



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



**TFG del Grado en Ingeniería
Informática**

JIZT

**Generación de resúmenes abstractivos en
la nube mediante Inteligencia Artificial**



Presentado por Diego Miguel Lozano
en la Universidad de Burgos — 9 de febrero de 2021
Tutores: Dr. Carlos López Nozal y
Dr. José Francisco Díez Pastor



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



D. Carlos López Nozal y D. José Francisco Díez Pastor, profesores del departamento de Ingeniería Informática, área de Lenguajes y Sistemas Informáticos.

Expone:

Que el alumno D. Diego Miguel Lozano, con DNI 71307413-F, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado “JIZT - Generación de resúmenes abstractivos en la nube mediante Inteligencia Artificial.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 9 de febrero de 2021

Vº. Bº. del Tutor:

D. Carlos López Nozal

Vº. Bº. del Tutor:

D. José Francisco Díez Pastor

Resumen

En los últimos cinco años, se han llevado a cabo grandes avances en el campo del Procesamiento de Lenguaje Natural (NLP). Sin embargo, el alcance de muchos de estos avances se ha visto limitado a ciertas áreas del NLP, como la traducción automática, los *bots* conversacionales, el filtrado de *spam*, etc., mientras que en el caso de otras tareas, como la generación de resúmenes, sigue existiendo a día de hoy escasez de alternativas.

JIZT es un servicio de generación de resúmenes en la nube que, gracias al desarrollo de una aplicación multiplataforma y de una API REST, ambos *open-source*, permite que cualquier persona, desde un usuario regular hasta un desarrollador con experiencia en el campo, pueda obtener resúmenes de sus textos de manera sencilla, eficiente y ajustada a sus necesidades.

La generación de estos resúmenes es, además, fruto de la utilización de los modelos de generación de lenguaje más avanzados a día de hoy. Como resultado, a diferencia de otros servicios, JIZT ofrece resúmenes abstractivos, esto es, resúmenes que contienen palabras y expresiones que no están presentes en el texto original.

Descriptores

NLP, generación de resúmenes abstractivos, *Cloud Native*, API REST, arquitectura de microservicios, aplicación multiplataforma.

Abstract

TODO (cuando me deis el visto bueno de la versión en español)

Keywords

TODO

Índice general

| | |
|---|------------|
| Índice general | III |
| Índice de figuras | v |
| Introducción | 1 |
| Objetivos del proyecto | 3 |
| 2.1. Objetivos generales | 3 |
| 2.2. Objetivos técnicos | 3 |
| Conceptos teóricos | 7 |
| 3.1. Pre-procesado del texto | 8 |
| 3.2. Codificación del texto | 10 |
| 3.3. Generación del resumen | 16 |
| 3.4. Post-procesado del texto | 25 |
| Técnicas y herramientas | 27 |
| 4.1. Modelo | 27 |
| 4.2. <i>Backend</i> | 28 |
| 4.3. <i>Frontend</i> | 36 |
| Aspectos relevantes del desarrollo del proyecto | 39 |
| 5.1. Metodología de desarrollo <i>software</i> : Kanban | 39 |
| 5.2. Motivación tras las arquitecturas desarrolladas | 42 |
| Trabajos relacionados | 51 |
| Conclusiones y Líneas de trabajo futuras | 55 |

| | |
|--|-----------|
| 7.1. Principales conclusiones | 55 |
| 7.2. Líneas futuras de trabajo | 56 |
| Bibliografía | 59 |

Índice de figuras

| | | |
|-------|---|----|
| 3.1. | Etapas en la generación de resúmenes. | 7 |
| 3.2. | Ejemplo de <i>tokenización</i> con el modelo T5. | 11 |
| 3.3. | Ejemplo de codificación del texto. | 12 |
| 3.4. | Ejemplo gráfico del algoritmo de balanceo. | 15 |
| 3.5. | Proceso de generación de resúmenes. | 16 |
| 3.6. | Ejemplo del modelo T5 de Google. | 18 |
| 3.7. | Ejemplo de generación con búsqueda voraz. | 20 |
| 3.8. | Ejemplo de generación con búsqueda <i>beam-search</i> | 21 |
| 3.9. | Distribución de probabilidades en la generación. | 22 |
| 3.10. | Ejemplo de muestreo. | 22 |
| 3.11. | Ejemplo de muestreo con temperatura. | 23 |
| 3.12. | Ejemplo de muestreo <i>top-k</i> | 23 |
| 3.13. | Ejemplo de muestreo <i>top-p</i> | 24 |
| 4.14. | Vista general de la arquitectura del <i>backend</i> | 28 |
| 4.15. | Componentes principales de Kubernetes. | 31 |
| 4.16. | Ejemplo de uso de Ingress con diferentes rutas. | 32 |
| 4.17. | Vista general de la arquitectura del <i>backend</i> | 33 |
| 4.18. | Diferentes enfoques en el despliegue de sistemas [49]. | 36 |
| 5.19. | Tablero Kanban utilizado. | 41 |
| 5.20. | Primer paso: realizar una petición POST. | 44 |
| 5.21. | Finalmente, obtenemos el resumen generado. | 45 |
| 5.22. | Arquitectura de la aplicación. | 46 |
| 5.23. | Ejemplo de jerarquía de <i>widgets</i> | 48 |

Introducción

El término Inteligencia Artificial (IA) fue acuñado por primera vez en la Conferencia de Dartmouth [1] hace ahora 65 años, esto es, en 1956. Sin embargo, ha sido en los últimos tiempos cuando su presencia e importancia en la sociedad han crecido de manera exponencial.

Uno de los campos históricos dentro de la AI, es el Procesamiento del Lenguaje Natural (NLP, por sus siglas en inglés), cuya significación se hizo patente con la aparición del célebre Test de Turing [2], en el cual un interrogador debe discernir entre un humano y una máquina conversando con ambos por escrito a través de una terminal.

Hasta los años 80, la mayor parte de los sistemas de NLP estaban basados en complejas reglas escritas a mano [3], las cuales conseguían generalmente modelos muy lentos, poco flexibles y con baja precisión. A partir de esta década, como fruto de los avances en Aprendizaje Automático (*Machine Learning*), fueron apareciendo modelos estadísticos, consiguiendo notables avances en campos como el de la traducción automática.

En la última década, el desarrollo ha sido aún mayor debido a factores como el aumento masivo de datos de entrenamiento (principalmente provenientes del contenido generado en la *web*), avances en la capacidad de computación (GPU, TPU, ASIC...) y el progreso dentro del área de la Algoritmia [4].

No obstante, ha sido desde la aparición del concepto de “atención” en 2015 [5, 6, 7] cuando el campo del NLP ha comenzado a lograr resultados cuanto menos sorprendentes [8, 9].

Con todo, la mayor parte de estos avances se han visto limitados al ámbito académico y corporativo. Los modelos cuyo código ha sido publicado, o bien no están entrenados, o bien requieren para ser usados conocimientos

avanzados de matemáticas o programación, o simplemente son demasiado grandes para ser ejecutados en ordenadores convencionales.

Con esta idea en mente, el objetivo de JIZT se centra en acercar los modelos NLP del estado del arte tanto a usuarios expertos, como no expertos.

Para ello, JIZT proporciona:

- Una API REST destinada a los usuarios con conocimientos técnicos, a través de la cual se pueden llevar a cabo tareas de NLP.
- Una aplicación multiplataforma que consume dicha API, y que proporciona una interfaz gráfica sencilla e intuitiva. Esta aplicación puede ser utilizada por el público general, aunque no deja de ofrecer opciones avanzadas para aquellos usuarios con mayores conocimientos en la materia.

En un principio, dado el alcance de un Trabajo de Final de Grado, la única tarea de NLP implementada ha sido la de generación de resúmenes. La motivación para esta decisión se ha fundamentado principalmente en la relativa menor popularidad de esta área frente a otras como la traducción automática, el análisis de sentimientos, o los modelos conversacionales. Para estas últimas tareas existe actualmente una amplia oferta de grandes compañías como Google [10], IBM [11], Amazon [12], o Microsoft [13], entre muchas otras. Nuestra mayor limitación reside en que los modelos pre-entrenados que utilizaremos para la generación de los resúmenes funcionan únicamente en inglés. Esperamos que en un futuro próximo aparezcan modelos que admitan otros idiomas.

En un mundo en el que en cinco años se producirán globalmente 463 exabytes de información al día [14], siendo mucha de esa información textual, la generación de resúmenes aliviará en cierto modo el tratamiento de esos datos.

Sin embargo, gran parte del esfuerzo de desarrollo de JIZT se ha centrado en el diseño de su arquitectura, la cual se describirá con detalle en el capítulo de **Conceptos Teóricos**. Por ahora adelantaremos que ha sido concebida con el objetivo de ofrecer la mayor escalabilidad y flexibilidad posible, manteniendo además la capacidad de poder añadir otras tareas de NLP diferentes de la generación de resúmenes en un futuro cercano.

Por todo ello, el presente TFG conforma el punto de partida de un proyecto ambicioso, desafiante, pero con la certeza de que, independientemente de su recorrido, habremos aprendido, disfrutado, y ojalá ayudado a alguien por el camino.

Objetivos del proyecto

2.1. Objetivos generales

- Ofrecer la capacidad de llevar a cabo tareas de NLP tanto al público general, como al especializado. Como se ha mencionado con anterioridad, la única tarea NLP que implementará el presente TFG será la de generación de resúmenes.
- Emplear modelos pre-entrenados del estado del arte para la generación de resúmenes abstractivos. Los resúmenes abstractivos se diferencian de los extractivos en que el resumen generado contiene palabras o expresiones que no aparecen en el texto original [15]. Dicho de forma más técnica, existe cierto nivel de paráfrasis.
- Diseñar una arquitectura con aspectos como la flexibilidad, la escalabilidad y la alta disponibilidad como principios fundamentales.
- Poner en práctica lo aprendido a lo largo de la carrera en áreas como Ingeniería del Software, Sistemas Distribuidos, Programación, Minería de Datos, Algoritmia y Bases de Datos.
- Ofrecer la totalidad del proyecto bajo licencias de *Software Libre*.

2.2. Objetivos técnicos

- Los modelos pre-entrenados de generación de texto admiten parámetros específicos para configurar dicha generación, por lo que se deberá

implementar una interfaz que permita a los usuarios establecer dichos parámetros de manera opcional. Por defecto, se proporcionarán los valores que mejores resultados ofrecen, extraídos mayoritariamente de manera experimental.

- Los modelos pre-entrenados de generación estado del arte presentan frecuentemente limitación en la longitud de los textos de entrada que reciben, derivada de la longitud de las secuencias de entrada con las que han sido entrenados. Esta longitud llega a ser tan baja como 512 *tókenes*¹ [16]. Por tanto, se deberá establecer algún mecanismo que permita sortear esta limitación para poder generar resúmenes de textos arbitrariamente largos.
- Gestionar el pre-procesado de los textos a resumir para ajustarlos a la entrada que los modelos pre-entrenados esperan.
- Algunos modelos pre-entrenados generan textos enteramente en minúsculas. Se deberá, por tanto, incluir mecanismos en la etapa de post-procesado que permitan recomponer el correcto uso de las mayúsculas en los resúmenes generados.
- Con el fin de cumplir con el objetivo general referente a la arquitectura, desarrollar una arquitectura de microservicios, basada en la filosofía *Cloud Native* [17, 18]. Este objetivo se divide a su vez en dos puntos:
 - Encapsular cada microservicio en un contenedor Docker.
 - Implementar la orquestación y balanceo de los microservicios a través de Kubernetes.
- Complementariamente al punto anterior, implementar una arquitectura dirigida por eventos [19]. La motivación detrás de la utilización de este patrón arquitectónico se justifica en el capítulo de [Conceptos Teóricos](#).
- Implementar una API REST escrita en Python empleando el *framework web* Flask. Dicha API será el punto de conexión con el servicio de generación de resúmenes en la nube.
- Desplegar PostgreSQL como servicio en Kubernetes mediante el Operador PostgreSQL de Crunchy [20]. Esta base de datos cumplirá la doble función de (a) servir como caché para no volver a producir resúmenes ya generados con anterioridad, incrementando la velocidad

¹ Este término se definirá posteriormente. Por ahora, el lector puede considerar que un *tóken* es equivalente a una palabra.

de respuesta, y (b) almacenar los resúmenes generados con fines de evaluación de la calidad de los mismos y extracción de métricas.

- Desarrollar, con ayuda de Flutter, una aplicación multiplataforma con soporte nativo para Android, iOS, y *web*. Esta aplicación consumirá la API y proporcionará una interfaz gráfica sencilla e intuitiva para que usuarios regulares puedan hacer uso del servicio de generación de resúmenes.
- Seguir el patrón de diseño Clean Architecture [21] y de *offline-first* [22] para la implementación de la aplicación.

Conceptos teóricos

En este capítulo, detallaremos de forma teórica el proceso de generación de resúmenes, desde el momento que recibimos el texto a resumir, hasta que se le entrega al usuario el resumen generado. En el [siguiente capítulo](#), explicaremos las herramientas que hacen posible que todo este proceso se pueda llevar a cabo de forma distribuida «en la nube».

La generación de resúmenes se divide en cuatro etapas fundamentales:

1. Pre-procesado.
2. Codificación.
3. Generación del resumen.
4. Post-procesado.

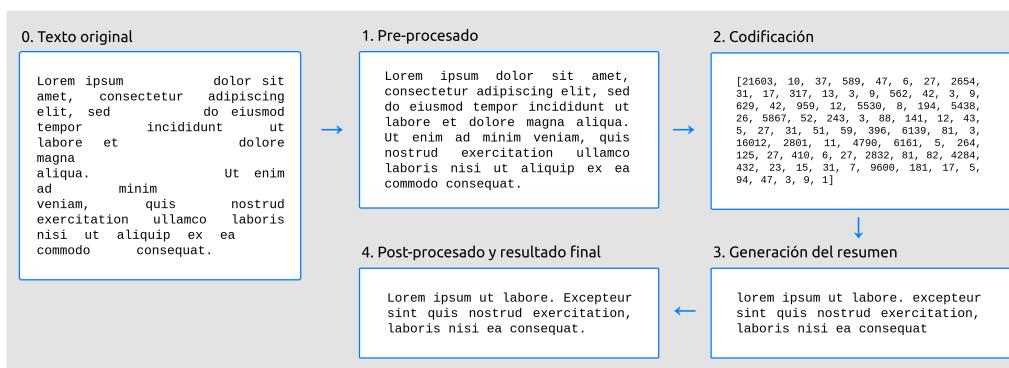


Figura 3.1: Etapas en la generación de resúmenes.

Veamos en detalle en qué consiste cada una de ellas.

3.1. Pre-procesado del texto

El principal objetivo de esta etapa es adecuar el texto de entrada para que se aproxime lo máximo posible a lo que el modelo espera. Adicionalmente, se separa en texto de entrada en frases. Esta separación puede parecer *a priori* una tarea trivial, pero involucra una serie de dificultades que se detallarán a continuación.

Cabe destacar que, como mencionábamos en la [Introducción](#), los modelos pre-entrenados de los que hacemos uso solo admiten textos en inglés, por lo que algunas de las consideraciones que tomamos en el pre-procesado del texto solo son aplicables a este idioma.

A grandes rasgos, en la etapa de pre-procesado se divide a su vez en los siguientes pasos:

- Eliminar retornos de carro, tabuladores (`\n`, `\t`) y espacios sobrantes entre palabras (p. ej. "`I am`" → "`I am`").
- Añadir un espacio al inicio de las frases intermedias (p. ej.: "`How's it going? Great!`" → "`How's it going? Great!`". Esto es especialmente relevante en el caso de algunos modelos, como por ejemplo BART [23], los cuales tienen en cuenta ese espacio inicial para distinguir entre frases iniciales y frases intermedias en la generación de resúmenes².
- Establecer un mecanismo que permita llevar a cabo la ya mencionada separación del texto en frases. Esto es importante dado que los modelos tienen un tamaño de entrada máximo. Dos estrategias comunes para eludir esta limitación consisten en (a) truncar el texto de entrada, lo cual puede llevar asociado pérdidas notables de información, o (b) dividir el texto en fragmentos de menor tamaño. En nuestro caso, la primera opción quedó rápidamente descartada ya que los textos que vamos a recibir, por lo general, superarán el tamaño máximo (en caso contrario tendría poco sentido querer generar un resumen). Refiriéndonos, por tanto, a la segunda opción, es frecuente llevar a cabo dicha separación de manera ingenua, únicamente atendiendo al tamaño de entrada máximo. Sin embargo, en nuestro caso decidimos refinar este proceso e implementamos un algoritmo original³ en el

² Por el momento, no hacemos uso de este modelo, aunque podría incluirse en el futuro.

³ Utilizamos el término «original» porque no encontramos ningún recurso en el que se tratara este problema, por lo que tuvimos que resolverlo sin apoyos bibliográficos. Esto no quiere decir, sin embargo, que no se hayan implementado estrategias similares en otros problemas diferentes al aquí expuesto.

que dicha separación se realiza de tal modo que ninguna frase queda dividida. Para garantizar el éxito de este algoritmo, es fundamental que las frases estén correctamente divididas; el porqué se clarificará en la [siguiente sección](#), referente a la codificación del texto, en la que también se incluye la implementación concreta del algoritmo.

A continuación, nos centraremos en el proceso de división del texto en frases. A la hora de llevar a cabo este proceso, debemos tener en cuenta que el texto de entrada podría contener errores ortográficos o gramaticales, por lo que debemos tratar de realizar el mínimo número de suposiciones posibles.

No obstante, la siguiente consideración se nos hace necesaria: el punto (.) indica el final de una frase solo si la siguiente palabra empieza con una letra *y* además mayúscula.

Por ejemplo, en el caso de: "Your idea is interesting. However, I would [...]." se separaría en dos frases, dado que la palabra posterior al punto empieza con una letra mayúscula. Sin embargo: "We already mentioned in Section 1.1 that this example shows [...]." conformaría una única frase, ya que tras el punto no aparece una letra. Procedemos de igual modo en el caso de los signos de interrogación (?) y de exclamación (!). Por ejemplo: "She asked 'How's it going?', and I said 'Great!'." se tomará correctamente como una sola frase; tras la interrogación, la siguiente palabra comienza con una letra *minúscula*.

Con la suposición anterior, también se agruparían correctamente los puntos suspensivos.

Sin embargo, fallaría en situaciones como: "NLP (i.e. Natural Language Processing) is a subfield of Linguistics, Computer Science, and Artificial Intelligence.", en la que la división sería: "NLP (i.e. " por un lado, y "Natural Language Processing) is a subfield [...].", por otro, ya que "Natural" empieza con mayúscula y aparece tras un punto.

Asimismo, la razón principal por la que no podemos apoyarnos únicamente en reglas predefinidas, reside en las llamadas Entidades Nombradas (*Named Entities*, en inglés), esto es, palabras que hacen referencia a personas, lugares, instituciones, empresas, etc. Si empleáramos reglas predefinidas, podríamos incluir en un diccionario todas las entidades nombradas existentes conocidas que contengan puntos, de forma que si aparecieran en el texto las palabras "U.K." o "A.W.O.L." las agruparíamos como tal, sin partir la frase. Sin embargo, existen potencialmente miles (o incluso decenas de miles)

de entidades nombradas que contienen puntos, por lo que crear y mantener manualmente dicho diccionario sería muy costoso.

Actualmente, para resolver este tipo de problemas, se emplean modelos estadísticos o de *deep learning*. Esta disciplina se conoce como Reconocimiento de Entidades Nombradas (NER, por sus siglas en inglés), y pese a los buenos resultados conseguidos por algunos de los modelos propuestos, se considera un problema lejos de estar resuelto [24].

En nuestro caso emplearemos un modelo pre-entrenado para solucionar, al menos en parte, el problema de las Entidades Nombradas. Este modelo también solventa situaciones como la descrita anteriormente, en las que las reglas escritas a mano se quedan cortas. En el capítulo de [Técnicas y Herramientas](#), hablaremos de dicho modelo y de la implementación concreta en código de los procedimientos expuestos anteriormente.

3.2. Codificación del texto

En esta etapa, se lleva a cabo lo que se conoce en inglés como *word embedding*⁴. Los modelos de IA trabajan, por lo general, con representaciones numéricas. Por ello, las técnicas de *word embedding* se centran en vincular texto (bien sea palabras, frases, etc.), con vectores de números reales [25]. Esto hace posible aplicar a la generación de texto arquitecturas comunes dentro de la IA (y especialmente, del *Deep Learning*), como por ejemplo las Redes Neuronales Convolucionales (CNN) [26].

Esta idea, conceptualmente sencilla, encierra una gran complejidad, dado que los vectores generados deben retener la máxima información posible del texto original, incluyendo aspectos semánticos y gramaticales. Por poner un ejemplo, los vectores correspondientes a las palabras «profesor» y «alumno», deben preservar cierta relación entre ambos, y a su vez con la palabra «educación» o «escuela». Además, su vínculo con las palabras «enseñar» o «aprender» será ligeramente distinto, dado que en este caso se trata de una categoría gramatical diferente (verbos, en vez de sustantivos). A través de este ejemplo, podemos comprender que se trata de un proceso complejo.

Dado que los modelos pre-entrenados se encargan de realizar esta codificación por nosotros, no entraremos en más detalle en los algoritmos

⁴ En el presente documento, hemos traducido este término como «codificación del texto».

concretos empleados, dado que consideramos que queda fuera del alcance de este trabajo⁵.

Lo que sí hemos tenido que implementar en esta etapa, ha sido la división del texto en fragmentos a fin de no superar el tamaño máximo de entrada del modelo.

De este modo, podremos realizar resúmenes de textos arbitrariamente largos, a través de los siguientes pasos:

1. Dividimos el texto en fragmentos.
2. Generamos un resumen de cada fragmento.
3. Concatenamos los resúmenes generados.

Anteriormente, habíamos mencionado el término *token*. Este concepto se puede traducir al español como «símbolo». En nuestro caso concreto, un *token* se corresponde con el vector numérico asociado a una palabra al realizar la codificación. Más concretamente, en modelos más actuales, como el modelo T5 [16], los *tókenes* pueden referirse a palabras completas o a *fragmentos* de las mismas.

Por lo general, las palabras que aparecen en el vocabulario con el que ha sido entrenado el modelo van a generar un único *token*. Sin embargo, las palabras desconocidas, se descompondrán en varios *tókenes*. Lo mismo sucede con palabras compuestas o formadas a partir de prefijación o sufijación. En la [siguiente figura](#), podemos ver un ejemplo de ello:

| | | | | |
|------------------------|------------------------|---|--------------------------------|-----------------------|
| Palabra simple: | <code>brutal</code> | → | <code>[14506]</code> | Un <i>token</i> |
| Palabra con sufijo: | <code>brutality</code> | → | <code>[14506, 485]</code> | Varios <i>tókenes</i> |
| Palabra compuesta: | <code>backbone</code> | → | <code>[223, 12269]</code> | Varios <i>tókenes</i> |
| Palabra no reconocida: | <code>JIZT</code> | → | <code>[446, 20091, 382]</code> | Varios <i>tókenes</i> |

Figura 3.2: Ejemplo de *tokenización* con el modelo T5.

En el anterior ejemplo, si decodificamos los *tókenes* correspondientes a la palabra "brutality", esto es, `[14506, 485]`, obtenemos los fragmentos

⁵ En cualquier caso, el lector curioso puede explorar los algoritmos más populares de codificación, los cuales, ordenados cronológicamente, son: word2vec [27, 28], GloVe [29], y más recientemente, ELMo [30] y BERT [31].

"brutal" e "ity", respectivamente. Análogamente, la palabra "backbone", se descompone en "back" y "bone".

La idea detrás de esta fragmentación se basa en la composición, uno de los mecanismos morfológicos de formación de palabras más frecuentes [32] en muchos idiomas, como el inglés, español o alemán. Por tanto, presupone que dividiendo las palabras desconocidas en fragmentos menores, podemos facilitar la comprensión de las mismas. Naturalmente, habrá casos en los que esta idea falle; por ejemplo, en la figura anterior, la palabra "JIZT" se descompone en "J", "IZ", "T", lo cual no parece hacerla mucho más comprensible.

Una vez explicado el concepto de *token*, volvamos al problema ya mencionado con anterioridad: algunos modelos de generación de texto (entre ellos, el T5) admiten un tamaño de entrada máximo, determinado en función del número de *tókens*. Debido a que la unidad de medida es el número de *tókenes*, y no el número de palabras, o de caracteres, debemos tener en cuenta algunos detalles, entre ellos el hecho de que los modelos generan *tókenes* especiales para marcar el inicio y/o el final de la secuencia de entrada.

El modelo T5 (el cual como mencionábamos anteriormente, es el único modelo que utilizamos por ahora), genera un único *token* de finalización de secuencia (EOS, *end-of-sequence*), que se coloca siempre al final del texto de entrada, una vez codificado, y en el caso de este modelo siempre tiene el *id* 1. En la [siguiente figura](#) podemos ver un ejemplo con un texto de entrada:

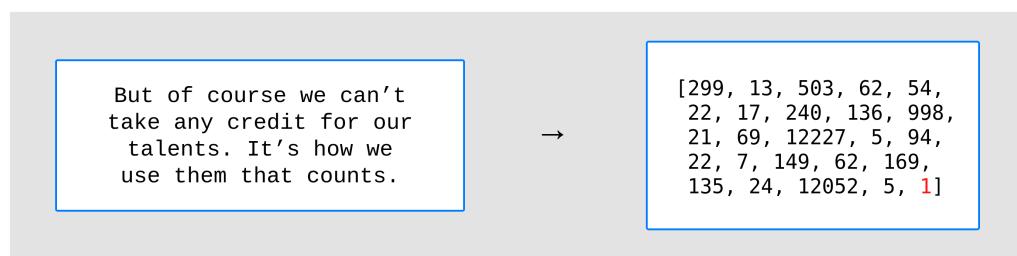


Figura 3.3: Pasaje del libro *A Wrinkle in Time*. El *tóken* EOS se ha marcado en rojo.

Como podemos ver, el *tóken* EOS aparece una única vez por cada texto de entrada, y es independiente de las palabras o frases que este contiene.

Otro aspecto a tener en cuenta, reside en que este modelo no solo es capaz de generar resúmenes, si no que puede ser empleado para otras tareas como la traducción, respuesta de preguntas [16], etc. Para indicarle

cuál de estas es la tarea que queremos que desempeñe, curiosamente se lo tenemos que indicar tal y cómo lo haríamos en la vida real; en nuestro caso, simplemente precedemos el texto a resumir con la orden «resume» («*summarize*»). Por poner otro ejemplo, si quisieramos traducir del alemán al español, le señalaríamos: «traduce de alemán a español» («*translate German to Spanish*») seguido de nuestro texto.

Por consiguiente, este prefijo deberá aparecer al principio de cada una de las subdivisiones generadas y, del mismo modo, deberemos tenerlo en cuenta a la hora de calcular el número de *tókenes* de las mismas.

Con las anteriores consideraciones en mente, el objetivo principal será llevar a cabo la división del texto de entrada de forma que el número de *tókenes* varíe lo mínimo posible entre las diferentes subdivisiones, y todo ello sin partir ninguna frase.

Esta es una tarea más compleja de lo que puede parecer. En nuestro caso, hemos propuesto un **algoritmo** que emplea una estrategia voraz para llevar a cabo una primera división del texto; posteriormente procede al *balanceo* de las subdivisiones generadas en el paso anterior, de forma que el número de *tókenes* en cada subdivisión sea lo más parecido posible. Y esto, evidentemente, sin superar el máximo tamaño de entrada del modelo en ninguna de las subdivisiones.

Algoritmo 1 División y codificación del texto.

```

1: procedure CODIFICACIÓNCONDIVISIÓN(texto, prefijo)
2:   frases  $\leftarrow$  dividirEnFrases(texto)
3:   frasesCodif  $\leftarrow$  []
4:   prefijoCodif  $\leftarrow$  codifica(prefijo)                                 $\triangleright$  Frases codificadas
5:   EOSCodif  $\leftarrow$  codifica(EOS)                                      $\triangleright$  Token EOS codificado
6:   subdivsCodif  $\leftarrow$  []                                          $\triangleright$  Subdivisiones codificadas
7:   for fr in frases do
8:     frasesCodif  $\leftarrow$  codifica(fr)[: -1]                                $\triangleright$  Excluir EOS
9:   end for
10:  ptosCorte  $\leftarrow$  divideVoraz(frasesCodif, prefijoCodif)
11:  ptosCorte  $\leftarrow$  balanceaSubdivs(ptosCorte)
12:  for i  $\leftarrow$  0, len(ptosCorte) - 1 do
13:    frasesSubvid  $\leftarrow$  frasesCodif[ptosCorte[i] : ptosCorte[i + 1]]   $\triangleright$  Frases en subdiv.
14:    subdivsCodif[i]  $\leftarrow$  concatena(prefijoCodif, frasesSubdiv, EOSCodif)
15:  end for
16:  return subdivsCodif
17: end procedure

```

Este algoritmo devuelve las subdivisiones en las que se ha separado el texto, ya codificadas. Por tanto, *subdivsCodif* tendrá la siguiente forma:

$[[23, 34, 543, 45, \dots, 1], [23, 32, 401, 11, \dots, 1], [23, 74, 25, 204, \dots, 1], \dots]$

Es decir, cada una de las listas contenidas en *subdivsCodif* contiene los *tókenes* correspondientes a dicha subdivisión, con el prefijo (23) y el *token* EOS (1) añadidos.

La lógica detrás de la función *divideVoraz* es la siguiente:

Algoritmo 2 División voraz del texto.

```

1: procedure DIVIDEVORAZ(frasesCodif, prefijoCodif)
2:   ptosCorte  $\leftarrow [0]$ 
3:   lenSubdiv = len(prefijoCodif) + len(frasesCodif[0]) + 1       $\triangleright$  Contar prefijo y EOS
4:   for i  $\leftarrow 0$ , len(frasesCodif) - 1 do
5:     lenSubdiv = lenSubdiv + len(frasesCodif[i])
6:     if lenSubdiv > model.maxLength then
7:       ptosCorte.añadir(i)
8:       lenSubdiv = len(prefijoCodif) + len(frasesCodif[i]) + 1
9:     end if
10:   end for
11:   ptosCorte.añadir(len(frasesCodif))
12:   return ptosCorte
13: end procedure

```

Es decir, *ptosCorte* será una lista que indique los índices que delimitan cada subdivisión, por ejemplo:

$$[0, 45, 91, 130, 179, 190]$$

En este caso, la primera subdivisión iría desde la frase 0 hasta la 45, la segunda subdivisión de la 46 a la 91, la tercera de la 92 a la 130, y así sucesivamente.

Como podemos ver en el ejemplo, el número de *tókenes* por subdivisión está en torno a los 45, menos en la última subdivisión que solo contiene 10 *tókenes* (190 – 180). Debido a la propia naturaleza del algoritmo voraz, será siempre la última subdivisión la que pueda contener un número de *tókenes* muy por debajo de la media, lo que puede causar que el resumen de esta última subdivisión sea demasiado corto (o incluso sea la cadena vacía). Para evitar esto, balanceamos las subdivisiones, de forma que el número de *tókenes* en cada una de ellas esté equilibrado.

En esencia, lo que este último algoritmo hace es comparar la diferencia en número de *tókenes* entre subdivisiones consecutivas, empezando por el final, de forma que primero se compara la penúltima con la última subdivisión, después la antepenúltima con la penúltima, y así sucesivamente. Si es necesario, va moviendo frases completas desde una subdivisión a la siguiente, por ejemplo, desde la penúltima a la última subdivisión. Este algoritmo tiene una complejidad en el peor de los casos de $O(n^3)$, siendo n el número de subdivisiones.

Algoritmo 3 Balanceo de las subdivisiones.

```

1: procedure BALANCEASUBDIVS(ptosCorte)
2:   ptosCorteBalan  $\leftarrow$  ptosCorte                                 $\triangleright$  Puntos de corte balanceados
3:   do
4:     ptosCorteBalanOld  $\leftarrow$  ptosCorteBalan
5:     for i  $\leftarrow$  len(ptosCorteBalan) - 1, 1, step : -1 do       $\triangleright$  Empezar por última subdiv.
6:       diffLen  $\leftarrow$  lenSubdiv(i - 1) - lenSubdiv(i)           $\triangleright$  Diferencia en n. de tókenes
7:       while diffLen > 0 do
8:         últimaFrase  $\leftarrow$  getÚltimaFrase(getSubdiv(i-1))
9:         if getSubdiv(i) + len(últimaFrase) <= model.maxLength  $\wedge$ 
           len(últimaFrase) <= diffLen then
10:            mueveÚltimaFrase(getSubdiv(i-1), getSubdiv(i))
11:            ptosCorteBalan[i - 1]  $\leftarrow$  ptosCorteBalan[i - 1] - 1
12:         else
13:           break
14:         end if
15:       end while
16:     end for
17:   while ptosCorteBalan  $\neq$  ptosCorteBalanOld
18:     return ptosCorte
19: end procedure

```

Podemos visualizarlo gráficamente con un ejemplo muy simple:



Figura 3.4: Ejemplo gráfico del algoritmo de balanceo. En este caso, la longitud máxima de cada subdivisión es de 100 *tókenes*. Las desviación estándar del número de *tókenes* de cada frase en t_1 es $\sigma_1 = 39,63$ y en t_5 , acaba siendo $\sigma_5 = 1,53$.

3.3. Generación del resumen

Una vez codificado y dividido el texto apropiadamente, generamos los resúmenes parciales para posteriormente unirlos, dando lugar a un único resumen del texto completo.

En la [Figura 3.5](#), podemos ver los pasos llevados a cabo tanto en la anterior etapa, la codificación y división del texto, como en esta, la generación del resumen.

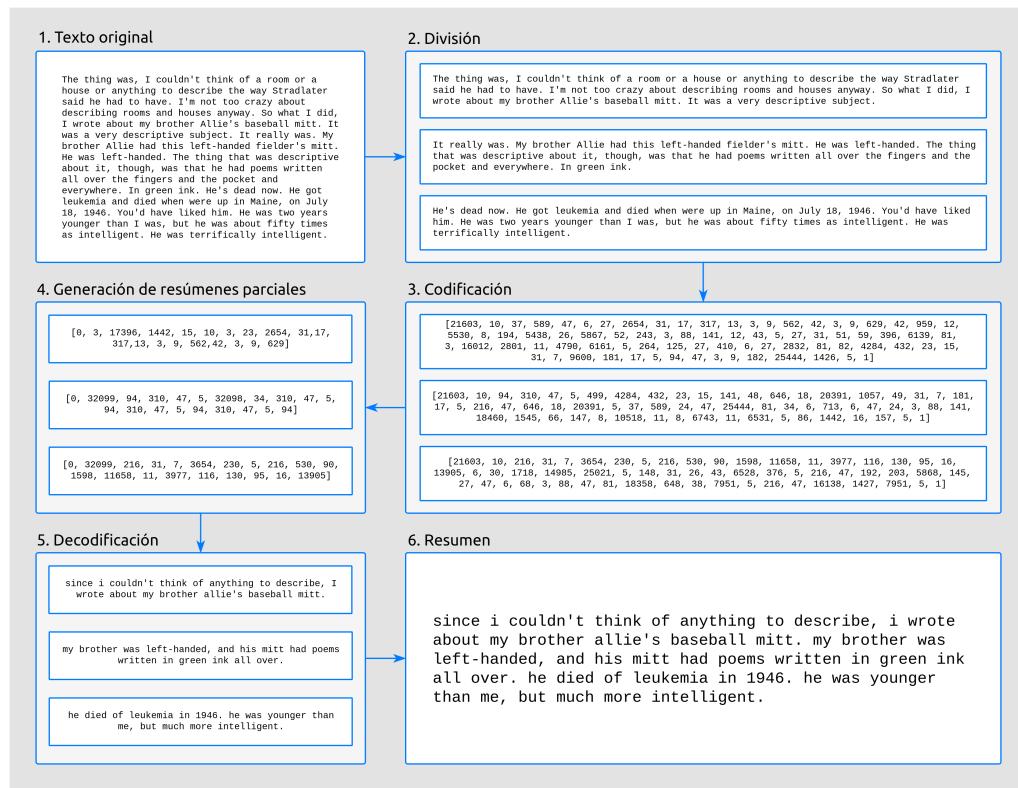


Figura 3.5: Proceso de generación de resúmenes, ilustrado con un fragmento del libro *The Catcher in the Rye*.

Como podemos apreciar en la anterior figura, el modelo generador de resúmenes toma el texto codificado, y devuelve una versión reducida del mismo, también codificado. Por ello, antes de poder unir y devolver el resumen generado, debemos realizar un paso de *decodificación*, que realiza el proceso contrario a la *codificación*, como veímos en la [anterior sección](#). Algo con lo que tendremos que lidiar en la siguiente etapa, el post-procesado,

será corregir el resumen generado para que se ajuste a las reglas ortográficas vigentes, en especial en lo relativo al uso de mayúsculas.

La ventaja de utilizar modelos pre-entrenados es clara: estos modelos son para nosotros cajas negras, a las que solo tenemos que encargarnos de proporcionarles la entrada en el formato concreto que esperan.

Cabe destacar que, el hecho de realizar la división del texto de esta manera, sin atender a aspectos semánticos, podría resultar en que en frases estrechamente relacionadas acabaran en distintas subdivisiones. Por ejemplo, en la [Figura 3.5](#), la frase final de uno de las subdivisiones es: «*It was a very descriptive subject*» («Era un tema muy descriptivo»), a la cual le sigue, ya en la siguiente subdivisión: «*It really was*» («De veras que lo era»), aludiendo a la anterior frase.

Estos casos son difíciles de resolver. Una posible idea sería tratar de determinar si una frase está relacionada con la anterior, quizás mediante el uso de otro modelo, y de ser así, tratar de mantenerlas en una misma subdivisión, a fin de que el resumen final mantenga la máxima cohesión y coherencia posibles. Esto incrementaría, no obstante, los tiempos de generación de resúmenes. Por ahora, creemos que los resultados obtenidos son lo suficientemente buenos.

Modelo empleado para la generación de resúmenes: T5

Come hemos mencionado previamente, JIZT hace uso del modelo T5 [16] de Google. Este modelo fue introducido en el artículo *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*, presentado en 2019. En él, Colin Raffel *et al.* estudian las ventajas de la técnica del aprendizaje por transferencia (*transfer learning*) al campo del Procesamiento del Lenguaje Natural (NLP).

Tradicionalmente, cada nuevo modelo se entrenaba desde cero. Esto ha cambiado con la inclusión del aprendizaje por transferencia; actualmente, la tendencia es emplear modelos pre-entrenados como punto de partida para la construcción de nuevos modelos.

Las tres principales ventajas del empleo del aprendizaje por transferencia son [33]:

- Mejora del rendimiento de partida. El hecho de comenzar con un modelo pre-entrenado en vez de un modelo ignorante (*ignorant learner*), proporciona un rendimiento base desde el primer momento.

- Disminución del tiempo de desarrollo del modelo, consecuencia del punto anterior.
- Mejora del rendimiento final. Esta mejora ha sido estudiada tanto en el caso del NLP [34], como de otros ámbitos, como la visión artificial [35], o el campo de la medicina [36].

La principal novedad de este artículo se encuentra en su propuesta de tratar todos los problemas de procesamiento de texto como problemas texto a texto (*text-to-text*), es decir, tomar un texto como entrada, y producir un nuevo texto como salida. Esto permite crear un modelo general, al que han bautizado como T5, capaz de llevar a cabo diversas tareas de NLP, como muestra el siguiente diagrama:

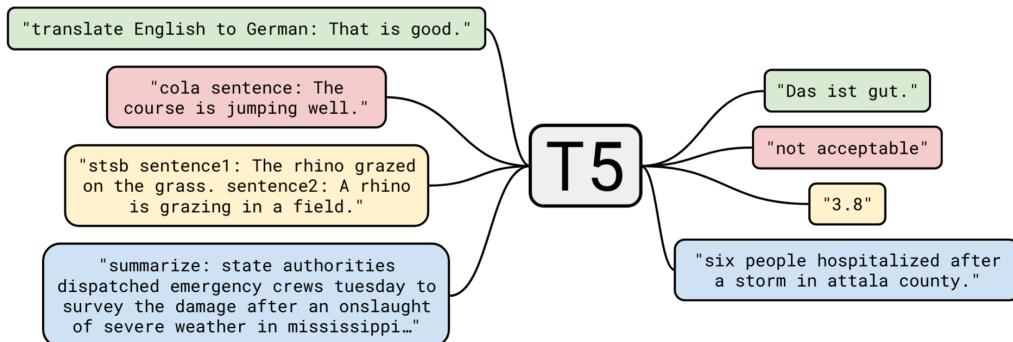


Figura 3.6: El *framework* texto a texto permite emplear el mismo modelo, con los mismos hiperparámetros, función de pérdida, etc., para aplicarlo a diversas tareas de NLP [16]. En esta figura, además de la traducción y el resumen, se recogen tareas basadas en el *Semantic Textual Similarity Benchmark* (STS-B) y el *Corpus of Linguistic Acceptability* (CoLA).

En cualquier caso, se puede realizar un ajuste fino del modelo para una de las tareas, a fin de mejorar su rendimiento en dicha tarea específica.

Las posibilidades que este modelo nos ofrece son muy interesantes, dado que en un futuro, nuestro proyecto podría incluir otras tareas de Procesamiento de Lenguaje Natural, haciendo uso de un solo modelo.

Principales estrategias de generación de resúmenes

JIZT permite al usuario avanzado configurar de manera precisa los parámetros con los que se genera el resumen. En este apartado, exploraremos las diferentes técnicas con las que se pueden generar resúmenes.

La generación de lenguaje, en general, se basa en la auto-regresión, la cual parte del supuesto de que la distribución de probabilidad de una secuencia de palabras puede descomponerse en el producto de las distribuciones de probabilidades condicionales de las palabras sucesivas [37]. Expresado matemáticamente:

$$P(w_{1:t}|W_0) = \prod_{t=1}^T P(w_t|w_{1:t-1}, W_0), \text{ siendo } w_{1:0} = \emptyset$$

donde W_0 es la secuencia inicial de *contexto*. En nuestro caso, esa secuencia inicial va a ser el propio texto de entrada. La longitud de T no se puede conocer de antemano, dado que se corresponde con el momento $t = T$ en el que el modelo genera el *token* de finalización de secuencia (EOS), mencionado anteriormente.

Una vez introducido el concepto de auto-regresión, podemos explicar brevemente las cinco principales estrategias de generación de lenguaje, las cuales se pueden aplicar todas ellas a la generación de resúmenes: búsqueda voraz, *beam search*, muestreo, muestreo *top-k*, y muestreo *top-p*.

Búsqueda voraz

La búsqueda voraz, en cada paso, simplemente selecciona la palabra con mayor probabilidad de ser la siguiente, es decir, $w_t = \operatorname{argmax}_w P(w|w_{t-1})$ para cada paso t .

Por ejemplo, dada la palabra "El", la siguiente palabra elegida sería "cielo", por ser la palabra con mayor probabilidad (0.5), y a continuación "está" (0.5), y así sucesivamente.

Este tipo de generación tiene dos problemas principales:

- Los modelos, llegados a cierto punto, comienzan a repetir las mismas palabras una y otra vez. En realidad, esto es un problema que afecta a todos los modelos de generación, pero especialmente a los que emplean búsqueda voraz y *beam search* [38, 39].

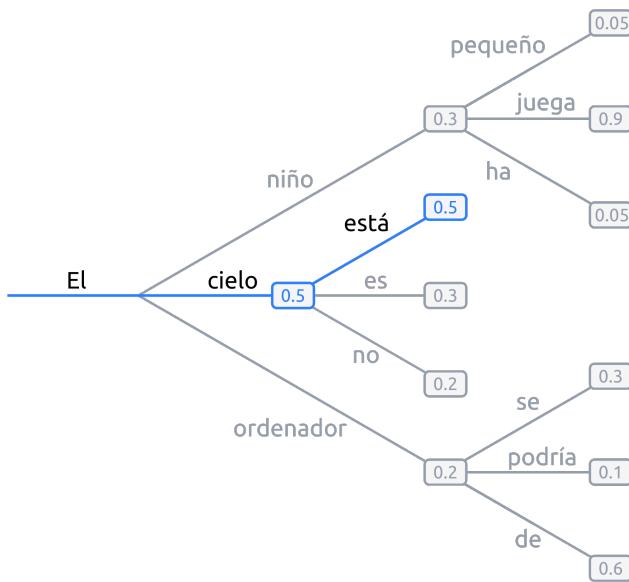


Figura 3.7: Ejemplo de búsqueda voraz: en cada paso, se toma la palabra con mayor probabilidad.

- Palabras con probabilidades altas pueden quedar enmascaradas tras otras con probabilidades bajas. Por ejemplo, en el anterior anterior ejemplo, la secuencia "El niño juega" nunca se dará, porque a pesar de que "juega" presenta una probabilidad muy alta (0.9), está precedida por 'niño', la cual no será escogida por tener una probabilidad baja (0.3).

Beam search

En este caso, durante el proceso de generación se consideran varios caminos simultáneamente, y finalmente se escoge aquel camino que presenta una mayor probabilidad conjunta. En la [siguiente figura](#) se ilustra un ejemplo con dos caminos (`num_beams = 2`).

En este ejemplo vemos que, aunque "cielo" presenta mayor probabilidad que "niño", la secuencia "El niño juega" tiene una mayor probabilidad conjunta ($0,3 \cdot 0,9 = 0,27$) que "El cielo está" ($0,5 \cdot 0,5 = 0,25$), y por tanto será la secuencia elegida.

Este tipo de búsqueda funciona muy bien en tareas en las que la longitud deseada de la secuencia generada se conoce de antemano, como es el caso de la generación de resúmenes, o la traducción automática [40, 41].

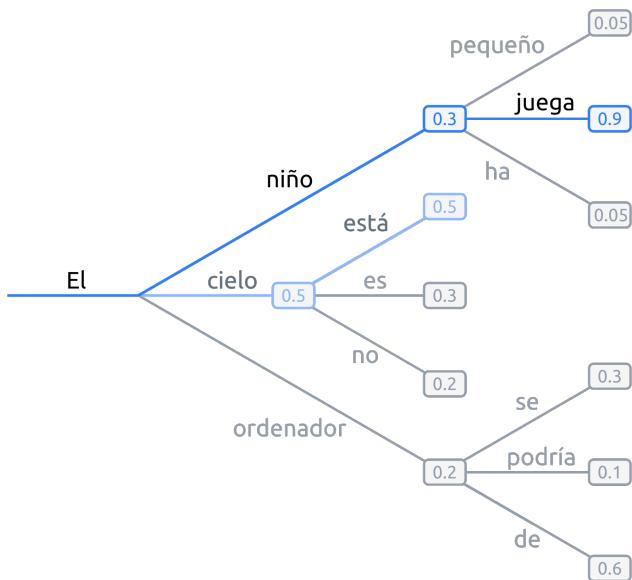


Figura 3.8: Ejemplo de *beam search* con $n_beams = 2$. Durante la búsqueda, se consideran los dos caminos con mayor probabilidad conjunta.

Sin embargo, presenta dos problemas fundamentales:

- De nuevo, aparece el problema de la repetición. Tanto en este caso, como en el de la búsqueda voraz, una estrategia común para evitar dicha repetición, consiste en establecer penalizaciones de *n-gramas* repetidos. Por ejemplo, en el caso de que empleáramos una penalización de 6-gramas, la secuencia "El niño juega en el parque" solo podría aparecer una vez en el texto generado.
- Como se razona en [42], el lenguaje humano no sigue una distribución de palabras con mayor probabilidad. Como vemos en la siguiente gráfica, extraída de dicho artículo, la estrategia de *beam search* puede resultar poco espontánea, dando lugar a textos menos «naturales»:

Muestreo

Es su forma más básica, el muestreo simplemente consiste en escoger la siguiente palabra w_t de manera aleatoria en función de la distribución de su probabilidad condicional, es decir:

$$w_t \sim P(w_t | w_{1:t-1})$$

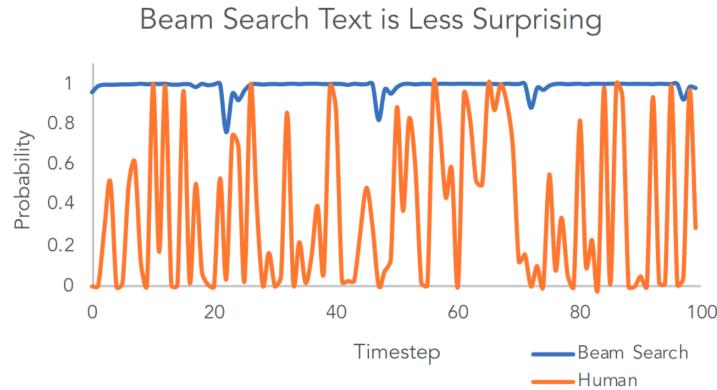


Figura 3.9: Distribución de probabilidades del lenguaje natural frente a la estrategia de *beam search* [42].

De manera gráfica, siguiendo con el ejemplo anterior:

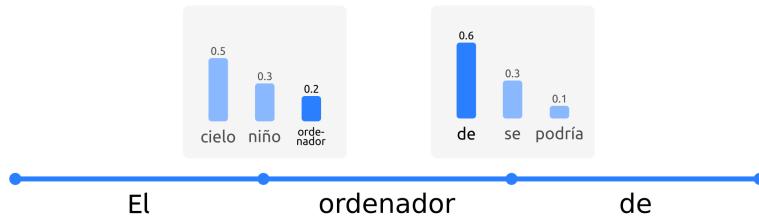


Figura 3.10: Ejemplo de muestreo. En cada paso, se elige una palabra aleatoriamente en función de su probabilidad.

Haciendo uso del muestreo, la generación deja de ser determinista, dando lugar a textos más espontáneos y naturales. Sin embargo, como se estudia en [42], esta espontaneidad es a menudo excesiva, dando lugar a textos poco coherentes.

Una solución a este problema consiste en hacer que la distribución $P(w_t|w_{1:t-1})$ sea más acusada, aumentando la verosimilitud (*likelihood*) de palabras con alta probabilidad, y disminuyendo la verosimilitud de palabras con baja probabilidad. Esto se consigue disminuyendo un parámetro denominado *temperatura*⁶. De esta forma, el [ejemplo anterior](#) queda de la siguiente forma:

⁶ Por motivos de brevedad, no incluiremos una explicación detallada de este parámetro.



Figura 3.11: Al decrementar la temperatura, las diferencias en las probabilidades se hacen más acusadas.

Con este ajuste de la temperatura, logramos reducir la aleatoriedad, pero seguimos manteniendo una orientación no determinista.

Muestreo *top-k*

En este tipo de muestreo, introducido en [43], en cada paso solo se consideran las k palabras con mayor probabilidad (la probabilidad del resto de las palabras será 0).

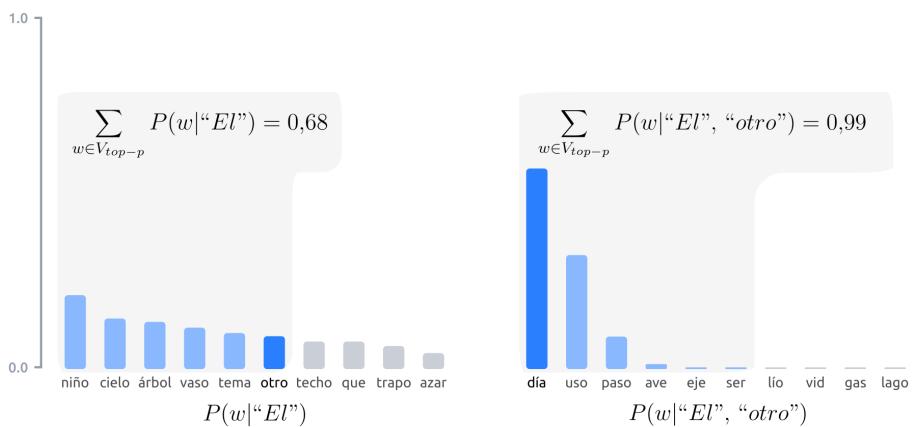


Figura 3.12: Ejemplo de muestreo *top-k*. En cada paso, solo se consideran las 6 palabras con mayor probabilidad.

Tanto la búsqueda voraz como el muestreo visto anteriormente, se pueden como casos particulares del muestreo *top-k*. Si establecemos $k = 1$, estaremos realizando una búsqueda voraz, y si establecemos $k = N$, donde N es la longitud total del vocabulario, estaremos llevando a cabo un muestreo «puro».

Este tipo de muestreo suele producir textos de mayor calidad en situaciones en las que el tamaño de secuencia no está prefijado. Sin embargo, presenta el problema de que el tamaño de k se mantiene fijo a lo largo de la generación. Como consecuencia, en pasos en los que la diferencia de probabilidades sea menos acusada, como en el primer paso de la ??, la espontaneidad del modelo será menor, y en pasos en los que ocurra lo contrario, el modelo será más propenso de escoger palabras que suenen menos naturales, como podría haber ocurrido en el segundo paso de la figura ya mencionada.

Muestreo $top-p$

Este tipo de muestreo, en vez de escoger entre un número prefijado de palabras, en cada paso considera el mínimo conjunto de palabras cuyas probabilidades acumuladas superan un cierto valor p [42].

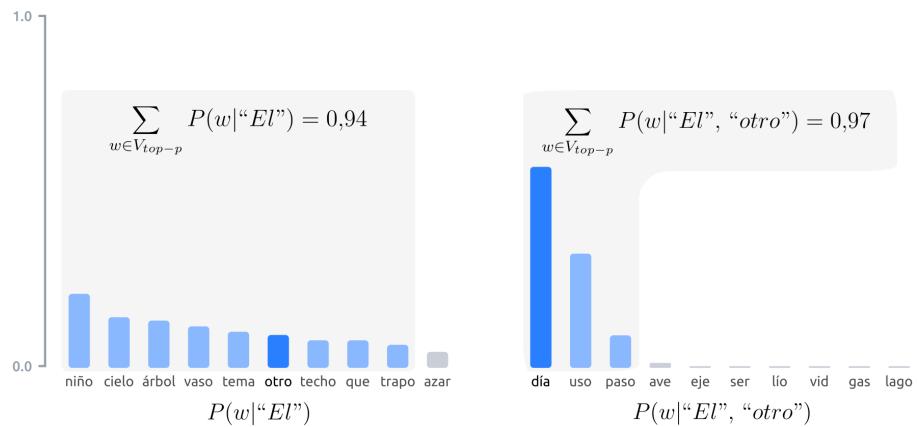


Figura 3.13: Con el muestreo $top-p$, el número de palabras entre las cuales elegir en cada paso varía en función de las probabilidades de las palabras candidatas.

La figura anterior muestra como, con $p = 0,9$, en el primer paso se consideran 9 palabras, mientras que en el segundo solo 3. De este modo, cuando la siguiente palabra a elegir es menos *predecible*, el modelo puede considerar más candidatas, como en el primer paso del ejemplo mostrado y, en el caso contrario, el número de palabras candidatas se reduce.

Los resultados del muestreo $top-k$ u $top-p$ son, en la práctica, similares. De hecho, se pueden utilizar de manera conjunta, a fin de evitar la selección de palabras con probabilidades muy bajas, pero manteniendo cierta variación en el número de palabras consideradas.

3.4. Post-procesado del texto

Como veíamos en la [Figura 3.5](#), el resumen producido por el modelo T5, una vez decodificado, se encuentra todo en minúsculas. Por lo demás, el modelo parece hacer un buen trabajo a la hora de generar el texto en lo que a colocación de puntuación y espacios se refiere, luego la principal labor de esta etapa será poner mayúsculas allí donde sean necesarias, lo que en inglés se denomina *truecasing* [44].

Las mayúsculas, tanto en inglés como español, se emplean principalmente en dos ocasiones:

- Al inicio de cada frase. Como veíamos en la sección referente al [pre-procesado](#) del texto, la separación de un texto en frases no es, por lo general, una tarea trivial. En este caso, podemos reutilizar lo aplicado en dicha etapa. Teniendo el resumen generado dividido en frases, podemos fácilmente poner la primera letra de cada una de ellas en mayúsculas.
- En los nombres propios. En este aspecto, de nuevo vuelve a aparecer el problema del Reconocimiento de Entidades Nombradas (NER). De modo similar a como procedíamos en el pre-procesado, emplearemos un modelo estadístico que realiza la labor de *truecasing*.

Tras esta etapa, el resumen está listo para ser entregado al usuario.

Técnicas y herramientas

En este capítulo, se recogen las tecnologías principales empleadas en el desarrollo del proyecto, así como los detalles más relevantes de su implementación.

Para facilitar la organización y comprensión de las mismas, se han separado en tres subsecciones: *Modelo*, *Backend* y *Frontend*.

4.1. Modelo

Como se ha venido mencionando a lo largo de los anteriores capítulos, nuestro proyecto, a la hora de generar resúmenes, solo hace uso del modelo T5 de Google [16] por el momento. Más concretamente, utilizamos la implementación `t5-large` Hugging Face [45], el cual ha sido entrenado con texto en inglés, procedente del Colossal Clean Crawled Corpus (C4), y contiene aproximadamente 770 millones de parámetros [46].

Esta implementación está escrita en Python, lo que nos facilita la integración con el resto de componentes de JIZT, también desarrollados en Python.

El modelo `t5-large` consta, por un lado, del *tokenizer*, encargado de la codificación del texto, y por otro, el modelo en sí, el cual recibe el texto codificado por el *tokenizer*, y genera el resumen a partir de él. Dicho resumen, sigue estando en forma de *tókenes* codificados, por lo que tenemos que hacer uso una vez más del *tokenizer* para proceder a su decodificación. Una vez decodificado, el texto vuelve a contener caracteres legibles.

Tanto el proceso de codificación, como el de generación de resúmenes, se pueden llevar a cabo empleando unidades de procesamiento gráfico (GPU).

No obstante, en nuestro caso, ambos procesos se ejecutan en unidades centrales de procesamiento (CPU), debido a limitaciones económicas⁷. Esto explica en parte los **tiempos de resumen obtenidos**.

Un último aspecto a destacar es que a la hora de generar los resúmenes, se pueden especificar los parámetros concretos con los que realizar dicha generación, permitiéndonos hacer uso de las diferentes estrategias vistas en la [Sección 3.3](#).

4.2. *Backend*

En la [Figura 4.14](#) se recoge una visión general de la arquitectura que conforma el *backend* de JIZT, y que posibilita la implementación en la nube de las diferentes etapas en la generación de resúmenes descritas en el capítulo de [Conceptos teóricos](#).

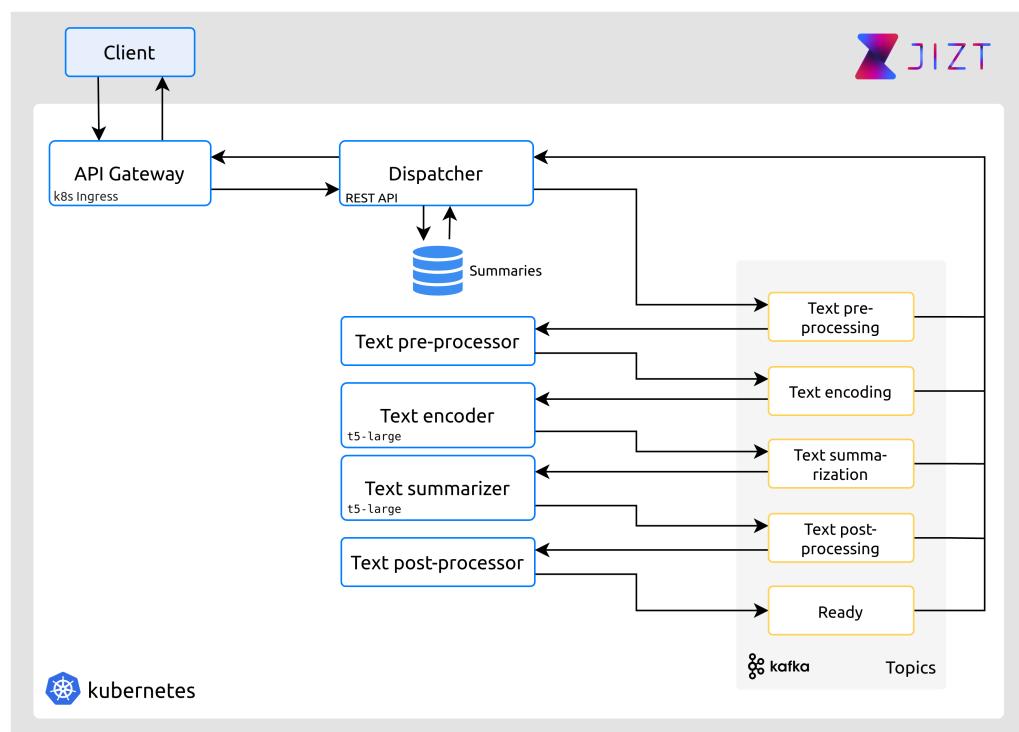


Figura 4.14: Vista general de la arquitectura del *backend*.

⁷Cabe recordar que los modelos se ejecutan en «la nube». Contratar equipos que dispongan de GPU aumentaría notablemente los costes.

En esta arquitectura, existen diferentes tecnologías, cada una encargada de realizar una tarea específica, pero a su vez integrándose con el resto. Veamos con más detalle cuáles son las características principales de dichas tecnologías.

Kubernetes

El *backend* sigue una arquitectura de microservicios [47], de forma que cada una de las etapas (pre-procesado, codificación, generación del resumen y post-procesado), está confinada en un contenedor Docker [48], conformando un microservicio. Adicionalmente, existe un microservicio más, el *Dispatcher*, el cual lleva a cabo las siguientes tareas:

- Implementa una API REST que permite a los clientes solicitar resúmenes.
- Gestiona una base de datos en la que se almacenan los resúmenes generados.
- Redirige las peticiones de los clientes al microservicio apropiado. Por ahora, todas las peticiones se redirigen hacia el pre-procesador de textos, pero en un futuro podría existir otro microservicio que se encargara, por ejemplo, de extraer el texto de un documento PDF o de una página web. En estos casos, el *Dispatcher* se encargaría de redirigirlo hacia el microservicio correspondiente.

Kubernetes es una plataforma *open-source* destinada a la gestión de servicios y cargas de trabajo en contenedores, que facilita su automatización en cuanto a aspectos como el escalado, gestión de red y recursos, monitorización, etc. [49]

Kubernetes comprende numerosos componentes, entre los cuales, los más relevantes en nuestro caso son:

- *Pod*: es la unidad de computación básica en Kubernetes. Un *Pod* puede ejecutar uno o varios contenedores intrínsecamente relacionados (compartirán almacenamiento, red, recursos, etc.).
- *Deployment*: los *deployments* se pueden ver como «plantillas» o «moldes» que contienen los detalles específicos para crear *pods* de un determinado tipo. Por ejemplo, en el caso del mencionado *Dispatcher*, dispondremos de un *deployment* que indicará cómo se deben crear los *pods* para este servicio. Estos *pods* a su vez, contendrán todos la misma imagen Docker que implementará la lógica del servicio.

- *Service*: cada *pod* dispone de una dirección IP propia. Sin embargo, los *pods* tienen un ciclo de vida *efímero*, dado que están concebidos para ser reemplazados dinámicamente si se producen errores, actualizaciones, etc. Por tanto, no podemos basar la configuración de red en las IPs específicas de los *pods*, ya que éstos son susceptibles de cambiar a lo largo del tiempo, según los *pods* vayan siendo reemplazados. Los *services* nos permiten asociar una IP fija y persistente a un conjunto concreto de *pods*. A la hora de realizar una conexión con dicha IP, Kubernetes se encarga de remitir los datos al *pod* que esté menos ocupado en ese instante, realizando por tanto un balance de carga de forma automática.
- *PersistentVolume*: al igual que en el caso de las IPs, los datos almacenados localmente en un *pod* desaparecerán cuando este sea reemplazado. Los *PersistentVolumes* nos proporcionan la capacidad de almacenar datos de manera persistente, independientemente del ciclo de vida de los *pods*. Nosotros, utilizamos este componente para almacenar los modelos de generación de resúmenes, ya que ocupan alrededor de 5 GB, de forma que los *pods* correspondientes a la codificación de texto y generación de resumen consumen los modelos desde una única fuente de datos, el *PersistentVolume*. Incluir los modelos dentro de los propios *pods* sería contraproducente porque (a) todos los *pods* van a hacer uso de los mismos modelos, y (b) los modelos ocupan alrededor de los 5 GB, por lo que si quisieramos crear varios *pods*, la demanda de almacenamiento crecería rápida e innecesariamente.

La Figura 4.15 pretende facilitar la comprensión de los diferentes componentes de manera más visual. Como podemos ver en dicha figura, existen n *pods*, todos ellos replicas de un mismo *deployment* y, por tanto, ejecutando los mismos contenedores, pero cada uno de ellos con una dirección IP propia. El *service* permite acceder a los diferentes *pods* a través de una única IP estática. Por último, todos los *pods* consumen un mismo *PersistentVolume* que, por ejemplo, podría contener los modelos ya mencionados.

De este modo, podemos escalar (o actualizar) cada uno de los micro-servicios de forma dinámica y sin períodos de inactividad (*downtime*). De hecho, Kubernetes permite configurar el escalado de manera automática. Así, en momentos en los que la carga de trabajo sea mayor, se crearán *pods* adicionales para responder ante dicha carga y, una vez esta desaparece, se volverán a eliminar. Al habilitar esta opción, es muy recomendable configurar el número máximo de *pods* que se podrán crear, a fin de evitar un escalado

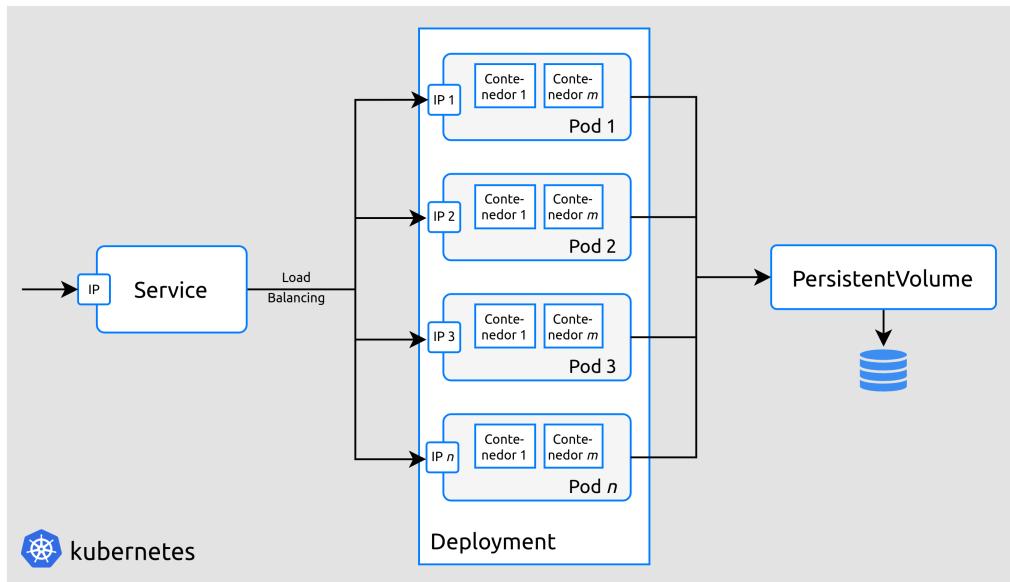


Figura 4.15: Componentes principales de Kubernetes.

descontrolado en momentos de carga extrema (en cualquier caso, Kubernetes detendría la creación de *pods* tan pronto como se consumieran los recursos del sistema disponibles [50]).

Existe un último componente de Kubernetes del que hacemos uso, llamado Ingress. Este componente implementa una API *Gateway*, enruteando las peticiones API de los clientes hacia el microservicio correspondiente [51]. Por ahora, la API REST que hemos implementado solo dispone de rutas relacionadas a la generación de resúmenes, pero en un futuro, cuando se implementen otras tareas de NLP, existirán otros *endpoints* para dichas tareas. Ingress se encargará entonces de, en función de a qué *endpoint* se esté realizando la petición, redirigirla al microservicio correspondiente.

Kafka y Strimzi

Uno de los principales aspectos a considerar a la hora de implementar una arquitectura de microservicios reside en la estrategia que se va seguir para permitir la comunicación entre los diferentes microservicios.

Dicha comunicación puede llevarse a cabo de forma síncrona, por ejemplo a través de peticiones HTTP, o asíncrona, con tecnologías como Apache Kafka [52].

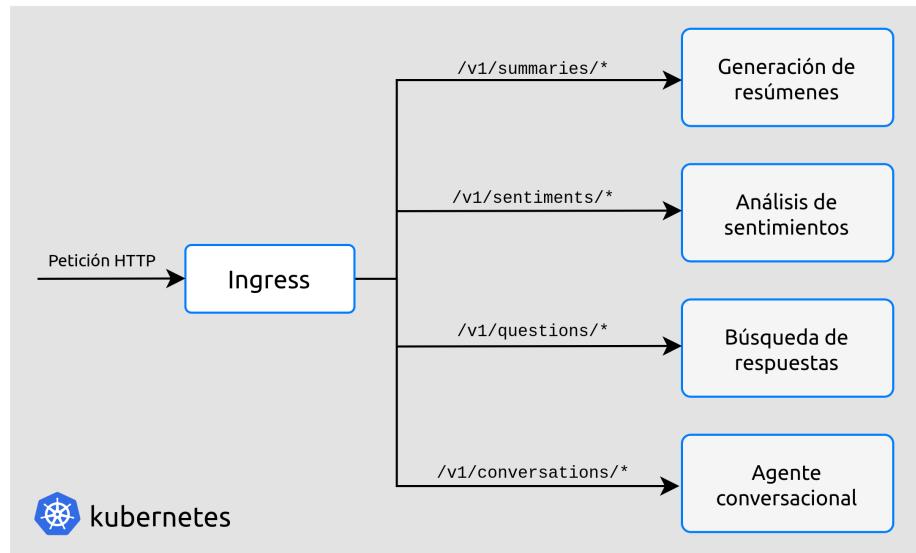


Figura 4.16: Ejemplo de uso de Ingress con diferentes rutas.

En nuestro caso la comunicación síncrona quedó rápidamente descartada, dado que la generación de resúmenes presenta tiempos de latencia que pueden ser elevados (del orden de segundos o incluso minutos).

Apache Kafka nació internamente en LinkedIn, aunque actualmente es *open-source* y su desarrollo corre a cargo de la Apache Software Foundation [53].

Kafka permite el intercambio asíncrono de mensajes entre productores y consumidores. En esencia, su funcionamiento es conceptualmente sencillo y está alineado con tecnologías más tradicionales: los consumidores se suscriben a un tema (*topic*), a los que los productores envían sus mensajes. La consumición de dichos mensajes es asíncrona.

La novedad de Kafka reside entre otras cosas, en su gran capacidad de escalado, soportando billones de mensajes al día; su funcionamiento distribuido, permitiendo fácilmente operar a lo largo de diferentes zonas geográficas; su gran fiabilidad en entornos críticos, en los que la pérdida de un solo mensaje es inadmisible; o su tolerancia frente a fallos [54].

Todas estas demandas no suponen, sin embargo, que Kafka no se pueda aplicar de igual modo a entornos más reducidos, como es el nuestro. Además, gracias a Strimzi, otro proyecto también *open-source*, el despliegue de Kafka en Kubernetes se simplifica en gran medida.

Si volvemos a observar la [figura](#) que incluíamos al principio de esta sección, podemos ver que JIZT dispone de cinco *topics*, los cuales se corresponden con cada una de las etapas en la generación resúmenes:

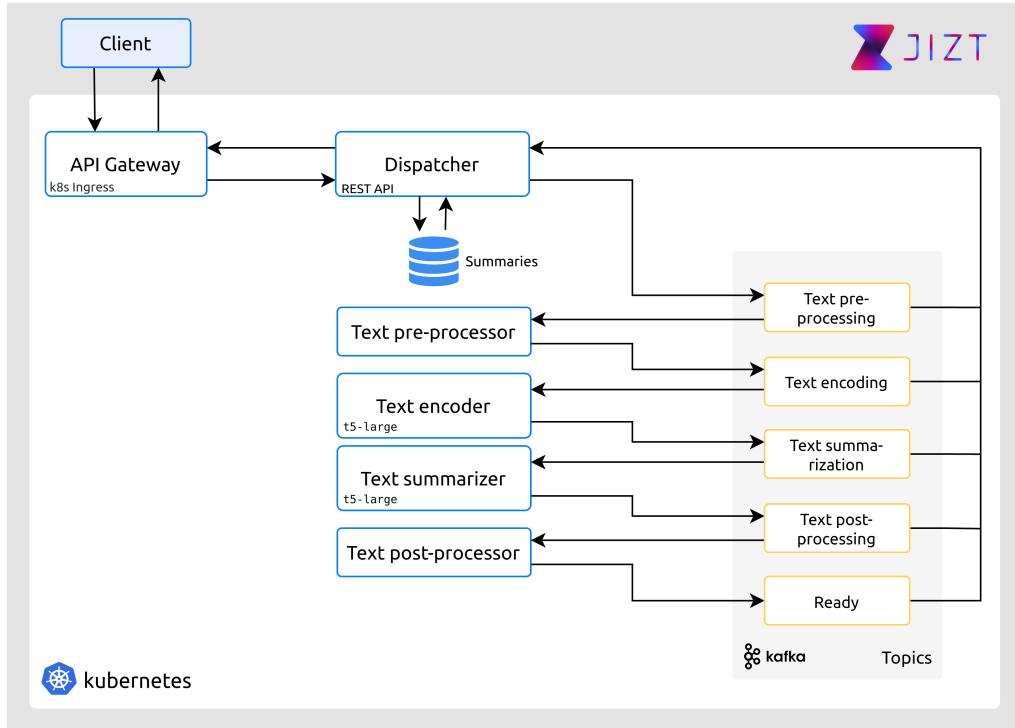


Figura 4.17: Vista general de la arquitectura del *backend*.

Con esta figura en mente, el proceso completo que se sigue es el siguiente:

1. El cliente realiza una petición HTTP, incluyendo en el cuerpo el texto a resumir, así como los parámetros del resumen a generar.
2. Ingress (API *Gateway*) comprueba que dicha petición se está haciendo a un *endpoint* válido, y en ese caso la redirige hacia el *Dispatcher*.
3. El *Dispatcher* realiza una serie de comprobaciones:
 - a) Si la petición no contiene ningún texto, se devuelve un error. En el caso de los parámetros, si son incorrectos o inexistentes, se ignoran y se utilizan valores por defecto.
 - b) Se consulta en la base de datos si ya existe un resumen generado para ese texto con esos parámetros. En ese caso, lo devuelve directamente, sin generar de nuevo el resumen.

- c) En caso contrario, produce un mensaje al *topic* del pre-procesador de textos, conteniendo el texto y los parámetros del resumen.
- 4. El pre-procesador está constantemente comprobando si existen mensajes nuevos en su *topic*. En ese caso los consume, realiza las tareas de pre-procesado, y produce el resultado en el *topic* del codificador.
- 5. Este proceso continua de forma análoga hasta llegar al post-procesador, el cual produce el resumen final al *topic* «Listo» (*Ready*). El *Dispatcher*, en ese momento, consume el mensaje, actualiza la base de datos, y proporciona el resumen al cliente.

En dicha [figura](#), vemos también que el *Dispatcher* consume de todos los *topics*. Esto permite actualizar el *estado* del resumen (pre-procesando, resumiendo, post-procesando, o listo), según va pasando por las diferentes etapas, a fin de proporcionar una retroalimentación más detallada al usuario⁸.

Finalmente, cabe destacar una vez más la facilidad de escalado que nos proporciona Kafka: si, por ejemplo, ampliásemos nuestra arquitectura de modo que tuviéramos tres réplicas de cada microservicio, Kafka se encargaría automáticamente de coordinar la producción y consumición de mensajes de cada *topic*, sin que nosotros tuviéramos que llevar a cabo ninguna acción adicional.

Helm

Helm se define frecuentemente como un gestor de paquetes para Kubernetes, aunque en la práctica va más allá.

La configuración de Kubernetes se lleva a cabo, principalmente, de forma declarativa a través de ficheros en formato `yml`, lo que en inglés se conoce como *templating*. Nuestro proyecto, el cual es relativamente pequeño, hace uso de más de 20 de estos ficheros de configuración. Es fácil imaginarse, por tanto, que un proyecto de mediana escala contendrá cientos de *templates*.

Helm permite, a través de un único comando, desplegar todos estos componentes de forma automática, gestionando aspectos como el orden en el que se crean los componentes, el cual en muchos casos no es trivial. Una vez instalados, a través de otro comando, podemos actualizar los posibles cambios que haya sufrido alguno de los *templates*, de forma que solo afecte

⁸ Por ahora, el *Dispatcher* solo muestra el estado «resumiendo». El resto de estados se implementarán en futuras iteraciones.

a los componentes involucrados en dichas modificaciones, y sin tiempos de interrupción.

Además, a través de las llamadas *Library Charts* [55], Helm nos permite generar una plantilla que varios componentes pueden reutilizar. Esto es muy apropiado en nuestro caso dado que todos nuestros microservicios tienen una estructura similar; lo único que cambia es la imagen (contenedor) que implementan.

Una última ventaja es que podemos distribuir el *backend* de JIZT como un único paquete, facilitando su instalación por parte de otros desarrolladores.

Crunchy PostgreSQL Operator

De igual modo que Strimzi facilita el despliegue de Kafka en Kubernetes, el operador para PostgreSQL de Crunchy automatiza y simplifica el despliegue de *clusters* PostgreSQL en Kubernetes [20].

De este modo, podemos implementar una base de datos que almacene los resúmenes generados⁹, con dos propósitos principales: (a) servir como capa de caché, evitando tener que producir el mismo resumen en repetidas ocasiones, y (b) construir un *dataset* que se podría utilizar en un futuro para tareas de evaluación, o incluso para el entrenamiento de otros modelos.

Este operador coordina de forma automática los accesos a la base de datos, asegurando la integridad de la misma. Esto es posible dado que solo existe un única instancia (*pod*) con capacidades de escritura-lectura. El resto de instancias que accedan a la base de datos, solo podrán leer de la misma. Si la instancia primaria fallara, el operador se encargaría inmediatamente de elegir otra instancia como primaria.

Docker

Docker nos permite encapsular nuestros microservicios en contenedores. De este modo, gracias a Kubernetes, podemos crear réplicas de cada microservicio, haciendo posible el escalado de nuestro sistema.

A diferencia de las máquinas virtuales, en las cuales el sistema operativo subyacente se comparte a través del hipervisor, cada contenedor Docker ejecuta su propio sistema operativo, como podemos ver en la [siguiente figura](#):

⁹ Una de las futuras historias de usuario implementará un «modo privado», de forma que los usuarios tengan la posibilidad de generar sus resúmenes sin que se almacenen de manera permanente.

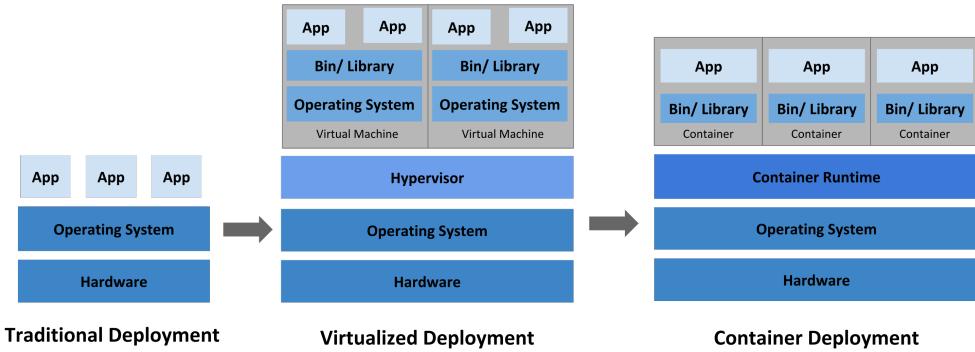


Figura 4.18: Comparativa de los diferentes enfoques en el despliegue de sistemas [49].

Otra ventaja de Docker es que nos permite distribuir la implementación de nuestros microservicios a través imágenes, por lo que un desarrollador que solo quisiera hacer uso de uno de los microservicios, podría hacerlo de manera sencilla.

Flask y Flask-RESTful

Flask es uno de los *frameworks* más populares para la creación de aplicaciones *web* en Python [56], concebido para ser lo más simple posible. En nuestro caso, hemos empleado esta herramienta para implementar la lógica de la API REST. Además, hemos utilizado una conocida extensión de Flask, Flask-RESTful [57], orientada a la construcción de APIs REST, como es nuestro caso.

Dado que es el *Dispatcher* quien implementa la REST API, es únicamente este microservicio el que hace uso de este *framework*.

4.3. *Frontend*

Flutter

Flutter es un *kit* de herramientas de UI (interfaz de usuario) que, a partir del mismo código fuente base, permite compilar de forma nativa aplicaciones para móvil, *web* y escritorio [58], lo cual permite [59]:

- Un desarrollo más rápido, dado que solo se trabaja en una única base de código.

- Costes más bajos, ya que solo mantenemos un proyecto en vez de varios.
- Una mayor consistencia, proporcionando al usuario la misma interfaz gráfica y herramientas en las distintas plataformas, conservando los patrones de interacción de cada una de ellas.

Pese a ser desarrollado por Google desde su nacimiento en 2017, Flutter cuenta en la actualidad con un gran apoyo de la comunidad *open-source*. Esto ha facilitado la resolución de dudas y errores a la hora de desarrollar nuestra aplicación.

Flutter emplea el lenguaje Dart, el cual guarda similitudes con otros lenguajes como Java o C#. Existen numerosos aspectos de Flutter y Dart que cabría explicar; no obstante, en pos de la brevedad introduciremos uno de los que más interesantes y relevantes nos parecen para este proyecto: ¿Cómo se consigue que Dart pueda ser ejecutado nativamente en plataformas que pueden resultar tan dispares como Android, iOS, *web*, Windows o Linux?

Para responder a esta pregunta, es importante comenzar indicando que en Flutter, en el entorno de desarrollo se opera de manera diferente al entorno de producción.

Veamos cuáles son las diferencias principales.

Desarrollo nativo (plataformas x64/ARM)

Así como Java requiere de la JVM (*Java Virtual Machine*) para ejecutarse, Dart también dispone de su propia DVM (*Dart Virtual Machine*).

Durante el desarrollo, la máquina DVM se utiliza en combinación con un compilador JIT (*Just In Time*), es decir, se lleva a cabo una compilación en tiempo de ejecución, en lugar de antes de la ejecución. Esto permite tratar con el código de forma dinámica independientemente de la arquitectura de la máquina del usuario.

Además, esta forma de operar, hace posible lo que se conoce como *hot reload*, que permite ver los cambios realizados en la aplicación de manera prácticamente instantánea, dado que los cambios en el código se transfieren a la DVM, pero se conserva el estado de la *app* [60]. Esto decrementa notablemente los tiempos empleados en el *debug* de las aplicaciones.

Desarrollo *web*

Durante el desarrollo, el compilador de desarrollo Dart, conocido como `dartdevc`, permite ejecutar y depurar aplicaciones *web* Dart en Google Chrome. Usado en combinación con otras herramientas como `webdev`, el cual proporciona un servidor *web* de desarrollo, podemos visualizar en nuestro navegador los cambios realizados en el código fuente de manera casi inmediata.

Producción nativa (plataformas x64/ARM)

En este caso se emplea lo que se conoce como compilación anticipada (AOT, *Ahead-of-time Compilation*). Gracias a este tipo de compilación, Flutter es capaz de traducir un lenguaje de alto nivel, como en este caso Dart, a código máquina x64/ARM nativo [61]. Este código máquina sí que va a ser dependiente del sistema.

Como consecuencia de lo anterior:

- En este caso ya no es necesario emplear una DVM, ya que con la compilación AOT obtenemos, para cada plataforma, un único binario ejecutable (`.apk` o `.aab` para Android, `.exe` para Windows, etc.).
- La compilación AOT es lo que realmente convierte a Flutter en una herramienta rápida y portable.

Producción *web*

El código Dart también puede ser traducido a HTML, CSS y JavaScript (en el caso de este último gracias a una herramienta llamada `dart2js`).

Esto significa que podemos ejecutar nuestra aplicación en Chrome o Firefox¹⁰, y la interfaz gráfica será la misma que en el resto de plataformas.

Es importante mencionar, que el soporte para *web* de Flutter se encuentra aún en fase *beta*, por lo que no se recomienda para producción [62]. No obstante, nosotros no hemos experimentado problemas con nuestra aplicación en ninguno de los navegadores soportados.

¹⁰ Por ahora, solo Chrome, Safari, Edge y Firefox están soportados, este último solo en su versión de escritorio con WebGL habilitado.

Aspectos relevantes del desarrollo del proyecto

A la hora de desarrollar el proyecto, nos encontramos con una serie de decisiones a tomar, retos, y cuestiones que tratamos de solucionar a través de formación y aplicación de mejores prácticas. Esta labor fue relativamente sencilla gracias a la disponibilidad de recursos *online* existente hoy en día, así como la participación activa y entusiasmo de la comunidad detrás de las herramientas empleadas en este proyecto.

En este apartado cubrimos los aspectos más destacados en este respecto.

5.1. Metodología de desarrollo *software*: Kanban

El contexto y características en los que se enmarcaba nuestro proyecto son las siguientes:

- **Tiempo limitado:** no cabía posibilidad de alargar los plazos, y existían importantes restricciones de tiempo (disponíamos de aproximadamente tres meses para la compleción del proyecto).
- **Eficiencia y velocidad:** debido a las restricciones mencionadas en el punto anterior, la implementación del proyecto había de ser rápida y eficiente, asegurando siempre un nivel de calidad óptimo.
- **Motivación y progreso del proyecto:** requeríamos de una metodología que estimulase de manera natural la inversión de tiempo y esfuerzo en el proyecto.

- **Satisfacción de los usuarios:** dado que ellos son la razón por la que este proyecto se desarrollaba en primer lugar.

Atendiendo a los puntos descritos anteriormente, decidimos adoptar un enfoque ágil para el desarrollo de nuestro proyecto. Dentro de las metodologías ágiles, se han consideraron las siguientes:

- **Scrum:** desarrollo iterativo e incremental centrado en la idea de *sprints*, es decir, iteraciones con una duración fija prefijada, al final de las cuales se produce una entrega parcial del producto.
- **Kanban:** esta metodología se centra en mantener un flujo constante de trabajo, maximizando la eficiencia del equipo de forma que cada tarea sea completada con la mayor celeridad posible.
- **Programación Extrema (XP):** se centra en producir *software* de la mejor calidad posible, siendo una de las metodologías ágiles que más profundiza en los aspectos de buenas prácticas de ingeniería para el desarrollo de software.

Tras una valoración de las ventajas y desventajas de cada una de estas metodologías, decidimos adoptar la metodología Kanban, gracias también a su mayor flexibilidad. Esto resultó de gran ayuda, dado que, al comenzar este proyecto desconocíamos el funcionamiento concreto de muchas de las herramientas y técnicas utilizadas, por lo que habría sido muy difícil producir ciclos de desarrollo predefinidos y cerrados, como en el caso de Scrum.

No obstante, sí que tomamos ciertos elementos interesantes de esta otra metodología, Scrum, acercándonos en cierto modo a lo que se conoce como Scrumban, una metodología híbrida entre Scrum y Kanban.

A continuación, se recogen las principales características de nuestro sistema de trabajo:

- Empleamos un tablero Kanban para organizar las historias de usuario, y hacemos uso de instrumentos como los límites WIP (*Work In Progress*) y de herramientas como los Diagramas de Flujo Acumulado (CFD, por sus siglas en inglés).
- Definimos *epics* para agrupar historias de usuario que conformen una misma *feature*, o funcionalidad a desarrollar.

- Debido a la naturaleza de Kanban, no existen *sprints* como tal: el flujo de trabajo es continuo. Sin embargo, sí que se definen tiempos estimados para completar cada tarea (sin emplear puntos de historia; los tiempos se expresan en número de horas empleadas).
- Cada tarea tiene asociada una complejidad, que va desde 0 (mínima complejidad), hasta 10 (máxima complejidad).
- Cada tarea tiene asociada una prioridad. Los niveles de prioridad van desde 0 hasta 3, siendo esta última la prioridad máxima.
- Además del *product backlog*, en el que se recogen las futuras historias de usuario a desarrollar, contamos con otras cuatro columnas: *preparado*, *trabajo en progreso*, *testing*, y *finalizado*.
- Periódicamente, se llevaron a cabo *revisiones* y *retrospectivas*, en las que participaron los tutores.

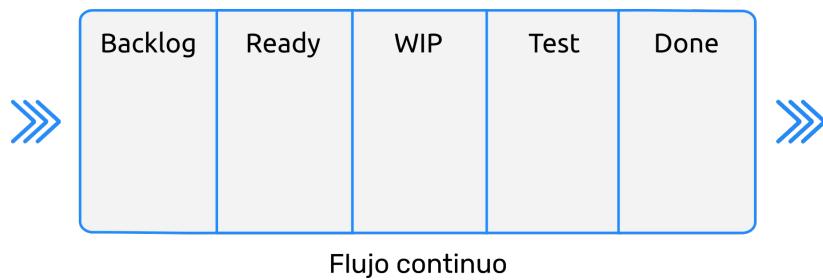


Figura 5.19: Tablero Kanban utilizado.

Como herramienta de gestión Kanban, empleamos Kanboard [63]. Se trata de una aplicación *web open-source* activamente desarrollada. Contratamos un servidor EC2 con Amazon Web Services (AWS) desde el cual podemos servir la aplicación *web*, la cual a su vez hace uso de una base de datos PostgreSQL en la cual almacena los datos generados. Dicha base de datos está desplegada a través del servicio RDS, también perteneciente a AWS.

Se puede acceder al tablero público a través de <https://kanban.jizt.it>.

5.2. Motivación tras las arquitecturas desarrolladas

Arquitectura de microservicios

Desde un primer momento, se concibió la arquitectura con los siguientes objetivos presentes:

- **Flexibilidad:** la Inteligencia Artificial y, en concreto, el Procesamiento de Lenguaje Natural, son campos en continuo desarrollo. Cada pocos meses aparecen modelos más potentes que proporcionan mejores resultados. Es por ello que nuestra arquitectura debe proporcionar una estructura lo más desacoplada como sea posible de los modelos concretos de NLP que empleados. De este modo, si aparecieran modelos más avanzados, la transición de unos modelos a otros resultará una labor relativamente sencilla.
- **Escalabilidad:** los elementos que conforman la arquitectura, deben tener la capacidad de replicarse a fin de responder correctamente a la demanda de usuarios. Adicionalmente, como se ha venido mencionado a lo largo de esta memoria, la implementación de otras tareas de NLP diferentes de la generación de resúmenes es algo que entra dentro de nuestros planes a medio plazo. La arquitectura debe estar estructurada de tal forma que esta expansión se pueda llevar a cabo sin inconvenientes.
- **Alta disponibilidad:** relacionada con el punto anterior, se debe poder prestar servicio de forma continua, independientemente de que se produzcan picos en la carga de trabajo, o de que alguno de los componentes falle en un momento dado.
- **Cloud native:** este punto engloba a todos los anteriores; los sistemas *cloud-native* están diseñados para adaptarse a entornos cambiantes, operar a gran escala y poseer resiliencia [17].

Una de las arquitecturas que permiten conseguir los objetivos recogidos anteriormente, es la **arquitectura de microservicios**. Con este patrón arquitectónico, la aplicación se divide en pequeños servicios, cada uno de los cuales cumple una labor específica, y encapsula todas sus dependencias, a fin de conseguir el máximo grado de independencia posible.

En nuestro caso, además, existen tareas que llevan considerablemente más tiempo que otras, como es el caso de la generación del resumen (que puede

5.2. MOTIVACIÓN TRAS LAS ARQUITECTURAS DESARROLLADAS

durar segundos), frente al pre-procesado del texto (el cual es instantáneo). Una arquitectura como esta nos permite replicar el microservicio encargado de la generación del resumen, para repartir la carga de trabajo entre las diferentes réplicas.

Además, si uno de los microservicios fallara, sería reemplazado inmediatamente por una nueva réplica, gracias a la tecnología de Kubernetes.

Arquitectura dirigida por eventos

Dado que ya ha sido introducida en la [sección referente a Kafka](#), no entraremos en mucho detalle para evitar repetirnos.

Simplemente recordaremos que este patrón arquitectónico hace posible la comunicación entre los microservicios de forma fiable y rápida. En nuestro caso, un evento sería la finalización del trabajo por parte de uno de los microservicios. Este evento genera una respuesta en otro de los microservicios, el cual lo procesa y comienza su labor específica.

Este patrón nos ofrece también flexibilidad a la hora de introducir nuevos microservicios, ya que, al menos en el caso de Kafka, el *topic* al que un microservicio produce (o consume) eventos podría ser modificado en tiempo de ejecución, sin necesidad de alterar el código fuente del microservicio.

API REST Asíncrona

La generación de resúmenes es un proceso que se puede dilatar varios segundos en el tiempo, dependiendo de factores como la longitud del texto o de los parámetros con los que se genere el resumen. Por lo tanto, realizar peticiones síncronas queda descartado, puesto que una petición HTTP no debe prolongarse durante tanto tiempo.

La forma común de solucionar este problema, logrando asincronismo, pasa por realizar una primera petición dándole a conocer al sistema que queremos generar un resumen. El sistema, entonces, responderá haciéndonos saber que la petición ha sido recibida y se está procesando. A partir de ese momento, consultaremos periódicamente al sistema para conocer el estado del resumen, hasta finalmente obtenerlo, una vez haya sido generado.

Veamos el proceso de manera un poco más detallada.

1. Petición HTTP POST

El cliente comienza realizando una petición POST incluyendo en el cuerpo de la misma el texto que quiere resumir. La API le responde con un identificador único del resumen, el `summary_id`, así como otros campos de interés:

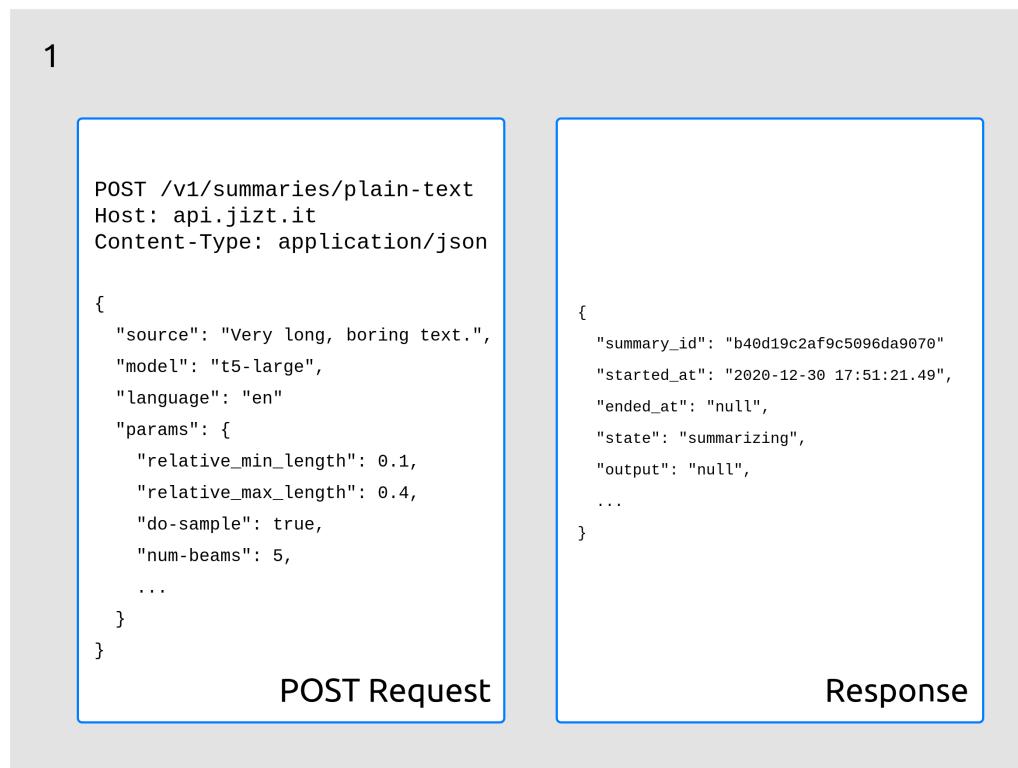


Figura 5.20: El primer paso es realizar una petición POST con el texto a resumir.

Como vemos en la [anterior figura](#), el estado del resumen es "**resumiendo**" ("`summarizing`"), y aún no tenemos acceso al resumen (`output`), el cual es por el momento "`null`".

Una de las principales ventajas de poder consultar el estado del resumen, es poder ofrecer al usuario retroalimentación de los pasos que se están llevando a cabo, mostrándole así que su resumen efectivamente está siendo procesado.

2. Peticiones HTTP GET sucesivas

En ese momento, el cliente puede llevar a cabo peticiones HTTP GET con el *id* del resumen de manera periódica a fin de consultar el estado del mismo.

En algún momento, el estado del resumen pasará a ser "completado" ("completed"), y la respuesta a nuestra petición contendrá el resumen generado:

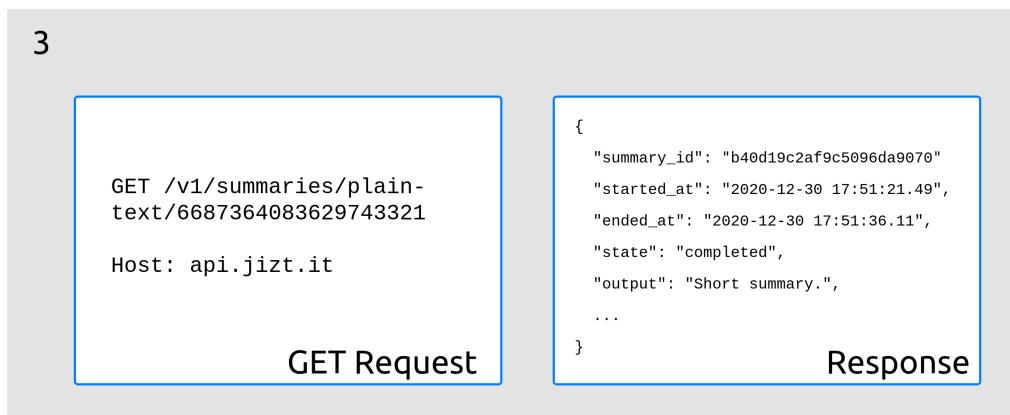


Figura 5.21: Finalmente, obtenemos el resumen generado.

En el caso de que previamente se hubiera solicitado un resumen del mismo texto, con el mismo modelo y parámetros, el resumen ya estaría almacenado en la base de datos, por lo que la respuesta al primer POST ya contendría dicho resumen.

Desarrollo de la aplicación

A la hora de desarrollar la aplicación, se ha dado gran importancia al diseño de la arquitectura. Con tal fin, nos hemos basado en varios patrones de diseño, dando lugar a una arquitectura que toma elementos de los patrones BLoC [59], *Domain-Driven Design* [64] o *Clean Architecture* [21].

Antes de explicar qué significan estos conceptos, veamos cómo se conforma la arquitectura de la aplicación:

Como podemos ver, la arquitectura se divide en cuatro capas: presentación, aplicación, dominio y datos, siendo las primeras las más cercanas al usuario.

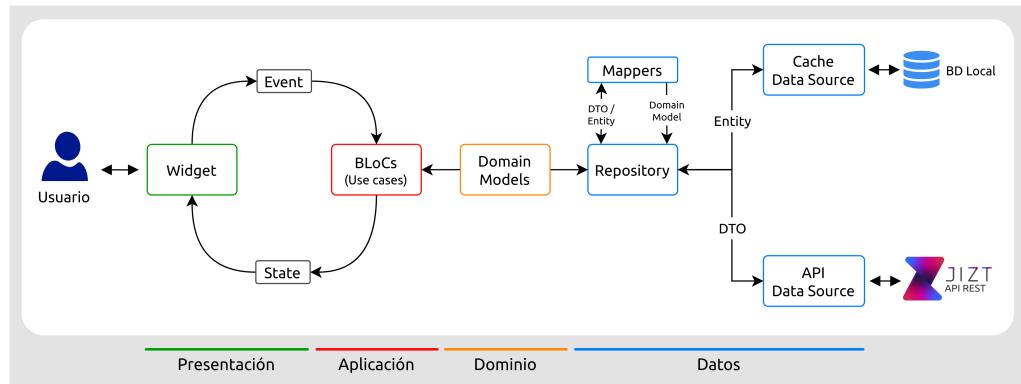


Figura 5.22: Arquitectura de la aplicación.

Expliquemos de forma más detallada cada una de ellas, comenzando por la capa de *datos*, a la derecha de la imagen.

Capa de datos

En esta capa se hace uso del patrón repositorio, el cual sitúa un componente intermedio entre la capa de dominio y la fuente de los datos a fin de aislar los objetos del dominio de los detalles de implementación concretos a la hora de acceder a la fuente de datos [64]. Es importante mencionar que el repositorio, se *implementa* en la capa de datos, pero se *define* en la capa de dominio. De esta forma, conseguimos que la capa de dominio sea a su vez independiente de la implementación concreta de la capa de datos.

Nuestra aplicación implementa un caché local, a fin de que los usuarios puedan acceder a su historial de resúmenes. Adicionalmente, cuando el usuario solicita un resumen, se comprueba primero si dicho resumen ya existe en la base de datos local, y en caso afirmativo, se recupera directamente de allí, lo que mejora los tiempos de respuesta; en caso contrario, se aguarda a la respuesta de la API REST.

Esta capa es el punto de conexión entre tres ámbitos diferentes: la capa de dominio, la cual explicaremos a continuación, la caché local, y la API REST. La forma en que cada uno de estos ámbitos represente los datos (en nuestro caso, los resúmenes) es independiente del resto. Por ejemplo, los datos procedentes de la API REST podrían contener más campos de los que realmente necesitamos o queremos almacenar localmente. Asimismo, por poner otro ejemplo, podría ser que la base de datos local no soportara algunos de los tipos de datos empleados en la capa de dominio, como por ejemplo las enumeraciones.

5.2. MOTIVACIÓN TRAS LAS ARQUITECTURAS DESARROLLADAS

Por ello, vamos a contar con tres representaciones diferentes, cada una correspondiéndose con uno de los ámbitos descritos:

- *Domain Model*: es la representación de los resúmenes propia de la capa de dominio.
- DTO (*Data Transfer Object*): se corresponde con la representación del *backend*, esto es, la API REST.
- *Entity*: es la representación de la base de datos local.

Para que estas tres representaciones diferentes se acoplen correctamente, se hace uso de los *mappers*, los cuales se encargan de transformar la representación de la capa de dominio a DTOs o *Entities*, y viceversa.

Capa de dominio

Esta capa define la lógica de dominio de la aplicación, y es independiente de la plataforma de desarrollo, es decir, en nuestro caso estará escrita puramente en Dart, sin contener ningún elemento de Flutter [65]. El motivo reside en que el dominio, como decíamos, solo debe ocuparse de la lógica de negocio, y no de los detalles de implementación. Esto también permite una fácil migración entre plataformas, en caso de ser necesario en algún momento.

Capas de aplicación y presentación

En estas capas entra en juego el patrón BLoC (*Business Logic Component*). Para entender este patrón, debemos primero explicar los conceptos de *evento* y *estado*.

Dicho de manera sencilla, un estado es aquello que se muestra en la pantalla en un momento específico. Y un evento no es más que una acción detectada por la aplicación, por ejemplo, un *click* del usuario.

Los actores centrales de este patrón son los casos de uso (llamados también BLoCs, como el propio patrón), los cuales contienen las reglas de negocio específicas de la aplicación.

Una vez introducidos los conceptos, podemos identificar cuatro pasos fundamentales:

1. Un componente (*widget*) envía un *evento* al componente BLoC.

2. El BLoC acepta el evento y lleva a cabo la tarea que tiene asociada, probablemente interaccionando con la capa de dominio.
3. El BLoC actualiza su *estado*.
4. Los componentes detectan el cambio de estado y reaccionan de algún modo.

Anteriormente, hemos mencionado el término *widget*. En Flutter, los *widgets* son los elementos que conforman la interfaz de usuario [66], como un botón o un *layout*. Los *widgets* se organizan de forma jerárquica, de modo que toda aplicación tendrá un *widget* raíz, del cual «colgarán» el resto de *widgets*, como podemos ver en la siguiente figura.

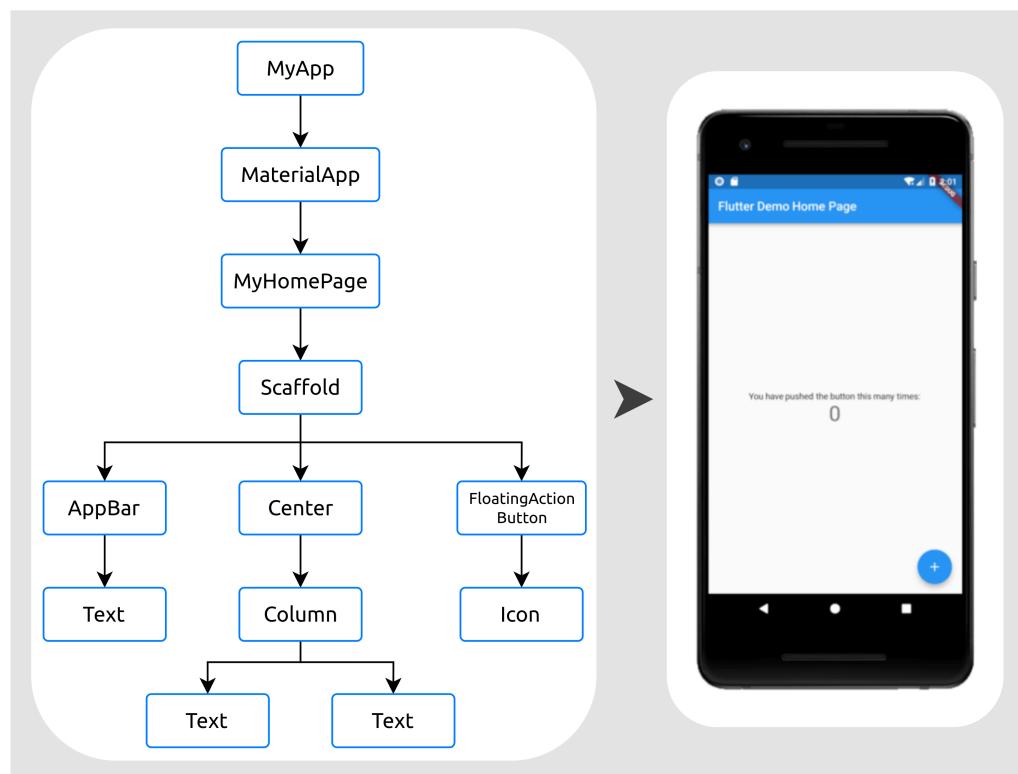


Figura 5.23: Ejemplo de jerarquía de *widgets* de una aplicación sencilla. Imagen del dispositivo móvil extraída de [59].

5.2. MOTIVACIÓN TRAS LAS ARQUITECTURAS DESARROLLADAS

Distribución de la aplicación

GitHub pages

TODO

Play Store

TODO

Trabajos relacionados

A continuación, enumeraremos algunos proyectos relacionados con nuestro trabajo.

Bert Extractive Summarizer

Este proyecto *open-source* implementa un generador de resúmenes extractivos haciendo uso del modelo BERT [67] de Google para la codificación de palabras, y aplicando *clustering* por *k-means* para determinar las frases que se incluirán en el resumen. Este proceso se detalla en [68].

El generador de resúmenes puede ser *dockerizado*, pudiéndose ejecutar como servicio, proporcionando una REST API para solicitar los resúmenes. El autor ofrece *endpoints* gratuitos con limitaciones a la hora de realizar peticiones, y *endpoints* privados de pago para aquellos particulares o empresas que requieran de mayores prestaciones.

Se puede acceder al proyecto a través del siguiente enlace:
<https://github.com/dmmiller612/bert-extractive-summarizer>.

ExplainToMe

ExplainToMe es un proyecto también *open-source* centrado en la generación de resúmenes extractivos de páginas *web*, permitiendo cómodamente pegar y copiar el *link* de la *web* que se quiere resumir.

Emplea el algoritmo de TextRank [69], el cual a su vez está inspirado en el conocido PageRank [70], el algoritmo basado en grafos que empleaba originalmente Google en su motor de búsqueda. En su caso, TextRank aplica

los principios del algoritmo de Google a la extracción de las frases más importantes de un texto.

Como en el caso anterior, también implementa una API REST.

El proyecto no ha sido actualizado desde finales de 2018. Se puede visitar a través de: <https://github.com/jjangsangy/ExplainToMe/tree/master>.

SMMRY

Se trata de una de las primeras opciones que aparecen en los motores de búsqueda a la hora de buscar «*summarizers*». También genera resúmenes extractivos, aunque a diferencia de los anteriores, no es un proyecto *open-source*.

Destacan su velocidad (*cachea* los textos resumidos recientemente), y sus múltiples opciones de resumen, como por ejemplo: ignorar preguntas, exclamaciones o frases entrecomilladas en el texto original, o la generación de mapas de calor en función de la importancia de las frases incluidas en el resumen.

Sin embargo, los resúmenes están compuestos de frases literales ordenadas cronológicamente en función de su importancia, por lo que la cohesión entre las mismas puede ser frágil e incluso, con frecuencia, se habla de personas o entidades que no han sido introducidas previamente en el resumen, pudiendo dificultar la comprensión del mismo.

En su página *web* no se explica el algoritmo concreto que se emplea, pero prestando atención a la descripción del proceso proporcionada [71], parecen emplear igualmente PageRank.

Se puede acceder a SMMRY en: <https://smmry.com/>.

Tabla comparativa

| Características | JIZT | Bert Extractive Summarizer | ExplainToMe | SMMRY |
|-----------------------------------|------------------------------|----------------------------|-------------|------------|
| Tipo de resumen ¹ | Abstractivo | Extractivo | Extractivo | Extractivo |
| Tiempo resumen corto ² | ~20 seg. | ~6 seg. | ~9 seg. | ~3 seg. |
| Tiempo resumen largo ³ | ~4 min. | No disponible ⁴ | Error | ~5 seg. |
| Ajustes básicos | ✓ | ✓ | ✓ | ✓ |
| Ajustes avanzados | ✓ | ✗ | ✗ | ✓ |
| Entrada: texto plano | ✓ | ✓ | ✗ | ✓ |
| Entrada: URL | Próximamente | ✗ | ✓ | ✓ |
| Entrada: fichero | Próximamente | ✗ | ✗ | ✓ |
| Entrada: imagen | Próximamente | ✗ | ✗ | ✗ |
| Soporte multi-modelo ⁵ | Próximamente | ✗ | ✗ | ✗ |
| Soporte multi-tarea ⁶ | Próximamente | ✗ | ✗ | ✗ |
| API REST | ✓ | ✓ | ✓ | ✓ |
| Arquitectura | Microservicios | Monolítica | Monolítica | ? |
| Plataforma | Multiplataforma ⁷ | Web | Web | Web |
| <i>Open-source</i> | ✓ | ✓ | ✓ | ✗ |
| <i>Gratuito</i> | ✓ | Limitado | ✓ | Limitado |
| Proyecto activo | ✓ | ✓ | ✗ | ✓ |

1. En los resúmenes *abstractivos*, se toman las frases literales del texto original. En los *extractivos*, se añaden palabras o expresiones nuevas.

2. Texto de entrada con ~6.500 caracteres.

3. Texto de entrada con ~90.000 caracteres.

4. La versión gratuita está limitada. No hemos tenido acceso a la versión completa.

5. Capacidad de generar resúmenes utilizando diferentes modelos.

6. Capacidad de realizar otras tareas de NLP diferentes a la generación de resúmenes.

7. Soporte nativo para Android, iOS y web. Pronto, soporte para Linux, macOS y Windows.

Tabla 6.1: Comparativa de las características ofrecidas por las diferentes alternativas para la generación de resúmenes.

Conclusiones y Líneas de trabajo futuras

Por último, y no por ello menos importante, se recogen a continuación las principales conclusiones extraídas de la realización de este proyecto. Además, se indican los posibles pasos a tomar en el futuro próximo.

7.1. Principales conclusiones

Como mencionábamos en la [Introducción](#), desde un principio supimos que JIZT era un proyecto ambicioso que requeriría gran inversión de tiempo y esfuerzo.

Cinco meses después, podemos decir, no sin cierto alivio, que hemos sido capaces de cumplir los objetivos que nos marcamos para la compleción de el presente Trabajo de Fin de Grado; JIZT es, ha día de hoy, una realidad.

Personalmente, nunca imaginamos que en torno a un 70 % del tiempo y esfuerzo se acabaría destinando al *backend*. Esta era, a su vez, el área que menos había trabajado con anterioridad, por lo que fue un reto aún mayor. Cabe preguntarnos, ¿ha valido la pena todo el esfuerzo? Y la respuesta es un rotundo sí. Contar con una buena infraestructura en el *backend* será la clave para el futuro de JIZT por los siguientes motivos:

- Facilita el escalado y asegura una alta disponibilidad de los componentes implementados actualmente.
- Permite la ampliación de las tareas de NLP proporcionadas por JIZT.
- Incentiva y facilita la colaboración de otros desarrolladores, dado que se siguen estándares de la industria.
- Todo ello se revierte en una mayor satisfacción de los usuarios.

La lección extraída de todo lo mencionado anteriormente es que la Ingeniería del *Software*, así como el Diseño de Arquitectura de *Software* son labores que pueden resultar muy complejas, pero a su vez gratificantes, especialmente en el momento en que finalmente todos los *test* se ejecutan con éxito tras horas de trabajo, e interminables «quebraderos de cabeza», si se nos permite la expresión.

No obstante, hablando de dificultades, el Procesamiento de Lenguaje Natural es también un muy buen candidato; con la realización de este proyecto nos hemos percatado de la enorme flexibilidad y ambigüedad del lenguaje natural, lo cual imposibilita establecer reglas prefijadas que sean válidas para todos los casos, como se intentó desde el inicio del NLP hasta ya entrado el siglo XXI. Cuando crees que has dado con una regla que se ajusta a todos los supuestos considerados, aparece un nuevo caso que lo desmonta todo. Por suerte, en los últimos cinco años se han producido grandes avances en el campo; no podemos esperar a poder analizar y probar los nuevos descubrimientos que el futuro nos traiga.

Podríamos mencionar muchas otras conclusiones, pero todas ellas se pueden resumir del siguiente modo: hemos aprendido *mucho*. Nuestro proyecto ha tratado con diseño de microservicios e infraestructura en la nube (Kubernetes, Docker, Kafka, API REST), bases de datos (PostgreSQL como servicio), Inteligencia Artificial (Procesamiento del Lenguaje Natural), desarrollo de aplicaciones multiplataforma (Flutter), validación y pruebas, despliegue e integración continua...

Este proyecto ha sido una oportunidad de aprendizaje y formación que creemos será muy positiva de cara a nuestra futura vida estudiantil y laboral.

7.2. Líneas futuras de trabajo

JIZT, dada su extensión, cuenta con innumerables aspectos a desarrollar en numerosos aspectos. A continuación listamos algunos de los más importantes y/o inmediatos:

- Incluir modelos en otros idiomas idiomáticos, como español, francés, alemán, chino, etc.
- Ampliar el rango de tareas de NLP que JIZT es capaz de llevar a cabo.
- Entrenar/reemplazar el modelo de *truecasing* (recomposición de mayúsculas), ya que el usado actualmente está entrenado con un corpus

pequeño, el cual generalmente consigue buenos resultados, pero en algunos casos es mejorable.

- Seguir mejorando la API REST y el *backend*. En este aspecto, las mejoras más destacables son:
 - Incluir la capacidad de extraer textos de ficheros, imágenes o URLs.
 - Incluir un «modo privado», dando al usuario la opción de que su texto no sea almacenado en la base de datos.
 - Actualmente, para detectar si un resumen ya ha sido generado previamente, se extrae un *hash* (SHA-256) a partir del texto original, el modelo, y los parámetros del resumen solicitados. Una mejora pasa por atender al texto pre-procesado, en vez del original, dado que ahora mismo si cambia un solo carácter del texto original, por ejemplo, un espacio, el texto se considera como diferente, y se genera un nuevo resumen. Esta mejora conlleva cierta dificultad dado que alteraría en cierto modo el orden secuencial del proceso de resumen, esto es: el *Dispatcher* enviaría el texto original al pre-procesador, el cual, una vez pre-procesado el texto, se lo devolvería al *Dispatcher*. En el caso de que el texto pre-procesado no existiera, el *Dispatcher* reenviaría el texto directamente al Codificador, dado que ya estaría pre-procesado.
 - Incluir la monitorización y la recogida de métricas del sistema. Actualmente, se implementa un *logging* básico, suficiente para la detección de errores, pero poco apropiado para llevar a cabo estudios de uso de recursos, carga de trabajo, etc.
 - Ofrecer al usuario mensajes de error más granulares. Por ejemplo, si el usuario ha definido parámetros de resumen inexistentes, la API le indicaría exactamente qué parámetros han sido, y por qué valores por defecto se han reemplazado.
- En cuanto a la aplicación desarrollada:
 - Iterar junto a la API REST para incluir las mejoras de esta en cuestiones como el «modo privado», la generación de resúmenes a partir de ficheros, imágenes o URLs, mensajes de error más descriptivos, etc.
 - Mejorar la internacionalización de la aplicación, traduciéndola a otros idiomas.

Bibliografía

- [1] “Daniel Crevier. AI: The tumultuous history of the search for artificial intelligence. NY: Basic Books, 1993. 432 pp. (Reviewed by Charles Fair)”. En: *Journal of the History of the Behavioral Sciences* 31.3 (1995), págs. 273-278. DOI: [https://doi.org/10.1002/1520-6696\(199507\)31:3<273::AID-JHBS2300310314>3.0.CO;2-1](https://doi.org/10.1002/1520-6696(199507)31:3<273::AID-JHBS2300310314>3.0.CO;2-1). eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/1520-6696%28199507%2931%3A3%3C273%3A%3AAID-JHBS2300310314%3E3.0.CO%3B2-1>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/1520-6696%28199507%2931%3A3%3C273%3A%3AAID-JHBS2300310314%3E3.0.CO%3B2-1>.
- [2] A. M. Turing. “Computing Machinery and Intelligence”. En: *Mind* LIX.236 (oct. de 1950), págs. 433-460. ISSN: 0026-4423. DOI: [10.1093/mind/LIX.236.433](https://doi.org/10.1093/mind/LIX.236.433). eprint: <https://academic.oup.com/mind/article-pdf/LIX/236/433/30123314/lix-236-433.pdf>. URL: <https://doi.org/10.1093/mind/LIX.236.433>.
- [3] Pamela McCorduck. *Machines Who Think*. USA: W. H. Freeman y Co., 1979. ISBN: 0716710722.
- [4] Joachim Rahmfeld. *Recent Advances in Natural Language Processing*. Sep. de 2019. URL: <https://venturebeat.com/2021/01/06/ai-models-from-microsoft-and-google-already-surpass-human-performance-on-the-superglue-language-benchmark/>. Último acceso: 26/01/2021.
- [5] Thang Luong, Hieu Pham y Christopher D. Manning. “Effective Approaches to Attention-based Neural Machine Translation”. En: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational

- Linguistics, sep. de 2015, págs. 1412-1421. DOI: [10.18653/v1/D15-1166](https://doi.org/10.18653/v1/D15-1166). URL: <https://www.aclweb.org/anthology/D15-1166>.
- [6] Dzmitry Bahdanau, Kyunghyun Cho y Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: [1409.0473 \[cs.CL\]](https://arxiv.org/abs/1409.0473).
 - [7] Ashish Vaswani y col. “Attention Is All You Need”. En: *CoRR* abs/1706.03762 (2017). arXiv: [1706.03762](https://arxiv.org/abs/1706.03762). URL: <http://arxiv.org/abs/1706.03762>.
 - [8] Thomas Macaulay. *Someone let a GPT-3 bot loose on Reddit — it didn't end well*. Oct. de 2020. URL: <https://thenextweb.com/neural/2020/10/07/someone-let-a-gpt-3-bot-loose-on-reddit-it-didnt-end-well>. Último acceso: 26/01/2021.
 - [9] Kyle Wiggers. *AI models from Microsoft and Google already surpass human performance on the SuperGLUE language benchmark*. Ene. de 2021. URL: <https://venturebeat.com/2021/01/06/ai-models-from-microsoft-and-google-already-surpass-human-performance-on-the-superglue-language-benchmark/>. Último acceso: 26/01/2021.
 - [10] Google. *Cloud Natural Language*. URL: <https://cloud.google.com/natural-language>. Último acceso: 26/01/2021.
 - [11] IBM. *Watson*. URL: <https://www.ibm.com/watson/about>. Último acceso: 26/01/2021.
 - [12] Amazon. *Comprehend*. URL: <https://aws.amazon.com/es/comprehend>. Último acceso: 26/01/2021.
 - [13] Microsoft. *Text Analytics*. URL: <https://azure.microsoft.com/es-es/services/cognitive-services/text-analytics>. Último acceso: 26/01/2021.
 - [14] Raconteur. *A Day in Data*. 2019. URL: <https://www.raconteur.net/infographics/a-day-in-data/>. Último acceso: 26/01/2021.
 - [15] Abigail See, Peter J. Liu y Christopher D. Manning. “Get To The Point: Summarization with Pointer-Generator Networks”. En: *CoRR* abs/1704.04368 (2017), pág. 1. arXiv: [1704.04368](https://arxiv.org/abs/1704.04368). URL: <http://arxiv.org/abs/1704.04368>.
 - [16] Colin Raffel y col. “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”. En: *CoRR* abs/1910.10683 (2019), pág. 11. arXiv: [1910.10683](https://arxiv.org/abs/1910.10683). URL: <http://arxiv.org/abs/1910.10683>.

- [17] Microsoft. *Defining Cloud Native*. Mayo de 2020. URL: <https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition>. Último acceso: 27/01/2021.
- [18] John Arundel y Justin Domingus. *Cloud Native DevOps with Kubernetes*. O'Reilly Media, Inc., mar. de 2019. ISBN: 9781492040767.
- [19] Adam Bellemare. *Building Event-Driven Microservices: Leveraging Organizational Data at Scale*. O'Reilly Media, Inc., 2020. ISBN: 9781492057895.
- [20] Crunchy Data. *Crunchy PostgreSQL Operator*. Mayo de 2021. URL: <https://access.crunchydata.com/documentation/postgres-operator/4.5.1/>. Último acceso: 04/02/2021.
- [21] Robert Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson Education, 2015. ISBN: 9780134494166.
- [22] Daniel Sauble. *Offline First Web Development*. Packt, 2015. ISBN: 9781785884573.
- [23] Mike Lewis y col. “BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension”. En: *CoRR* abs/1910.13461 (2019). arXiv: [1910.13461](https://arxiv.org/abs/1910.13461). URL: <http://arxiv.org/abs/1910.13461>.
- [24] Wikipedia. *Reconocimiento de entidades nombradas - Wikipedia, La enciclopedia libre*. 2020. URL: https://es.wikipedia.org/wiki/Reconocimiento_de_entidades_nombradas. Último acceso: 27/01/2021.
- [25] Christopher Manning - Stanford University. *Stanford CS224N: NLP with Deep Learning. Winter 2019. Lecture 13. Contextual Word Embeddings*. 2019. URL: <https://www.youtube.com/watch?v=S-CspeZ8FHc>. Último acceso: 28/01/2021.
- [26] Linlin Hou y col. *Method and Dataset Entity Mining in Scientific Literature: A CNN + Bi-LSTM Model with Self-attention*. 2020. arXiv: [2010.13583 \[cs.AI\]](https://arxiv.org/abs/2010.13583).
- [27] Tomas Mikolov y col. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: [1301.3781 \[cs.CL\]](https://arxiv.org/abs/1301.3781).
- [28] Tomás Mikolov y col. “Distributed Representations of Words and Phrases and their Compositionality”. En: *CoRR* abs/1310.4546 (2013). arXiv: [1310.4546](https://arxiv.org/abs/1310.4546). URL: <http://arxiv.org/abs/1310.4546>.

- [29] Jeffrey Pennington, Richard Socher y Christopher Manning. “GloVe: Global Vectors for Word Representation”. En: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, abr. de 2014, págs. 1532-1543. doi: [10.3115/v1/D14-1162](https://doi.org/10.3115/v1/D14-1162). URL: <https://www.aclweb.org/anthology/D14-1162>.
- [30] Matthew E. Peters y col. “Deep contextualized word representations”. En: *CoRR* abs/1802.05365 (2018). arXiv: [1802.05365](https://arxiv.org/abs/1802.05365). URL: [http://arxiv.org/abs/1802.05365](https://arxiv.org/abs/1802.05365).
- [31] Jacob Devlin y col. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. En: *CoRR* abs/1810.04805 (2018). arXiv: [1810.04805](https://arxiv.org/abs/1810.04805). URL: [http://arxiv.org/abs/1810.04805](https://arxiv.org/abs/1810.04805).
- [32] Bożena Cetnarowska. “Ingo Plag, Word-formation in English (Cambridge Textbooks in Linguistics). Cambridge: Cambridge University Press, 2003. Pp. xiv 240.” En: *Journal of Linguistics* 41.1 (2005). doi: [10.1017/S002226704303233](https://doi.org/10.1017/S002226704303233).
- [33] Dipanjan Sarkar, Raghav Bali y Tamoghna Ghosh. *Hands-On Transfer Learning with Python*. Packt Publishing, 2018. ISBN: 9781788831307.
- [34] Manoj Kumar y col. *ProtoDA: Efficient Transfer Learning for Few-Shot Intent Classification*. 2021. arXiv: [2101.11753 \[cs.CL\]](https://arxiv.org/abs/2101.11753).
- [35] Nuredin Ali. *Exploring Transfer Learning on Face Recognition of Dark Skinned, Low Quality and Low Resource Face Data*. 2021. arXiv: [2101.10809 \[cs.CV\]](https://arxiv.org/abs/2101.10809).
- [36] Yi Liu y Shuiwang Ji. *A Multi-Stage Attentive Transfer Learning Framework for Improving COVID-19 Diagnosis*. 2021. arXiv: [2101.05410 \[eess.IV\]](https://arxiv.org/abs/2101.05410).
- [37] Patrick von Platen. *How to generate text: using different decoding methods for language generation with Transformers*. Mar. de 2020. URL: <https://huggingface.co/blog/how-to-generate>. Último acceso: 31/01/2021.
- [38] Ashwin K. Vijayakumar y col. “Diverse Beam Search: Decoding Diverse Solutions from Neural Sequence Models”. En: *CoRR* abs/1610.02424 (2016). arXiv: [1610.02424](https://arxiv.org/abs/1610.02424). URL: [http://arxiv.org/abs/1610.02424](https://arxiv.org/abs/1610.02424).
- [39] Louis Shao y col. “Generating Long and Diverse Responses with Neural Conversation Models”. En: *CoRR* abs/1701.03185 (2017). arXiv: [1701.03185](https://arxiv.org/abs/1701.03185). URL: [http://arxiv.org/abs/1701.03185](https://arxiv.org/abs/1701.03185).

- [40] Kenton Murray y David Chiang. “Correcting Length Bias in Neural Machine Translation”. En: *CoRR* abs/1808.10006 (2018). arXiv: [1808.10006](#). URL: <http://arxiv.org/abs/1808.10006>.
- [41] Yilin Yang, Liang Huang y Mingbo Ma. “Breaking the Beam Search Curse: A Study of (Re-)Scoring Methods and Stopping Criteria for Neural Machine Translation”. En: *CoRR* abs/1808.09582 (2018). arXiv: [1808.09582](#). URL: <http://arxiv.org/abs/1808.09582>.
- [42] Ari Holtzman y col. *The Curious Case of Neural Text Degeneration*. 2020. arXiv: [1904.09751 \[cs.CL\]](#).
- [43] Angela Fan, Mike Lewis y Yann Dauphin. *Hierarchical Neural Story Generation*. 2018. arXiv: [1805.04833 \[cs.CL\]](#).
- [44] Lucian Vlad Lita y col. “TRuEcasIng”. En: *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*. ACL '03. Sapporo, Japan: Association for Computational Linguistics, 2003, págs. 152-159. DOI: [10.3115/1075096.1075116](#). URL: <https://doi.org/10.3115/1075096.1075116>.
- [45] Hugging Face. *Model t5-large*. Feb. de 2021. URL: <https://huggingface.co/t5-large>. Último acceso: 03/02/2021.
- [46] Hugging Face. *Pretrained models*. Feb. de 2021. URL: https://huggingface.co/transformers/pretrained_models.html. Último acceso: 03/02/2021.
- [47] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., feb. de 2015. ISBN: 9781491950357.
- [48] Docker. *Why Docker?* 2021. URL: <https://www.docker.com/why-docker>. Último acceso: 29/01/2021.
- [49] Kubernetes. *What is Kubernetes?* Oct. de 2020. URL: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes>. Último acceso: 04/02/2021.
- [50] Kubernetes. *Scheduling and Eviction*. Jun. de 2020. URL: <https://kubernetes.io/docs/concepts/scheduling-eviction>. Último acceso: 04/02/2021.
- [51] Nginx. *What is an API Gateway?* Sep. de 2020. URL: <https://www.nginx.com/learn/api-gateway>. Último acceso: 04/02/2021.

- [52] Microsoft Docs. *Communication in a microservice architecture*. Ene. de 2020. URL: <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture>. Último acceso: 04/02/2021.
- [53] Wikipedia. *Apache Kafka*. Ene. de 2021. URL: https://en.wikipedia.org/wiki/Apache_Kafka. Último acceso: 04/02/2021.
- [54] Apache Software Foundation. *Apache Kafka*. Nov. de 2020. URL: <https://kafka.apache.org>. Último acceso: 04/02/2021.
- [55] Helm - The package manager for Kubernetes. *Library Charts*. Ene. de 2021. URL: https://helm.sh/docs/topics/library_charts. Último acceso: 04/02/2021.
- [56] The Pallets Projects. *Flask*. 2021. URL: <https://palletsprojects.com/p/flask>. Último acceso: 29/01/2021.
- [57] Flask-RESTful Community. *Flask-RESTful*. 2021. URL: <https://flask-restful.readthedocs.io/en/latest>. Último acceso: 29/01/2021.
- [58] Flutter. *Flutter - Hermosas apps nativas en tiempo record*. Sep. de 2020. URL: <https://esflutter.dev>. Último acceso: 05/02/2021.
- [59] Alberto Miola. *Flutter Complete Reference: Create beautiful, fast and native apps for any device*. Sep. de 2020. ISBN: 9798691939952.
- [60] Flutter. *Hot reload*. Mayo de 2020. URL: <https://flutter.dev/docs/development/tools/hot-reload>. Último acceso: 09/02/2021.
- [61] Wikipedia. *Compilación anticipada*. Dic. de 2020. URL: https://es.wikipedia.org/wiki/Compilaci%C3%B3n_anticipada. Último acceso: 05/02/2021.
- [62] Flutter. *Web FAQ*. Oct. de 2020. URL: <https://flutter.dev/docs/development/platform-integration/web>. Último acceso: 05/02/2021.
- [63] Kanboard. *Kanboard - Kanban Project Management Software*. Feb. de 2021. URL: <https://github.com/kanboard/kanboard>. Último acceso: 06/02/2021.
- [64] Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013. ISBN: 0321834577.
- [65] Shady Boukhary y Rafael Monteiro. *Flutter Clean Architecture Package*. Ene. de 2021. URL: https://pub.dev/packages/flutter_clean_architecture. Último acceso: 07/02/2021.

- [66] Flutter API. *Widget class*. Sep. de 2020. URL: <https://api.flutter.dev/flutter/widgets/Widget-class.html>. Último acceso: 07/02/2021.
- [67] Jacob Devlin y col. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: [1810.04805 \[cs.CL\]](https://arxiv.org/abs/1810.04805).
- [68] Derek Miller. “Leveraging BERT for Extractive Text Summarization on Lectures”. En: *CoRR* abs/1906.04165 (2019). arXiv: [1906.04165](https://arxiv.org/abs/1906.04165). URL: <http://arxiv.org/abs/1906.04165>.
- [69] Rada Mihalcea y Paul Tarau. “TextRank: Bringing Order into Text.” En: jul. de 2004.
- [70] Lawrence Page y col. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Previous number = SIDL-WP-1999-0120. Stanford InfoLab, nov. de 1999. URL: <http://ilpubs.stanford.edu:8090/422/>.
- [71] Smmry Team. *Smmry*. 2021. URL: <https://smmry.com/about>. Último acceso: 31/01/2021.