En el apartado anterior hicimos referencia a “*The Imitation Game*” como inicio de lo que hoy conocemos como procesamiento del lenguaje natural, sin embargo, las aplicaciones en este campo han crecido de forma vertiginosa en estos 70 años, principalmente en las últimas décadas. Hoy en día, si tuviéramos que contestar a la pregunta: “¿son capaces de pensar las máquinas?”, implicaría algo más que superar el test de Turing. Mirando a nuestro alrededor nos encontrariamos con asistentes de voz como Alexa o Siri que, no solo contestan a nuestras preguntas, si no que realizan un trabajo de pasar nuestra voz a texto (*Speech to Text*) y de nuevo el texto resultante a voz (*Text to Speech*). Nos encontraríamos también con sistemas capaces de realizar traducciones simultáneas, otros capaces de autocompletar textos, de identificar preguntas y respuestas, de clasificar textos de acuerdo a temas o autores, incluso de analizar sentimientos positivos o negativos teniendo como entrada un texto u opinión.

Según [13] todos estos problemas tan diversos podríamos clasificarlos según en el punto del análisis que nos centremos:

- **Análisis de palabras:** En este tipo de problemas se pone foco en las palabras, como pueden ser “perro”, “hablar”, “piedra” y necesitamos decir algo sobre ellas. Por ejemplo:

“¿estamos hablando de un ser vivo?”, “¿a qué lenguaje pertenece?”, “¿cuáles son sus sinónimos o antónimos?”. Actualmente este tipo de problemas son menos frecuentes, ya que normalmente no pretendemos analizar palabras aisladas sino que es preferible basarse en un contexto.

- **Análisis de textos:** En este tipo de problemas no trabajamos solo con palabras aisladas, sino que disponemos de una pieza de texto que puede ser una frase, un párrafo o un documento completo y tenemos que decir algo sobre él. Por ejemplo: “¿se trata de spam?”, “¿qué tipo de texto es?”, “¿el tono es positivo o negativo?”, “¿quién es su autor?”. Este tipo de problemas son muy comunes y nos vamos a referir a ellos como **problemas de clasificación de documentos**.
- **Análisis de textos pareados:** En esta clase de análisis disponemos de dos textos (también podrían ser palabras aisladas) y tenemos que decir algo sobre ellos. Por ejemplo, “¿los textos son del mismo autor?”, “¿son pregunta y respuesta?”, “¿son sinónimos?” (para el caso de palabras aisladas).
- **Análisis de palabras en contexto:** En estos casos de uso, a diferencia del primer análisis que trataba únicamente con palabras aisladas, tenemos que clasificar una palabra en particular en función del contexto en el que se encuentra.
- **Análisis de relación entre palabras:** Este último tipo de análisis tiene como objetivo deducir la relación entre dos palabras existentes en un documento.

Dependiendo del problema que queramos abordar usaremos un tipo de características del lenguaje u otro, por ejemplo, es usual que si estamos analizando palabras aisladas nos centremos en las letras de una palabra, sus prefijos o sufijos, su longitud, la información léxica extraída de diccionarios como *WordNet* [11], etc. En cambio, si estamos trabajando con texto, lo normal es que nos fijemos en otros conceptos estadísticos como el histograma de las palabras dentro del texto, ratio de palabras cortas vs largas, número de veces que aparece una palabra en un texto comparado con el resto de textos, etc.

El proyecto que se presenta en este documento está centrado en el análisis de textos, concretamente en extraer los temas de un documento (o llamada). Este tipo de problemas se conoce como modelización de *topics*.

En el siguiente punto de este apartado nos centraremos en algunos modelos y avances en este área que puedan servirnos de apoyo para nuestro proyecto.

2.1.3. Modelización de temas

La modelización de topics hace referencia a un grupo de algoritmos de *Machine Learning* que infieren la estructura latente existente en un grupo de documentos.

Aunque la mayoría de los algoritmos de modelización son no supervisados, al igual que los algoritmos tradicionales de *clustering*, existen también algunas variantes supervisadas que necesitan disponer de documentos etiquetados.

Quizás el algoritmo más conocido para la modelización de topics sea el *Latent Dirichlet Allocation* (normalmente conocido por su acrónimo, LDA). LDA fué presentado en 2003 en el artículo [6]. Este algoritmo no supervisado asume que cada documento es una distribución probabilística de *topics* y cada *topic*, a su vez, es una distribución de palabras del documento. LDA usa una aproximación llamada “*bag of words*”, en la que cada documento es tratado como un vector con el conteo de las palabras que aparecen en el mismo. La principal característica de LDA es que la colección de documentos comparten los mismos topics, pero cada documento contiene esos topics en una proporción diferente.

A partir de LDA surgieron numerosas variantes que repasaremos de forma breve, por ejemplo, en el mismo año de la creación de LDA y también presentado por los mismos autores en [3], surgió una **variante jerárquica** que permitía representar los *topics* jerárquicamente. En 2006 en [5] se desarrolla un modelo LDA dinámico denominado DTM (*Dinamic Topic Model*), en el que se introduce la variable temporal y los *topics* pueden ir cambiando a lo largo del tiempo. En el artículo [4] nos encontramos con otra variante de LDA llamada CTM (*Correlated topic model*) que nos permite encontrar correlaciones entre *topics*, ya que algunos temas es probable que sean más similares entre sí. Por último, nos encontramos con una variante de LDA denominada ATM (*Author-Topic Model*) propuesta por Michal Rosen-Zvi en su artículo [30] y desarrollada por el mismo en 2010, en la que los documentos son una distribución probabilística tanto de autores como de *topics*.

Podemos encontrar un resumen más completo del estado del arte en cuanto a la modelización de *topics* en el artículo [21].

2.2. Deep Learning y aplicación al PLN

El objetivo de esta sesión es entender el concepto de *Deep Learning* y analizar el estado del arte del *Deep Learning* aplicado al Procesamiento del Lenguaje Natural. Para poder entender el *Deep Learning* es conveniente entender los modelos de aprendizaje supervisados y saber qué provoca su aparición y popularidad de los últimos años. Posteriormente nos centraremos en los fundamentos del *Deep Learning* y cómo es utilizado en la representación de palabras. Por último, comentaremos algunas arquitecturas especializadas y su aplicación en el ámbito del

Procesamiento del Lenguaje Natural.

2.2.1. Aprendizaje supervisado

El aprendizaje supervisado consiste en aprender una función a través de un conjunto de datos llamados de entrenamiento, mediante la cual podamos obtener una salida a partir de una determinada entrada. Se espera que esta función, una vez realizado el entrenamiento, sea capaz de producir una salida correcta incluso para datos nunca vistos. Es muy habitual el uso de estos tipos de algoritmos para casos de clasificación y/o predicción.

Buscar entre todas las posibles infinitas funciones para encontrar la que mejor se adapte a nuestro conjunto de datos es un trabajo inviable, es por ello que normalmente se realiza la búsqueda entre un conjunto de funciones limitadas. En un primer lugar, y hasta hace aproximadamente una década, los modelos más populares de aprendizaje supervisado fueron los modelos lineales, provenientes del mundo de la estadística, estos modelos son fáciles de entrenar, fáciles de interpretar y muy efectivos en la práctica.

A partir de entonces, y motivado en parte por el aumento en las capacidades de cómputo, surgen otros modelos como las máquinas de vectores de soportes (*Support Vector Machines*, SVMs) o las redes neuronales, en las que nos centraremos en el siguiente apartado.

2.2.2. Deep Learning

Dentro del *Machine Learning* y usualmente relacionado con el aprendizaje supervisado, nos encontramos con un sub-campo denominado **Deep Learning** que utiliza las redes neuronales para la creación de modelos.

Como su nombre indica las redes neuronales consisten en unidades de cómputo llamadas neuronas que están interconectadas entre sí. Una neurona es una unidad de cómputo que posee múltiples entradas y una salida, esta neurona multiplica cada entrada por un peso para posteriormente realizar una suma y, por último, aplicar una función de salida no lineal. Si los pesos se establecen correctamente y tenemos un número suficiente de neuronas, una red neuronal puede aproximar a un conjunto muy amplio de funciones matemáticas.

En las redes neuronales, las neuronas suelen organizarse por capas que se encuentran conectadas entre sí. Mientras más capas tengamos, más características podremos extraer de nuestros datos de entrada y podremos aproximar un mayor número de funciones (sin perder de vista el sobrentrenamiento).

El primero y más simple de los tipos de redes neuronales es el denominado *Feed Forward Neural Network* (**FFNN**), este tipo de redes recibe este nombre porque no existen ciclos entre sus neuronas y las conexiones se realizan siempre desde las capas anteriores a las capas

posteriores.

Una de las arquitectura más comunes de FFNN es el preceptrón multicapa (en inglés multi-layer perceptron o **MLP**). Esta arquitectura contiene tres o más capas de neuronas totalmente conectadas, es decir, la salida de una neurona de una capa se encuentra conectada a la entrada de todas las neuronas de la siguiente capa. Las capas de una arquitectura MLP son:

- **Capa de entrada:** Se trata de la capa en la que introduciremos los datos en la red. Esta capa carece de procesamiento.
- **Capas ocultas:** Son las capas intermedias, cuyo número puede variar, tienen como entrada la salida de las neuronas de la capa anterior y su salida alimenta a las neuronas de la capa posterior.
- **Capa de salida:** Los valores de salida de las neuronas de esta capa se corresponden con la salida de la red.

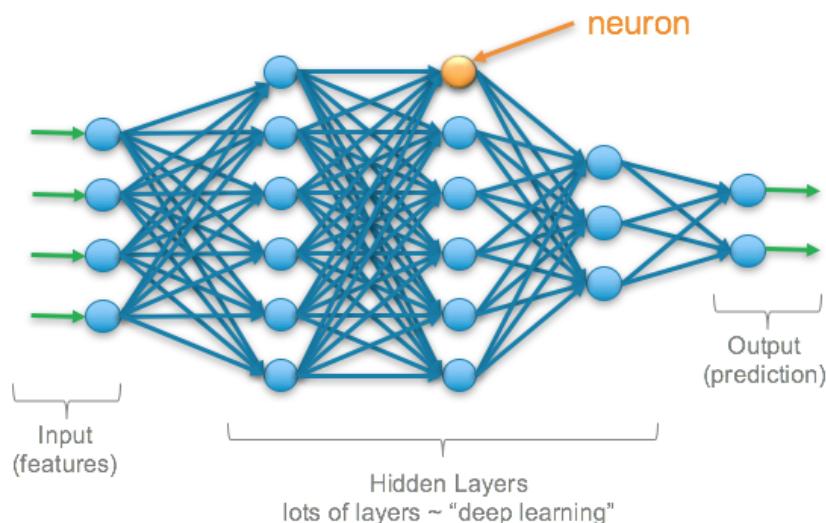


Figura 2.1: Ejemplo de arquitectura MLP. Fuente [29]

Hablamos que una red es profunda cuando contiene un gran número de capas, por ello el término de *Deep Learning*. En la Figura 2.1 observamos un ejemplo de arquitectura MLP y como la denominamos *Deep Learning* al crecer el número de capas ocultas.

2.2.3. Representación de palabras en PLN

Es usual, en el ámbito del reconocimiento de imágenes, utilizar información acerca de la dimensionalidad de las mismas. Este tipo de información nos permite extraer características

teniendo en cuenta los píxeles vecinos. Tradicionalmente, en el ámbito del Procesamiento del Lenguaje Natural, esto no se ha llevado a cabo debido a que cada palabra (o n-grama) se trataba como una entidad aislada utilizando una codificación de las palabras denominada **one-hot encoding**.

En cambio, existe otro método de representar las palabras en el lenguaje natural que sí es capaz de captar la “dimensionalidad” de una forma similar a como lo realizamos en las imágenes. Este modo, conocido como ***word embedding***, deja de tratar la palabra como un ente aislado y es capaz de captar el significado de la misma, esta representación se denomina distribuida y consiste en convertir las palabras en vectores en los que cada dimensión capte características diferentes de las palabras. Este tipo de representaciones dará lugar a vectores similares para palabras semánticamente parecidas.

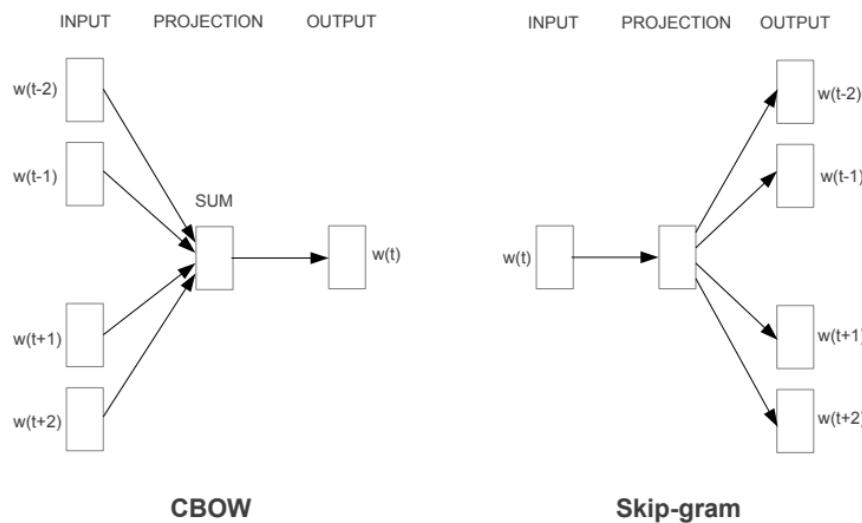


Figura 2.2: Arquitecturas CBOW y Skip-gram. Fuente [23]

Una de las soluciones más populares que nos permiten convertir una palabra a un vector (*word2vec*) que contenga información de la palabra en función del contexto se detallan en el artículo [23]. Aquí se presentaron dos modelos llamados **Skip-Gram** y **CBOW** cuya arquitectura podemos ver en la Figura 2.2. Estos modelos utilizan redes neuronales para predecir una palabra en función de su contexto o el contexto en función de una palabra, el vector que se utiliza para representar la palabra es el vector de pesos de la capa oculta.

2.2.4. Arquitecturas especializadas

Después de introducir las redes neuronales y el modo en el que podemos representar las palabras, frases o documentos para ser usados como entrada en nuestro modelo; vamos a cen-

trarnos en comentar dos tipos de redes neuronales que se usan de manera tradicional en tareas de Procesamiento de Lenguaje Natural.

Los dos tipos de redes neuronales que comentaremos son las redes neuronales convolucionales y las redes neuronales recurrentes. Haremos una introducción a cada una de ellas, comentaremos sus aplicaciones al PLN y sus ventajas e inconvenientes con respecto a otro tipo de métodos.

2.2.4.1. Redes neuronales convolucionales

Las redes neuronales convolucionales (llamadas usualmente CNN, por su nombre en inglés *Convolutional Neural Networks*) son un tipo de redes neuronales que deben su nombre a la operación matemática de convolución que realizan. Esta operación consiste en aplicar a una matriz de entrada multidimensional, un filtro o kernel también multidimensional y obtener una salida, también denominada mapa de características. En la Figura 2.3 podemos ver una representación gráfica de esta operación para un ejemplo de 2 dimensiones.

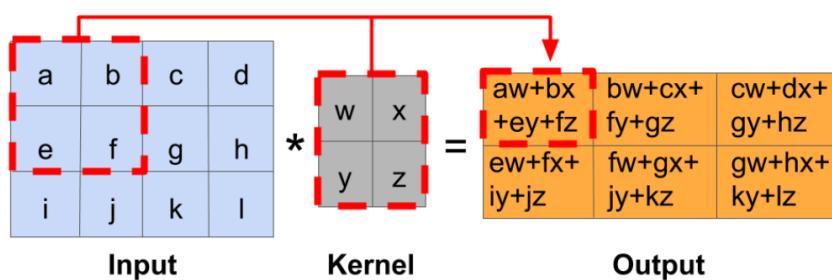


Figura 2.3: Ejemplo de una convolución de dos dimensiones. Fuente [2]

Es usual en las redes convolucionales utilizar diferentes kernels sobre una misma entrada, obteniendo diferentes salidas que permitan reconocer distintos patrones. Los pesos del kernel junto con el sesgo, son los parámetros que serán necesarios calcular en el entrenamiento y en el caso de las redes convolucionales, se denominan mapas de características.

Aunque la convolución simple que comentamos es la operación básica en las redes convolucionales, es usual añadirle algunas variantes (o configurarla con algunos parámetros) que nos permitan variar la dimensión de salida una vez hemos aplicado la convolución; algunas de estas variaciones más comunes son el **zero padding** y la **convolución por pasos**.

Como observamos en la Figura 2.3, al aplicar el kernel a los datos de entrada estamos reduciendo la dimensionalidad de la salida. A menudo es posible que esto no nos interese y queramos mantener la dimensionalidad en la salida; para ello recurrimos a un método denominado **zero padding** que consiste en añadir '0s' en los bordes de nuestra entrada con el objetivo de preservar la dimensionalidad en la salida. En la Figura 2.4 podemos ver un ejemplo de aplicación de **zero padding**.

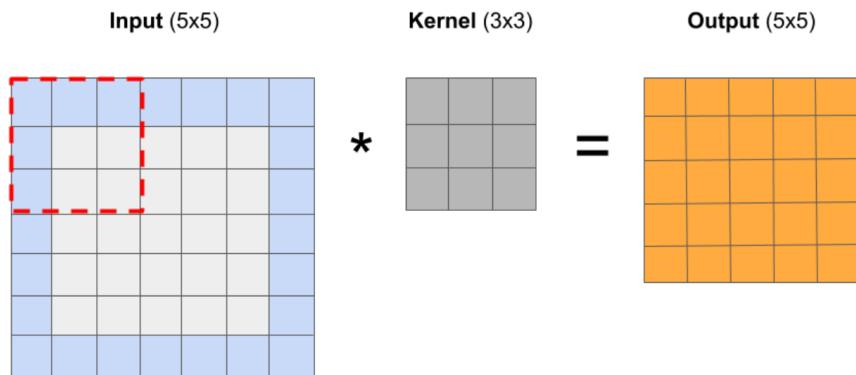


Figura 2.4: Ejemplo de aplicación de *zero padding* para mantener la dimensionalidad. Fuente [2]

Por otro lado, es también posible que queramos reducir aún más la dimensión de salida, principalmente por un tema de eficiencia y reducción de los tiempos de ejecución, a costa de perder información de algunas características en la salida. Para ello podemos utilizar la convolución por pasos (o *strided* por su nombre en inglés). Este método consiste en aplicar el kernel realizando saltos en lugar de hacerlo sobre celdas consecutivas, tal y como podemos ver en la Figura 2.5.

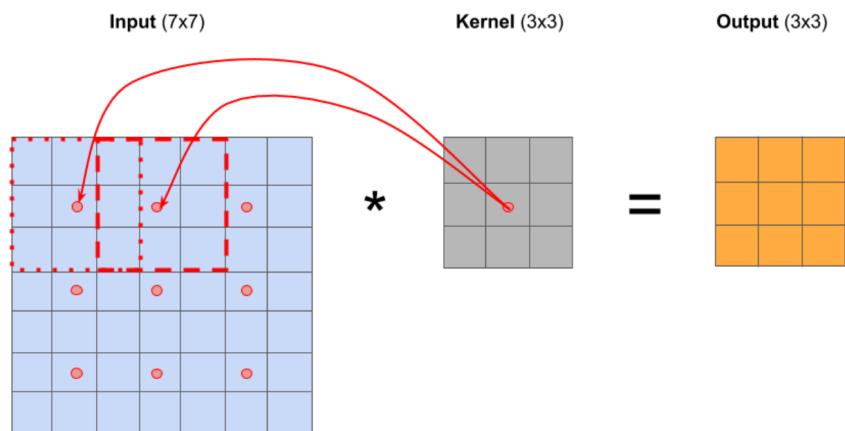


Figura 2.5: Ejemplo de aplicación de convolución por pasos para reducir la dimensionalidad. Fuente [2]

El proceso explicado anteriormente correspondería con una capa convolucional, que son el corazón de las redes neuronales convolucionales, sin embargo, en una red neuronal convolucional estas capas coexisten con otro tipo de capas que nos ayudaran a mejorar nuestros modelos. Las más usuales son:

- Capa de agrupamiento (*pooling* en inglés): El objetivo de esta capa es agrupar un conjunto

de salidas para obtener un único valor. Al conjunto de valores de entrada (seleccionado de nuevo con un filtro) se le aplica una función para obtener un único valor. Aunque se pueden utilizar diferentes funciones, como puede ser la media, lo más usual es aplicar la función de máximo (*max-pooling*). Es habitual, intuir que usando esta función nos estamos quedando con las características más relevantes de cada cuadrante (del tamaño del filtro) de la entrada. En la Figura 2.6 podemos ver un ejemplo de agrupamiento utilizando la función de máximo.

- Capa totalmente conectada: Hemos visto ejemplos de capas totalmente conectadas al introducir las redes neuronales, esta capas usualmente se usan al final de nuestra red para tareas de clasificación, teniendo la última capa un número de neuronas igual al número de clases que pretendemos clasificar.
- Capa RELU: Si observamos la descripción de la operación de convolución nos damos cuenta de que se trata de una operación totalmente lineal, es por ello que después de cada capa de convolución es usual agregar una capa no lineal (también llamada capa de activación). Aunque se pueden utilizar otras funciones como la tangente o la función sigmoide, lo más usual es utilizar la función RELU.
- Capa de Dropout: Esta capa tiene como funcionalidad prevenir el sobreentrenamiento en las redes neuronales, desactivando un número aleatorio de entradas de la capa, forzando a la red a ser redundante y permitiendo dar una clasificación correcta sin tener todas las entradas activas.

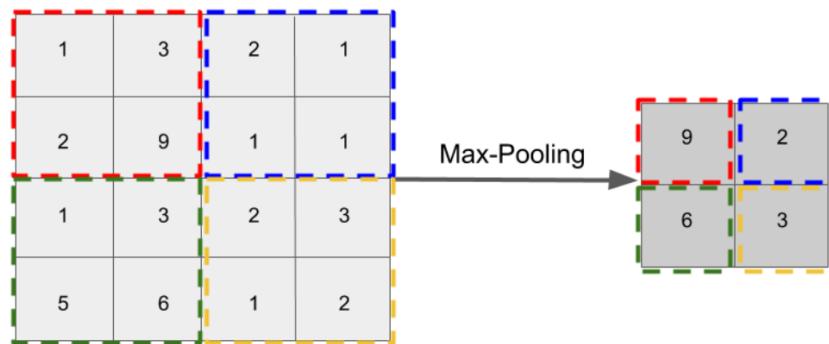


Figura 2.6: Ejemplo de aplicación de *max-pooling*. Fuente [2]

Tras esta visión general sobre las redes neuronales generales, podemos enumerar las ventajas que conllevan:

- Por un lado, aunque no hemos entrado en detalles sobre el proceso de entrenamiento de las redes convolucionales, se puede intuir que el aplicar un mismo kernel sobre toda la entrada provoca que el número de parámetros a aprender (los valores del kernel) con respecto a una red totalmente conectada será mucho menor. Esto provoca una **reducción del tiempo de entrenamiento necesario**.
- Por otro lado, el hecho de compartir el kernel provoca que podamos **capturar una misma característica en la entrada a pesar de su traslación**. Por ejemplo, si estamos detectando un objeto en una imagen un modelo convolucional podrá detectar ese objeto a pesar de su movimiento por la imagen.

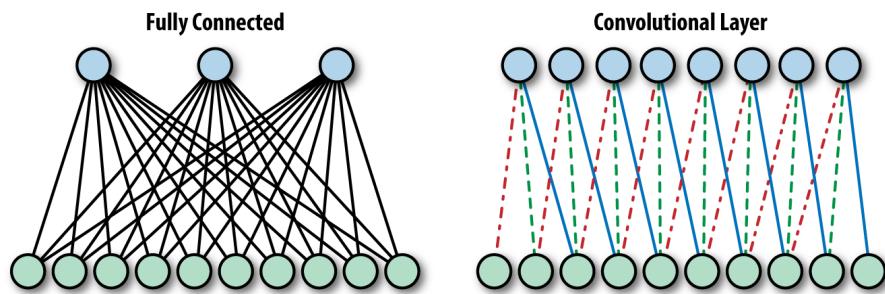


Figura 2.7: Comparación capa tradicional totalmente conectada con capa convolucional. Fuente [15]

Sin embargo, las redes neuronales convolucionales deben usarse en datos que contengan coherencia local ya que esa es su fortaleza. **En datos sin coherencia local las redes neuronales convolucionales no lograrían obtener un buen rendimiento** como las redes neuronales tradicionales vistas anteriormente. Si observamos la Figura 2.7 podemos ver la diferencia entre las capas de ambos tipos de redes y cómo la capa convolucional se centra más en las estructuras locales.

Encontramos una explicación más profunda sobre las redes convolucionales y su uso general en [2]. Aunque, como podemos imaginar, el uso más extendido de este tipo de redes es para el tratamiento de imágenes, nosotros nos centraremos en tener una breve visión de su aplicación al Procesamiento del Lenguaje Natural, que también puede ampliarse en [13].

Al aplicar las redes tradicionales al PLN, solemos ignorar el orden en el que las palabras aparecen en las frases, o las frases en el documento, siguiendo una aproximación CBOW; esto suele ser problemático a la hora de realizar, por ejemplo, un análisis de sentimientos ya que no es lo mismo encontrar la palabra “malo” aislada que el bigrama “no malo”. Aunque el uso de bi-gramas y N-gramas de mayor orden puede mejorar esta situación el coste puede volverse inasumible.

Es en este ámbito dónde las redes neuronales convolucionales pueden ser de gran ayuda, ya que gracias a la capacidad comentada para detectar estructuras locales serían capaces de identificar estos N-gramas de forma automática para ser usados posteriormente en tareas predictivas.

2.2.4.2. Redes neuronales recurrentes

Otro de los modelos usualmente usados en tareas de Procesamiento del Lenguaje Natural son las redes neuronales recurrentes, (llamadas usualmente RNN, por su nombre en inglés *Recurrent Neural Networks*). Hasta ahora todos los modelos de redes neuronales que hemos citado funcionaban siempre en una dirección, las neuronas de una capa anterior producían una salida que era la encargada de activar las neuronas de la capa posterior. En las redes recurrentes veremos que las salidas de una neurona en una capa posterior pueden tener una conexión con una neurona de una capa anterior. Esto crea una especie de **memoria** que nos permite modificar la respuesta de la red en función de los datos que se hayan procesado anteriormente (incluso reaccionar con datos que lleguen posteriormente).

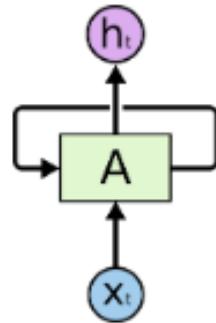


Figura 2.8: Ejemplo RNN. Fuente [28]

Las conexiones en una red neuronal recurrente pueden tener muchas variaciones por lo que es usual hablar del concepto de **celda**. Una celda suele tener como entrada los valores de la secuencia y el estado de la red neuronal en el paso anterior; y como salida la respuesta de la red neuronal a dicha entrada y el estado de la red neuronal en el paso actual.

En la Figura 2.8 observamos un ejemplo de red neuronal recurrente en el que tenemos como entrada el valor de la secuencia x_t y el estado de la red en el paso anterior (conexión en bucle). Producimos una respuesta h_t y un estado (conexión en bucle). Sin embargo, posiblemente esta representación sea algo más confusa para comprender su funcionamiento que si procedemos a “desenrollar” la red neuronal haciéndola más similar a los modelos vistos hasta ahora. En la Figura 2.9 podemos ver el resultado de “desenrollar” la red; hay que tener en cuenta con esta representación que los parámetros usados por cada celda son exactamente los mismos.

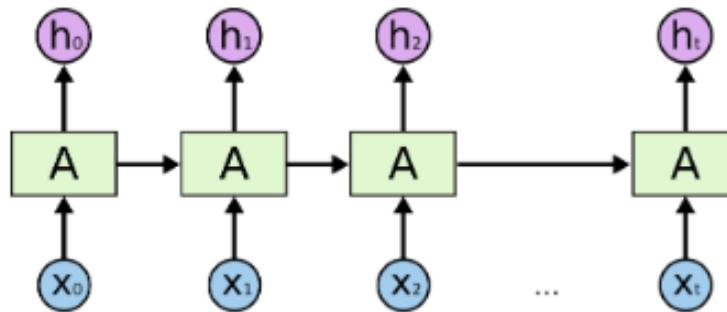


Figura 2.9: Ejemplo RNN “desenrollada”. Fuente [28]

Aunque no entraremos en detalles sobre el entrenamiento de redes neuronales, es importante saber que existen dos problemas diferentes provocados ambos por usar los mismos parámetros en todas las celdas que provocan la inestabilidad durante el proceso de entrenamiento. Estos problemas son la **desaparición del gradiente**, que ocurre al multiplicar el gradiente consigo mismo múltiples veces cuando este es menor que 1, y la **explosión del gradiente**, que ocurre por el mismo motivo cuando este es mayor que 1.

Para mitigar estos problemas es importante el diseño de las celdas, a continuación veremos de manera resumida los dos tipos de celdas más usados en las redes neuronales recurrentes. Las celdas que comentaremos están compuestas por diferentes mecanismos internos, denominados puertas, que gestionan el flujo de información a lo largo de la misma.

El primer tipo de celdas son las celdas *Long Short Term Memory* (**LSTM**). Este tipo de celdas se comportan bien en situaciones que queremos encontrar patrones entre registros que se encuentran separados en la secuencia, esto es algo muy usual, por ejemplo, en el caso de PLN cuando en una misma frase una palabra hace referencia a otra que apareció a una distancia de varias palabras.

Para conseguir este objetivo, parece evidente que es importante controlar la memoria en cada una de las celdas. Una celda LSTM realiza esta tarea con las siguientes puertas:

- Puerta de entrada: Controla qué información se añade a la memoria de la red.
- Puerta de olvido: Controla, a partir de la memoria del paso anterior y de la entrada, qué información debe conservarse en la memoria.
- Puerta de salida: Es la encargada de calcular la salida de la red, h_t , en el paso actual.

Debido al éxito de las celdas LSTM, y al constante esfuerzo de optimización de las mismas, han surgido diferentes variantes. La más conocida de todas son las *Gated Recurrent Unit* (**GRU**) introducidas en 2014. La celda GRU es una simplificación de la celda LSTM y produce unos

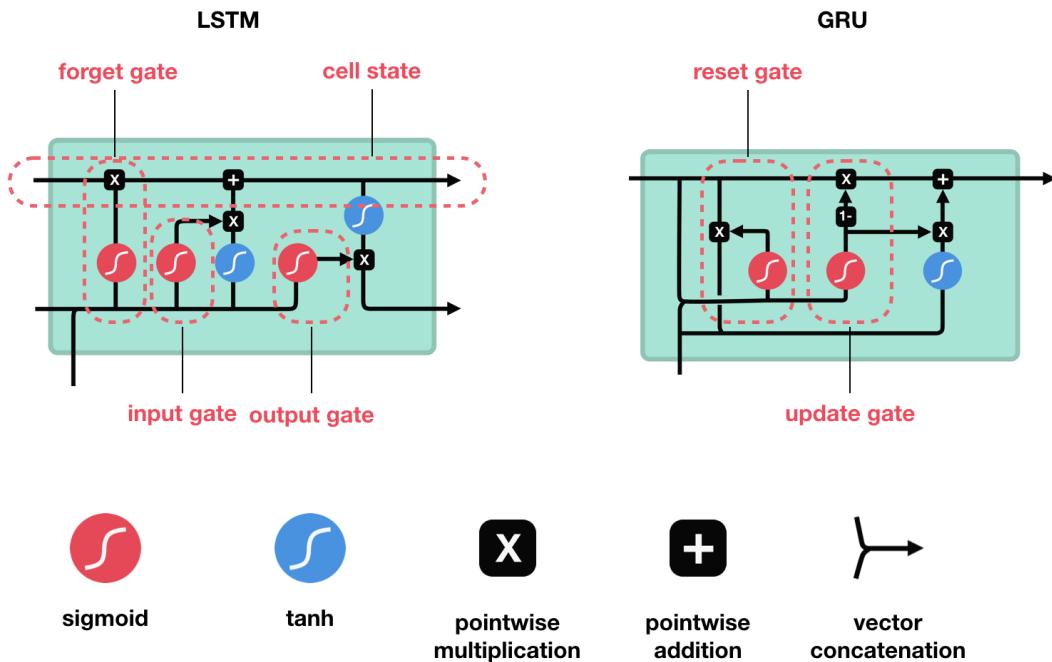


Figura 2.10: Arquitectura celdas LSTM y GRU. Fuente [27]

rendimientos bastante similares con un menor coste. De manera muy resumida una celda GRU se compone de:

- Puerta de reset: Permite seleccionar que información de la memoria va a ser utilizada en un paso concreto.
- Puerta de actualización: Realiza la función de las puertas de olvido y de entrada que hemos visto en las celdas LSTM.

Aunque no hemos entrado en el detalle del funcionamiento de cada una de las puertas, en la Figura 2.10 podemos ver la arquitectura completa de ambos tipos, y observar la mayor simplicidad de las celdas GRU frente a las LSTM.

Como podemos imaginar, las redes neuronales recurrentes son ampliamente usadas en el mundo del Procesamiento del Lenguaje Natural, debido a que una palabra no es otra cosa que una secuencia de letras, una frase a su vez se trata de una secuencia de palabras y un documento una secuencia de frases. Alguno de los usos de las RNN en este ámbito son:

- **Análisis de sentimientos:** Detectar el sentimiento positivo o negativo de un texto utilizando únicamente la última salida de la red. En [20] tenemos un ejemplo de análisis de sentimientos utilizando redes recurrentes con LSTM sobre los datos de una red social China.

- **Generador de texto:** Predecir la siguiente palabra de una secuencia, utilizando la salida de cada una de las celdas. Podemos ver una comparación de métodos para generar textos de diferentes temáticas para poder entrenar posteriormente modelos *Deep Learning* en [1]. Entre los métodos de la comparación se encuentran las redes neuronales recurrentes con celdas GRU y con celdas LSTM.
- **Traductores:** Traducción automática de textos entre idiomas, llamado *Neural Machine Translation* (NMT) cuando se utilizan redes neuronales. Quizás el mayor caso de éxito en este punto es el *Google's Neural Machine Translation System* [36] cuya base es una red neuronal profunda construida con celdas LSTM.

Una vez analizadas, de manera global, las redes neuronales recurrentes podemos ver que nos ofrecen **numerosas ventajas al trabajar con secuencias** como hacemos en el ámbito del PLN. Entre ellas podemos ver, que vamos incluso más allá que con las CNN analizando la relación entre las diferentes palabras, fundamentalmente con las celdas LSTM y GRU, pudiendo detectarlas entre palabras que estén alejadas entre sí y no ceñirnos al tamaño del kernel. Sin embargo, como hemos visto, y aunque sean mitigados por el uso de celdas LSTM y GRU, las redes neuronales recurrentes presentan **problemas durante el entrenamiento** tanto de desvanecimiento como de explosión del gradiente.

2.3. *BigData* y *Fast Data*

A lo largo de esta sección intentaremos tener una visión general del *Big Data* y su evolución a lo largo del tiempo hasta llegar al *Fast Data*. Posteriormente veremos las arquitecturas más usadas en el mundo del *Big Data*.

2.3.1. Evolución: del *Big Data* al *Fast Data*

El primer uso del término *Big Data* se da en un artículo de Michael Cox y David Ellsworth de la NASA publicado en 1997 [8], donde hacen referencia a la dificultad de procesar grandes volúmenes de datos con los métodos de la época. Sin embargo, fue en 2001 cuando encontramos la definición más conocida y aceptada de *Big Data* hecha por el analista Laney Douglas en su artículo “*3D Data Management: Controlling Data Volume, Velocity y Variety*” [17] en el que se hacía referencia a las ya “famosas” tres Vs:

- **Volumen:** Cada vez los volúmenes de datos son mayores.
- **Velocidad:** Es cada vez mayor la velocidad con la que se generan los datos.

- **Variedad:** Dejamos de tener únicamente datos completamente estructurados para trabajar con datos no estructurados y/o semi-estructurados.

Google, como es obvio, también se enfrentó a un importante problema a la hora de procesar la ingente cantidad de datos que generaba día a día y que no podían ser procesados de manera eficiente con el *software* existente, es por ello que en el año 2003 presenta en [12] su “*Google File System*” (GFS) y un año después *Map Reduce* [9], estas dos capas de almacenamiento y procesamiento distribuido dieron lugar al nacimiento de lo que hoy conocemos como ***Big Data***.

Sin embargo, estas aportaciones no empezaron a tomar una repercusión relevante fuera de Google hasta el nacimiento del *framework* Hadoop en 2006, un ecosistema con una gran cantidad de servicios pero cuya base fue Map Reduce y HDFS (basado en GFS). La complejidad del ecosistema *Hadoop* hizo que éste no empezara a aparecer en la mayoría de las empresas hasta la creación de la compañía *Cloudera* en 2009, que empezó a empaquetar los diferentes componentes del ecosistema *Hadoop*, ofreciendo distribuciones estables y soporte para sus clientes.

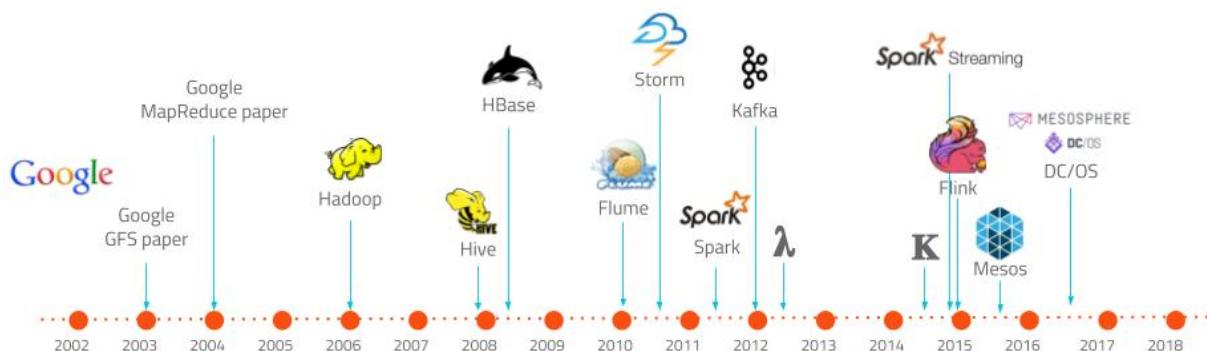


Figura 2.11: Evolución del *Big Data*. Fuente [10]

Durante estos 10 años la popularidad de *Hadoop* ha crecido exponencialmente y junto con las BBDD NoSQL, nacidas también a partir de Google con su BigTable, forman lo que hoy conocemos como Big Data. En la Figura 2.11 observamos esta evolución en el mundo del *Big Data* con los hitos de aparición de algunas tecnologías representativas.

El auge del **Big Data** ha llevado a algunas empresas a tener verdadera obsesión por el almacenamiento de todos los datos de sus clientes y las operaciones realizadas, creando inmensos *datalakes* donde tener enormes históricos de todos sus datos. Este “síndrome de Diógenes digital” creado por falsas expectativas, por la imposibilidad de extraer valor de los datos o por la dimensión cambiante de las empresas actuales (en la que los datos de años atrás pueden no ser relevantes en el presente), es uno de los posibles motivos por lo que el tratamiento de los datos está cambiando. Otro de los motivos para el cambio de rumbo del *Big Data* está relacionado con la *V* de Velocidad, hoy en día no solo es importante la capacidad de ingestar rápidamente

los datos, sino la capacidad de poder procesar y obtener decisiones o actuar en tiempo real a partir de los datos, aportando valor al negocio. En este escenario se vuelve más importante la velocidad que el volumen de datos, esto es lo que se denomina *Fast Data*.

Dentro del *Fast Data* es habitual el uso de BBDD *in-memory*, de buses de eventos y de tecnologías de procesamiento capaces de procesar los eventos en tiempo real. Como veremos posteriormente al desarrollar nuestra arquitectura, el *Fast Data* será una parte fundamental en nuestro proyecto en el que tendremos que clasificar las llamadas en tiempo real y tomar decisiones (o alarma) en función de las mismas.

Observando de nuevo la Figura 2.11 podemos ver este cambio en la tendencia hacia el *Fast Data* a partir del 2012 cuando aparece la tecnología Kafka y en los años posteriores con la incorporación de diferentes herramientas para el procesamiento de eventos como son *Spark Streaming* o *Flink*.

2.3.2. Arquitecturas *RealTime*

La evolución que hemos visto en el apartado anterior, con la explosión del *Big Data* y la irrupción del *Fast Data*, hace necesaria la incorporación en las empresas de arquitecturas de procesamiento de datos en tiempo real que, como cualquier otra arquitectura de datos, sean capaces de ingestar, procesar y permitir la explotación y análisis de los datos. La diferencia fundamental en las arquitecturas *RealTime* y las arquitecturas de datos tradicionales son el **volumen de los datos a tratar** y la **capacidad para hacerlo en tiempo real**.

Como veremos, no existe una arquitectura que se adapte a todos los casos de uso (*one-size-fits-all*) y, según la necesidad, será necesario aplicar una u otra.

2.3.2.1. Arquitectura Lambda λ

La arquitectura lambda, representada por la letra griega λ , fue presentada en 2011 por Nathan Marz en un artículo publicado en su blog titulado “*How to beat the CAP theorem*” [22].

El propósito de Nathan Marz cuando crea su arquitectura, como indica el título del artículo, es batir el teorema CAP popularizado por la irrupción de las bases de datos NoSQL. Este teorema, ilustrado en la Figura 2.12, viene a decir que si queremos tener tolerancia a particiones (imprescindible para bases de datos distribuidas necesarias para el *Big Data*), tenemos que optar entre consistencia, asegurar que el dato que leemos es el último que hemos escrito, o disponibilidad, que la base de datos se encuentra siempre lista.

El método propuesto para conseguir este objetivo con grandes cantidades de datos se basa en los siguientes principios:

- Una capa *batch* eventualmente consistente de una manera extrema, en la que las escrituras

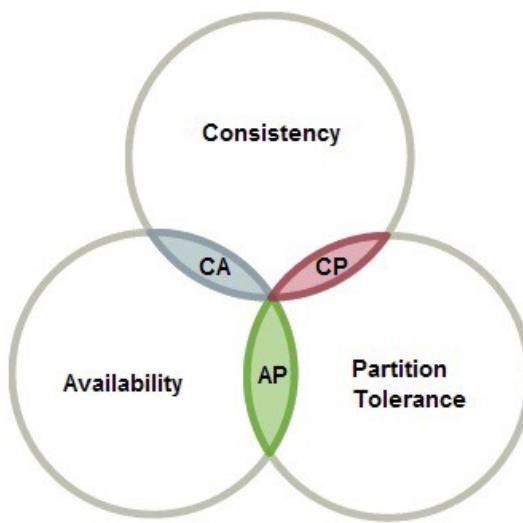


Figura 2.12: Teorema CAP. Fuente [26]

tardan siempre unas pocas horas en estar disponibles. Eliminando algunos problemas complejos con los que tratar como la concurrencia o las reparaciones de lectura.

- Reducir las operaciones CRUD (*Created, Read, Update, Delete*) en la capa *batch* por únicamente CR, tratando los datos como objetos inmutables. Esto nos soluciona el problema de la consistencia, ya que de este modo un dato existe o no existe, pero no puede tener varias versiones.
- Una capa *realtime* que se encarga de los datos de las últimas horas (los que no están disponibles en la capa *batch*)
- Las querys atacan a ambas capas de forma simultanea realizando un *merge* de los datos.

Podemos ver un esquema de la arquitectura en la Figura 2.13. Esta arquitectura, aparte de resolver los problemas de consistencia y disponibilidad, tiene algunas ventajas:

- Disponer de todos los datos en un único punto (capa *batch*) pudiendo realizar cualquier tipo de consulta sobre los mismos.
- Al utilizar los datos como un ente inmutable facilita las auditorias.
- Según Marz, evita el error humano en la capa *batch* (en parte también por usar datos inmutables), y cualquier error en la capa *realtime* sería subsanado en pocas horas en la capa *batch*.

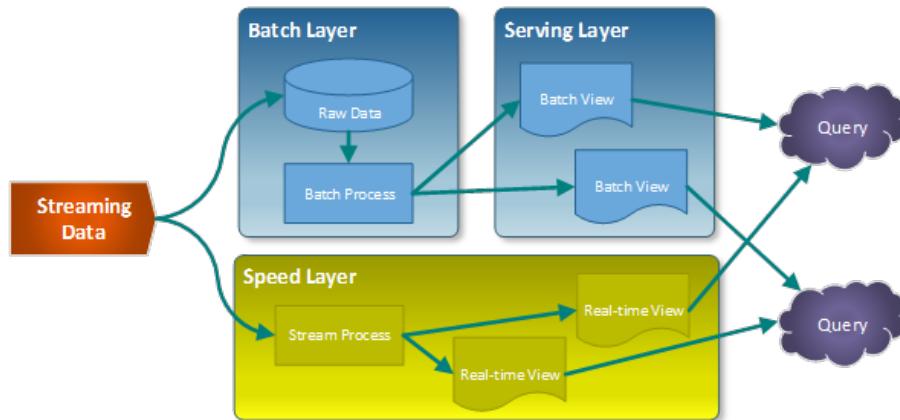


Figura 2.13: Arquitectura Lambda definida por Nathan Marz. Fuente [10]

2.3.2.2. Arquitectura Kappa κ

En su artículo “Questioning the Lambda Architecture”[16], Jay Kreps cuestiona la arquitectura Lambda propuesta por Nathan Marz y propone una simplificación de la misma, basada en su experiencia en LinkedIn trabajando con Kafka y Samza. Esta simplificación se denomina arquitectura Kappa y viene representada por la letra griega κ .

Kreps describe la complejidad que supone en una arquitectura Lambda mantener idénticos procesos en *realtime* y *batch*. También expone que se menosprecia la capacidad de la capa *realtime* (probablemente por la madurez del procesamiento *realtime* con respecto al *batch*) y opina que es posible realizar el mismo procesamiento, incluso reprocesar el histórico de datos, en la capa *realtime*.

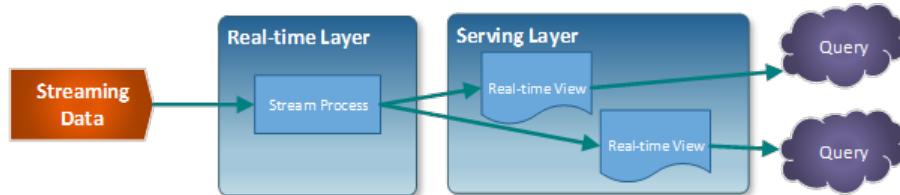


Figura 2.14: Arquitectura Kappa definida por Jay Kreps. Fuente [10]

Con esta premisa, en el artículo se presenta la arquitectura que observamos en la Figura 2.14, en la que existe un único flujo de procesamiento *realtime* para todo el modelo. La simplicidad de Kappa con respecto a Lambda es tal que el propio Kreps afirma que puede ser una idea demasiado simple para merecer una letra griega.

2.4. Trabajos anteriores

Una vez abordado el estado del arte desde diferentes puntos de vista, es importante tener una visión de los trabajos anteriores que se han realizado con objetivos similares y sus resultados. De este modo entenderemos si existe una justificación para nuestro trabajo, los problemas a los que podemos enfrentarnos y las expectativas que podemos gestionar.

En este apartado nos vamos a centrar en la aplicación de técnicas de Procesamiento del Lenguaje Natural a un *Call Center*. Encontramos varios artículos interesantes que hacen hincapié en el valor de la información y el conocimiento que puede extraerse de un *Call Center*, ya que se trata de un intermediario importante entre el cliente y la empresa. Entre estos trabajos podemos destacar:

- “Metodología para estimar el impacto que generan las llamadas realizadas en un call center en la fuga de los clientes utilizando técnicas de text mining” [31]: Que como su nombre indica, investiga si existe relación entre las llamadas realizadas al *Call Center* y la pérdida de clientes. El trabajo, al igual que el proyecto que intentamos abordar, parte de las llamadas transcritas a texto y, aunque se utiliza como base para la modelización de temas LDA, se apoya en etiquetas existentes en las llamadas para validar los resultados.
- “Customer voice sensor: A comprehensive opinion mining system for call center conversation” [19]: Se trata de un trabajo más basado en el análisis de sentimientos, pero se encuentra realizado con llamadas de los clientes a una operadora de telecomunicaciones (en este caso China Telecom) al igual que nuestra fuente de información.
- “Topic mining for call centers based on A-LDA and distributed computing” [14]: En este caso, se realiza una modelización de temas sobre los datos del *Call Center* de *China Central Television*. En este proceso de modelización se utiliza una mejora del modelo LDA llamada A-LDA que utiliza no solo el corpus de la llamada, si no también algunas propiedades externas como el tiempo de llamada o el número de origen.
- “Author-topic based representation of call-center conversations” [25]: Este último artículo que comentamos, parte también de los datos generados a través de un *Performance of Automatic Speech Recognition* (ASR), el trabajo pone de manifiesto la pobre calidad de estas transcripciones automáticas. Por este motivo, propone una modificación de LDA basada en el modelo *Author Topic*, utilizando además del corpus del texto información del tema de la conversación.

Probablemente, dentro de las empresas, existen muchos más trabajos destinados a la explotación de esta información y que no se encuentren publicados, por lo que podemos concluir

que es un campo que despierta interés y que se trata de una información con potencial que nos permita, entre otros objetivos, comprender las necesidades de los clientes de una compañía.

Por otro lado, observamos que en la mayoría de los casos, aunque la base de la modelización sea el uso de LDA, debido al ruido o a otros factores como la ausencia de información semántica, se utilizan modificaciones del modelo que incorporan etiquetas o propiedades externas al corpus para mejorar la clasificación.

Por último, pensamos que, aún con estos antecedentes, existe la necesidad de abordar este proyecto, debido a la diferencia entre unos datos y otros, por factores como el idioma o las distintas necesidades de cada empresa. Además nuestro proyecto tiene como objetivo final la integración de este modelo en un proceso de la compañía que sea capaz de aplicarlo en tiempo real y alertar en caso de anomalías para poder tomar decisiones.

Capítulo 3

Arquitectura y tecnologías

El objetivo de este capítulo es tener una visión global de lo que será nuestro proyecto, antes de entrar en más profundidad en cada una de las partes. El proyecto que presentamos en este documento consta de dos partes bien diferenciadas. Por un lado, una parte que podemos considerar “de laboratorio”, en la que realizaremos las labores más analíticas sobre los datos disponibles: extracción, procesamiento, estudio y creación de modelos; esta última la veremos en la sección 3.1. Y por otro lado, una parte “de explotación”, en la que trataremos de poner en valor los resultados de la primera parte en el mundo real y que será tratada en la sección 3.2.

El capítulo se completará con la sección 3.3, dedicada al mantenimiento del proyecto una vez llevado a producción y con la sección 3.4, en la que se describirán a grandes rasgos todas la tecnologías usadas en el proyecto.

3.1. Modelado

La primera parte de nuestro proyecto pretende conseguir el objetivo principal propuesto en la sección 1.3, **clasificar las llamadas**. Para ello tendremos que analizar y entender los datos que poseemos de las transcripciones y posteriormente construir los modelos necesarios (supervisados o no supervisados) que nos permitan clasificar las llamadas.

Para lograr este objetivo necesitaremos disponer de un *datalake* en el que se almacene todo el histórico de transcripciones de las llamadas, ya que para el entrenamiento de muchos modelos será necesario utilizar un amplio histórico para su entrenamiento. Este *datalake* será un sistema de archivos HDFS perteneciente a una plataforma Hadoop Hortonworks. El procedimiento de carga de las transcripciones en HDFS se encuentra ya realizado y queda fuera del alcance del proyecto. Debido a que no tenemos ningún requisito temporal para la ingestión de las transcripciones, este no será un elemento crítico de nuestro proyecto.

En cuanto a la plataforma que utilizaremos para realizar la analítica, puede ser independiente

del entorno de almacenamiento siempre que podamos transferir los datos a la misma. A lo largo del proyecto hemos trabajado con dos entornos diferentes:

- Entorno **Spark**: Ubicado en un clúster Hadoop Hortonworks, nos ha permitido procesar grandes cantidades de datos en un espacio de tiempo muy reducido, gracias a los beneficios de la programación distribuida. Sin embargo, no se trata del entorno ideal para el entrenamiento de modelos, principalmente para modelos *deep learning*.
- Entorno **GPUs**: Una única máquina con diferentes *GPUs* NVIDIA. Este entorno nos ha permitido entrenar los modelos de *deep learning* con una rapidez pasmosa (comparado con los entornos con *CPUs*). Ha sido la plataforma principal de analítica durante este proyecto, aunque el preprocesado se viera penalizado con respecto al entorno Spark.

El modelo es un elemento vivo en nuestra arquitectura y, además de por posibles mejoras en los hiperparámetros o por la tecnología, debe re-entrenarse conforme se vayan recibiendo datos nuevos en el histórico, ya que es lógico pensar que la temática de las consultas variarán a lo largo del tiempo debido por ejemplo al lanzamiento de nuevos productos.

En los capítulos [4](#), [5](#) y [6](#) trataremos en más detalle toda la parte de modelado, desde el análisis de los datos hasta la creación de modelos supervisados y no supervisados.

3.2. Explotación

La segunda parte del proyecto tendrá como meta llevar a producción el resultado de la etapa de modelado. Teniendo en cuenta que uno de nuestros objetivos consiste en clasificar las llamadas en tiempo real, el primer paso que debemos abordar es seleccionar un modelo de arquitectura de procesamiento en tiempo real de entre las opciones vistas en la sección [2.3.2.2](#): Lambda y Kappa. Lo que intentamos en este paso, ya que no existe una solución ideal que se adapte a todos los casos de uso, es encontrar la solución que mejor se adapte a nuestro proyecto.

En nuestro caso, disponemos de todas las llamadas en tiempo real, que nos llegarán en forma de eventos. Además cada llamada podrá ser tratada de forma individual y no debemos preocuparnos por eventos que lleguen con retraso. Estos son los motivos principales que nos han llevado a decidirnos por la arquitectura Kappa propuesta por Jey Kreps, debido a la mayor simplicidad tanto a la hora de elaborar la capa de servicio, como a la hora de mantener un único desarrollo *real-time*.

En la figura [3.1](#) podemos ver la arquitectura global de la solución propuesta. Esta arquitectura consta de dos capas principales. Por un lado, una capa de *streaming* o capa rápida donde

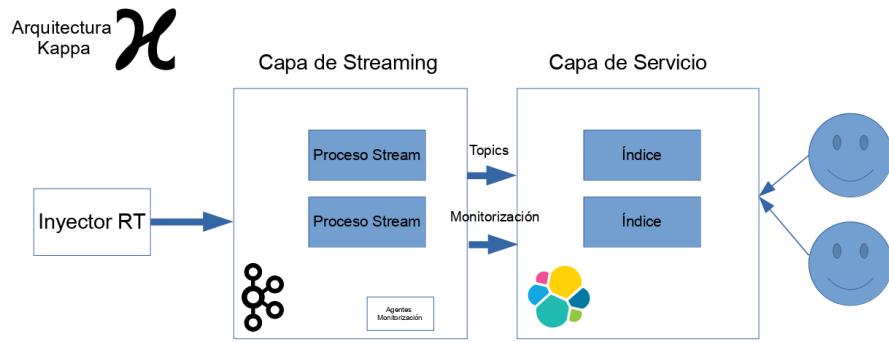


Figura 3.1: Arquitectura Kappa

se realizará todo el procesamiento en *streaming* de las transcripciones de las llamadas conforme vayan llegando. Por otro lado, una capa de servicio que permite a los usuarios explotar la información procesada en la primera capa.

El *core* de la primera capa será Apache Kafka, que además nos dará la posibilidad de retener los eventos y poder reprocesarlos en caso de error, siendo una solución ideal para arquitecturas Kappa. De hecho Jey Kreps, quien propuso la arquitectura Kappa es el co-fundador de Confluent, la compañía que se encuentra detrás de Apache Kafka. Los detalles de la capa de *streaming* podemos verlos con más detalle en el capítulo 7.

El núcleo de la capa de servicio será el *stack* de Elastic. Esta solución nos permitirá ingestar los eventos desde la capa rápida en tiempo real y exponerlos a los usuarios. Además usaremos este mismo flujo para todos los eventos de monitorización. El hecho de utilizar el *stack* de Elastic nos dará posibilidad de realizar el alarmado estático y dinámico en esta misma capa. Los detalles de esta capa se describen en detalle en el capítulo 8.

Debido a la situación actual, las llamadas no se ingestan en *real-time* si no que se reciben mediante procesos *batch* cada cierto tiempo. Este escenario cambiará en el futuro por lo que se construirá un elemento de entrada a la capa rápida que actúe como inyector, generando eventos en tiempo real a partir de los datos en *batch*. Esta pieza será suprimida una vez las llamadas sean recibidas en tiempo real.

3.3. Mantenimiento e Integración Continua

Como ya hemos visto al hablar del entrenamiento del modelo, el desarrollo de este tipo de proyectos no tiene un principio y un final, si no que se trata de un proceso cíclico en el que por necesidades del negocio, por cambio en los datos o por cambio en las tecnologías, será necesario añadir mejoras o modificaciones en nuestro desarrollo. También el despliegue del *software* en

producción es un proceso susceptible de futuros cambios.

Por estos motivos será necesario disponer de algún método para seguir evaluando la calidad del modelo una vez sea llevado a producción, tener la posibilidad de realizar test A/B en el futuro, y definir mecanismos que nos permitan, tras cada cambio efectuado, poder realizar las pruebas necesarias y desplegar estos cambios de una manera totalmente automatizada y sin intervención humana.

Para conseguir estos objetivos, con la mayor agilidad posible, será necesario trabajar en un marco de trabajo *DevOps* contando con mecanismos que nos permitan realizar tanto integración, como despliegue continuos. En el capítulo 10 veremos con más detalle como aplicamos esta metodología.

3.4. Tecnologías

Al igual que la arquitectura descrita anteriormente era la encargada de responder a las necesidades de negocio, las tecnologías descritas en este apartado nos darán las piezas necesarias para poder construir esa arquitectura y dar respuesta a nuestro caso de uso.

En el proceso de selección de las tecnologías, no solo se ha tenido en cuenta la idoneidad de las mismas para el caso de uso, si no que se ha valorado también la experiencia en la misma y la disponibilidad dentro del entorno de trabajo. Esto puede provocar que en algunos casos aunque la tecnología se adapte al caso de uso, puedan existir otras soluciones más óptimas cuyo uso era menos viable, dados los plazos de ejecución del proyecto.

A continuación enumeraremos las tecnologías agrupadas en las diferentes capas que hemos comentado en el apartado de arquitectura, además añadiremos las tecnologías que se usarán para la integración y despliegue continuo.

3.4.1. Modelado

La parte de modelado la realizaremos trabajando siempre con Python 3.6 y Jupyter, trabajar en un entorno dinámico e interactivo mediante *notebooks* nos dará mucha flexibilidad para poder realizar nuestro análisis.

A continuación vemos las bibliotecas más relevantes que hemos usado, tanto en entorno Spark como en el entorno de GPUs.

- **Spark SQL y *dataframes***: Dentro del entorno de Spark, para el procesamiento de los datos trabajaremos con las bibliotecas de Spark SQL y *dataframes*, que nos permitirán de una manera sencilla realizar transformaciones en nuestros datos de forma distribuida.

- **MLlib:** También en el entorno de Spark usaremos la biblioteca nativa MLlib, que nos permite crear algoritmos y modelos de *machine learning* sobre un cluster Spark.
- **Pandas:** Al igual que en *Spark* utilizamos las bibliotecas de SQL y *dataframes*, Pandas nos permitirá hacer el análisis y manipulación de los datos en cualquier entorno Python.
- **Keras:** Será la biblioteca que usaremos, con **Tensorflow** como backend, para crear modelos de *deep learning*.
- **Gensim:** Se trata de una biblioteca para el modelado de temas no supervisados y el procesamiento del lenguaje natural.
- **NLTK:** Son un conjunto de bibliotecas que nos ayudarán a realizar tareas de procesamiento de lenguaje natural.
- **Sklearn:** Se trata de una biblioteca para tareas de *machine learning*, que contiene multitud de funciones que nos ayudarán en nuestro desarrollo.
- **Optuna:** Por último, la biblioteca Optuna nos ayudará a optimizar los modelos supervisados que creemos. Hablaremos de ella con más detalle en la sección [6.2](#).

3.4.2. Explotación

A continuación veremos las tecnologías y plataformas que harán posible el despliegue de nuestra capa de *streaming*. Estas tecnologías serán vistas con más detalle a lo largo del capítulo [7](#).

- **Apache Kafka:** Es el *core* de la capa rápida, se trata de un bus de eventos distribuidos, a través del cual se realizará la ingesta o publicación de los eventos (llamadas). Las diferentes capas de procesamiento que requieran estos eventos se suscribirán a este Bus.
- **Tensorflow Serving:** Servicio para servir modelos de *machine learning* creado dentro del ecosistema Tensorflow.
- **Kafka Stream:** A la hora de procesar la información en eventos ingestada en nuestro Bus Kafka disponemos de Kafka Stream. Una serie de bibliotecas que nos permiten construir aplicaciones y microservicios cuyo origen y destino sean un Bus Kafka.
- **Openshift:** Se trata de una plataforma de contenedores de Kubernetes creada por *Red Hat*. Todos los desarrollos de la capa de explotación serán implementados sobre contenedores.

3.4.3. Capa Servicio

La capa de servicio estará compuesta, como hemos indicado, por el *stack* de Elastic. Este *stack* posee diferentes piezas, cada una de las cuales realiza una función determinada.

- **Elasticsearch:** Se trata del núcleo del *stack*. Aunque no se trata en el sentido más estricto de una BBDD No-SQL, si no de un motor de búsqueda, Elasticsearch nos permite almacenar la información en forma de documentos json en tiempo real y realizar consultas y agregaciones sobre cualquier campo. Entre las características que podemos aprovechar de Elasticsearch para nuestro objetivo están:
 - Ingesta en tiempo real.
 - Consulta en tiempo real.
 - Disponibilidad de mecanismos de ingesta (Logstash) y consulta (Kibana).
 - Posibilidad de crear alarmas en base a consultas.
 - Posibilidad de crear *jobs* de *machine learning* que detecten anomalías en series temporales.
 - API REST: Posibilidad de realizar cualquier operación o consulta mediante API REST.
- **Logstash:** Será la pieza que nos permitirá transportar la información desde la capa de *streaming* a la capa de servicio. Logstash nos permitirá leer de Apache Kafka, realizar las transformaciones necesarias y volcar la información resultante en Elasticsearch. Además de los datos propios de la clasificación, Logstash nos ayudará a extraer también los datos de monitorización de la capa de Streaming.
- **Kibana:** Será el frontal donde los usuarios podrán consultar sus diferentes cuadros de mando y construir nuevos de acuerdo a sus necesidades. También, gracias al módulo de *machine learning* de Elasticsearch, los usuarios podrán crear *jobs* de *machine learning* para detectar anomalías en los temas tratados y generar las alertas necesarias.
- **Beats:** Se trata de agentes ligeros que nos permitirán extraer información variada de distintas fuentes. En nuestro caso usaremos *Metricbeat* para extraer datos de monitorización de la capa de *streaming*.

3.4.4. Integración y Despliegue Continuo

Por último, para conseguir los objetivos de integración y despliegue continuos utilizaremos diversas tecnologías. Estas tecnologías, y la interacción entre ellas se verán con más detalle en

el capítulo [10](#).

- **BitBucket:** Será el repositorio usado para almacenar las nuevas versiones de nuestro *software* de manera que podamos tener un control de versiones. Almacenará tanto el código de nuestra capa de *streaming*, como de los modelos que necesitemos poner en producción.
- **Jenkins:** Es un servidor de integración continua *open source* que, mediante la creación de tareas, nos ayudará a realizar el *build* de nuestro software realizando de manera automática las pruebas necesarias.
- **Nexus:** Se trata de un gestor de repositorios, en nuestro caso utilizaremos repositorios de Maven. Es usual usar este tipo de gestores de repositorios en las empresas para disponer de bibliotecas propias (y no públicas) y para no tener una dependencia con ningún agente externo.
- **S2I:** Se trata de una funcionalidad de Openshift que será vital para el desarrollo del despliegue continuo. Esta funcionalidad nos permitirá crear nuevas imágenes de contenedores a partir de código o binarios propios.

Parte II

Modelado: datos, modelos y optimizaciones

Capítulo 4

Conjunto de datos

El primer paso cuando nos enfrentamos a un problema de minería de datos, es comprender el conjunto de datos con el que contamos y ver si se adapta a nuestras necesidades. En este caso, al tratarse de un proyecto que se desarrolla dentro de una empresa, en caso de duda, hemos tenido la posibilidad de acudir a las áreas dueñas del dato para solicitarle información adicional sobre el mismo.

En este capítulo abordaremos todos los aspectos relacionados con los datos que hemos ido recopilando durante el desarrollo del proyecto. En la sección 4.1, describiremos el conjunto de datos con el que hemos estado trabajando, posteriormente en la sección 4.2 veremos las diferentes formas que hemos usado para representar los datos que alimentan los modelos que hemos construido (y que describiremos en los siguientes capítulos) y, por último, en la sección 4.3 echaremos la vista atrás para ver el conjunto de datos inicial y poder apreciar la evolución que ha ido sufriendo.

Otro de los objetivos del capítulo es poner de manifiesto la importancia de la parte quizás menos “glamurosa” que está presente en todos los proyectos de minería de datos. Cuando exponemos un proyecto de minería de datos siempre ponemos el foco en los modelos creados, sin embargo, el grueso del trabajo en los proyectos de este tipo se encuentra en recopilar datos, entenderlos y preprocesarlos (o limpiarlos). Todo este proceso es el que intentaremos describir en las siguientes páginas.

En el código expuesto a lo largo del capítulo se hará referencia a un módulo llamado *mgmtfm*, se trata de un módulo en *Python* creado especialmente para este proyecto y que nos permitirá realizar las tareas necesarias con un nivel de abstracción mayor.

4.1. Análisis del conjunto de datos

En este apartado analizaremos el conjunto de datos con el que trabajaremos a lo largo de todo el documento. Como veremos en el apartado 4.3, este conjunto de datos ha sido fruto de un proceso evolutivo y de recopilar información de diferentes fuentes.

El resto de esta sección de análisis lo dividiremos en secciones como si se tratara de un *notebook* de *Jupyter*, que ha sido la interfaz utilizada para realizar el análisis.

4.1.1. Imports

Importamos las librerías y establecemos los parámetros que necesitaremos en la ejecución del *notebook*.

```
[1]: import sys
sys.path.append("..")
import warnings
warnings.filterwarnings('ignore')
import pandas as pd
import numpy as np
from datetime import datetime
import nltk
from nltk.tokenize.toktok import ToktokTokenizer
from nltk.corpus import stopwords
from mgmtfm import clean
from wordcloud import WordCloud
import matplotlib.pyplot as plt

pd.set_option('max_rows',9999)
pd.set_option('max_columns', 9999)
pd.set_option('display.max_colwidth', 500)

print("Notebook ejecutado el {}".format(datetime.now().
    strftime("%d-%m-%Y")))
```

Notebook ejecutado el 12-11-2019.

4.1.2. Carga de datos

El *core* de nuestros datos son las llamadas en sí. Actualmente disponemos de grabaciones de un 20 % de las llamadas de las que se transcriben un 20 % (lo que supone un 4 % del total), estas llamadas llegan a nuestra plataforma de análisis mediante un proceso *batch*. El objetivo es disponer en un futuro del 100 % de las transcripciones en tiempo real.

Para comenzar el análisis, cargamos el *dataset*.

```
[2]: verint_raw = pd.read_parquet('/data/datasets/input_data/verint_dataset.  
→parquet')
```

Describimos brevemente los campos que pueden ser de utilidad para nuestro proyecto:

- ***co_llamada_verint***: Código de la llamada en el sistema, nos servirá para identificar de manera única la transcripción.
- ***ucid***: Código único de la llamada.
- ***fx_evento***: Fecha del evento.
- ***it_llamada***: Timestamp del evento.
- ***datecreated***: Timestamp en el que se crea la transcripción.
- ***plaintext***: Transcripción de la llamada en texto plano.
- ***audio_start_time***: Hora de inicio del audio. Formato UTC.
- ***cd22***: Provincia desde la que se ha realizado la llamada.
- ***no_destino_pa***: Código IVR que categoriza la llamada en función de las opciones que elige el usuario al llamar.
- ***duration***: Duración de la llamada.

Aunque existen más campos en el conjunto de datos original, estos no son relevantes para nuestro caso de uso.

Utilizamos un subconjunto de estos campos, para realizar un análisis previo. Vemos en primer lugar el número total de llamadas disponibles:

```
[3]: verint = verint_raw[['co_llamada_verint','ucid','fx_evento','it_llamada',
                       'duration', 'plaintext', 'no_destino_pa']]
verint.plaintext = verint.plaintext.str.lower()
print(verint.count())
```

co_llamada_verint	509374
ucid	509374
fx_evento	509374
it_llamada	508524
duration	3457
plaintext	509374
no_destino_pa	508497
dtype:	int64

4.1.2.1. Monitorizaciones

Otra fuente de datos de la que disponemos, son las monitorizaciones de las llamadas. Se trata de un cuestionario realizado por un grupo de personas sobre llamadas escuchadas. A nosotros nos interesa la parte relacionada con el tipo de la llamada, que viene identificada en el cuestionario con los prefijos C y D .

```
[4]: monitorizaciones = pd.read_parquet('/data/datasets/input_data/
                                         ↪monitorizaciones.parquet')
print("Tenemos {:,} llamadas monitorizadas.".format(len(np.
                                         ↪unique(monitorizaciones["ucid"]))))
```

Tenemos 45,465 llamadas monitorizadas.

Al igual que hemos hecho anteriormente, vemos los datos más relevantes de este Dataset:

- ***co_llamada_verint***: Código de la llamada en el sistema, nos servirá para identificar de manera única la transcripción.
- ***ucid***: Código único de la llamada.
- ***it_llamada***: Timestamp de la llamada.
- ***duration***: Duración de la llamada.

- **unidad_negocio**: Unidad de negocio (no aplica en nuestro caso de uso).
- **name**: Nombre de lo que se mide en el cuestionario.
- **value**: Valor de lo que se mide en el cuestionario.

Nos quedamos con los datos de categorización de las llamadas (prefijos C y D). Y mostramos un ejemplo de registro.

```
[5]: monitor = monitorizaciones[(monitorizaciones.name.str.startswith('C') | 
                                ~monitorizaciones.name.str.startswith('D'))].drop_duplicates()

diccionario = {'C#1':'información', 'C#2':'contratar', 'D#1':'información', |
                ~'D#2':'consulta', 'D#3':'queja', 'D#4':'trámite'}

for k in diccionario.keys():
    monitor.loc[monitor.name.str.startswith(k), 'tipo'] = diccionario[k]

monitor.name = monitor.name.apply(lambda x: 'comercial' if x.
                                    ~startswith('C') else 'no_comercial')
monitor.dropna(inplace=True)
monitor.it_llamada = monitor.it_llamada.dt.date
monitor.columns = ['co_llamada_verint', 'ucid', 'fx_evento', 'duration', |
                    ~'unidad_negocio', 'motivo_llamada', 'subtipo', 'tipo']
monitor.head(1)
```

```
[5]:      co_llamada_verint          ucid  fx_evento duration \
35  9133922894230003051  00028029411538374830  2018-10-01     612.0

      unidad_negocio motivo_llamada           subtipo     tipo
35            GRP      no_comercial  D.3.4 Problemas Técnicos   queja
```

Combinamos los datos de las monitorizaciones con las llamadas:

```
[6]: monitored_calls = pd.merge(verint, monitor, on=['co_llamada_verint', 'ucid', |
                                ~'fx_evento', 'duration'], how='inner')
display(monitored_calls.head(1))
```

```
print("{} llamadas totales monitorizadas.".format(len(np.
    unique(monitored_calls["co_llamada_verint"]))))
```

```
co_llamada_verint          ucid  fx_evento it_llamada duration \
9136067963760004051  00026022771559825158  2019-06-06      NaT      507.0
                           plaintext \
buenas tardes soy ana de zaragoza en qué puedo ayudarle\nbuenas tardes..

no_destino_pa unidad_negocio motivo_llamada      subtipo      tipo
None           GRP     no_comercial  D.2.6 Factura  consulta
```

3,438 llamadas totales monitorizadas.

El hecho de que el número de llamadas transcritas sea tan bajo en relación con las llamadas totales puede ser un *handicap* a la hora de entrenar un modelo supervisado.

4.1.2.2. IVR

Como hemos visto los datos de monitorizaciones tienen el problema de que son escasos, es por eso que nos planteamos analizar los datos de *IVR*. Estos datos son proporcionados por el usuario al llamar al *call-center*. En primer lugar vemos el número de categorías diferentes que obtenemos del *dataset* de llamadas.

```
[7]: verint["no_destino_pa"].count()
```

```
[7]: 438
```

Como vemos el número de elementos es enorme y, sin agrupar, es difícil entender la categoría de las llamadas. Gracias a una jerarquía que nos han proporcionado desde el área de marketing (y que actualizan periódicamente), hemos podido obtener los datos agrupados.

Obtenemos la categoría más actual a través de una extracción de los datos originales y mostramos el formato.

```
[8]: ivr_hierarchy = pd.read_excel('/data/mgm/data/dwproyp0.
    dwen_1004_14_etiquetas_23102019.xlsx', index_col=0, header=0)
last_date = ivr_hierarchy["fx_carga"].max()
print("Nos quedamos con la fecha: {}".format(last_date))
```

```
ivr_hierarchy = ivr_hierarchy[ivr_hierarchy["fx_carga"] == last_date].
    drop(columns="fx_carga")
ivr_hierarchy.columns = ["tipo", "subtipo", "no_destino_pa"]
ivr_hierarchy.head(10)
```

Nos quedamos con la fecha: 20190508

	tipo	subtipo	no_destino_pa
3	Avería	Averias_Incidencias	MovistarTV_Conectividad_Futbol
6	No Reconocido	No reconocido	IVR-DTScontenidos
7	Avería	Averias_Incidencias	Internet_Velocidad_NoValidaNum
9	Resto	Deuda	IVR-DEUDAF
11	Comercial	Movistar TV	MovistarTV_Desambig
12	Avería	Averias_Incidencias	Silencio_AdminKO
13	Avería	Averias_Incidencias	Internet_NoCol_AdminKO_NoPulsa
14	Avería	Averias_Incidencias	Averia_Telefono_Movil_Otros_AdminKO
15	Resto	Idiomas	Idiomas_Arabe
17	No Reconocido	No reconocido	Facturacion_Desambiguar_NoCol

Por último, utilizamos esta jerarquía para obtener los tipos de las llamadas dependiendo de su *IVR*.

```
[9]: ivr_calls = pd.merge(verint, ivr_hierarchy, on=['no_destino_pa'], how='inner')
display(ivr_calls.head(1))
print("{} llamadas totales con IVR.".format(len(np.unique(ivr_calls["co_llamada_verint"]))))
```

co_llamada_verint	ucid	fx_evento	\
9136040272670004051	00028114201559548193	2019-06-03	
	it_llamada	duration	\
2019-06-03 09:52:04+00:00		NaN	
	plaintext	\	
buenos días soy marina habla dígame puedo ayudarle buenos días			

```

no_destino_pa tipo                      subtipo
MenuOpciones_NoCol Resto Gestiones Productos y Servicios - Resto

```

495,830 llamadas totales con IVR.

Vemos que en este caso tenemos categorizadas un número bastante considerable de llamadas que pueden ser muy útiles para entrenar nuestros modelos.

4.1.3. Análisis preliminar

En esta sección vamos a hacer un pequeño estudio sobre los datos anteriormente extraídos y sobre la distribución de las llamadas. Lo haremos tanto con los datos de las monitorizaciones como con los datos de *IVR*.

4.1.3.1. Monitorizaciones

En primer lugar mostramos la distribución de los datos de monitorizaciones, tanto en formato tabular como en un gráfico de tarta.

```
[10]: df_group = (monitored_calls.groupby(['tipo', 'motivo_llamada']).count() .
    ↪reset_index()[['tipo', "motivo_llamada", 'ucid']]).set_index(["tipo", ↪
    ↪"motivo_llamada"])

df_group.columns = [ "count"]
df_group.plot.pie(y='count', figsize=(7, 7))
display(df_group)
del(df_group)
```

	tipo	motivo_llamada	count
consulta	no_comercial	2204	
contratar	comercial	108	
información	comercial	204	
	no_comercial	41	
queja	no_comercial	476	
trámite	no_comercial	771	

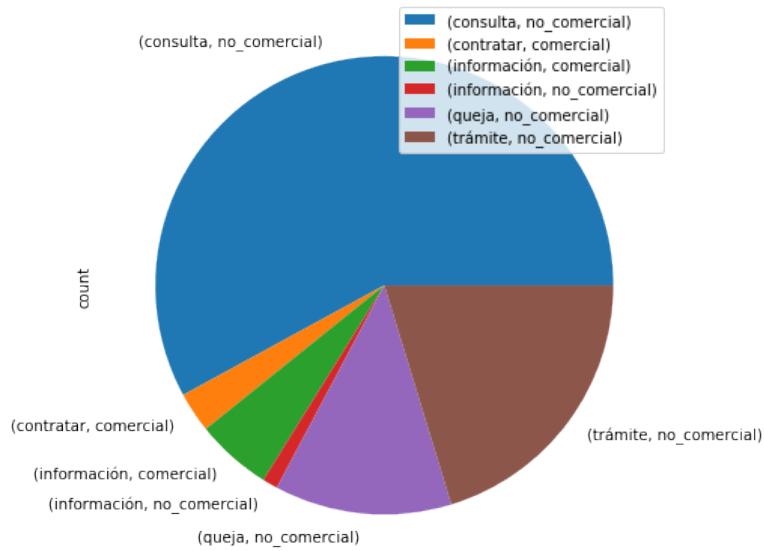


Figura 4.1: Distribución de llamadas por monitorizaciones

Observamos en la figura 4.1 las llamadas monitorizadas, además de ser un porcentaje muy bajo del total, no están balanceadas, teniendo más de la mitad de las llamadas concentradas en un mismo tipo.

El análisis lo continuaremos con las llamadas *tokenizadas* con el objetivo de no mostrar en la nube *stopwords* o palabras comunes. Para ahorrar procesamiento y código cargaremos directamente los datos *tokenizados* de un fichero. En la sección 4.2 podremos ver en qué consiste este proceso de *tokenizado*. La carga del fichero la hacemos usando el módulo *mgmtfm* que contiene clases y funciones para simplificarnos la ejecución del proyecto.

```
[11]: file_tokens = "/data/mgm/data/pandas/tokens_monitored_quit_commons_12112019.  
→pkl"  
clean_steps = clean.Clean()  
clean_steps.load_tokens(file_tokens)  
monitored_tokens = clean_steps.tokens
```

Found 20,422 unique tokens.

Para tener una idea inicial de las llamadas que se hacen, vamos a utilizar una nube de palabras que nos permita visualizar de manera sencilla los términos más usados:

```
[12]: def plot_wordcloud(tokens, title=None):
    text = " ".join(tokens["plaintext"].apply(" ".join))
    wordcloud = WordCloud(background_color="white").generate(text)
    fig = plt.figure( figsize=(16,10) )
    if (title):
        fig.suptitle(title)
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.axis("off")
    plt.show()

plot_wordcloud(monitored_tokens)
```



Figura 4.2: Nube de palabras a partir de transcripciones monitorizadas

En la figura 4.2, vemos como las palabras que nos encontramos parece que están claramente relacionadas con el servicio que se puede dar en el *call center* como “línea móvil”, “número teléfono”, “contrato”, “oferta”, “servicio”, etc.

A continuación vamos a representar la nube de palabras centrándonos en cada categoría de las monitorizaciones para apreciar las diferencias entre ellas.

```
[13]: tipos = clean_steps.distribucion_tipos.index.tolist()

for tipo in tipos:
    tokens_cat = monitored_tokens[monitored_tokens['tipo']==tipo]
    plot_wordcloud(tokens_cat, "Categoría: " + tipo.upper() )
```



Figura 4.3: Nube de palabras para la categoría consulta de las monitorizaciones

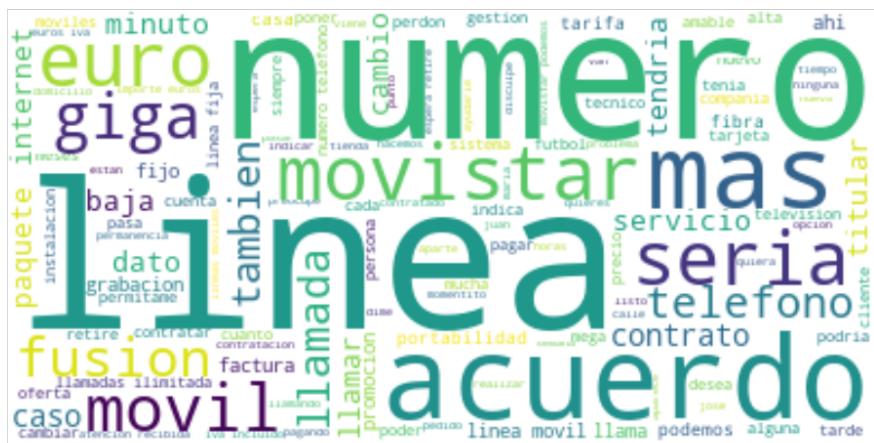


Figura 4.4: Nube de palabras para la categoría contratar de las monitorizaciones

En las figuras 4.3, 4.4, 4.5, 4.6 y 4.7 vemos que efectivamente hay variaciones en las nubes de palabras de las categorías, por destacar algunas:

- Vemos “línea” y “número” como palabras muy representativas a la hora de contratar.
- Vemos “reclamación” y “incidencia” como palabras presentes en la categoría queja.
- En trámite nos aparece, por ejemplo, el bigrama “dar baja”.



Figura 4.5: Nube de palabras para la categoría información de las monitorizaciones



Figura 4.6: Nube de palabras para la categoría queja de las monitorizaciones

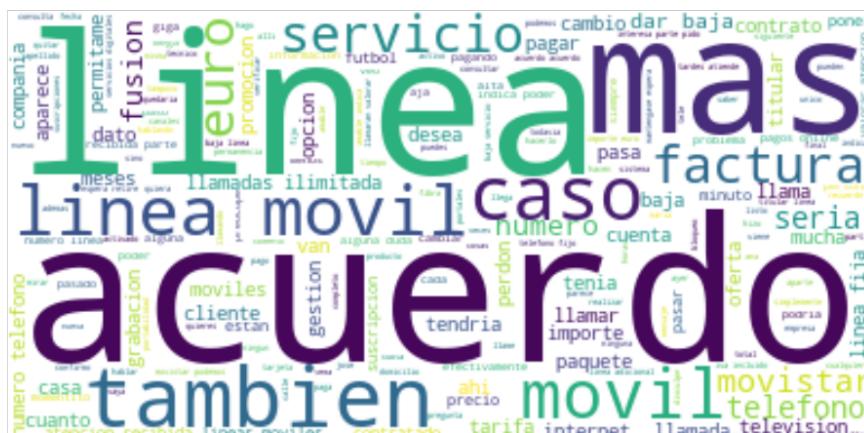


Figura 4.7: Nube de palabras para la categoría trámite de las monitorizaciones

4.1.3.2. IVR

Realizamos el mismo ejercicio para las categorizaciones de las llamadas mediante *IVR*. En primer lugar mostramos su distribución tanto en formato tabular como en gráfico de tartas:

```
[14]: df_group = (ivr_calls.groupby(['tipo']).count().reset_index()[['tipo','ucid']].set_index("tipo"))
df_group.columns = [ "count"]
df_group.plot.pie(y='count', figsize=(7, 7))
display(df_group)
del(df_group)
```

	count
tipo	
Avería	12350
Baja	24999
Comercial	204924
Factura	67892
No Reconocido	53198
Reclamación	23802
Resto	108746

De nuevo, como hicimos con las llamadas monitorizadas, cargamos los *tokens* de las llamadas que poseen *IVR*, con el objetivo de ahorrar tiempo de procesamiento.

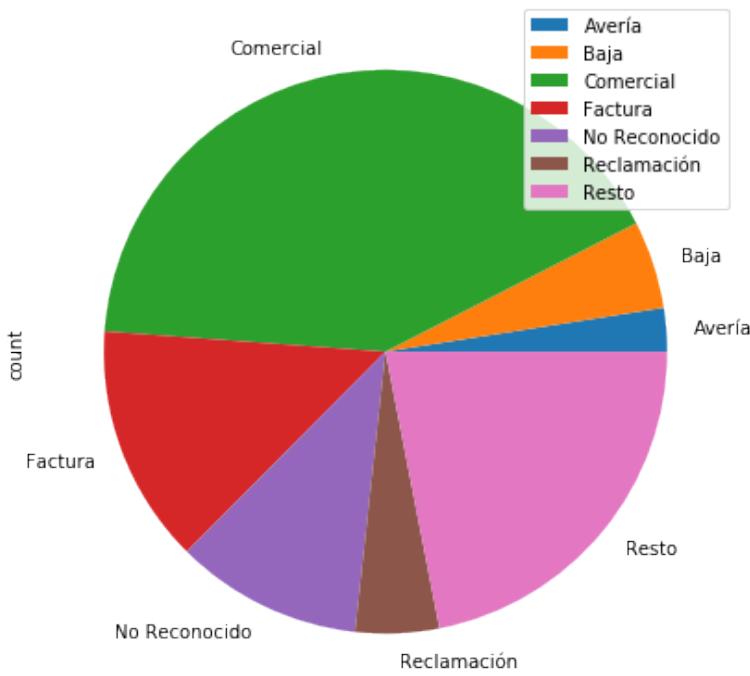
Una vez tenemos los *tokens*, mostramos la nube de palabras.

```
[15]: file_tokens = "/data/mgm/data/pandas/tokens_ivr_quit_commons_02112019.pkl"
clean_steps = clean.Clean()
clean_steps.load_tokens(file_tokens)
ivr_tokens = clean_steps.tokens
plot_wordcloud(ivr_tokens)
```

Found 47,990 unique tokens.

Vemos que aparecen más términos diferentes al aumentar la cantidad de la muestra, recordemos que el porcentaje de llamadas con *IVR* sobre el total era mucho mayor.

Para cada categoría de *IVR* mostramos la nube de palabras obtenida.

Figura 4.8: Distribución de llamadas por *IVR*Figura 4.9: Nube de palabras a partir de transcripciones con *IVR*

```
[16]: tipos = clean_steps.distribucion_tipos.index.tolist()
```

```
for tipo in tipos:
    tokens_cat = ivr_tokens[ivr_tokens['tipo']==tipo]
```




Figura 4.12: Nube de palabras de la categoría comercial de *IVR*



Figura 4.13: Nube de palabras de la categoría factura de *IVR*



Figura 4.14: Nube de palabras de la categoría “no reconocido” de *IVR*

Figura 4.15: Nube de palabras de la categoría reclamación de *IVR*Figura 4.16: Nube de palabras de la categoría resto de *IVR*

Así pues, tras el estudio de ambos tipos de clasificaciones, podemos concluir que la calidad que presentan las etiquetas de monitorizaciones es mayor y la clasificación es menos difusa; sin embargo el número de muestras se presenta *a priori* insuficiente para ser usado en modelos de *deep learning*.

Por estos motivos, en el capítulo 6 los modelos supervisados serán entrenados usando las etiquetas *IVR*, en un futuro, cuando los datos de monitorizaciones crezcan se realizará el mismo ejercicio con las etiquetas de monitorizaciones.

4.2. Preprocesado y Representación de transcripciones

Todos los modelos que se presentarán a lo largo de esta sección utilizan como entrada las transcripciones de las llamadas recibidas al *call-center*. En esta sección nos centraremos en los diferentes métodos que hemos escogido para representar estas llamadas.

El primer paso una vez recibimos la transcripción, y en cualquier proceso de minería de datos, es limpiar los datos. Para ello, como ya hemos visto en apartados anteriores, realizaremos un proceso de ***tokenizado*** que constará de los siguientes pasos:

- Eliminar caracteres especiales.
- Pasar a minúsculas.
- Eliminar números.
- Eliminar nombres propios.
- Eliminar palabras *stopwords*.
- Eliminar palabras comunes que no aporten información.

A partir de la lista de *tokens* obtenida por este proceso hemos creado un diccionario, en el que cada *token* tiene asignado un identificador numérico, obteniendo de este modo una **secuencia numérica** para cada llamada.

Los modelos que presentaremos en la siguiente sección necesitan que esta secuencia numérica sea de una longitud fija, por lo que usualmente limitaremos las secuencias a 866 elementos, ya que el 99 % de las llamadas poseen un tamaño menor. En el caso de que las llamadas posean una longitud menor usaremos *left padding*, es decir, rellenaremos con 0 por la izquierda la secuencia hasta los 866 elementos.

Las secuencias obtenidas ya podrían alimentar la mayoría de nuestros modelos, utilizando *one hot encoding*, sin embargo, hemos decidido añadir algunas transformaciones que nos permitan entender el contexto de las palabras.

4.2.1. Word2vec

Como ya vimos en el estado del arte, una de las representaciones que más popularidad ha tomado en los últimos tiempos es *word2vec* y esta será la entrada de la mayoría de nuestros modelos supervisados. Nuestro objetivo es huir de representaciones que traten a las palabras como elementos aislados y poder disponer de una representación que cuente con el contexto de la palabra.

Entre las dos representaciones de *word2vec* hemos optado por *SKIP-Gram* y hemos realizado pruebas entrenando el *embedding* con datos de *Wikipedia* y con nuestro corpus. Finalmente hemos optado por usar el modelo entrenado con nuestro corpus. Esto se debe a que tenemos un corpus lo suficientemente extenso y, de este modo, el modelo aprende las palabras según el contexto de las llamadas al *call-center* y no en un contexto general como puede ser *Wikipedia*.

En el caso de nuestros modelos, el *embedding*, es decir el paso a la representación de cada palabra como un vector de números reales, se realizará dentro del mismo modelo en una primera capa. Esta capa no será entrenada durante la fase de entrenamiento del modelo y consistirá en una matriz que contendrá por cada identificador de *token* un vector representando las palabras.

Usualmente utilizaremos una dimensionalidad de 150 en los vectores que representaran cada palabra.

4.2.2. Doc2vec

Otra opción para representar nuestras transcripciones que hemos abordado, tanto para modelos supervisados como no supervisados, es la de representar el documento completo como un único vector huyendo de la representación secuencial.

Una de las posibilidades de obtener un único vector consiste en hacer la media de los vectores *word2vec* obtenidos, pero en 2014 Le y Mikolov [18] propusieron un nuevo método denominado *doc2vec* que aplica un procedimiento muy similar a *word2vec*, pero a los documentos en lugar de las palabras. De hecho posee dos implementaciones “*Paragraph Vector - Distributed Memory*” (*PV-DM*) y “*Paragraph Vector - Distributed Bag of Words*” (*PV-DBOW*) que son análogas a las implementaciones *CBOW* y *SKIP-Gram* de *Word2Vec* que ya vimos en el estado del arte.

Nosotros hemos optado por entrenar un modelo *doc2vec* con nuestro corpus *tokenizado*, usando el modelo *PV-DBOW* (análogo a *SKIP-Gram*), para obtener un vector por transcripción con 500 dimensiones.

4.3. Evolución del conjunto de datos

Durante este capítulo hemos realizado un análisis del conjunto de datos más completo que teníamos hasta la fecha, no obstante, el proceso para conseguir y entender este conjunto de datos no ha sido una tarea trivial, sino que se ha tratado de un proceso iterativo en el que ha sido necesario involucrar a varias áreas y extraer información de diversas fuentes de datos de la compañía. El hecho de disponer de datos incompletos o de baja calidad nos han llevado a crear modelos poco eficientes que nos han situado otra vez en el punto de partida (como volveremos a ver en el siguiente capítulo).

Aunque el hecho de volver a la fase de recopilación y entendimiento de los datos es algo que ya preveíamos cuando presentamos el estándar ***CRISP-DM*** (apartado 1.4), vamos a describir en esta sección una breve muestra de los análisis iniciales para que pueda compararse con el análisis final y quede patente la evolución de los datos.

Al igual que en el apartado anterior expondremos los datos como si se tratara de un *notebook*, pero en este caso mostraremos únicamente las sentencias claves (ignorando por ejemplo los *imports*).

Todo el análisis que se muestra en este apartado fue realizado utilizando *PySpark* sobre un clúster de *Hadoop Hortonworks*.

4.3.1. Las llamadas

El primer paso, como es obvio, fue cargar los datos en un *dataframe* y comprobar la estructura del mismo:

```
[1]: domo_dataset = sqlContext.read.parquet("dataset/domo_dataset.parquet")
domo_dataset
```



```
[1]: DataFrame[co_llamada_verint: string, id_descarga:
           →string, nu_telefono_actuacion:string, it_llamada: timestamp, nu_llamada_ic:
           → string, co_grabacion: string, raw_verint:array<string>, __index_level_0__:
           → bigint]
```

De los campos listados únicamente era factible extraer información del texto de la llamada (“raw_verint”). El siguiente paso fue comprobar el número de llamadas que no contenían una transcripción nula. Además reparticionamos los datos y los dejamos en caché para realizar un análisis más eficiente:

```
[2]: raw_verint = domo_dataset.select("raw_verint").rdd.filter(lambda x:x
           →x["raw_verint"] is not None) \
           .map(lambda x: " ".join(map(lambda y: " ".join(y), x))) \
           .repartition(17)
raw_verint.cache()
print('Llamadas disponibles : {:.,}'.format(raw_verint.count()))
```

Llamadas disponibles : 185,109

A través de todas las llamadas obtuvimos una lista de palabras:

```
[3]: raw_list_word = raw_verint.map( lambda document: document.strip().
    lower() ) \
        .map( lambda document: re.split(" ", document))
```

Y eliminamos las *Stopwords* en español usando el paquete *nlk*.

```
[4]: StopWords = stopwords.words("spanish")

raw_no_stop = raw_verint.map( lambda document: document.strip().lower() ) \
    .map( lambda document: re.split(" ", document)) \
    .map( lambda word: [x for x in word if x not in StopWords])
```

Volvimos a unir las palabras de la lista (ahora sin las *stopwords*) para mostrar una nube de palabras más frecuentes

```
[5]: list_words = raw_no_stop.reduce(lambda a,b: list(set(a+b)))
wordcloud = WordCloud().generate(" ".join(list_words))
plt.figure(figsize = (20,20))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()
```



Figura 4.17: Primeros datos: *wordcloud* inicial

Como podemos observar en la nube de palabras obtenida (figura 4.17), el resultado no fue

el esperado. Las llamadas que aparecían no eran significantes y entre ellas existían muchas malas transcripciones. A partir de aquí intentamos abordar distintas aproximaciones que nos permitieran poder extraer algo de valor de los datos.

Un ejemplo de estas aproximaciones fue *taggear* las palabras en función de su categoría gramatical para quedarnos solo con una lista de candidatos. Para hacerlo de un modo eficiente de manera distribuida, en primer lugar creamos un diccionario con las categorías y la raíz de las palabras:

```
[6]: tagger = treetaggerwrapper.TreeTagger(TAGLANG='es', TAGPARFILE="/tmp/tree/spanish.par", TAGDIR="/tmp/tree/tree-tagger-3.2.1/")
vocabulary = raw_no_stop.filter(lambda x: len(x)>0) \
    .flatMap(lambda document: document) \
    .map(lambda word: (word, 1)) \
    .reduceByKey( lambda x,y: x + y) \
    .map(lambda tuple: tuple[0])
vocabulary.cache()
vocabulary.count()
vocabulary_tags = list(map(lambda el: (el[0], (el[1], el[2])) , \
    map(lambda y: y.split("\t"),list(tagger.tag_text(" ".join(vocabulary. \
    collect())) ))))
tags_dict = sc.broadcast({key: value for (key, value) in \
    vocabulary_tags})
```

Una vez que habíamos realizado el filtrado nos quedamos únicamente con la raíz de las palabras que fueran verbos o nombres.

```
[7]: def get_stem_of_candidates(x):
    good = [u'VInf', u'NC']
    candidates = list(filter(lambda word: word in tags_dict.value \
        and tags_dict.value[word][0] in good ,x))
    stem = list(map(lambda word: tags_dict.value[word][1], \
        candidates))
    return stem

stemmed_candidates = raw_no_stop.map(get_stem_of_candidates)
stemmed_candidates.cache()
```


4.3.2. Las etiquetas

Tras ver la pobre calidad de los datos iniciales, intentamos encontrar algún dato que nos sirviera para etiquetar las llamadas pensando que podrían ser de utilidad para un modelo supervisado.

Una posibilidad era revisar y etiquetar llamadas manualmente, pero dada la cantidad de datos que podían ser necesarios para entrenar un modelo supervisado, convertía esta posibilidad en inviable.

A continuación vamos a analizar los primeros datos que obtuvimos de etiquetas. En primer lugar cargamos las etiquetas y los datos originales:

```
[1]: domo_dataset = sqlContext.read.parquet("dataset/domo_dataset.parquet")
label_dataset = sqlContext.read.csv("dataset/20190802_meta_todo.csv", u
↪header=True)
print(label_dataset)
```

```
DataFrame[co_llamada_verint: string, nu_telefono_actuacion: string, u
↪it_llamada:
string, nu_llamada_ic: string, co_grabacion: string, no_destino_pa: string,
in_poc1: string, satisfaccion: string]
```

La etiqueta se corresponde con el dato *no_destino_pa* del *dataset*. Vemos las diferentes etiquetas que tenemos disponibles:

```
[2]: label_dataset.createOrReplaceTempView("label_tmp")
labels = sqlContext.sql("SELECT DISTINCT no_destino_pa FROM label_tmp")
labels.count()
```

[2]: 438

El primer problema que nos encontramos es el gran número de etiquetas que aparecen sin tener una jerarquía clara, además aunque intentamos agruparlas, su clasificación (sin un conocimiento previo de los datos) se antojaba una tarea bastante compleja.

No obstante, mirando el número de llamadas con etiqueta:

```
[3]: domo_dataset.createOrReplaceTempView("domo_tmp")
label_calls = sqlContext.sql("""SELECT a.raw_verint, b.no_destino_pa
FROM domo_tmp a JOIN label_tmp b
```

```
    ON a.co_llamada_verint = b.co_llamada_verint  
    WHERE a.raw_verint IS NOT NULL""")  
  
print('Llamadas etiquetadas disponibles : {:.}{}'.format( label_calls.count()))
```

Llamadas etiquetadas disponibles : 170,844

Nos dimos cuenta que teníamos un gran número de llamadas etiquetadas, convirtiéndose en un dato muy interesante para una clasificación supervisada. Como hemos visto en la sección [4.1.3](#) de este mismo capítulo, este dato agrupado ha sido uno de los usados en el *dataset* definitivo.

Capítulo 5

Aprendizaje No Supervisado

Capítulo 6

Aprendizaje supervisado

En este capítulo, realizaremos el análisis de diferentes modelos supervisados que hemos construido con el fin de clasificar las llamadas en función de las etiquetas que hemos analizado en el capítulo 4.

Durante este capítulo describiremos las diferentes arquitecturas que proponemos para construir nuestros modelos (sección 6.1), posteriormente en la sección 6.2 evaluaremos la precisión de los modelos construidos con cada una de las arquitecturas propuestas y por último, en la sección 6.3, propondremos el modelo que pasaremos a producción y que hará de enlace con la siguiente parte del documento.

La creación y entrenamiento de estos modelos se ha realizado en Python 3.6, usando la librería Keras con Tensorflow como *backend* sobre un equipo con las siguientes características:

- Modelo: HP Apollo 6500 – X270d Gen 10
- Memoria RAM: 768 Gb
- Procesadores: 2 x Xeon-G 6142 (16 cores , 2,60Ghz)
- Disco duro: H.D: 16 x 7.68TB SAS RI SFF SC DS SSD
- Tarjetas Graficas: 8 x HPE NVIDIA Tesla V100-32GB SXM2

6.1. Arquitecturas

A lo largo de esta sección vamos a describir las diferentes arquitecturas que hemos utilizado para construir los modelos que posteriormente hemos entrenado y evaluado con los datos descritos en el capítulo anterior.

6.1.1. Red neuronal convolucional

La primera aproximación a la hora de crear un modelo que nos permitiera clasificar llamadas, ha sido recurrir a las redes neuronales recurrentes (que ya introdujimos en el estado del arte, en el apartado 2.2.4.1).

El objetivo principal que perseguimos al aplicar esta estructura era poder capturar la estructura local de las llamadas, de una forma similar al modo en el que se hace en el tratamiento de imágenes. Aplicar *kernels* de diferentes tamaños a la imagen inicial podía servirnos para obtener de manera automática especies de N-Gramas que capten patrones en las transcripciones de las llamadas.

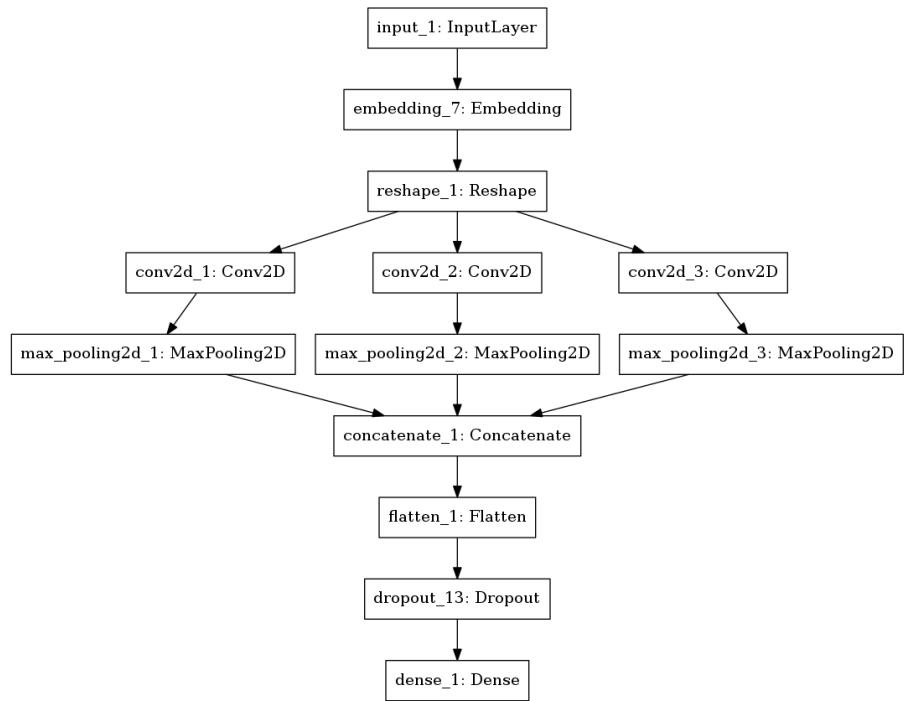


Figura 6.1: Arquitectura CNN

En la figura 6.1 podemos ver un ejemplo de un modelo de arquitectura de este tipo. Los modelos con la arquitectura que proponemos estarán formados por los siguientes tipos de capas:

- **Entrada:** La entrada del modelo consistirá en una secuencia de identificadores de *tokens*.
- **Embeddings:** Una vez tengamos la entrada se le aplicará un *embedding*, en nuestro caso hemos realizado la mayoría de los modelos con *Skip-gram*. Este modelo estará pre-entrenado y simplemente traducirá cada *ID* de la secuencia de palabras en el vector de números reales a partir de la matriz entrenada con anterioridad.

- **Capas convolucionales:** Este será el núcleo de este tipo de modelos y consistirán en un número N de capas en las que se aplicarán distintos tipos de *kernels* al conjunto de entrada.
- **Capas Pooling:** Se aplicará *max-pooling* a la salida de cada una de las capas convolucionales. El objetivo es quedarnos con las características más relevantes de cada “cuadrante”.
- **Capa de Dropout:** Tras concatenar y aplanar la salida introduciremos una capa de *dropout* previa a la capa totalmente conectada. El objetivo de esta capa será evitar el sobreentrenamiento.
- **Capa totalmente conectada:** Por último, una capa totalmente conectada nos dará la salida del modelo. En función de si nuestro objetivo es clasificación multiclas o binaria utilizaremos como salida la función *softmax* o *sigmoid* respectivamente.

Esta arquitectura podrá utilizarse para crear diferentes parámetros, en nuestro caso hemos jugado con los parámetros:

- Número de capas Convolucionales
- Tamaño de los *kernels* que se aplicarán en cada capa.
- Número de *kernels* que tendrá cada capa.
- Valor de *dropout*.
- Valor de regularización nivel dos. Este parámetro intentará reducir el sobreentrenamiento de la red, reduciendo los pesos altos.

El código de creación y entrenamiento de un modelo de este tipo, usando el módulo *mgmtfm* sería:

```
models_steps = models.Models(n_classes, input_shape, embedding_dim,
                             vocabulary_size, embedding_matrix)
models_steps.model_cnn_1(n_classes, filter_sizes=filter_sizes, drop=dropout,
                        num_filters=num_filters)
models_steps.compile_and_train(X_train, y_train, batch_size=batch,
                               epochs=epochs, lr=lr, decay=decay,
                               validation_data=(X_test, y_test),
                               loss=loss_function)
```

6.1.2. Red neuronal recurrente

En este apartado vamos a crear una arquitectura basándonos en otro de los modelos que vimos al analizar el estado del arte, las redes neuronales recurrentes (apartado 2.2.4.1). De nuevo usaremos los modelos entrenados con este tipo de arquitectura para clasificar llamadas.

Las redes recurrentes a simple vista pueden parecer la solución más intuitiva a la hora de abordar un problema de análisis/clasificación de textos con *deep learning*. En este caso iremos pasando toda la secuencia de *tokens*, proveniente de la transcripción de nuestras llamadas, y, en cada paso (o *token*), la red irá olvidando características poco relevantes del mensaje y quedándose con las características importantes para su clasificación. Como podemos imaginar el entrenamiento de estas redes es un proceso más complejo y costoso computacionalmente que el de las redes convolucionales anteriormente vistas.

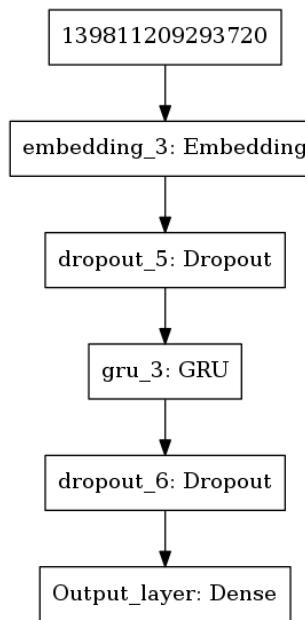


Figura 6.2: Arquitectura RNN

En la figura 6.2 observamos un modelo de ejemplo con la arquitectura descrita. Los modelos con la arquitectura recurrente propuesta estarán formados por los siguientes tipos de capas:

- **Entrada:** La entrada del modelo consistirá en una secuencia de identificadores de *tokens*.
- **Embeddings:** Una vez tengamos la entrada, se le aplicará un *embedding*, en nuestro caso hemos realizado la mayoría de los modelos con SKIP-Gram. Este modelo estará pre-entrenado y simplemente traducirá cada *ID* de la secuencia de palabras en el vector de números reales a partir de la matriz entrenada con anterioridad.

- **Capa de Dropout:** Posteriormente tendremos una capa de *dropout* previa a la red recurrente. El objetivo de esta capa será evitar el sobreentrenamiento. Añadiremos otra capa de *dropout* con el mismo objetivo previa a la capa totalmente conectada.
- **Capa recurrente:** En este punto introduciremos una capa formada por un número N de celdas recurrentes.
- **Capa totalmente conectada:** Por último, una capa totalmente conectada nos dará la salida del modelo. En función de si nuestro objetivo es clasificación multiclase o binaria, utilizaremos como salida la función *softmax* o *sigmoid* respectivamente.

Esta arquitectura podrá utilizarse para crear diferentes modelos, en nuestro caso hemos jugado con los parámetros:

- Tipo de celda: LSTM o GRU.
- Número de celdas en la capa recurrente.
- Valor de *dropout*.

El código de creación y entrenamiento de un modelo de este tipo, usando el módulo *mgmtfm* sería:

```
models_steps = models.Models(n_classes, input_shape, embedding_dim,
                             vocabulary_size, embedding_matrix)
models_steps.model_rnn_1(memory_units, cell_type, dropout)
models_steps.compile_and_train(X_train, y_train,
                               batch_size=batch, epochs=epochs,
                               lr=lr, decay=decay,
                               validation_data=(X_test, y_test),
                               loss=loss_function)
```

6.1.3. Red neuronal “híbrida” convolucional/recurrente

Vistas las características de ambas redes pensamos en combinar ambas para obtener las bondades de cada una. Por un lado una capa convolucional que se encargara de recoger las características locales de la secuencia (extraer los “N-Gramas”) y posteriormente, una capa recurrente que utilice la información enriquecida por la capa convolucional para realizar la clasificación.

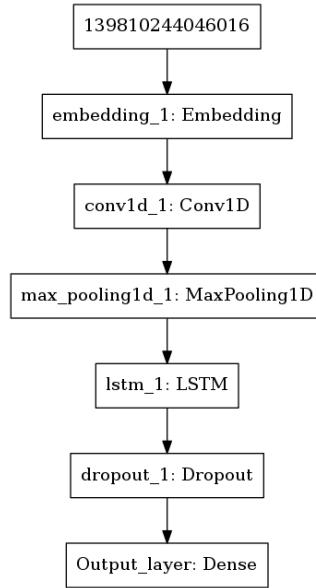


Figura 6.3: Arquitectura “Híbrida” CNN/RNN

En la figura 6.3 podemos ver un ejemplo de un modelo de arquitectura de este tipo. Los modelos con la arquitectura que proponemos estarán formados por los siguientes tipos de capas:

- **Entrada:** La entrada del modelo consistirá en una secuencia de identificadores de *tokens*.
- **Embeddings:** Una vez tengamos la entrada se le aplicará un *embedding*, en nuestro caso hemos realizado la mayoría de los modelos con SKIP-Gram. Este modelo estará pre-entrenado y simplemente traducirá cada *ID* de la secuencia de palabras en el vector de números reales a partir de la matriz entrenada con anterioridad.
- **Capa convolucional:** En esta capa se aplicará un número determinado de *kernels* con el tamaño especificado.
- **Capa de Pooling:** Se aplicará *max-pooling* a la salida de la capa convolucional. El objetivo es quedarnos con las características más relevantes de cada “cuadrante”.
- **Capa de Dropout:** Tras la capa recurrente introduciremos una capa de *dropout* previa a la capa totalmente conectada. El objetivo de esta capa será evitar el sobreentrenamiento.
- **Capa recurrente:** En este punto introduciremos una capa formada por un número *N* de celdas recurrentes.
- **Capa totalmente conectada:** Por último, una capa totalmente conectada nos dará la salida del modelo. En función de si nuestro objetivo es clasificación multiclas o binaria utilizaremos como salida la función *softmax* o *sigmoid* respectivamente.

Esta arquitectura podrá utilizarse para crear diferentes modelos, en nuestro caso hemos jugado con los parámetros:

- Tipo de celda de la capa recurrente: LSTM o GRU.
- Número de celdas en la capa recurrente.
- Valor de *dropout*.
- Tamaño de los *kernels* que se aplicarán en la capa convolucional.
- Número de *kernels* que tendrá la capa convolucional.
- *padding* (str): Tipo de padding a realizar al aplicar la capa convolucional..

El código de creación y entrenamiento de un modelo de este tipo, usando el módulo *mgmtfm* sería:

```
models_steps = models.Models(n_classes,input_shape,embedding_dim,
                             vocabulary_size,embedding_matrix)
models_steps.model_rnn_2(memory_units, cell_type, dropout,
                        filters,filter_size,pooling_size, padding)
models_steps.compile_and_train(X_train, y_train,batch_size=batch,
                               epochs=epochs, lr=lr,decay=decay,
                               validation_data=(X_test, y_test),
                               loss=loss_function)
```

6.1.4. MLP

Los modelos que hemos visto hasta hora tenían siempre como entrada una secuencia de palabras, sin embargo con la representación de palabras *Doc2Vec* es posible obtener un vector con números reales para cada transcripción que contenga información semántica del mismo. Esto nos permite poder abordar otro tipo de modelos más simples.

La arquitectura que proponemos, y que podemos ver en la figura 6.4, consiste en utilizar únicamente un número N de capas totalmente conectadas y de *dropout* (para evitar el sobre-entrenamiento).

Los parámetros que podremos usar en este caso serían:

- Número de capas totalmente conectadas.

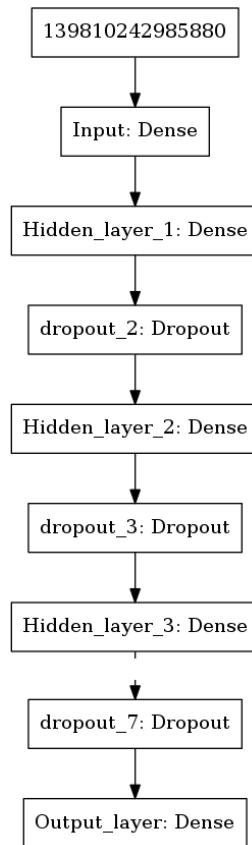


Figura 6.4: Arquitectura con capas totalmente conectadas

- Número de neuronas en cada una de las capas. El número de neuronas de la última capa corresponderá con la salida y , en función de si nuestro objetivo es clasificación multiclas o binaria, utilizaremos como salida la función *softmax* o *sigmoid* respectivamente.
- Valor de *dropout*.

El código de creación y entrenamiento de un modelo de este tipo, usando el módulo *mgmtfm* sería:

```

models_steps = models.Models(n_classes,input_shape,embedding_dim,
                             vocabulary_size,embedding_matrix)
models_steps.model_dense_1(layers_list,drop=dropout)
models_steps.compile_and_train(X_train, y_train,batch_size=batch,
                               epochs=epochs, lr=lr,decay=decay,
                               validation_data=(X_test, y_test),
                               loss=loss_function)
  
```

6.1.5. Combinación de modelos

Otra opción que hemos abordado es la de combinar la salida de los modelos y que estos formen la entrada de un nuevo modelo de capas totalmente conectadas.

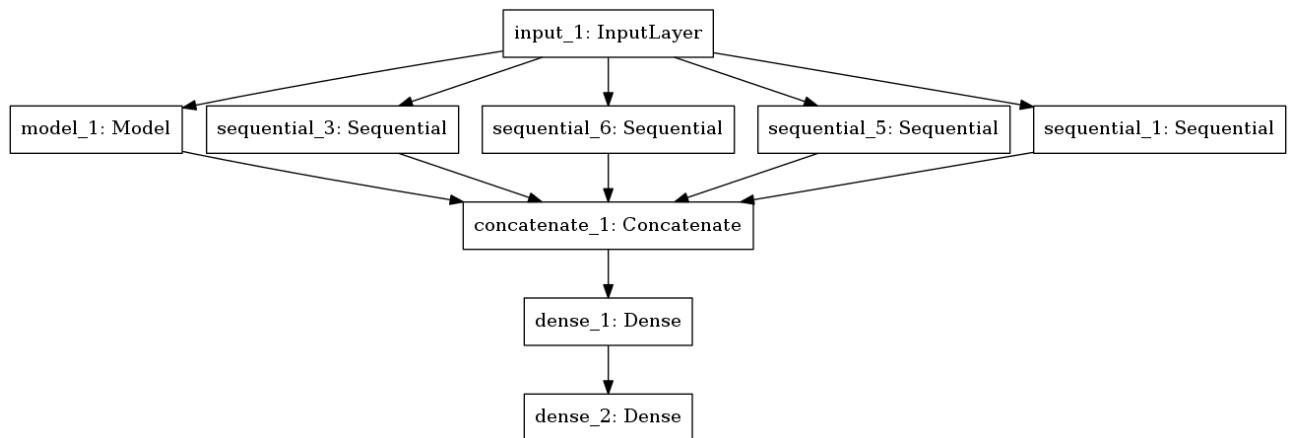


Figura 6.5: Arquitectura con capas totalmente conectadas

En la figura 6.5, vemos un ejemplo de este tipo de modelos. En este caso el input alimenta a los diferentes modelos (5 en este caso), que pueden estar construidos tanto con el modelo *Model* de keras como con el modelo *Sequential*.

La salida de los diferentes modelos se concatena y, antes de la salida, se añade una capa oculta totalmente conectada con función de activación *relu*. Normalmente este tipo de modelos no vamos a usarlos para tareas de clasificación binaria, por lo que la función de activación de la capa de salida será *sigmoid*.

Hay que destacar que los sub-modelos que forman parte de este modelo están preentrenados y no volverán a entrenarse en el proceso de aprendizaje.

En este caso no tenemos implementado el modelo en el módulo *mgmtfm*, pero podemos basarnos en una lista de modelos creados en *mgmtfm* “*prev_models*” para crearlo:

```

from keras.layers import Input, Dense, concatenate
from keras.models import Model

for pm in prev_models:
    for layer in pm._model.layers:
        layer.trainable = False
  
```

```
models_out = [pm._model(in_model) for pm in prev_models]
combined = concatenate(models_out)
hidden = Dense(n_hidden, activation='relu')(combined)
out_model = Dense(n_classes, activation='softmax')(hidden)
model = Model(in_model , out_model)
```

Y posteriormente entrenarlo:

```
from keras.optimizers import Adam
from keras.preprocessing.sequence import pad_sequences

adam = Adam(lr, decay=decay)

model.compile(loss=loss,
              optimizer=adam,
              metrics=metrics )

model.fit(X_train, y_train, batch_size=batch_size, epochs=epochs,
           validation_data=(X_test , y_test))
```

6.2. Evaluación y optimización

Una vez hemos creado los modelos acordes a las diferentes arquitecturas comentadas en el apartado anterior, surge la necesidad de encontrar los hiperparámetros que en cada arquitectura nos lleven a la mayor precisión de nuestro modelo.

Encontrar los mejores parámetros es una tarea crucial a la hora de probar un modelo y, aunque existen diferentes métodos para intentar conseguir los mejores resultados al entrenar nuestro modelo, como puede ser definir un *grid* de parámetros o probar de manera aleatoria; nosotros hemos probado por un mecanismo más automático utilizando **Optuna**.

Optuna es un *framework* de optimización de modelos que utiliza métodos Bayesianos para la optimización de modelos. Entre las ventajas de Optuna tenemos:

- Utilización de una BBDD (en nuestro caso PostgreSQL) que nos permite continuar con los entrenamientos cuando queramos y lanzar varios entrenamientos en paralelo o en cualquier momento.

- Posibilidad de definir la duración de las pruebas que queremos lanzar. Lo que nos permite, por ejemplo, lanzar pruebas por la noche cuando el clúster posee más recursos.
- Uso de la poda para descartar modelos con mala calidad en fases tempranas, ahorrando tiempo de cómputo.

Desde que empezamos a usar *Optuna* en la plataforma formada por *GPUs TeslaV100 – SXM232GB*, hemos dedicado unas 340 horas a la optimización de modelos, lo que supone algo más de 14 días. Hay que tener en cuenta que muchas de las pruebas no prometedoras *Optuna* las ha cancelado automáticamente, por lo que en condiciones normales hubiéramos necesitado bastante más tiempo. También hay momentos en los que se ha paralelizado el entrenamiento entre varias *GPUs*.

A lo largo del apartado veremos las distintas pruebas realizadas agrupándolas por tipo de arquitectura. Los nombres de los modelos nos darán información sobre el tipo del modelo. Algunos datos que tenemos que tener en cuenta son:

- ***CNN/RNN/RNN2/DENSE***: Nos indica el tipo del modelo y estará al comienzo del nombre.
- ***IVR/MONI***: Nos indica si el modelo ha sido entrenado y validado con los tipos de IVR o de monitorizaciones. De momento solo usamos *IVR* debido al porcentaje tan bajo de llamadas que poseen monitorizaciones.
- ***W2V150QC***: Nos indica que utiliza un *Word2Vec* con una dimensión de 150 y quitando las palabras comunes en los *tokens*.
- ***BIN/MULTI***: Nos indica que se trata de un modelo binario o multiclas. Si es binario al final del nombre nos encontramos la clase, si es multiclas y no clasifica todas las clases nos encontramos las iniciales de las clases.
- ***QX***: Nos indica las clases que se eliminan. Por ejemplo *QR_QN* elimina las clases *resto* y *no reconocido* antes de entrenar. La *no reconocido* la eliminamos siempre ya que puede contener elementos del resto de clases.
- ***866***: Indica que en estos modelos se ha limitado las secuencias a 866 tokens, que engloban más del 99 % de las llamadas.

Una última consideración a tener en cuenta es que, solo los modelos más prometedores se han pasado a la fase de optimización con *Optuna*. Otros tantos, como pueden ser los modelos con datos de monitorizaciones, se han descartado desde un primer momento observando que los primeros resultados no eran prometedores.

6.2.1. Modelos *CNN* binarios

Aunque se han hecho pruebas con modelos multi-clase, cuando se ha empezado a utilizar *Optuna* únicamente se han entrenado modelos binarios que presentaban mucho mejores resultados. Por ello en esta primera sesión únicamente presentaremos datos de modelos convolucionales.

Para cada modelo mostraremos los siguientes campos:

- ***study***: Nombre del modelo del que se está realizando el estudio.
- ***accuracy***: Mejor precisión sobre el conjunto de test.
- ***params***: Parámetros con los que se ha obtenido la mejor precisión.
- ***ntrials***: Número de pruebas realizadas del estudio. Cuenta las pruebas podadas, pero no las fallidas.
- ***total_time***: Tiempo total invertido en el entrenamiento del modelo.
- ***best_trial***: Identificador de la mejor prueba realizada. Nos será de utilidad ya que *Optuna* guarda los pesos automáticamente del mejor modelo.

Los modelos se han entrenado con datos balanceados, teniendo el mismo número de muestras de la clase binaria y del resto de clases.

En la figura 6.6 podemos ver los resultados obtenidos. Los resultados parecen bastante buenos para la mayoría de clases que van desde el 72 % al 86 % de precisión. Cuando eliminamos las clases *resto* y *no reconocido* vemos que los datos mejoran. Esto se debe a que, probablemente, existan datos en estas clases que hagan referencia a elementos de otras clases.

6.2.2. Modelos *RNN* Binarios

A continuación presentamos los resultados de los modelos formados por redes recurrentes que realizan clasificación binaria en el mismo formato que el apartado anterior. Hemos agrupado en los modelos recurrentes también aquellos que tienen una capa convolucional junto a la capa recurrente.

En general, observando la figura 6.7, vemos una ligera mejora con respecto a los modelos formados por redes convolucionales. Vemos por ejemplo como en el caso de la clase *Factura* hemos mejorado el modelo hasta casi el 86 % con una red recurrente y el caso de la clase *Comercial* con una red recurrente con una capa convolucional hemos llegado al 79.6 %.

También podemos observar que no existe una diferencia apreciable entre los modelos que incluyen una capa convolucional y los que no la tienen. Otro punto a destacar de estas pruebas

6.2. Evaluación y optimización

81

	study	accuracy	params	ntrials	total_time	best_trial
19	CNN_IVR_BIN_W2V150QC_QN_BAJA	0.862927	((epochs, 14.0), (kernel_size_conv_1, 2.0), (lr, 0.0466210906487132), (decay, 0.0973688009931105), (batch_size, 190.0), (num_filters, 141.0), (dropout, 0.452728938708042), (kernel_size_conv_3, 5.0), (kernel_size_conv_2, 6.0))	28	0 days 02:00:11.240753	7163
8	CNN_866_IVR_BIN_W2V150QC_QN_BAJA	0.857002	((epochs, 15.0), (kernel_size_conv_3, 7.0), (lr, 0.0783251859934289), (num_filters, 116.0), (decay, 0.08655927910488), (kernel_size_conv_1, 4.0), (kernel_size_conv_2, 4.0), (batch_size, 179.0), (dropout, 0.08564310007749515))	87	0 days 03:01:23.436193	17765
12	CNN_IVR_BIN_W2V150QC_BAJA	0.844259	((kernel_size_conv_1, 3.0), (kernel_size_conv_3, 7.0), (lr, 0.0706824812409031), (batch_size, 130.0), (num_filters, 138.0), (decay, 0.0817036289265317), (dropout, 0.465363078907312), (epochs, 13.0), (kernel_size_conv_2, 6.0))	157	1 days 03:25:09.192230	428
9	CNN_866_IVR_BIN_W2V150QC_QN_FACTURA	0.837989	((epochs, 15.0), (kernel_size_conv_3, 7.0), (kernel_size_conv_2, 3.0), (num_filters, 180.0), (decay, 0.0677721044360595), (kernel_size_conv_1, 4.0), (batch_size, 255.0), (dropout, 0.052583716431868), (lr, 0.0308687095584748))	67	0 days 02:58:44.532765	17630
20	CNN_IVR_BIN_W2V150QC_QN_FACTURA	0.834412	((kernel_size_conv_1, 3.0), (batch_size, 283.0), (kernel_size_conv_3, 7.0), (lr, 0.0689525683093965), (decay, 0.0837186209217048), (num_filters, 140.0), (kernel_size_conv_2, 5.0), (epochs, 12.0), (dropout, 0.141975383278602))	11	0 days 02:03:49.792338	7180
7	CNN_IVR_BIN_W2V150QC_FACTURA	0.833933	((kernel_size_conv_3, 6.0), (decay, 0.0621436033315488), (kernel_size_conv_1, 2.0), (lr, 0.0754551969975402), (dropout, 0.060162085990921), (batch_size, 476.0), (epochs, 5.0), (kernel_size_conv_2, 5.0), (num_filters, 99.0))	18	0 days 02:01:35.712605	547
10	CNN_866_IVR_BIN_W2V150QC_QN_AVERÍA	0.805820	((decay, 0.05700552618666), (kernel_size_conv_3, 7.0), (num_filters, 125.0), (lr, 0.0663904473012762), (batch_size, 328.0), (kernel_size_conv_1, 4.0), (kernel_size_conv_2, 4.0), (epochs, 17.0), (dropout, 0.286388351609826))	165	0 days 03:00:18.650700	17588
17	CNN_IVR_BIN_W2V150QC_QN_AVERÍA	0.797523	((epochs, 14.0), (kernel_size_conv_1, 3.0), (kernel_size_conv_3, 6.0), (lr, 0.0548823090135392), (kernel_size_conv_2, 3.0), (num_filters, 94.0), (decay, 0.0988380792860522), (dropout, 0.329528601396929), (batch_size, 172.0))	69	0 days 02:01:26.599629	7271
39	CNN_866_IVR_BIN_W2V150QC_QN_COMERCIAL	0.775623	((lr, 0.0755546150840543), (batch_size, 136.0), (kernel_size_conv_1, 5.0), (decay, 0.0868261840964118), (epochs, 9.0), (kernel_size_conv_2, 5.0), (kernel_size_conv_3, 4.0), (dropout, 0.122009867836421), (num_filters, 123.0))	24	0 days 03:11:02.989281	19246
21	CNN_IVR_BIN_W2V150QC_QN_COMERCIAL	0.775587	((epochs, 14.0), (kernel_size_conv_3, 7.0), (lr, 0.0651886911467839), (kernel_size_conv_2, 3.0), (num_filters, 94.0), (decay, 0.0434100642097045), (kernel_size_conv_1, 4.0), (batch_size, 486.0), (dropout, 0.378186034581814))	4	0 days 02:16:05.047722	7200
18	CNN_IVR_BIN_W2V150QC_QN_RECLAMACIÓN	0.744399	((epochs, 14.0), (decay, 0.0450894277841205), (kernel_size_conv_3, 6.0), (batch_size, 232.0), (kernel_size_conv_1, 5.0), (lr, 0.024960142439257), (kernel_size_conv_2, 4.0), (dropout, 0.206429106576552), (num_filters, 80.0))	119	0 days 02:02:40.433888	7390
11	CNN_866_IVR_BIN_W2V150QC_QN_RECLAMACIÓN	0.743755	((kernel_size_conv_1, 3.0), (kernel_size_conv_3, 7.0), (batch_size, 276.0), (num_filters, 182.0), (decay, 0.0832651044273547), (epochs, 16.0), (lr, 0.0191576591916923), (dropout, 0.295038818783405), (kernel_size_conv_2, 6.0))	81	0 days 03:02:28.056508	17691

Figura 6.6: Optimización de modelos binarios con redes convolucionales

es que el tiempo de ejecución medio, si observamos el tiempo total y el número de pruebas, es mucho mayor que en los modelos convolucionales.

En cuanto al tipo de celdas vemos que aunque aparecen ambos (valor 0 para *LSTM* y valor 1 para *GRU*) los mejores resultados los hemos obtenido con *GRU* siendo además los modelos con este tipo de celdas menos costosos para el entrenamiento.

	study	accuracy	params	ntrials	total_time	best_trial
24	RNN_IVR_BIN_W2V150QC_QR_QN_BAJA	0.873912	<code>((memory_units, 51.0), (batch_size, 430.0), (lr, 0.0530475544836823), (cell, 1.0), (decay, 0.0675610167878156), (epochs, 4.0), (dropout, 0.457826302587858))</code>	48	07:02:19.799293	10927
37	RNN2_IVR_BIN_W2V150QC_QR_QN_BAJA	0.865334	<code>((filters, 59.0), (batch_size, 744.0), (memory_units, 389.0), (cell, 0.0), (decay, 0.0767493546430331), (padding, 1.0), (kernel_size, 3.0), (lr, 0.0392491376088012), (pool_size, 4.0), (dropout, 0.235471010425368), (epochs, 13.0))</code>	41	05:04:49.337967	11505
23	RNN_IVR_BIN_W2V150QC_QR_QN_FACTURA	0.858087	<code>((decay, 0.0563279925787701), (memory_units, 274.0), (cell, 1.0), (lr, 0.0126883161110917), (batch_size, 581.0), (epochs, 12.0), (dropout, 0.43667104698883))</code>	5	09:13:43.006154	11063
1	RNN_866_IVR_BIN_W2V150QC_QN_FACTURA	0.854054	<code>((epochs, 14.0), (lr, 0.0750196610781145), (cell, 1.0), (memory_units, 36.0), (decay, 0.0690200826771197), (dropout, 0.121860865166101), (batch_size, 185.0))</code>	29	07:40:09.109754	19200
35	RNN2_IVR_BIN_W2V150QC_QR_QN_FACTURA	0.848858	<code>((kernel_size, 5.0), (batch_size, 469.0), (pool_size, 2.0), (memory_units, 303.0), (cell, 0.0), (decay, 0.0769827807899012), (padding, 1.0), (lr, 0.0418375676518619), (epochs, 7.0), (dropout, 0.358486430661762), (filters, 23.0))</code>	126	14:41:51.247073	11814
38	RNN2_IVR_BIN_W2V150QC_QR_QN_AVERÍA	0.797152	<code>((kernel_size, 7.0), (lr, 0.0378537983014233), (pool_size, 3.0), (decay, 0.0270850530925684), (padding, 0.0), (batch_size, 380.0), (cell, 0.0), (memory_units, 87.0), (epochs, 11.0), (dropout, 0.179448066463683), (filters, 23.0))</code>	582	16:53:32.080155	11673
32	RNN2_IVR_BIN_W2V150QC_QR_QN_COMERCIAL	0.795804	<code>((filters, 33.0), (kernel_size, 4.0), (cell, 0.0), (batch_size, 765.0), (epochs, 10.0), (decay, 0.0119204339719964), (memory_units, 343.0), (padding, 0.0), (pool_size, 4.0), (dropout, 0.251991845650367), (lr, 0.018451764339489))</code>	5	05:18:37.171854	11735
0	RNN_866_IVR_BIN_W2V150QC_QN_AVERÍA	0.789845	<code>((decay, 0.0103630280252387), (cell, 1.0), (epochs, 4.0), (lr, 0.0156142046274686), (batch_size, 659.0), (dropout, 0.493818845486034), (memory_units, 170.0))</code>	446	06:58:10.432142	19208
4	RNN_866_IVR_BIN_W2V150QC_QN_COMERCIAL	0.771328	<code>((decay, 0.0524916679520997), (epochs, 15.0), (cell, 1.0), (batch_size, 528.0), (memory_units, 66.0), (dropout, 0.102299572623736), (lr, 0.0201976677768787))</code>	33	20:13:44.980418	19897
25	RNN_IVR_BIN_W2V150QC_QR_QN_RECLAMACIÓN	0.761911	<code>((batch_size, 661.0), (cell, 1.0), (lr, 0.0169383441457147), (decay, 0.0782390568010709), (epochs, 7.0), (dropout, 0.359685372530682), (memory_units, 127.0))</code>	13	04:05:32.114764	11485
31	RNN2_IVR_BIN_W2V150QC_QR_QN_RECLAMACIÓN	0.741888	<code>((decay, 0.0285933960141452), (kernel_size, 5.0), (pool_size, 3.0), (cell, 1.0), (batch_size, 822.0), (filters, 39.0), (memory_units, 216.0), (padding, 0.0), (epochs, 7.0), (dropout, 0.427476318900752), (lr, 0.0107399796003009))</code>	519	17:55:52.174787	11524
2	RNN_866_IVR_BIN_W2V150QC_QN_RECLAMACIÓN	0.708280	<code>((lr, 0.0752684606427781), (epochs, 2.0), (cell, 1.0), (memory_units, 36.0), (decay, 0.05740054362473), (dropout, 0.133298599976467), (batch_size, 761.0))</code>	425	09:54:46.982383	19227
26	RNN_IVR_BIN_W2V150QC_QR_QN_AVERÍA	0.651022	<code>((batch_size, 622.0), (memory_units, 338.0), (cell, 0.0), (decay, 0.069262688750063), (epochs, 5.0), (lr, 0.0332309364336034), (dropout, 0.212773998634383))</code>	162	15:05:11.902764	10924
22	RNN_IVR_BIN_W2V150QC_QR_QN_COMERCIAL	0.538402	<code>((lr, 0.0746340107224834), (cell, 1.0), (batch_size, 990.0), (epochs, 12.0), (dropout, 0.450618173283494), (decay, 0.0080378958110251), (memory_units, 287.0))</code>	5	13:56:57.396916	11484

Figura 6.7: Optimización de modelos binarios con redes recurrentes

6.2.3. Modelos totalmente conectados

A continuación presentamos algunas pruebas, también de clasificación binaria, que realizamos utilizando como entrada directamente los vectores *Doc2Vec* de las transcripciones.

	study	accuracy	params	ntrials	total_time	best_trial
27	DENSE_IVR_BIN_D2V500w7m2_RECLAMACIÓN	0.725599	<code>((decay, 0.0377856747431033), (epochs, 10.0), (dropout, 0.298847478196375), (lr, 0.0593762691588953), (batch_size, 207.0))</code>	762	01:53:49.283701	4799
28	DENSE_IVR_BIN_D2V500w7m2_COMERCIAL	0.692321	<code>((batch_size, 469.0), (dropout, 0.0872092559873194), (lr, 0.0188969464466535), (decay, 0.00128187964520694), (epochs, 9.0))</code>	1517	04:47:27.896225	5075

Figura 6.8: Optimización de modelos binarios con *MLP*

Como observamos en la figura 6.8, únicamente hemos realizado pruebas de optimización con las clases *Reclamación* y *Comercial* devolviendo unos resultados de precisión sobre el conjunto de test del 72 % y 69 % respectivamente. El hecho de que los modelos anteriormente presentados tuvieran una mayor precisión nos llevó a no continuar por esta vía.

6.2.4. Modelos multiclas

Hasta ahora todos los modelos que hemos presentado son binarios y hemos tenido resultados bastante aceptables en estos casos. En este apartado vamos a mostrar modelos multiclas basados en las mismas arquitecturas vistas anteriormente.

	study	accuracy	params	ntrials	total_time	best_trial
13	RNN_866_IVR_MULALL_W2V150QC_QN_BF	0.810985	{(decay, 0.0332287311530309), (epochs, 15.0), (cell, 1.0), (memory_units, 149.0), (lr, 0.0130772443848036), (batch_size, 486.0), (dropout, 0.11319879246438)}	29	05:04:55.720171	20368
15	RNN_866_IVR_MULALL_W2V150QC_QN_BC	0.739601	{(decay, 0.0328426707805091), (lr, 0.0414983886251781), (cell, 1.0), (batch_size, 215.0), (epochs, 11.0), (dropout, 0.191091313671187), (memory_units, 42.0)}	192	15:06:43.790940	20646
14	RNN_866_IVR_MULALL_W2V150QC_QN_CF	0.721566	{(decay, 0.0455766288716424), (batch_size, 923.0), (lr, 0.067137891794526), (cell, 1.0), (epochs, 12.0), (dropout, 0.273525299277387), (memory_units, 43.0)}	114	23:58:15.661986	20379
5	RNN_866_IVR_MULALL_W2V150QC_QN_BCF	0.683083	{(memory_units, 39.0), (cell, 1.0), (batch_size, 682.0), (decay, 0.0629939815112358), (epochs, 9.0), (lr, 0.0124448720550143), (dropout, 0.13267072561035)}	114	06:57:08.013540	20497
16	RNN_866_IVR_MULALL_W2V150QC_QN_CR	0.643817	{(lr, 0.0982570484018716), (cell, 1.0), (decay, 0.0632519386277805), (batch_size, 944.0), (dropout, 0.212708265102609), (epochs, 13.0), (memory_units, 31.0)}	203	09:58:49.809159	20639
3	RNN_866_IVR_MULALL_W2V150QC_QN_	0.612574	{(decay, 0.0564302421625528), (lr, 0.0118536116122586), (cell, 1.0), (memory_units, 100.0), (dropout, 0.146082920845253), (epochs, 13.0), (batch_size, 876.0)}	100	06:02:53.511342	20151
6	RNN2_866_IVR_MULALL_W2V150QC_QN_	0.491579	{(lr, 0.0563633882292892), (kernel_size, 5.0), (pool_size, 3.0), (filters, 45.0), (cell, 1.0), (memory_units, 205.0), (decay, 0.0149661429365363), (padding, 1.0), (epochs, 11.0), (batch_size, 371.0), (dropout, 0.196451369147809)}	72	04:59:21.213054	20292
40	RNN_866_IVR_MULALL_W2V150QC_QN_AC	0.475396	{(batch_size, 1024.0), (memory_units, 38.0), (epochs, 2.0), (cell, 1.0), (decay, 0.0276762850659422), (dropout, 0.173462689227962), (lr, 0.0106123552430217)}	279	07:57:48.110430	20672

Figura 6.9: Optimización de modelos multiclas

En la figura 6.9 podemos observar los modelos multiclas que hemos construido y optimizado. Vemos que todos ellos los hemos construido con redes recurrentes de distintos tipos, que son las que, hasta ahora, nos han dado mejores resultados. Aunque hicimos también varias pruebas de conectar varios modelos binarios con la arquitectura propuesta en 6.1.5 los resultados no fueron satisfactorios y la calidad disminuía considerablemente.

Las últimas iniciales del campo *study* nos indican las clases que el modelo es capaz de distinguir, si acaba en “_” el modelo clasifica entre todas las clases.

Vemos que la mejor precisión en un modelo multiclas la tenemos hasta el momento en un modelo con redes recurrentes que distingue entre las clases *baja*, *facturas* y el resto de llamadas con una precisión del 81 %.

6.3. Modelo Mínimo Viable

El título de este apartado esta basado en el concepto de mínimo producto viable (MVP por sus siglas en inglés) acuñado por la metodología *Lean Startup*. Un producto mínimo viable debe reunir las características mínimas para satisfacer a los primeros clientes y poder crear una retroalimentación que ayude a que sea mejorado en el futuro. En nuestro caso queremos llevar a producción uno de los modelos diseñados hasta ahora para poder construir un sistema completo y obtener *feedback* que nos ayude en futuras mejoras.

El modelo que hemos seleccionado es el mejor modelo del estudio “RNN_866_IVR_MULLALL_W2V150QC_QN_BF” y podemos verlo en la figura 6.9. Se trata de un modelo basado en la arquitectura de redes recurrentes, con 149 celdas tipo *GRU* y que toma como entrada una secuencia de 866 *Tokens* (El 99 % de las transcripciones están por debajo de este número de *tokens*).

A continuación vamos a volver a medir la precisión del modelo usando tanto datos balanceados como sin balancear con el objetivo de verificar que el modelo pueda ser usado en un entorno productivo.

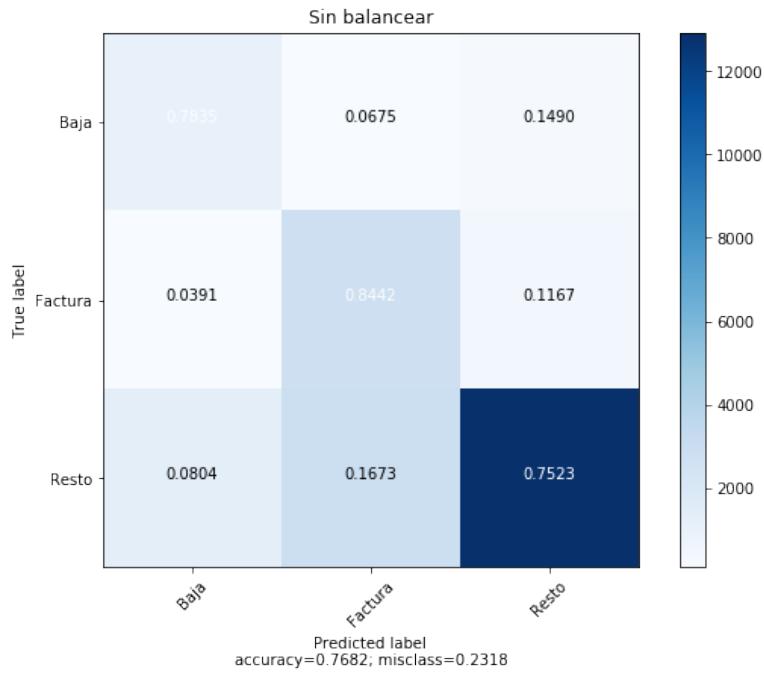


Figura 6.10: Matriz de confusión datos balanceados

En la figura 6.10 podemos ver la precisión del modelo con datos balanceados, mientras que en la figura 6.11 vemos estos mismos datos sin balancear. Observamos que en el modelo sin balancear la mayoría de muestras se concentran en el resto, lo que es normal, ya que hay más

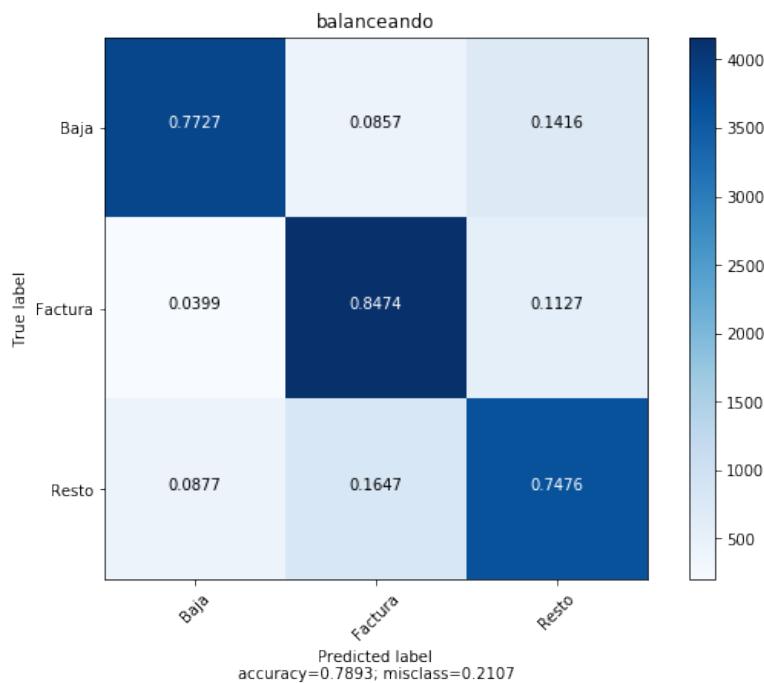


Figura 6.11: Matriz de confusión datos no balanceados

llamadas de otros tipos que de *Baja* y *Factura*. Sin embargo, podemos ver que la precisión no varía sustancialmente por lo que el modelo es susceptible de usar en producción.

La siguiente parte del documento tratará de llevar a producción este modelo para que sea capaz de clasificar las llamadas en tiempo real con todo lo que ello implica. A partir de ahora nos referiremos a él como el modelo ***bajafactura*** y lo veremos como una caja negra en la que solo nos preocuparemos por la forma en la que debemos introducir los datos y por las clasificaciones devueltas por el mismo.

Parte III

Explotación: procesamiento, visualización y alarmados

Capítulo 7

Explotación

En los anteriores capítulos hemos descrito el proceso completo que hemos realizado con los datos: Los hemos localizado, los hemos limpiado, hemos creado diferentes modelos de minería de datos y hemos repetido estos pasos de manera iterativa. En este capítulo comentaremos el camino seguido para llevar los modelos construidos a un entorno productivo real.

Este paso que relata el viaje “del laboratorio a la fábrica”, nos permitirá cumplir el segundo objetivo planteado en la sección 1.3 del documento: **extraer esta temática para nuevas llamadas en tiempo real**. Además como veremos en la sección 7.1 será necesaria para cumplir con otros requisitos propios de un sistema productivo.

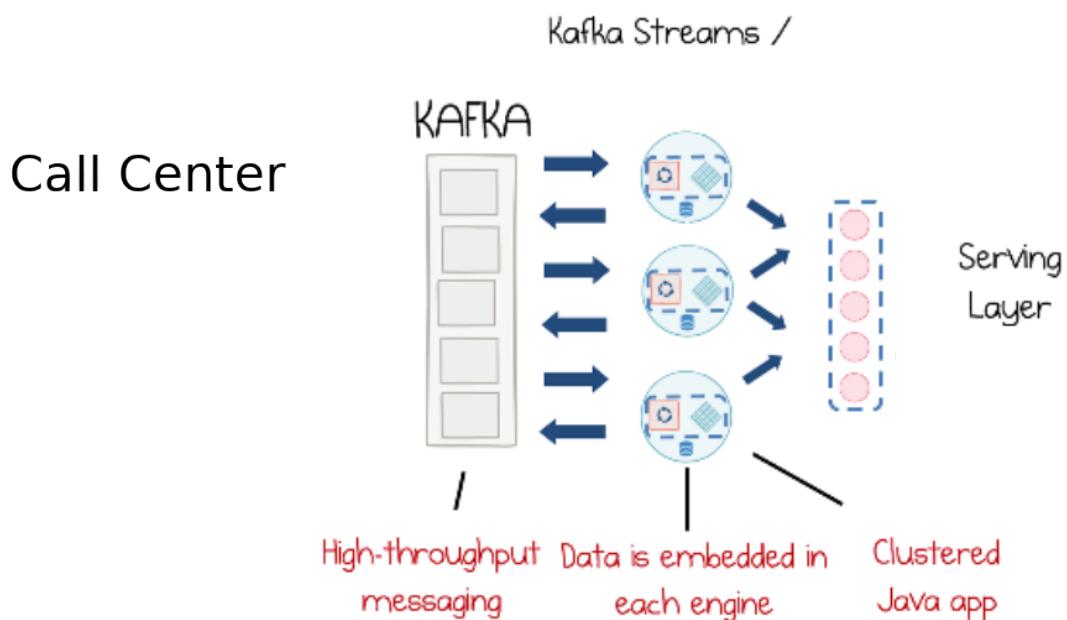


Figura 7.1: Capa de *streaming*

En la figura 7.1 podemos ver un *zoom* de la capa de servicio de nuestra arquitectura que

describiremos a lo largo del capítulo.

7.1. Requisitos del sistema productivo

En esta primera sección, antes de empezar con la definición de la capa de *streaming*, vamos a recapitular los requisitos que deberá cumplir el sistema encargado de clasificar las llamadas recibidas del *call center* en tiempo real.

- **Tiempo real:** El sistema debe ser capaz de clasificar las llamadas en tiempo real con una latencia máxima del orden de segundos.
- **Escalabilidad:** El sistema debe poder escalar horizontalmente de modo que pueda responder en un futuro a un número de llamadas mayor.
- **Alta disponibilidad:** El sistema debe estar siempre disponible sin que exista en el mismo un punto único de fallo (SPOF , *Single Point Of Failure*).
- **Integración y despliegue continuos:** La integración y el despliegue del sistema deben ser continuos. Esto significa que una vez subido el código a un gestor de versiones deben poder realizarse de manera automática las pruebas necesarias, la compilación y el despliegue del sistema.

A lo largo del capítulo veremos como la aplicación cumple con los tres primeros puntos: tiempo real, escalabilidad y alta disponibilidad. El cuarto punto referente a la integración y el despliegue continuo se tratará de manera individual en el capítulo [10](#).

7.2. Arquitectura del sistema

Al abordar la construcción del sistema hemos decidido usar una **arquitectura de microservicios**, cada uno de los cuales ejecute una parte del proceso. A la hora de diseñar estos microservicios hemos tenido en cuenta que cada uno ejecute una unidad de *software* que tenga sentido por si misma y que pueda ser actualizada, sustituida o utilizada por terceros de una forma independiente.

Una idea fundamental que subyace detrás de toda arquitectura de microservicios, es que cada microservicio desempeñe su tarea lo más aisladamente del resto y su comunicación con otros microservicios se realice usando protocolos simples y agnósticos a cualquier tecnología, usualmente el protocolo HTTP mediante API REST.

Sin embargo el uso del protocolo HTTP tiene una serie de implicaciones que pueden no ser muy ventajosas para un sistema que requiere un procesamiento en tiempo real y que busca la simplicidad, como son la sincronía y el acoplamiento con otros microservicios. La sincronía nos lleva a tener que “preguntar” constantemente a un microservicio sobre la existencia de nuevos datos, lo que, desde el punto de vista del rendimiento, implica que el *thread* encargado del procesamiento deba esperar la respuesta HTTP antes de seguir con el procesamiento (esta perdida de rendimiento variará en función del tiempo de procesamiento necesario del microservicio llamado y de la latencia de la red). Por otro lado, una arquitectura en la que los microservicios se comuniquen mediante REST implica un mínimo de acoplamiento, debido a que los microservicios deben conocer la existencia unos de otros, traduciéndose en una mayor complejidad a la hora de la orquestación de los mismos.

Es por ello que una solución más optima para nuestro sistema consiste en combinar los beneficios de las arquitecturas *event-driven* con los beneficios de los microservicios utilizando una **arquitectura de microservicios *event-driven* (EDM)**. Para desarrollar la arquitectura propuesta, necesitaremos un sistema de mensajería o *streaming* por el que “viajen” los eventos y gracias al cual los microservicios puedan comunicarse entre sí. En nuestro caso hemos optado por utilizar **Apache Kafka como bus central de eventos**.

El uso de Apache Kafka solventa los inconvenientes planteados por la arquitectura *REST* eliminando la sincronía y el acoplamiento. Por un lado, los microservicios que tienen como origen un bus de mensajería son por naturaleza asíncronos, lo que implica que realizarán el procesamiento de los eventos en el momento que estos sean publicados en el bus; eliminando la necesidad de tener que “preguntar” por nuevos eventos y aumentando el rendimiento. Por otro lado, el uso de Apache Kafka hace que los microservicios estén totalmente desacoplados entre sí, eliminando la necesidad de que un microservicio conozca siquiera la existencia del resto.

Además la arquitectura propuesta nos aporta otras características ventajosas para nuestro proyecto:

- **Reprocesamiento:** Gracias a la capacidad de retención del bus, tenemos la posibilidad de reprocesar los mensajes pasados de ser necesario (p.e. para solventar errores de código).
- **Alta disponibilidad.** El sistema es tolerante a fallos. Los datos se encuentran en el bus con un número de réplicas que evita la pérdida de los mismos ante caídas de nodos.
- **Escalabilidad:** Apache Kafka nos da la posibilidad de dividir nuestros *topics* en particiones que pueden ubicarse en diferentes nodos. Esto nos proporciona escalabilidad horizontal tanto desde el punto de vista del almacenamiento como del cómputo.
- **Soportar picos de carga:** Al existir una retención en el bus, en el caso de que ocurriese un pico en el número de eventos recibido que no puedan ser abordados por los

procesadores, estos podrán ir procesando la información según su capacidad.

- **Evento en el centro:** Este tipo de arquitectura hace que diseñemos nuestras aplicaciones situando al evento, al dato, en el centro de nuestro sistema.

Sin embargo, aunque el uso de este tipo de microservicios *event-driven* (EDM) posea todas estas ventajas, es cierto que también hay que asumir que el uso de REST está muy implementado actualmente y muchas aplicaciones poseen un API REST *out-of-the-box*. Esto nos llevará como veremos en el siguiente apartado a incorporar microservicios REST en nuestro modelo *event driven*, creando en cierto modo una arquitectura híbrida.

7.3. Microservicios

En la sección anterior decidimos el modelo de arquitectura que vamos a utilizar para construir nuestro sistema, en esta sección nos centraremos en los microservicios que compondrán esa arquitectura.

7.3.1. Tecnología

En el apartado 3.4 hicimos un repaso por las tecnologías usadas en el proyecto, en este punto queremos tratar de profundizar con un poco más de detalle en las tecnologías que nos permitirán desarrollar y desplegar nuestros microservicios. Estas tecnologías son Kafka Streams y Tensorflow Serving.

7.3.1.1. *Kafka Streams*

El núcleo de nuestra capa de *Streaming* estará compuesto por microservicios desarrollados mediante Kafka Streams. Kafka Streams [33] es una librería cliente que nos permite desarrollar aplicaciones y microservicios cuya entrada y salida sea un bus Kafka.

Kafka Streams nos permite desarrollar aplicaciones en *Java* (el lenguaje que usaremos) o *Scala* de una manera sencilla e intuitiva, además, a diferencia de otros *frameworks* de procesamiento en tiempo real, como puede ser *Spark Streaming*, no tiene necesidad de disponer de un clúster aparte (usa el mismo bus Apache Kafka) lo que simplifica bastante nuestra arquitectura. Además Kafka Streams nos permitirá **garantizar la escalabilidad y la tolerancia a fallos** de nuestros despliegues.

Kafka Streams posee dos APIs diferentes: Processor API y Streams DSL. Processor API nos proporciona una capacidad a muy bajo nivel para definir la lógica de nuestro proceso *streaming*; en cambio *DSL*, que está construido sobre Processor API, nos permite de una manera

sencilla, con un lenguaje declarativo, realizar la mayoría de operaciones posibles en un proceso de *streaming*. Es por ello que en nuestros servicios utilizaremos principalmente **Streams DSL**, aunque en algún momento tengamos que recurrir a la Processor API para realizar tareas más avanzadas.

Una característica interesante de Streams DSL es la existencia de las denominadas KTables que nos permiten usar el Bus como si de una BBDD NoSQL Clave-Valor se tratara. Las Ktables utilizan un *topic* de Apache Kafka, pero al acceder a ellas solo accedemos al último registro disponible para cada clave. Esta característica suele usarse sobre *topics* compactos sin límite de retención, que son aquellos que periódicamente borran las versiones antiguas de cada clave y retienen, de forma ilimitada, las versiones más recientes. Esta posibilidad nos permitirá cargar en el bus las tablas de *lookup* necesarias para enriquecer registros en nuestro procesamiento y eliminará las dependencias de nuestro flujo con BBDD externas.

7.3.1.2. *Tensorflow Serving*

Tensorflow Serving [32] está pensado para llevar a producción los modelos que hemos creado con *Tensor Flow*, pudiéndose adaptar también para otros modelos. Tensorflow Serving nos permite desplegar nuestros modelos y algoritmos manteniendo intacta la API de consulta, esto lo hace ideal para entornos en los que queremos ir actualizando las versiones de nuestros modelos sin modificar el resto del sistema.

Los modelos desplegados con Tensorflow Serving pueden consultarse a través de un *API* en C++ o mediante un **API REST** que es la opción escogida por nosotros debido a que, por la simplicidad del protocolo, es la que más encaja con nuestra arquitectura de microservicios.

Una característica interesante de *Tensorflow Serving* es el **versionado**, ya que nos permite servir en paralelo varias versiones de un modelo, algo que puede ser muy interesante para, por ejemplo, realizar un test A/B.

7.3.2. Microservicios

Una vez determinada la arquitectura del sistema y las tecnologías escogidas para su implementación es el momento de diseñar los microservicios necesarios para nuestro caso de uso.

El primer paso es localizar las funciones básicas que queremos que realice nuestro sistema agrupadas en dos grandes etapas: preprocesamiento y predicción. La etapa de preprocesamiento será la encargada de preparar las llamadas para que puedan convertirse en la entrada de nuestro modelo, mientras que la etapa de predicción será la encargada de aplicar el modelo.

Dentro de la etapa de preprocesamiento es necesario, en un primer lugar, la extracción de los *tokens* de la transcripción de llamada, de esto se encargará un microservicio que hemos

denominado **Tokenizer**. Una vez obtenidos los *Tokens* también será necesario, para que puedan convertirse en entrada del modelo, transformarlos en una secuencia numérica; esto será realizado por otro microservicio denominado **Sequencer**.

Por último en la capa de predicción son necesarios otros dos pasos: por un lado aplicar el modelo obteniendo una probabilidad de pertenencia a cada clase, de esto se encargará un microservicio denominado **tf-BajaFactura** y, por otro lado, enriquecer estas predicciones con las etiquetas necesarias y enriquecer la salida del microservicio anterior; esto será responsabilidad de un microservicio denominado **Predicter**. Como podemos imaginar por la definición, existirá un ligero acoplamiento entre estos dos microservicios, que comentaremos a lo largo de esta sección.

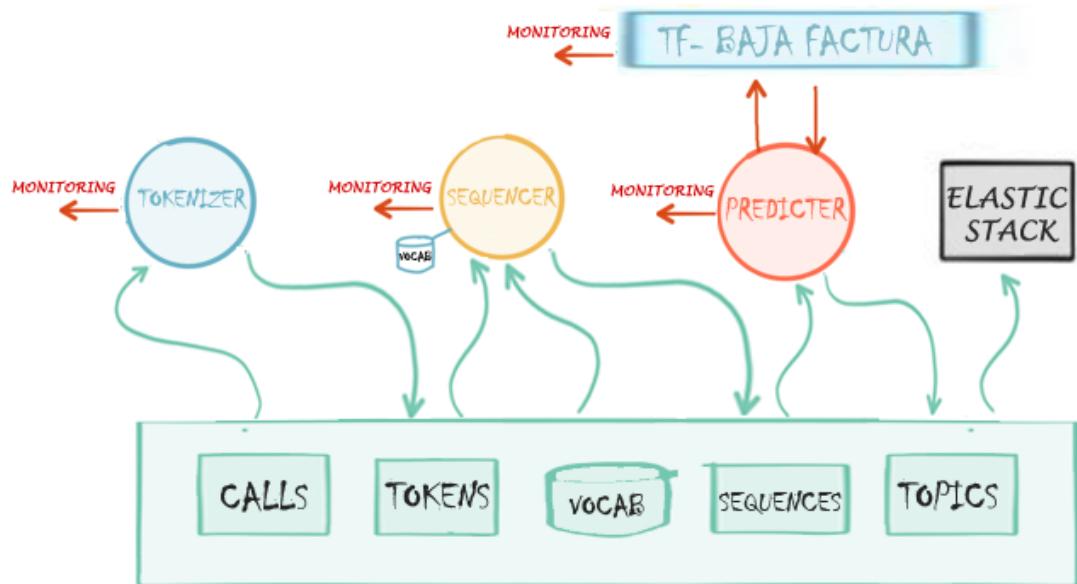


Figura 7.2: Arquitectura microservicios

En la figura 7.2 podemos ver como los microservicios propuestos interactúan entre sí a través del bus, con excepción del microservicio **tf-BajaFactura**. Estos microservicios conforman la parte capa de *streaming* de nuestro sistema. En el capítulo siguiente veremos como la salida de esta capa de *streaming* alimentará la capa de servicio.

A continuación, en los siguientes subapartados describiremos uno a uno los diferentes microservicios.

7.3.3. *Tokenizer*

Este microservicio tiene como entrada las llamadas transcritas que llegan al bus en tiempo real. A partir del texto de las mismas el sistema inicia un proceso de *tokenizacion* eliminan-

do *stopwords*, caracteres especiales (signos de puntuación, 'ñ's, acentos...), palabras comunes, números, nombres propios y pasando a minúscula cada uno de los tokens. El proceso de *tokenización* pretende ser lo más fiel posible al proceso realizado en *Python* durante la etapa de entrenamiento del modelo.

Una vez obtenida la lista de *tokens* esta se vuelve a disponibilizar en el bus en un nuevo *topic*. Este nuevo *topic* con las llamadas *tokenizadas* puede ser de utilidad no solo para nuestro sistema, si no también para cualquier otro sistema de análisis de texto que necesite partir de los datos *tokenizados*.

El *topic* de entrada del microservicio *Tokenizer*, llamado *CALLS*, tendrá como clave el código de la llamada y como cuerpo un objeto *json* con los siguientes campos:

- ***co_verint*** : Código de la llamada.
- ***call_text*** : Texto de la llamada.
- ***call_timestamp***: Hora de la llamada.
- ***province***: Provincia desde la que se ha realizado la llamada.
- ***co_province***: Código de provincia desde la que se ha realizado la llamada.
- ***duration***: Duración en segundos de la llamada.
- ***control_type***: De existir, tipo de llamada (usado para el proceso de verificación).

Como salida escribirá en un *topic* llamado *TOKENS.CALLS* un evento cuya clave será el código de la llamada y cuyo cuerpo un objeto *json* con los siguientes campos:

- ***co_verint*** : Código de la llamada.
- ***call_text*** : Texto de la llamada.
- ***call_timestamp***: Hora de la llamada.
- ***province***: Provincia desde la que se ha realizado la llamada.
- ***co_province***: Código de provincia desde la que se ha realizado la llamada.
- ***duration***: Duración en segundos de la llamada.
- ***control_type***: De existir, tipo de llamada (usado para el proceso de verificación).
- ***tokens***: Lista de tokens extraída de *call_text*.

7.3.4. Sequencer

Este microservicio toma como entrada la salida del microservicio anterior y a partir de la lista de *tokens* devuelve una secuencia de tamaño fijo determinado T . Para ello, utiliza un diccionario en el que cada *token* se corresponde con un número, utilizando el 0 para *tokens* que no existan en este diccionario. Esta secuencia está limitada a un tamaño determinado T , por lo que llamadas con un número mayor de *tokens* son recortadas y se utiliza *padding* por la izquierda para completar la secuencia de llamadas con un tamaño menor a T .

El diccionario ha sido calculado sobre el conjunto de datos de entrenamiento y contiene el vocabulario del modelo. Este diccionario contiene como clave todos los *tokens* posibles y como valores el identificador usado para cada *token*. La lectura del mismo se realiza desde el mismo bus, utilizando la funcionalidad *KTable* de *Kafka Streams*.

Una vez obtenida la secuencia de caracteres, esta se disponibiliza en un nuevo *topic* del bus. Además de por nuestro sistema, este valor puede ser consumido por otros microservicios o sistemas que tengan como objetivo aplicar diferentes modelos a las secuencias obtenidas.

El *topic* de entrada del microservicio *Sequencer* será el topic *TOKENS.CALLS* descrito en el apartado anterior.

Como salida escribirá en un *topic* llamado *SEQUENCES.CALLS* un evento cuya clave será el código de la llamada y cuyo cuerpo un objeto *json* con los siguientes campos:

- *co_verint* : Código de la llamada.
- *call_text* : Texto de la llamada.
- *call_timestamp*: Hora de la llamada.
- *province*: Provincia desde la que se ha realizado la llamada.
- *co_province*: Código de provincia desde la que se ha realizado la llamada.
- *duration*: Duración en segundos de la llamada.
- *control_type*: De existir, tipo de llamada (usado para el proceso de verificación).
- *sequence*: Lista de 866 enteros en el que cada uno representa el identificador de un *token*. La secuencia se completa con ceros en caso de ser necesario.

Además se apoyará en un topic compacto que no tendrá límite de retención, denominado *TBL.VOCABULARY.CALLS*, cuya clave son los *tokens* existentes en el vocabulario y, cuyo valor es un identificador para cada uno de ellos.

7.3.5. *tf-BajaFactura*

tf-BajaFactura quizás sea el microservicio más tradicional y discordante de los que vamos a utilizar en nuestro sistema debido a que este microservicio se encuentra totalmente desacoplado del bus y del resto de microservicios. Mediante un API REST aplica nuestro modelo de TensorFlow a una o varias secuencias de entradas y, para cada secuencia, devuelve la probabilidad de pertenencia a las clases: “Baja”, “Factura” y “Resto”.

El motivo principal para usar API REST en este microservicio se debe al uso de la tecnología Tensorflow Serving comentada en el capítulo anterior y que nos permite con mucha facilidad desplegar nuevos modelos y algoritmos, manteniendo la misma estructura en el servidor y la misma API. Además su integración es automática con los modelos generados con TensorFlow (con o sin Keras). Esta decisión permite a los *data-scientist* desplegar nuevas versiones de sus modelos, ya sea por degradación o por mejora, mientras el resto del flujo permanece inalterable.

La entrada del modelo *tf-BajaFactura* será un documento *json* que contenga el campo *instances* compuesto por una lista de secuencias. En nuestro caso una sola secuencia, ya que el modelo será llamado con cada evento recibido. La salida será también en *json* y contendrá una lista de predicciones, en cada predicción tendremos la probabilidad de pertenencia a cada clase.

7.3.6. *Predicter*

Este último microservicio tiene como entrada el *topic* generado por *Sequencer* y a partir del mismo realiza una llamada a *tf-BajaFactura*. Una vez obtenidas las predicciones, las enriquece con las etiquetas necesarias. Además en el caso de existir un atributo de control comprueba si la predicción ha sido correcta, esto nos servirá para validar la eficiencia del modelo a lo largo del tiempo.

Como podemos observar, es el único microservicio que tiene una dependencia con otro (*tf-BajaFactura*), sin embargo la API de un modelo implementado con Tensorflow Serving [32] es siempre idéntica. Esto provoca que la llamada no varíe aunque cambie el modelo. Además, en el servicio *Predicter*, el número de clases y las etiquetas de las mismas son configurables , pudiendo reutilizarse este microservicio para cualquier servicio predictivo desplegado mediante Tensorflow Serving.

La salida de este microservicio es publicada en un nuevo *topic* en el bus, esta información puede ser de utilidad para diversos sistemas que quieran realizar analítica en función de la temática de las llamadas o que quieran visualizarla. Nosotros la usaremos en nuestra capa de servicio que describiremos en el siguiente capítulo.

El *topic* de entrada del microservicio *Predicter* será el topic *SEQUENCES.CALLS* descrito

en el servicio *Sequencer*.

Como salida escribirá en un *topic* llamado *TOPICS.CALLS* un evento cuya clave será el código de la llamada y cuyo cuerpo un objeto *json* con los siguientes campos:

- *co_verint* : Código de la llamada.
- *call_text* : Texto de la llamada.
- *call_timestamp*: Hora de la llamada.
- *province*: Provincia desde la que se ha realizado la llamada.
- *co_province*: Código de provincia desde la que se ha realizado la llamada.
- *duration*: Duración en segundos de la llamada.
- *control_type*: De existir, tipo de llamada (usado para el proceso de verificación).
- *predictions*: Lista con las predicciones devueltas por el servicio *tf-BajaFactura*.
- *error*: De producirse, contendrá el error devuelto por el servicio *tf-BajaFactura* o el error de conexión con el mismo.
- *model*: ID del modelo aplicado, en nuestro caso siempre será BajaFactura.
- *pred_type*: Etiqueta de la clase más probable según la predicción.
- *control_success*: En el caso de existir el campo *control_type*, Booleano que nos indica si la predicción ha sido correcta.

7.4. Monitorización de Microservicios

Si bien es cierto que el hecho de utilizar una arquitectura orientada a eventos, aparte de la sencillez intrínseca de nuestro sistema, nos simplifica en cierto modo la orquestación de nuestro servicios y nos permite prescindir de herramientas de APM (*Application Performance Monitoring*); una monitorización del desempeño de nuestros microservicios es siempre necesaria, tanto para garantizar el correcto funcionamiento de los mismos y poder detectar posibles puntos de fallo, como para detectar pérdidas de rendimiento en cualquier punto del sistema.

En esta sección veremos los métodos utilizados para monitorizar los microservicios dependiendo de su tecnología.

7.4.1. Servicios *Kafka Streams*

Los servicios de Kafka Streams corren sobre la máquina virtual de Java lo que nos permite tener acceso a sus métricas por medio de la JMX(*Java Management Console*). Ademas Confluent posee una documentación de los MBeans y de las métricas que contienen en su documentación oficial [24].

El inconveniente de estas métricas es que tienen que extraerse por medio de protocolos RMI (*Java Remote Method Invocation*) lo que complica su explotación, sin embargo a partir de la versión 6 del JDK (*Java Development Kit*) tenemos la posibilidad de exportar estas métricas de JMX directamente mediante API REST. Esto nos da la posibilidad de poder recolectar las métricas directamente mediante una llamada HTTP.

Para recolectar estás métricas utilizaremos los Beats de Elastic, concretamente Metricbeat. Este será el encargado de recolectar las métricas necesarias y enviarlas a Logstash para su posterior almacenamiento en Elasticsearch. El flujo de Logstash a Elasticsearch lo trataremos con más detalle en el próximo capítulo en el que veremos la capa de servicio, ya que las métricas de monitorización seguirán el mismo recorrido que los datos de clasificación.

De todas las métricas disponibles de Confluent, nosotros extraeremos con una frecuencia de un minuto las siguientes métricas:

- *mbean 'java.lang:type=Runtime'* : Métrica *Uptime*.
- *mbean 'kafka.streams:type=stream-metrics'*: Métricas *process-latency-avg*, *process-latency-max* y *process-rate*.

Esto nos permitirá para cada microservicio tener información del tiempo de servicio del mismo, de la latencia máxima y media de procesamiento de los eventos y de la tasa de procesamiento.

7.4.2. Servicios *Tensorflow Serving*

En el caso de la monitorización del servicio *tf-BajaFactura*, habilitaremos la monitorización de Prometheus de Tensorflow Serving para que las métricas sean accesibles mediante API REST, pero en lugar de utilizar Prometheus (para no incluir complejidad en nuestro sistema), usaremos directamente el *Input Http_poller* de Logstash.

De todas las métricas disponibles en Tensorflow Serving nos quedaremos únicamente con:

- **Estado:** Que nos dará información sobre si el modelo ha sido cargado con éxito y se encuentra operativo.
- **Total peticiones:** Total de peticiones resueltas por el modelo.

- **Total nanosegundos del grafo:** El total de tiempo que el modelo ha dedicado en procesar todas las peticiones. Utilizaremos el total de peticiones para extraer la latencia media y lo expresaremos en milisegundos.

Una vez recolectadas las métricas y enviadas a Logstash, seguirán el mismo flujo descrito en el apartado anterior hacia la capa de servicio, este flujo será tratado en más detalle en el capítulo siguiente.

Además de las métricas recolectadas, la monitorización del microservicio *predicter* nos dará información combinada de ambos ya que internamente realiza una llamada a *tf-BajaFactura*.

7.5. Inyector: simulador de tiempo real

Como ya hemos comentado en anteriores capítulos actualmente las transcripciones de las llamadas no se encuentran disponibles en tiempo real, es por ello que se ha diseñado un inyector de llamadas.

Este inyector, realizado en Python utiliza como fuente la transcripción de las llamadas realizadas en batch. Y a partir de la misma cumplirá los siguientes objetivos:

- **Extraer la frecuencia de llamadas por hora.** De este modo simularemos el comportamiento real de un *call-center* inyectando más llamadas a las horas del día con mayor actividad.
- **Multiplicar el número de llamadas.** Teniendo en cuenta que actualmente solo se transcriben un 4% del total de las llamadas, el inyector multiplicará por 25 el número de llamadas recibidas en un día para simular el comportamiento real de un sistema que ingeste el 100% de las llamadas.
- **Introducir datos de control:** El inyector insertará en un porcentaje de llamadas los datos reales de clasificación con el fin de poder verificar el comportamiento del modelo. En un entorno real se podrían insertar periódicamente un número de llamadas de control que nos permitan este objetivo.

Una vez realizadas estas tareas, el inyector publicará todas las llamadas al bus, simulando un comportamiento real. De este modo las llamadas estarán disponibles en tiempo real para cualquier consumidor que las necesite.

Aunque el inyector podría verse como un microservicio más, hemos preferido dejarlo fuera del sistema a lo largo de todo el apartado, debido a que lo consideramos una pieza auxiliar que nos permite simular el comportamiento real del sistema sin disponer de los datos en este momento.

Capítulo 8

Capa de servicio

8.1. Carga y modelo

8.2. Visualizaciones

8.2.1. Monitorización

8.2.2. Sistema

8.3. Alarmado

Capítulo 9

Despliegue en contenedores

Una vez desarrollados los microservicios y su monitorización, es necesario ponerlos en producción, y una de las decisiones importantes es dónde queremos que corran, si en máquinas físicas, máquinas virtuales o en contenedores (en nuestro caso, se descarta la opción de implementarlo en *Cloud*). La decisión tomada es que todo corra sobre contenedores en una plataforma *OCP OpenShift Container Platform*, los motivos de esta decisión son los siguientes:

- ***Stateless***: Todos los servicios implementados no necesitan estado (los datos necesarios se almacenan en el bus). Lo que lo hace un caso de uso ideal para su ejecución en contenedores.
- **Agilidad de despliegue**: El proceso de creación y destrucción de un contenedor es inmediato, lo que no da mucha flexibilidad a la hora desplegar nuevos contenedores.
- **Servicios Autocontenidos**: Los contenedores pueden ejecutarse en multitud de plataformas lo que nos permite tener aplicaciones inmutables y autocontenido que podemos mover entre plataformas y entornos. Este punto y el anterior serán de mucha utilidad a la hora de implementar la integración y el despliegue continuo.
- **Alta disponibilidad**: Al tener un clúster de *OpenShift* con múltiples nodos, en el caso que se produjera un fallo en un nodo que provocara la caída de un contenedor, este se levantaría automáticamente en otro nodo.

A lo largo del apartado nos referiremos a los siguientes conceptos propios de *OpenShift* (muchos de ellos heredados en *Kubernetes*):

- ***Pod***: Un pod es la unidad mínima en *OCP*. Puede estar formado por uno o varios contenedores (*Docker*), pero todos ellos correrán en una única máquina y tendrán una única dirección IP.

- **Despliegue:** Un despliegue (*Deployment*) podemos decir, de una manera muy simplificada, que gestiona el número de réplicas de un *pod*.
- **Configuración de despliegue:** La configuración de despliegue (*Deployment Config*) dota a los despliegues de un ciclo de vida proporcionando versiones de los mismo, disparadores para crear nuevos despliegues de manera automática y estrategias para realizar transiciones entre diferentes versiones de despliegues sin pérdida de servicio.
- **Servicio:** Un servicio expone un número determinado de Pods para que sean accesibles desde otros elementos de la plataforma.
- **ConfigMap:** Se trata de un mecanismo de *OCP* para almacenar elementos de configuración mediante un mecanismo de clave-valor. Es posible almacenar tanto ficheros como valores simples.

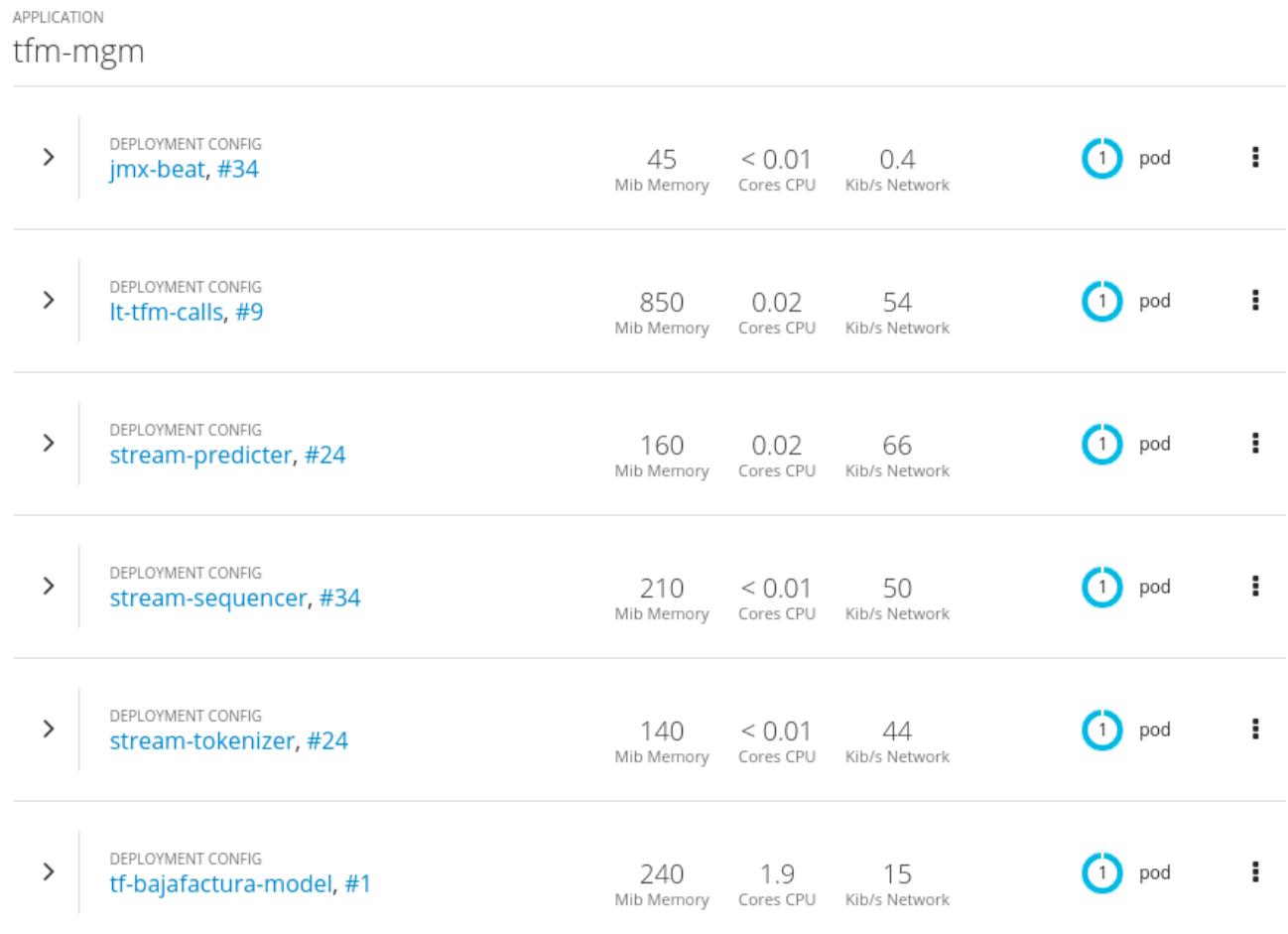


Figura 9.1: configuración de despliegue y recursos usados en OCP

En la figura 9.1 podemos ver las diferentes configuraciones de despliegue con los *pods* que tienen levantados cada uno y los recursos de memoria, cpu y red que consumen cada uno.

A continuación veremos un resumen de los servicios, despliegues y configuraciones desplegadas en Openshift para cada tipo de microservicio y monitorización. En el apéndice [?] podemos encontrar todo el código de despliegue en Openshift, que nos permitiría desplegar todo nuestro entorno en otro clúster en cuestión de minutos.

9.1. Servicios *Kafka Streams*

Para desplegar los servicios de *Kafka Streams* en *OCP* es necesario desplegar una configuración de despliegue por microservicio, un servicio para exponer las métricas y disponer en un *configmap* de las opciones necesarias para su ejecución.

En este apartado únicamente presentaremos a modo de ejemplo la configuración de despliegue y el servicio del microservicio *Predicter* explicando las características más relevantes del mismo. Tanto el *configmap* con todos los ficheros como el resto de despliegues pueden consultarse en el apéndice [?].

La configuración de despliegue de *Predicter* es la siguiente:

```
1 apiVersion: apps.openshift.io/v1
2 kind: DeploymentConfig
3 metadata:
4   labels:
5     app: tfm-mgm
6     project: topic-model
7     service: stream-predicter
8   name: stream-predicter
9   namespace: nbia-prod
10 spec:
11   replicas: 1
12   revisionHistoryLimit: 10
13   selector:
14     deploymentconfig: stream-predicter
15   strategy:
16     activeDeadlineSeconds: 21600
17     recreateParams:
18       timeoutSeconds: 600
19     resources: {}
```

```
20      type: Recreate
21
22  template:
23    metadata:
24      creationTimestamp: null
25    labels:
26      app: tfm-mgm
27      deploymentconfig: stream-predicter
28      project: topic-model
29      service: calls-predicter
30
31  spec:
32    containers:
33      - env:
34          - name: JAVA_MAIN_CLASS
35            value: com.telefonica.topicmodel.PredicterLauncher
36
37      image: >-
38        docker-registry.default.svc:5000/nbia-prod/
39        topic-model-streaming:latest
40
41      imagePullPolicy: Always
42
43      name: stream-predicter
44
45      ports:
46        - containerPort: 8778
47          protocol: TCP
48
49      resources:
50        limits:
51          cpu: '1'
52          memory: 512Mi
53
54        requests:
55          cpu: 200m
56          memory: 256Mi
57
58      terminationMessagePath: /dev/termination-log
59
60      terminationMessagePolicy: File
61
62      volumeMounts:
63        - mountPath: /opt/jolokia/etc/jolokia.properties
64          name: calls-config
65          subPath: jolokia.properties
66
67        - mountPath: /deployments/application.json
```

```

56      name: calls-config
57      subPath: topic-model-streaming.json
58      dnsPolicy: ClusterFirst
59      restartPolicy: Always
60      schedulerName: default-scheduler
61      terminationGracePeriodSeconds: 30
62      volumes:
63        - configMap:
64          defaultMode: 420
65          name: calls-config
66          name: calls-config
67      test: false
68      triggers:
69        - imageChangeParams:
70          automatic: true
71          containerNames:
72            - stream-predictor
73          from:
74            kind: ImageStreamTag
75            name: 'topic-model-streaming:latest'
76            namespace: nbia-prod
77            type: ImageChange
78        - type: ConfigChange

```

Del despliegue anteriormente expuesto podemos destacar las siguientes características:

- **labels** (línea 4): Nos ayudan a identificar el despliegue, el proyecto y la aplicación. Todos los despliegues relacionados con el TFM que se presentan en este documento tendrán el indicador *tfm-mgm*.
- **replicas** (línea 11): Indica el número de réplicas de cada *pod* que pueden correr al mismo tiempo. En este caso lo establecemos a uno, pero podría escalar horizontalmente incluso hacerlo de manera automática por uso de CPU o memoria mediante un *Horizontal Pod Autoscaler*.
- **strategy** (línea 15): Establecemos la estrategia con la que se implantará una nueva versión del *deployment*, en este caso utilizamos una estrategia *Recreate* que detiene todos los servicios de un *deployment* anterior antes de iniciar uno nuevo. Debido a que el bus tiene

capacidad de retención de los eventos, esto no implicará una interrupción en el servicio, si no un ligero retraso en los eventos que se ingesten durante el despliegue.

- **Container** (línea 30): Aquí encontramos toda la información del contenedor que se desplegará. Destacamos:
 - **env** (línea 31): Variables globales del *deployment*, en este caso la clase Java principal que se lanzará al ejecutar el contenedor.
 - **image** (línea 34): Contiene la referencia a la imagen del contenedor que queremos desplegar. Se trata de una imagen de Java con nuestro código que se encuentra almacenada en el repositorio de la plataforma. El parámetro *imagePullPolicy* indica en qué circunstancias se descargará la imagen del repositorio (en este caso lo hará siempre, exista o no en el nodo en el que se despliega el *Pod*).
 - **Ports** (línea 34): Indica los puertos que expondrá el contenedor, en este caso se expondrá únicamente el puerto 8778 que es el puerto por defecto del *endpoint* de *Jolokia*.
 - **Resources** (línea 42): Contiene los recursos de CPU y memoria que solicitará el *Pod* al arrancar y a los que estara limitados.
 - **VolumeMounts** (línea 51): Indica los volúmenes que montará el contenedor, en este caso son todos procedentes de un *configmap* y contienen los ficheros de configuración de *jolokia* y de la aplicación.
- **volumes** (línea 62): Contiene los volúmenes que se utilizaran en los contenedores en el apartado *VolumeMounts*. En este caso únicamente contiene un *configmap*.
- **triggers** (línea 68): Define los disparadores que provocarán que se realice un nuevo *deployment*, en este caso la modificación de la imagen o la modificación de la configuración de despliegue.

El servicio de *Predicter* es el siguiente:

```

1 apiVersion: v1
2 kind: Service
3 metadata:
4   labels:
5     app: tfm-mgm
6   name: stream-predictter
7   namespace: nbia-prod

```

```

8   spec:
9     ports:
10    - name: 8778-tcp
11      port: 8778
12      protocol: TCP
13      targetPort: 8778
14    selector:
15      deploymentconfig: stream-predictor
16    sessionAffinity: None
17    type: ClusterIP

```

A parte de las etiquetas que tienen las misma utilidad que la configuración de despliegue vista anteriormente, cabe destacar los siguientes parámetros:

- **name** (línea 6): El nombre del servicio que usaremos para llamarlo dentro de la plataforma.
- **ports** (línea 9): El puerto que expondrá el servicio y al puerto de los *pods* que atacará.
- **selector** (línea 14): Los *pods* a los que atacará el servicio. En este caso los *pods* pertenecientes al configuración de despliegue *stream-predictor*.
- **sessionAffinity** (línea 16): Si quisieramos establecer alguna afinidad, por ejemplo a nivel de *IP*, para que un mismo origen ataque siempre a un mismo *Pod*. Hay que recordar que los servicios en *Openshift* (y *Kubernetes*) actúan como un balanceador entre los *pods*.

9.2. Servicios *Tensorflow Serving*

En el caso del servicio de Tensorflow Serving, mostraremos únicamente la configuración de despliegue, aunque también consta de un servicio para que podamos realizar llamadas *HTTP* sobre el mismo desde la plataforma de *Openshift*. En el caso de que se quisiera acceder al servicio desde fuera de la plataforma será necesario recurrir las *Routes* de *Openshift*.

La configuración de despliegue de *tf-BajaFactura* es la siguiente:

```

1 apiVersion: apps.openshift.io/v1
2 kind: DeploymentConfig
3 metadata:
4   labels:

```

```
5     app: tfm-mgm
6     appName: tf-bajafactura-model
7     appTypes: tensorflow-serving-s2i
8     appid: tf-serving-tf-bajafactura-model
9     name: tf-bajafactura-model
10    namespace: nbia-prod
11
12  spec:
13    replicas: 1
14    revisionHistoryLimit: 10
15    selector:
16      deploymentconfig: tf-bajafactura-model
17    strategy:
18      activeDeadlineSeconds: 21600
19      rollingParams:
20        intervalSeconds: 1
21        maxSurge: 25%
22        maxUnavailable: 25%
23        timeoutSeconds: 600
24        updatePeriodSeconds: 1
25      type: Rolling
26    template:
27      metadata:
28        labels:
29          app: tfm-mgm
30          appName: tf-bajafactura-model
31          appTypes: tensorflow-serving-s2i
32          appid: tf-serving-tf-bajafactura-model
33          deploymentconfig: tf-bajafactura-model
34
35  spec:
36    containers:
37      - env:
38        - name: PORT
39          value: '8501'
40        - name: MODEL_NAME
41          value: bajafactura
42        - name: RUN_OPTIONS
```

```
41   image: >-
42     docker-registry.default.svc:5000/nbia-prod/
43     tf-bajafactura-model:latest
44   imagePullPolicy: Always
45   livenessProbe:
46     failureThreshold: 10
47     httpGet:
48       path: /v1/models/bajafactura
49       port: 8501
50       scheme: HTTP
51     initialDelaySeconds: 20
52     periodSeconds: 30
53     successThreshold: 1
54     timeoutSeconds: 5
55   name: tf-bajafactura-model
56   ports:
57     - containerPort: 8501
58       protocol: TCP
59   readinessProbe:
60     failureThreshold: 1
61     httpGet:
62       path: /v1/models/bajafactura
63       port: 8501
64       scheme: HTTP
65     initialDelaySeconds: 20
66     periodSeconds: 10
67     successThreshold: 1
68     timeoutSeconds: 5
69   resources:
70     limits:
71       cpu: '2'
72       memory: 2Gi
73     requests:
74       cpu: '1'
75       memory: 1Gi
76   terminationMessagePath: /dev/termination-log
```

```

77     terminationMessagePolicy: File
78     dnsPolicy: ClusterFirst
79     restartPolicy: Always
80     schedulerName: default-scheduler
81     terminationGracePeriodSeconds: 30
82   test: false
83   triggers:
84     - imageChangeParams:
85       automatic: true
86       containerNames:
87         - tf-bajafactura-model
88       from:
89         kind: ImageStreamTag
90         name: 'tf-bajafactura-model:latest'
91         namespace: nbia-prod
92       type: ImageChange
93     - type: ConfigChange

```

En este caso solo comentaremos los parámetros que varían con respecto al despliegue de los servicios *Kafka Streams*:

- **strategy** (línea 16): En este caso la estrategia en el caso de que exista un nuevo despliegue se denomina *Rolling*. Esto implica, de forma muy resumida, que el nuevo *deployment* se hará de manera paulatina, no eliminando los *deployment* anteriores hasta que no se hayan realizado los nuevos. Esto garantiza que no exista una pérdida de servicio en los nuevos despliegues.
- **image** (línea 41): La imagen usada en este caso es la imagen de *docker* de TensorFlow Serving con nuestro modelo cargado, esta imagen se encuentra almacenada en el repositorio de la plataforma.
- **livenessProbe** (línea 45): Comprueba que el contenedor sigue con vida, de no ser así durante el número de intentos establecido se reiniciará el contenedor. En este caso la comprobación es una petición *HTTP* al *endpoint* de estado del modelo.
- **readinessProbe** (línea 59): Antes de enviar tráfico a un *pod*, este debe estar en estado *ready*. Con este parámetro establecemos que el *pod* no se encuentre *ready* hasta que no se haya verificado el *endpoint* de estado del modelo.

Aunque existen otros parámetros que varían como las variables de entornos o los recursos definidos el funcionamiento es igual a la configuración de despliegue definida anteriormente.

9.3. Monitorización

Los despliegues vistos hasta ahora nos permiten tener funcionando nuestro sistema, sin embargo, no debemos olvidarnos de la monitorización necesaria, que ya describimos en el capítulo 7, para verificar el correcto funcionamiento del mismo.

Dentro de Openshift la monitorización esta formada por dos despliegues diferentes:

- **Logstash**: Podemos identificarlo en la figura 9.1 con el nombre **lt-tfm-calls** y será el encargado de llevar a la capa de servicio no solo la monitorización de todo el sistema, si no también los datos de clasificación de llamadas y de realizar las transformaciones necesarias.
- **Metricbeat**: Podemos identificarlo en la figura 9.1 con el nombre **jmx-beat** y será el encargado de extraer las métricas de Jolokia de los servicios de *Kafka Streams*.

A continuación veremos la configuración de ambos despliegos.

9.3.1. Logstash

Al igual que en los casos anteriores, vemos únicamente la configuración del despliegue, pudiendo encontrar el resto de recursos en el anexo ??.

La configuración de despliegue es la siguiente:

```
1 apiVersion: apps.openshift.io/v1
2 kind: DeploymentConfig
3 metadata:
4   labels:
5     app: tfm-mgm
6     service: lt-tfm-calls
7     version: '1.0'
8   name: lt-tfm-calls
9   namespace: nbia-prod
10 spec:
11   replicas: 1
12   revisionHistoryLimit: 10
```

```
13 selector:
14   deploymentconfig: lt-tfm-calls
15 strategy:
16   activeDeadlineSeconds: 21600
17   resources: {}
18   rollingParams:
19     intervalSeconds: 1
20     maxSurge: 25%
21     maxUnavailable: 25%
22     timeoutSeconds: 600
23     updatePeriodSeconds: 1
24 type: Rolling
25 template:
26   metadata:
27     labels:
28       app: tfm-mgm
29       deploymentconfig: lt-tfm-calls
30       service: lt-tfm-calls
31       version: '1.0'
32 spec:
33   containers:
34     - env:
35       - name: PIPELINE
36         value: tfm-calls
37       - name: LS_JAVA_OPTS
38         value: '-Xmx2g'
39       - name: USER_LOGSTASH
40         valueFrom:
41           secretKeyRef:
42             key: username
43             name: logstash-internal-user
44       - name: PASS_LOGSTASH
45         valueFrom:
46           secretKeyRef:
47             key: password
48             name: logstash-internal-user
```

```
49     image: 'docker-registry.default.svc:5000/openshift/logstash:7.4.2'
50   imagePullPolicy: IfNotPresent
51   livenessProbe:
52     failureThreshold: 10
53     httpGet:
54       path: /
55       port: 9600
56       scheme: HTTP
57     initialDelaySeconds: 300
58     periodSeconds: 30
59     successThreshold: 1
60     timeoutSeconds: 5
61   name: lt-tfm-calls
62   ports:
63     - containerPort: 5010
64       protocol: TCP
65     - containerPort: 9600
66       protocol: TCP
67   readinessProbe:
68     failureThreshold: 1
69     httpGet:
70       path: /
71       port: 9600
72       scheme: HTTP
73     initialDelaySeconds: 120
74     periodSeconds: 10
75     successThreshold: 1
76     timeoutSeconds: 5
77   resources:
78     limits:
79       cpu: '1'
80       memory: 2200Mi
81     requests:
82       cpu: 500m
83       memory: 2Gi
84   terminationMessagePath: /dev/termination-log
```

```

85      terminationMessagePolicy: File
86      volumeMounts:
87        - mountPath: /usr/share/logstash/config/logstash.yml
88          name: calls-config
89          subPath: logstash.yml
90      dnsPolicy: ClusterFirst
91      restartPolicy: Always
92      schedulerName: default-scheduler
93      terminationGracePeriodSeconds: 30
94      volumes:
95        - configMap:
96          defaultMode: 420
97          name: calls-config
98          name: calls-config
99      test: false
100     triggers:
101       - type: ConfigChange

```

Algunas características que podemos observar diferentes a las vistas actualmente son:

- ***Secrets*** (líneas 40 y 45): En este despliegue vemos que las variables globales para la autenticación se cargan de un recurso de *OCP* denominado *Secret*, utilizado para almacenar información sensible.
- ***image*** (línea 49): En este caso la imagen no contiene ningún código propio como en resto de despliegues vistos anteriormente. Se trata de la imagen oficial de *Elastic*. El código a ejecutar (*pipeline*) se almacena de forma centralizada en *Elasticsearch* y utilizamos la variable de entorno *PIPELINE* y los ficheros de configuración (*configmaps*) para acceder a la información.
- ***resources*** (línea 77): En este caso vemos que los recursos solicitados son mayores que en el resto de despliegues. Esto se debe a que el *Heap* que tiene configurado la imagen de *Logstash* por defecto es mayor, podría modificarse para optimizar el uso de recursos, pero no hemos tenido esa necesidad.

9.3.2. Metricbeat

El último despliegue que nos queda por ver, es probablemente el más simple, el de *Metricbeat*, el agente encargado de extraer las métricas de *Jolokia* de los servicios de *Kafka Streams*. Este

despliegue no tiene asociado ningún servicio, ya que no será llamado por otros componentes.

La configuración de despliegue de *Metricbeat* es la siguiente:

```
1 apiVersion: apps.openshift.io/v1
2 kind: DeploymentConfig
3 metadata:
4   labels:
5     app: tfm-mgm
6     deploymentconfig: jmx-beat
7     service: jmx-beat
8   name: jmx-beat
9   namespace: nbia-prod
10 spec:
11   replicas: 1
12   selector:
13     deploymentconfig: jmx-beat
14   strategy:
15     activeDeadlineSeconds: 21600
16     resources: {}
17   rollingParams:
18     intervalSeconds: 1
19     maxSurge: 25%
20     maxUnavailable: 25%
21     timeoutSeconds: 600
22     updatePeriodSeconds: 1
23   type: Rolling
24 template:
25   metadata:
26     labels:
27       app: tfm-mgm
28       deploymentconfig: jmx-beat
29       project: tfmmgm
30       service: jmx-beat
31   spec:
32     containers:
33       - env:
34         - name: ELASTICSEARCH_USERNAME
```

```
35      valueFrom:  
36          secretKeyRef:  
37              key: username  
38              name: logstash-internal-user  
39      - name: ELASTICSEARCH_PASSWORD  
40          valueFrom:  
41              secretKeyRef:  
42                  key: password  
43                  name: logstash-internal-user  
44  image: >-  
45      docker-registry.default.svc:5000/nbia-prod/metricbeat:7.4.2  
46  imagePullPolicy: Always  
47  name: jmx-beat  
48  resources:  
49      limits:  
50          cpu: '1'  
51          memory: 512Mi  
52  terminationMessagePath: /dev/termination-log  
53  terminationMessagePolicy: File  
54  volumeMounts:  
55      - mountPath: /usr/share/metricbeat/metricbeat.yml  
56          name: calls-config  
57          subPath: metricbeat.yml  
58      - mountPath: /usr/share/metricbeat/modules.d/jolokia.yml  
59          name: calls-config  
60          subPath: jolokia.yml  
61  dnsPolicy: ClusterFirst  
62  restartPolicy: Always  
63  schedulerName: default-scheduler  
64  terminationGracePeriodSeconds: 30  
65  volumes:  
66      - configMap:  
67          defaultMode: 292  
68          name: calls-config  
69          name: calls-config  
70  test: false
```

```

71   triggers:
72     - type: ConfigChange
73     - imageChangeParams:
74       automatic: true
75       containerNames:
76         - jmx-beat
77       from:
78         kind: ImageStreamTag
79         name: 'metricbeat:7.4.2'
80       namespace: nbia-prod
81     type: ImageChange

```

Todos los parámetros vistos en este despliegue deberían resultarnos ya familiares. La imagen de *docker* usada en este caso también se trata de la imagen oficial desarrollada por *Elastic*.

9.4. S2I

Los despliegues en *OCP* descritos a lo largo de este capítulo podrían clasificarse en dos: Aquellos que son imágenes oficiales con alguna configuración propia y aquellos que contienen código fuente o binarios propios. En el primer grupo se encuentra el agente *Metricbeat* y *Logstash* (aunque ejecuta código propio no está almacenado en la imagen), mientras que al segundo grupo pertenecen los microservicios de *Kafka Streams* y el modelo de *Tensorflow Serving*. A partir de ahora, nos centraremos en este segundo grupo.

La opción más simple para tener una imagen con nuestro código sería generarlas de forma individual y subirlas al repositorio de imágenes de la plataforma, sin embargo esta tarea se volvería algo tediosa conforme el número de servicios crezca, además presentaría dificultades en su mantenimiento y en su integración (como veremos más adelante) con flujos de integración y despliegue continuos. Es por eso que utilizaremos una característica de *OCP* denominada *S2I* (*Source to Image*), que nos permitirá a partir de una imagen base (a partir de ahora imagen S2I) generar una nueva imagen personalizada con nuestra aplicación.

El proceso de crear la imagen de nuestra aplicación a partir de una imagen S2I y nuestro código o binarios se denomina *Build* y al igual que con la configuración de despliegue para los despliegues *OpenShift*, nos proporciona una configuración de *Build* para controlar las versiones y los cambios que se realizan en un mismo tipo de *Build*. A continuación veremos como hemos abordado los casos de *Kafka Streams* y *Tensorflow Serving*.

9.4.1. *Kafka Streams*

En este caso tenemos dos opciones a la hora de crear nuestra imagen de aplicación mediante S2I, partir del código fuente o partir de un binario (un jar generado a partir del código fuente). Hemos optado por esta segunda opción para dejar la compilación y tests fuera de la plataforma *OCP*.

La imagen *S2I* utilizada es la imagen oficial proporcionada por RedHat para Java. Y a continuación podemos ver como quedaría nuestra configuración de *Build*:

```

1  apiVersion: build.openshift.io/v1
2  kind: BuildConfig
3  metadata:
4    labels:
5      app: tfm-mgm
6    name: topic-model-streaming
7    namespace: nbia-prod
8  spec:
9    failedBuildsHistoryLimit: 5
10   output:
11     to:
12       kind: ImageStreamTag
13       name: 'topic-model-streaming:latest'
14   runPolicy: Serial
15   source:
16     binary: {}
17     type: Binary
18   strategy:
19     sourceStrategy:
20       from:
21         kind: ImageStreamTag
22         name: 'redhat-openjdk18-openshift:1.4'
23         namespace: openshift
24       type: Source
25   triggers: []

```

Algunos aspectos relevantes de la configuración de *Build* son:

- ***output*** (línea 10): Nos indica la imagen destino de aplicación que se creará. En nuestro

caso la imagen es común ya que todas las clases están embebidas en un mismo binario, por eso posee un valor fijo, pero podría tener un valor variable. La imagen destino debe estar definida en la plataforma.

- **from** (línea 20): La imagen *S2I* que se utilizará para crear la imagen de aplicación.
- **triggers** (línea 25): Al tratarse de un *S2I* binario no existen disparadores por lo que el *Build* debe iniciarse de forma externa proporcionando los binarios.

9.4.2. *Tensorflow Serving*

En este caso no tenemos elección entre realizar el *build* con fuente o binario, debido a que los modelos que tenemos de *Tensorflow* son binarios. La aproximación, por tanto, será la misma que en el apartado anterior.

Sin embargo, en este caso no existe ninguna imagen oficial *S2I* para *Tensorflow Serving* por lo que nos hemos visto obligados a crear nuestra propia imagen *S2I* a partir de la imagen oficial de *docker* de *Tensorflow Serving*.

El proceso de creación de una imagen *S2I* consta, a grandes rasgos, de los siguientes pasos:

- **Assemble**: Crear los scripts necesarios para una vez recibido los ficheros, colocarlos en la ruta adecuada y construir con ellos la imagen de aplicación.
- **run**: Establecer como debe comportarse la imagen de aplicación.
- **usage**: Recopilar el modo de uso de la imagen *S2I*.

Todo el código de creación de la imagen así como el build de *Tensorflow Serving* (prácticamente idéntico al anterior) puede encontrarse en el anexo ??.

Parte IV

Conclusiones: mantenimiento y futuros trabajos

Capítulo 10

DevOps

Una vez hemos llegado a esta parte del documento ya hemos realizado el análisis completo de los datos y hemos dejado el modelo en un entorno productivo clasificando llamadas en *real-time*; sin embargo sería muy atrevido dar el proyecto por finalizado.

Probablemente surgirán nuevos datos, dispondremos de nuevas etiquetas, nuestro *software* se degradará, crearemos nuevos y mejores modelos con ideas más o menos brillantes, etc. Este universo cambiante provocará a que se modificarán y/o se crearán nuevos *software* y modelos con bastante frecuencia que será necesario llevar de nuevo a producción.

Este capítulo tiene objetivo definir los mecanismos y la metodología que hemos establecido para poder reaccionar con gran agilidad ante el cambio, algo esencial hoy en día. En la sección 10.1 haremos una introducción a la metodología DevOps que acuñaremos en nuestro proyecto, posteriormente pondremos foco en algunos aspectos importantes: La degradación del modelo (sección 10.2) y la integración y el despliegue continuos (sección 10.3).

10.1. Introducción DevOps

10.2. Degradación del modelo

Aunque el *software* desarrollado también es propenso de degradarse con el paso del tiempo, si hay algo que tenemos seguro que se va a degradar en este universo cambiante es el modelo. Nuestro modelo se alimenta con las transcripciones de los clientes que están basadas en el servicio que ofrecemos en cada momento, en las inquietudes de nuestros clientes y otros muchos factores externos que pueden influir en la temática de las conversaciones que recibimos.

La certeza en la degradación de nuestro modelo hace que tengamos monitorizar constantemente el rendimiento del mismo, en el caso del modelo supervisado que hemos llevado a producción nos lleva a seguir testeándolo constantemente introduciendo en el flujo llamadas de

control etiquetadas para medir su eficiencia.

10.3. Integración y despliegue continuos

Bibliografía

- [1] O. Abdelwahab and A. Elmaghraby. Deep learning based vs. markov chain based text generation for cross domain adaptation for sentiment classification. In *2018 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 252–255, Julio 2018.
- [2] Toni Lozano Bagén Anna Bosch Rué, Jordi Casas Roma. *Deep Learning Principios y Fundamentos*. 2018.
- [3] David M. Blei, Michael I. Jordan, Thomas L. Griffiths, and Joshua B. Tenenbaum. Hierarchical topic models and the nested chinese restaurant process. In *Proceedings of the 16th International Conference on Neural Information Processing Systems*, NIPS'03, pages 17–24, Cambridge, MA, USA, 2003. MIT Press.
- [4] David M. Blei and John D. Lafferty. Correlated topic models. In *Proceedings of the 18th International Conference on Neural Information Processing Systems*, NIPS'05, pages 147–154, Cambridge, MA, USA, 2005. MIT Press.
- [5] David M. Blei and John D. Lafferty. Dynamic topic models. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 113–120, New York, NY, USA, 2006. ACM.
- [6] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, 2003.
- [7] Pete Chapman, Julian Clinton, Randy Kerber, Thomas Khabaza, Thomas Reinartz, Colin Shearer, and Rudiger Wirth. Crisp-dm 1.0 step-by-step data mining guide. Technical report, The CRISP-DM consortium, Agosto 2000.
- [8] Michael Cox and David Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings of the 8th Conference on Visualization '97*, VIS '97, pages 235–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

- [9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
- [10] Jesús Domínguez. De lambda a kappa: evolución de las arquitecturas big data. <https://www.paradigmadigital.com/techbiz/de-lambda-a-kappa-evolucion-de-las-arquitecturas-big-data/>, Abril 2018. Último acceso 2019-10-13.
- [11] Christine Fellbaum. *WordNet: an electronic lexical database*. MIT Press, 1998.
- [12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.
- [13] Yoav Goldberg. *Neural network methods in natural language processing*. Morgan & Claypool publishers, 2017.
- [14] Wenming Guo, Lihong Liang, and Tianlang Deng. Topic mining for call centers based on a-lda and distributed computing. *Concurrency and Computation: Practice and Experience*, 29(3):e3776, 2017. e3776 CPE-15-0479.R1.
- [15] Tom Hope, Itay Lieder, and Yehezkel S. Resheff. *Learning TensorFlow: a guide to building deep learning systems*. O'Reilly Media, 2017.
- [16] Jay Kreps. Questioning the lambda architecture. <https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e>, Julio 2014. Último acceso 2019-10-14.
- [17] Douglas Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group, February 2001.
- [18] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, pages II–1188–II–1196. JMLR.org, 2014.
- [19] P. Li, Y. Yan, Chaomin Wang, Zhijie Ren, Pengyu Cong, Huixin Wang, and Junlan Feng. Customer voice sensor: A comprehensive opinion mining system for call center conversation. In *2016 IEEE International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pages 324–329, Julio 2016.

- [20] F. Long, K. Zhou, and W. Ou. Sentiment analysis of text based on bidirectional lstm with multi-head attention. *IEEE Access*, 7:141960–141969, 2019.
- [21] A. S. M. Ashique Mahmood. *Literature Survey on Topic Modeling*. 2013.
- [22] Nathan Marz. How to beat the cap theorem. <http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html>, Octubre 2011. Último acceso 2019-10-14.
- [23] Tomas Mikolov, Kai Chen, Greg S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.
- [24] Confluent. Streams Monitoring. <https://docs.confluent.io/current/streams/monitoring.html>. Último acceso 2019-12-09.
- [25] M. Morchid, R. Dufour, M. Bouallegue, and G. Linarès. Author-topic based representation of call-center conversations. In *2014 IEEE Spoken Language Technology Workshop (SLT)*, pages 218–223, Diciembre 2014.
- [26] Syed Sadat Nazrul. Cap theorem and distributed database management systems. <https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e>, Abril 2018. Último acceso 2019-10-14.
- [27] Michael Nguyen. Illustrated guide to lstm's and gru's: A step by step explanation. <https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21>, Septiembre 2018. Último acceso 2019-10-13.
- [28] Christopher Olah. Understanding lstm networks – colah's blog. <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>, Agosto 2015. Último acceso 2019-10-13.
- [29] Stacey Ronaghan. Deep learning: Common architectures. <https://medium.com/@srnghn/deep-learning-common-architectures-6071d47cb383>, Agosto 2018. Último acceso 2019-10-13.
- [30] Michal Rosen-Zvi, Thomas Griffiths, Mark Steyvers, and Padhraic Smyth. The author-topic model for authors and documents. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, UAI '04, pages 487–494, Arlington, Virginia, United States, 2004. AUAI Press.

- [31] Jullian Sepúlveda. *Metodología para estimar el impacto que generan las llamadas realizadas en un call center en la fuga de los clientes utilizando técnicas de text mining*. PhD thesis, Universidad de Chile, 2015.
- [32] Tensorflow Serving. <https://www.tensorflow.org/tfx/guide/serving>. Último acceso 2019-12-10.
- [33] Apache Kafka. Kafka Streams. <https://kafka.apache.org/documentationstreams/>. Último acceso 2019-12-10.
- [34] A. M. Turing. *Computing machinery and intelligence*. Blackwell for the Mind Association, 1950.
- [35] Kevin Warwick and Huma Shah. *Turing's imitation game: conversations with the unknown*. Cambridge University Press, 2016.
- [36] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.