Aplicación para el estudio de japonés

Trabajo de Fin de Máster

Máster Universitario en Ingeniería Informática

VNiVERSiDAD D SALAMANCA

Febrero 2024

Autor

Francisco García Encinas

Tutores

Juan Francisco de Paz Santana

Vivian Félix López Batista

RESUMEN

El japonés tiene muchas diferencias con respecto a los lenguajes occidentales. Entre ellas, una de las más notables es el uso de caracteres logográficos, esto es, caracteres que por sí mismos tienen un significado, los *kanjis*. Existen aproximadamente 3.000 *kanji* de uso común, pero la cifra puede aumentar hasta unos 13.000 si se consideran estándares técnicos o, incluso, 50.000 si se consideran *kanjis* infrecuentes.

Como se puede suponer, para un nativo de una lengua occidental, que no está acostumbrado al uso de esta clase de caracteres, estudiar japonés no es una tarea sencilla. De hecho, el *Foreign Service Institute* de Estados Unidos lo clasifica como uno de los lenguajes más difíciles.

Este trabajo pretende analizar las aplicaciones existentes en el campo del aprendizaje de la lengua japonesa, centrándose en el estudio de *kanjis*, y mejorar sobre ellas. Para ello, se desarrollará una aplicación web que potencie la experiencia de aprendizaje por medio de técnicas de inteligencia artificial, específicamente, *embeddings* para encontrar *kanjis* similares. Gracias a esto se ayudará a los usuarios a detectar más fácilmente las diferencias. También, se hará uso de algoritmos de repetición espaciada con el objetivo de alcanzar una frecuencia de estudio óptima.

Para implementar esta aplicación se hace uso de bastantes herramientas diferentes, desde TypeScript con NestJS para el servidor hasta Python con PyTorch para los modelos pasando por TypeScript con React para el cliente. Además, se utiliza una base de datos PostgreSQL con el plugin pgVector para almacenar los vectores de *embedding* y realizar consultas sobre ellos. También se utiliza Kuromoji y parsers implementados con Parsimmon para diversas tareas de procesamiento del lenguaje natural.

SUMMARY

Japanese is very different from occidental languages. Among them, one of the most notable differences is the usage of logographic characters, that is, characters that have a meaning on their own, the *kanjis*. There are around 3.000 frequently used *kanji*, but this count can increase to up to 13.000 if we consider technical standards, or even 50.000 if more infrequent characters are considered.

Obviously, studying Japanese is not a simple task for a native speaker of an occidental language since they are not accustomed to these kinds of characters. Indeed, the Foreign Service Institute of the United States of America classifies Japanese as one of the hardest languages.

This work analyzes already existing Japanese studying applications, especially those centered around *kanjis*, and improves over them. To do this, we propose a web application that empowers the learning process with the use of artificial intelligence algorithms, specifically, embeddings to find similar *kanjis*. Thanks to these users can learn differences between similar *kanjis* in a simple way. Also, the spaced repetition algorithm is used to reach an optimal review frequency.

Many different tools have been used to develop this application, from TypeScript with NestJS for the server to Python with Pytorch for the machine learning models passing by TypeScript with React for the from. Furthermore, a PostgreSQL database with the pgVector plugin is used to store embedding vectors and query them. Also, Kuromoji and some custom parsers developed using Parsimmon are used for natural language processing.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

Japón es una de las principales potencias económicas a nivel global. De hecho, en 2022, Japón cerró el año con uno de los productos interiores brutos (PIB) más altos, sólo superado por Estados Unidos y China [1]. Además, gracias a su estrategia de apertura hacia el exterior con planes como Cool Japan [2], también es uno de los principales exportadores de cultura mundialmente. Esta exportación cultural ha hecho que Japón se convierta en uno de los referentes en sectores como el de las novelas visuales y el de la animación. Tan importante es su influencia en estos que existen géneros que identifican exclusivamente a los productos de origen japonés, manga y anime, respectivamente. También, Japón es uno de los países con mayor repercusión en la industria de los videojuegos con empresas mundialmente conocidas como Sony (matriz de Play Station), Nintendo (con franquicias como Pokémon, Mario Bros o The Legend of Zelda), Bandai Namco (con franquicias como Pac-Man), Square Enix (con franquicias como Final Fantasy) y Sega (con franquicias como Sonic the Hedgehog) ...

Dada la importancia económico-cultural del país, el estudio de su lengua, el japonés, puede ser deseable para el público internacional. Sin embargo, alcanzar un nivel de competencia adecuado puede ser una tarea ardua para hablantes cuyas lenguas maternas sean lenguas occidentales, como el inglés o el español, debido a las diferencias que tiene el japonés con estas: sistema de escritura, reglas gramaticales, normas sociales, contexto cultural... De hecho, el Foreign Service Institute de Estados Unidos (la entidad responsable de entrenar a su cuerpo diplomático) clasifica el japonés como una de las lenguas más complejas junto al árabe, el chino (mandarín y cantonés) y el coreano [3].

Existen multitud de técnicas y herramientas que pretenden acelerar y facilitar este proceso de aprendizaje. Dado que es un campo muy amplio, el alcance de este trabajo se limita a analizar un subconjunto de estas: las aplicaciones móvil o web, por ejemplo, WaniKani, Dulingo, Anki o Kanji Study. Estas herramientas realizan un gran trabajo, pero cuentan con múltiples limitaciones que podrían ser mejoradas, por ejemplo: el estudio de partículas, la diferenciación de *kanjis* similares, estudio de *kanjis* en el contexto, facilidad de uso...

Este trabajo utiliza técnicas de inteligencia artificial y procesamiento del lenguaje natural para mejorar las alternativas existentes para el caso específico del estudio de *kanjis*. En

concreto, se pretende facilitar la tarea de distinción de *kanjis* similares (visual o semánticamente) y permitir su estudio en el contexto de las palabras en las que aparecen. Para ello, se analizarán y utilizarán diversos mecanismos de *embedding* (autocodificadores y Word2Vec aplicado a *kanjis*) y bases de datos vectoriales que permitan recuperar *kanjis* en función de su representación en forma de *embeddings*. También, se pretende conseguir que la herramienta desarrollada sea más fácil de usar que competidores como Anki.

En la siguiente sección 1.1, se detallará qué hitos se marcan para el desarrollo de esta tarea. Posteriormente, en la sección 1.2 se describirá cómo se organiza esta memoria y, por último, en la sección 1.3 se explican algunas convecciones utilizadas en este trabajo.

1.1. Objetivos

Este trabajo pretende desarrollar una alternativa a las aplicaciones existentes para el estudio de japonés. En concreto, esta herramienta le dará un mayor énfasis a la diferenciación entre *kanjis* similares para mejorar la eficiencia del proceso de estudio del usuario. Para ello, se plantean los siguientes objetivos:

- Analizar algunas de las aplicaciones existentes para el estudio del japonés y detectar sus puntos fuertes y sus puntos débiles. Específicamente, se analizarán herramientas que se puedan utilizar para el estudio de *kanji*.
- Analizar técnicas de inteligencia artificial y del procesamiento del lenguaje natural que permitan mejorar esta experiencia de estudio. En concreto, este trabajo se centrará en encontrar relaciones entre *kanji* similares para ayudar al usuario a discernir entre ellos.
- Diseñar una solución software alternativa a las analizadas que mejore la experiencia de aprendizaje del usuario. Se conseguirá incorporando información sobre la similitud de los *kanjis* a la hora de generar el material de estudio.
- Implementar un prototipo funcional de la solución diseñada.

1.2. Estructura del documento

Esta memoria se divide en seis secciones principales seguidas de una sección de bibliografía (sección 7) donde se citan trabajos relevantes a la temática tratada y otra de

anexos (sección 8) donde se hablará en más profundidad de algunos temas que se consideren importantes.

Dentro de las secciones principales, la primera de ellas es la introducción, que es la sección madre de este subapartado. A esta le sigue un apartado sobre el marco teórico (sección 2) que describe las bases necesarias para comprender el dominio del problema. Después, se presenta una sección de estado del arte (sección 3) que presenta unos fundamentos teóricos que permiten entender el dominio de la solución. Tener una sección de marco teórico e, independientemente, una sección de estado del arte puede ser algo ortodoxo, pero en este trabajo se ha considerado que era una necesidad debido a la alta complejidad del problema tratado. En el apartado 4 y el apartado 5 se expondrán los medios y el proceso de desarrollo software, respectivamente, que se han seguido para alcanzar los objetivos planteados. Por último, en el apartado 6 se explicarán los resultados obtenidos, las lecciones aprendidas y se delineará un posible camino de mejora para futuros incrementos de esta solución.

1.3. Convenciones utilizadas en este trabajo

Frente a otros trabajos similares, este cuenta con una gran cantidad de menciones a términos en japonés debido a la temática tratada. Típicamente, esto no habría sido un problema, ya que habría bastado con escribir una romanización de estas palabras en cursiva de manera similar a como se estila con idiomas como el inglés. Sin embargo, en este caso hay múltiples ocasiones en las que es necesario mantener la escritura original. Por ello, es imprescindible una distinción entre dos tipos de usos de términos en japonés: para conceptualizar y para ejemplificar.

Se considera que un término se utiliza para conceptualizar cuando hace referencia a realidades que no pueden describirse con una palabra en español manteniendo el mismo significado. Algunos ejemplos son *kanji* (carácter del japonés que proviene originalmente del chino) o *rendaku* (un fenómeno lingüístico propio de la lengua japonesa). Estos términos serán escritos utilizando la romanización *waapuro* (un tipo de romanización del japonés

muy utilizada en escritura con teclado) para facilitar su lectura[1] a lectores que no conozcan los caracteres japoneses. Además, se formatearán en cursiva para destacar que son palabras extranjeras.

Se considera que un término se utiliza para ejemplificar cuando se use con el objetivo de mostrar alguna característica o fenómeno del idioma. En este caso, el termino se escribirá con el conjunto de caracteres que mejor presente la característica o fenómeno a destacar, que en la mayoría de las ocasiones será alguno de los conjuntos de caracteres del japonés. En los casos en que se utilicen caracteres japoneses se les acompañará, siempre que sea posible[2], de su romanización *waapuro* sobre ellos para permitir su lectura. También, cuando se pueda, se mencionará el significado entrecomillado y entre paréntesis. Por ejemplo, nichi you bi 日曜日(“domingo”) podría leerse “nichiyoubi” y significa “domingo”. Además, en los casos en los que se quiera destacar una parte específica del ejemplo, esta se enmarcará dentro de un rectángulo, por ejemplo, en nichi 日 you 曜 bi 日(“domingo”) se está destacando el primer carácter de la palabra.

A veces, será necesario presentar alguna escritura adicional de una palabra. Para ello, se usará el siguiente formato: nichi you bi 日曜日【にちようび】(“domingo”) donde にちようび es una escritura alternativa de nichi you bi 日曜日 con la misma lectura y significado.

Cuando un ejemplo sea demasiado complejo como para introducirlo en línea con el texto, se hará uso del rotulo “Ejemplo”. Por ejemplo, ver Ejemplo 1. En estos casos, dado que el ejemplo ya no interrumpe el flujo de lectura de una oración, la romanización podrá omitirse cuando no sea relevante. En su lugar, podrá aparecer el *furigana*[3] de los *kanjis* presentes.

[1] No es una lectura muy precisa, pero es lo suficientemente cercana como para ilustrar los temas tratados en esta memoria y permite que el lector identifique más fácilmente las palabras al encontrarse en un sistema de escritura que conoce y comprende.
[2] Hay casos en los que un carácter puede no tener una pronunciación asignada, por ejemplo, un *kanji* sin contexto.
[3] Transcripción fonética de un *kanji* o palabra del japonés a base de caracteres del *hiragana* y que se escribe sobre la palabra original.

これは日本語(にほんご)の例(れい)です。

Esto es un ejemplo de japonés.

Ejemplo 1. Este es un ejemplo de japonés.

2. MARCO TEÓRICO

En este apartado se describen los conceptos necesarios para entender el dominio del problema. En concreto, se considera imprescindible conocer algunos conceptos básicos del japonés, que se describen en el apartado 2.1. También, en la sección 2.2, se comentará brevemente cuál es el consenso actual sobre el funcionamiento de la memoria humana, lo que será necesario para entender cómo funcionan los algoritmos de repetición espaciada del apartado 3.1 y algunas de las propuestas de este trabajo.

2.1. Japonés

En este apartado se pretende hacer una introducción lo más simple posible a la lengua japonesa de manera que sea posible comprender el resto de este documento. Sin embargo, hay que tener en cuenta que este tema es complejo y requeriría de mucho más espacio del que se dispone en esta memoria para explicarlo con un alto rigor y precisión. Por lo tanto, se hacen algunas simplificaciones y se omiten detalles importantes para una correcta compresión del idioma pero que quedan fuera del alcance de este escrito. De hecho, esta explicación se limitará al sistema de escritura, uno de los temas principales de este trabajo. Para conocer el idioma de una manera más exhaustiva y rigurosa se recomienda acudir a libros como みんなの日本語 (*Minna no Nihongo*), げんき (*Genki*) y la saga *A Dictionary of Japanese Grammar*.

En la actualidad el sistema de escritura japonés utiliza una combinación de logogramas y caracteres silábicos. Estos caracteres se usan conjuntamente a la hora de escribir oraciones (ver Ejemplo 2). Además, en el japonés moderno también puede utilizarse el alfabeto latino y los numerales arábigos para ciertos usos. Por ejemplo, en Ejemplo 2 se puede ver el uso de caracteres logográficos (円, 出, 私 o 買), silábicos (て[te], を[wo], し[shi], チ[chi], レ[re], ト[to] ...), caracteres latinos (típicamente llamados *romaji*) y numerales arábigos. Las reglas que se siguen a la hora de combinar todos estos conjuntos de caracteres distintos se explican en detalle en el apartado 2.1.3.

El conjunto de todos los caracteres silábicos del japonés es conocido como *kana* y se divide en dos silabarios (i.e., alfabetos de sílabas): el *hiragana* y el *katakana*. En Ejemplo 2 los caracteres て[te], を[wo] y し[shi] son ejemplos del *hiragana* y los caracteres チ[chi], レ[re] y ト[to] son ejemplos

del *katakana*. En Tabla 2 y Tabla 5 del anexo 8.1 se puede ver una tabla con los caracteres del *hiragana* y otra con los del *katakana*, respectivamente. Además, en el subapartado 2.1.1Kana se describirá más en detalle su uso.

ＡＴＭで１００円(えん)を出(だ)して、チョコレートを私(わたし)に買(か)ってください。

Saca 100 yenes en un cajero y cómprame chocolate, por favor.

Ejemplo 2. Uso de los distintos tipos de caracteres en el japonés moderno.

Los logogramas mencionados anteriormente son llamados *kanjis*, y son caracteres que se importaron del chino[4] en torno al siglo V. Desde entonces, los *kanjis* y el sistema de escritura chino han evolucionado independientemente hasta el punto en que cualquier parecido entre ambos es fortuito. En el apartado 2.1.2 se explican más detalladamente estos logogramas.

Para los caracteres latinos y los numerales arábigos existen variantes monoespaciadas, normalmente denominadas “de ancho completo”, que encajan mejor con los caracteres japoneses (que son siempre monoespaciados), ver Ejemplo 3 para una muestra. Estas variantes tienen un punto de código Unicode distinto a los caracteres utilizados en lenguas occidentales.

ATM で 100 円を出して... ↔ ＡＴＭで１００円を出して．．．

Ejemplo 3. Comparativa entre caracteres latinos (izquierda) y su alternativa de ancho completo (derecha).

2.1.1. Kana

Como se ha anticipado en la introducción de este apartado, los *kana* son caracteres silábicos, es decir, son caracteres que representan sílabas (una secuencia de fonemas). Nótese la diferencia con los sistemas de escritura alfabéticos (por ejemplo, el latino) donde

[4] El japonés antiguo existía previamente a la adopción de los *kanjis*, al menos de manera hablada. Si existía sistema de escritura o no es objeto de debate entre expertos en arqueología y filología japonesa.

cada símbolo se corresponde con un único fonema. Por ejemplo, el carácter も es equivalente a la sílaba “mo” del español. Los *kana* suelen pronunciarse siempre igual independientemente de la palabra en que aparezcan[5]. Además, cada carácter (o combinación de caracteres) del *hiragana* se corresponde con otro del *katakana* con el mismo sonido y viceversa[6]. Por ejemplo, el carácter del *hiragana* ふ(fu) y el carácter del *katakana* フ(fu) tienen el mismo sonido.

Algunos *kana* se pueden acentuar utilizando *dakuten*, ◌゙, o *handakuten*, ◌゚. Cuando un carácter se acentúa con *dakuten* su sonido consonántico pasa a ser sonoro y cuando un carácter se acentúa con *handakuten* su sonido consonántico pasa a ser semisonoro. Un ejemplo de cómo afectan estos acentos al carácter ふ(fu), se puede ver en Ejemplo 4. El *dakuten* se utiliza en palabras como, por ejemplo, テ(te)レ(re)ビ(bi) (“televisión”) y でんわ(de n wa)（電話）(“teléfono”) y el *handakuten* en, por ejemplo, ス(su)ペ(pe)イ(i)ン(n) (“España”) y えん(e n)ぴ(pi)つ(tsu)（鉛筆）(“lapicero”). En Tabla 3 y Tabla 6 del anexo Tablas de *kana* se pueden consultar los caracteres acentuables con *dakuten* y *handakuten*.

ふ(fu) ＋ ゙ （d a k u t e n）⇒ ぶ(bu)

ふ(fu) ＋ ゚ （h a n d a k u t e n）⇒ ぷ(pu)

Ejemplo 4. Ejemplo de uso de *dakuten* y *handakuten*.

Tanto en *hiragana* como en *katakana* es posible formar dígrafos, es decir, combinaciones de dos *kana* que forman una sílaba nueva. Estos dígrafos se forman utilizando un *kana* que termine con la vocal “i” y un *kana* que comience con la consonante “y”. Ambos *kana* deben pertenecer al mismo silabario (dos caracteres del *hiragana* o dos del *katakana*). Al formar

[5]Algunas excepciones notables son は(ha) que se pronuncia “wa” al utilizarse como marcador de tema, を(wo) que se pronuncia “o” cuando se utiliza como marcador de objeto directo y へ(he) que se pronuncia “e” cuando se utiliza como marcador de dirección.

[6] Las sílabas de nueva creación del *katakana* (ver Tabla 8 del anexo 8.1), que son normalmente dígrafos, no tienen correspondencia en *hiragana*.

un dígrafo el carácter que empieza por el fonema “y” debe aparecer a un menor tamaño tras el carácter que termina en el fonema “i” como se muestra en el Ejemplo 5. Algunas palabras que utilizan esto son きょう（今日）(“hoy”) y チョコレート (“chocolate”). En el anexo Tablas de *kana* se incluyen Tabla 4 y Tabla 7 con la lista completa de dígrafos para hiragana y katakana.

ki + ya ⇒ kya
き + や ⇒ きゃ

chi + yu ⇒ chu
チ + ユ ⇒ チュ

Ejemplo 5. Formación de dígrafos con hiragana (arriba) y katakana (abajo).

También es frecuente elongar un sonido vocálico al final de una sílaba. La forma de representar esto varía en función de si se está utilizando *hiragana* o *katakana*. En *hiragana* se representa una elongación de cualquier sílaba terminada en vocal utilizando los *kana* puramente vocálicos: normalmente se utilizan あ (para sílabas terminadas en “a”), い (para sílabas terminadas en “e” o “i”) y う (para sílabas terminadas en “o” o “u”), pero también puede verse el uso de え (para sílabas terminadas en “e”) y お (para sílabas terminadas en “o”). La palabra びょういん【病衣】(“hospital”)[7] es un ejemplo de elongación de vocales en *hirgana*. En *katakana* la elongación es mucho más sencilla ya que se utiliza la marca de vocal larga (o *chouonpu* en japonés), ー. La palabra アイスクリーム (“helado”) es un ejemplo de elongación de vocales en *katakana*. En Tabla 1 se muestra una comparativa sobre cómo elongar las vocales en ambos silabarios.

Al igual que en ocasiones se elonga la vocal al final de una sílaba, a veces se duplica la consonante inicial de una sílaba. En este caso se marca esta doble consonante añadiendo el hiragana つ o el katakana ツ, dependiendo del silabario que se esté utilizando, en un menor tamaño delante de la sílaba cuya consonante se quiere duplicar. Por ejemplo, este fenómeno lingüístico ocurre en las palabras はっぴゃく【八百】(“800”) y チケット (“tique”).

[7] Nótese que, aunque se transcriba como “byouin”, se lee “byooin”.

Tabla 1. Elongación de vocales.

	Hiragana	Katakana
k a + a	かあ	カー
s e + e	せい[8] o せえ	セー
c h i + i	ちい	チー
h o + o	ほう[9] o ほお	ホー
k y u + u	きゅう	キュー

2.1.2. Kanjis

Previamente, se ha mencionado que los *kanjis* son logogramas, lo que significa que son caracteres que tienen un significado intrínseco (aunque en muchas ocasiones puede ser simbólico o metafórico). Por ejemplo, el *kanji* 出 indica movimiento del interior hacia el exterior y aparece en 出る(“salir”), 出す(“sacar”) o 出口 (“salida”), comparativamente, en el español la letra “a” de “salir”, “sacar” y “salida” no tiene ninguna relación con su significado.

Además de un significado, cada *kanji* tiene una o más pronunciaciones asociadas, es decir, un *kanji* puede pronunciarse diferente en función de la palabra en la que aparezca. Por ejemplo, 出 se pronuncia diferente en 出る(“salir”) y 出す(“sacar”). Es frecuente que un *kanji* tenga dos pronunciaciones, una heredada del chino y otra original del japonés, pero hay muchas excepciones. Una excepción notable es el *kanji* 日 (“sol”) que aparece con distintas pronunciaciones en 日(“día”), 日曜日 (“domingo”), 今日(“hoy”), 一日 (“primer día de mes”) o 二十日 (“veinteavo día del mes”). Nótese que las tres últimas palabras son un caso de pronunciación irregular en el que sería difícil asociar qué fonemas de la pronunciación corresponden a cada *kanji*.

Una de las razones por las que un *kanji* pasa a pronunciarse de manera diferente es el fenómeno lingüístico conocido como *rendaku*: cuando se forma una palabra nueva a partir

[8] Nótese que, aunque se transcriba como “sei”, se lee “see”.
[9] Nótese que, aunque se transcriba como “hou”, se lee “hoo”.

de otras dos palabras, la primera sílaba de la segunda palabra pasa a ser sonora (se modifica con *dakuten*) si era sorda como se ilustra en Ejemplo 6.

toki 時【とき】 + toki 時【と|き】 ⇒ toki doki 時々【とき|ど|き】[10]

tiempo + tiempo ⇒ a veces

hito 人【ひと】 + hito 人【ひ|と】 ⇒ hito bito 人々【ひと|び|と】

persona + persona ⇒ personas

de 出【で】 + kuchi 口【く|ち】 ⇒ de guchi 出口【で|ぐ|ち】

salida (como en salida del sol) + boca o apertura ⇒ salida (puerta de, orificio de)

ori 折【おり】 + kami 紙【か|み】 ⇒ ori gami 折紙【おり|が|み】

doblez + papel ⇒ papiroflexia

Ejemplo 6. *Rendaku* en varias palabras compuestas.

Al igual que un *kanji* puede tener varias pronunciaciones, son extremadamente frecuentes los *kanjis* con las mismas pronunciaciones, por ejemplo, 日 (“sol”) y 二 (“2”) se pronuncian “ni” en ni hon 日本 (“Japón”) y ni 二 (“2”), respectivamente. Debido a esto, las palabras homófonas en japonés son comunes, por ejemplo, kami 紙 (“papel”), kami 神 (“dios”) y kami 髪 (“pelo”) se pronuncian igual.

Una palabra con *kanjis* puede transcribirse fonéticamente utilizando *hiragana* o *katakana* (se matiza cuándo se hace uso de cada uno de los silabarios en el apartado 2.1.3). Por ejemplo, da su 出す y da su だす son la misma palabra (“sacar”).

Aunque los *kanjis* tengan significado, hay ocasiones en las que estos se usan por su sonido y no por su significado. Por ejemplo, en el caso de su shi 寿司 (“sushi”) los *kanjis* significan

[10] El *kanji* 々 se utiliza para evitar duplicar el mismo *kanji* dos veces seguidas, es decir, indica que se debe repetir el *kanji* que le precede.

“longevidad” y “director” que, claramente, no están muy relacionados con “sushi”. Este fenómeno es conocido como *ateji*.

Por último, mencionar la existencia de radicales en los *kanjis*, es decir, componentes que se repiten en múltiples *kanjis* con significados similares. Por ejemplo, 木 (“árbol”), 林 (“arboleda”), 森 (“bosque”), 机 (“escritorio”), 材 (“madera”) y 本 (“libro”) comparten el radical de “árbol” (木) porque tienen significados de una misma familia semántica. Algunos radicales no se parecen tanto al kanji del que parten, por ejemplo, en el caso de 水 (“agua”) el radical puede ser 氵 o 氺. Es un error muy común creer que un radical aparece en un kanji y asumir un significado incorrecto ya que hay kanjis comunes que se parecen mucho pero no tienen nada que ver, por ejemplo, 牛 (“vaca”), 午(“mediodía”) y 年 (“año”) no tienen ningún radical en común. También, hay veces que un radical aparece con un significado metafórico, aportando la pronunciación o, simplemente, por casualidad.

2.1.3. Reglas de uso

Cuándo se utiliza cada uno de los componentes de la escritura japonesa (*hiragana*, *katakana* y *kanjis*) está especificado por medio de múltiples reglas (la mayoría de las cuales son más una recomendación estilística que un mandato) que, a grandes rasgos, podrían resumirse en las siguientes:

1. Las palabras de origen japonés (incluidos nombres propios en japonés) se escriben utilizando una combinación de *kanjis* e *hiragana*. Por ejemplo, 出(da)す(su) (“sacar”) es una mezcla de ambos, 日(ni)本(hon)語(go) (“japonés”) utiliza sólo *kanjis*, y する(su)(ru) (“hacer”) utiliza sólo *hiragana*.
2. Los *kanjis* pueden sustituirse por *hiragana*. Por ejemplo, 日(ni)本(hon)語(go) puede escribirse に(ni)ほん(ho)ん(n)ご(go) y 出(da)す(su) como だ(da)す(su). La utilización de *kanjis* da lugar a un estilo de escritura más culto, pero es habitual utilizar *hiragana* en su lugar si el escritor o su público objetivo desconocen los *kanjis* de la palabra (por ejemplo, en libros infantiles). También, cuando es un *kanji* muy complejo o en desuso. Un ejemplo de este último caso es する(su)(ru) que, infrecuentemente, puede encontrarse como 為(su)(ru).

3. Las palabras de origen extranjero, sobre todo si son de importación reciente, se escriben utilizando *katakana*, por ejemplo, チョコレート(cho ko re e t o) (“chocolate”) y パソコン(pa so ko n) (“ordenador”).
4. Hay casos de palabras importadas en el pasado que cuentan, además, con una escritura que utiliza *kanjis*, por ejemplo, タバコ(ta ba ko) (“tabaco”) puede escribirse también 煙草(tabako). Incluso en estos casos, es más frecuente el uso de *katakana*.
5. Estas reglas pueden incumplirse por motivos estilísticos. Por ejemplo, es común utilizar *katakana* de una forma similar a la cursiva en español o para representar una voz robótica (aunque sean palabras típicamente escritas con *kanjis* e *hiragana*).

2.2. Memoria y aprendizaje

La memoria es la capacidad humana de almacenar y recuperar información. Los detalles precisos sobre su funcionamiento siguen siendo objeto de estudio en áreas del conocimiento como la neurociencia y la psicología.

El modelo de Atkinson y Schiffrin [4] divide la memoria en tres componentes: el registro sensorial, la memoria a corto plazo y la memoria a largo plazo.

El registro sensorial almacena la información percibida por los órganos de los sentidos durante un periodo muy breve de tiempo. Si se le presta atención a esta información, esta pasaría a la memoria a corto plazo, pero si no, esta información desaparecerá.

La memoria a corto plazo almacena la información con la que la persona está trabajando en el momento. Tiene una capacidad bastante limitada y una duración de unos 30 segundos, pero puede alargarse por medio del ensayo de memoria. También puede recibir información desde la memoria a largo plazo.

La memoria a largo plazo almacena la información, según Atkinson y Schiffrin, casi permanentemente, así que sería el componente con mayor capacidad y duración. Más adelante se presentarán trabajos que analizan la duración de esta memoria más exhaustivamente. La probabilidad de que la información pase de la memoria a corto plazo a la memoria a largo plazo aumenta cuanto más tiempo permanezca en la memoria a corto plazo.

En este trabajo se le prestará más atención a la memoria a largo plazo dado que es la responsable de generar recuerdos duraderos y, por tanto, la memoria más relevante para una aplicación de ayuda al estudio.

Dentro de la memoria a largo plazo, Anderson [5] distingue dos tipos: la memoria implícita y la memoria explícita. Por un lado, está la memoria implícita que es un tipo de memoria a largo plazo que permite recuperar información de manera inconsciente, sin intervención voluntaria de la propia persona. Uno de los tipos más conocidos de memoria implícita es la memoria procedural que es la responsable de almacenar el cómo realizar ciertos procesos, principalmente los relacionados con habilidades motoras. Por otro lado, se encuentra la memoria explícita que sí requiere un esfuerzo a la hora de recuperar la información. Este segundo tipo de memoria es, por ejemplo, la responsable de almacenar recuerdos (en la memoria episódica) y datos (en la memoria semántica).

En definitiva, para el caso del aprendizaje de un idioma, la parte fonética haría uso de la memoria implícita (la pronunciación es una habilidad motora) y el vocabulario y las reglas sintáctico-gramaticales, en la mayoría de los casos, implican utilizar la memoria explícita.

Llegados a este punto, aún queda una pregunta por responder, ¿cuánto tiempo permanece la información en la memoria? Ebbinghaus realizó diversos experimentos (sobre sí mismo) en los que analizaba cuánto tiempo era capaz de rememorar información. El resultado de dichos experimentos es la conocida como curva del olvido [6] que relaciona la probabilidad de recordar la información con el tiempo que transcurre desde la última vez que se recordó. Ebbinghaus postula que la probabilidad de recordar un fragmento de información con éxito decrece exponencialmente con una velocidad que varía en base a la complejidad de la información a recordar y la relación con otros conceptos conocidos, entre otros. En la Figura 1 se muestra una representación de la evolución de la probabilidad de recordar algo a lo largo del tiempo.

Sobre los avances de Ebbinghaus, Wozniak [7] observó que al recordar la información se mejora la probabilidad de recordarla a corto plazo. De hecho, no sólo ocurre esto, sino que también se reduce la velocidad a la que esta probabilidad decrece. Reformulando la oración anterior, cada vez que se recupera información desde la memoria a largo plazo las

representaciones de los conceptos almacenados se consolidan, lo que facilita el proceso de recuperación.

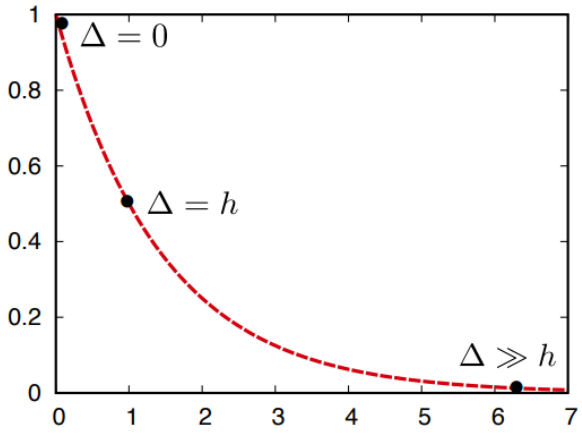

Figura 1. Modelo de Ebbinghaus[11].

En la Figura 2 se muestra el efecto que tiene realizar múltiples revisiones sobre la curva del olvido. En cada revisión la probabilidad de recordar el concepto se maximiza y, además, la pendiente de la curva es cada vez menor conforme aumenta el número de revisiones realizadas.

En base a las observaciones sobre el modelo de Ebbinghaus, Wozniak formula su propio modelo de memoria basado en dos componentes [8]: la estabilidad y la recuperabilidad. La recuperabilidad hace referencia a la probabilidad de recordar un concepto en un instante de tiempo. Este valor es menor cuanto más tiempo haya pasado desde la última revisión y se puede representar utilizando una curva del olvido. La estabilidad hace referencia al tiempo que puede permanecer en memoria un concepto si no se recupera. Cada vez que un concepto es recordado aumenta su estabilidad. Más tarde, Wozniak mejora este modelo añadiendo un tercer componente, la complejidad, que cuantifica la dificultad de aumentar la estabilidad al recordar un concepto. Esta última iteración del modelo es conocida como modelo de memoria basado en tres componentes o modelo de memoria DSR (siglas de dificultad, estabilidad y recuperabilidad).

[11] Esta figura ha sido obtenida de [10].

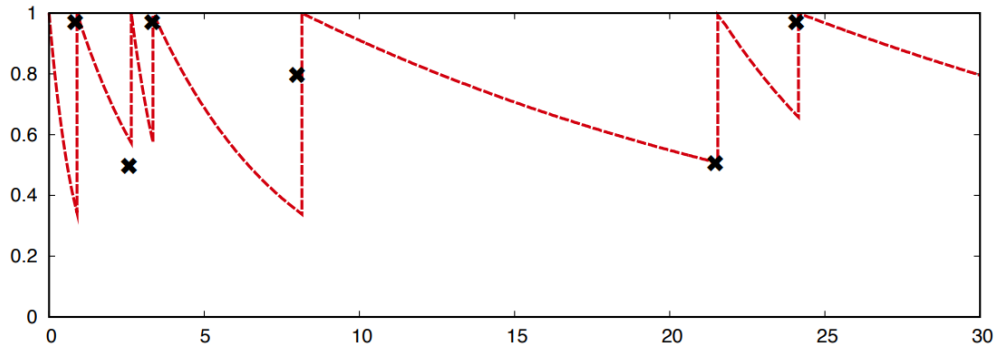

Figura 2. Curva del olvido tras realizar diversas repeticiones[12].

[12] Esta figura ha sido obtenida de [10].

3. ESTADO DEL ARTE

En este apartado se describen los conocimientos necesarios para comprender el dominio de la solución propuesto. En concreto, se detalla la teoría relacionada con la repetición espaciada en el apartado 3.1 y el *embedding* en el apartado 3.3. Además, se hace un análisis de diversas aplicaciones para el estudio de japonés en el apartado 3.2.

3.1. Repetición espaciada

Previamente a poder explicar la repetición espaciada es necesario definir el término "repetición" o "revisión" en este contexto. En concreto una revisión hace referencia a una prueba de memoria, es decir, una actividad que involucre recordar un dato. La repetición será "exitosa" si el dato se recuerda y "fallida" si el dato no es recordado (se ha olvidado).

La repetición espaciada es una técnica de estudio que tiene como objetivo garantizar que un concepto permanezca en memoria indefinidamente, en concreto, maximiza el aprovechamiento de la memoria explícito basándose en la teoría de la curva de olvido de Ebbinghaus [6] y las observaciones de Wozniak [7]. Esta optimización se consigue realizando revisiones a intervalos de tiempo (o, en otras palabras, espaciadas) que fuercen al usuario a recordar el concepto. Si la revisión es exitosa, la estabilidad será mayor, así que el intervalo hasta la siguiente prueba aumentará. Si la prueba se falla, el concepto ha sido olvidado y, por tanto, necesita ser aprendido de nuevo. En este caso, el intervalo de repetición se reduce o, incluso, se reinicia a un valor inicial.

Existen múltiples variantes de este método que se diferencian, principalmente, en el mecanismo utilizado para calcular el tiempo entre pruebas. Algunos algoritmos de repetición espaciada son el sistema Leitner [9], SuperMemo 2 [7], la regresión de la vida media [10] y Free Spaced Repetition Scheduler [11]. A lo largo de este apartado se describirán en detalle estos métodos de repetición espaciada.

3.1.1. Sistema de cajas de Leitner

El sistema de cajas de Leitner [9] es el método más antiguo de repetición espaciada conocido y surge como un mecanismo para ordenar un *zettelkasten* (una biblioteca de

conocimiento basada en tarjetas con información que se referencian unas a otras) para facilitar su recuperabilidad.

Este método implementa la repetición espaciada ordenando tarjetas de información en una serie de cajas. Las tarjetas de la primera caja se revisarán con más frecuencia que las de la segunda caja, las de la segunda con más frecuencia que las de la tercera y así sucesivamente. Cuando una tarjeta se revisa con éxito, la tarjeta avanza a la siguiente caja. Y, cuando se falla la revisión de una tarjeta, esta se devuelve a la primera caja (o, en algunas variantes, a la caja inmediatamente anterior a la actual). De esta forma, cada una de las cajas agrupa tarjetas por niveles de estabilidad desde el más bajo que se corresponde con la primera caja al más alto que sería la última caja.

Este sistema cuenta con diversas desventajas, por ejemplo, la dificultad para establecer el intervalo de revisión de cada caja, los problemas de acumulación de tarjetas (ya que no se establece límite de tarjetas por caja) o que el número óptimo de cajas necesarias puede ser bastante alto.

Una variante de este sistema fue utilizada por Duolingo en el pasado [10].

3.1.2. SuperMemo 2

SuperMemo 2 [7] es el primer algoritmo de repetición espaciada que está especialmente diseñado para ser implementado en equipos informáticos. Este algoritmo se define a nivel de “objetos de conocimiento del menor tamaño posible” a los que llamaremos “concepto”.

En este método se definen dos variables para cada concepto: el intervalo de repetición (Δt) y el factor de facilidad (f). El intervalo de repetición define el tiempo entre una revisión de un concepto y la siguiente en días. Y, por otro lado, el factor de facilidad caracteriza su dificultad. Cuanto mayor sea el factor de simplicidad, más fácil será.

Durante el estudio, el usuario realizará revisiones de cada concepto en base al intervalo de repetición. En cada repetición, tanto el intervalo de repetición como el factor de facilidad se recalcularán para tener en cuenta los cambios en la recuperabilidad, estabilidad y facilidad del concepto. Este ciclo de estudio se produce de la siguiente manera:

1. Al añadir un concepto, el sistema configuraría los valores iniciales para el intervalo de repetición, Δt, y el factor de facilidad, f, utilizando los valores descritos como Δt_1 y f_1, respectivamente, en Ecuación 1 y Ecuación 2.

$$\Delta t_1 = 1$$

$$\Delta t_2 = 6$$

$$\Delta t_i = \lceil \Delta t_{i-1} * f_{i-1} \rceil$$

Ecuación 1. Cálculo del intervalo de repetición en SuperMemo 2.

$$f_1 = 2.5$$

$$f_i = \max(f_{i-1} - 0.8 + 0.28 * q - 0.02 * q^2, 1.3)$$

Ecuación 2. Cálculo del factor de facilidad en SuperMemo 2.

2. Al pasar el intervalo de repetición, se realizará la primera revisión del concepto y el usuario evaluará su facilidad para recordarlo utilizando un número del 0 (completamente olvidada) al 5 (recordada a la perfección). Esta valoración se nombrará q. Tras esto se actualizarán el intervalo de repetición y el factor de facilidad siguiendo los siguientes criterios:
    - Si $q < 3$, entonces se reiniciará el intervalo de repetición (comenzando de nuevo por Δt_1) y el factor de facilidad se actualizará siguiendo la Ecuación 2 (sin reiniciarlo).
    - Si $q \geq 3$, entonces tanto el intervalo de repetición como el factor de facilidad se actualizarán siguiendo la Ecuación 1 y Ecuación 2.
3. Si $q < 4$, continuar revisando el concepto durante el día de hoy hasta que $q \geq 4$.
4. Volver al paso 2.

Este algoritmo es notablemente más complejo que el sistema de Leitner y cuenta con una gran cantidad de constantes calculadas heurísticamente que pueden ser subóptimas para algunas personas. Sin embargo, cuenta con la ventaja de que los intervalos de repetición

son individualizados por concepto, así que no se acumulan demasiadas revisiones el mismo día. También, permite que conceptos más fáciles incrementen su intervalo de repetición a mayor velocidad, reduciendo el número de revisiones innecesarias.

Una modificación de este algoritmo es la utilizada por Anki en la actualidad.

3.1.3. Regresión de la vida media

La teoría de Ebbinghaus plantea que la curva del olvido decrecía exponencialmente con el tiempo. Settles *et al.* [10] representa esta curva con Ecuación 3, donde $\mathcal{R}$ hace referencia a la recuperabilidad (probabilidad de recordar un concepto), Δt el tiempo desde la última repetición y h es la vida media que representa el tiempo que debe pasar desde la última revisión para que la probabilidad de recordar la tarjeta sea del 50% (nótese que este valor es un método de medir la estabilidad del modelo de la memoria de dos componentes [8]).

$$\mathcal{R}_h(\Delta t) = 2^{-\frac{\Delta \mathrm{t}}{h}}$$

Ecuación 3. Modelo matemático de la curva del olvido de Ebbinghaus.

En base a este modelo matemático se destacan tres casos:

- Nada más realizar una revisión la recuperabilidad es máxima, $\mathcal{R}_h(0) = 1$.
- Cuando el tiempo que ha pasado desde la última revisión es igual a la vida media, recuperabilidad es del 50%, $\mathcal{R}_h(h) = 0.5$. En este método se asume que este es el punto óptimo para realizar una revisión.
- Cuando ha pasado mucho tiempo desde la última repetición y, por tanto, el tiempo desde la última revisión es muy superior a la vida media, $\Delta t \gg h$, la recuperabilidad será mínima, $\lim_{\Delta t \to \infty} \mathcal{R}_h(\Delta t) = 0$.

En este método, el quid de la cuestión es poder aproximar el valor de la vida media, ya que es equivalente al intervalo de repetición (en este método). Para ello los autores utilizan un método de regresión basado en Ecuación 4 donde:

- $h_{c,i}$ es el tiempo de vida media del concepto c tras la repetición i.

- θ es el vector de parámetros a optimizar.
- x_c es un vector de características asociado al concepto c. Este vector está formado por un identificador (para que la predicción pueda variar en función del concepto) y diversas características relacionadas con el historial de estudio del usuario (entre otras, el número de veces que ha visto el concepto, el número de veces que se recordó con éxito y el número de veces que no se recordó).

Nótese que, además, los autores están asumiendo que el tiempo de vida medio debería variar exponencialmente.

$$h_{c,i} = 2^{\theta \cdot \mathrm{x_c}}$$

Ecuación 4. Ecuación de regresión de la vida media.

Los autores, que son trabajadores de Duolingo, utilizan el algoritmo del descenso del gradiente para encontrar el valor de θ que minimice una función de pérdida $\mathcal{L}(\theta)$ sobre un conjunto de datos recuperado de usuarios de Duolingo. La función de pérdida penaliza en base al error cuadrático sobre $h_{c,i}$ y sobre la recuperabilidad $\mathcal{R}_{h_{c,i}}$ (ver Ecuación 3). También, se incorpora una regularización L2 sobre θ en la función de pérdida.

En base a los resultados del estudio de Settles *et al.*, este algoritmo realiza una mejor predicción de la curva del olvido que el método de Leitner. Además, puede, dado que incorpora información sobre los conceptos de estudio, tener en cuenta relaciones entre múltiples conceptos a la hora de estimar su tiempo de vida medio. Sin embargo, asume que el punto óptimo para revisar un concepto es el tiempo de vida medio cuando se ha demostrado que, a menor recuperabilidad, mejores son las ganancias obtenidas en la estabilidad tras la revisión [12]. También, dada la simplicidad del modelo (es equivalente a una única neurona de una red neuronal) es posible que no sea capaz de modelar adecuadamente relaciones complejas entre diferentes conceptos. Debido a esto, dado que no utiliza ningún modelo de dificultad explícita, es improbable que este se tenga en cuenta con éxito. Por último, es importante destacar que es bastante impreciso al calcular el nuevo tiempo de repetición tras revisiones fallidas [11].

3.1.4. Free Spaced Repetition Scheduler

Ye *et al.* [11] se inspiraron en el método de regresión de la vida media para desarrollar su propia aproximación. En su algoritmo, Ye *et al.* proponen un modelo basado en la propiedad de Markov (la evolución del sistema es independiente del histórico, es decir, el estado actual depende únicamente del estado inmediatamente anterior) para aproximar el valor de la vida media y, además, la dificultad del concepto. En concreto, este modelo se formula como en la Ecuación 5 donde:

- h_i es el tiempo de vida medio para un concepto tras la repetición i.
- d_i es la dificultad de un concepto tras la repetición i.
- r_i es 1 si se recordó el concepto en la repetición i y 0 si no se recordó.
- θ_1 y θ_2 son vectores de parámetros que se optimizarán en base a datos experimentales. Son los mismos para todos los conceptos y repeticiones.
- x_i es el vector de características de un concepto. Está formado por el logaritmo de la dificultad anterior (d_{i-1}), el logaritmo del tiempo de vida medio anterior (h_{i-1}) y el logaritmo de la probabilidad de haber olvidado el concepto, es decir, $1 - \mathcal{R}_{h_i}(\Delta t_{i-1})$ calculado con la Ecuación 3 del método de regresión de la vida media con el tiempo transcurrido hasta la revisión actual, Δt_{i-1}.

Los valores iniciales de h_1 y d_1 se calculan en base a datos empíricos para distintos grupos de conceptos en función de su dificultad prevista.

$$\begin{bmatrix} h_i \\ d_i \end{bmatrix} = \begin{bmatrix} h_{i-1} \cdot e^{\theta_1 \cdot x_i + 1} & e^{\theta_2 \cdot x_i} \\ d_{i-1} & d_{i-1} + \theta_3 \end{bmatrix} \begin{bmatrix} r_i \\ 1 - r_i \end{bmatrix}$$

Ecuación 5. Función de transición de estados del método Free Spaced Repetition Scheduling Algorithm.

Para calcular el intervalo de revisión, se define una función $\Delta T(h_i, d_i)$ que lo computa a partir de un tiempo de vida medio h_i y una dificultad d_i. La definición que Ye *et al.* proponen en [11] se basa en una matriz que asocia discretizaciones de h_i y d_i a un valor de tiempo de vida específico, aunque en mejoras del método esta fue sustituida por una definición

continua. En cualquier caso, los parámetros de esta función si optimizan con un método de programación dinámica estocástica que minimiza una función de coste $J(h_i, d_i)$ como la de Ecuación 6.

$$J(h_i, d_i) = \mathcal{R}_{h_i}\big(\Delta T(h_i, d_i)\big) \cdot J_{r=1}(h_{i+1}, d_{i+1}) + \Big(1 - R_{h_i}\big(\Delta T(h_i, d_i)\big)\Big) \cdot J_{r=0}(h_{i+1}, d_{i+1})$$

$$J_{r=r}(h_i, d_i, \Delta t_i) = k_{r=r} + J(h_{r=r}, d_{r=r}, \Delta t_{r=r})$$

Ecuación 6. Función de coste de Free Spaced Repetition Scheduler.

Los autores demuestran la superioridad de este método frente al de regresión de la vida media, especialmente en los casos en los que la revisión fue fallada. Además, incorpora un modelo explícito de la dificultad, que va variando con el tiempo. También, es un método con una licencia abierta, que se sigue actualizando hoy en día y que tiene múltiples implementaciones avaladas por los autores en GitHub[13]. Como desventaja se podría mencionar el no tener en cuenta el historial de revisiones en el modelo, aspecto que podría mejorar los resultados.

3.2. Aplicaciones para el estudio de japonés

El japonés es uno de los idiomas más difíciles para los nativos de idiomas occidentales [3]. Por ello, es una de las áreas donde los estudiantes más se benefician de utilizar herramientas que potencien su aprendizaje. Algunas de estas herramientas son las aplicaciones para el estudio de propósito general como Anki[14] o las de estudio de idiomas, ya sean específicas para el japonés como WaniKani[15] o Kanji Study[16], o para múltiples idiomas como Duolingo[17]. En este apartado se realizará una breve descripción de cada una de ellas y se resaltarán sus puntos fuertes y sus puntos débiles.

[13] Todas ellas se encuentran en https://github.com/open-spaced-repetition.
[14] Anki está disponible en https://apps.ankiweb.net/.
[15] WaniKani está disponible https://www.wanikani.com/.
[16] Kanji Study está disponible en https://play.google.com/store/apps/details?id=com.mindtwisted.kanjistudy.
[17] Duolingo está disponible en https://www.duolingo.com/.

3.2.1. Anki

Anki es una aplicación de código abierto que mejora la eficiencia del estudio optimizando por medio de un algoritmo de repetición espaciada, en concreto, una versión ligeramente modificada de SuperMemo 2 [7]. Está disponible para Window, Mac, Linux y Android de manera gratuita y para iOS por 29,99€. También cuenta con una versión web.

El funcionamiento de Anki gira en torno al concepto de "tarjeta" (una metáfora tomada del método de cajas de Leitner). Cada tarjeta es una correspondencia, que puede ser bidireccional, entre dos ideas (cada una de sus dos caras), por ejemplo, una palabra y su traducción en otro idioma, un concepto y su definición, una imagen y una descripción de su contenido, una oración con huecos a rellenar y las palabras que deberían aparecer en esos huecos... Estas tarjetas, además, se pueden ordenar en barajas.

En el momento del estudio se van mostrando en un orden configurable (aleatorio, de más nueva a más antigua...) las tarjetas a repasar. El usuario vería una de sus caras (ver Figura 3), en caso de que sea reversible, o su cara frontal, en caso de que no lo sea, y rememora el contenido de la otra cara. Cuando el usuario ha sido capaz de recordar el contenido o se ha dado por vencido (es posible configurar un temporizador), pulsa sobre la pantalla para ver la cara trasera (ver Figura 3). En este punto, el usuario califica el resultado de su revisión con una de cuatro dificultades en función de si la ha recordado y el esfuerzo que le ha conllevado.

Las tarjetas se pueden crear también desde la aplicación móvil, pero es mucho más eficiente utilizar la aplicación de escritorio para esta tarea. Aun así, como se ve en Figura 4, es una interfaz bastante compleja y con mucho ruido. Además, si se quiere hacer algo ligeramente personalizado es necesario editar los tipos de tarjetas, para lo cual hace falta utilizar HTML, JavaScript y CSS. La versión web no permite crear, editar o eliminar tarjetas.

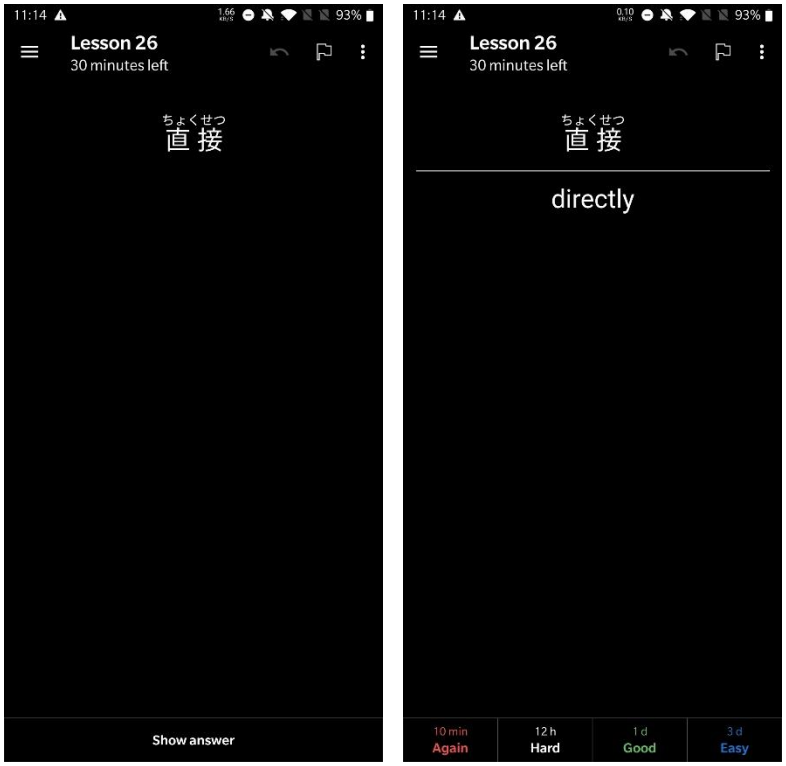

Figura 3. A la izquierda, la cara frontal de una tarjeta y, a la derecha, la cara trasera.

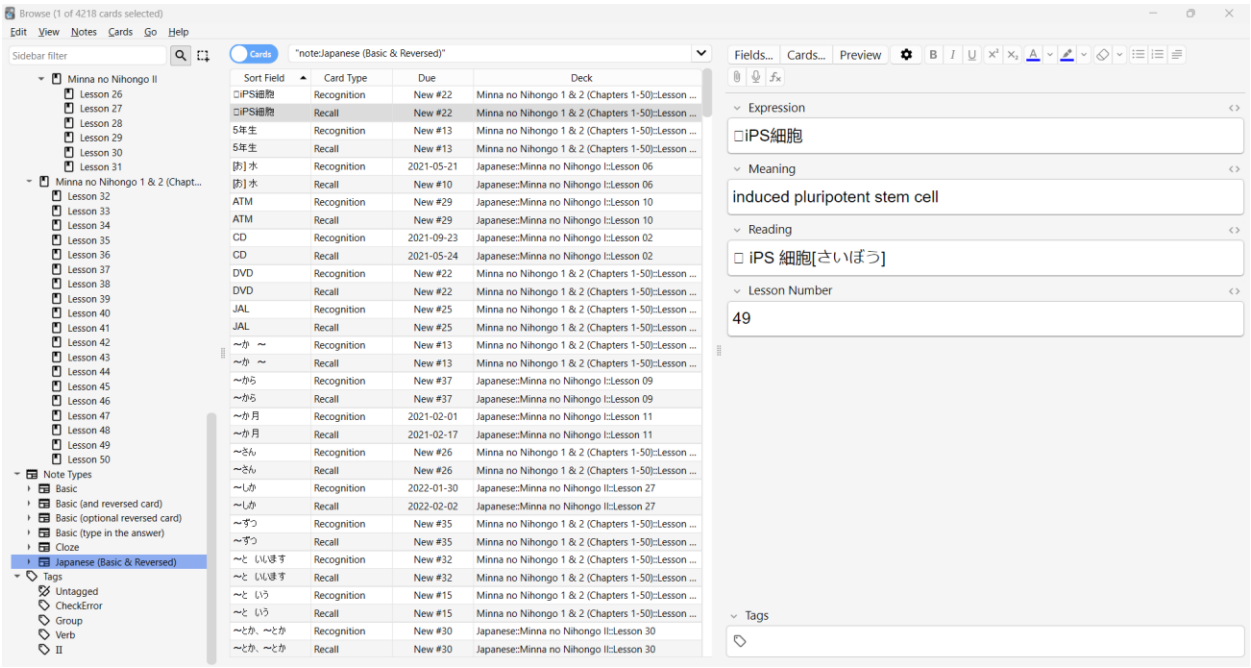

Figura 4. Pantalla de gestión de tarjetas de Anki en Windows.

Anki permite al usuario consultar una infinidad de datos sobre el proceso de estudio. Para ello, pone a disposición del usuario diversas gráficas: predicción del tiempo que llevará dominar todas las cartas de cada uno de las barajas, número de revisiones previstas para cada día, tiempo de revisión esperado para cada día, distribución de los intervalos de repetición, porcentaje de aciertos en función de la hora del día... Gracias a esto, el usuario es capaz de planificar su estudio con precisión e, incluso, adaptar los parámetros del algoritmo a su gusto, ya que son configurables desde el menú de configuración.

En definitiva, Anki es una herramienta muy potente, configurable y personalizable. También es gratuita (en la mayoría de las plataformas) y de código abierto, lo que constituye una gran ventaja. Sin embargo, cuenta con múltiples desventajas: la interfaz es compleja y no muy estética (en comparación a las alternativas), necesita ajustar la configuración y plantillas para algunos casos de uso (como japonés) y carece de contenido propio de calidad (aunque se pueden encontrar barajas compartidas de otros usuarios).

3.2.2. WaniKani

WaniKani es una aplicación web de pago (a partir de cierto nivel) para el estudio de japonés, en concreto, para radicales, kanji y vocabulario. Al igual que Duolingo, proporciona un temario diseñado por expertos y que no puede ser personalizado por el usuario. A partir de este temario, cada día se generan sesiones de estudio utilizando un algoritmo de repetición espaciada. Las sesiones de estudio presentan al usuario un conjunto de ejercicios de traducción directa y de pronunciación (ver Figura 5).

Además, esta aplicación cuenta con un diccionario de radicales, kanji y vocabulario (ver Figura 6) con sus correspondientes significados, pronunciación, ejemplos y reglas mnemotécnicas. Todo esto es capaz de llevarlo a cabo con una interfaz simple y agradable que no genera demasiada fricción al usuario.

En definitiva, es una herramienta simple y de calidad para aprender radicales, kanji y vocabulario. Además, cuenta con una estética bastante agradable. Sin embargo, cuenta con una limitación importante de cara a no poder formar un currículum de estudio propio, por lo que no es adecuada como herramienta de estudio para un curso externo que no

utilice el mismo orden. Además, el tipo de ejercicios, dado que sólo consisten en traducción directa y pronunciación, no fortalecen suficientemente la tarea de traducción inversa.

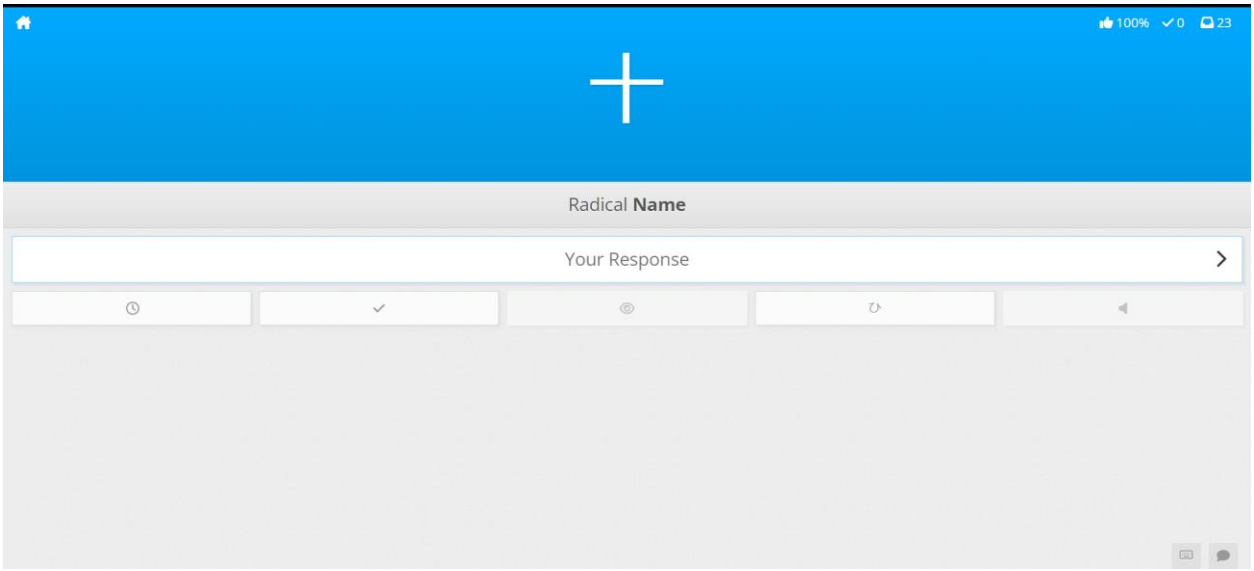

Figura 5. Ejercicio en WaniKani.

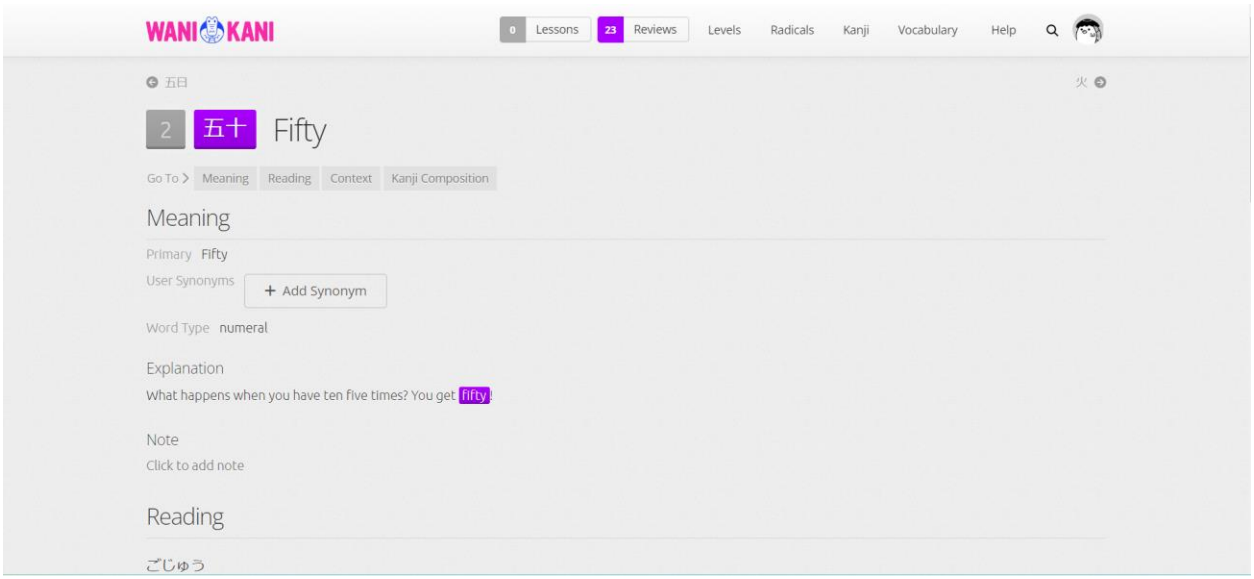

Figura 6. Diccionario de WaniKani.

3.2.3. Kanji Study

Kanji Study es una aplicación gratuita (aunque con extras que requieren un pago) diseñada única y exclusivamente para aprender *kanjis*. Cuenta con una inmensa (y de calidad) biblioteca de contenido de *kanji*, radicales, palabras de ejemplo y definiciones de ambos. Además, cuenta con un complemento de pago en colaboración con Outlier Linguistics que añade aún más información: reglas mnemotécnicas, etimología e información sobre el tono.

Existen diversas colecciones ya creadas por defecto en la propia aplicación asociadas a los cursos escolares en los que se estudian esos *kanjis* en Japón y, también, es posible crear colecciones personalizadas. El usuario puede elegir estudiar cualquiera de estas colecciones utilizando uno de los múltiples métodos disponibles (ninguno utiliza repetición espaciada por defecto): tarjetas similares a las de Anki en las que se asocia un significado a su *kanji* pero sin repetición espaciada, preguntas de respuesta múltiple en las que hay que asociar el *kanji* a su significado, ejercicios de dibujar el *kanji* asociado a un significado y tarjetas similares a las de Anki pero con oraciones completas (requiere un pago adicional de 24,99€). Estos métodos se pueden ver en la Figura 7. Recientemente se ha publicado un paquete que permite añadir repetición espaciada a esta aplicación por medio de un pago adicional, pero no ha podido ser incluido en esta revisión.

En resumen, esta aplicación es un recurso invaluable para el estudio de *kanji* gracias a la calidad y cantidad de su contenido. Además, cuenta con una gran diversidad de modos de estudio que permiten adaptar el sistema a las preferencias de aprendizaje de cada usuario. Incluso es fácil de usar y cuenta con una interfaz agradable. Sin embargo, la repetición espaciada se encuentra tras un paquete de pago y, por lo tanto, requiere una gestión manual por parte del usuario. También es una gran desventaja el que sólo esté disponible en el sistema operativo Android.

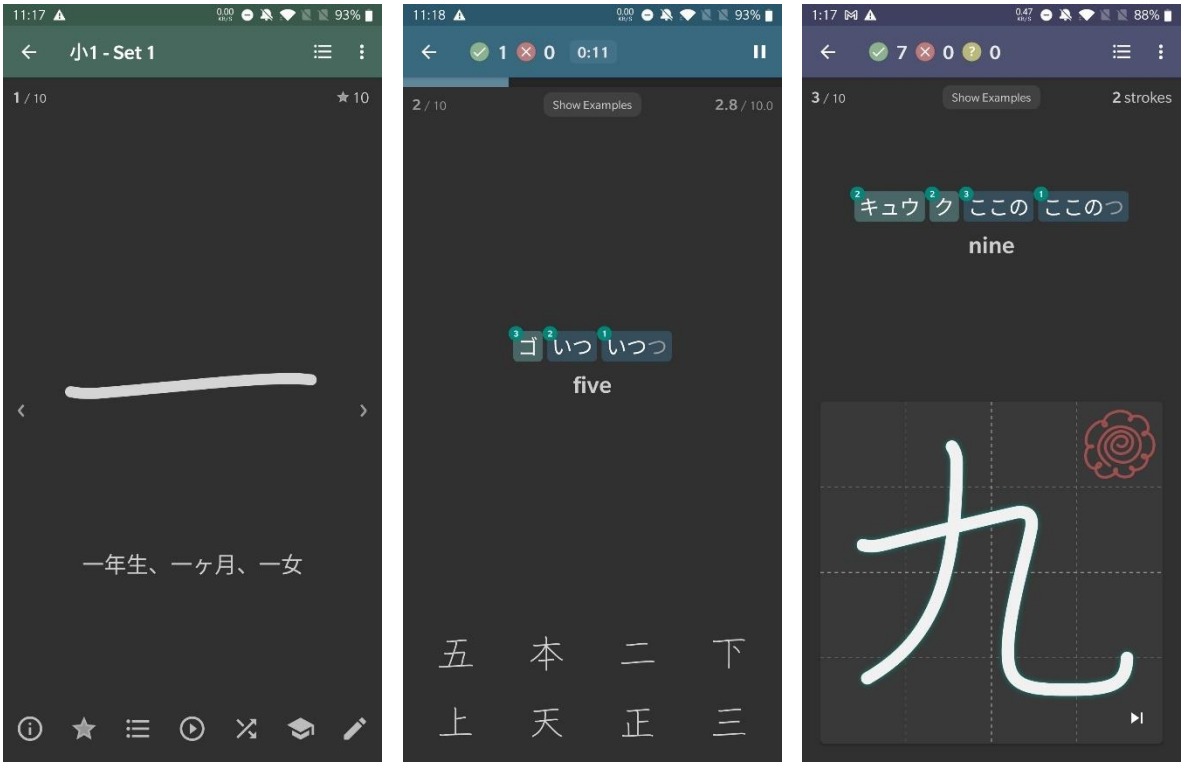

Figura 7. Distintos métodos de estudio en Kanji Study: a la izquierda, un método basado en tarjetas similar a Anki; en el centro, un método basado en preguntas de respuesta múltiple; a la derecha, un método basado en dibujar el *kanji*.

3.2.4. Duolingo

Duolingo es una aplicación gratuita (aunque cuenta con diversos pagos para obtener vidas y moneda interna de la aplicación) para estudiar idiomas, entre ellos el japonés, disponible en iOS, Android, MacOS y Windows. Además, cuenta con una versión web. En cada idioma cuenta con varias secciones en las que permite aprender diversas estructuras gramaticales, vocabulario y, en el caso de japonés, *kanji*. Estas lecciones se deben tomar en el orden marcado por la aplicación.

En cada sesión de estudio Duolingo muestra una selección de preguntas de diferentes conceptos que se revisan con el algoritmo de repetición espaciada de regresión de la vida media [10]. Estos ejercicios son bastante variados y, además, hacen uso de unos personajes muy carismáticos que animan el estudio. En Figura 8 se puede ver un ejemplo de los diversos tipos de preguntas que pueden aparecer en una sesión de estudio:

preguntas de traducción directa con respuesta múltiple, preguntas de transcripción de oraciones y preguntas de traducción inversa de oraciones.

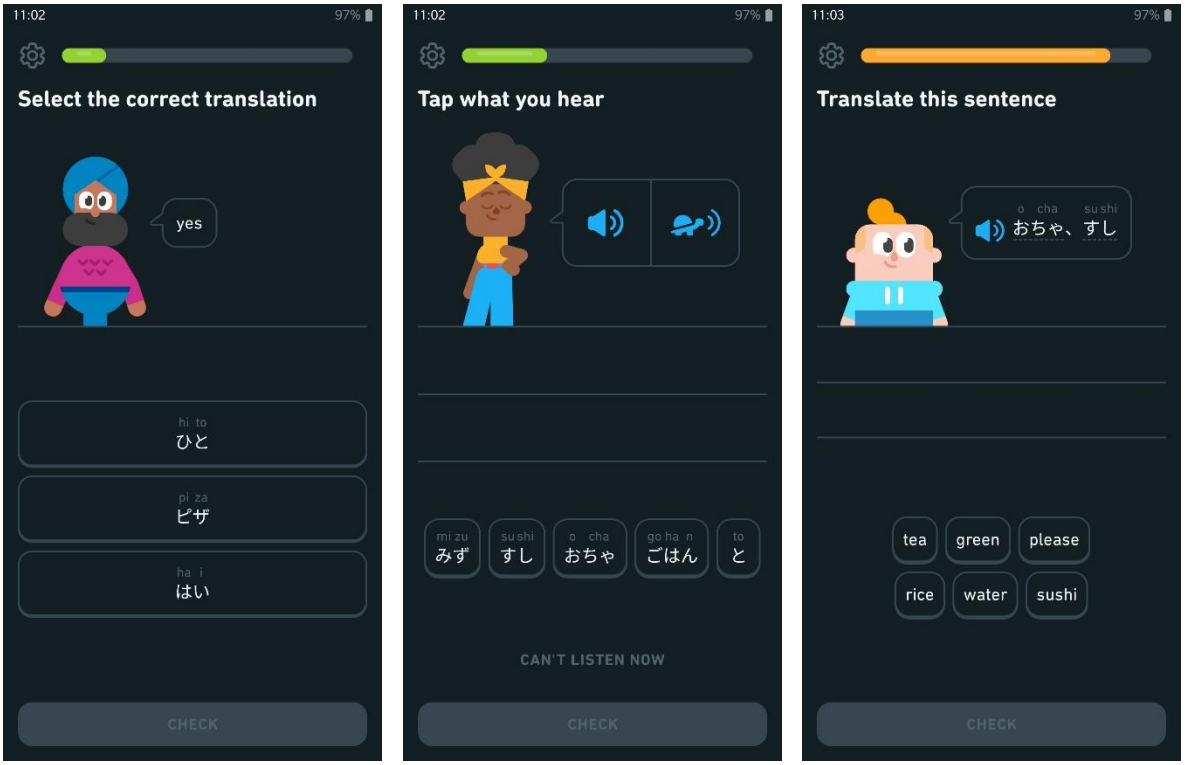

Figura 8. Tipos de preguntas de gramática y vocabulario en Duolingo: a la izquierda, una pregunta de respuesta múltiple; en el centro, una pregunta de transcripción de un audio; a la derecha, una pregunta de traducción.

El estudio de *kanji* se desbloquea conforme se avanza en las lecciones de gramática y vocabulario. De hecho, típicamente se conoce alguna palabra en la que aparezca antes de desbloquear la posibilidad de estudiarlo. Dentro de las preguntas de estudio de *kanji* y *kana* también hay bastante variedad. En concreto, en la Figura 9 se muestran algunos ejemplos de ejercicios: traducción directa con respuesta múltiple, selección de la pronunciación con respuesta múltiple, emparejamiento de pares y dibujo de *kanji* (además, de este hay varios niveles de ayuda en función de la práctica que se tenga con un *kanji* en específico).

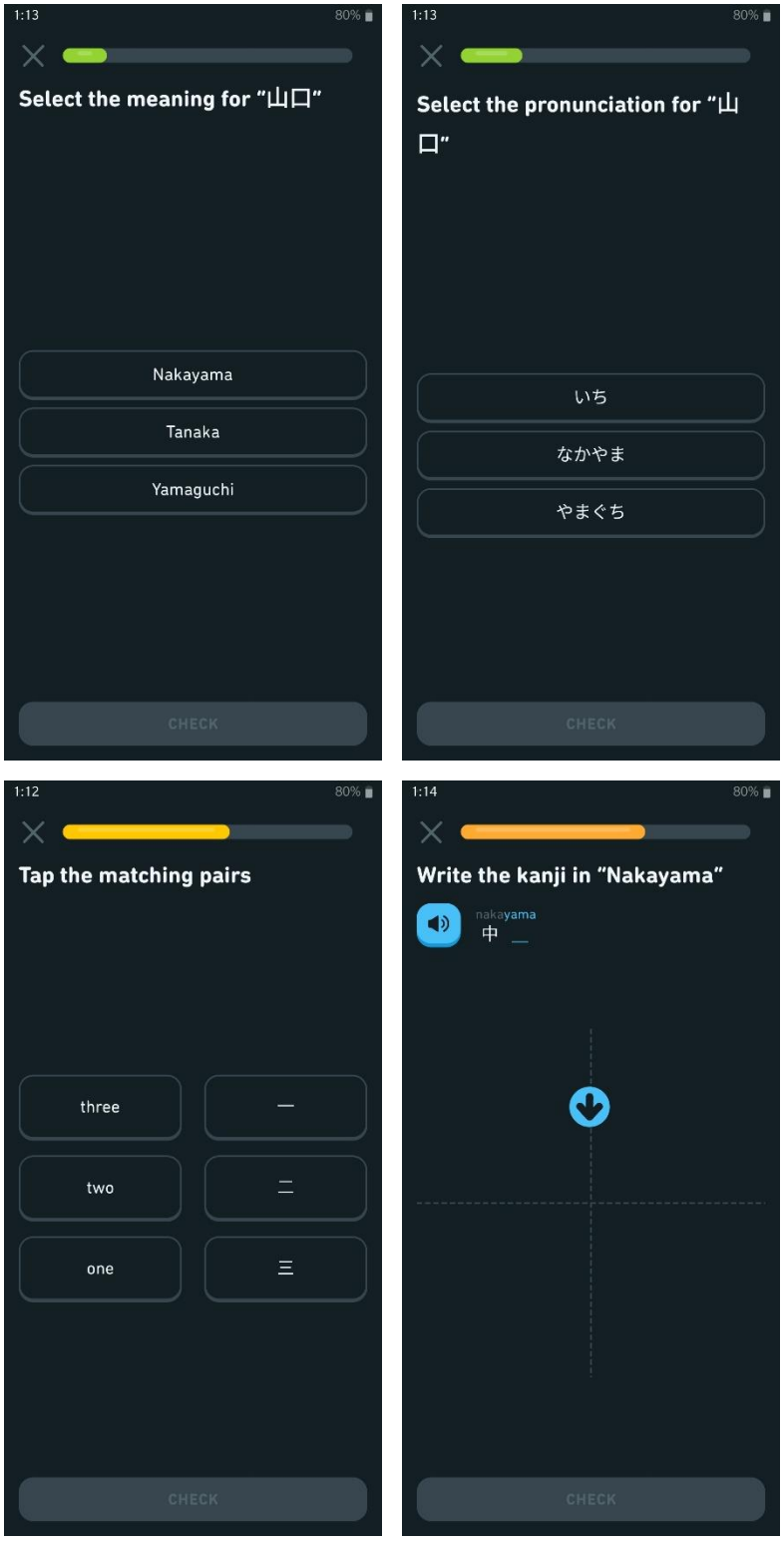

Figura 9. Ejemplos de ejercicios de *kanji* en Duolingo: arriba a la izquierda, un ejercicio de traducción directa con respuesta múltiple; arriba a la derecha, un ejercicio de selección de la pronunciación con respuesta múltiple; abajo a la izquierda, un ejercicio de emparejamiento de pares; y, abajo a la derecha, un ejercicio de dibujo de *kanji*.

Más allá de las lecciones, Duolingo realiza un muy buen trabajo en la gamificación de la herramienta y la formación de hábitos. Al terminar las sesiones de estudio ganas experiencia que te hace subir de nivel y obtener logros. También, se obtienen recompensas cuando se consiguen rachas de días en los que se ha estudiado. E, incluso es posible apostar una cantidad de puntos que se multiplicarán si se consigue una racha.

Como conclusión, Duolingo realiza un trabajo excelente haciendo la experiencia de estudio más amena gracias a la variedad en ejercicios, gamificación y carisma de sus personajes. Además, utiliza un algoritmo de repetición espaciada bastante competitivo. Sin embargo, debido a que su plan de estudio es muy rígido y no es personalizable para adaptarlo a un temario externo, no es adecuada como ayuda al estudio cuando se está siguiendo un curso del idioma. También, carece de explicaciones detalladas similares a las disponibles en Kanji Study.

3.3. *Embedding* de palabras e imágenes

Un *embedding* es una representación en forma de vector n-dimensional de un elemento de un dominio específico (*e.g.*, una palabra o una imagen). Para que sean consideradas *embeddings*, estas representaciones deben capturar las propiedades de los elementos y las relaciones entre ellos.

Los *embeddings* funcionan como un mecanismo para transferir conocimiento entre modelos gracias a que incorporan de manera implícita información sobre el dominio del problema (propiedades y relaciones entre los datos). También, simplifican la arquitectura de los modelos y mejorar su eficiencia ya que son un mecanismo de reducción de la dimensionalidad. Por ejemplo, la alternativa típica en el caso de palabras es utilizar un vector con codificación *one-hot* que es una representación mucho más larga (número de palabras en el vocabulario frente a una constante) y, por tanto, mucho menos eficiente.

En el caso de *embeddings* de palabras se pueden distinguir dos tipos: los estáticos y los dinámicos o contextualizados. Los *embeddings* estáticos son siempre los mismos, independientemente del contexto en el que aparezca cada palabra. Algunos ejemplos de modelos para computar *embeddings* estáticos son Word2Vec [13], GloVe [14] y FastText

[15]. Los *embeddings* dinámicos varían en función de la posición dentro de la oración y el sentido de la palabra en esa situación específica. Dos de las técnicas más conocidas para calcular esta clase de representaciones son ELMo [16] y BERT [17].

Al desarrollar modelos de visión por ordenador es muy común que estos no se diseñen desde cero y en su lugar utilicen una modificación de una serie de modelos conocidos que muestran un buen rendimiento como extractores de características, e.g., VGG [18], ResNet [19], Inception [20] o EfficientNet [21]. A estos troncales se les incorporan diversos módulos que transforman los mapas de características en otro tipo de información para una tarea específica como, por ejemplo, detección de objetos [22]. El vector de características generado puede ser utilizado como *embedding*. También, es bastante común utilizar la salida del codificador de un modelo codificador-decodificador como los autocodificadores [23] o los autocodificadores variacionales [24].

3.3.1. Word2Vec

Word2Vec surge como respuesta a la aparición de conjuntos de datos con cada vez más palabras y la incapacidad de las alternativas existentes en el momento para procesar tal magnitud de datos. Su premisa es simple, proponer una alternativa a los métodos del estado del arte que permita procesar una gran cantidad de datos de manera eficiente, pero, al mismo tiempo, consiguiendo *embeddings* de calidad.

Para conseguir este objetivo, se plantean dos arquitecturas: el modelo de bolsa de palabras continua (CBOW) y el modelo de *skip-grams* continuos. Ambos se basan en la hipótesis de que las palabras de una oración tienen una relación semántica con las palabras de su contexto (que se definirá como una ventana de n palabras a su izquierda y n a su derecha). Hay que tener en cuenta que en ninguna de las arquitecturas se incorpora la noción de posición de las palabras.

El modelo CBOW se basa en predecir una palabra en función de las de su contexto (es decir, un problema de clasificación). Para ello, proponen una red con una capa de proyección (que generará los *embeddings*), una capa lineal compartida por todas las palabras del contexto y una capa de agregación utilizando la media que proporcionará el resultado final.

El modelo de *skip-grams* es la operación opuesta: dada una palabra predecir su contexto (de nuevo, otro problema de clasificación). Sin embargo, la composición de la arquitectura de la red es bastante similar: una capa de proyección y una capa lineal que generará las probabilidades de que las palabras del vocabulario pertenezcan al contexto de la palabra actual. En general, esta alternativa suele obtener mejores resultados que CBOW.

Esta propuesta tiene diversas desventajas como la dificultad para trabajar con *tokens* desconocidos, la necesidad de un conjunto de datos de gran tamaño para converger exitosamente y que se limita a un contexto local para inferir el *embedding* de la palabra. Pero, también, cuenta con una gran cantidad de ventajas como su simplicidad, eficiencia y el rendimiento capturando las relaciones entre palabras.

3.3.2. GloVe

Word2Vec utiliza únicamente un contexto local e infiere el significado de las palabras en base a ello. Sin embargo, las estadísticas globales de una palabra sobre todo el corpus pueden dictar también la influencia en su significado. Por ejemplo, un mayor número de coocurrencias en una palabra muy frecuente podría ser menos significativo que un menor número de ocurrencias en una palabra menos frecuente.

GloVe sostiene que el significado de una palabra no depende tanto de la probabilidad de coocurrencias entre palabras (hipótesis de Word2Vec), sino del denominado ratio de coocurrencia. Este ratio se define con la Ecuación 7, donde $P(j|i)$ es la probabilidad de que la palabra j aparezca en el contexto de la palabra i o, en otras palabras, el cociente entre las apariciones de j en el contexto de i (X_{ij}) y el total de palabras en el contexto de i (X_i). Si la ratio es mayor que 1, w_i estará más relacionada con $\widetilde{w}_k$ (una palabra del contexto) que w_j y, si es menor, ocurriría el caso contrario.

$$F(w_i, w_j, \widetilde{w}_k) = \frac{P(k|i)}{P(k|j)}$$

Ecuación 7. Ratio de coocurrencia.

Tras realizar diversas asunciones (linealidad en los *embeddings*, conmutatividad en la matriz de coocurrencias) y transformaciones a partir de la Ecuación 7 se infiere la Ecuación 8 (ver [14] para conocer este proceso en detalle) para la relación entre los *embeddings* de una palabra i, w_i, y una palabra de su contexto k, $\widetilde{w}_k$.

$$w_i^T \widetilde{w}_k = \log(X_{ik}) - \log(X_i)$$

Ecuación 8. Relación entre el *embedding* de una palabra i y una palabra de su contexto k.

Por tanto, encontrar los valores de los *embeddings* se puede traducir en un problema de optimización que minimice la expresión de pérdida dada por la Ecuación 9 sobre un vocabulario, V. El factor $f(X_{ik})$ limita el peso de palabras muy frecuentes en esta función de pérdida.

$$J = \sum_{i,k=1}^{V} f(X_{ik}) \left(w_i^T \widetilde{w}_k + b_i + \tilde{b}_k - \log(X_{ik}) \right)^2$$

Ecuación 9. Función de pérdida de GloVe.

En definitiva, GloVe mejora sobre Word2Vec incluyendo la noción de contexto global a costa de parte de su simplicidad. Por tanto, comparte la mayoría de sus ventajas y desventajas.

3.3.3. Otros métodos de *embedding* de palabras

FastText mejora el problema del procesamiento de palabras desconocidas en tiempo de inferencia que presenta Word2Vec. Para ello, introduce el contexto de subpalabras: en lugar de asociar *embeddings* únicamente a las palabras se asocian también a n-gramas de estas. Además, se añade la restricción de que el *embedding* de una palabra debe coincidir con la suma de los *embeddings* de todos sus n-gramas. Gracias a esto, es posible inferir

embeddings para palabras que no hayan aparecido en tiempo de entrenamiento y generar *embeddings* más precisos para palabras con poca frecuencia en el corpus.

ELMo introduce el concepto de *embeddings* contextualizados. Estos *embeddings* ya no representan a una palabra independientemente de la situación en la que aparezca, sino que dependerán de la posición y oración en la que esta aparezca. De esta forma, las palabras polisémicas, por ejemplo, rosa que puede ser una flor, una persona o un color, podrán tener *embeddings* muy diferentes en función del significado específico en el que se encuentran. Para conseguir esto se valen de una arquitectura basada en capas *long short-term memory* bidireccionales (BiLSTM) que permiten tener en cuenta el contexto de toda la oración en el proceso de generación.

BERT continúa con el concepto de *embeddings* contextualizados y subpalabras, pero, en este caso, utilizando una arquitectura basada en *transformers*. Este cambio permite alcanzar una mejor precisión en muchas tareas debido a la mejora en las capacidades para capturar relaciones entre *tokens* dentro de la frase. Sin embargo, se incrementa el coste de ejecución y requiere una mayor cantidad de datos.

3.3.4. Arquitecturas codificador-decodificador y autocodificadores

Las arquitecturas codificador-decodificador se basan en generar redes neuronales artificiales compuestas por dos módulos: el codificador y el decodificador. Por un lado, el módulo codificador transforma los datos de entrada desde su representación en un espacio de entrada hacia una representación en un espacio latente de menor dimensión (similar a un *embedding*). Por otro lado, el módulo decodificador transforma la representación sobre el espacio latente a una representación en el espacio de salida.

En el caso de los autocodificadores los espacios de entrada y salida son el mismo, de hecho, el resultado del decodificador debería ser una reconstrucción de la entrada del codificador. Este proceso codificación-decodificación se trata de una reducción dimensional en la que se pretende extraer las características más importantes que definen a un ejemplar en el espacio de entrada y codificarlas sobre el espacio latente para que el decodificador tenga información suficiente a la hora de realizar la reconstrucción. De esta manera, los vectores de este espacio latente representan componentes semánticos de los vectores del espacio

de entrada y entradas con características similares generarán vectores latentes similares (utilizando, *e.g.*, la similitud del coseno). Además, los autocodificadores también pueden utilizarse para generar nuevos elementos del espacio de salida muestreando vectores del espacio latente.

La distribución de los *embeddings* de las entradas sobre el espacio latente generado por un autocodificador es discontinua (puede haber grandes espacios vacíos entre grupos de ejemplares) y genera grupos con distribuciones muy dispares. Estas desventajas son, en general, más problemáticas cuando se usan como un modelo generativo dado que provoca resultados muy similares entre sí y de poca calidad.

Los autocodificadores variacionales pretenden precisamente resolver este problema. Para ello, en lugar de generar directamente *embeddings*, se producen distribuciones que serán muestreadas por el decodificador para reconstruir las imágenes de entrada. Al introducir un muestreo de una distribución estadística, el modelo deja de ser determinista y pasa a ser estocástico. Dado que los componentes estocásticos no son diferenciables es necesario introducir un valor muestreado de una distribución normal estándar como entrada y transformarlo acordemente a la distribución generada por el codificador, esto es conocido como el truco de la reparametrización.

4. HERRAMIENTAS, TÉCNICAS Y METODOLOGÍAS

En este apartado se detallarán los métodos y medios utilizados durante el desarrollo de este trabajo. En concreto, en el apartado 4.1 se describe la metodología de desarrollo utilizada. En el resto de los apartados se mencionarán los medios utilizados, específicamente, las herramientas, que se dividirán en herramientas de gestión de proyectos, análisis y diseño (sección 4.2); herramientas de desarrollo (sección 4.3); y herramientas de documentación, pruebas y calidad (sección 4.4).

4.1. Metodología de desarrollo

A lo largo del desarrollo de este trabajo se ha realizado un esfuerzo consciente en asegurar que la metodología de desarrollo estuviese supeditada a las necesidades del equipo de desarrollo y optimizada para garantizar la mayor calidad posible en el software. Por ello, ha estado sujeta a escrutinio y mejora continuos. La única constante que ha existido a lo largo de todo el proyecto en lo que a metodología se refiere es la filosofía *Lean*.

Esta filosofía se centra en el proceso y su mejora de manera continuada con el objetivo de generar el mayor valor posible eliminando las actividades que no contribuyen a este. Para ello, esta filosofía propone los siguientes principios [25]:

- Eliminar el desperdicio: el desperdicio son todas aquellas actividades que no contribuyen directamente a generar valor en el proyecto. Por ejemplo, implementar funcionalidades que no se esté usando, esperar o resolución de defectos[18]. Se debería aspirar a identificar esta clase de actividades y, en lo posible, eliminarlas. Hay actividades que pueden no generar valor pero que deben realizarse, por ejemplo, por motivos legislativos.
- Amplificar el aprendizaje: el desarrollo software es una disciplina que requiere un aprendizaje continuo. Por ello, el proceso de desarrollo debe orientarse a generar la mayor cantidad de información posible cuanto antes. Para conseguirlo se fomenta una mentalidad de experimentación para encontrar la mejor solución y la retroalimentación rápida tanto del cliente como de los procesos de prueba.

[18] La resolución de defectos no genera valor porque se está trabajando en funcionalidades ya existentes que no funcionan como deberían. En su lugar, se deberían invertir recursos en evitar que estos defectos se detecten lo antes posible y se produzcan lo menos posible.

- Decidir lo más tarde posible: cuanto más tarde se tome una decisión más información habrá disponible en ese momento y, por tanto, menor será la incertidumbre asociada a una tarea. Se toma una aproximación basada en opciones y no en compromisos para evitar crear expectativas poco realistas.
- Entregar lo antes posible: el producto no tiene valor hasta que no llega a las manos del cliente. Se deben buscar todas las dificultades y factores que bloquean el avance de las características hasta el usuario. Algunos ejemplos típicos de estos bloqueos son la sobreplanificación y las soluciones que tienen una complejidad mayor a la que el problema necesita.
- Potenciar al equipo: se debe confiar en la capacidad técnica del equipo y fomentar un entorno de trabajo sostenible.
- Basado en la integridad: el producto debe satisfacer las necesidades del usuario y todo el proceso debe estar supeditado a ello. La calidad del código también es un factor que se debe cuidar y, por tanto, las pruebas y la refactorización son herramientas recomendadas.
- Ver el todo: la optimización no debe centrarse en sistemas aislados. Se deben tener en cuenta todas las entidades y actividades que participan en el proceso a la hora de implementar mejoras.

En este trabajo se han implementado estos principios de la siguiente manera:

Al principio, se planteó utilizar una metodología Scrum con una alta cobertura (90-100%) de sentencias a base de pruebas unitarias, pero rápidamente fue evidente que esta metodología no se adecuaba ni al proyecto ni al desarrollador. En concreto se identificaron los siguientes problemas:

- La disponibilidad del desarrollador es muy irregular y poco predecible a lo largo del tiempo debido a la existencia de otros compromisos externos al proyecto. Ha habido semanas en las que el tiempo que se le ha podido dedicar ha sido mínimo y semanas en las que se le ha dedicado más de 8 horas diarias. Planificar cualquier tipo de sprint en estas circunstancias y que se acerque a la realidad es una tarea imposible.
- Algunas partes del desarrollo tienen un alto grado de incertidumbre debido a, primero, que el proyecto cuenta con bastantes procesos experimentales donde los pasos futuros dependen de unos resultados que son desconocidos en un primer

momento y, segundo, la incorporación de herramientas desconocidas para el desarrollador (e.g., pgVector, Parsimmon, Kuromoji, Sequelize o Nodemailer). En esta situación, cualquier tipo de estimación del coste de una tarea es arbitrario lo cual, de nuevo, hace que comprometer tareas a *sprints* sea una tarea de azar.

- El estrés del desarrollador fue alto durante todo este tiempo debido a retrasos constantes en las tareas comprometidas a los *sprints*.
- La planificación de *sprints* y gestión de tareas (redacción y priorización) llevaba demasiado tiempo porque el desarrollador debía hacer también el papel de producto owner y scrum master.
- El desarrollo de pruebas llevaba demasiado tiempo. El desarrollador invertía más tiempo en implementar pruebas que características. Además, incrementó la deuda técnica dado que se generaban soluciones más complejas de lo necesario sólo para facilitar las pruebas.

Llegados a este punto, se introdujo la metodología de mejora de procesos Kanban. Este método se basa en una visualización, el tablero kanban, que ilustra el proceso que sigue una historia de usuario desde que es identificada hasta que es publicada. Este tablero tiene diversas columnas y cada una ellas está asociada a una actividad del proceso de desarrollo. Las historias comenzarían en una cola de entrada y avanzarían columna a columna hasta ser publicadas. En base a esta visualización se podrán identificar diversos problemas en el proceso que generan desperdicios y pueden ser mejorados (por ejemplo, muchas tareas en una columna indicarían un cuello de botella).

Desde entonces se introdujeron los siguientes cambios:

- Para reducir los tiempos de gestión de tareas se pasó a planificar por historias en lugar de por tareas. Las historias se describen a un nivel de detalle suficiente como para mantener el alcance definido, pero sin llegar a ser tan exhaustivas que limiten posibilidades durante el desarrollo y tomen demasiado tiempo de redacción. Las tareas se identifican en el momento de implementar una historia y no antes.
- Para reducir el estrés provocado por los errores de planificación en los *sprints* se ha decidido virar hacia una planificación basada en opciones en lugar de compromisos y pasar de un sistema *push* a un sistema *pull*. Esto es, en vez de comprometer tareas a un sprint determinado, se cuenta con una pila de *backlog* (que está siempre

priorizada) con opciones de historias en las que se puede trabajar. El desarrollador coge historias de esta pila cuando tenga disponibilidad para ello en lugar de recibir tareas periódicamente (*e.g.*, en cada *sprint*) en base a unas necesidades de planificación previas a conocer la disponibilidad de recursos o el coste real de las tareas.

- Para reducir los tiempos invertidos en el desarrollo de pruebas unitarias, se optó por sustituirlas por pruebas de extremo a extremo. Este cambio tiene la desventaja de que las pruebas son más superficiales y pueden no detectar tantos errores. En el futuro sería interesante comprobar si es viable añadir pruebas unitarias sólo para ciertos componentes en lugar de buscar una cobertura de sentencias completa.

Aunque introducir estos cambios mejoró la productividad del desarrollador, aún hay aspectos en los que es posible mejorar y se trabajará en ello en el futuro.

4.2. Herramientas de gestión de proyectos, análisis y diseño

El sistema elegido para la gestión de versiones del software ha sido GIT. Con él se ha podido garantizar la existencia de una versión siempre funcional en la rama “develop” mientras se trabajaba en versiones de características en ramas que partían de esta. Además, ofrece mecanismos para trazar los cambios y asegurar que es posible volver a versiones anteriores en caso de que surja algún problema.

En todo momento se ha mantenido una copia remota de este repositorio en GitHub. Esta herramienta además de como *backup* se ha utilizado como sistema de planificación de proyectos. En concreto, se ha utilizado GitHub Issues para documentar las características del software y GitHub Projects para visualizar un tablero Kanban con estas.

La documentación en UML del software se ha generado utilizando PlantUML, un DSL para la generación de diagramas de ingeniería del software. Utilizar esta herramienta frente a alternativas como Visual Paradigm o Start UML aporta una mayor agilidad y velocidad al no necesitar preocuparse por la disposición de los elementos y la facilidad para modificar los diagramas.

4.3. Herramientas de desarrollo

El editor de código elegido para el desarrollo ha sido Visual Studio Code dado que es una herramienta gratuita, con una gran comunidad y una amplia extensibilidad. Además, está desarrollado por una de las compañías con mayor influencia en la industria, Microsoft.

Tanto el cliente web como la API han sido desarrollados utilizando el lenguaje de programación TypeScript. Este lenguaje añade una capa sintáctica adicional sobre JavaScript que permite introducir información de tipado. Es muy configurable y permite regular con bastante libertad lo estricto que es su sistema de tipos. Tanto es así, que el código JavaScript válido es código TypeScript válido. Un sistema de tipado fuerte ayuda a garantizar una mayor calidad del código desde el momento de la compilación sin demasiadas desventajas y, por tanto, se ha usado una configuración muy estricta de TypeScript. Por ejemplo, se han deshabilitado los tipos implícitos en la mayoría de los contextos o el acceso a índices sin comprobación.

En el caso del servidor se ha utilizado Node.js, un motor de ejecución de código JavaScript construido sobre V8 (que es el motor de ejecución de código utilizado en Chrome) que permite ejecutar JavaScript fuera del navegador. Recientemente han surgido un par de alternativas a Node.js que han cogido bastante tracción entre la comunidad. Dos ejemplos de esto son Bun y Deno. Aunque ambas son tecnologías prometedoras con mejoras en diversos frentes (rendimiento y seguridad), se ha decidido utilizar Node.js porque es un software más maduro y con una mayor adopción en la industria, lo que significa una comunidad mayor y más facilidad para encontrar ayuda en caso de necesitarla. El cliente web también utilizará Node.js pero no como motor de ejecución propio sino como una dependencia de alguna de las herramientas de desarrollo. Como gestor de paquetes en ambos se ha utilizado NPM, que es el estándar *de facto* dentro de la comunidad de Node.js (aunque existen buenas alternativas, *e.g.*, yarn).

La arquitectura del servidor está dictada en gran parte por Nest.js, un *framework* para el desarrollo de APIs web en Node.js. Fomenta prácticas como la inyección e inversión de dependencias y la organización de la aplicación en diversos módulos de alta cohesión y, presumiblemente, bajo acoplamiento entre sí. Se eligió esta herramienta con la expectativa de generar una base de código más reutilizable, cohesiva y probable. Sin embargo, de haber

utilizado Express o Fastify el proceso de desarrollo habría sido más corto y la base de código más sencilla.

Nest.js cuenta con diversas integraciones que adaptan bibliotecas a su filosofía. Una de las más adoptadas es la integración con class-transformer y class-validator, dos bibliotecas que permiten transformar objetos planos a instancias de clases y validarlas por medio de decoradores en la definición de la clase. Estas bibliotecas son muy prácticas y elegantes.

También tiene una integración con Passport. Passport es una biblioteca que proporciona una forma probada y relativamente sencilla de implementar autenticación en APIs. Cuenta con diversas estrategias de autenticación, *e.g.*, *token*s JWT, correo electrónico con contraseña o OAuth. En este trabajo se han utilizado varias estrategias de validación por *token* y correo electrónico y contraseña, y esta biblioteca ha ayudado enormemente a simplificar el código relacionado con la autenticación. Durante el registro de cuentas la contraseña se encripta utilizando la biblioteca Bcrypt y, también, se utiliza Nodemailer para la gestión de envíos de correo electrónico que, durante el desarrollo, son enviados a un servidor de pruebas de Smtp4dev que permite inspeccionar todos los correos electrónicos enviados.

Otra de las integraciones disponibles es con Sequelize. Sequelize es una de las bibliotecas de mapeo relacional de objetos (ORM) con más trayectoria en el ecosistema JavaScript. Tiene compatibilidad con múltiples bases de datos, entre las que se encuentra la utilizada en esta aplicación, PostgreSQL y, también, para el plugin de manejo de vectores multidimensionales pgVector. Esto último y su robustez fueron las razones principales para su elección. Sin embargo, la compatibilidad con pgVector terminó siendo muy frágil y tuvo que gestionarse manualmente.

Como se ha anticipado se ha utilizado PostgreSQL como base de datos. Esta sigue el paradigma de base de datos relacional y cuenta con una gran extensibilidad por medio de *plugins*. En este trabajo ha sido de especial relevancia la disponibilidad de pgVector que permite tratar los vectores de *embeddings* (hacer consultas sobre las distancias e, incluso, generar índices). También permite incluir búsqueda de texto completo, muy útil para los campos de definiciones de palabras y *kanji*. Otras alternativas no ofrecen tanta

funcionalidad. Además, en base de datos se guardarán los parámetros del algoritmo de repetición espaciada que se implementa por medio de fsrs.js.

Siguiendo con la búsqueda, se ha utilizado la biblioteca Kuromoji para tokenizar y lematizar oraciones en japonés que se introduzcan en el buscador. Es una de las pocas bibliotecas que se han encontrado con esta funcionalidad disponibles para el JavaScript y en diversas pruebas realizadas pareció funcionar de manera similar a Mecab, un referente en el ámbito de la tokenización y lematización de la lengua japonesa.

También, se hizo necesario poder traducir entre los diversos mecanismos de transcripción fonética del japonés (*romaji*, *hiragana* y *katakana*), para lo que hubo que implementar un *parser* propio. Para ello se utilizó Parsimmon, una biblioteca que implementa el patrón de *parser combinators* para la generación de *parsers*. De entre todas las alternativas, es de las más robustas, con mejor mantenimiento y una interfaz más agradable.

Cambiando de ámbito, el cliente web se ha desarrollado haciendo uso de la biblioteca React. Esta biblioteca fomenta un desarrollo basado en componentes dentro de un paradigma basado en la programación reactiva. Los componentes son una función de su estado y parámetros (*props* en la jerga de React) y se actualizan automáticamente cuando estos cambian. Existen diversas alternativas a React como Angular o Vue, pero todas ellas son bastante robustas, con una comunidad muy grande y tienen unas ventajas y desventajas similares, así que la decisión depende de la experiencia y preferencias del desarrollador.

A diferencia de Angular, React es mucho menos opinionado y deja en manos del desarrollador muchas más decisiones de diseño y la elección de cómo gestionar todo lo que no sea el renderizado de componentes. Por ello, se ha hecho necesario el uso de React Router para gestionar el enrutado de la aplicación y de TanStack Query (anterior React Query) junto con Axios para gestionar las peticiones al servidor. Estas tres son estándares *de facto* en el ecosistema React. Se ha decidido no utilizar sistemas de gestión de estado adicionales (como Redux o Zustand) porque el estado del cliente no es muy complejo y esos sistemas sólo complicarían la base de código.

La aplicación cuenta con bastantes formularios y, por eso, se ha hecho uso de Formik, una biblioteca que permite manejar formularios con facilidad en React. Su funcionamiento es

similar a los formularios dirigidos por plantillas de Angular, pero, al mismo tiempo, es mucho más sencilla. Gracias a esta herramienta, se reduce enormemente la complejidad de las vistas con formularios y, por tanto, la cantidad de errores que surgen al implementarlas.

Además, dado que estilar desde cero una aplicación es una tarea compleja y que requiere de grandes habilidades en diseño, se ha hecho uso de la biblioteca de componentes Radix UI Themes con un tema personalizado. Aun así, algunos de los componentes han sido diseñados e implementados exclusivamente para esta aplicación. También se ha hecho uso de su biblioteca de iconos Radix UI Icons aunque, de nuevo, algunas ilustraciones e iconos son propios.

En la tercera parte del trabajo, los modelos de *embedding*, se ha hecho uso del lenguaje Python que, de nuevo, es un estándar *de facto* en el ámbito de la inteligencia artificial gracias a su simplicidad y velocidad de desarrollo. Además, se han utilizado cuadernos Jupyter para prototipado, análisis de datos y análisis de resultados dado que ayudan a alcanzar una rápida velocidad de iteración. Además, en estos cuadernos se ha utilizado Matplotlib para la visualización de información.

Dentro del ecosistema Python, se ha utilizado PyTorch para el desarrollo de los modelos de *embedding*. Adicionalmente, se ha hecho uso de torchvision en el modelo de *embeddings* visuales. Se ha decidido utilizar PyTorch en lugar de Tensorflow porque es mucho más flexible. Para generar y preprocesar los conjuntos de datos utilizados en estos modelos se ha utilizado PIL, una biblioteca de tratamiento de imágenes, y Pandas, una herramienta para el tratamiento de datos tabulares.

4.4. Herramientas de documentación, pruebas y calidad

Tanto en el cliente como en el servidor se ha utilizado ESLint que es el *linter* más usado en JavaScript. Un *linter* es una herramienta que analiza el código de una aplicación y notifica sobre diferentes aspectos que puedan perjudicar la calidad del código, en concreto, avisa de fallos en la uniformidad del código y de prácticas poco recomendables que puedan provocar errores. Se ha partido de las configuraciones más estrictas proporcionadas por TypeScript-ESlint y se han desactivado algunas opciones que resultaban inconvenientes o

entraban en conflicto con algún estilo de desarrollo de las bibliotecas o marcos de trabajo utilizados. Por ejemplo, se han permitido las clases vacías cuando tienen decoradores, ya que en general es la forma de crear los módulos de NestJS. También, se ha utilizado Prettier, que es un formateador de código, es decir, una herramienta que modifica los ficheros de código para que se adhieran a unas mismas reglas de estilo.

Las bases de código tanto del cliente como del servidor están plagadas de anotaciones de documentación siguiendo el formato JSDoc. Estas anotaciones son utilizadas por TypeDoc para generar una web de documentación. Además, en el caso del cliente web, se ha utilizado Storybook para generar una documentación más visual de los distintos componentes de la aplicación. Esta última herramienta es muy útil, además, para facilitar el desarrollo de los componentes de una manera aislada de la aplicación ya que cuenta con herramientas que permiten interactuar con las entradas (los *props*) que reciben y observar sus salidas (su representación visual y los eventos que lanzan).

Para las pruebas automáticas del servidor (el cliente se ha probado manualmente) se ha utilizado Jest. Este es un *framework* de pruebas desarrollado por Facebook y con una gran adopción. Tiene una interfaz que facilita mucho la escritura de pruebas y, al tener una base de usuarios tan grande, cuenta con muchas extensiones. Para facilitar la prueba de los endpoints se ha usado Supertest, una biblioteca que permite hacer aserciones sobre los resultados de peticiones HTTP.

Por último, mencionar GitHub Actions: un servicio para la ejecución de *pipelines* ante eventos en el repositorio de GitHub. En concreto, en este proyecto se ha utilizado para crear un pipeline de integración continua que compruebe que el proyecto se compile con éxito, el *linter* no lance ningún error y, además, pase todas las pruebas del servidor.

5. ASPECTOS RELEVANTES DEL DESARROLLO

En este apartado se describen características de interés sobre distintas áreas del proyecto. En primer lugar, en el apartado 5.1 se habla de cómo esta filosofía *Lean* y la metodología Kanban han afectado al ciclo de vida de las historias durante el desarrollo. Tras esto, se comentan los resultados de las fases de requisitos (apartado 5.2) y análisis (apartado 5.3). Después, en la sección 5.4 se describe como ha sido el proceso de diseño y se detallan algunos patrones arquitectónicos y de diseño necesarios para poder comprender el funcionamiento del proyecto. A continuación, se habla de los mayores retos de la implementación en el subapartado 5.5. Por último, en la sección 5.6 se describe la metodología de pruebas utilizada.

5.1. Ciclo de vida

Este proyecto se ha encuadrado utilizando un ciclo de vida de mejora continua en el que nuevas historias de usuario pueden ser añadidas en cualquier momento a una pila de *backlog* priorizada. Esta prioridad se gestiona en base al orden de las tareas en la pila. Cuando existen recursos suficientes (cada trabajador puede estar trabajando en un instante de tiempo en una única tarea) este tomaría la primera historia de la pila y comenzaría a desarrollarla hasta completarla. Se considera que una historia se ha completado cuando el usuario es capaz de realizar el caso de uso descrito y se cumple con los criterios de calidad (*i.e.*, pasa el pipeline de integración continua y no se detecta ningún fallo) establecidos. Puede haber casos en los que debido a que se descubre una dependencia inesperada o a que la prioridad de esta cambie (porque se encuentra información adicional), la historia se devuelva al backlog para ser continuada en otro momento más adecuado o se descarte completamente.

Las historias se desarrollan siguiendo un proceso iterativo. Primero, se diseña una prueba que represente el resultado esperado de alguna de las circunstancias que pueden darse en la historia. Esta prueba debería fallar inicialmente, porque sino significaría que el sistema ya cumple con ese requisito. Y, a continuación, se analizan, diseñan e implementan los cambios necesarios para pasar esa prueba. Este proceso de dos fases se repite indefinidamente hasta que la historia puede completarse con éxito en cualquier circunstancia posible. En este punto, se refactoriza la implementación para mejorar su

calidad lo máximo posible. Adicionalmente, se ha hecho una refactorización al inicio del proceso cuando el sistema debía ser adaptado para asimilar la nueva característica.

Este proceso de desarrollo iterativo de una historia se desarrolla sobre una rama de característica paralela a la rama base del repositorio de software. Las ramas de características no se podrán unificar con la rama base hasta que la historia esté completada (*i.e.*, el flujo de la historia se puede completar y pasa la integración continua). Esto permite generar mejoras incrementales sobre una versión base que es siempre funcional.

5.2. Especificación de requisitos

Los requisitos de la aplicación se han identificado a lo largo de todo el proyecto y durante todo ese tiempo se ha tenido una mente abierta a nuevas características. Los requisitos se han especificado utilizando historias de usuario, por ejemplo, “Como un estudiante de japonés, me gustaría poder formar colecciones con el temario de mi curso para poder estudiar los *kanjis* y palabras necesarios para aprobarlo.” Para consultar una lista detalla de historias de usuario se puede consultar el anexo 8.3. Adicionalmente, se adjuntan algunos diagramas de casos de uso que documenten estos requisitos.

5.3. Análisis

El análisis de estas historias se ha realizado y refinado incrementalmente durante la implementación y no como una fase previa a esta. En la Figura 10 se incluye el modelo del dominio, que representa las diferentes entidades involucradas en el problema y sus relaciones. Además, en el anexo 8.4 se documentan los paquetes de la fase de análisis con mayor detalle.

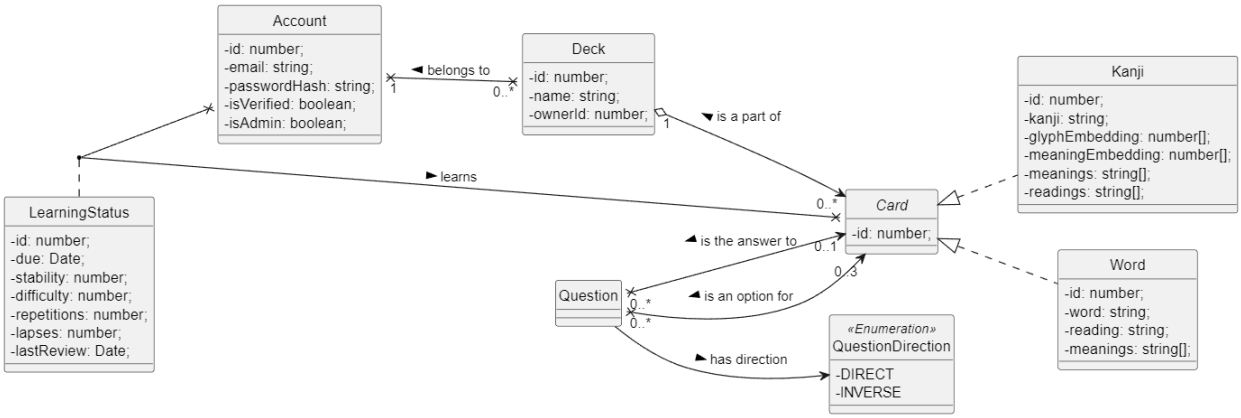

Figura 10. Modelo de dominio.

5.4. Diseño

Al igual que el análisis, el diseño de las tareas se ha realizado en el momento de la implementación. En este apartado se detallan los patrones más importantes para entender la arquitectura de la aplicación.

5.4.1. Modelo-vista-controlador

El patrón modelo-vista-controlador favorece una segregación de responsabilidades entre diferentes componentes de la aplicación: la vista es responsable de mostrar la información al usuario, el modelo es responsable de almacenar la información y el controlador actúa como un intermediario que los desacopla. En la Figura 11 se ve un ejemplo de uso de este patrón.

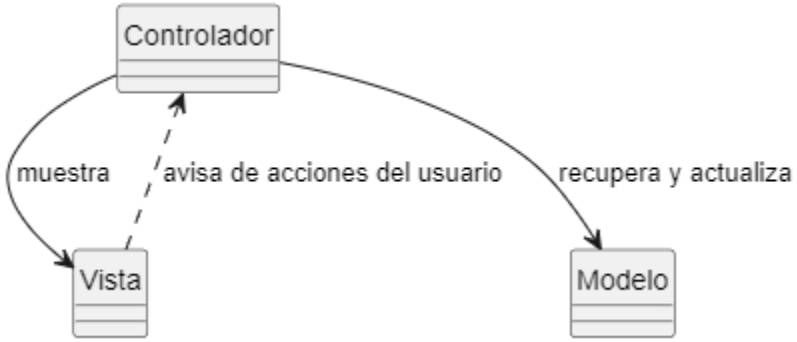

Figura 11. Patrón model-vista-controlador

En este trabajo se ha hecho uso de este patrón en el cliente de la aplicación. El controlador se encarga de recuperar los modelos del servidor, instanciar las vistas con ellos, generar las modificaciones causadas por acciones del usuario y actualizar los modelos en el servidor.

5.4.2. Capas

En esta aplicación se ha hecho uso de la arquitectura por capas para el servidor. Esta arquitectura también se basa en el principio de segregación de responsabilidades, ya que cada capa tiene asignada una función. La versión específica de la arquitectura por capas utilizada en esta aplicación se puede consultar en la Figura 12 y consta de cuatro capas: la capa de presentación, la capa de aplicación, la capa de dominio y la capa de infraestructura.

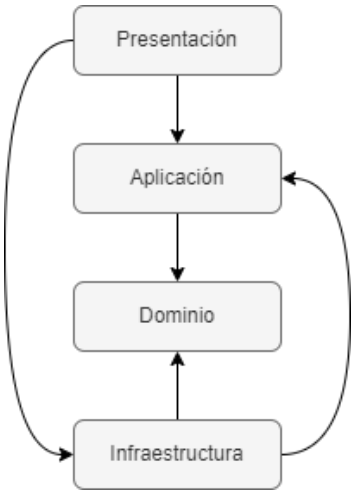

Figura 12. Arquitectura por capas.

La capa de dominio es la responsable de implementar la lógica de negocio y es la responsable de garantizar que el modelo de la aplicación este en un estado consistente en todo momento. Sus dependencias con el exterior (por ejemplo, sistema en el que se ejecuta, otras capas de la aplicación y librerías externas) se deberían mantener a un mínimo. Las entidades del modelo pertenecen a esta capa. Por practicidad, se utilizarán decoradores del ORM para anotar las entidades, aunque eso sea introducir una dependencia.

La capa de aplicación coordina el trabajo entre diferentes servicios de la capa de dominio o la propia capa de aplicación. Si la capa de dominio estaba dirigida por el modelo, la capa de aplicación está dirigida por los casos de uso. En esta capa se encuentran la mayoría de los servicios de la aplicación y sus métodos son bastante similares a los distintos casos de uso.

La capa de presentación es la implementación de la interfaz que publica el sistema para que otros sistemas interactúen con él. Los componentes de esta capa son intermediarios entre el mundo exterior y la capa de aplicación. Los controladores HTTP forman parte de esta capa.

La capa de infraestructura implementa adaptadores entre el sistema y otros sistemas externos (*e.g.*, la base de datos) de los que depende. Sirve para desacoplar el núcleo (capas de dominio y aplicación) de sistemas externos. Los repositorios son parte de esta capa.

5.4.3. Inyección de dependencias

El patrón de inyección de dependencias se basa en liberar al cliente de un servicio de la responsabilidad de creación de este y cederla a un inyector, una clase responsable de crear el servicio y proporcionárselo al cliente tal y como se muestra en la Figura 13. Este cambio reduce el acoplamiento entre componentes lo que, al mismo tiempo, tiene consecuencias positivas en la facilidad de prueba y la reusabilidad de los diferentes componentes. Sin embargo, también aumenta la complejidad del código y hace que las interacciones entre componentes puedan ser menos evidentes. En esta aplicación se usa inyección de dependencias en el servidor con bastante frecuencia debido a estas ventajas.

Existen diversos tipos de inyección de dependencias en función del mecanismo utilizado para proporcionar la instancia del servicio al cliente. Si la instancia se proporciona en el constructor, será inyección por constructor. Si la instancia se proporciona en un atributo del cliente, será inyección por *setter*. Y, si la instancia se proporciona a través de un método, será inyección por método. Las tres técnicas tienen sus usos recomendados y pueden convivir en un mismo proyecto. En este proyecto se ha hecho uso, principalmente, de inyección por constructor, que es especialmente útil para dependencias necesarias ya que estarán disponibles desde el instante de creación, y, en algún caso en el que la dependencia

está en una clase abstracta, de inyección por *setter* para evitar que sus clases hijas tengan que proporcionarle la instancia en el constructor.

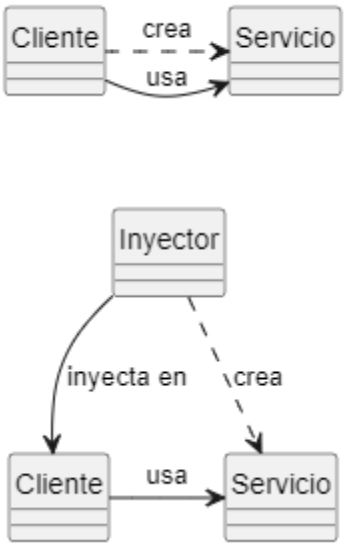

Figura 13. Arriba, un cliente que usa un servicio sin inyección de dependencias y, abajo, con inyección de dependencias.

Normalmente, cuando se utiliza este patrón se usa un *framework* de inyección de dependencias que resuelve estas automáticamente e instancia los distintos componentes cuando corresponde.

5.4.4. Inversión de dependencias

La inversión de dependencias es un patrón que busca alterar la dirección en la que fluyen las dependencias entre una clase cliente y el servicio del que depende. Para ello, se hace que el cliente publique una interfaz de la que dependerán los servicios que use como en la Figura 14. Gracias a esto, el cliente dejará de estar acoplado al servicio y es el servicio el que pasa a estar acoplado al cliente. Esto significa que cambios en el servicio no implicarán cambios en el cliente (pero sí a la inversa).

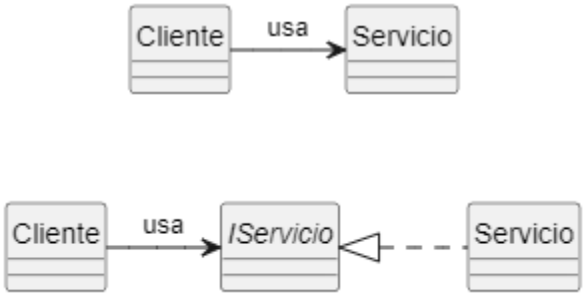

Figura 14. Arriba, cliente que depende de un servicio sin inversión de dependencias y, abajo, con inversión de dependencias.

Se puede utilizar en arquitecturas por capas para eliminar las dependencias de la capa de dominio y la capa de aplicación de la capa de infraestructura, razón por la que se usa en este trabajo (en concreto, por ejemplo, los repositorios). También es útil para facilitar las pruebas unitarias de diferentes componentes ya que la inversión de dependencias facilita sustituir las dependencias por *mocks* más sencillos con un comportamiento más predecible. Es muy utilizado en conjunto con la inyección de dependencias.

5.4.5. Adaptadores

El patrón adaptador sirve para transformar la interfaz de un servicio en una que el cliente entienda por medio de una clase adaptador que actúa como intermediaria y transforma los datos entre los contratos utilizados por ambas interfaces como se ve en la Figura 15. Se puede utilizar en casos en los que el cliente depende de una interfaz diferente a la que el servicio proporciona o cuando se quiere desacoplar un cliente de un servicio cuya interfaz es controlada por un agente externo. De esta forma, el servicio se puede reemplazar por otro diferente sin que el cliente tenga que ser modificado para adaptarse a la nueva interfaz.

En este trabajo se ha utilizado este patrón en ambos contextos. El primer caso de uso ha sido útil a la hora de adaptar la interfaz de Logger de NestJS a la interfaz esperada por Sequelize. El segundo caso de uso se ha utilizado para abstraer el envío de correos electrónicos con Nodemailer y evitar que, en caso de desear cambiarlo en el futuro, suponga modificaciones en los servicios que dependan del envío de correos electrónicos.

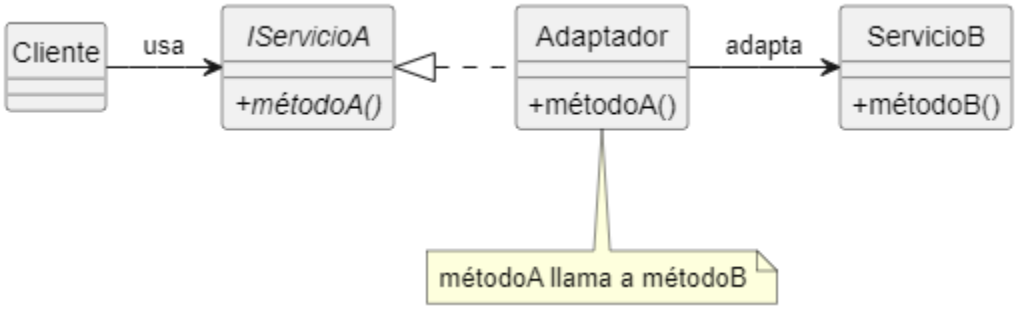

Figura 15. Patrón adaptador.

5.4.6. Promesa

El patrón promesa es un método para representar el resultado de una computación asíncrona que estará disponible (o no) en el futuro con una interfaz similar a la de la Figura 16. Las promesas se crean con una tarea que comenzará su ejecución inmediatamente, pero completará con éxito (resolverá) o fallará (se rechazará) en el futuro. El resultado de esta computación puede ser manejado con métodos como “then” para procesar los casos de éxito, “catch” para procesar los casos de error o “finally” que se ejecuta independientemente del resultado. TypeScript y JavaScript además cuentan con la sintaxis async/await que facilita mucho trabajar con este patrón.

Se ha hecho un uso extensivo de este patrón para representar cualquier tipo de computación asíncrona como, por ejemplo, consultas a base de datos o peticiones HTTP.

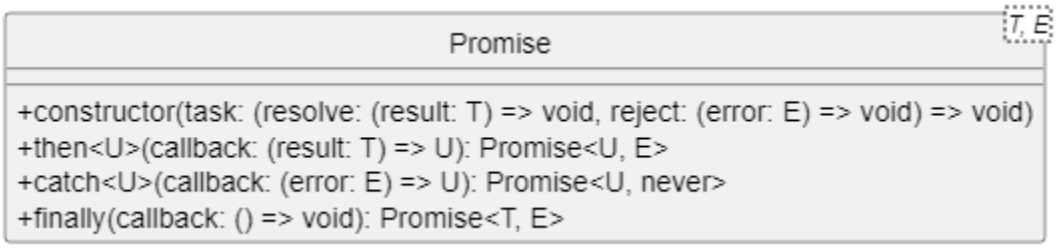

Figura 16. Patrón promesa.

5.4.7. Repositorio

El patrón repositorio abstrae el acceso a datos por medio de una interfaz que emula una colección de objetos del dominio. Los repositorios son una variación específica del patrón adaptador que se utiliza para desacoplar la capa de dominio del acceso a datos (que es una responsabilidad de la capa de infraestructura). Una buena implementación del patrón repositorio permitiría cambiar el mecanismo de acceso a datos sin tener que modificar nada más que componentes de la capa de infraestructura. Gracias a esto, este patrón facilita el desarrollo de pruebas de un sistema.

5.5. Implementación

En este apartado se detallan algunos aspectos destacables de la aplicación. En primer lugar, se hablará de cómo se ha afrontado el reto de diseñar la interfaz del cliente web en el apartado 5.5.1. Más tarde, en los apartados 5.5.2 y 5.5.3 se habla de dos de los grandes retos que se han afrontado en la aplicación. Por último, en los apartados 5.5.4 y 5.5.5 se habla de la generación de *embeddings*.

5.5.1. UI/UX

Pese a su importancia, el UI/UX y el diseño no ha sido una prioridad durante el proyecto debido a la escasez de recursos y la enorme complejidad del resto de áreas. Sin embargo, sí que se ha intentado tener en cuenta sobre todo de cara a generar una imagen de marca fuerte y una metáfora consistente.

La primera de las acciones en este aspecto ha sido la elección de un nombre para la aplicación. Se buscó que este nombre fuera sencillo y fácilmente pronunciable por una persona no japonesa pero que, además, tuviera algún significado en japonés. En base a estos criterios, el nombre elegido ha sido “Momokado” (桃カード en japonés) que podría traducirse como “carta melocotón”.

En base al nombre de la aplicación se diseñó un logotipo personalizado desde cero. Para ello se partió de un melocotón. En la Figura 17 se incluyen algunas de las iteraciones del logotipo de la aplicación. Se comenzó con un diseño plano buscando sencillez, pero pareció que era un estilo poco elegante. Por ello, se pasó a opciones con más volumen a base de degradados para, finalmente, volver a un estilo simple con un único color ya que

era más elegante y mejoraba la legibilidad. Además, en la Figura 18 se muestran variantes a los tres logotipos de la fila inferior incluyendo el logotipo de la aplicación. Por último, en el anexo 8.2 se incluyen las medidas detalladas del logotipo.

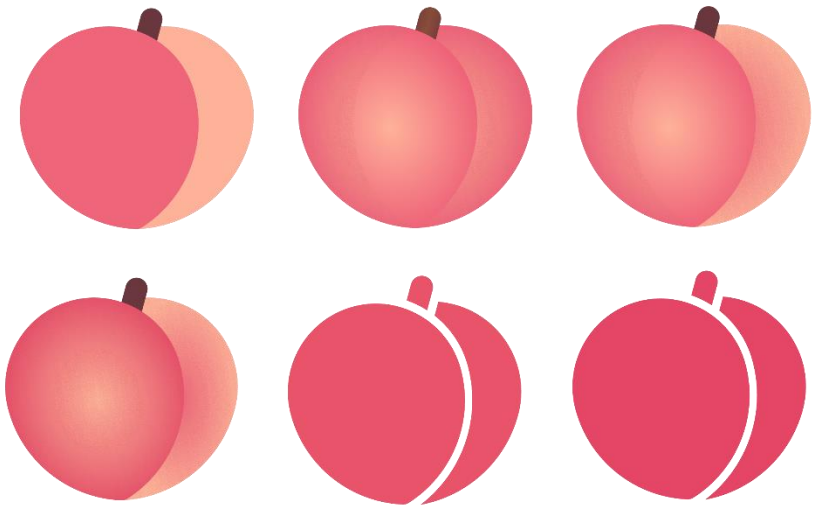

Figura 17. Diversas iteraciones del logotipo de la aplicación. La versión definitiva es la de la derecha de la fila inferior.

Como se ha podido anticipar con las ilustraciones asociadas a las iteraciones del logotipo, un tono entre rosa y rojo se ha seleccionado como color primario para la aplicación. La razón de esto es que los todos rosas son muy típicos de la cultura japonesa gracias a los paisajes con cerezos en flor y el rojo sobre blanco es reminiscente de la bandera japonesa.

Figura 18. Logotipo de la aplicación con el nombre.

En cuanto al lenguaje de los componentes de la aplicación se ha optado por un lenguaje sencillo y sobrio que potencie la legibilidad (ver, por ejemplo, la Figura 19). Se pretende dar una imagen carismática y divertida que anime al usuario a utilizar la aplicación. De esta forma, se lucha con la posible resistencia al estudio que pueda tener el usuario. Este lenguaje de diseño está basado en el lenguaje de diseño de Radix UI Themes (la biblioteca de componentes utilizada), pero se le ha dado una identidad propia a base de ilustraciones y componentes personalizados diseñados para este proyecto, por ejemplo, en la Figura 19 las tres ilustraciones, el logotipo y el icono de perfil.

La funcionalidad principal de la aplicación, el estudio, está basada en la metáfora de tarjetas. De hecho, los grupos de tarjetas se llaman barajas. Para reforzar esta metáfora y ayudar al usuario a comprender el funcionamiento de la aplicación de manera intuitiva, se ha hecho uso de esqueumorfismo tanto en tarjetas como en barajas (ver Figura 20).

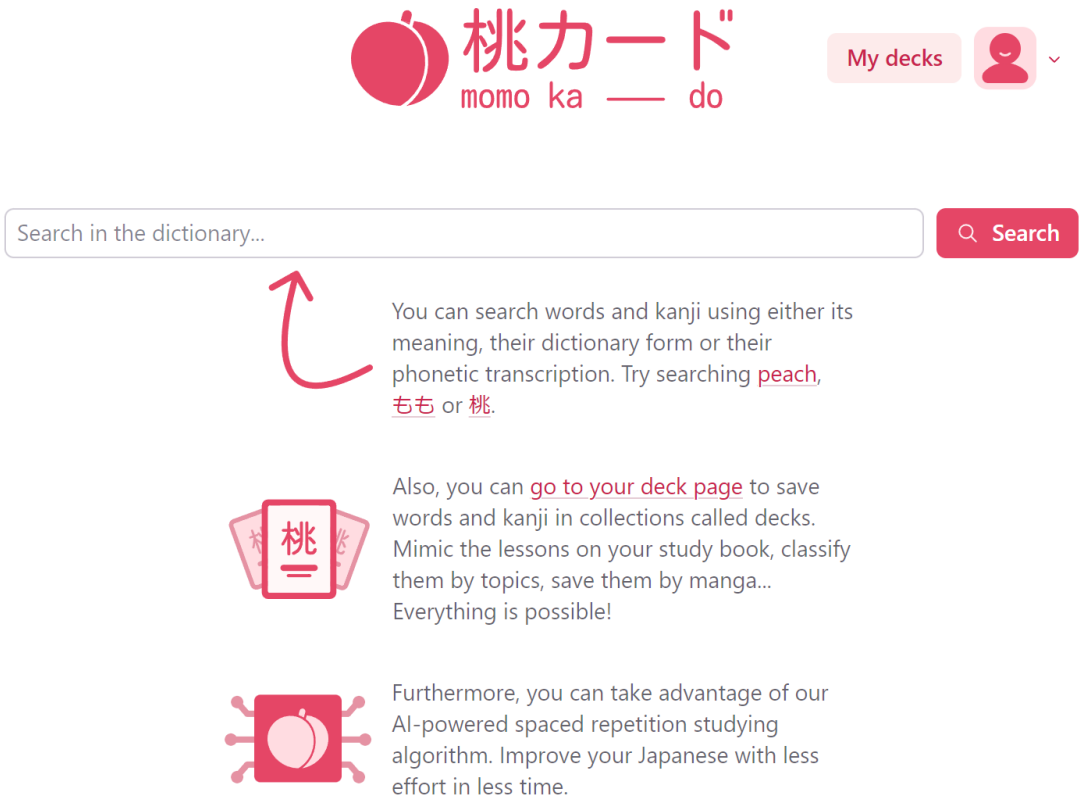

Figura 19. Ejemplo de pantalla de la aplicación.

Figura 20. Ejemplos de esqueumorfismo en la aplicación.

En definitiva, se ha trabajado en encontrar un balance entre el uso de bibliotecas de componentes que aceleren el desarrollo del software y una identidad única y carismática.

5.5.2. Búsqueda de palabras

Como se mencionó en el apartado 2, el japonés tiene dos silabarios, los kanjis y, además, hay una transcripción fonética con caracteres latinos (romaji) muy utilizada por los occidentales que están aprendiendo el lenguaje. Si, además, se tiene en cuenta que cada palabra puede tener una transcripción fonética diferente en cada uno de estos es fácil ver cómo crece la complejidad de la búsqueda. Para dejar claro el alcance de esta funcionalidad se definen los siguientes requisitos:

- El usuario puede buscar una palabra o *kanjis* por su transcripción fonética en *katakana*, *hiragana* o *romaji*. Un *kanji* puede tener múltiples lecturas, una palabra sólo una.
- El usuario puede buscar una palabra por su propia escritura. Si se proporciona una oración en japonés, se buscarán todas y cada una de las palabras. En estos casos, se buscarán todos los *kanjis* que aparezcan en la palabra.
- El usuario puede buscar palabras o *kanjis* por su significado. Ambos pueden tener varios significados.

Para resolver este problema se planteó el siguiente algoritmo:

1. Si la consulta de la búsqueda tiene algún carácter japonés:
    a. Se tokeniza la oración y se lematizan cada uno de los tokens. Para cada lema:
        i. Si es un lema conocido se buscan todas las palabras en base de datos que tengan el lema proporcionado y/o la lectura inferida por el lematizador. También se buscan los *kanjis* que aparezcan en ese lema (si los hay) o con esa pronunciación (si no los hay).
        ii. Si es un lema desconocido:
            1. Se transforma el token hasta *katakana* (originalmente podría estar en *hirgana*, *katakana* o *romaji*) dividiéndolo en sílabas con un tokenizador y se transcriben con una función que transforma

sílabas entre los tres sistemas de transcripción. Esta transformación es necesaria porque la transcripción fonética en base de datos se guarda como *katakana*.

2. Se buscan palabras y *kanjis* en base de datos con esa pronunciación.

2. Si la consulta no tiene ningún carácter japonés, pero puede ser tokenizada (con el tokenizador del apartado anterior) como romaji:
    a. Se divide la palabra en sílabas con el tokenizador y se transcribe utilizando el silabario *katakana*.
    b. Se busca en base de datos las palabras o *kanjis* con esa pronunciación.
3. Si la consulta no tiene ningún carácter japonés y no puede ser tokenizada como romaji se buscan las palabras y *kanjis* con un significado que encaje con esa consulta utilizando el motor de búsqueda de texto completo de PostgreSQL.

En resumen, para poder implementar la búsqueda ha hecho falta desarrollar una función que permita detectar el modo de búsqueda en base a únicamente la consulta, un tokenizador para cada uno de los tres mecanismos de escritura fonéticos del sistema de escritura japonés, una función de transcripción de los tokens extraídos por este tokenizador a katakana y multitud de consultas a base de datos que recuperan información de maneras muy diversas.

5.5.3. Sistema de estudio

El aspecto diferenciador principal de este trabajo es el uso de *embeddings* para ayudar al estudiante a diferenciar con mayor facilidad *kanjis* similares. Sin embargo, hasta el momento no se ha entrado en detalle de cómo se consigue esto. Actualmente, la aplicación permite estudiar tanto *kanjis* como palabras, pero utiliza métodos de generación de preguntas distintos para ambos, ya que no se cuenta con *embeddings* para palabras.

En ambos casos las preguntas se pueden generar como traducción directa (japonés a inglés) o como traducción inversa (inglés a japonés). En las preguntas de traducción directa se muestra un *kanji* o una palabra y se pregunta cuál de cuatro opciones de significados se

corresponden con ella. En las preguntas de traducción inversa se muestra un significado y se pregunta cuál de cuatro *kanjis* o palabras son los correspondientes a este.

En el caso de las preguntas basadas en palabras se usan, además de la respuesta correcta, tres opciones muestreadas aleatoriamente de todo el conjunto de palabras de base de datos asignándole una probabilidad uniforme a cada una de ellas. Esta consulta es bastante sencilla y se consigue utilizando la función "random" de PostgreSQL en la cláusula "order by".

El caso más complejo es el de generación de las opciones para *kanjis*. En este caso el reto está en generar respuestas similares al *kanji* elegido, pero, al mismo tiempo, aleatorias. El criterio de similitud entre dos *kanjis* se ha definido como la distancia coseno entre sus *embeddings* semánticos en el caso de la traducción directa (se buscan generar significados similares) y la distancia coseno entre sus *embeddings* visuales en el caso de la traducción inversa (se busca mostrar *kanjis* visualmente similares). Estos *embeddings* son generados con los modelos descritos en los dos próximos apartados y almacenados en base de datos para su uso posterior.

Parte del reto radicó en cómo conseguir realizar esta tarea sin recuperar todos los kanjis en base de datos para poder realizar el muestreo aleatorio. Esto es, precisamente, un problema de muestreo de depósito. Para resolverlo, se ha utilizado el algoritmo propuesto por Efraimidis y Spirakis en [26] que, en el caso de la implementación de este trabajo se reduce a realizar una consulta que recupere los primeros valores al ordenar por el resultado de calcular $f(x)$ para cada fila según la Ecuación 10. En esta ecuación w_r hace referencia a los *embeddings* de la referencia y w_x a los *embeddings* de la fila.

$$f(x) = -\log \frac{random()}{1 - \frac{d(w_r, w_x)}{2}}$$

Ecuación 10. Formula de orden de muestreo de *kanji*.

Estas preguntas se mostrarán al usuario y su respuesta se usará para alimentar al algoritmo de repetición espaciada. En concreto se ha tenido que inferir la dificultad que supone la

pregunta para el usuario y, en este caso, se ha optado por usar un mecanismo basado en el tiempo de respuesta. Si se tardan menos de cinco segundos se considera que la respuesta está bien afianzada y es fácil para el usuario. Si la respuesta tarda entre 5 y 20 segundos se considera que tiene una dificultad normal. Y, desde 20 segundos hasta 45 segundos se considera que es difícil. Estos tiempos podrían ajustarse con estadísticas de tiempos de usuarios una vez se tenga una base de usuarios sustancial.

5.5.4. *Embedding* semántico

Los *embeddings* semánticos se han generado utilizando una implementación propia de la arquitectura Word2Vec basada en *skip-grams*. Pese a que es de las menos utilizadas en la actualidad porque el resto se consideran superiores, se ha seleccionado esta de todas las opciones presentadas en la sección de estado del arte por diversos motivos. En primer lugar, FastText, ELMo y BERT generan *embeddings* contextuales que, para el caso de uso general, conforman una opción superior, pero en este caso serían una clara desventaja. Los *kanjis* que se van a presentar al usuario van a aparecer aislados, sin contexto a su alrededor, así que esta alternativa no es aplicable. Además, estos *embeddings* habría que generarlos para cada contexto en el que aparece el *kanji* (cada vez que aparezca) y no podrían ser guardados en base de datos (como se hace ahora), lo cual constituiría una alternativa menos eficiente. Y, en segundo lugar, se selecciona Word2Vec frente a GloVe por ser más sencilla de implementar.

Se han utilizado dos corpus para entrenar este modelo: el corpus Tatoeba [27] y el corpus CC-100 Ja [28]. El primero de los dos es demasiado pequeño y tras el primer experimento se descartará. Y, CC-100 Ja es un corpus enorme (15 GB) y, debido a que la máquina donde se ha entrenado no tenía suficientes recursos, se ha tenido que limitar su uso a las 25.000 primeras oraciones.

Dado que es aprendizaje no supervisado (no se disponen de ejemplos del resultado esperado del entrenamiento), es difícil evaluar el rendimiento de los modelos. Típicamente, en casos de *embedding* similares a este, se puede utilizar un criterio basado en el uso final que se les vaya a dar, por ejemplo, si van a ser usados para entrenar un clasificador, los *embeddings* se evalúan con el resultado de entrenar un clasificador. En este caso, la

utilidad final es calcular la similitud, algo que es bastante difícil de evaluar sin un conjunto de datos de similitudes (del que no se dispone). Por lo tanto, el criterio de selección final ha tenido un componente cualitativo sobre los *embeddings* (se ha comprobado que se generen *kanjis* relacionados cuando los *embeddings* sean cercanos) y un componente basado en la función de pérdida del modelo durante el entrenamiento.

Utilizando este modelo se han realizado los siguientes experimentos:

El experimento 1 se ha realizado el entrenamiento sobre Tatoeba durante 30 epochs con un tamaño de contexto de 8 y un tamaño de espacio latente de 64. Con esta prueba se pretendía comprobar el funcionamiento de la implementación. Además, se valida que el modelo se podía utilizar con el objetivo deseado (generaban *embeddings* similares para *kanjis* similares), ya que estos modelos están diseñados para palabras y no *kanjis*, que tienen un significado menos concreto. En este caso, el resultado fue mejorable (algunos *embeddings* cercanos no tenían una relación semántica) y, por tanto, en el siguiente experimento se probará a utilizar un corpus más grande.

El experimento 2 es similar al experimento 1 pero utilizando las 25.000 primeras frases de CC-100 Ja. En este caso, Los resultados fueron mejores y en los ejemplos que se ha comprobado los *embeddings* cercanos están más relacionados entre sí.

El experimento 3 busca encontrar el valor óptimo de tamaño de espacio latente. Para ello se entrena el modelo utilizando las 25.000 primeras frases de CC-100 Ja variando el tamaño de *embedding* para las potencias de dos entre 2 y 512. Adicionalmente, más tarde se repitió con el valor de 300. La pérdida final de cada modelo se puede observar en la Figura 21. Es fácil observar que el valor óptimo es 256 y con bastante diferencia frente a los valores de su entorno.

En definitiva, en base a estos experimentos, se ha decidido utilizar un tamaño de *embedding* semántico de 256 y utilizar los primeros 25.000 elementos del corpus CC-100 Ja como conjunto de entrenamiento.

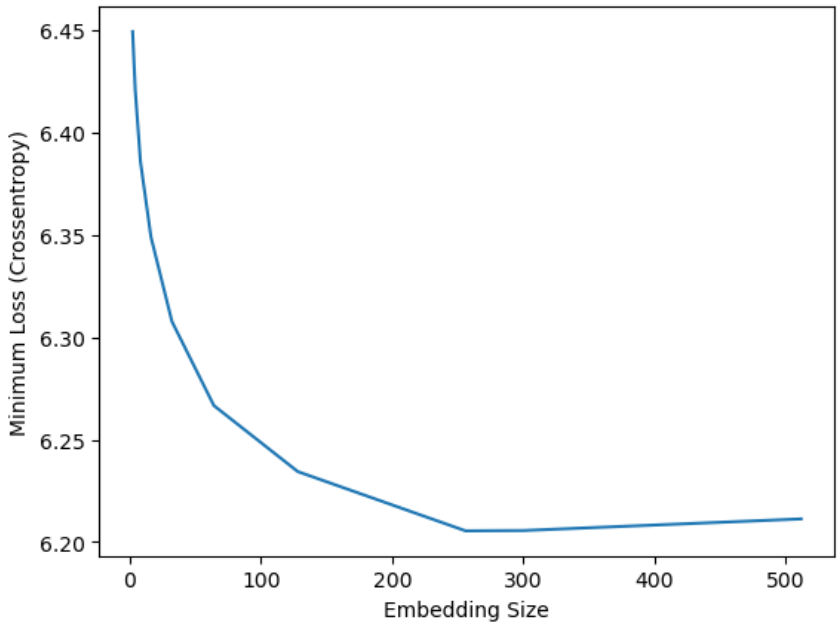

Figura 21. Resultado del tercer experimento de los *embeddings* semánticos.

5.5.5. *Embedding* visual

La generación de los *embedding* visuales se ha basado en una arquitectura autocodificador personalizada. Esta arquitectura está formada por dos componentes: el codificador, que transforma la imagen a una representación vectorial, y el decodificador, que realiza la transformación inversa.

En este caso, tanto el codificador como el decodificador están compuestas en base a un bloque que se ha llamado bloque de convolución-*pool* que está formado por una convolución con un filtro de tamaño 3x3, una normalización de lote, una función de activación (*leaky ReLU*) y una capa de *pooling* con una ratio de 2.

Por un lado, el codificador se encuentra formado por una secuencia de siete bloques convolución-*pool* seguidos de una capa de neuronas densamente conectadas sin activación que dan el resultado del *embedding*. Por otro lado, el decodificador está formado por una capa de neuronas densamente conectadas con normalización de lotes, seguida de siete bloques convolución-pool y, finalmente, un bloque convolución (como el convolución-*pool* pero sin *pooling*) con un filtro 7x7.

Esta arquitectura se ha entrenado utilizando un conjunto de datos generado a partir de renderizar cada carácter de la lista Jouyou kanji[19] en diversas variantes de fuentes de Google Fonts para un total de 62 muestras por cada uno de los 2136 *kanjis*. Inicialmente se planteó utilizar alguna fuente más como por ejemplo las de la Figura 22 pero se descartó debido a que usan un estilo demasiado artístico que perjudica al aprendizaje de la red.

Figura 22. Ejemplo de fuentes demasiado artísticas.

Por unas razones similares a la de los *embeddings* semánticos, el criterio de selección final ha considerado una parte cualitativa sobre las reconstrucciones (un autocodificador que genere mejores reconstrucciones estará capturando mejor las características) y otra parte basada en la función de pérdida durante el entrenamiento del modelo. Para validar que no se ha producido un sobreajuste, se realiza una comprobación cualitativa ya que los colapsos de modo son fáciles de detectar al visualizar los resultados de la generación.

Utilizando variantes de esta arquitectura se han realizado cinco experimentos:

En el experimento 1, se entrenó un modelo pequeño durante un par de épocas con el objetivo de comprobar que el modelo funcionara correctamente.

En el experimento 2, el objetivo era detectar el tamaño de vector latente óptimo. Para ello, se probaron los valores de potencias de dos entre 2 y 256. En este caso, además, se usaron las fuentes estilísticas mencionadas unos párrafos más arriba durante el entrenamiento. La pérdida en el momento de finalizar el entrenamiento se representa en la Figura 23, donde se observa que el tamaño óptimo está en 128 elementos. Sin embargo, en la práctica se

[19] Una lista del Ministerio de Educación japonés que estandariza los *kanjis* que deben ser enseñados en los colegios e institutos. Se consideran los *kanjis* indispensables para conseguir una fluidez suficiente.

usará 64 debido a que la diferencia es despreciable y se trata de la mitad de tamaño, así que su tratamiento será más eficiente.

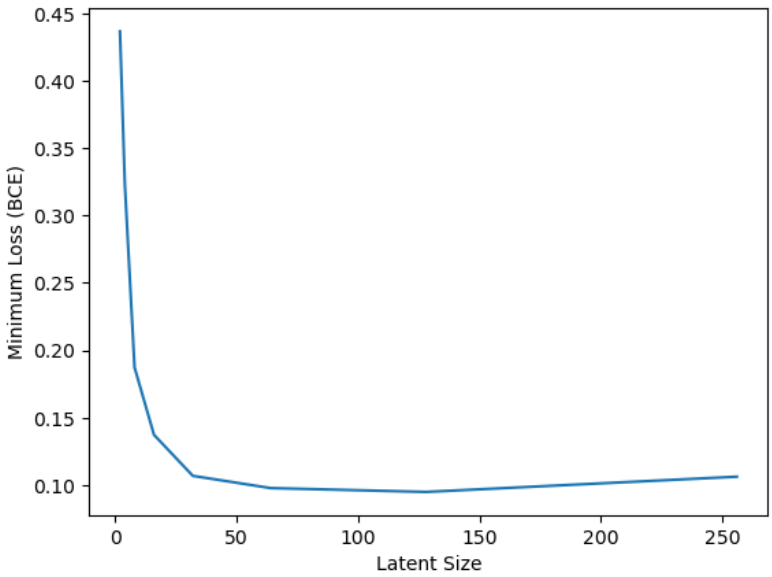

Figura 23. Resultado del segundo experimento de los *embeddings* visuales.

En el experimento 3 se realiza la misma prueba que en el experimento 2 pero esta vez eliminando las fuentes con una estética demasiado artística. Los resultados mejoraron bastante tanto cuantitativa como cualitativamente (se consigue una perdida menor utilizando 64 elementos en este caso que 128 en el caso anterior).

El experimento 4 varía la arquitectura para introducirle un componente estocástico convirtiéndola en un autocodificador variacional. Este experimento es análogo al experimento 1 pero con esta arquitectura alternativa.

El experimento 5 se realiza con la misma metodología que el experimento 3 pero esta vez se utilizando el autocodificador variacional. En este caso, se repite el mismo resultado que en los experimentos 2 y 3: una gráfica con una forma similar a la Figura 23 con el mínimo en 128 valores en el espacio latente, pero sin mucha diferencia con respecto a 64 valores. La diferencia en el rendimiento con respecto al modelo del apartado 3 es despreciable. Sin embargo, se utilizando el autocodificador variacional se obtiene una gran ventaja: los

embeddings son continuos. Esto podría permitir, en futuros trabajos, generar mezclas de *kanji* como la de la Figura 24. Nótese que el *kanji* del medio no existe, pero utiliza patrones que pueden verse en otros *kanji* y, ciertamente es verosímil para un ojo no experto.

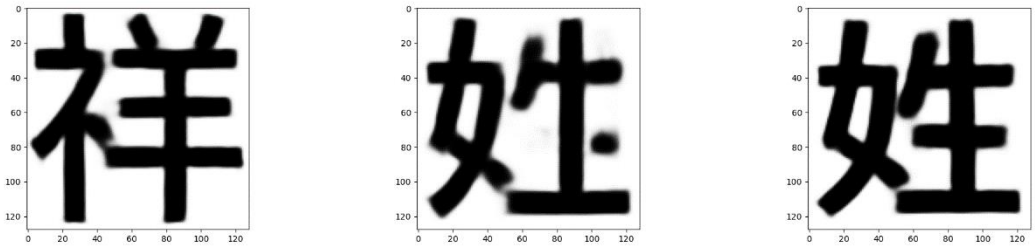

Figura 24. A los extremos, dos *kanjis* seleccionados al azar y, en el medio, el resultado de combinar ambos *kanjis* utilizando el autocodificador variacional.

Tras estos experimentos, se decidió utilizar la arquitectura de autocodificador variacional con 64 elementos en el espacio latente entrenada sobre el conjunto de datos sin fuentes artísticas.

5.6. Pruebas

La calidad del software ha sido importante para el proyecto y, por eso, se han tomado varias acciones para asegurar una calidad mínima. Una de ellas ha sido el despliegue de un pipeline de integración continúa en GitHub Actions. Este pipeline tiene las funciones de verificar la calidad del código (cumplimiento de estándares y buenas prácticas) y de verificar el correcto funcionamiento del software en todo momento.

Este pipeline cuenta con dos procesos principales. El primero de ellos ejecuta ESLint sobre todos los ficheros del proyecto. En caso de que ESLint detecte alguna digresión con las reglas de estilo o alguna mala práctica, el *pipeline* fallará. El segundo de ellos compila el proyecto software y ejecuta un banco de pruebas automáticas sobre él con Jest. En caso de que alguna de estas pruebas falle, el *pipeline* falla. Este segundo proceso se realiza

únicamente en el servidor debido a que implementar esta clase de pruebas en el cliente era más complejo y el desarrollador tiene menos experiencia en ese ámbito.

Este pipeline se ejecutaba automáticamente en dos momentos. En primer lugar, al integrar ramas de características hacia la rama base (*pull requests* hacia la rama "develop") para garantizar que las nuevas funcionalidades se integran adecuadamente con las ya existentes. Y, en segundo lugar, en cada cambio (o *commit*) para detectar posibles alteraciones en los componentes del sistema y posibles regresiones de errores. Al ejecutar estas pruebas automáticas, se registran los tiempos de ejecución de cada una de ellas y la memoria consumida, aunque no existe ninguna alerta automática para ellas (a no ser que sea algo demasiado notable y haga que falle el proceso por completo).

El banco de pruebas automáticas cuenta un total de 23 grupos de pruebas, cada uno de ellos asociados a un método de la interfaz pública del servidor[20]. Entre todos estos grupos existen un total de 130 pruebas. Todas estas pruebas son pruebas de caja negra y se limitan a comprobar las salidas observables del sistema (resultados devueltos y resultados generados en la base de datos). Cada una de ellas verifica el correcto funcionamiento de un camino de éxito (e.g., si un usuario intenta recuperar un *kanji* existente se debería devolver un código HTTP 200) o un camino de excepción (e.g., si un usuario intenta recuperar un kanji no existente se debería devolver un código HTTP 404).

Ha habido dos casos en los que se han roto los compromisos de observar únicamente el sistema desde fuera y en las que se han utilizado *mocks* para poder verificar más fácilmente la funcionalidad del sistema. La primera de ellas es que, al ejecutar las pruebas automáticas (y sólo en este caso) el módulo Bcrypt es reemplazado por una implementación que devuelve las contraseñas sin *hashear* para verificar más fácilmente que, en efecto, es la que se guarda en base de datos y obtener un mejor rendimiento en las pruebas (generar el *hash* puede ser costoso). El segundo de ellos es el envío de correos electrónicos que, de otra manera, no habrían podido ser incorporados en las pruebas automáticas. En este caso, existe una implementación alternativa que guarda los correos

[20] En un primer momento se planteó realizar pruebas de granularidad más fina en componentes más pequeños, pero el tiempo coste era demasiado alto.

electrónicos en memoria para que el código de las pruebas pueda recuperarlos y comprobar que, en efecto, se está notificando de su envío.

En el caso del cliente, dado que el sistema automático de pruebas no existía (pero sí se comprobaba el código con ESLint de manera automática en el *pipeline*), se han intentado realizar más pruebas manuales en forma de pruebas unitarias sobre componentes utilizando Storybook y pruebas del sistema completo utilizando, también, el servidor.

La implementación de esta metodología de pruebas ha sido de gran importancia para la calidad de la aplicación y ha ayudado enormemente a encontrar errores. Por ejemplo, uno de los fallos más elusivos que se detectó era una fuga de memoria provocada por el *logger* de la aplicación (nestjs-pino) que tuvo que ser reemplazado por una alternativa (nest-winston).

6. CONCLUSIONES Y LÍNEAS DE TRABAJO FUTURAS

6.1. Conclusiones

En primer lugar, se destaca que se ha logrado cumplir con los objetivos detallados al inicio de esta memoria:

- Se han analizado con éxito las aplicaciones existentes para el estudio del japonés y se han detectado los puntos fuertes a imitar y los puntos débiles a evitar. Gracias esto, se ha detectado una debilidad existente en todas ellas y que se puede explotar como un factor diferenciador: la falta de herramientas para facilitar al usuario el encontrar patrones diferenciadores entre *kanjis* similares (tanto a nivel de significado como visualmente). Además, se ha mantenido la metáfora de "cartas" utilizada en la mayoría de ellas, los algoritmos de repetición espaciada y un diseño agradable.
- Se han analizado diversas técnicas de inteligencia artificial y procesamiento del lenguaje natural y se han propuesto soluciones a problemas con ellas. En primer lugar, se han desarrollado dos modelos de *embedding* de *kanjis*: uno basado en la arquitectura autocodificador para poder calcular la similitud visual y otro basado en la arquitectura Word2Vec para poder calcular la similitud semántica. Y, en segundo lugar, se ha diseñado un algoritmo de búsqueda de *kanjis* y palabras japonesas que permite utilizar diversos tipos de consultas, para lo que ha sido necesario desarrollar algunas gramáticas formales para interpretar transcripciones del japonés y utilizar técnicas del procesamiento del lenguaje natural como la tokenización o la lematización.
- Se ha diseñado una solución software que mejora las ya existentes incorporando información de similitud de *kanjis*. Para ello, se ha utilizado una arquitectura basada en un cliente y un servidor que se comunica con una base de datos relacional con extensiones para el almacenamiento de datos vectoriales. Se ha hecho uso de una arquitectura modelo-vista-controlador en el cliente y una arquitectura por capas en el servidor que favorecen la escritura de código de calidad, cohesivo y con poco acoplamiento. Además, se han diseñado dos modelos de inteligencia artificial que alimentarán esta base de datos con extensiones vectoriales con *embeddings* de *kanjis* que utiliza el servidor.

- Se ha implementado la solución diseñada utilizando tecnologías como TypeScript, NestJS, React o PostgreSQL. Además, se ha implementado un mecanismo de integración continua que ha garantizado la existencia de una versión funcional y de calidad en todo momento en la rama base del repositorio. También, se han implementado los modelos diseñados utilizando Python y PyTorch.

La filosofía *Lean* y la metodología Kanban utilizadas en este proyecto han ayudado enormemente a la consecución de estos objetivos. Además, en lo personal, son una filosofía y una metodología con las que me siento bastante cómodo. Así que, es probable que las utilice en futuros proyectos.

Por otro lado, me gustaría destacar el buen funcionamiento de herramientas como Storybook o Jest que, definitivamente, han sido un acierto para el proyecto. También, me he sentido cómodo con la interfaz de Sequelize y ha realizado sin problemas su trabajo. Sin duda, otra de las herramientas que tendré en cuenta en el futuro. Por otro lado, quizás me hubiera gustado probar otras alternativas a NestJS, ya que en numerosas ocasiones me he sentido limitado por algunas de sus restricciones en los mecanismos que proporciona para inyectar dependencias y fomenta código demasiado verboso en algunas ocasiones.

En lo personal, creo que este proyecto me ha ayudado a profundizar en la lengua japonesa y el aprendizaje espaciado, dos áreas que me interesaban. A mayores, me ha permitido aprender herramientas que no había utilizado previamente como NestJS y Sequelize. Incluso, me ha ayudado a descubrir algunas áreas que desconocía y me interesan. Sin embargo, ha sido un proceso complicado y lleno de sacrificios que ha tenido un alto coste a nivel personal.

6.2. Líneas de trabajo futuras

Actualmente, no hay ninguna forma de consultar el progreso de aprendizaje de un concepto. Saber esto puede ser útil para que el usuario planifique su estudio. Sin embargo, esta tarea no es sencilla porque implica representar parámetros complejos y se quiere conseguir esto sin ensuciar la interfaz simple que se propone.

También se quiere dar más importancia al diseño UI/UX del proyecto en el futuro. Para eso, un primer paso será incorporar herramientas que permiten analizar el comportamiento del

usuario para encontrar patrones de uso de la aplicación (por ejemplo, Mixpanel). Incluso, podría estudiarse el incorporar a un diseñador al equipo.

Se desea trabajar también en mecanismos de evaluación de los algoritmos utilizados. Esto probablemente implique despliegues incrementales con pruebas A/B para comprobar si realmente nuevas versiones mejoran el comportamiento existente.

Además, también se quiere trabajar en la mejora de los contenidos (por ejemplo, reglas mnemotécnicas o etimología) y traducir la aplicación a más idiomas, ya que ahora sólo está disponible en inglés. Para esta tarea probablemente se necesitará la ayuda de un experto en filología japonesa y/o traducción.

7. BIBLIOGRAFÍA

[1] The World Bank, «GDP (current US$),» [En línea]. Available: https://data.worldbank.org/indicator/NY.GDP.MKTP.CD?most_recent_value_desc=true.

[2] Public Relations Office, Government of Japan, «Exporting the Attractions of "Cool Japan",» [En línea]. Available: https://www.gov-online.go.jp/eng/publicity/book/hlj/html/202006/202006_01_en.html.

[3] Foreign Service Institute, U.S. Department of State, «Foreign Language Training,» [En línea]. Available: https://www.state.gov/foreign-language-training/.

[4] R. C. Atkinson y R. M. Shiffrin, «Human Memory: A proposed System and Its Control Processes,» de *Psychology of Learning and Motivation*, Academic Press, 1968, pp. 89-195.

[5] J. R. Anderson, Language, memory, and thought, Lawrence Erlbaund Associates, 1976.

[6] H. Ebbinghaus, Memory: A Contribution to Experimental Psychology, New York City: Teachers College, Columbia University, 1913.

[7] P. A. Wozniak, Optimization of learning: A new approach and computer application, Poznan: University of Technology in Poznan, Computer Science Center, 1990.

[8] P. A. Wozniak, E. J. Gorzelanczyk y J. A. Murakowski, «Two components of long-term memory,» de *Neurobiol Exp (Wars.)*, Varsovia, 1995.

[9] S. Leitner, So lernt man lernen: der Weg zum Erfolg, Augsburg Weltbild-Verl, 1999.

[10] B. Settles y B. Meeder, «A Trainable Spaced Repetition Model for Language Learning,» de *54th Annual Meeting of the Association for Computational Linguistics*, Berlin, 2016.

[11] J. Ye, J. Su y Y. Cao, «A Stochastic Shortest Path Algorithm for Optimizing Spaced Repetition Scheduling,» de *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, Nueva York, 2022.

[12] SuperMemo, «Stabilization curve,» [En línea]. Available: https://supermemo.guru/wiki/Stabilization_curve.

[13] T. Mikolov, K. Chen, G. Corrado y J. Dean, «Efficient Estimation of Word Representations in Vector Space,» de *International Conference on Learning Representations (ICLR)*, Scottsdale, 2013.

[14] J. Pennington, R. Socher y C. D. Manning, «GloVe: Global Vectors for Word Representation,» de *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, 2014.

[15] P. Bojanowski, E. Grave, A. Joulin y T. Mikolov, «Enriching Word Vectors with Subword Information,» *Transactions of the Association for Computational Linguistics,* vol. 5, pp. 135-146, 2017.

[16] M. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee y L. Zettlemoyer, «Deep contextualized word representations,» de *International Conference on Learning Representations (ICLR)*, Vancouver, 2018.

[17] J. Devlin, M.-W. Chang, K. Lee y K. Toutanova, «BERT: Pre-training of Deep Bidirectional Transformers for Language,» de *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, Minneapolis, 2019.

[18] K. Simonyan y A. Zisserman, «Very Deep Convolutional Networks for Large-Scale Image Recognition,» de *International Conference on Learning Representations (ICLR)*, San Diego, 2015.

[19] K. He, X. Zhang, S. Ren y J. Sun, «Deep Residual Learning for Image Recognition,» de *IEEE Conference on Computer Vision and Patern Recognition (CCVPR)*, Boston, 2015.

[20] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke y A. Rabinovich, «Going Deeper with Convolutions,» de *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, 2015.

[21] M. Tan y V. L. Quoc, «EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks,» de *Proceedings of Machine Learning Research (PMLR)*, Long Beach, 2019.

[22] K. He, G. Gkioxari, P. Dollár y R. Girshick, «Mask R-CNN,» de *IEEE International Conference on Computer Vision (ICCV)*, Venecia, 2017.

[23] G. E. Hinton y R. Salakhutdinov, «Reducing the Dimensionality of Data with Neural Networks,» *Science,* vol. 313, nº 5786, pp. 504-507, 2006.

[24] D. Foster, Generative Deep Learning, O'Reilly, 2019.

[25] A. Stellman y J. Greene, Learning Agile, O'Reilly, 2014.

[26] P. S. Efraimidis, Spirakis y P. G., «Weighted random sampling with a reservoir,» *Information Processing Letters,* vol. 97, nº 5, pp. 181-185, 2006.

[27] Tatoeba, «Tatoeba,» [En línea]. Available: https://tatoeba.org/.

[28] A. Conneau, K. Khandelwal, N. Goyal, V. Chaudhary, G. Wenzek, F. Guzmán, E. Grave, M. Ott, L. Zettlemoyer y V. Stoyanov, «Unsupervised Cross-lingual Representation Learning at Scale,» *Association for Computational Linguistics (ACL),* vol. 58, pp. 8440-8451, 2020.

[29] SuperMemo, «Algorithm SM-17,» 2014-2016. [En línea]. Available: https://supermemo.guru/wiki/Algorithm_SM-17.

8. ANEXOS

8.1. Tablas de *kana*

En este anexo se adjuntan diversas tablas de caracteres de los dos silabarios del japonés. Estas son referenciadas y explicadas a lo largo de la sección 2.1.1.

Tabla 2. Caracteres del *hiragana*.

	H◌	K◌	M◌	N◌	R◌	S◌	T◌	W◌	Y◌
◌A	あ A	は HA	か KA	ま MA	な NA	ら RA	さ SA	た TA	わ WA	や YA
◌E	え E	へ HE	け KE	め ME	ね NE	れ RE	せ SE	て TE		
◌I	い I	ひ HI	き KI	み MI	に NI	り RI	し SHI	ち CHI		
◌O	お O	ほ HO	こ KO	も MO	の NO	ろ RO	そ SO	と TO	を WO	よ YO
◌U	う U	ふ FU	く KU	む MU	ぬ NU	る RU	す SU	つ TSU		ゆ YU
				ん -N						

Tabla 3. Caracteres del *hiragana* modificables con *dakuten* y *handakuten*.

	B◌	P◌	G◌	Z◌	D◌
◌A	ば	ぱ	が	ざ	だ
	BA	PA	GA	ZA	DA
◌E	べ	ぺ	げ	ぜ	で
	BE	PE	GE	ZE	DE
◌I	び	ぴ	ぎ	じ	ぢ
	BI	PI	GI	JI	DJI
◌O	ぼ	ぽ	ご	ぞ	ど
	BO	PO	GO	ZO	DO
◌U	ぶ	ぷ	ぐ	ず	づ
	BU	PU	GU	ZU	DZU

Tabla 4. Dígrafos del *hiragana*.

	H Y◌	K Y◌	M Y◌	N Y◌	R Y◌	S H◌	C H◌
◌A	ひゃ	きゃ	みゃ	にゃ	りゃ	しゃ	ちゃ
	HYA	KYA	MYA	NYA	RYA	SHA	CHA
◌O	ひょ	きょ	みょ	にょ	りょ	しょ	ちょ
	HYO	KYO	MYO	NYO	RYO	SHO	CHO
◌U	ひゅ	きゅ	みゅ	にゅ	りゅ	しゅ	ちゅ
	HYU	KYU	MYU	NYU	RYU	SHU	CHU

	B Y◌	P Y◌	G Y◌	J◌	D J◌
◌A	びゃ	ぴゃ	ぎゃ	じゃ	ぢゃ
	BYA	PYA	GYA	JA	DJA
◌O	びょ	ぴょ	ぎょ	じょ	ぢょ
	BYO	PYO	GYO	JO	DJO
◌U	びゅ	ぴゅ	ぎゅ	じゅ	ぢゅ
	BYU	PYU	GYU	JU	DJU

Tabla 5. Caracteres del *katakana*.

	H◌	K◌	M◌	N◌	R◌	S◌	T◌	W◌	Y◌
◌A	ア	ハ	カ	マ	ナ	ラ	サ	タ	ワ	ヤ
	A	HA	KA	MA	NA	RA	SA	TA	WA	YA
◌E	エ	ヘ	ケ	メ	ネ	レ	セ	テ		
	E	HE	KE	ME	NE	RE	SE	TE		
◌I	イ	ヒ	キ	ミ	ニ	リ	シ	チ		
	I	HI	KI	MI	NI	RI	SHI	CHI		
◌O	オ	ホ	コ	モ	ノ	ロ	ソ	ト	ヲ	ヨ
	O	HO	KO	MO	NO	RO	SO	TO	WO	YO
◌U	ウ	フ	ク	ム	ヌ	ル	ス	ツ		ユ
	U	FU	KU	MU	NU	RU	SU	TSU		YU
					ン					
					-N					

Tabla 6. Caracteres del *katakana* modificables con *dakuten* y *handakuten*.

	B◌	P◌	G◌	Z◌	D◌
◌A	バ	パ	ガ	ザ	ダ
	BA	PA	GA	ZA	DA
◌E	ベ	ペ	ゲ	ゼ	デ
	BE	PE	GE	ZE	DE
◌I	ビ	ピ	ギ	ジ	ヂ
	BI	PI	GI	JI	DJI
◌O	ボ	ポ	ゴ	ゾ	ド
	BO	PO	GO	ZO	DO
◌U	ブ	プ	グ	ズ	ヅ
	BU	PU	GU	ZU	DZU

Tabla 7. Dígrafos del *katakana*.

	H Y◌	K Y◌	M Y◌	N Y◌	R Y◌	S H◌	C H◌
◌A	ヒャ	キャ	ミャ	ニャ	リャ	シャ	チャ
	HYA	KYA	MYA	NYA	RYA	SHA	CHA
◌O	ヒョ	キョ	ミョ	ニョ	リョ	ショ	チョ
	HYO	KYO	MYO	NYO	RYO	SHO	CHO
◌U	ヒュ	キュ	ミュ	キュ	リュ	シュ	チュ
	HYU	KYU	MYU	NYU	RYU	SHU	CHU

	B Y◌	P Y◌	G Y◌	J◌	D J◌
◌A	ビャ	ピャ	ギャ	ジャ	ヂャ
	BYA	PYA	GYA	JA	DJA
◌O	ビョ	ピョ	ギョ	ジョ	ヂョ
	BYO	PYO	GYO	JO	DJO
◌U	ビュ	ピュ	ギュ	ジュ	ヂュ
	BYU	PYU	GYU	JU	DJU

Tabla 8. Sílabas de nueva creación en el *katakana*.

	F◌	T◌	T S◌	W◌	D◌	V◌
◌A	ファ	タ	ツァ	ワ	ダ	ヴァ
	FA	TA	TSA	WA	DA	VA
◌E	フェ	テ	ツェ	ウェ	デ	ヴェ
	FE	TE	TSE	WE	DE	VE
◌I	フィ	ティ	ツィ	ウィ	ディ	ヴィ
	FI	TI	TSI	WI	DI	VI
◌O	フォ	ト	ツォ	ヲ	ド	ヴォ
	FO	TO	TSO	WO	DO	VO
◌U	フ	トゥ	ツ		ドゥ	ヴ
	FU	TU	TSU		DU	VU

8.2. Logotipo de la aplicación.

En este apartado se adjuntan las medidas del logotipo de la aplicación en la Figura 25. Las medidas de esta se incluyen sin unidad debido a que su objetivo es mostrar la proporción entre las distintas partes y no un tamaño físico específico. También es importante matizar que no se trata de un plano técnico destinado a ingenieros sino de una ayuda para diseñadores que intenten replicar el mismo resultado.

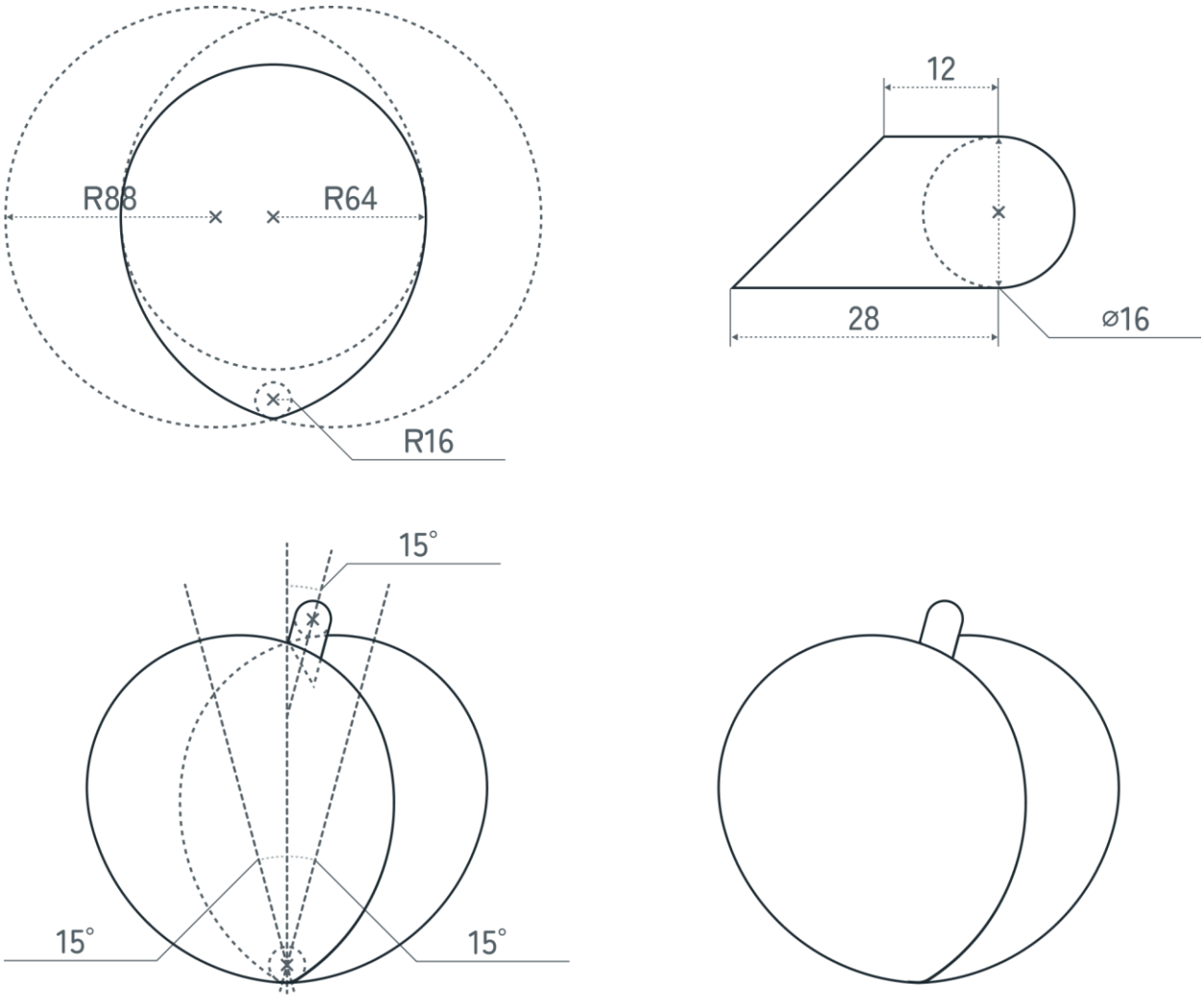

Figura 25. Medidas del logotipo de la aplicación.

8.3. Especificación de requisitos

En este anexo se recoge la especificación de requisitos del proyecto software. Como se ha comentado a lo largo de la memoria, se ha utilizado un formato de captura basado en historias de usuario. Además, por documentación, se han plasmado también en forma de diagramas de caso de uso.

A lo largo de este anexo de documentaran las historias para cada uno de los subsistemas de la aplicación, identificados en la Figura 26. Además, se plasmarán estas historias también en forma de requisitos especificados con un diagrama de casos de uso.

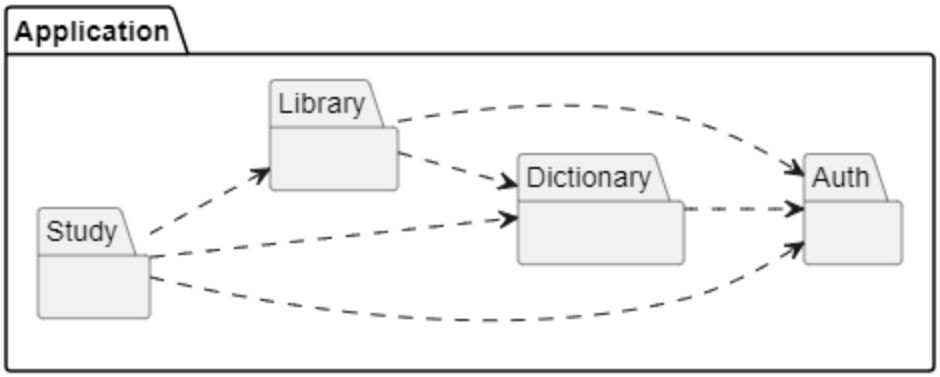

Figura 26. Diagrama de paquetes de la aplicación.

8.3.1. Roles

A continuación, en la Figura 27 se adjunta un diagrama que identifica los diversos actores del sistema: el actor invitado, el actor administrador y el actor estudiante. El actor invitado representa a usuarios que no han creado una cuenta en el sitio. Cuando un usuario se registra, pasa a ser un actor estudiante. Además, un miembro del equipo de administración de la página está representado por el actor administrador. En ocasiones aparecerá también un actor sistema que representa acciones desencadenadas automáticamente por este.

Figura 27. Actores del sistema.

8.3.2. Sistemas de autorización y autenticación

El sistema de autorización y autenticación (Auth) tiene como objetivo actuar como soporte para el resto de los sistemas de la aplicación proporcionando la infraestructura de cuentas que habilite a otros sistemas a generar contenido personalizado para cada usuario. Se han identificado las siguientes historias de usuario asociadas a este sistema:

1. Como un estudiante, me gustaría poder registrar una cuenta en la aplicación para poder sincronizar mi progreso de estudio en distintos dispositivos como el móvil y la tablet.
2. Como un estudiante, me gustaría poder identificarme con mi cuenta en múltiples dispositivos para sincronizar mi proceso.
3. Como un estudiante preocupado por la seguridad, me gustaría que mi contraseña se guardase de manera segura para evitar que pueda ser vulnerada fácilmente en caso de filtraciones de datos.
4. Como un estudiante olvidadizo, me gustaría poder recuperar mi contraseña cuando se me olvida para poder seguir accediendo, incluso en las ocasiones en las que no la recuerdo.
5. Como un estudiante, me gustaría cerrar sesión cuando estoy usando un ordenador público, como los de la biblioteca, para que terceras personas no tengan acceso a mi cuenta.
6. Como estudiante de la aplicación desde hace tiempo, me gustaría poder cambiar el correo electrónico para poder utilizar el que uso habitualmente y no el que usaba en el momento del registro.
7. Como un estudiante cuya contraseña ha sido filtrada recientemente (en esta u otra página), me gustaría poder cambiarla por otra distinta para que mi cuenta siga siendo segura.

8. Como un estudiante que usa la aplicación con frecuencia, me gustaría no tener que identificarme continuamente y que mis credenciales se queden guardados al menos por un tiempo para poder acceder más fácilmente en la aplicación.

Los requisitos identificados por estas se describen en forma de diagrama de casos de uso en la Figura 28.

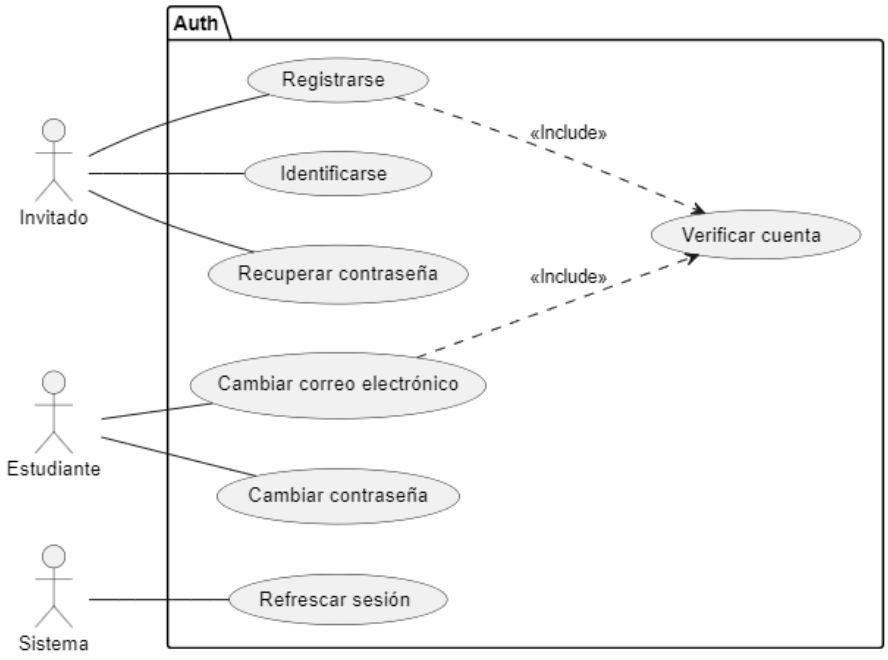

Figura 28. Diagrama de casos de uso del sistema de autenticación y autorización.

8.3.3. Sistema de diccionario

El subsistema de diccionario tiene como función actuar como repositorio de los contenidos utilizados por los sistemas de librería y estudio. Además, se pretende mantener la mayor parte de contenidos en este sistema públicos y que actúen como gancho de usuarios que luego conviertan a usuarios registrados en la aplicación. Se han identificado las siguientes historias:

9. Como el administrador de la aplicación con conocimientos en informática, me gustaría poder añadir *kanjis* sin tener que acceder a la base de datos para hacerlo de una forma más fácil y rápida.

10. Como el administrador de la aplicación con conocimientos en informática, me gustaría poder añadir palabras sin tener que acceder a la base de datos para hacerlo de una forma más fácil y rápida.
11. Como un estudiante aficionado al anime, me gustaría poder buscar palabras y *kanjis* utilizando su transcripción fonética en romaji para poder encontrar el significado de palabras que oigo en los animes, pero no sé escribir.
12. Como un estudiante de japonés, me gustaría poder buscar palabras y *kanjis* en base a su transcripción fonética con katakana o hiragana para poder encontrar fácilmente palabras que he visto en clase usando transcripción fonética, pero no en su forma diccionario con *kanjis*.
13. Como un estudiante aficionado a mangas, me gustaría poder buscar palabras y *kanjis* por su forma de diccionario para poder encontrarlas más fácilmente cuando las veo en los mangas que leo.
14. Como un estudiante con amigos japoneses, me gustaría poder buscar palabras por su significado en mi lengua nativa para poder redactar mensajes que enviarles.

De nuevo, se adjunta en la Figura 29 un diagrama de casos de uso que representa los requisitos identificados por estas historias.

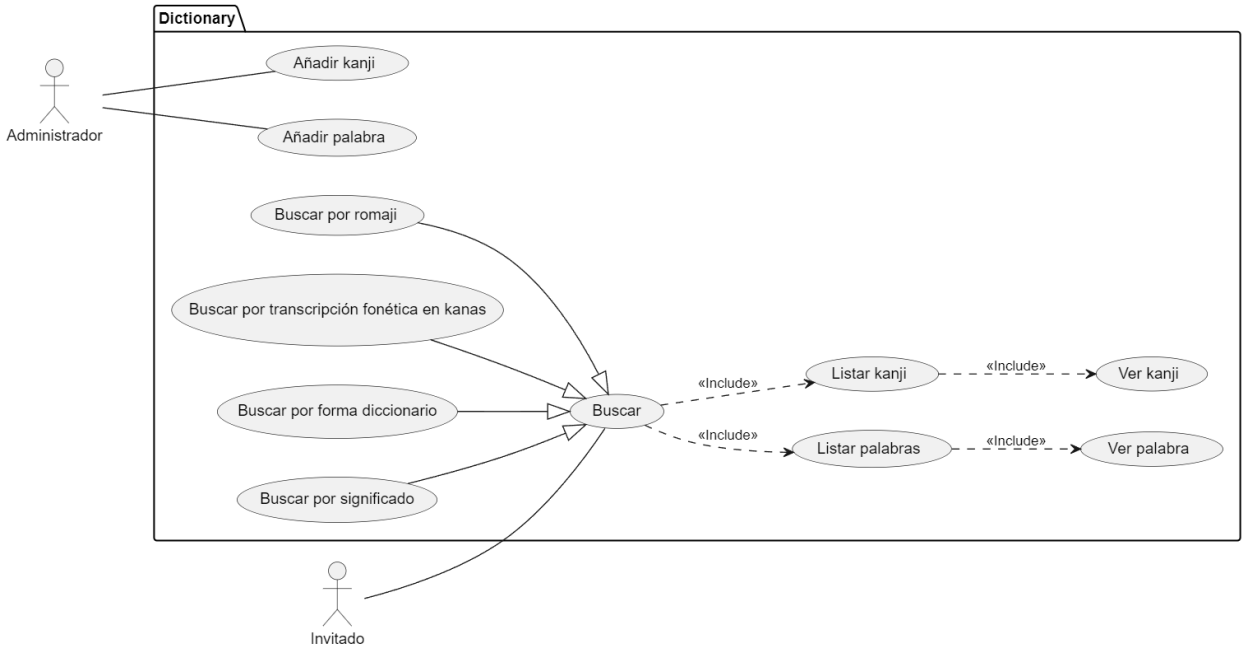

Figura 29. Diagrama de casos de uso del subsistema de diccionario.

8.3.4. Sistema de librería

Este sistema constituye la base de la organización del usuario y se basa en la metáfora de tarjetas (asociadas a una palabra o *kanji*) y barajas (grupos de tarjetas). Su objetivo es proporcionar un mecanismo para que el usuario pueda seleccionar los conceptos que le interesan dentro de la totalidad de conceptos existentes en el diccionario. Se han identificado las siguientes historias de usuario:

15. Como un estudiante que acude a clases de japonés, me gustaría poder crear barajas de *kanjis* y palabras que se correspondan con los temarios del curso que estudio para poder estudiar más fácilmente para las pruebas del curso.
16. Como un estudiante que usa la aplicación con frecuencia, me gustaría poder eliminar algunas barajas que ya no me interesan para tener mi cuenta más limpia y ordenada.
17. Como un estudiante, me gustaría poder añadir palabras a barajas después de crearlos para poder adaptarlos en caso de que se dé algo más de temario en clase.
18. Como un estudiante, me gustaría poder añadir *kanjis* a barajas después de crearlos para poder adaptarlos en caso de que se dé algo más de temario en clase.
19. Como un estudiante, me gustaría poder eliminar palabras de barajas cuando ya no me interesen para evitar que se me acumulen.
20. Como un estudiante, me gustaría poder eliminar *kanjis* de barajas cuando ya no me interesen para evitar que se me acumulen.
21. Como un estudiante, me gustaría poder cambiar el nombre de una baraja para corregir errores tipográficos que pueda cometer al escribir.
22. Como un estudiante, me gustaría ver una lista de las barajas que tengo para saber cuántas y cuáles tengo.
23. Como un estudiante, me gustaría poder ver el detalle de una baraja para saber qué contenidos tiene.
24. Como un estudiante, me gustaría poder ver el detalle de las palabras pertenecientes a la baraja para poder estudiarlas.
25. Como un estudiante, me gustaría poder ver el detalle de los *kanjis* pertenecientes a la baraja para poder estudiarlos.

En la Figura 30 se representa el diagrama de casos de usos de los requisitos especificados por medio de estas historias de usuario. Dentro de este también se incluye parte del subsistema de bibliotecas para mostrar las relaciones entre ambos.

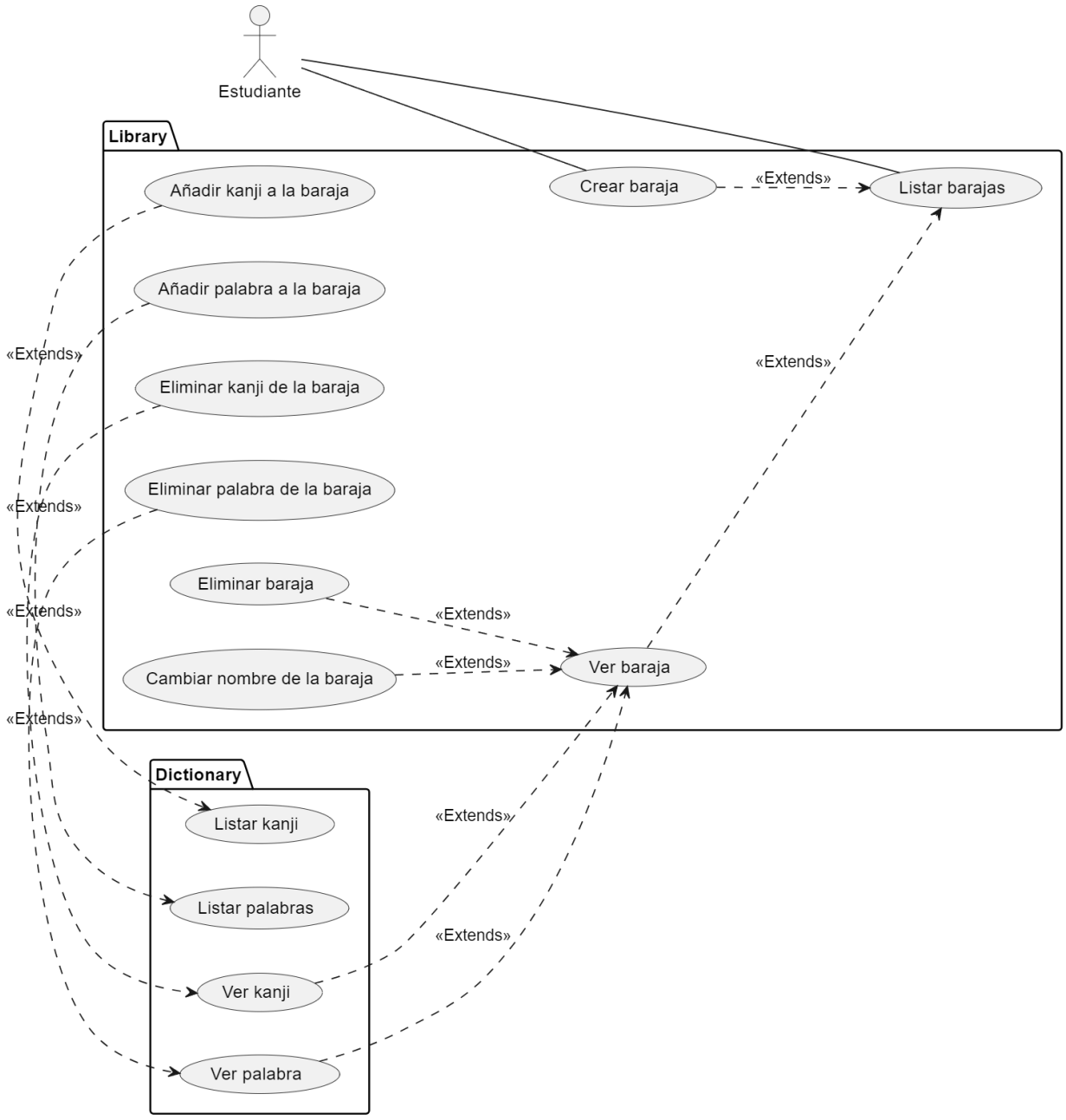

Figura 30. Diagrama de casos de uso del subsistema de biblioteca.

8.3.5. Sistema de estudio

El sistema de estudio es el principal de la aplicación. Su objetivo es proporcionar al usuario un mecanismo para estudiar los conceptos almacenados en sus barajas de una forma eficiente. En este subsistema se han encontrado las siguientes historias de usuario:

26. Como estudiante, me gustaría que la aplicación me generara preguntas de revisión de los diferentes *kanjis* y vocabulario en mis barajas para evitar olvidarlos.
27. Como estudiante, me gustaría que la aplicación aprendiera de mis aciertos y errores y lo tuviese en cuenta a la hora de decidir la frecuencia con que mostrar conceptos para tener una técnica de estudio más eficiente y tener más tiempo para otras cosas.
28. Como estudiante, me gustaría que la aplicación me ayude a descubrir patrones entre *kanji* que inicialmente pueden ser difíciles de distinguir (porque son similares) para poder generar modelos mentales más robustos.
29. Como estudiante, me gustaría que el progreso de estudio de un concepto (palabra o kanji) sea compartido por todas las barajas para no tener que repetir el proceso de estudio de un concepto que ya conozco.

Estas historias se convierten en requisitos que se representan en el diagrama de casos de uso de la Figura 31.

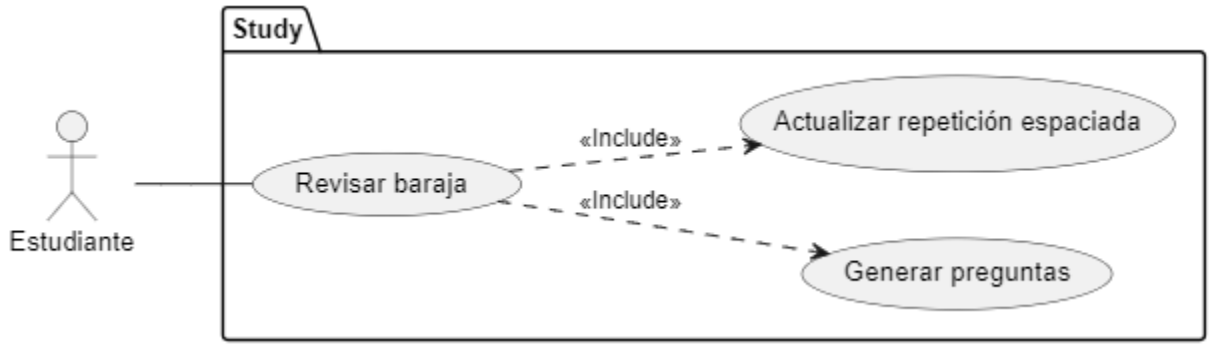

Figura 31. Diagrama de casos de uso del subsistema de estudio.

8.4. Paquetes de análisis

En este apartado se muestran los distintos paquetes del sistema y las clases que forman parte de ellos junto a sus relaciones. Estas clases se anotan con los estereotipos <<Boundary>>, <<Control>> o <<Entity>> en función de las responsabilidades que cumplan en la realización de casos de uso.

La aplicación está divida en los paquetes descritos en la Figura 32 y que realizan las siguientes funciones:

- Auth: responsable de proporcionar el subsistema de autenticación y autorización al resto de módulos. Las clases pertenecientes a este paquete se muestran en la Figura 33.
- Dictionary: responsable de gestionar las *palabras* y los *kanjis* de la aplicación. Una de sus principales funciones es la búsqueda de palabras. Sus clases se muestran en la Figura 34.
- Library: responsable de gestionar los mecanismos de organización de los conceptos del diccionario para el usuario. En este paquete es donde existen las tarjetas y las barajas. Su composición se delinea en la Figura 35.
- Study: responsable de gestionar la información y algoritmos de estudio. Su estructura se especifica en la Figura 36.

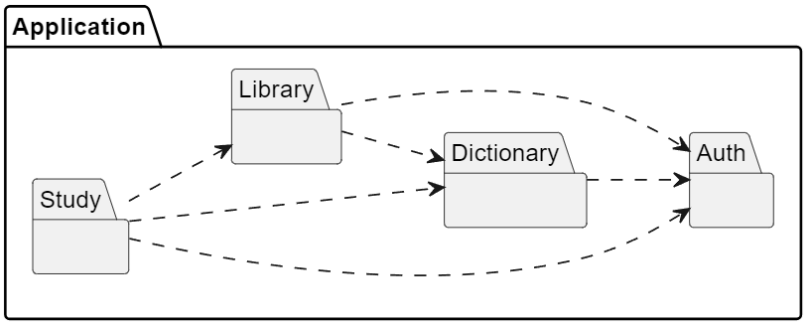

Figura 32. Paquetes de análisis.

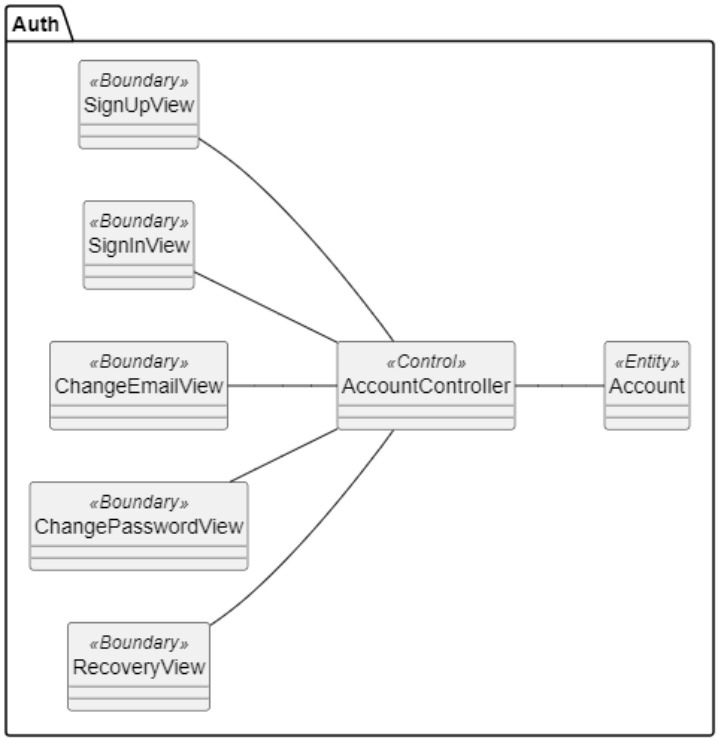

Figura 33. Paquete Auth análisis.

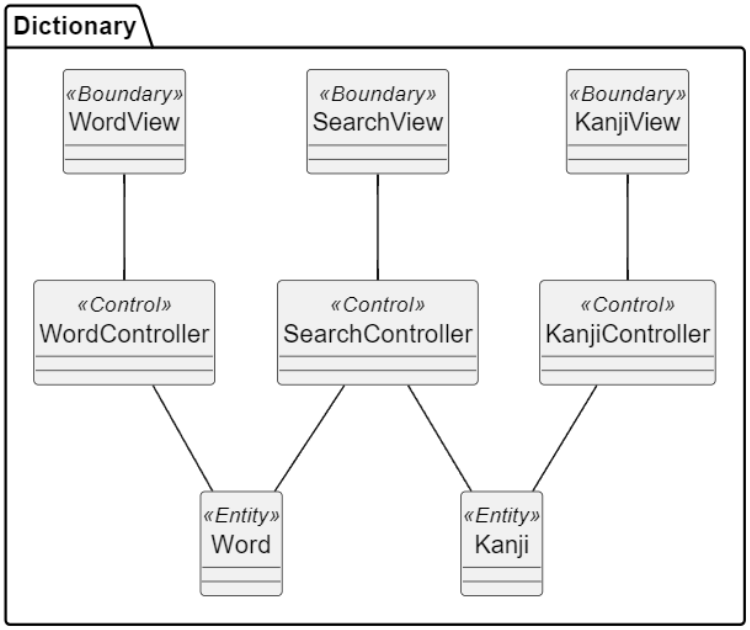

Figura 34. Paquete Dictionary de análisis.

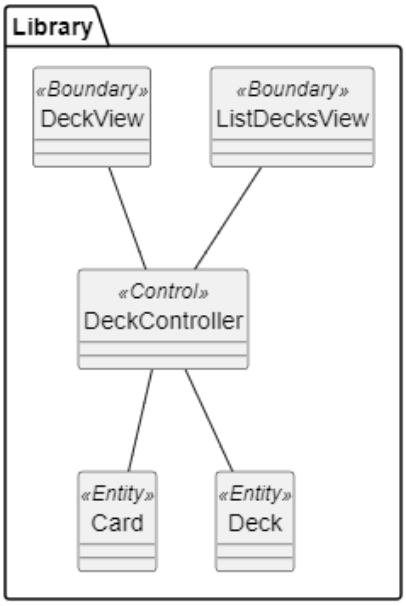

Figura 35. Paquete Library de análisis.

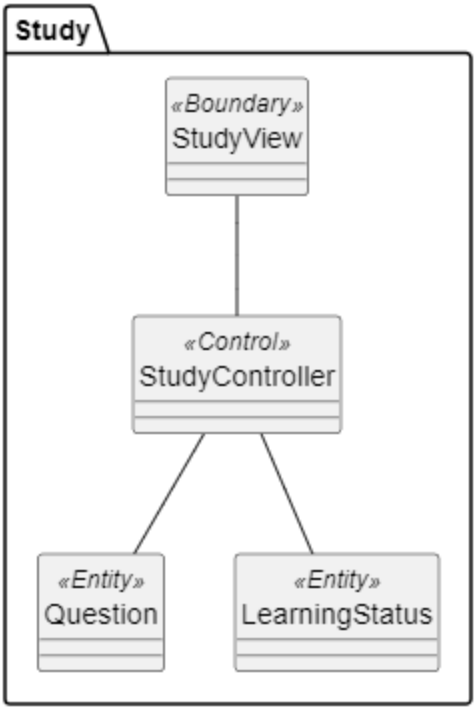

Figura 36. Paquete Study de análisis.

8.5. Documentación técnica

Este trabajo está estructurado en tres proyectos diferentes. El primero de ellos, "momokado-web", es la aplicación web desarrollada con React. El segundo de ellos, "momokado-api", es el servidor de la API desarrollado en NestJS. Y, el tercero de ellos, "momokado-embeddings", contiene el código de los modelos de *embeddings* desarrollados, así como los experimentos y los cuadernos Jupyter utilizados para analizar los resultados. Junto a esta memoria, en una carpeta llamada "Código" se adjunta un fichero ZIP para cada uno de ellos. Además, en una carpeta llamada "Documentación Técnica" se incluye la documentación del cliente y del servidor generada automáticamente con TypeDoc a partir de los comentarios de documentación.

El código del cliente web y el servidor está estructurado de una manera similar. Ambos tienen una carpeta "src" dentro de la cuál existen varias subcarpetas, una por cada paquete de la aplicación. Dentro de las carpetas asociadas a cada paquete puede haber varias subcarpetas que agrupan componentes por funcionalidad. Por ejemplo, las más comunes serían:

- "controllers": los controladores HTTP.
- "dtos": objetos de transferencia de datos utilizados en los controladores.
- "models": modelos.
- "repositories": interfaces e implementaciones del patrón repositorio.
- "services": servicios de aplicación.
- "api": llamadas a la API desde el cliente.
- "components": componentes puros.
- "views": vistas de la aplicación. Los controladores de las vistas se guardan junto a ellas en esta carpeta.

Además, es útil conocer la existencia de múltiples comandos para realizar diferentes tareas sobre los proyectos. En concreto:

- "npm install": instala las dependencias del proyecto.
- "npm run start:dev": ejecuta el proyecto en modo desarrollo.
- "npm run build": compila el proyecto.
- "npm run start": ejecuta el proyecto.

- “npm run storybook:start”: ejecuta Storybook una herramienta utilizada para hacer *debug* de componentes (sólo en el cliente).
- “npm run test:e2e”: ejecuta las pruebas del servidor API.

El proyecto de *embeddings* es algo más plana y no requiere demasiada explicación. Dentro de cada una de las carpetas se encuentra uno de los proyectos de *embedding* y, en la raíz, los cuadernos utilizados para interpretar los resultados.

8.6. Características de la aplicación

La experiencia de un usuario de Momokado comienza en la página de búsqueda de la Figura 37, donde se le recibe con una serie de textos que pretenden describir cómo funciona la aplicación y actuar como gancho para el registro que, en realidad, es la acción primaria de esta pantalla.

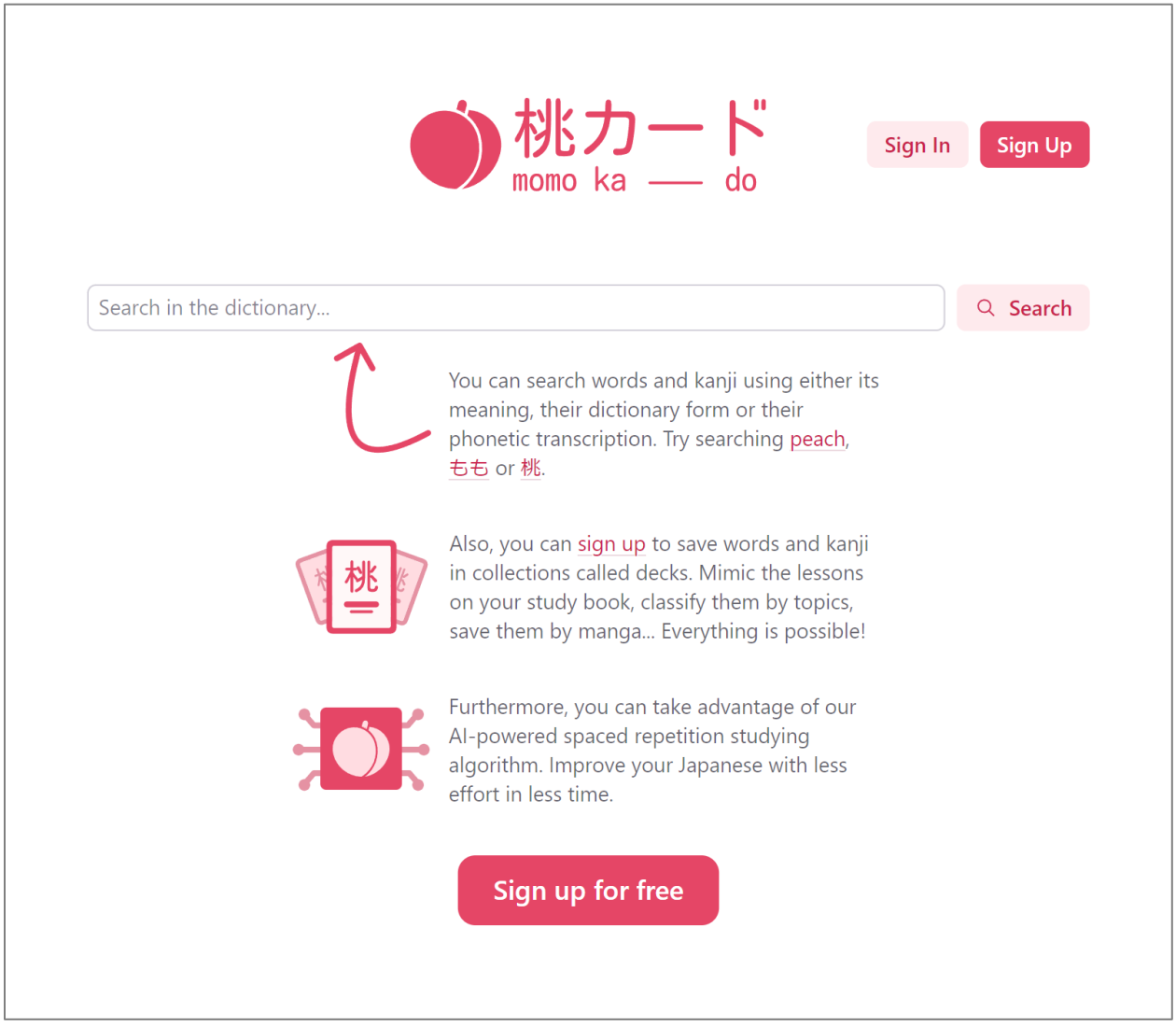

Figura 37. Pantalla de búsqueda de Momokado.

Si todo sale bien, el usuario decidirá registrarse para lo que accederá a la página de registro a partir de alguno de los botones de la pantalla anterior. Llegados a este punto, se le mostrará el formulario de la Figura 38. Aquí, el usuario deberá introducir sus datos y presionar el botón de continuar.

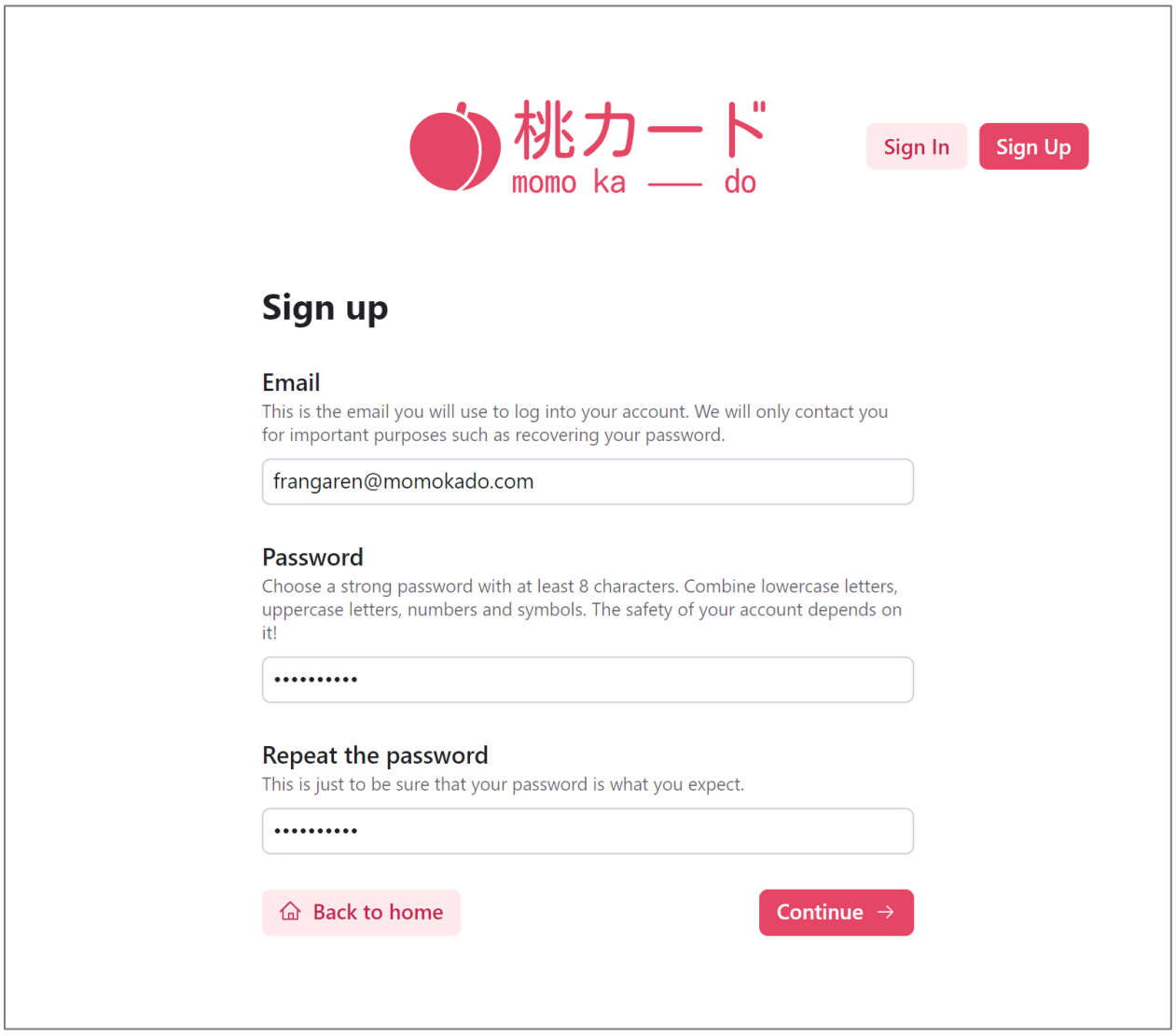

Figura 38. Pantalla de registro de Momokado.

Tras presionar el botón de continuar se le llevará a una página que les indica que se ha enviado un correo electrónico de validación y deben seguir las instrucciones para poder

continuar. Una vez se acceda al enlace del correo electrónico, se avisará al usuario de que su cuenta ha sido confirmada y se le llevará a la pantalla de identificación.

En esta pantalla de identificación, que se puede ver en la Figura 39, el usuario podrá escribir los datos de su cuenta y acceder. En caso de que no recuerde alguno de ellos podrá entrar en el proceso de recuperación de contraseña se le pedirá al usuario su correo electrónico y se le enviará un enlace con las instrucciones y que redireccionará a una página que permite el cambio de contraseña.

Continuando con el caso de una identificación correcta, el usuario volverá de nuevo a la pantalla de búsqueda, pero, esta vez la acción primaria (y, por tanto, destacada con el botón más llamativo) será la búsqueda.

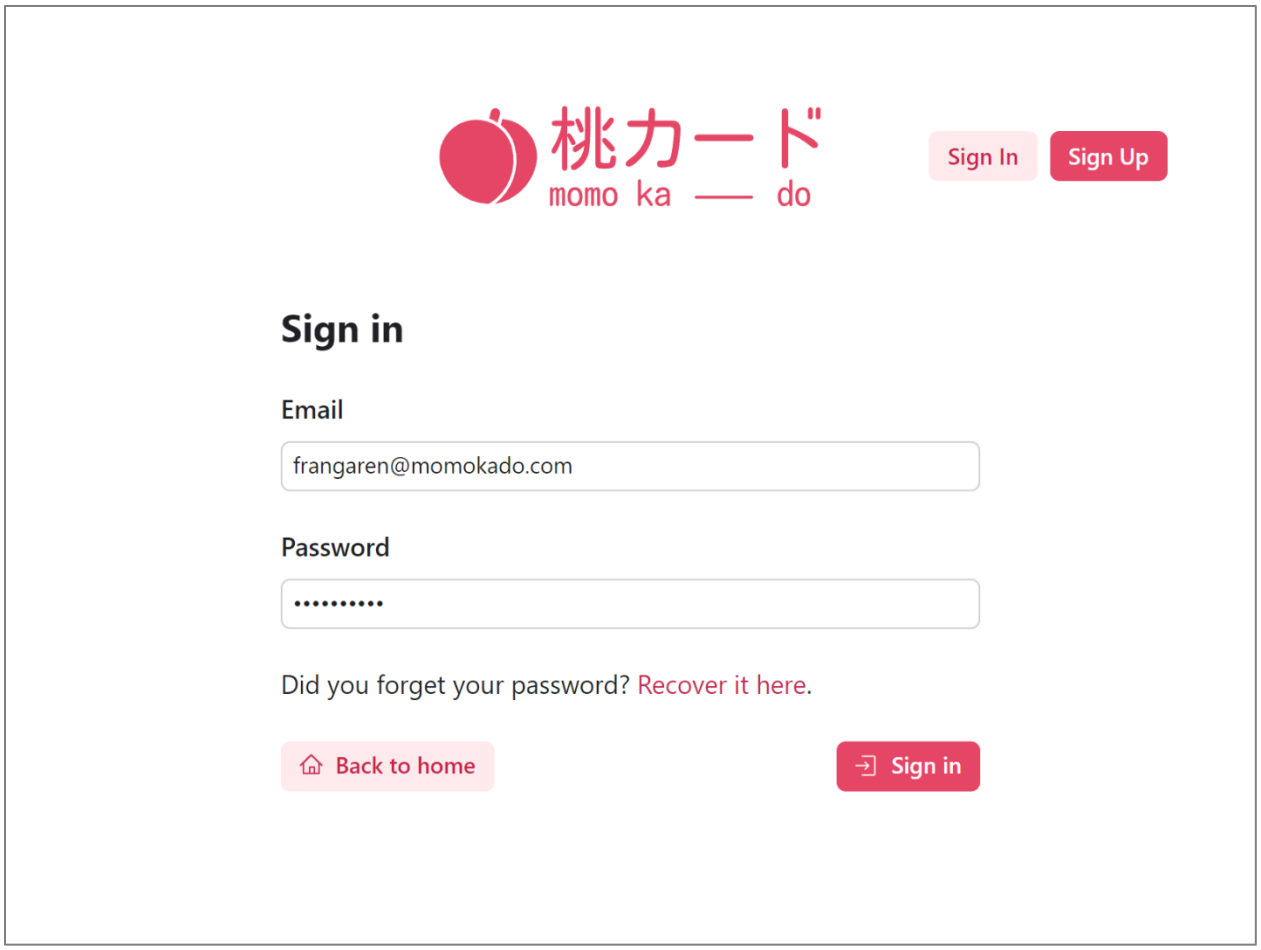

Figura 39. Pantalla de identificación de Momokado.

Existen varios métodos de búsqueda en Momokado y ya han sido comentados en otras partes de la memoria (por transcripción en romaji, *hiragana* o *katakana*; por forma diccionario o por significado). En este caso, por ejemplo, se muestra la pantalla resultante de buscar la oración “日曜日は火曜日じゃありません。” (el domingo no es martes) en la Figura 40. Como se puede ver, la frase aparece dividida por palabras en la parte superior de la pantalla. Cada una de ellas tiene unos resultados de búsqueda independientes. Debajo de la frase aparecen otras pestañas con la opción de ver las palabras y de ver los *kanjis* encontrados.

Figura 40. Pantalla de resultados de búsqueda.

En este punto, el usuario puede decidir que esa palabra le resulta útil, para lo que puede tocar sobre el icono del corazón. En ese momento, se abrirá un diálogo que permite seleccionar en qué barajas colocarla. En este caso no hay ninguno, así que sólo mostrará la opción de crear como en la Figura 41, pero si se tuviese alguno se mostraría la Figura 42. En cualquier caso, se pueden marcar las barajas que se desee para añadir el kanji a ellas.

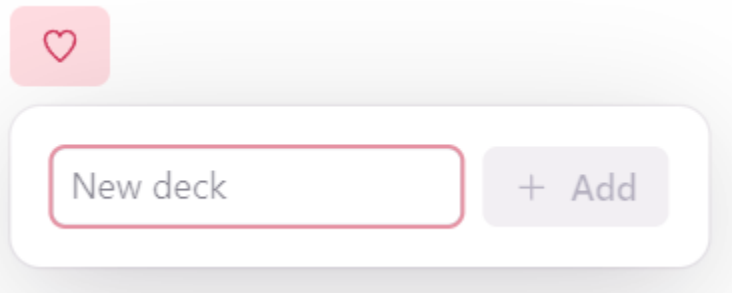

Figura 41. Diálogo de añadir una palabra a una baraja cuando no hay ninguna existente.

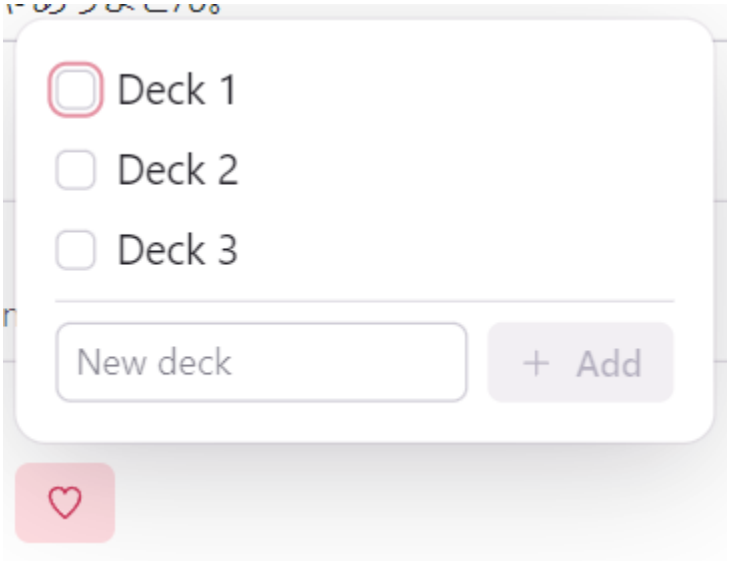

Figura 42. Diálogo de añadir una palabra a una baraja cuando hay varias existentes.

Tras esto el usuario podría acceder al apartado de sus barajas con el botón “My decks” que se encuentra al lado de su perfil. En esta pantalla, como se ve en la Figura 43, se muestran

las barajas con un estilo esqueumórfico junto a un botón que permite añadir una nueva. Tocar sobre cualquiera de ellas llevará a la pantalla de cartas de la baraja.

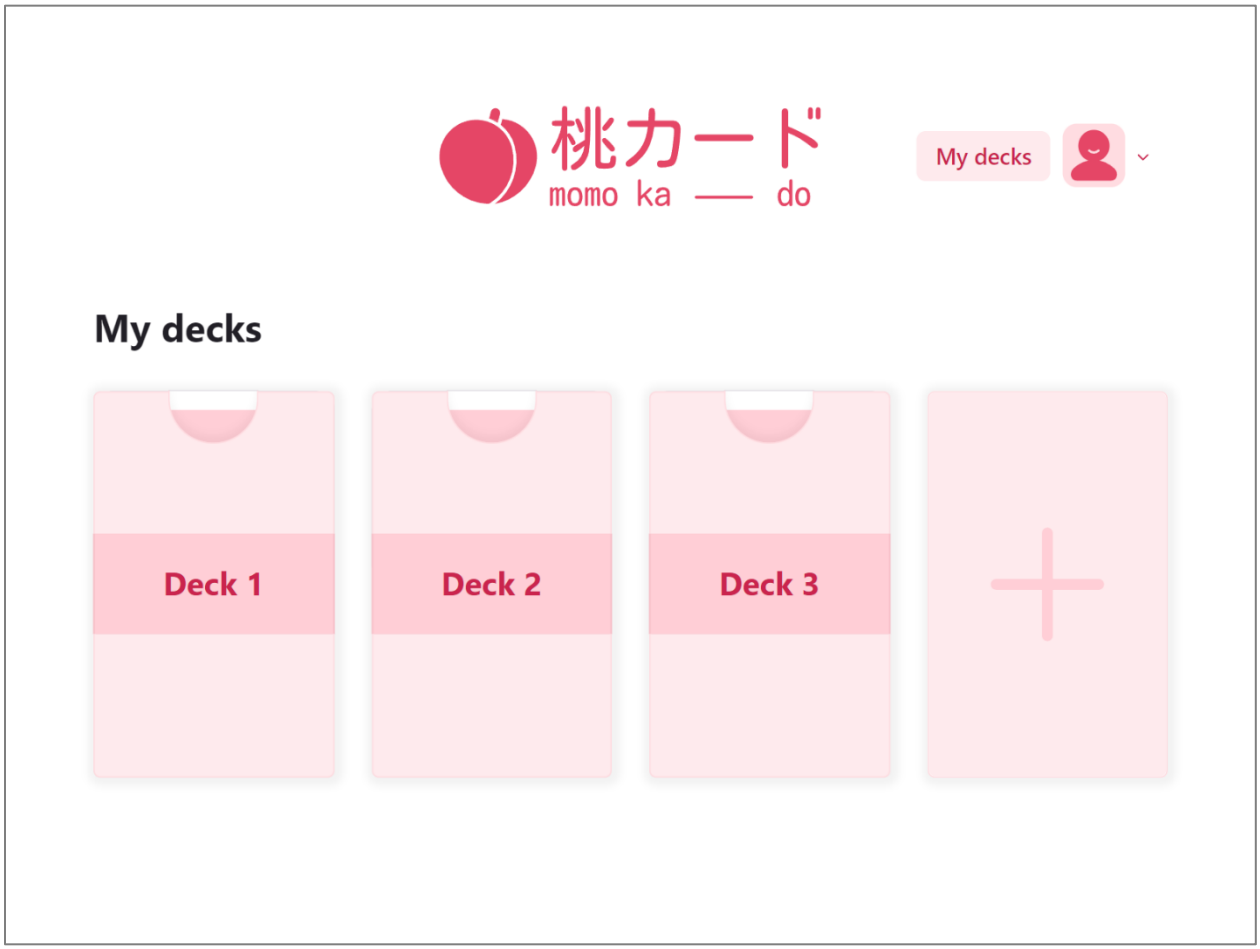

Figura 43. Pantalla de barajas.

En la pantalla de barajas, que es la que se muestra en la Figura 44, se pueden realizar bastantes funciones. En primer lugar, se puede renombrar la baraja presionando sobre el lápiz que hay a la derecha del nombre. En segundo lugar, se puede eliminar presionando sobre el botón de la parte inferior. En tercer lugar, se puede ver el detalle de cualquiera de las cartas presionando sobre ellas. En la pantalla del detalle de carta se mostrará su significado de manera similar a como se ve en el diccionario y, además, se permitirá borrarla

de la baraja. Pero, sin duda, la acción más importante es estudiar esa baraja. Para ello se puede presionar sobre el botón destacado de la parte superior.

Figura 44. Pantalla de detalles de la baraja.

Al entrar en la página de estudio se mostrarán una serie de preguntas sobre los conceptos pendientes de estudio para ese día. Como se ha mencionado previamente, las preguntas pueden ser de traducción directa (japonés a inglés) o inversa (inglés a japonés). Una vez respondidas, las preguntas muestran la respuesta correcta resaltada en verde y las incorrectas en rojo. En las figuras Figura 45 y Figura 46 se muestran ejemplos de preguntas.

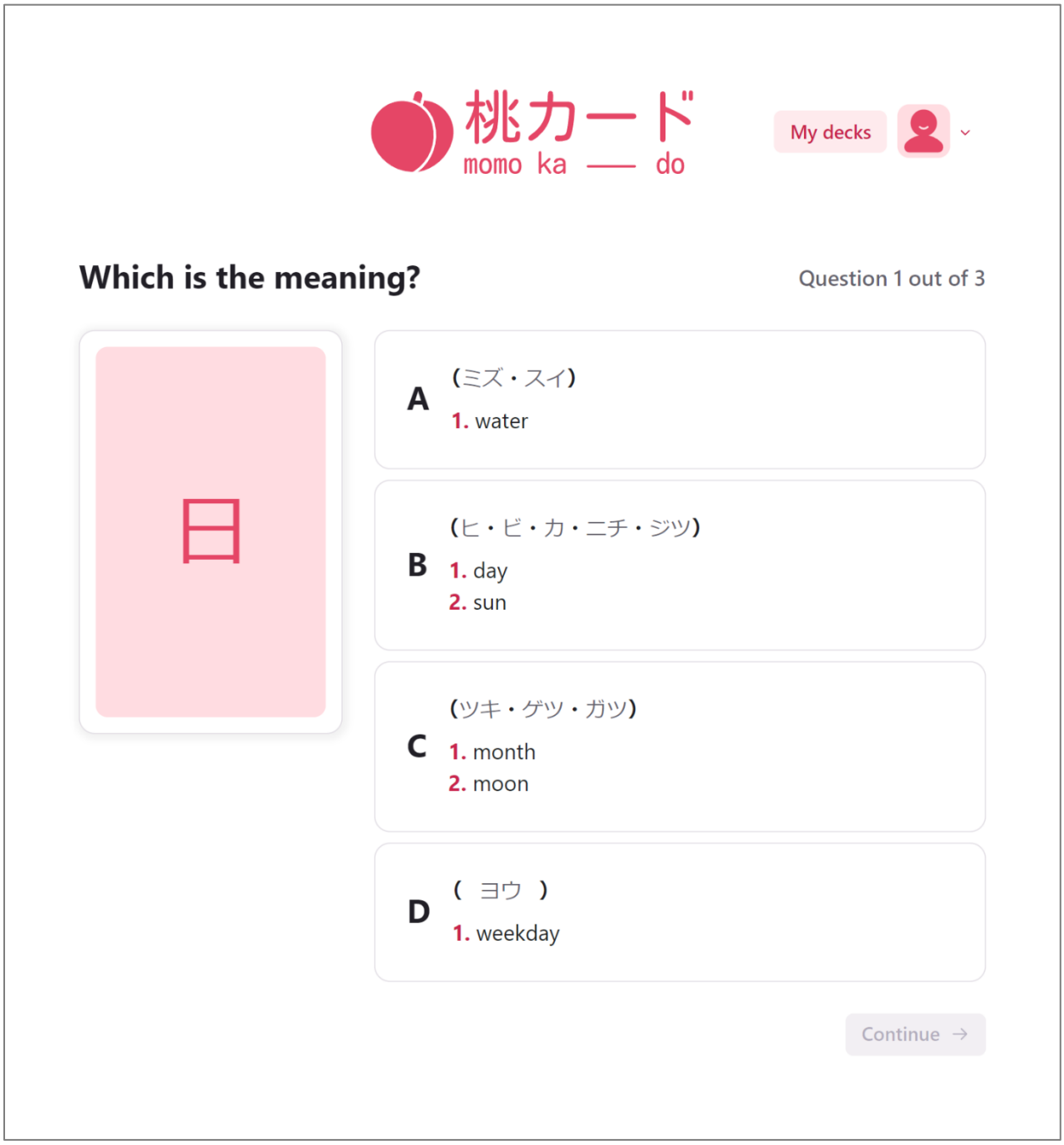

Figura 45. Pantalla de pregunta de traducción directa sobre un *kanji* antes de ser respondida.

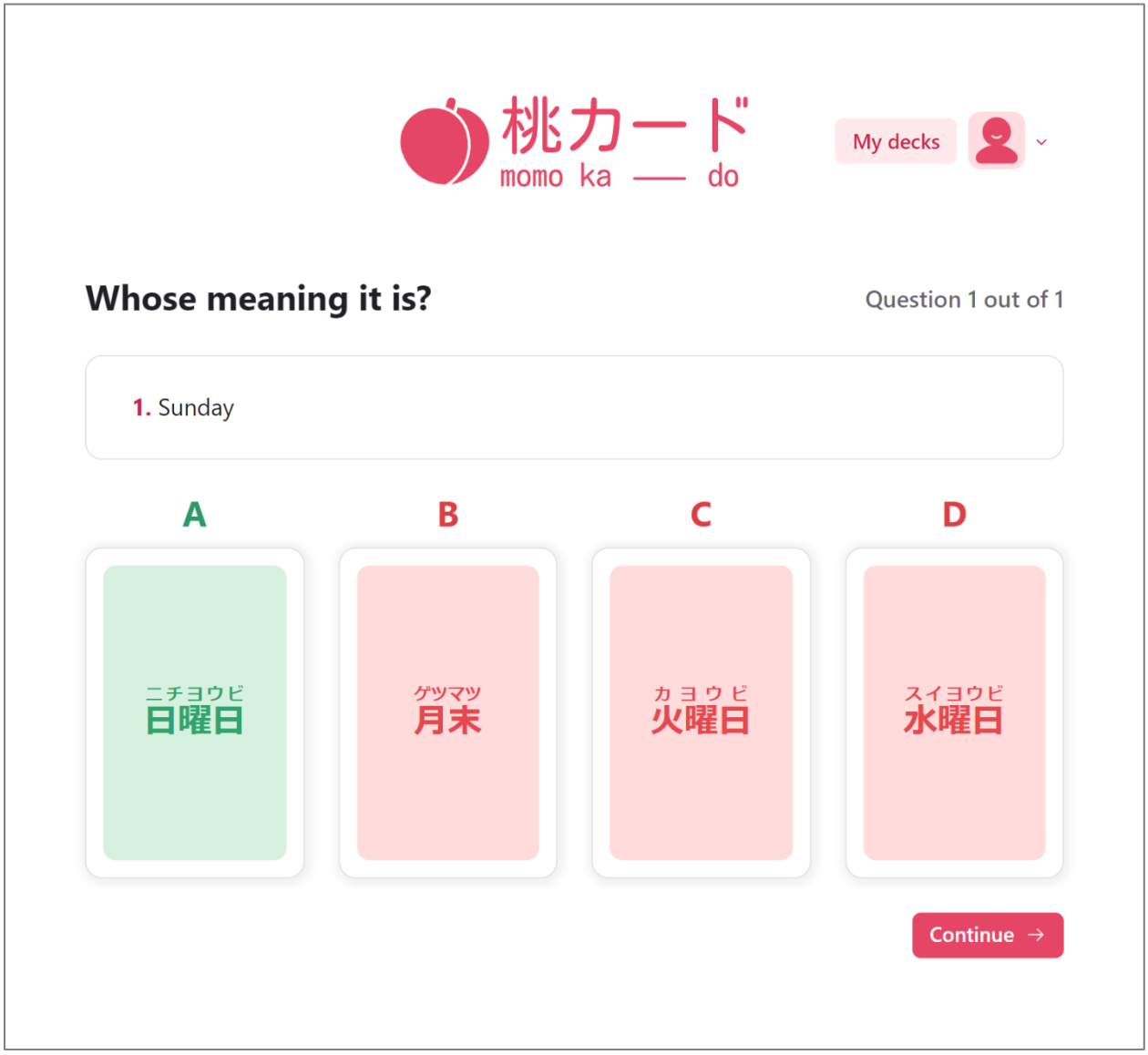

Figura 46. Pantalla de traducción inversa sobre una palabra tras ser respondida.

El usuario podrá pasar a la siguiente pregunta en cualquier punto a partir de responderla. Además, inmediatamente, se notificará al servidor de si la respuesta que ha dado era correcta o no, y la dificultad que ha tenido para darla. En base a esos criterios y revisiones anteriores el algoritmo de repetición espaciada le asignará una nueva fecha de revisión en el futuro. Pero antes, una vez se termine la revisión de las tarjetas, se le felicitará al estudiante por el trabajo realizado con la pantalla de la Figura 47.

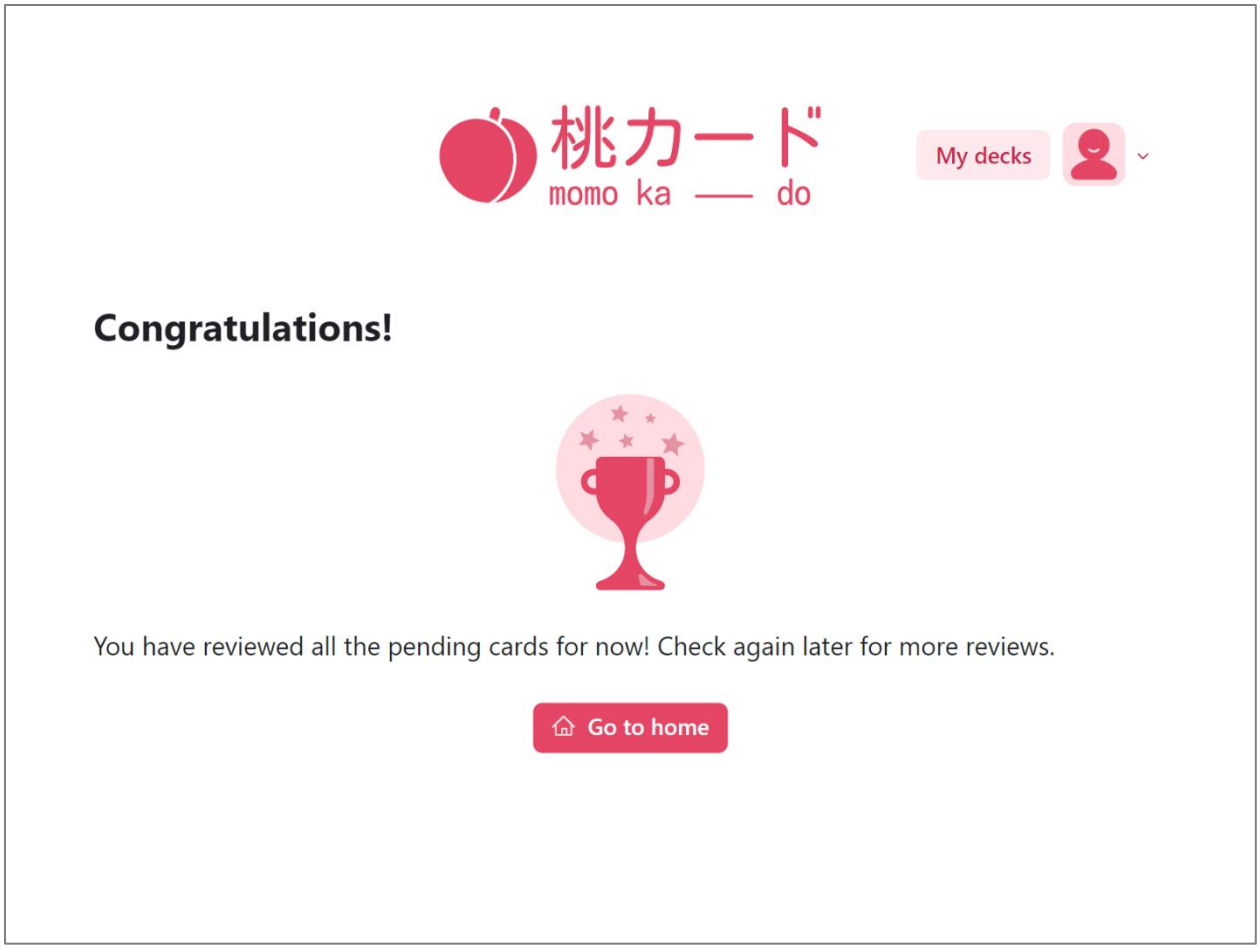

Figura 47. Pantalla de fin de estudio.

Por último, mencionar la existencia de un menú al presionar sobre la foto de perfil que permite cambiar la contraseña, el correo electrónico y cerrar la sesión.