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 — 3 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. nombre tutor, profesor del departamento de nombre departamento, área de nombre área.

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, 3 de febrero de 2021

Vº. Bº. del Tutor:

D. Carlos López Nozal

Vº. Bº. del Tutor:

D. José Francisco Díez Pastor

Resumen

En este primer apartado se hace una **breve** presentación del tema que se aborda en el proyecto.

Descriptores

Palabras separadas por comas que identifiquen el contenido del proyecto Ej: servidor web, buscador de vuelos, android ...

Abstract

A **brief** presentation of the topic addressed in the project.

Keywords

keywords separated by commas.

Índice general

Índice de figuras

Índice de tablas

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 empresarial. 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 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 preentrenados 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, las generación de resúmenes aliviara 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 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 orquestración 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.

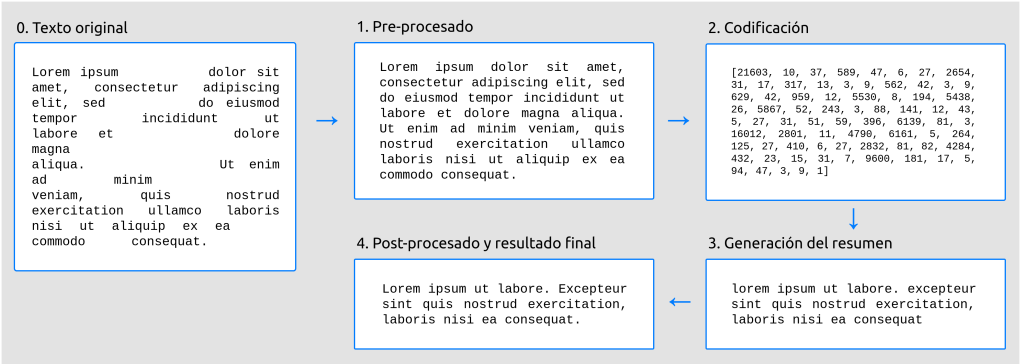

Figura 3.1: Etapas en la generación de resúmenes.

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 resúmenes[2].
- 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 original[3] 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.

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. Existe toda una disciplina dedicada la identificación de este tipo de palabras, conocida 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 númericas. 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 concretos empleados, dado que consideramos que queda fuera del alcance de este trabajo[5].

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.

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

[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].

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:

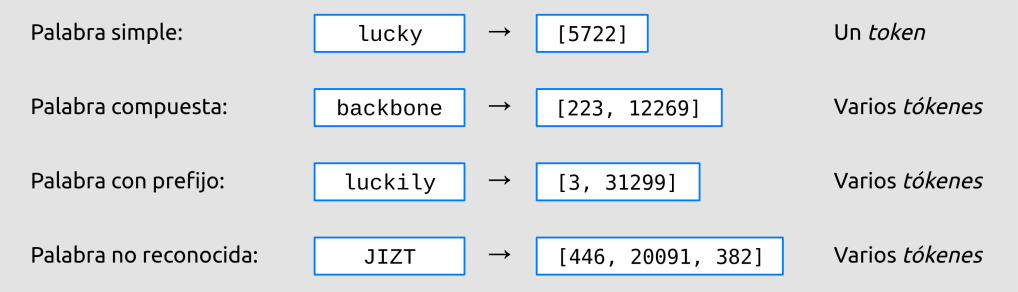

Figura 3.2: Ejemplo de *tokenización* con el modelo T5.

En el anterior ejemplo, si decodificamos los *tókenes* correspondientes a la palabra `"backcone"`, esto es, `[223, 12269]`, obtenemos los fragmentos `"back"`, y `"bone"`, respectivamente.

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 de este modelo siempre tiene el *id* 1. En la siguiente figura podemos ver un ejemplo con un texto de entrada:

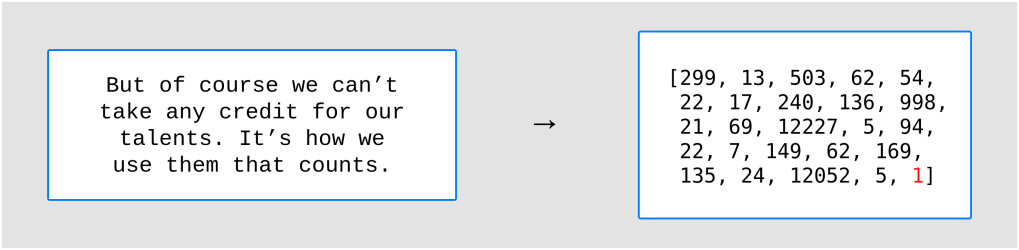

Figura 3.3: Pasaje del libro *A Wrinkle in Time*. El *token* EOS se ha marcado en rojo.

Como podemos ver, el *token* 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 quisiéramos traducir del alemán al español, le señalaríamos: «traduce de alemán a español» seguido de nuestro texto («*summarize German to Spanish*»).

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 ← dividirEnFrases(texto)
3:      frasesCodif ← [ ]                                   ▷ Frases codificadas
4:      prefijoCodif ← codifica(prefijo)
5:      EOSCodif ← codifica(EOS)                            ▷ Token EOS codificado
6:      subdivsCodif ← [ ]                                  ▷ Subdivisiones codificadas
7:      for fr in frases do
8:          frasesCodif ← codifica(fr)[: −1]                ▷ Excluir EOS
9:      end for
10:     ptosCorte ← divideVoraz(frasesCodif, prefijoCodif)
11:     ptosCorte ← balanceaSubdivs(ptosCorte)
12:     for i ← 0, len(ptosCorte) - 1 do
13:         frasesSubvid ← frasesCodif[ptosCorte[i] : ptosCorte[i + 1]]   ▷ Frases en subdiv.
14:         subdivsCodif[i] ← 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, $subdivsCofif$ tendrá la siguiente forma:

$$[[23, 34, 543, 45, ..., 1], [23, 32{,}401, 11, ..., 1], [23, 74{,}25, 204, ..., 1], ...]$$

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:

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.

Algoritmo 2 División voraz del texto.

```
 1: procedure DivideVoraz(frasesCodif, prefijoCodif)
 2:     ptosCorte ← [0]
 3:     lenSubdiv = len(prefijoCodif) + len(frasesCodif[0]) + 1        ▷ Contar prefijo y EOS
 4:     for i ← 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
```

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 está 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.

Algoritmo 3 Balanceo de las subdivisiones.

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

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.

Podemos visualizarlo gráficamente con un ejemplo muy simple:

Figura 3.4: Ejemplo gráfico del algoritmo de balanceo. 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.

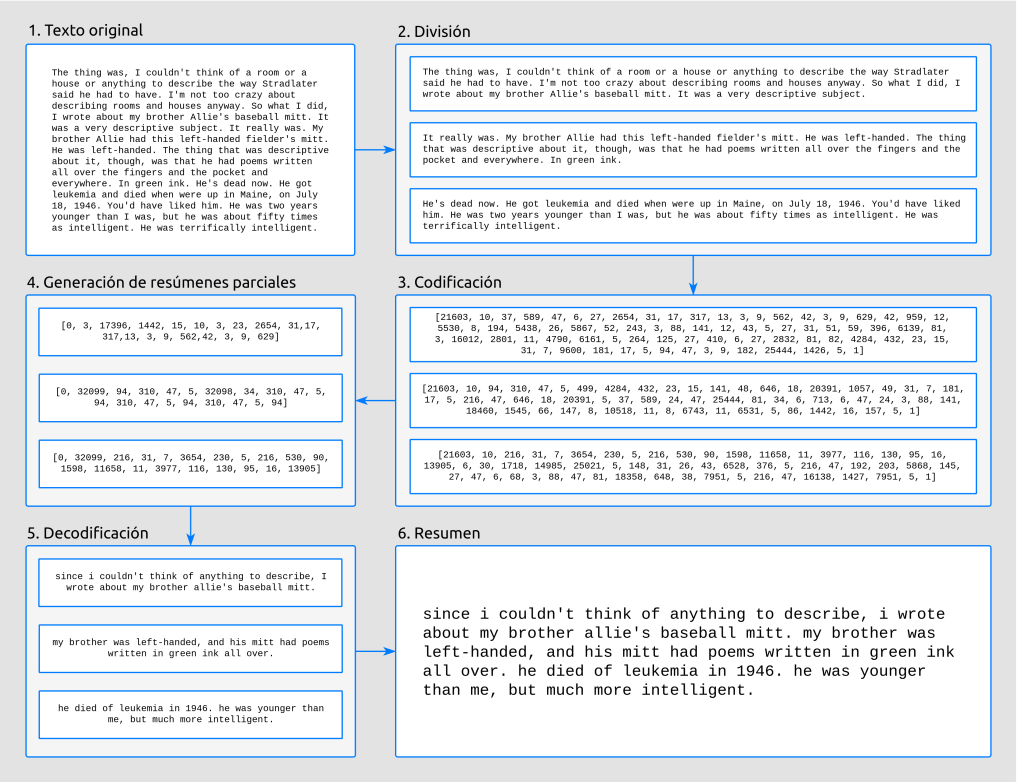

Figura 3.5: Proceso de generación de resúmenes, ilustrado con un fragmento del libro *The Catcher in the Rye*.

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 veíamos 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:

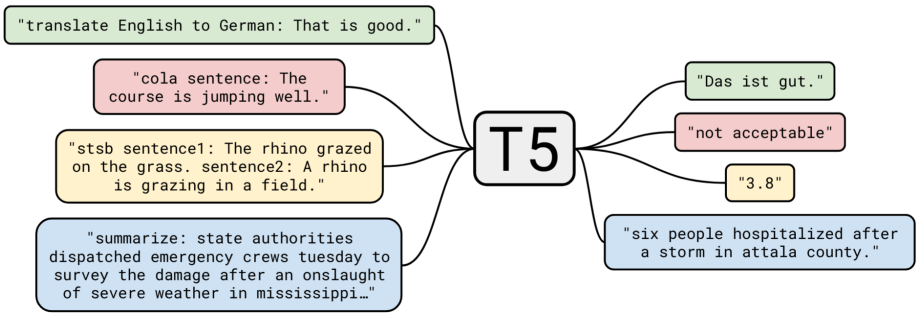

Figura 3.6: El *framework* texto a texto permite emplear el mismo modelo, con los mismos hiperparámetros, función de pérdida, etc., para aplicarlo a diversas tareas de NLP [16].

En 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:

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

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, $w_t = argmax_w P(w|w_{t-1})$ 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].
- 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).

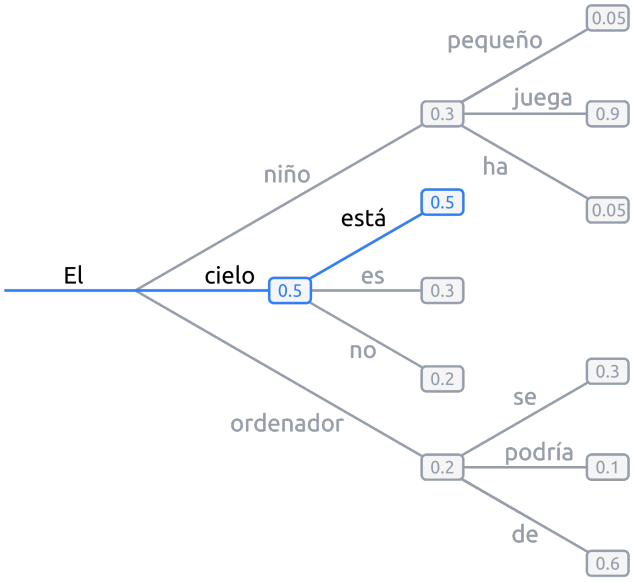

Figura 3.7: Ejemplo de búsqueda voraz: en cada paso, se toma la palabra con mayor probabilidad.

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].

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.

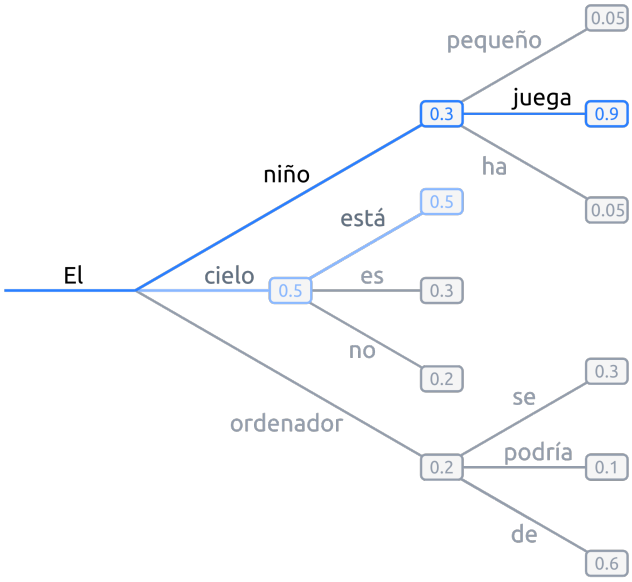

Figura 3.8: Ejemplo de *beam search* con `n_beams = 2`. Durante la búsqueda, se consideran los dos caminos con mayor probabilidad conjunta.

- 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»:

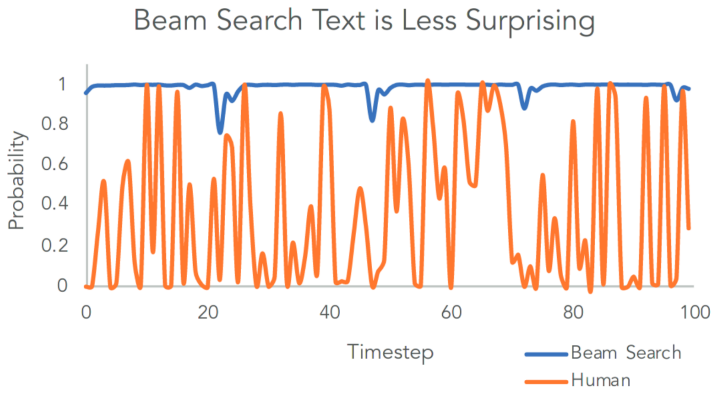

Figura 3.9: Distribución de probabilidades del lenguaje natural frente a la estrategia de *beam search* [42].

Muestreo

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

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

De manera gráfica, siguiendo con el ejemplo anterior:

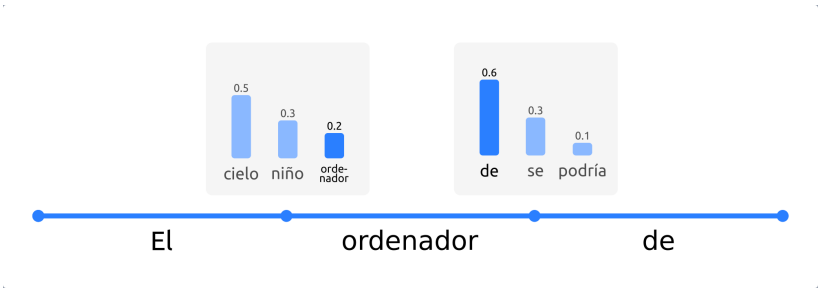

Figura 3.10: Ejemplo de muestreo. En cada paso, se elige una palabra aleatoriamente en función de su probabilidad.

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:

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

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

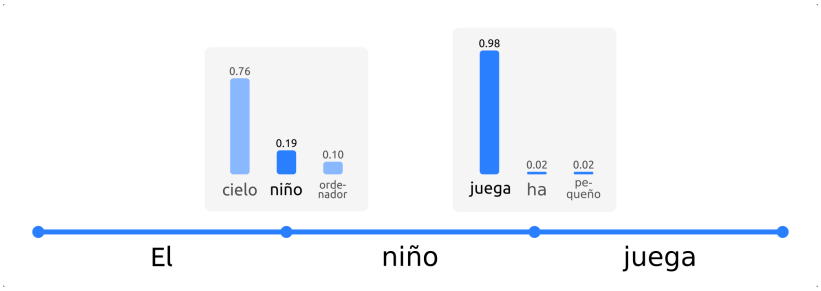

Figura 3.11: Al decrementar la temperatura, las diferencias en las probabilidades se hacen más acusadas.

Muestreo *top-k*

En ste 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).

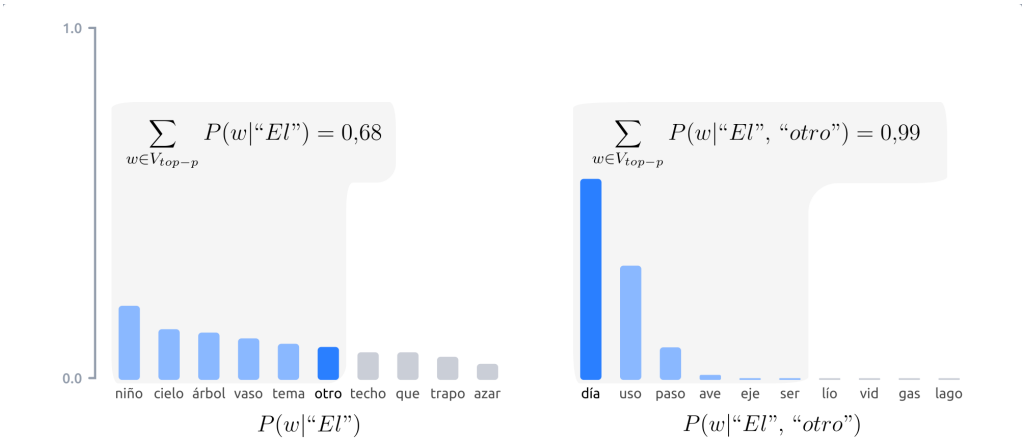

Figura 3.12: Al decrementar la temperatura, las diferencias en las probabilidades se hacen más acusadas.

Tanto la búsqueda voraz como el muestreo visto anteriormente, se pueden como casos particulares del muestreo *top-k*. Si establececeмo $k = 1$, 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 Figura 3.12, 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].

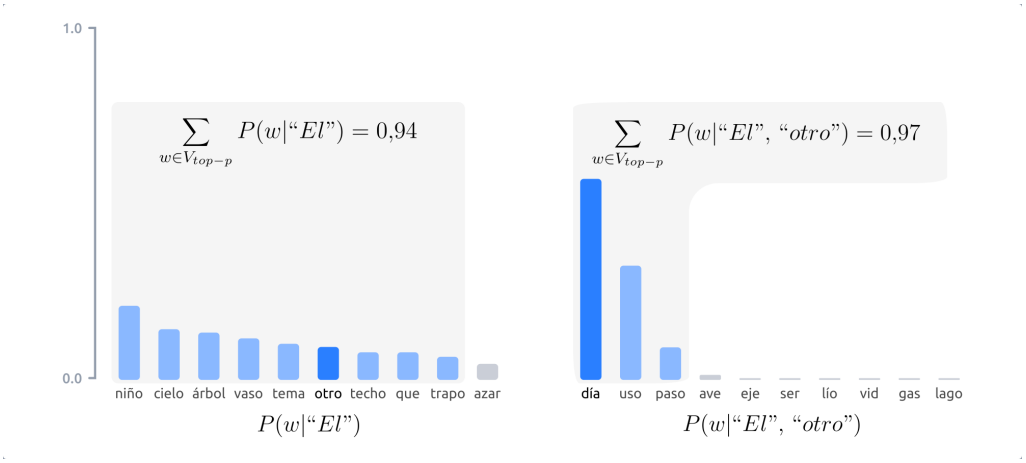

Figura 3.13: Con el muestreo *top-p*, el número de palabras entre las cuales elegir en cada paso varía en función de las probabilidades de las palabras candidatas.

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 económicas[7]. 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*

4.3. `Frontend`

4.4. Flask y Flask-RESTful

Flask es uno de los *frameworks* más populares para la creación de aplicaciones *web* en Python[47]. Está concebido para ser lo más simple posible. En nuestro caso, la hemos empleado para implementar la lógica de la API REST. Además, hemos utilizado una conocida extensión de Flask, Flask-RESTful [48], que facilita aún más dicha implementación.

4.5. Docker

Se trata de una serie de servicios como plataforma (PaaS), que proporcionan virtualización a nivel de sistema operativo, permitiendo ejecutar *software* en paquetes llamados *contenedores* [49].

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.

Docker nos va a permitir encapsular cada servicio en un contenedor, posibilitando la implementación de la arquitectura de microservicios.

4.6. Kubernetes

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

Aspectos relevantes del desarrollo del proyecto

#TODO

Mencionar Cloud Native.

Trabajos relacionados

6.1. Proyectos similares

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 [50] de Google para la codificación de palabras, y aplicando *clústering* por *k-means* para determinar las frases que se incluirán en el resumen. Este proceso se detalla en [51].

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 [52], el cual a su vez está inspirado en el conocido PageRank [53], 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 uno 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 explicita el algoritmo concreto que se emplea, pero prestando atención a la descripción del proceso proporcionada [54], parecen emplear igualmente PageRank.

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

Tabla comparativa

Caraterísticas	**JIZT**	Bert Extracive 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	×	✓	✓
Soporte multi-modelo[5]	Próximamente	×	×	×
Soporte multi-tarea[6]	Próximamente	×	×	×
API REST	✓	✓	✓	✓
Arquitectura	Microservicios	Monolítica	Monolítica	?
Plataforma	Multiplataforma[7]	Web	Web	Web
Open-source	✓	✓	✓	×
Gratuito	✓	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

Todo proyecto debe incluir las conclusiones que se derivan de su desarrollo. Éstas pueden ser de diferente índole, dependiendo de la tipología del proyecto, pero normalmente van a estar presentes un conjunto de conclusiones relacionadas con los resultados del proyecto y un conjunto de conclusiones técnicas. Además, resulta muy útil realizar un informe crítico indicando cómo se puede mejorar el proyecto, o cómo se puede continuar trabajando en la línea del proyecto realizado.

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. 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. 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. 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].

[7] Ashish Vaswani y col. "Attention Is All You Need". En: *CoRR* abs/1706.03762 (2017). arXiv: 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. 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. 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/latest. Último acceso: 27/01/2021.

[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. 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].

[27] Tomas Mikolov y col. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[28] Tomás Mikolov y col. "Distributed Representations of Words and Phrases and their Compositionality". En: *CoRR* abs/1310.4546 (2013). arXiv: 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. 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. URL: http://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. URL: http://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/S0022226704303233.

[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].

[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].

[36] Yi Liu y Shuiwang Ji. *A Multi-Stage Attentive Transfer Learning Framework for Improving COVID-19 Diagnosis*. 2021. arXiv: 2101.05410 [eess.IV].

[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. URL: http://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. URL: http://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] The Pallets Projects. *Flask*. 2021. URL: https://palletsprojects.com/p/flask. Último acceso: 29/01/2021.

[48] Flask-RESTful Community. *Flask-RESTful*. 2021. URL: https://flask-restful.readthedocs.io/en/latest. Último acceso: 29/01/2021.

[49] Docker. *Why Docker?* 2021. URL: https://www.docker.com/why-docker. Último acceso: 29/01/2021.

[50] Jacob Devlin y col. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[51] Derek Miller. "Leveraging BERT for Extractive Text Summarization on Lectures". En: *CoRR* abs/1906.04165 (2019). arXiv: 1906.04165. URL: http://arxiv.org/abs/1906.04165.

[52] Rada Mihalcea y Paul Tarau. "TextRank: Bringing Order into Text." En: jul. de 2004.

[53] 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/.

[54] Smmry Team. *Smmry.* 2021. URL: https://smmry.com/about. Último acceso: 31/01/2021.