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, 11 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

Índice de figuras

Parte I

Introdución: 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:

- ***S**pecific*: Deben plantearse de una forma detallada y concreta.
- ***M**easurable*: Deben poder medirse con facilidad.
- ***A**chievable*: Deben ser objetivos realistas.
- ***R**elevant*: Tienen que ser relevantes para la empresa y ofrecernos un beneficio claro.
- ***T**imely*: 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 cuasi 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:

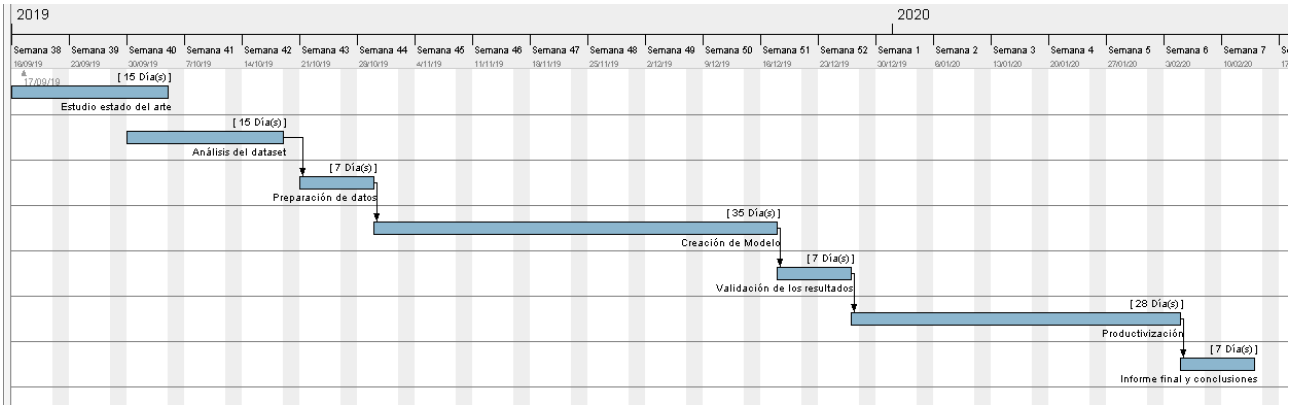

Figura 1.1: Diagrama de Gantt

- **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 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*** ([6]), 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.

1.5. Estructura del documento

TODO Hacer repaso breve de los apartados del documento final.

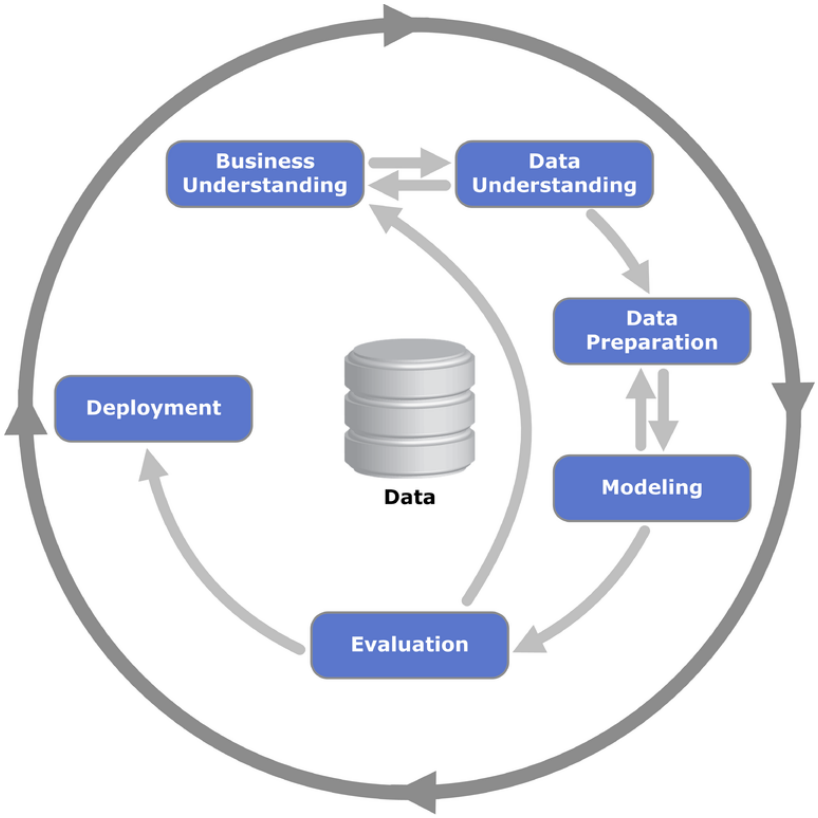

Figura 1.2: Fases del modelo CRISP-DM

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 [26]. 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 [27].

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 encontraríamos 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 [12] 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 unicamente 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* [10], 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 [5]. 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 [2], surgió una **variante jerárquica** que permitía representar los *topics* jerárquicamente. En 2006 en [4] 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 [3] 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 [23] 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 [16].

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 multilayer 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.

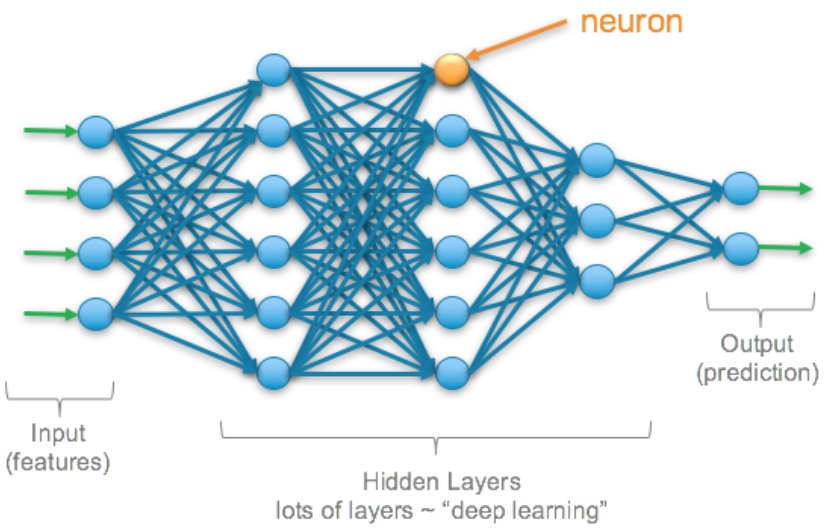

Figura 2.1: Ejemplo de arquitectura MLP. Fuente [22]

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.

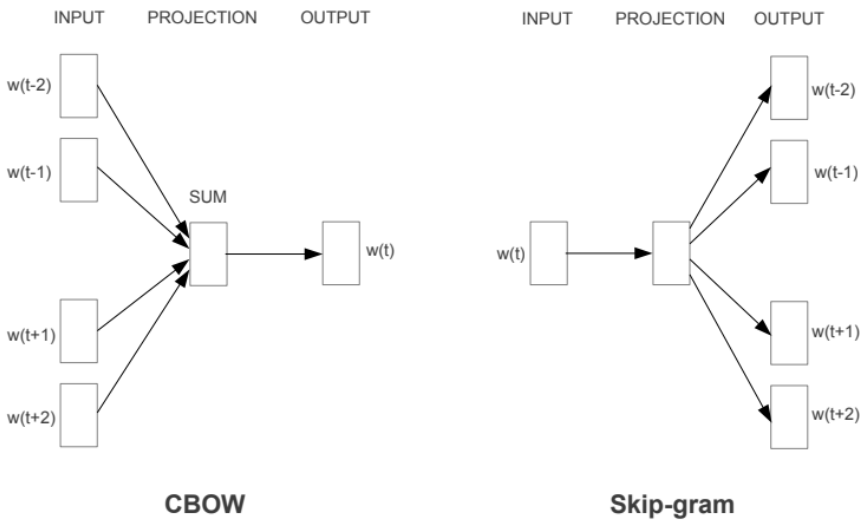

Figura 2.2: Arquitecturas CBOW y Skip-gram. Fuente [18]

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 [18] . 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 ***C**onvolutional **N**eural **N**etworks*) 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.

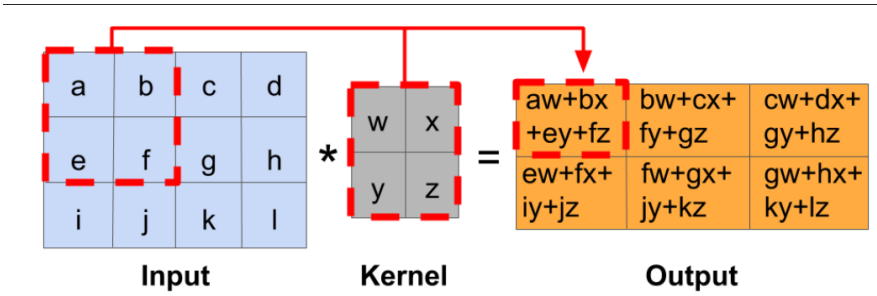

Figura 2.3: Ejemplo de una convolución de dos dimensiones. Fuente [1]

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*.

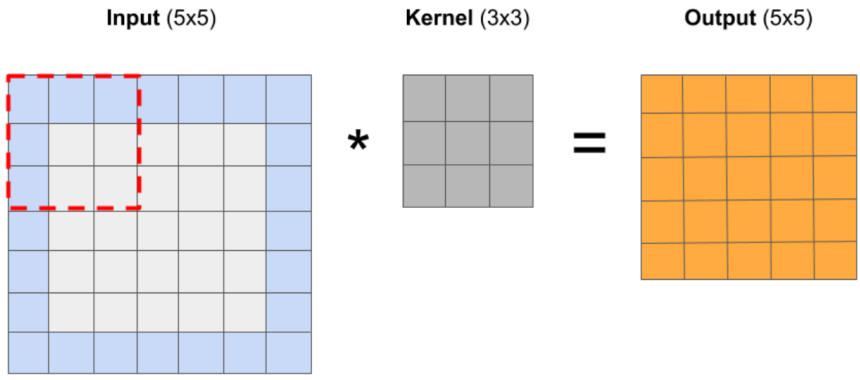

Figura 2.4: Ejemplo de aplicación de *zero padding* para mantener la dimensionalidad. Fuente [1]

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.

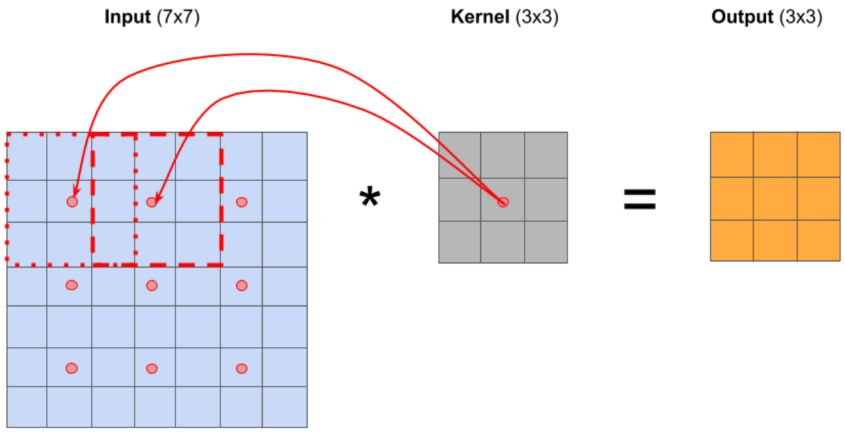

Figura 2.5: Ejemplo de aplicación de convolución por pasos para reducir la dimensionalidad. Fuente [1]

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 (*polling* 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-polling*). 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.

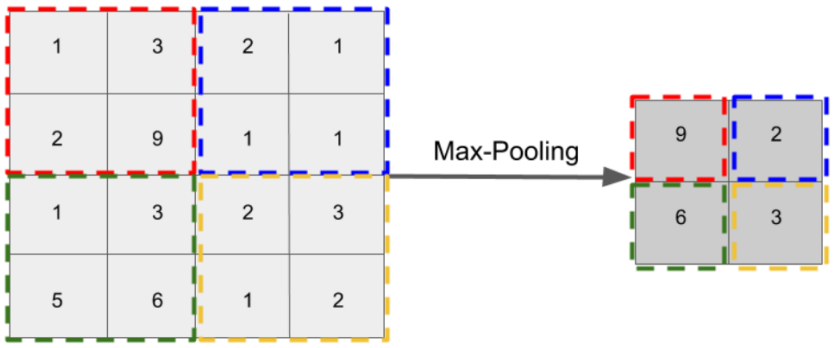

Figura 2.6: Ejemplo de aplicación de *max-pooling*. Fuente [1]

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.

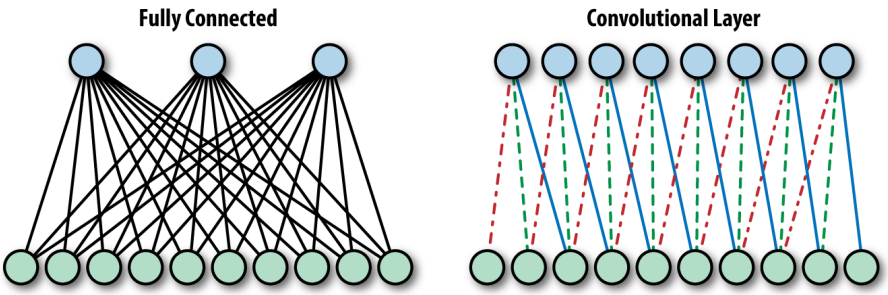

Figura 2.7: Comparación capa tradicional totalmente conectada con capa convolucional. Fuente [13]

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 [1]. 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 [12].

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 ***R**ecurrent **N**eural **N**etworks*). 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).

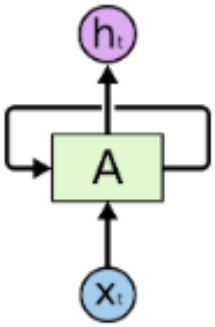

Figura 2.8: Ejemplo RNN. Fuente [21]

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.

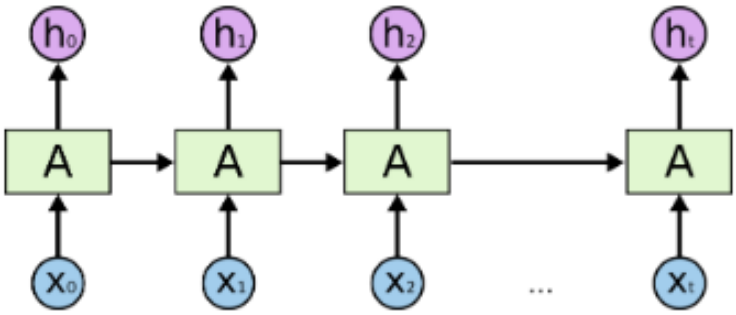

Figura 2.9: Ejemplo RNN "desenrollada". Fuente [21]

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 que 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

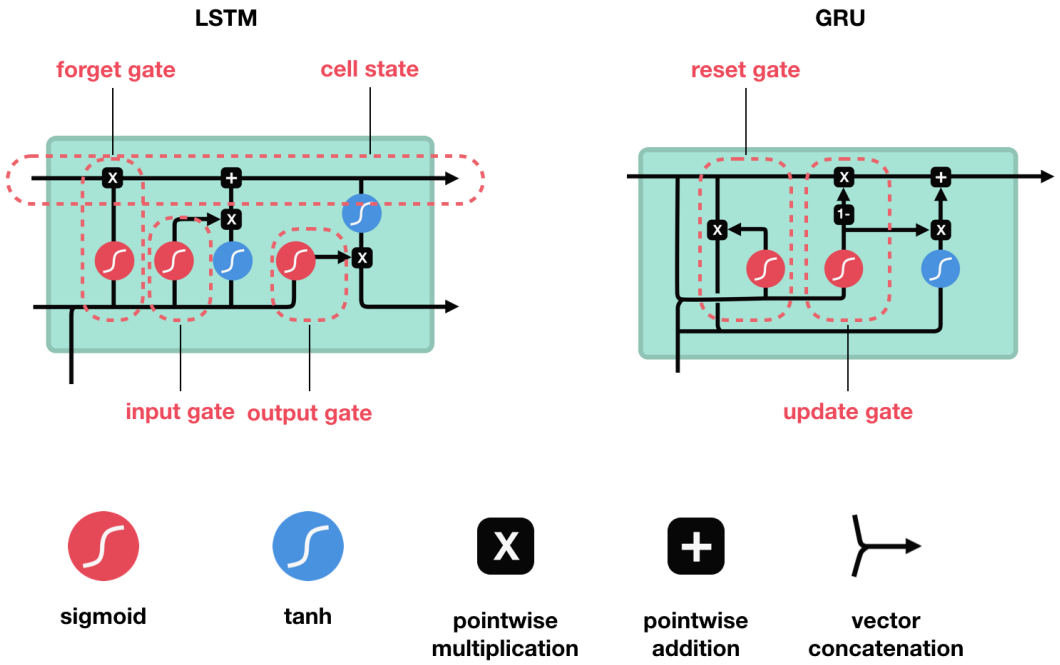

Figura 2.10: Arquitectura celdas LSTM y GRU. Fuente [20]

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 [**?**] 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 [**?**]. 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* [**?**] 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 [7], 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*" [15] en el que se hacía referencia a las ya "famosas" tres *V*s:

- **V**olumen: Cada vez los volúmenes de datos son mayores.

- **V**elocidad: Es cada vez mayor la velocidad con la que se generan los datos.

- **V**ariedad: 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 [11] su *"Google File System"* (GFS) y un año después *Map Reduce* [8], 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.

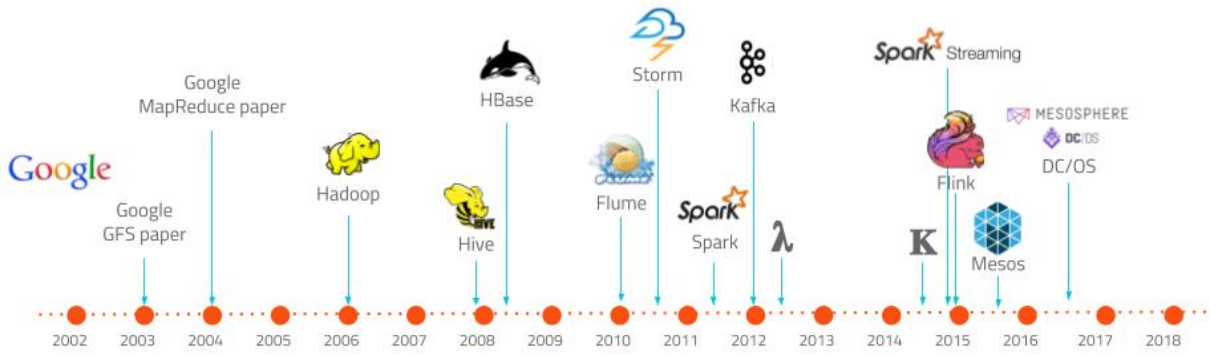

Figura 2.11: Evolución del *Big Data*. Fuente [9]

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 esta 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 alarmar) 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*" [17].

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

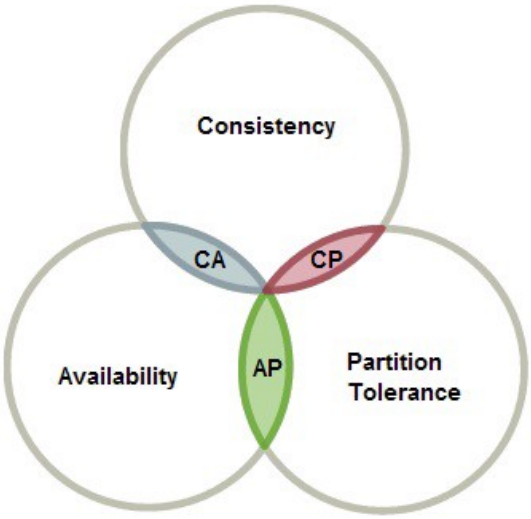

Figura 2.12: Teorema CAP. Fuente [19]

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 (***C**reated, **R**ead, **U**pdate, **D**elete*) 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 3.1. 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*.

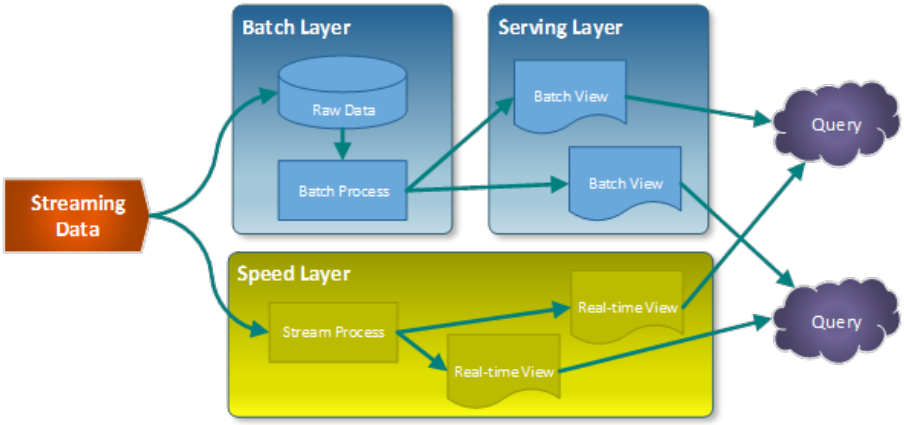

Figura 2.13: Arquitectura Lambda definida por Nathan Marz. Fuente [9]

2.3.2.2. Arquitectura Kappa κ

En su artículo "Questioning the Lambda Architecture"[14], 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 *bacth*) y opina que es posible realizar el mismo procesamiento, incluso reprocesar el histórico de datos, en la capa *realtime*.

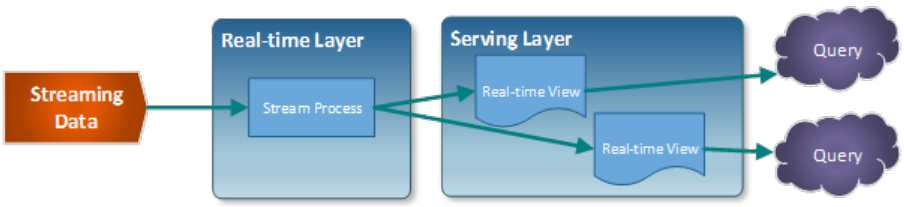

Figura 2.14: Arquitectura Kappa definida por Jay Kreps. Fuente [9]

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" [**?**]: 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" [**?**]: 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" [**?**]: 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" [**?**]: 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 un diseño esquemático de la solución planteada, este diseño será abordado en primer lugar desde un punto de vista lógico, de acuerdo a las necesidades del proyecto, posteriormente aterrizaremos esta arquitectura con tecnologías concretas que nos ayuden a conseguir el objetivo deseado.

3.1. Arquitectura

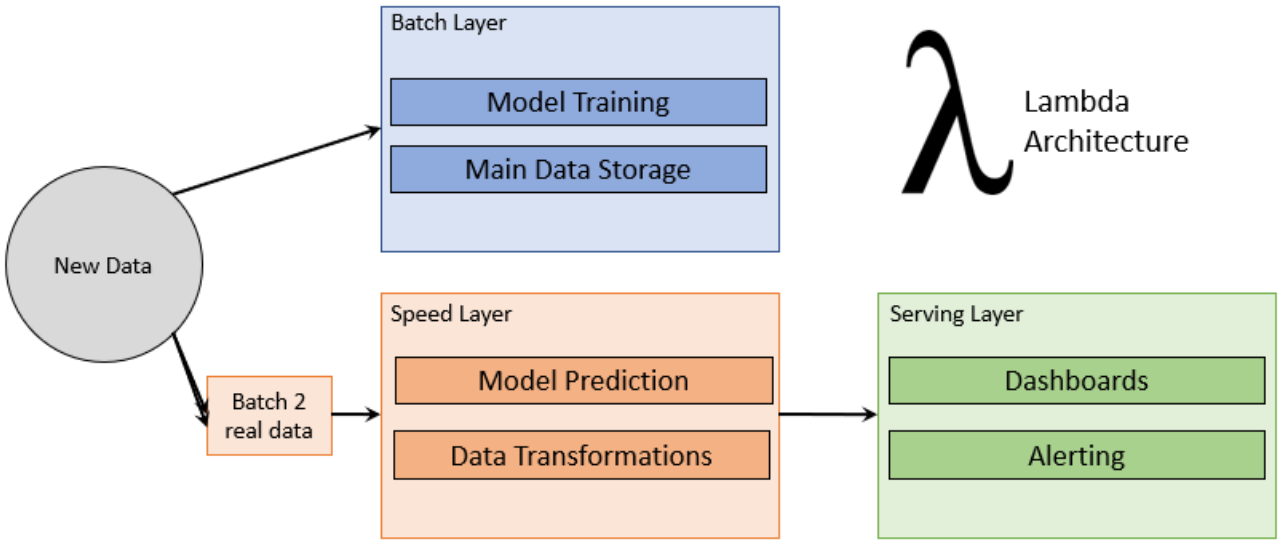

Figura 3.1: Arquitectura Lambda propuesta

En este apartado definiremos la arquitectura desde un punto de vista lógico, esta arquitectura debe responder a los objetivos propuestos para cumplir con las necesidades de negocio existentes.

En líneas generales, podemos ver la arquitectura de nuestro sistema como una arquitectura *Lambda* en la que disponemos de una capa *batch*, una capa rápida o *real-time* y una última capa de servicio.

La capa batch será la encargada de entrenar el modelo a través de los datos de las llamadas, la capa rápida obtendrá los *topics* de las llamadas en tiempo real usando el modelo previamente entrenado y por último la capa de servicio será la encargada de mostrar estos datos al usuario mediante cuadros de mando.

En la Figura 3.1 podemos ver un esquema general de nuestra arquitectura. A continuación definimos con más detalle cada una de las capas del modelo.

3.1.1. Capa Batch

El *core* del proyecto que abordamos es el modelo, encargado de extraer los *topics* de las transcripciones de llamadas al servicio de atención al cliente. Este modelo debe entrenarse usando un histórico suficientemente amplio.

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.

3.1.2. Capa Real-Time

La *speed layer* de nuestro proyecto será la encargada de recibir los datos en tiempo real, las llamadas serán publicadas en un bus de eventos y estos eventos serán consumidos por una capa de procesamiento que será la encargada de aplicar el modelo entrenado en la capa *batch* a los nuevos datos. Los *topics* resultantes de cada llamada serán publicados de nuevo en este bus para poder ser ingestados posteriormente a una BBDD NO-SQL, que será la encargada de proporcionar la información a la capa de servicio.

Aprovecharemos también las características de la BBDD NoSQL para, mediante un módulo de *Machine Learning*, detectar anomalías en series temporales y poder alarmar en el caso de que un *topic* concreto se dispare en algún momento.

Debido a la situación actual, las llamadas no se ingestan en *real-time* si no que se procesan en mini *batches* cada cierto tiempo. Es muy probable que este escenario cambie en el futuro por lo que se construirá un elemento de entrada a la capa rápida que transformará el mini-batch en eventos conforme vayan ejecutándose (este elemento puede observarse en la figura 3.1 como *Batch 2 real data*). Esta pieza será suprimida una vez las llamadas sean recibidas en tiempo real.

3.1.3. Capa de Servicio

Una vez almacenados los datos en la BBDD debemos construir un frontal que muestre al usuario el número de llamadas analizadas, el modelo utilizado y principalmente la evolución de los *topics* a lo largo del tiempo.

Lo ideal es que esta capa de servicio se vaya actualizando en tiempo cuasi-real y permita a los diferentes usuarios realizar cuadros de mando personalizados según sus necesidades.

Otro requisito esencial en este tipo de proyectos es que la información este disponible vía API-REST para poder ser consumida por terceros.

3.2. Integración y Despliegue Continuos

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 la tecnologías, será necesario añadir mejoras o modificaciones en nuestro desarrollo.

Por estos motivos es conveniente 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 del usuario.

Aunque este apartado queda fuera de la implementación inicial del proyecto es imprescindible llevarlo a cabo para que este sea sostenible a lo largo del tiempo.

TODO (Posible ampliar con apuntes sobre el ciclo de vida de los datos y contar beneficios de un despliegue continuo)

3.3. 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 dado 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.

3.3.1. Capa batch

- **Spark**: Es un framework de computación en clúster. Este *framework* se encuentra desplegado sobre un clúster Hadoop, específicamente una distribución HortonWorks, y posee librerías específicas para Machine Learning como MLLIB.
- **Tensorflow** sobre GPUs: Para el entrenamiento de los modelos también disponemos de un cluster con GPUs sobre el que podremos correr código usando la librería de Google Tensorflow para entrenar nuestros modelos.

3.3.2. Capa Real-Time

- **Kafka**: Es el *core* de la capa rápida, se trata de un bus de evento distribuido a través del cual se realizara la ingesta o publicación de los eventos (llamadas). Las diferentes capas de procesamiento que requieran estos eventos se suscribirán a este Bus.
- **Microservicios**: En nuestra capa Real Time construiremos diferentes microservicios en la capa de procesamiento, estos microservicios serán por ejemplo los encargados de devolver los *topics* de cada llamada a partir de una llamada API REST. Estos microservicios correran sobre contenedores en una plataforma Openshift.
- **Kafka Stream y KSQL**: A la hora de procesar la información en eventos ingestada en nuestro Bus Kafka disponemos de dos herramientas muy potentes que son Kafka Stream y KSQL. El primero consiste en una serie de librerías que nos permiten construir aplicaciones y microservicios cuyo origen y destino sean un Bus Kafka. KSQL en cambio es un motor SQL aplicable a eventos que nos llegan mediante *streaming*.
- **Logstash**: Una vez hayamos extraído los *topics* correspondientes a cada llamada o evento, tendremos que ingestar esta información en nuestra BBDD, que en este caso será Elasticsearch. Logstash nos permitirá leer de Kafka, realizar las transformaciones necesarias y volcar la información resultante en Elasticsearch.
- **Elasticsearch**: 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 y realizar consultas y agregaciones sobre cualquier campo. Entre las características que podemos aprovechar de Elasticsearch para nuestro objetivo están:
  - Ingesta en tiempo casi real.
  - Consulta en tiempo casi 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.

3.3.3. Capa Servicio

- **Elasticsearch API REST**: Toda la información almacenada previamente en Elasticsearch puede ser accedida a través de su API REST por lo que será nuestro método de publicación de la información.
- **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 modulo 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.

3.3.4. Integración y Despliegue Continuo

Aunque la implementación de esta parte se escapa de los plazos del proyecto, las tecnologías que se usarán para llevarla a cabo serán.

- **BitBucket**: Será el repositorio usado para almacenar las nuevas versiones de nuestro *software* de manera que podamos tener un control de versiones.
- **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.

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 los datos con los que contamos y ver si se adaptan a nuestras necesidades. En este caso, al tratarse de un proyecto que se realiza dentro de una empresa, tenemos la posibilidad de acudir a las áreas dueñas del dato para solicitarle información adicional sobre el mismo.

En este apartado haremos un recorrido

4.1. Evolución del *dataset*

4.1.1. Las llamadas

Durante este apartado 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. Estos datos, como veremos de nuevo en el siguiente capítulo, nos han llevado a crear modelos poco eficientes que nos han situado otra vez en el punto de partida.

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 utilizar 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.

Este análisis fue realizado en *PySpark* y 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 unicamente es 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["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))

    raw_list_word.take(1)
```

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

[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])
    raw_no_stop.take(1)
```

Volvemos 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 4.1: Primeros datos: *wordcloud* inicial

Como podemos observar en la nube de palabras obtenida, el resultado no fue el esperado. Las llamadas que aparecían no eran significante 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 promer 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 hemos filtrado nos quedamos unicamente con la raíz de las palabras que sean verbos o nombres.

[7]:
```
        def get_stem_of_candidates(x):
            good = [u'VLinf', 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()
```

Con estos resultado volvemos a crear la nube de palabras.

[8]:
```
        list_stemmed_words = stemmed_candidates.reduce(lambda a,b:␣
↪list(set(a+b)))

        wordcloud = WordCloud().generate(" ".join(list_stemmed_words))

        # Display the generated image:
        plt.figure(figsize = (20,20))

        plt.imshow(wordcloud, interpolation='bilinear')
        plt.axis("off")
        plt.show()
```

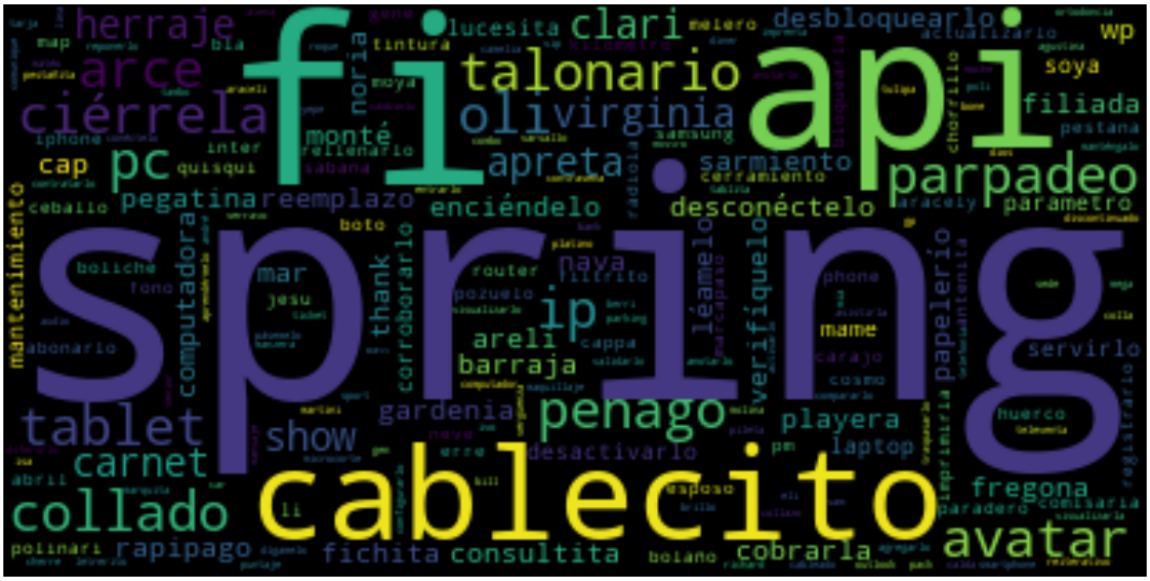

Figura 4.2: Primeros datos: *wordcloud* filtrado

Observando que los resultados no mejoran con esta propuesta, tampoco lo hicieron con la obtención de N-Gramas, y en el capítulo siguiente veremos que la creación de los modelos sobre este conjunto de datos nos llevó a volver a buscar y entender nuevos datos.

4.1.2. 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 e etiquetar llamadas manualmente, pero dado la cantidad de datos que podían ser necesarios para entrenar un modelo supervisado convertía esta posibilidad en inviable.

Capítulo 5

Modelos Minería de datos

A lo largo de este capítulo haremos un repaso por diferentes modelos, que representan diferentes métodos para conseguir un objetivo común, **clasificar las llamadas**. La mayoría de estos modelos intentan aplicar las técnicas vistas en el capítulo 2, dedicado al estado del arte, a los datos descritos en el capítulo 4.

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

5.1. Entorno de ejecución

5.1.1. Plataformas

5.1.2. Tecnologías

5.2. Preprocesado y representación de palabras

5.3. No supervisados

5.4. supervisados

Parte III

Explotación: procesamiento, visualización y alarmados

Capítulo 6

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 6.1 será necesaria para cumplir con otros requisitos propios de un sistema productivo.

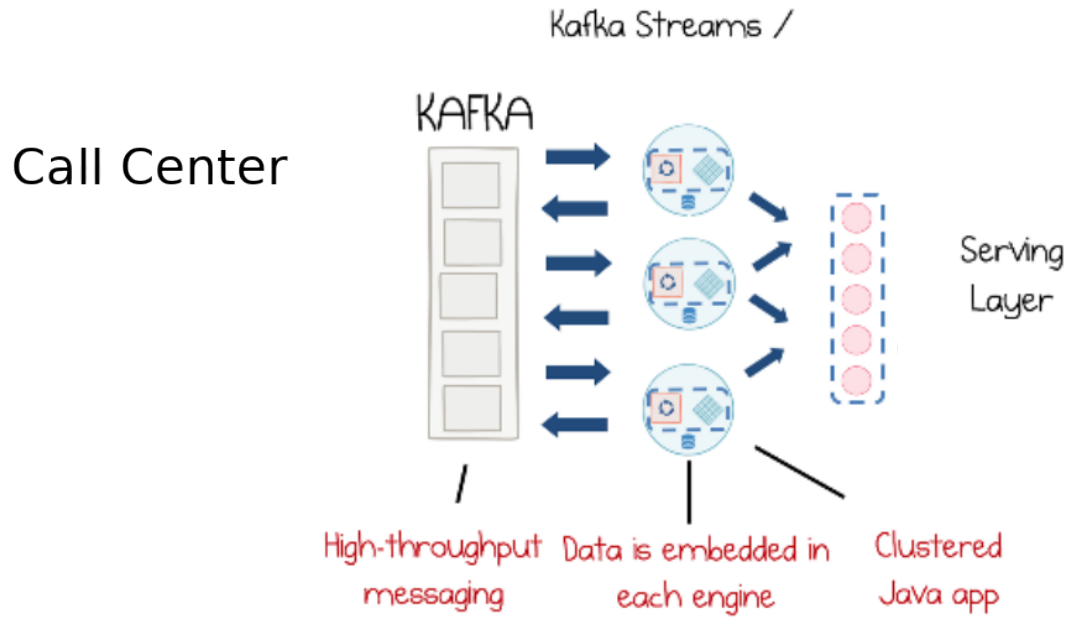

Figura 6.1: Streaming Layer

En la figura 6.1

6.1. 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 de 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).
- **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 debe 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 **??**.

6.2. 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 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 *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 *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.

6.3. 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.

6.3.1. Tecnología

En el apartado 3.3 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*.

6.3.1.1. *Kafka Streams*

El núcleo de nuestra capa de *Streaming* estará compuesto por microservicios desarrollados mediante *Kafka Streams*. *Kafka Streams* [25] 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 *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 esta 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 ***Streams DSL***.

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 *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.

6.3.1.2. *Tensorflow Serving*

Tensorflow Serving [24] 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 protocola, 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.

6.3.2. Microservicios

Una vez visto 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 obtenido 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.

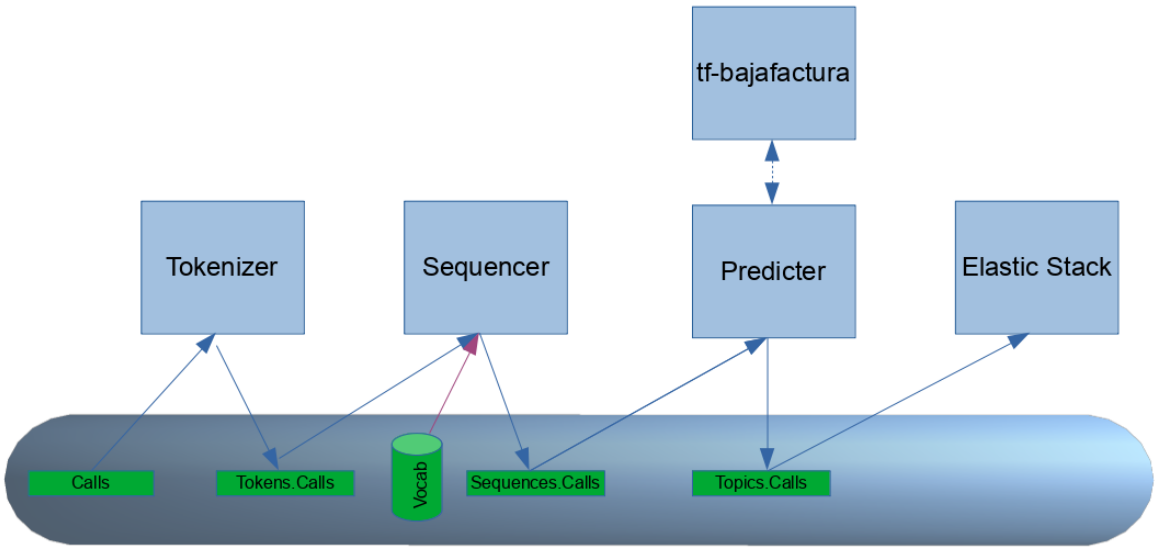

Figura 6.2: Arquitectura microservicios

En la figura 6.2 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 siguiente subapartados describiremos uno a uno los diferentes microservicios.

6.3.3. *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* eliminando *stopwords*, carácteres 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*.

6.3.4. *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 esta 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 en conjunto 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 nuvo *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.

un topic compacto sin límite de retención. 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 sin límite de retención, denominado *TBL.VOCABULARY.CALLS*, que tendrá como clave los *tokens* existentes en el vocabulario y, como valor, un identificador para cada uno de ellos.

6.3.5. *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 secuencia 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.

6.3.6. 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*[**?**] es siempre idéntica por lo que la llamada no varía aunque cambie el modelo y el número de clases y las etiquetas de las mismas son configurables en ***Predicter*** por lo que puede reutilizarse para cualquier servicio predictivo desplegado con *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 *Sequencer* 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 siembre 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.

6.4. 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 fallos 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.

6.4.1. 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 [**?**].

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.

6.4.2. 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 unicamente con:

- **Estado**: Que nos dará información sobre si el modelo está 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 en *Logstash*, seguirán el mismo flujo descrito en el apartado anterior que veremos en más detalle en el apartado 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*.

6.5. Inyector: simulador de tiempo real

Como ya hemos comentado en anteriores capítulos actualmente las transcripciones de las llamadas no se encuentran disponible 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 de 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 transcriba 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 7

Capa de servicio

7.1. Carga y modelo

7.2. Visualizaciones

7.2.1. Monitorización

7.2.2. Sistema

7.3. Alarmado

Capítulo 8

Despliegue en contenedores

Una vez desarrollados los microservicios y su monitorización, es necesario ponerlos en producción, y una de las decisiones importantes es donde 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 implementado 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.
- **Serivicios Autocontenidos**: Los contenedores pueden ejecutarse en multitud de plataforma lo que nos permite tener aplicaciones inmutables y autocontenidas 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 provocará la caída de un contenedor, este se levantaría automáticamente en otro nodo.

A la 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 correran en una única máquina y tendrán una única dirección IP.

- ***Despliegue***: Un despliegue (*Deployment*) pordemos 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.

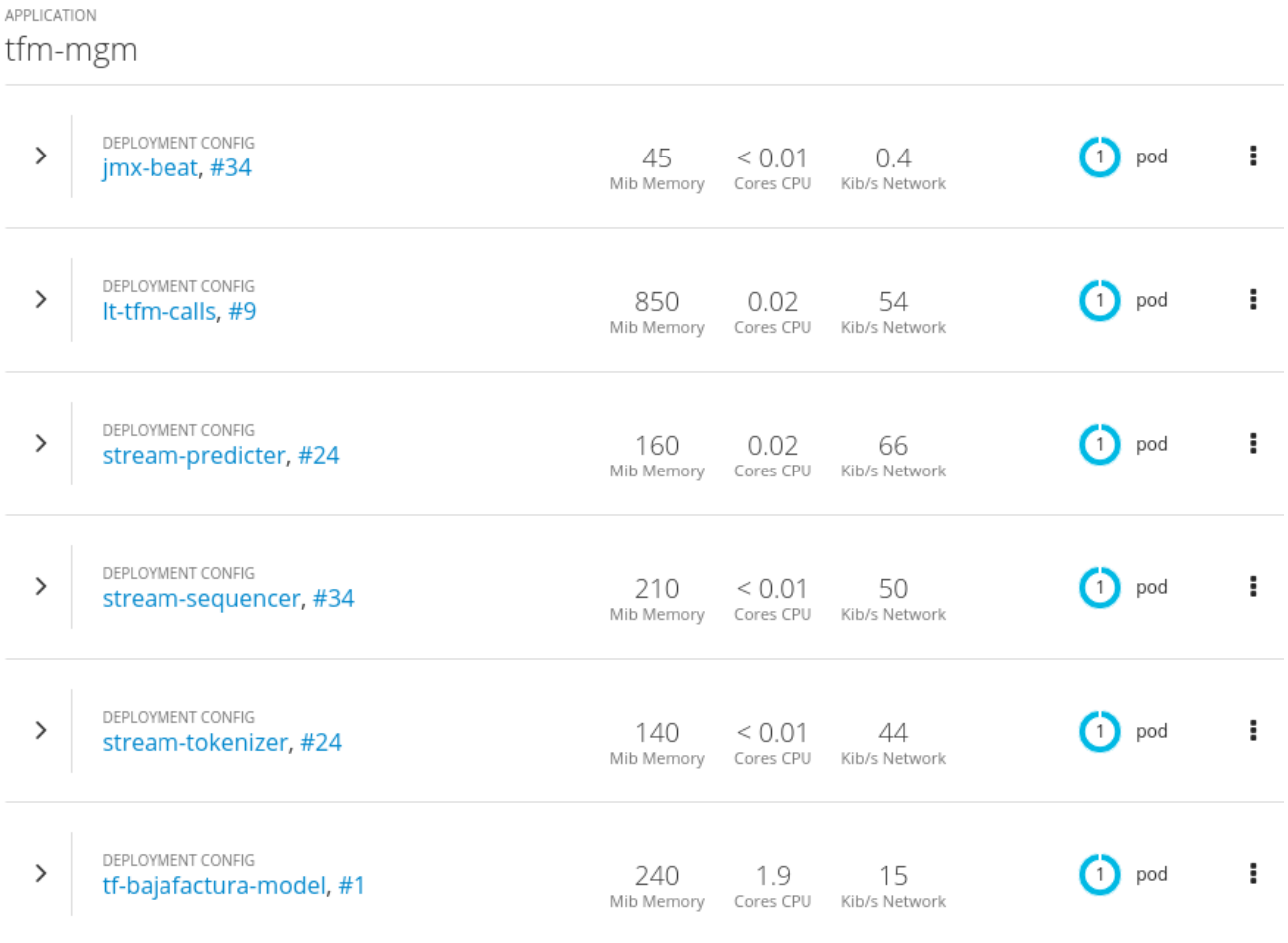

Figura 8.1: configuración de despliegue y recursos usados en OCP

En la figura 8.1 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.

8.1. Servicios *Kafka Streams*

Para desplegar los servicios de *Kafka Streams* en *OCP* es necesario desplegar un 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 el 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 [**?**].

El configuración de despliegue de *Predicter* es el siguiente:

```
apiVersion: apps.openshift.io/v1
kind: DeploymentConfig
metadata:
  labels:
    app: tfm-mgm
    project: topic-model
    service: stream-predicter
  name: stream-predicter
  namespace: nbia-prod
spec:
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    deploymentconfig: stream-predicter
  strategy:
    activeDeadlineSeconds: 21600
    recreateParams:
      timeoutSeconds: 600
    resources: {}
```

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

- **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 para 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 el momento del despliegue.

- **Container** (línea 30): Aquí encontramos toda la información del del contenedor que se desplegará. Destacamos:

  - **env** (línea 31): Variables globales del *deploynment*, 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 que 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 unicamente 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 unicamente contiene un *configmap*.

- **triggers** (línea 68): Define los disparadores que provocaran que se realice un nuevo *deployment*, en este caso la modificación de la imagen o la modificación del 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-predicter
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-predicter
16   sessionAffinity: None
17   type: ClusterIP
```

Aparte de las etiquetas que tienen las misma utilidad que el configuración de despliegue visto anteriormente, cabe destacar los siguiente parámetros:

- **name** (línea 6): El nombre del servicio que usaremos para llamarlo dentro de la plataforma.
- **name** (línea 6):
- **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-predicter*.
- **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*) actuan como un balanceador entre los *pods*.

8.2. Servicios *Tensorflow Serving*

En el caso del servicio de TensorflowServing mostraremos solo 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 el siguiente:

```
apiVersion: apps.openshift.io/v1
kind: DeploymentConfig
metadata:
  labels:
    app: tfm-mgm
    appName: tf-bajafactura-model
    appTypes: tensorflow-serving-s2i
    appid: tf-serving-tf-bajafactura-model
  name: tf-bajafactura-model
  namespace: nbia-prod
spec:
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    deploymentconfig: tf-bajafactura-model
  strategy:
    activeDeadlineSeconds: 21600
    rollingParams:
      intervalSeconds: 1
      maxSurge: 25%
      maxUnavailable: 25%
      timeoutSeconds: 600
      updatePeriodSeconds: 1
    type: Rolling
  template:
    metadata:
      labels:
        app: tfm-mgm
        appName: tf-bajafactura-model
        appTypes: tensorflow-serving-s2i
        appid: tf-serving-tf-bajafactura-model
        deploymentconfig: tf-bajafactura-model
    spec:
      containers:
        - env:
```

```
36            - name: PORT
37              value: '8501'
38            - name: MODEL_NAME
39              value: bajafactura
40            - 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 que 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 varian como las variables de entornos o los recursos definidos el funcionamiento es igual al configuración de despliegue definido anteriormente.

8.3. 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 6, para verificar el correcto funcionamiento del mismo.

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

- **Logstash**: Podemos identificarlo en la figura 8.1 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 8.1 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 despliegues.

8.3.1. Logstash

Al igual que en los casos anteriores, vemos unicamente 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.

8.3.2. 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:

```
apiVersion: apps.openshift.io/v1
kind: DeploymentConfig
metadata:
  labels:
    app: tfm-mgm
    deploymentconfig: jmx-beat
    service: jmx-beat
  name: jmx-beat
  namespace: nbia-prod
spec:
  replicas: 1
  selector:
    deploymentconfig: jmx-beat
  strategy:
    activeDeadlineSeconds: 21600
    resources: {}
    rollingParams:
      intervalSeconds: 1
      maxSurge: 25%
      maxUnavailable: 25%
      timeoutSeconds: 600
      updatePeriodSeconds: 1
    type: Rolling
  template:
    metadata:
      labels:
        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*.

8.4. 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* (*Sorce 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*.

8.4.1. *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*:

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  labels:
    app: tfm-mgm
  name: topic-model-streaming
  namespace: nbia-prod
spec:
  failedBuildsHistoryLimit: 5
  output:
    to:
      kind: ImageStreamTag
      name: 'topic-model-streaming:latest'
  runPolicy: Serial
  source:
    binary: {}
    type: Binary
  strategy:
    sourceStrategy:
      from:
        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.

8.4.2. *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* (practicamente idéntico al anterior) puede encontrarse en el anexo **??**.

Parte IV

Conclusiones: mantenimiento y futuros trabajos

Bibliografía

[1] Toni Lozano Bagén Anna Bosch Rué, Jordi Casas Roma. *Deep Learning Principios y Fundamentos*. 2018.

[2] 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.

[3] 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.

[4] 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.

[5] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, 2003.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Christine Fellbaum. *WordNet: an electronic lexical database*. MIT Press, 1998.

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[12] Yoav Goldberg. *Neural network methods in natural language processing*. Morgan & Claypool publishers, 2017.

[13] Tom Hope, Itay Lieder, and Yehezkel S. Resheff. *Learning TensorFlow: a guide to building deep learning systems*. OReilly Media, 2017.

[14] Jay Kreps. Questioning the lambda architecture. https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e, Julio 2014. Último acceso 2019-10-14.

[15] Douglas Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group, February 2001.

[16] A. S. M. Ashique Mahmood. *Literature Survey on Topic Modeling*. 2013.

[17] 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.

[18] Tomas Mikolov, Kai Chen, Greg S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[19] 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.

[20] 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.

[21] Christopher Olah. Understanding lstm networks – colah's blog. https://colah.github.io/posts/2015-08-Understanding-LSTMs/, Agosto 2015. Último acceso 2019-10-13.

[22] Stacey Ronaghan. Deep learning: Common architectures. https://medium.com/@srnghn/deep-learning-common-architectures-6071d47cb383, Agosto 2018. Último acceso 2019-10-13.

[23] 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.

[24] Tensorflow Serving. https://www.tensorflow.org/tfx/guide/serving. Último acceso 2019-12-10.

[25] Apache Kafka. Kafka Streams. https://kafka.apache.org/documentation/streams/. Último acceso 2019-12-10.

[26] A. M. Turing. *Computing machinery and intelligence*. Blackwell for the Mind Association, 1950.

[27] Kevin Warwick and Huma Shah. *Turings imitation game: conversations with the unknown*. Cambridge Univeristy Press, 2016.