Universidad Autónoma de Madrid

Escuela Politécnica Superior

Máster en Ingeniería Informática

TRABAJO DE FIN DE MÁSTER

CRAWLERS BASADOS EN REINFORCEMENT LEARNING

Adrián Pertejo Mangas
Co-tutor: Pablo Castells Azpilicueta
Co-tutor: Alejandro Bellogin Kouki

24 de enero de 2021

CRAWLERS BASADOS EN REINFORCEMENT LEARNING

Autor: Adrián Pertejo Mangas
Co-tutor: Pablo Castells Azpilicueta
Co-tutor: Alejandro Bellogin Kouki

Departamento de Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid

24 de enero de 2021

Abstract

Abstract — In recent years, the field of Reinforcement Learning (RL) has acquired significant relevance in multiple areas of study. Through the proposal of an interactive learning paradigm, it seeks to model the idea of natural learning that a living being has in relation to its environment. In the literature, work has begun on the integration of this new area with classic Machine Learning problems, and among them, web crawling is no exception.

A web crawler is a tool that walks the Web getting pages automatically along the way. A special type of crawler is the focused crawlers, which seek to obtain pages of a certain topic. It is here where we integrate the RL techniques, in order to focus the crawler more precisely towards relevant pages to the chosen topic.

During this project, considerable work has been done in the design and development of three crawler proposals: a basic one, which does not use RL strategies, and two based on MDP resolution algorithms (Markov Decision Processes, one of the main RL models): Linear Function Approximation and Deep Q-Learning, respectively.

To implement these crawlers we will need to represent the pages as vectors, either with Word2Vec models to calculate relevance measures with similarity functions, or extracting some features designed specifically for this problem. In addition, as input data sets, public collections of Wikipedia articles will be used.

Once the crawlers have been developed, an extensive experimental study will be carried out, with the aim of evaluating the effectiveness of our crawlers, comparing different ways of implementing them, and the differences between them in terms of relevance discovery.

Key words — Information Retrieval, Web Minery, Web Crawlers, Word2Vec, Reinforcement Learning, Linear Function Approximation, Q-Learning, Deep Learning.

Resumen

Resumen — En los últimos años, el campo del Aprendizaje por Refuerzo (RL, de Reinforcement Learning) ha adquirido una relevancia notoria en múltiples áreas de estudio. Mediante la propuesta de un paradigma de aprendizaje interactivo, busca modelar la idea del aprendizaje natural que tiene un ser vivo en relación con su entorno. En la literatura, se ha empezado a trabajar en la integración de este nuevo área con problemas clásicos de Machine Learning, y entre ellos, el crawling web no es una excepción.

Un crawler web es una herramienta que recorre la Web obteniendo páginas automáticamente. Un tipo especial de crawler son los crawlers enfocados, que buscan obtener páginas de una cierta temática. Es aquí donde integramos las técnicas de RL, para orientar con mayor precisión al crawler hacia páginas relevantes a la temática deseada.

Durante este proyecto, se ha realizado un considerable trabajo de diseño y desarrollo de tres propuestas de crawlers: uno básico, que no utiliza estrategias de RL, y dos basados en algoritmos de resolución de MDPs (Procesos de Decisión de Markov, uno de los principales modelos de RL): Aproximación Lineal de Funciones y Deep Q-Learning, respectivamente.

Para implementar estos crawlers necesitaremos representar las páginas como vectores, ya sea con modelos Word2Vec para calcular relevancias con funciones de similitud, o extrayendo unas características diseñadas específicamente para este problema. Además, como conjuntos de datos de entrada, se usarán colecciones públicas de artículos de Wikipedia.

Una vez desarrollados los crawlers, se realizará un amplio estudio experimental, con el objetivo de evaluar la eficacia de nuestros crawlers, comparando diferentes formas de implementarlos, y las diferencias que hay entre ellos en términos de descubrimiento de relevancia.

Palabras clave — Recuperación de Información, Minería Web, Crawlers web, Word2vec, Aprendizaje por Refuerzo, Aproximación lineal de funciones, Q-Learning, Aprendizaje Profundo.

Índice general

Índice de tablas

Índice de figuras

1 Introducción

1.1. Motivación del proyecto

El campo del Aprendizaje por Refuerzo (RL, de Reinforcement Learning, sus siglas en inglés) ha sufrido un incremento notable en los últimos años en investigación y en aplicaciones. Surgió con la motivación de modelar computacionalmente el aprendizaje interactivo propio de los seres vivos, y aporta una formulación matemática formal al proceso interactivo entre un agente y un entorno. El primero toma acciones sobre el segundo, alterando su estado. Este paradigma puede ser aplicado a diferentes ámbitos: recomendación, vehículos inteligentes, finanzas, ó el área que nos ocupa, el crawling web. Un crawler web es una herramienta automática que recorre la Web obteniendo las páginas a su paso, ya sea sin discriminar entre ellas (crawlers universales), o centrándose en páginas de cierta temática (crawlers enfocados o temáticos). A la hora de integrar el paradigma del RL con el crawling web, apenas se han publicado propuestas en esta dirección. Es por ello que surge este trabajo, donde el crawler actúa como agente, decidiendo qué página visitar en cada paso e interactuando con el entorno que, en este caso, se trata de la propia web.

Dentro del campo del Reinforcement Learning hay varias formas de modelar este proceso interactivo entre agente y entorno, siendo la más importante y estudiada el modelo del Proceso de Decisión de Markov [1] (MDP, por sus siglas en inglés). Su principal característica es cumplir la Propiedad de Markov, que asegura que el estado actual es independiente de los anteriores. Esta propiedad permite luego aplicar algoritmos más complejos para resolver el MDP, obteniendo una función que nos indique en cada estado cuál es la mejor acción a tomar: la política.

La integración del Aprendizaje por Refuerzo con el Crawling Web no es trivial, y

en la literatura no encontramos demasiadas líneas de investigación desarrolladas. Una de las pocas propuestas [2] utiliza una Aproximación Lineal de Funciones (ALF), donde iterativamente se ajustan unos pesos, que combinados linealmente con unos vectores de características que representan a estados y acciones, obtenemos el valor de una función Q, que evalúa cómo de deseable es cada decisión. Nosotros tomamos esta propuesta como punto de partida teórico, para implementarla y centrarnos en el problema de ingeniería que afrontamos durante el desarrollo, ya que hay multitud de subsistemas que tienen que interactuar entre sí para hacer que el crawler funcione. Es importante recalcar esta idea, ya que este trabajo se desarrolla en el contexto de un Máster en Ingeniería Informática, y es por ello que aunque el tema tiene claros tintes de investigación, el objetivo es centrarse más en el trabajo de ingeniería necesario para desarrollar todos los componentes desde cero, tomando una propuesta teórica ya presentada como base.

Sin embargo, aunque el objetivo primordial del proyecto sea realizar un considerable trabajo de diseño y desarrollo, también proponemos un crawler novedoso basado en Deep Q-Learning [1] [3] [4] (DQN, por las siglas Deep Q-Network, que es el aproximador que usa este método). Este algoritmo es una variación del algoritmo clásico de RL, Q-Learning. Con este método, se utiliza una red neuronal profunda como aproximador de la función Q. El motivo de utilizar este algoritmo como propuesta adicional es múltiple. Primero, con el anterior algoritmo estamos realizando una aproximación lineal de funciones, y con la propuesta del Deep Q-Learning estamos realizando una aproximación no lineal, gracias al uso de redes neuronales profundas. Además, este trabajo se realiza tras haber desarrollado otro TFM para el Máster en I2-ICSI, donde se exploraron sistemas de recomendación basados en Deep Q-Learning, lo que lleva al autor a desear explorar su aplicación en otras áreas, en este caso, el crawling web. En ese trabajo ya se realizó una investigación más exhaustiva sobre el campo del Aprendizaje por Refuerzo, lo que nos sirve como base para profundizar en este nuevo proyecto.

Como ya hemos mencionado brevemente, existen varios tipos de crawler. Existen dos grandes grupos: los que recorren la web obteniendo todas las páginas sin distinción, como son los crawlers universales, y los que obtienen páginas que pertenecen a cierta temática. Dentro de este último grupo, existen a su vez dos tipos de crawler: enfocados y temáticos. La diferencia entre ambos radica en como calculan la prioridad de los enlaces con respecto a la temática: si realizan un aprendizaje supervisado (utilizan un conjunto de datos etiquetados), los llamamos crawlers enfocados; si utilizan un aprendizaje no supervisado, reciben el nombre de crawlers temáticos. En nuestro caso, usamos las técnicas de RL para orientar al crawler hacia páginas de una cierta temática. Además, como vamos a utilizar un aprendizaje no supervisado (utilizaremos un conjunto de datos que no está etiquetado en diferentes clases), vamos a implementar **crawlers temáticos**, ó **topical crawlers**, en inglés.

Sintetizando, este trabajo se centra en desarrollar de principio a fin dos crawlers enfocados basados en Aprendizaje por Refuerzo, uno original (basado en Deep Q-Learning), y uno ya propuesto en la literatura (basado en Aproximación Lineal de Funciones), para luego analizar el rendimiento de ambos en términos de descubrimiento de relevancia. Aunque el tema necesite de un amplio estudio previo e investigación sobre el área del RL, la realización previa del proyecto anteriormente mencionado nos permite

centrarnos en el trabajo de ingeniería realizado durante el diseño y el desarrollo de los sistemas necesarios, pudiendo orientar correctamente el trabajo hacia detalles técnicos.

1.2. Alcance

El objetivo principal de este Trabajo Fin de Máster es implementar tres propuestas de crawlers enfocados, uno básico, y dos basados en Aprendizaje por Refuerzo. Después, realizaremos un estudio experimental para analizar el rendimiento de cada uno, enfrentándolos al crawler básico. Este objetivo general podemos subdividirlo en otros objetivos más concretos.

El primero, entender la formalización del paradigma del Reinforcement Learning:

- Realizar un estudio teórico del campo. Entender el modelado del proceso interactivo agente-entorno: estados, acciones, funciones de valor, políticas, recompensas, etc.
- Estudiar el modelo de los Procesos de Decisión de Markov, tanto su planteamiento teórico como los algoritmos para su resolución:
  - Aproximación lineal de funciones.
  - Aproximación no lineal de funciones: Deep Q-Learning.

Después, definir e implementar las tres propuestas de crawlers enfocados: básico, basado en ALF (Aproximación Lineal de Funciones), y basado en DQN (Deep Q-Learning). Para ello, necesitamos poner a punto diferentes subsistemas, que justificaremos más adelante en la Sección 3.

- Estudiar el uso de Dumps de Wikipedia como conjuntos de datos de entrada.
- Estudiar el uso del modelo Word2Vec para modelar las páginas como vectores, para ser usados en la similitud coseno, que será utilizada como función de relevancia.
- Estudiar la extracción de vectores de características para modelar las páginas y los enlaces (estados y acciones).
- Estudiar la implementación de la frontera, adaptada a las operaciones necesarias para su uso en problemas de RL.
- Definir la tarea de crawling como MDP, y proponer las dos alternativas para su resolución: ALF y DQN.
- Implementar de manera efectiva diferentes subsistemas independientes que implementen la funcionalidad descrita, que serán utilizados en los crawlers para conformar el algoritmo final de crawling.

Por último, realizar un estudio experimental, para comparar el rendimiento de los tres crawlers propuestos:

- Definir una tarea experimental concisa y representativa del problema.
- Poner a punto un sistema simple de simulación de un entorno de crawling. Para ello, se parte de unos datos de entrada, donde están representadas las páginas y los enlaces entre ellos, y simulamos el proceso secuencial que realiza el crawler, donde en cada paso obtiene una página, analiza su contenido y sus enlaces, y actualiza las prioridades de la frontera.
- Implementar las tres propuestas de crawling que van a ser usadas en los experimentos.
- Ejecutar los experimentos y analizar los resultados obtenidos.

1.3. Estructura del documento

En la Sección 2 se incluye un estudio del estado del arte de las principales áreas involucradas en el proyecto, con el objetivo de contextualizar el trabajo. Estas áreas son Crawling Web, Deep Learning, y Reinforcement Learning.

En la Sección 3 se presenta el diseño del sistema desarrollado, definiendo todos los subsistemas por separado, e integrándolos todos en las propuestas de crawlers.

En la Sección 4 se concretan los experimentos ejecutados, definiendo las tareas a realizar y las configuraciones escogidas. Se presentan también los resultados de los mismos.

En la Sección 5 cerramos este trabajo con las conclusiones extraídas a raíz del desarrollo realizado y los resultados de los experimentos. Además, planteamos las líneas de un posible trabajo futuro.

Por último, se presenta la bibliografía utilizada en este trabajo.

1.4. Contenido común a otro Trabajo Fin de Máster

Como ya hemos comentado, este trabajo se ha realizado tras desarrollar otro TFM para el Máster en I2-ICSI, titulado "Recomendación basada en Procesos de Decisión de Markov", ya que he cursado el doble máster en Ingeniería Informática e I2-ICSI, donde realicé un estudio de la integración del RL en el ámbito de la recomendación. Hay por tanto relación entre ambos trabajos, y en esta Sección vamos a detallar exactamente qué partes son comunes.

Las partes comunes de ambos TFM se concentran en la Sección 2, donde detallamos el Estado del Arte:

- La Sección 2.2, donde se detalla el estado del arte del Deep Learning. En esta Sección hay una excepción, el Apartado 2.2.3, de métodos Word2Vec, que es completamente nuevo.
- La Sección 2.3, donde se detalla el estado del arte del Reinforcement Learning, al completo.
- La Sección 2.4, donde se detalla el estado del arte de los Procesos de Decisión de Markov, al completo.
- La Sección 2.6, donde se detalla el estado del arte del algoritmo de Q-Learning, al completo.

Cabe destacar que aunque estas secciones han sido reutilizadas del anterior TFM, se han visto sometidas a revisión y no son exactamente iguales a las presentadas en el anterior trabajo. Por lo demás, el resto del contenido del presente TFM, (concretamente, 51 de las 67 páginas de la memoria, sin contar anexos), es completamente distinto al anterior.

2

Estado del Arte

En esta Sección se va a contextualizar el trabajo realizando un estudio del arte de las principales áreas involucradas en el mismo: Crawlers, (ó Arañas Web), en la Sección 2.1, Deep Learning en la Sección 2.2, y, por último, Reinforcement Learning, dividido a su vez en una contextualización general, en la Sección 2.3, seguida de un estudio de Procesos de Decisión de Markov (MDP) en la Sección 2.4, y el estudio de los algoritmos aplicados en el trabajo, Aproximación Lineal de Funciones en la Sección 2.5, y Q-Learning en la Sección 2.6.

2.1. Crawlers

Un crawler es una herramienta que se encarga de recorrer la Web, obteniendo a su paso las páginas que haya recorrido. Lo hace, típicamente, de manera automática, y siguiendo unas reglas prefijadas, que definen su comportamiento.

Aunque en el contexto de la Minería Web se utilizan con diversos objetivos, una de sus principales aplicaciones es recorrer la web obteniendo todas las páginas que pueda, para que sean posteriormente indexadas en un buscador web. Estos crawlers no discriminan entre páginas, y son generalmente conocidos como **crawlers universales** [5]. Sin embargo, también existen crawlers que buscan páginas web con ciertas temáticas concretas (por ejemplo, reseñas de películas), con el objetivo de obtener páginas relacionadas a un tema, ó *topic*. Dentro de este espectro, se encuentran los **topical crawlers** [6] [7] y los **focused crawlers** [8] [2] [9].

2.1.1. Funcionamiento de un crawler

Los crawlers tienen como objetivo general recorrer la web automáticamente, y obtener el mayor número de páginas en el menor tiempo posible. Para ello, existen diversos elementos que componen un ciclo iterativo que el crawler ejecuta constantemente, obteniendo páginas en el proceso.

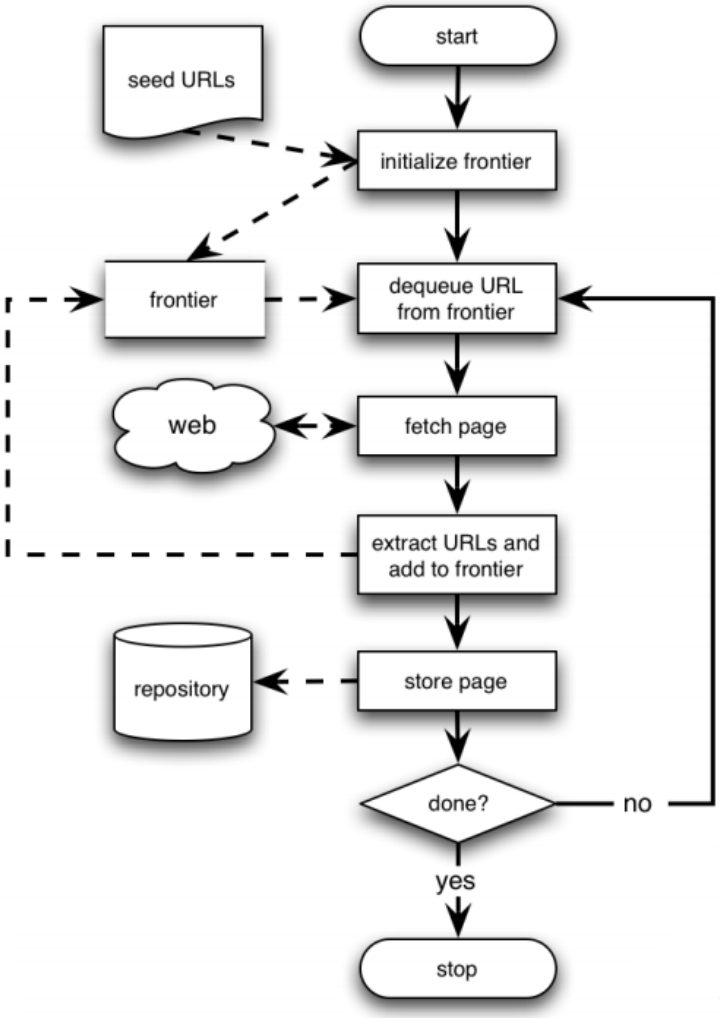


Figura 2.1: Funcionamiento de un crawler genérico. Imagen extraída de [5].

Como punto de partida, los crawlers usan típicamente un conjunto de URLs iniciales, normalmente llamado **semilla**. Con esta semilla se inicializa una **frontera**, que es el conjunto de URLs que el crawler conoce, pero que aún no ha visitado. De esta frontera extrae el crawler la URL que va a visitar en cada paso, haciendo un **fetch**, o **descargado** de la página en cuestión, y obteniendo su contenido HTML. Después, el crawler procesa ese código HTML, obteniendo las URLs que están presentes en esa página, y añadiéndolas de nuevo a la frontera, si no estaban presentes ya. Repite este proceso hasta que se alcance una condición de parada, como podría ser, por ejemplo, obtener un número fijo de páginas.

Este ciclo iterativo puede visualizarse esquemáticamente en la Figura 2.1.

Por la estructura de la Web, podemos entender el proceso de crawling como un algoritmo de búsqueda en grafos. El grafo de la web tendría a las páginas como nodos y los hiperenlaces como aristas. En este grafo, el crawler parte de un conjunto de nodos iniciales (semillas), y sigue las aristas para llegar a nuevos nodos. El paso de expandir un nodo en la búsqueda en grafo ahora consiste en realizar el descargado de la página y procesar las URLs nuevas. Además, como el proceso de crawling puede verse como una búsqueda en grafo, podemos realizar una taxonomía de los crawlers basada en como recorremos ese grafo: en profundidad, o en anchura, aunque nosotros nos vamos a centrar solo en estos últimos, ya que son el tipo más extendido y el que se corresponde con los crawlers que vamos a implementar en este trabajo.

Además del núcleo de funcionamiento que hemos descrito, los crawlers implementan a menudo funcionalidades extra. Una de ellas puede ser la implementación de *timeouts*, con los que establecemos un límite de espera a la hora de recibir respuestas de los servidores web, para no dejar al crawler bloqueado cuando un servidor es muy lento a la hora de realizar el fetch de la página. Otro aspecto muy importante de los crawlers es la etiqueta de comportamiento, una especie de código ético que todo crawler debe respetar para no saturar las comunicaciones en la Web. La mayoría de servidores con los que se comunican los crawlers cuentan en su directorio raíz con un fichero "robots.txt", en los que los servidores especifican una serie de reglas sobre como debe comunicarse el crawler con ellos. Por ejemplo, un servidor podría especificar que no desea recibir más de 3 peticiones por minuto para no ser bloqueado. Este es solo un ejemplo de las restricciones que los servidores especifican en su fichero *robots*, aunque también pueden especificar que ciertos directorios del servidor no sean accedidos por los crawlers, bloquear archivos multimedia, etc.

2.1.2. Tipos de crawler

Dependiendo de cómo implementemos la frontera, regulamos qué páginas son accedidas primero en el proceso iterativo del crawler. Es por ello, que podemos realizar una taxonomía de crawlers, con diferentes tipos.

Breadth-First Crawler

Cuando implementamos la frontera como una cola FIFO, (First In, First Out), estamos realizando la búsqueda en anchura en el grafo. Se escogen primero las URLs que se descubrieron antes, y las nuevas URLs se añaden al final de la cola. Esta estrategia favorece a las páginas con mayor **in-degree**, es decir, a las páginas que son apuntadas por un mayor número de páginas. Esto suele estar correlacionado con el índice de PageRank: cuanto mayor in-degree, mayor PageRank. Aunque a priori esto puede sonar como algo deseable para cualquier crawler, se está introduciendo un sesgo hacia aquellas páginas más populares, que actúan como atractores para los Breadth-First Crawlers.

Crawlers preferenciales

Si en vez de utilizar una cola FIFO, utilizamos una cola con prioridad para la frontera, estamos ante un Crawler Preferencial. Estos crawlers se encargan de asignar a cada página una puntuación, que priorice antes a ciertas páginas, dependiendo del criterio que quieran seguir. Por ejemplo, esta prioridad puede basarse en propiedades topológicas, como podría ser el in-degree de la página, o podría basarse en el contenido de la página, como una medida de similitud entre la página y un topic. Escogiendo adecuadamente esta prioridad, se puede enfocar el crawler hacia determinados tipos de páginas, lo que puede ser muy útil cuando nos interesa ya no obtener una visión amplia de la web, sino un determinado tipo de páginas concretas (páginas de artículos de Amazon, reseñas de películas, etc.).

Dentro de esta categoría, existen 2 subtipos principales: los **focused crawlers**, y los **topical crawlers** [5]. La diferencia principal entre ambos es la forma en la que estiman si una página pertenece al tema que se está buscando. Los focused crawlers utilizan aprendizaje supervisado. Es decir, requieren de un conjunto de páginas, ya etiquetadas, para entrenar un clasificador que más adelante se utilizará para clasificar a su vez las páginas que el crawler recorra.

Por otro lado, los topical crawler utilizan aprendizaje no supervisado. A menudo, contar con un gran conjunto de datos de páginas etiquetadas es demasiado costoso. Por ello, aparece esta categoría de crawlers, que estiman en tiempo de consulta la probabilidad de que una página pertenezca a una categoría. Es por esto que durante este trabajo nos centramos en este tipo de crawler. Para realizar las estimaciones sobre a qué categoría pertenece cada página utilizan el principio de localidad.

Este uso de dos nombres explícitos diferentes para los focused crawler y los topical crawler ha estado presente en la literatura [5], pero a día de hoy se usan los dos conceptos indistintamente para referirse a un crawler preferencial, como puede verse en [2], donde aunque su propuesta es un topical crawler, desde el título se refieren a él como focused crawler. Es por ello que para los crawlers que propondremos en la Sección 3 vamos a usar el nombre de focused crawler, ó crawlers enfocados, aunque realmente encajarían más en la categoría de topical crawler, por utilizar aprendizaje no supervisado.

2.1.3. Principio de localidad

El principio de localidad establece que cada página web contiene señales que indican qué contenido podemos encontrar en las páginas vecinas (aquellas que son enlazadas por la primera). Para demostrarlo, podemos dividir el problema en dos cuestiones más simples:

- Conjetura de contenido (link-content conjecture): se pregunta si dos páginas que se enlazan la una a otra son similares en contenido, comparado a otras dos páginas escogidas aleatoriamente.
- Conjetura de cluster (link-cluster conjecture): se pregunta si dos páginas que se enlazan la una a otra son similares semánticamente, comparado a otras dos páginas

escogidas aleatoriamente.

Estas conjeturas en su conjunto fueron demostradas en el año 2000 en [10], donde los autores realizaron diversos experimentos en un conjunto de datos de páginas escogidas aleatoriamente. Así, proponen una demostración fiable de que la mayoría de páginas están enlazadas a otras con contenido similar, además de comprobar que el contenido de los textos ancla (*anchor texts*), y el texto que los rodea, describen las páginas a las que apuntan. Ambas hipótesis son fundamentales para la implementación de nuestros crawlers enfocados, ya que se basan en la asunción de que páginas relevantes llevarán con alta probabilidad a otras páginas relevantes, y analizamos el texto que rodea a los enlaces como guía a la hora de priorizar unos u otros.

2.2. Deep Learning

El Aprendizaje Profundo, ó *Deep Learning* en inglés, es una rama del Aprendizaje Automático, que modela abstracciones complejas de datos, que requieren de transformaciones no lineales, por lo que suelen utilizarse redes neuronales profundas.

Es por tanto importante poner en contexto el estado del arte de los modelos de redes neuronales, y los algoritmos usados en Deep Learning para entrenarlas eficazmente.

2.2.1. Redes Neuronales

Los algoritmos de redes neuronales son algoritmos "bioinspirados". Es decir, surgieron como un intento de imitar computacionalmente un modelo natural y existente como es la interacción de las neuronas biológicas de los sistemas nerviosos de los mamíferos. Es por ello, que vamos a definir primero las neuronas biológicas, para poder hacer una analogía más adelante, y poder identificar qué partes de la neurona artificial corresponden con las de la neurona biológica, y entender así mejor el modelo computacional.

Neurona biológica

El modelo de neurona biológica que se enseña hoy en día fue originado por Ramón y Cajal, en 1906, con la publicación de su "Doctrina de la Neurona". La teoría original no está disponible en repositorios públicos, pero en [11] está disponible un artículo que resume el original. Ramón y Cajal instauró el estándar actual, en el que la neurona biológica se considera la unidad básica del sistema nervioso. En la figura 2.2 se puede visualizar la concepción actual de una neurona.

La función principal de las neuronas es recibir, procesar y transmitir información a través de impulsos eléctricos, que generan gracias a la excitabilidad de sus membranas. Se comunican con otras neuronas a través de sinapsis, conformando así una red neuronal biológica.

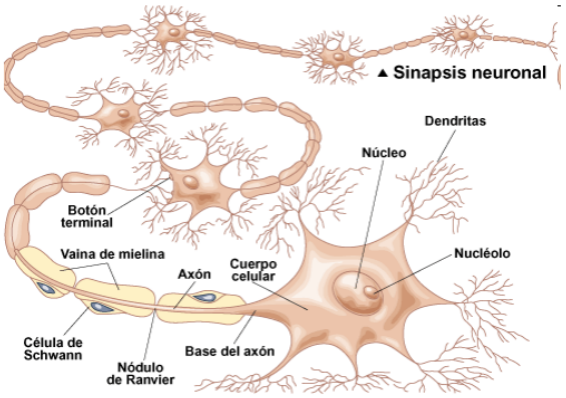


Figura 2.2: Imagen esquemática de una neurona biológica. Imagen extraída de las diapositivas de teoría de la asignatura Neurocomputación, del Grado en Ingeniería Informática, EPS, UAM.

El proceso de sinapsis es el proceso de conexión y comunicación que utilizan las neuronas para relacionarse entre ellas. Una neurona, llamada célula emisora, genera una descarga química, y segrega neurotransmisores que alcanzan el axón de la neurona receptora. A través de estos neurotransmisores, la neurona receptora se excitará o inhibirá, dependiendo de la situación.

En cuanto a la morfología de una neurona, hay varios elementos que conviene definir. El primero de ellos son las dendritas. Las dendritas son ramificaciones que salen del cuerpo celular de la neurona, y que se encargan de procesar los neurotransmisores recibidos en una sinapsis. Este cuerpo celular del que salen las dendritas, recibe el nombre de soma. Y a partir de este soma, en dirección contraria a las dendritas, se encuentra una prolongación llamada Axón, que conduce los impulsos desde el soma hacia otras neuronas.

Neurona artificial

En 1943, Warren McCulloch y Walter Pitts diseñaron lo que se considera el primer modelo de una red neuronal artificial, publicado en [12]. En su modelo, se pueden identificar varios elementos biológicos. Según este modelo, una red neuronal artificial se compone de:

- Un vector de entrada, X, donde cada entrada será $x_i \in X$.
- Un vector de pesos, W, donde cada peso será $w_i \in W$.
- Una función de activación, a, cuya entrada será una agregación, g, de las entradas.
- Una salida, f, que se computa con la función de activación, $f = a(g(x))$.

El modelo McCulloch-Pitts se muestra en la figura 2.3. Las entradas, representadas en el vector X, realizan la misma función que las dendritas. Estas se encargan de recibir el

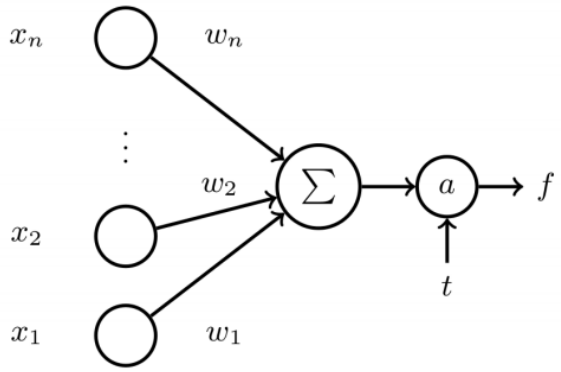


Figura 2.3: Representación simplificada del modelo McCulloch-Pitts. Imagen extraída de [13].

impulso de otra neurona, que viaja a través del soma hacia el axón, que lo lleva a su vez hasta las dendritas de la siguiente neurona. Aquí, la salida f simula ese impulso que viaja a través del axón. La función de activación, a, puede ser lineal o no lineal. Un ejemplo muy sencillo, sería que la función de activación se activara si se superase un umbral, comúnmente conocido como el umbral de activación:

$$f = \begin{cases} 1, & \text{si } \sum_{i=1}^{n} x_i w_i > t \\ 0, & \text{en cualquier otro caso} \end{cases}$$

2.2.2. Entrenando Redes Neuronales

Las redes neuronales son aproximadores muy potentes, pero necesitan de un entrenamiento previo para poder aprender. Resuelven problemas de clasificación y regresión, y por tanto, necesitan que se les presente una serie de casos ya clasificados, para aprender a clasificar nuevos ejemplos. El aprendizaje de una red neuronal se concentra en los pesos de sus enlaces, y por tanto, cualquier algoritmo de entrenamiento de una red neuronal modificará esos pesos con el objetivo de minimizar una función de error determinada. Una técnica muy utilizada es la del descenso por gradiente. Generalizando, la actualización de pesos puede definirse de la siguiente forma:

$$W_t = W_{t-1} - \alpha \overrightarrow{\nabla E}$$

donde α es la tasa de aprendizaje, y $\overrightarrow{\nabla E}$ es el gradiente del error.

Hay múltiples modelos de redes neuronales, y múltiples algoritmos de entrenamiento: Regla Delta es uno muy sencillo, utilizado para entrenar redes de una sola capa, y retropropagación, su generalización para redes profundas, entre otros. Ya que el objetivo es poner en contexto el problema de Deep Learning, y este suele implicar resolver problemas no lineales, vamos a definir en profundidad el algoritmo de retropropagación.

Algoritmo de retropropagación

Las redes neuronales de una sola capa resuelven problemas lineales, y su entrenamiento es muy sencillo. Sin embargo, si necesitamos resolver un problema no lineal, necesitamos hacer uso de una red neuronal profunda. Estas redes neuronales cuentan con un algoritmo de entrenamiento muy potente: el algoritmo de retropropagación [14], ó *backpropagation* en inglés. Los problemas de Deep Learning suelen implicar relaciones no lineales. Es por esto que el algoritmo de retropropagación es un pilar básico en el área.

Este algoritmo utiliza el descenso por gradiente introducido en el apartado anterior para minimizar el error cuadrático total en la salida. Para ello, el algoritmo se divide en 3 partes. Primero, el patrón de entrada se propaga hacia delante en la red (fase de feedforward). Después, se calcula el error asociado, viendo la diferencia entre la salida obtenida y, y la salida esperada t. Tras esto, se retropropaga este error, desde las últimas capas hacia las primeras. Y por último, se ajustan los pesos de la red. Estos pasos se repiten mientras el error cuadrático medio no haya alcanzado un mínimo admisible.

Aunque el entrenamiento es muy costoso, en la fase de explotación, tan solo es necesario propagar hacia delante el patrón de entrada, por lo que, aunque el entrenamiento de un perceptrón multicapa es muy lento, se compensa con la rapidez con la que clasifica nuevos ejemplos en la fase de explotación.

Activación ReLU

En las redes neuronales, la función de activación es uno de los elementos más determinantes, ya que determina cómo se transforma la suma ponderada de las entradas para obtener la salida de cada neurona. Como hemos visto, el mecanismo que usa el algoritmo de backpropagation para minimizar el error es el descenso por gradiente. Por tanto, nos interesa que la función de activación sea fácilmente derivable, para que computacionalmente sea rápido calcular ese gradiente.

Una función de activación más intuitiva que la que vamos a presentar ahora es la función sigmoidal. Esta función se planteó como un buen candidato a función de activación, debido a que cumplía ciertos requisitos: sus posibles valores están en el rango $(0, 1)$, es monótona, pues siempre crece, y es diferenciable en todos sus puntos. Sin embargo, tanto con esta función, como con la tangente hiperbólica (otra función muy usada previamente), apareció el problema de los *Vanishing Gradients*, donde en determinados puntos, el gradiente se va "desvaneciendo" hacia valores muy pequeños, que impiden actualizar correctamente los pesos. Es por ello, que en el dominio del Deep Learning empezó a utilizarse la función ReLU [15], (*Rectified Linear Unit*). En la Figura 2.4 puede observarse las definiciones gráficas de ambas funciones.

La función de activación ReLU otorga valores nulos a entradas negativas, e iguala la salida a la entrada en el caso de entradas positivas. Con esta función, ya no tenemos el problema del desvanecimiento de los gradientes. Además, derivar esta función es mucho más sencillo que derivar la sigmoide, por lo que se acelera el aprendizaje de la red. En

concreto, la definición de la derivada de la función ReLU es la siguiente:

$$\frac{\partial R}{\partial z} = \begin{cases} 1, & \text{si } z \geq 0 \\ 0, & \text{si } z < 0 \end{cases}$$

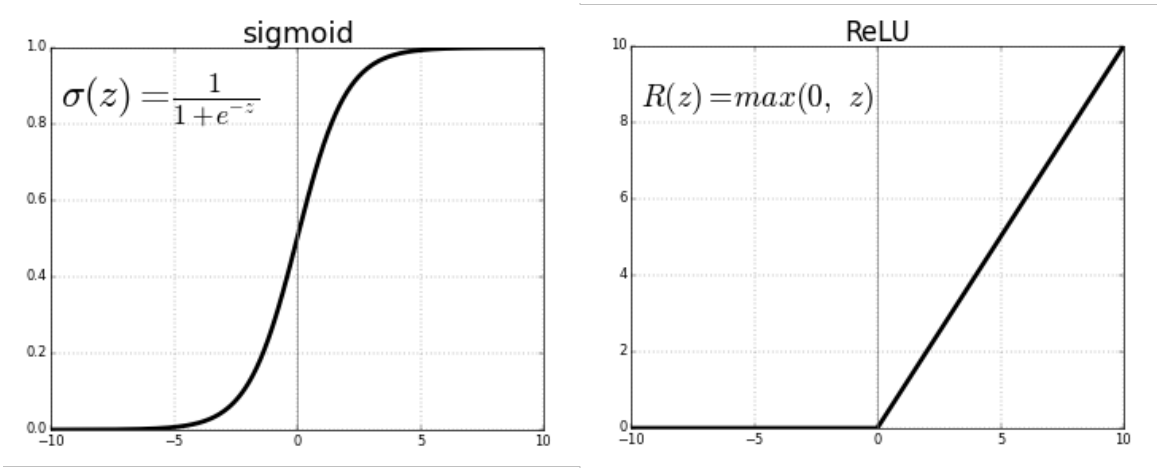


Figura 2.4: Representación de las funciones sigmoidal y ReLU. Imagen extraída del portal Towards Data Science: `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`

Inicialización

A la hora de entrenar redes profundas, es muy importante escoger con cuidado cómo inicializamos los pesos de la red. Esta inicialización es el punto de partida hacia la deseada convergencia, y según qué punto escojamos, el entrenamiento puede acelerarse mucho, o volverse muy lento, llegando a veces a no converger.

Inicializar los pesos aleatoriamente es la primera opción que surge, aunque no es buena idea por varios motivos. Primero, si los valores que le damos a los pesos son muy grandes, o muy pequeños, corremos el riesgo de que el gradiente se "desvanezca", ya que al aplicar una función de activación, como una sigmoide, obtendremos un valor proximo al 1, donde la dirección del gradiente (la pendiente) cambia muy lentamente y el entrenamiento se hace muy lento. Lo mismo ocurre con valores muy pequeños, donde la función mapeará a valores próximos al 0.

Existen varios métodos que, teniendo en cuenta los problemas mencionados, buscan aportar una inicialización que acelere la convergencia. Hay 2 métodos destacados: el primero, la inicialización de He, propuesto en [16], y el segundo, la inicialización de Xavier Glorot, propuesto en [17]. Ambos métodos son muy parecidos, y ya que el método de Xavier Glorot está más extendido, es el que vamos a definir en este trabajo.

El objetivo del método de Xavier Glorot es mantener la varianza de los pesos entre diferentes capas de la red. Es decir, queremos que la señal que va pasando a través de

la red, no alcance un valor demasiado alto, (típicamente, nos referimos a esto como que la señal explote), ni que se "desvanezca" hacia el cero. Para ello, inicializamos los pesos escogiéndolos de una distribución, cuya media μ y varianza $Var(w_i)$ son las siguientes:

$$\mu = 0$$
$$Var(w_i) = \frac{1}{n_{in} + n_{out}},$$

dónde n_{in} es el número de neuronas de entrada, ó de la capa anterior; y n_{out} es el número de neuronas de salida, ó de la capa siguiente. Utilizando este método, conseguimos que la varianza se mantenga a través de los pesos de las diferentes capas, y evitamos el problema de que la señal explote o se desvanezca.

Regularización

En los problemas de Aprendizaje Automático, surge siempre un fenómeno conocido como el sobreajuste, ó *overfitting*. Se produce cuando el aproximador, en este caso, nuestra red neuronal, aprende demasiado bien los datos de entrada, perdiendo así toda capacidad para generalizar y clasificar bien nuevos ejemplos. En estos casos, el error en entrenamiento es muy bajo, pero el error en test es altísimo. Típicamente, existen 2 formas de resolver este problema: obtener más datos, o usar técnicas de regularización. A menudo la primera opción es imposible, debido al alto coste que puede suponer recolectar nuevos datos. Es por ello, que es importante entender algunas técnicas de regularización. Vamos a centrarnos en 2 técnicas muy extendidas, el uso de regularización L2, y el uso de dropouts en las redes neuronales.

La regularización L2 [18] añade un nuevo término a la función de coste de la red, que penalizará a los pesos que crezcan demasiado. Por tanto, la función de coste es ahora:

$$C_{L2} = Coste + \lambda ||W||^2,$$

dónde $||W||$ es la norma de la matriz de pesos, y λ es el factor de regularización. Es este factor el que podemos ajustar según el efecto que deseemos conseguir. Los pesos grandes serán más penalizados cuanto más grande hagamos este factor, y lo serán menos si decrece.

Además de este método, existe la opción de utilizar *dropouts* [19]. Con este método, establecemos una probabilidad de que cada nodo de la red sea descartado. Lo que nosotros ajustamos es el umbral de *dropout*. Así, si nuestro umbral es 0,6, hay un 40 % de probabilidades de que un nodo sea descartado. Aunque parezca que no tiene sentido deshacerse de información relevante de la red descartando algunos nodos, este método ha sido probado y se ha demostrado que mejora el rendimiento de las redes neuronales. El razonamiento detrás de este método de regularización es que la red, al no poder confiar en que un nodo no vaya a desaparecer, no asigna pesos demasiado grandes a ninguna de sus conexiones, porque puede perderlas en cualquier momento. Así, mantiene los pesos bien distribuidos, sin que ninguno sea más importante que el resto, regularizando el modelo de

la red.

2.2.3. Capas de Embedding

Durante el desarrollo de este trabajo utilizaremos a menudo modelos Word2Vec, un modelo de Word Embedding donde representamos las palabras de un vocabulario en forma vectorial. Es por ello que introducimos en esta Sección las capas de embedding. Un embedding es un mapeado de un espacio discreto a uno continuo. Una capa de embedding, por tanto, se encarga de aprender una representación vectorial continua de variables discretas. Además, al realizarse de manera supervisada, en este espacio continuo, los elementos similares discretamente estarán cerca a su vez en el espacio continuo. Esto es una gran ventaja, ya que representaciones más tradicionales, como puede ser la codificación One Hot, tienen el problema de no respetar estas similitudes entre elementos al realizar la transformación. En concreto, en la codificación One Hot, si aplicamos la similitud coseno, siempre obtenemos una similitud nula entre cada par de vectores One Hot.

Estas capas de embedding tienen su origen en el campo del procesamiento de lenguaje natural, donde se usa para transformar el espacio discreto de las palabras a uno continuo. En la figura 2.5 podemos observar un ejemplo de transformación realizada mediante embeddings.

Word2Vec

Una posible aplicación de la técnica de embedding que acabamos de comentar es la de representar las palabras de un documento (o vocabulario) en forma vectorial, lo que recibe el nombre de **Word Embedding**. Análogamente a la técnica de embedding general, aplicando este método conseguimos que aquellas palabras similares, tanto sintáctica como semánticamente, también tengan representaciones que tengan posiciones cercanas en el espacio. En la Figura 2.6 se muestra un ejemplo de esto, donde podemos apreciar que los vectores de $w1$ y $w2$ están más próximos en el espacio, ya que se tratan de palabras similares, mientras que el de $w3$ está mucho más alejado. Esto es un caso concreto en 2 dimensiones, para facilitarnos su visualización, pero en la práctica los Word Embeddings pueden extraer vectores con muchas más dimensiones.

Una técnica concreta para extraer un WordEmbedding es Word2Vec. Puede usarse con dos variantes diferentes: *Skip Gram* y *Common Bag of Words* (CBOW). Ambos métodos hacen uso de redes neuronales para generar los embeddings.

El modelo CBOW utiliza una red neuronal que recibe a la entrada la representación *One-Hot* de una palabra, o varias, que representan el contexto de otra palabra, que es la objetivo. A partir del contexto, el objetivo de la red es predecir la palabra que corresponde a ese contexto. Mediante el algoritmo de *back-propagation*, y utilizando redes neuronales profundas, la red acaba aprendiendo representaciones de las palabras objetivo a la salida, que conforman los word embeddings deseados.

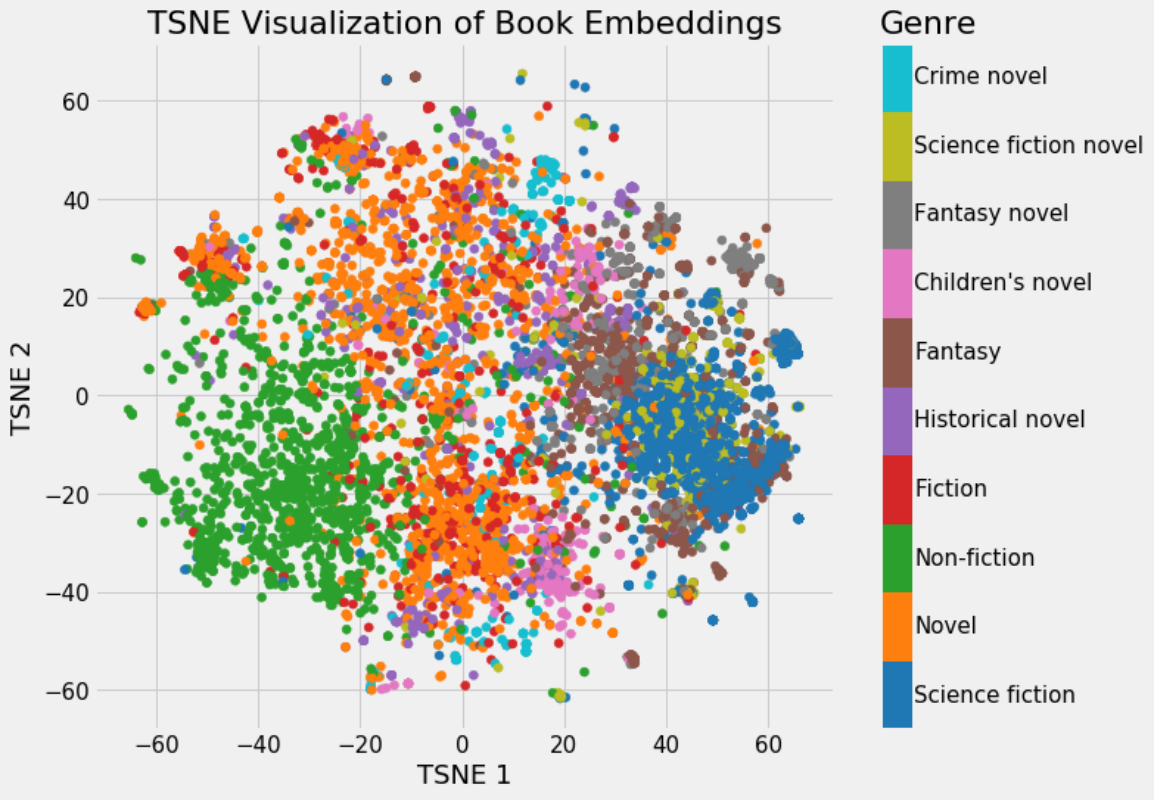


Figura 2.5: Ejemplo de una transformación realizada mediante embeddings. Transformamos una variable discreta, libros, a un espacio continuo. En la imagen se observa como se distribuyen los libros por género. Extraída del portal Towards Data Science.

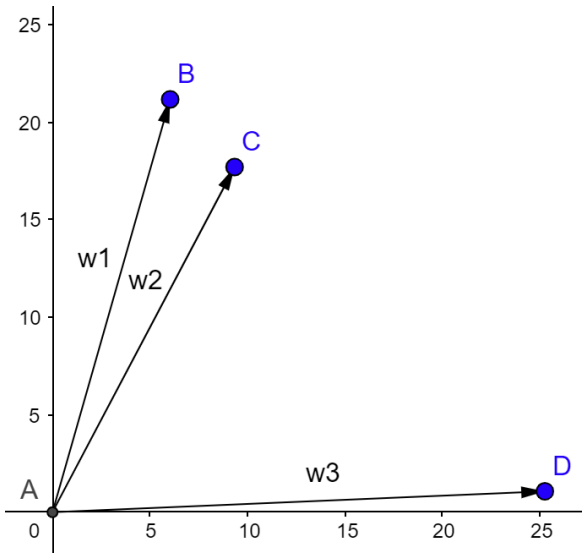


Figura 2.6: Ejemplo de un Word Embedding, donde las palabras $w1$ y $w2$ son más similares entre ellas que a $w3$.

El modelo Skip Gram "invierte" el proceso anterior. Ahora, a la entrada, introducimos palabras objetivo, y sus contextos están a la salida. De nuevo, utiliza *back-propagation* y redes profundas para aprender los Word Embeddings.

Ambos métodos tienen sus ventajas y desventajas: mientras que Skip Gram rinde mejor con conjuntos de datos pequeños, y extrae mejores representaciones para palabras poco frecuentes, CBOW es más rápido y extrae mejores representaciones para palabras más frecuentes.

2.3. Aprendizaje por refuerzo

Cuando un niño se encuentra en las fases tempranas de su vida, intuitivamente aprende siguiendo un proceso natural de interacción con el entorno que lo rodea. Cuando toca algo que quema, el entorno le está mandando información a través de su tacto, que interpreta que tocar algo que quema duele, y por tanto, es peligroso. Este modelo de aprendizaje no es exclusivo de los humanos, está presente en toda la naturaleza. Los animales aprenden ciertos comportamientos a través de esa interacción con lo que les rodea, sin que tengan ningún profesor que les diga exactamente qué deben hacer. El paradigma del Aprendizaje por Refuerzo, ó *Reinforcement Learning, RL* en inglés, surge con la intención de aproximar computacionalmente esta idea tan natural del aprendizaje a través de una interacción con el entorno, que está presente en nuestras vidas desde que nacemos. Este campo conforma una alternativa clara a los algoritmos clásicos de Machine Learning, típicamente clasificados como algoritmos supervisados (clasificación) o no supervisados (clustering).

La idea central del Aprendizaje por Refuerzo [1], es modelar la interacción de un agente con un entorno, a través del tiempo. Este agente puede tomar acciones, que producen cambios en el entorno, y observará una recompensa, que guía su aprendizaje. Estos cambios producidos en el entorno, motivan que se utilicen estados como modelo del estado actual del entorno. En cada paso del sistema, t, el agente toma una acción, observa el estado siguiente, y extrae una recompensa, que depende del entorno. Esta interacción se puede visualizar en la Figura 2.7.

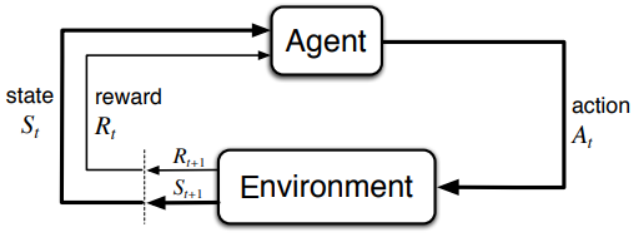


Figura 2.7: Esquemático de la interacción entre un agente y un entorno en el paradigma del Aprendizaje por Refuerzo. Imagen extraída del libro de Sutton y Barto [1].

En un sistema de Aprendizaje por Refuerzo, se suele definir el conjunto de estados posibles como S, siendo uno de esos estados $s \in S$, el de acciones posibles como A, siendo

una de esas acciones $a \in A$, y el de recompensas posibles como R, siendo una de esas recompensas $r \in R$.

Una vez definida la idea básica, es importante definir otros elementos que también conforman un sistema de Aprendizaje por Refuerzo. En concreto, los elementos básicos que vamos a introducir son: la política, la señal de recompensa, la función de valor y el concepto de retornos.

2.3.1. Políticas

Una política es una función que define qué acción tomar dado el estado actual. Es decir, define el comportamiento del agente en un determinado momento. Esta decisión de qué acción tomar se puede modelar con políticas estáticas o estocásticas.

En una política estática, dado un estado s, solo hay una acción posible a ser elegida, con probabilidad 1. Es decir, la política estática es una función $f : S \rightarrow A$, e indica directamente qué acción tomar en cada momento.

Por otro lado, una política estocástica es una función que define una distribución de probabilidades sobre las acciones ($a \in A$) para cada estado ($s \in S$). Es decir, una política π es una función $f : A, S \rightarrow \pi(a|s)$, siendo esto último la probabilidad de tomar tal acción dado el estado actual.

La política es el núcleo de un sistema de Aprendizaje por Refuerzo, en el sentido de que define completamente el aprendizaje y el comportamiento del agente. Es decir, es en la política donde se almacena el aprendizaje alcanzado por el agente, y el objetivo del problema es aprender una política óptima. Se profundizará en este tema en la Sección 3.

2.3.2. Señales de Recompensa

A lo largo de los pasos de un sistema de Aprendizaje por Refuerzo, el entorno manda al agente la recompensa, un número, que se puede definir como función del estado: $f : S \rightarrow \mathbb{R}$. Maximizar esta recompensa a largo plazo es el objetivo de un agente, y es por tanto una heurística que puede utilizarse como métrica de calidad en un instante de tiempo t. Esta recompensa solo depende del estado actual del entorno, y por tanto, la única manera en la que el agente puede influir en ella es mediante las acciones que toma. Estas acciones provocarán un cambio de estado, que a su vez, recibirá una recompensa diferente por parte del entorno.

2.3.3. Funciones de valor

Las recompensas del entorno son inmediatas, se otorgan a un estado concreto del entorno, el estado actual. Sin embargo, existe una variante análoga a la recompensa, pero esta vez computada teniendo en cuenta el largo plazo. La función que otorga un valor a

cada estado, indicando cómo de deseable es estar en ese estado a largo plazo, es la función de valor, que puede definirse como $f : S \rightarrow \mathbb{R}$. En otras palabras, el valor de un estado es la suma total de recompensas que un agente espera acumular, partiendo desde ese estado. Aunque recompensas y valores son parecidos, en el sentido de que ambos evalúan como de deseable es encontrarse en un cierto estado, a la hora de tomar decisiones, será la función de valor la que se tendrá en cuenta, ya que es la que nos ayuda a maximizar las recompensas que obtenemos durante todo el proceso iterativo del Aprendizaje por Refuerzo.

Existen dos tipos de funciones de valor, la que acabamos de describir, es la función de valor del estado. Dado un estado, la función nos indica como de bueno es para el agente encontrarse en ese estado. Sin embargo, existe otra variante, la función de valor estado-acción. En este caso, dado un estado, la función de valor nos indica cómo de bueno es para el agente tomar una acción. Esta función puede definirse en el espacio $f : S, A \rightarrow \mathbb{R}$.

2.3.4. Retornos

En el apartado anterior hemos introducido sin mencionarlo el concepto de retorno. Hemos visto que el valor de un estado está relacionado con la suma de recompensas que obtendremos de media si partimos desde ese estado. Por tanto, podemos definir formalmente, el retorno esperado como:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T,$$

donde T es el paso final. Esta definición sin embargo tiene un problema, ya que cuando la tarea es infinita, es decir, no tiene un paso final, el retorno esperado tiende a ∞. Es por ello que necesitamos introducir la idea del descuento:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-1} R_T = \sum_{i=0}^{\infty} \gamma^i R_{t+i+1},$$

donde γ es el factor de descuento, $0 \leq \gamma \leq 1$. Este factor de descuento nos sirve para regular cómo de importantes son las recompensas inmediatas, en comparación con las recompensas futuras. A medida que γ se aproxima a 1, las recompensas futuras tienen más importancia. Si $\gamma = 0$, tan solo importa la recompensa inmediata. Cabe destacar además, que si $\gamma = 1$, estaríamos en el caso anterior, y en una secuencia infinita, el retorno sería ∞.

2.4. Procesos de Decisión de Markov

Cuando una tarea de Aprendizaje por Refuerzo, cuenta con estados que cumplen la propiedad de Markov, recibe el nombre de Proceso de Decisión de Markov (ó MDP, de

Markov Decission Process).

2.4.1. Propiedad de Markov

Un estado, $S_t \in S$, es un estado de Markov, si cumple la siguiente propiedad:

$$P[S_{t+1}|S_1, ..., S_t] = P[S_{t+1}|S_t]$$

Es decir, el estado es de Markov si contiene toda la información útil que el agente había obtenido hasta ese momento, o dicho de otra manera, que los estados que se han recorrido anteriormente pueden ser deshechados, ya que su información está incluida en el último de ellos.

Otra forma de verlo, es que el futuro es independiente del pasado, dado el presente. Análogamente, conocido el estado actual, los estados anteriores pueden ser descartados ya que no aportan nueva información relevante que no esté ya contenida en el estado actual.

2.4.2. Tipos de MDP

Si los espacios de acciones y de estados del agente son finitos, hablamos entonces de un MDP finito. Es importante definir que son espacios finitos, para poder hablar en los siguientes apartados de probabilidades, y no de funciones de densidad. En el resto del trabajo, asumiremos que hablamos de MDPs finitos, por conveniencia.

A su vez, existen entornos plenamente observables, o parcialmente observables. Cuando el entorno actualiza su estado, y el agente puede observar ese cambio con plenitud, se define formalmente el proceso como un Proceso de Decisión de Markov Plenamente Observable (ó FOMDP, de *Fully Observable MDP*).

Sin embargo, hay otros entornos que son parcialmente observables, en los que el agente no tiene acceso total al estado del entorno, y no puede observar exactamente cómo cambia. En estos casos, el agente tiene que construir un modelo para definir su propio espacio de estados, a partir de lo que pueda observar del entorno. Se define formalmente este proceso como un Proceso de Decisión de Markov Parcialmente Observable (ó POMDP, de *Partially Observable MDP*). En este trabajo se ha trabajado con la primera opción, así que conceptos y acercamientos específicos de los POMDPs no van a ser tratados en los siguientes apartados.

2.4.3. Fully Observable MDPs

Podemos definir un FOMDP como una tupla (S, A, P, R, γ) [1], donde S es el espacio de estados, A es el espacio de acciones, P es la función de transición, R es la función de recompensa, y γ es el factor de descuento.

La función de transición P define las transiciones entre cada par de estados, dada la acción tomada en ese momento:

$$p(s'|s,a) = Pr\{S_{t+1} = s'|S_t = s, A_t = a\}$$

La función de recompensa R define las recompensas esperadas para cada par estado-acción:

$$r(s,a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$$

El factor de descuento γ, como se introdujo en el Apartado 2.2.4, regula la importancia de las recompensas inmediatas, frente a las futuras.

Estos 5 elementos definen el MDP, pero resolver la tarea de Aprendizaje por Refuerzo suele implicar estimar funciones de valor. Recordemos que una función de valor es una función que estima cómo de bueno es, ó estar en un estado, ó tomar una acción, dado el estado actual. Además, también hemos introducido el concepto de política. Una política π es una función que asigna a cada par estado-acción, una probabilidad, $\pi(a|s)$. Por tanto, podemos definir, para un FOMDP, el valor de un estado como

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right]$$

$$v_\pi(S_T) = 0,$$

donde S_T es el estado terminal, si hay alguno, y $\mathbb{E}_\pi$ es la esperanza del retorno, dado que el agente sigue la política π.

Además, podemos definir el valor de una acción, como vimos en el Apartado 2.2.4. Llamamos función de valor acción-estado [1], de una acción a, en un estado s, bajo una política π, a la esperanza del retorno empezando desde el estado s, tomando la acción a, y siguiendo la política π:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right]$$

Estas funciones de valor se pueden desarrollar recursivamente, desarrollando la expresión del retorno esperado. Con esta transformación, llegamos a obtener lo que puede que sea la ecuación más importante de un MDP: la Ecuación de Bellman.

Ecuación de Bellman

Si cogemos la expresión inicial de la función de valor, y la desarrollamos [1], obtenemos una relación entre el valor de un estado, y el valor de los estados que le suceden:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\middle|S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}\middle|S_t = s\right] \\ &= \sum_a \pi(a|s)\sum_{s'}\sum_r p(s', r|s, a)\left[r + \gamma\mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}\middle|S_{t+1} = s'\right]\right] \\ &= \sum_a \pi(a|s)\sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{2.1}$$

Gracias a la ecuación de Bellman hemos dado una expresión más interpretable al valor de un estado. Ahora, el valor de un estado es la recompensa que recibimos en ese momento, sumado a una media descontada, (con el factor γ), de todos los posibles estados sucesores, ponderados con la probabilidad de llegar a ellos.

Para resolver la tarea de RL, se busca encontrar una política óptima que maximice este valor del estado, siguiendo la ecuación de Bellman. Hay muchas metodologías para encontrar esta política óptima, pero antes de entrar en ellas, vamos a definir el concepto de optimalidad.

2.4.4. Valores y políticas óptimos

Decimos que una política π es mejor que otra π', si su retorno esperado es mayor o igual que el de π', para todos los estados [1]. Es decir:

$$\pi \geq \pi' \Leftrightarrow v_\pi(s) \geq v_{\pi'}(s), \forall s \in S$$

Decimos, que una política es **óptima**, si es mejor o igual que todas las demás. Puede haber más de una política óptima, pero denotamos a todas ellas con el símbolo π_*.

Todas ellas, siguen funciones de valor óptimas, ya sean de estado:

$$v_*(s) = \max_\pi v_\pi(s) \forall s \in S,$$

ó de estado-acción:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \forall s \in S$$

Además, podemos aplicar la ecuación de Bellman a ambas expresiones, obteniendo la llamada ecuación de Bellman óptima:

$$v_*(s) = \max_{a \in A(s)} \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')]$$

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_a q_*(S_{t+1}, a') \middle| S_t = s, A_t = a\right] = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')]$$

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')]$$

Definir estos valores óptimos es necesario ya que, al extraer las funciones de valor óptimas, extraer una política óptima es trivial. En cada paso temporal, habrá una o más acciones que maximicen el valor, cualquier política que priorice esas acciones, será por tanto una política óptima.

En la práctica, calcular la política óptima puede no ser viable en términos de computabilidad. Es por ello, que la mayoría de algoritmos lo que extraen es una aproximación de la política óptima. Algunos de estos métodos, son métodos basados en Programación Dinámica [20], cómo Policy Iteration, ó Value Iteration; métodos de Monte Carlo [21]; ó métodos basados en Aprendizaje de Diferencias Temporales, cómo Q-Learning [3] [22] [4], o métodos lineales, como Aproximación Lineal de Funciones [23] [24] [25]. En el trabajo hemos utilizado estos dos últimos algoritmos, y es por ello que vamos a presentarlos formalmente en los siguientes apartados.

2.5. Aproximación Lineal de Funciones

La Aproximación Lineal de Funciones [1] es un método de estimación con el que podemos estimar el valor de una función de valor, ya sea del estado, V, o del estado-acción, Q. Con este método, definimos una función $\hat{v}$, (función que aproxima el valor real, v), como una combinación lineal de un vector de parámetros, w, y un vector de características, $x(s) = (x_1(s), x_2(s), ..., x_n(s))$, que debe tener el mismo número de componentes que w. Para estos métodos, estas características deben ser diseñadas específicamente para cada problema, teniendo en cuenta el conocimiento del dominio. Esta es una de las principales desventajas con respecto a otros métodos de aproximación, como por ejemplo, una aproximación no lineal de funciones, realizada con una red neuronal, donde podemos diseñar soluciones más genéricas. Sin importar cómo construyamos estas características, definimos la función de aproximación como:

$$\hat{v}(s,w) = w^T x(s) = \sum_{i=1}^{n} w_i x_i(s)$$

En este caso, el conocimiento del algoritmo está concentrado en el valor de los pesos. Es por ello que necesitamos un mecanismo para, progresivamente, actualizar esos pesos minimizando un error cuadrático. Un posible método que podemos aplicar para conseguir una expresión de regla de actualización es el descenso por gradiente. El primer paso es, por tanto, formalizar la expresión del error cuadrático que queremos minimizar. Clásicamente, en los algoritmos de Aprendizaje Automático supervisados, se ha buscado minimizar el *root mean squared error* (RMSE). En [1], se propone la siguiente expresión para este error cuadrático:

$$RMSE(w) = \sqrt{\sum_{s \in S} d(s) \Big[v_\pi(s) - \hat{v}(s,w) \Big]^2},$$

donde $d : S \to [0,1]$ es una distribución de probabilidad sobre los estados, v_π es el valor real del estado siguiendo la política π, y $\hat{v}$ es la función de valor aproximada. Este error hay que derivarlo con respecto a w, para obtener la expresión del gradiente:

$$\nabla = \frac{\partial RMSE}{\partial w} = -\Big[v_\pi(s) - \hat{v}(s,w) \Big] \nabla \hat{v}(s,w),$$

asumiendo que la probabilidad de los estados es uniforme. Por tanto, tenemos que actualizar los pesos en la dirección de ese gradiente a la inversa, ya que es donde más rápido decrece el error:

$$w_{t+1} = w_t + \alpha \Big[v_\pi(s) - \hat{v}(s,w) \Big] \nabla \hat{v}(s,w_t),$$

donde α es un número positivo que regula el tamaño de cada paso.

Sin embargo, nosotros buscamos encontrar una función de valor estado-acción óptima. Para ello, podemos aplicar este razonamiento de manera análoga a la función de valor estado-acción, empezando por definirla como una combinación lineal entre el vector de pesos w y un vector de características, en este caso, del estado y la acción, $x(s,a)$:

$$\hat{q}(s,a,w) = w^T x(s,a) = \sum_{i=1}^{d} w_i x_i(s,a)$$

Además, podemos aplicar el descenso por gradiente utilizando esta vez el error de diferencias temporales, δ, en vez del $RMSE$ que habíamos definido previamente. Este error se llama así por la familia de algoritmos de resolución de MDPs homónima, y

podemos definirlo como:

$$\delta(w) = r_{t+1} + \gamma\hat{q}(s_{t+1}, a_{t+1}, w) - \hat{q}(s_t, a_t, w)$$

Por tanto, nuestro nuevo $RMSE$ y nuestro nuevo gradiente, ∇ son:

$$RMSE_{TD}(w) = \sqrt{\left[r_{t+1} + \gamma\hat{q}(s_{t+1}, a_{t+1}, w) - \hat{q}(s_t, a_t, w)\right]^2}$$

$$\nabla = \frac{\partial RMSE_{TD}(w)}{\partial w} = -\left[r_{t+1} + \gamma\hat{q}(s_{t+1}, a_{t+1}, w) - \hat{q}(s_t, a_t, w)\right]\nabla\hat{q}(s_t, a_t, w)$$

Por tanto, obtenemos la siguiente regla de actualización de los pesos, obtenida tras aplicar el descenso por gradiente, minimizando el error de diferencias temporales:

$$w_{t+1} = w_t + \alpha\left[r_{t+1} + \gamma\hat{q}(s_{t+1}, a_{t+1}, w) - \hat{q}(s_t, a_t, w)\right]\nabla\hat{q}(s_t, a_t, w_t)$$

donde $\nabla\hat{q}(s_t, a_t, w_t) = x(s, a)$, ya que habíamos definido $\hat{q}$ como una combinación lineal de las características y los pesos. Siempre y cuando la tasa de aprendizaje, γ, decrezca con el tiempo y sea pequeña, el algoritmo está garantizado que converja a un máximo global (Sección 9.3 de [1]).

Con esta regla de actualización, podemos definir un algoritmo general para estimar $Q(s, a)$ con una aproximación lineal de funciones, presentado en el Algoritmo 1. Cabe destacar que hemos introducido una pequeña variación en la expresión del error de diferencias temporales. El objetivo de la expresión $r + \gamma\,\text{máx}_{a'}\,\hat{q}(s', a', w) - \hat{q}(s, a, w)$ es obtener la diferencia entre el valor objetivo (el "target"), que en este caso es $r + \gamma\,\text{máx}_{a'}\,\hat{q}(s', a', w)$ (es decir, sumar la recompensa actual y el valor obtenido tras tomar aquellas acciones que maximizan los sucesivos retornos) y el valor obtenido, $\hat{q}(s, a, w)$.

Algorithm 1: Aproximación lineal de funciones general para aprender $\hat{q} \approx Q$

```
Initialize weights vector w = (w_1, w_2, ..., w_n) randomly between [0, 1];
foreach episode do
    Set state to initial one: s ← s_0;
    while s is not terminal do
        Get action from policy: a ← π(a, s);
        Take action a, observe reward r and next state s';
        w ← w + α[r + γ máx_{a'} q̂(s', a', w) − q̂(s, a, w)] ∇q̂(s, a, w)
```

2.6. Q Learning

Q-Learning es un algoritmo de Aprendizaje por Refuerzo, que busca estimar la función Q a través de la experiencia para encontrar una política óptima. Esta función Q es la función de valor estado-acción que hemos introducido en apartados anteriores. A partir del valor óptimo, q_*:

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_a q_*(S_{t+1}, a') \middle| S_t = s, A_t = a\right],$$

el algoritmo estima la función Q, a través de la siguiente regla de actualización [4]:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma max_{a'} Q(s',a') - Q(s,a)\right],$$

donde α es la tasa de aprendizaje.

Para estimar esta función Q, el algoritmo clásico utiliza una tabla de tamaño $S \times A$, es decir, número de estados por número de acciones. En cada entrada en la tabla, se almacena el valor Q correspondiente a ese estado y esa acción. El agente, según va pasando por diferentes estados y diferentes acciones, actualiza las entradas de la tabla iterativamente, convergiendo al valor Q real cuando $t \to \infty$.

2.6.1. Deep Q Learning

El método clásico de Q Learning genera un problema evidente. Cuando tratamos un problema real, y el espacio de estados y/o el de acciones se convierten en espacios gigantes, se vuelve inviable en términos de memoria mantener una tabla tan grande. Es por ello, que en la práctica, el problema se traslada al campo del Deep Learning, donde en vez de calcular la función Q real, se utiliza un aproximador funcional, para estimarla. Para ello, se utiliza el método de Deep Q Learning (DQN).

Ahora, en vez de estimar los valores Q de nuevos estados, a partir de los anteriores, utilizamos una red neuronal profunda, que aproxima la función Q. La ventaja de este método, es que ya no necesitamos una tabla de tamaño $S \times A$, tan solo necesitamos una red neuronal con S neuronas de entrada, las neuronas ocultas que decidamos utilizar, y A neuronas de salida.

2.6.2. Double Deep Q Learning

Con el algoritmo original de Q Learning surgieron varios problemas que llevaron a la comunidad científica a buscar variaciones. La primera que apareció fue la variante del Double Deep Q Learning [26], de *Hado van Hasselt*. Este algoritmo se presentó como una solución al problema de sobreestimación presente en el Q Learning clásico. Consideremos el valor objetivo de la función Q:

$$Q^*(s,a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q^*(s',a') \Big| s,a]$$

En concreto, el problema reside en:

$$\max_{a'} Q^*(s',a')$$

Q Learning es un algoritmo que utiliza estimaciones para calcular otras. Es por ello, que en las fases tempranas de entrenamiento, cuando hay mucho ruido y apenas se han visto ejemplos que hagan que estas estimaciones sean fiables, se está introduciendo una sobreestimación en esa expresión. Esto hace que nuestro sistema introduzca un sesgo hacia los valores máximos en el aprendizaje, lo cuál es un problema muy importante.

Para explicarlo mejor, consideremos un ejemplo: un estado tiene todos los valores Q reales de sus acciones a 0. Sin embargo, los valores estimados están distribuidos por encima de 0. Al tomar el valor máximo de estas estimaciones, estamos actualizando nuestras estimaciones con un valor positivo, lo que hace que estemos sobreestimando la función Q.

La solución propuesta en [26], Double Deep Q Learning, propone la utilización de un doble estimador, donde cada uno es usado para actualizar el otro. En concreto, tenemos un modelo Q, y un modelo objetivo Q'. El algoritmo usa Q' para seleccionar la acción, y Q para evaluar esa acción (estimar los valores Q de la acción). La formulación es la siguiente:

$$Q^*(s_t,a_t) \approx r(s_t,a_t) + \gamma Q(s_{t+1}, \arg\max_{a_{t+1}} Q'(s_t,a_t))$$

El modelo Q se encarga de estimar $\arg\max_{a_{t+1}} Q'(s_t,a_t)$, y el modelo Q' utiliza el valor anterior para estimar $Q(s_{t+1}, \arg\max_{a_{t+1}} Q'(s_t,a_t))$.

Fujimoto et al. profundiza aún mas y propone la variación Clipped Double Q Learning en [27]. Se sigue la propuesta original del Double Deep Q Learning. De nuevo, se calculan dos estimaciones por separado del valor Q real. La distinción se encuentra al calcular Q^*. Para ello, se coge el mínimo de los dos valores Q de los dos estados siguientes (junto a sus respectivas acciones), producidos por los dos estimadores independientes. Cogiendo el mínimo de los valores Q estimados, evitan la sobreestimación. Este método tiene una ventaja importante, mediante esta minimización, se actualizan antes estados con baja varianza en sus estimaciones, lo que lleva a obtener políticas más controladas, y a realizar actualizaciones más seguras.

2.6.3. Dueling Deep Q Learning

Con el algoritmo clásico de Q Learning surge otro nuevo problema. Si conocemos que un estado es intrínsecamente malo, ¿por qué deberíamos estimar el valor de las acciones,

dado ese estado? En el contexto de los videojuegos, un ejemplo muy claro es un estado en el que sin importar la acción que tomemos, nuestro personaje va a morir.

Para atajar este planteamiento, se propone el algoritmo de Dueling Deep Q Learning [28]. De nuevo, la solución vuelve a ser dividir la tarea a realizar en dos estimadores. Uno de ellos se encarga de estimar el valor de un estado, y el otro se encarga de estimar el valor de una acción, dado ese estado.

La arquitectura del modelo es una red neuronal, donde en la entrada hay una serie de capas ocultas, al igual que en una arquitectura clásica de Deep Q Learning. A partir de ahí, dividimos la red en dos flujos diferentes: uno para cada estimador. Por último, añadimos otra capa que se encarga de agregar los resultados de ambos estimadores. Como la entrada y la salida de la red neuronal, es la misma que la del modelo clásico del Deep Q Learning, podemos entrenar este modelo igual que hacíamos con el original. Es decir, se puede realizar un descenso por gradiente, minimizando una función de error cuadrático, y la formulación será idéntica a la del problema original.

Volviendo a la agregación final, ésta puede realizarse de diferentes maneras. En [28] se propone primero realizar una agregación simple, sumando ambos resultados:

$$Q(s_t, a_t) = V(s_t) + A(s_t, a_t)$$

Sin embargo, los autores acaban identificando a la suma simple como una agregación problemática, ya que sumarlos directamente hace que, a partir del resultado Q, no se pueda recuperar el valor de V y A por separado. En los experimentos, muestran como esta falta de "trazabilidad" resulta en unos resultados muy pobres.

Por tanto, se propone una nueva agregación para la capa final, donde se fuerza que el valor Q sea igual a V, en el caso de que la acción sea aquella con el valor máximo A:

$$Q(s_t, a_t) = V(s_t) + (A(s_t, a_t) - \max_{a' \in |A|} A(s_t, a_t))$$

El artículo profundiza en nuevas agregaciones, pero para el propósito de este trabajo, con las que hemos definido es suficiente para dar contexto al método del Dueling Deep Q Learning.

3

Sistema, diseño y desarrollo

El objetivo de este trabajo es implementar tres crawlers enfocados, uno básico, y dos basados en técnicas de Aprendizaje por Refuerzo, y comparar su eficacia. En este capítulo, se definen todos los sistemas desarrollados para la implementación de estos tres crawlers.

Se ha realizado un considerable trabajo de desarrollo, implementando todos los sistemas desde cero, con una excepción importante, donde se han usado recursos externos, al construir el agente en el caso del crawler basado en Deep Q-Learning, utilizando la librería de Keras para RL [29].

3.1. Definición del problema

Antes de empezar a detallar los sistemas que se han diseñado e implementado para el desarrollo del trabajo, es importante definir claramente cuáles son los objetivos, limitaciones y en general, poner en contexto el problema a resolver.

Partimos de una idea base: queremos diseñar un crawler enfocado que, utilizando algoritmos de Aprendizaje por Refuerzo, realice una búsqueda sesgada en la web, obteniendo páginas relevantes a un determinado tema. Es decir, utilizaremos las técnicas propias del Aprendizaje por Refuerzo para hacer que el crawler asigne más prioridad a aquellas páginas que son más relevantes para el tema (llamado **topic** de aquí en adelante). Para poder identificar las limitaciones que vamos a tener que asumir, es importante definir desde este momento qué técnicas de Aprendizaje por Refuerzo vamos a usar, ya que dependiendo de qué algoritmo escojamos necesitaremos asumir unas limitaciones u otras.

En la literatura, no hay demasiado contenido sobre la integración del Aprendizaje por Refuerzo en tareas de crawling web. Encontramos 2 propuestas: una proponiendo un

crawler genérico para la Deep Web, que aproxima una función Q, utilizando definiciones propias de pesos y recompensas [9]; y otra proponiendo un crawler enfocado que usa una aproximación lineal de funciones para calcular la prioridad de los enlaces [2]. La primera propuesta se vuelve demasiado compleja matemáticamente, pues utilizan métodos estadísticos avanzados para calcular la recompensa, el valor de las funciones (utilizan una función de kernel para calcular la distancia entre dos acciones, utilizando modelos estadísticos), etc. Además, la segunda propuesta plantea una tarea que podemos adaptar mejor al trabajo: maximizar la relevancia descubierta por un crawler enfocado. Es por ello que vamos a tomar este trabajo como modelo a seguir, ya que define un sistema de manera más clara y sencilla. Por tanto, vamos a implementar un crawler sencillo que nos sirva como baseline a la hora de comparar, que no utilice estrategias de RL; un crawler basado en aproximación lineal de funciones (ALF), basándonos en la propuesta de [2], y por último, uno que utilice Q-Learning. Es decir, un algoritmo base, uno ya propuesto y probado en la literatura, y uno novedoso que no se ha propuesto (hasta donde hemos encontrado) o probado en la literatura.

Hay varias motivaciones que nos llevan a utilizar el algoritmo de Q-Learning. Además de tratarse de uno de los principales algoritmos de Aprendizaje por Refuerzo, previamente a la realización de este trabajo se ha desarrollado otro Trabajo Fin de Máster titulado "Recomendación basada en Procesos de Decisión de Markov". En ese trabajo, se aplicó el algoritmo de Deep Q-Learning (una variante al Q-Learning que utiliza una red neuronal como aproximador) a un contexto de recomendación. Es por ello que una motivación adicional para utilizarlo es seguir investigando su aplicación a otras áreas como, en este caso, el crawling web. Esta elección supone una propuesta novedosa de un Crawler Enfocado basado en Deep Q-Learning.

Los dos algoritmos de RL que vamos a usar ya fueron introducidos en las Secciones 2.5 y 2.6, (Aproximación Lineal de Funciones y Deep Q-Learning, respectivamente). En el primero se utiliza un vector de pesos que se actualiza iterativamente para ser combinado linealmente con un vector de características en la estimación del valor de las acciones y estados (la función Q). En el segundo, se usa una red neuronal para estimar unos pesos, que en este caso, conforman una combinación no lineal con el vector de características para realizar la estimación de la función Q.

3.2. Sistema de representación de páginas

El primer paso para realizar la implementación de nuestro crawler es definir el entorno con el que va a interactuar. En un crawler simple, el entorno puede ser la propia web. El crawler, por tanto, debería implementar métodos para realizar el fetch de una página a partir de su URL, procesar los enlaces a partir de un código HTML, y gestionar la frontera con estos datos. Sin embargo, no tenemos por qué usar datos en vivo de la propia web, sino que podemos usar una base de datos ya construida. Este es el caso de los Dumps de Wikipedia[1], donde se almacenan diferentes bases de datos de

[1] `https://dumps.wikimedia.org/`

artículos de Wikipedia. En nuestro caso, vamos a usar el dump correspondiente a la Wikipedia de Inglés Simple (Dump of Simple English Wikipedia), presente en el enlace `https://dumps.wikimedia.org/simplewiki/20201001/` (el dump lo van actualizando cada cierto número de días, nosotros usamos la versión correspondiente al 1 de Octubre de 2020).

Como punto de partida, el Dump se representa con 3 archivos:

- Un archivo XML, donde cada artículo tiene una representación XML, que incluye metadatos y el contenido del propio artículo. Los autores del dump lo llaman *simplewiki-20201001-pages-articles-multistream.xml*, pero nosotros lo renombramos a *dump.xml*.
- Un archivo de índice, donde en cada línea se incluye el índice y el título de cada página. Los autores lo llaman *simplewiki-20201001-pages-articles-multistream-index.txt*, pero nosotros lo renombramos a *index.txt*.
- Un archivo de enlaces, en formato SQL. Este archivo contiene una serie de consultas y operaciones SQL, que deben ser ejecutadas en un motor de base de datos, para construir una base de datos que contiene a los enlaces, en una tabla con 4 columnas: id de la página origen, espacio de nombres de la página origen, título de la página destino, y espacio de nombres de la página destino. Los autores lo nombran *simplewiki-20201001-pagelinks.sql*.

Aunque en estos 3 archivos ya está contenida toda la información que necesitaremos en nuestro crawler, el formato de los mismos no queda demasiado limpio. Además, se incluye mucha información que nosotros no vamos a utilizar, ya que tan solo estamos interesados en el contenido de los artículos; en sus índices y títulos, para poder identificarlos; y en la estructura de enlaces, pudiendo identificar de manera sencilla a qué páginas apunta cada artículo. Es por ello que antes de empezar a implementar el crawler, hemos realizado una fase de preprocesamiento de los datos, para poder partir de un conjunto de datos con una estructura más sencilla, que facilite el trabajo de desarrollo posterior.

Preprocesamiento del dump

Como fase previa a la ejecución del crawler, realizamos un pequeño preprocesamiento del dump, para poder tener un conjunto de datos con un formato que nos facilite su manejo. Lo primero que tenemos en cuenta es la distribución de los identificadores. Para ahorrarnos complejidad en el programa, normalizamos todos los identificadores al rango $[0, N]$, siendo N el número de páginas totales. Para hacer esto, modificamos el fichero de índice, transformando todos los identificadores.

Después, limpiamos el archivo XML del dump, donde están almacenadas las páginas. Es decir, eliminamos todos los metadatos que no nos interesan, y dividimos cada página en un fichero aparte. Lo hacemos así para que sea más fácil luego a nivel de programación acceder a una página concreta y, además, porque se parece más a cómo accederíamos a las

páginas en un crawling normal en el que recurrimos a la web. En estos casos, lo normal es acceder a cada página por separado, y almacenar los HTML que descargamos de la web en local.

En cuanto a su representación XML, cada página es un objeto XML con 3 campos: identificador (id), título (title), y contenido (text). Una página de ejemplo sería la siguiente:

```
<page>
<id>0</id>
<title>Pagina de ejemplo</title>
<text>Esta es una pagina de ejemplo, cuyo id es 0, y titulo es "
   ↪ Pagina de ejemplo"</text>
</page>
```

Cabe destacar, que este dump es nuestro entorno, de manera análoga a como la web sería el entorno en un crawler normal. Es por ello, que el título de la página nos sirve de URL, ya que no hay 2 páginas diferentes con el mismo título, y además, en nuestro directorio, almacenamos cada página en un archivo diferente, cuyo nombre es "<título>.xml".

Por último, tenemos que transformar también el fichero de enlaces. Recordemos que estos enlaces se daban en forma de instrucciones SQL, lo cual, para nuestro cometido, no nos sirve realmente. Lo que necesitamos nosotros es un fichero de enlaces, donde en cada línea representemos un enlace. En nuestro caso, en cada línea hay 2 números, separados por un tabulador, indicando la página origen, y la página destino. Además, filtramos todos los enlaces cuyas ambas páginas no pertenecen a nuestro dump. Aunque en el dump original esto puede no tener mucho sentido, sí lo tiene para aquellos dumps que son particiones del original. Esto es porque nos interesa hacer particiones, para poder utilizar en los experimentos conjuntos de datos de un tamaño más reducido.

Tras este preprocesamiento, acabamos con un directorio de dump con la siguiente estructura:

- Un directorio "xml", donde almacenamos cada página en un archivo diferente. Cada archivo sigue la estructura descrita en esta misma sección.
- Un archivo "index.txt", donde cada línea se compone de 2 campos: el índice numérico y el título de la página, separados por un tabulador. Estos índices están todos en el rango $[0, N]$.
- Un archivo "links.txt", donde cada línea se compone de 2 índices, separados por un tabulador: primero, el de la página origen, y después, el de la página destino. Todas las páginas que aparecen en estos enlaces son páginas de nuestro dump.

Este dump sirve de entrada a cualquiera de los crawlers implementados, que lo utilizarán para acceder a las diferentes páginas.

3.3. Sistema de vectorización de páginas

Necesitamos modelar las páginas como vectores por varias razones, siendo la más directa el propio uso de los algoritmos de LFA y Deep Q-Learning. En el primero, necesitamos modelar los estados y acciones con características (formalizaremos más adelante las páginas y los enlaces como estados y acciones, respectivamente), pero también en el segundo caso usamos una red neuronal que recibe a la entrada el estado actual, por lo que si lo definimos como un vector, es directo y trivial introducirlo a la red. Tanto las características de las páginas como de las acciones deben reflejar el conocimiento que el crawler ha adquirido en su proceso secuencial. Debemos incorporar información sobre el entorno que los rodea, por ejemplo, qué páginas les apuntan (sus padres en el grafo web); pero también información sobre su contenido, como la relevancia con respecto al topic.

3.3.1. Extracción de características de las páginas y enlaces

A la hora de extraer características de una página, el crawler puede utilizar el contenido de la propia página (su texto HTML), y también información contextual sobre las páginas que la rodean (según descubre nuevas páginas, el crawler lleva la cuenta de los padres de cada una de ellas). Con esta información podemos modelar unas características que representen con más precisión el grado de relevancia de la página con respecto al topic. Cabe destacar que aunque utilizamos un valor continuo para la relevancia con respecto al topic, habrá algunas situaciones en las que necesitemos una decisión discreta: la página es relevante, o no es relevante. En estos casos, consideramos que una página es relevante al topic si el texto de la página contiene la palabra clave del topic.

Cabe recalcar que todas las características que vamos a definir están extraídas de la propuesta realizada en [2], donde los autores proponen este diseño de características. El planteamiento teórico de las mismas está basado en su propuesta, pero los detalles de implementación son todos propios de este trabajo, como son la forma de calcular las medias, o los métodos de discretización.

- Relevancia de la página con respecto al topic. Calculamos la similitud coseno entre el vector W2V (Word2Vec) de la página (Apartado 3.3.2) y el vector W2V del topic. Este valor está comprendido entre -1 y 1.
- Relevancia con respecto a unas categorías. Para cada topic, podemos predefinir unas categorías. En general, son palabras clave que están relacionadas con el ámbito del topic. Incluir este aspecto puede orientar al crawler hacia páginas relevantes para el topic, ya que si conseguimos que el crawler esté siempre trabajando con páginas relacionadas con su ámbito, es más probable que acabe visitando algunas relevantes. Para cada categoría calculamos la relevancia de manera análoga a como hacemos con el topic, y las añadimos una a una a nuestro vector de características.
- Relevancia media de los padres. Calculamos la relevancia media de los padres mediante una media incremental, para optimizar su cálculo. Así, cada vez que se

descubre un nuevo padre, añadimos su relevancia a la media mediante la siguiente actualización:

$$\mu_t = \mu_{t-1} + \frac{r_n - \mu_{t-1}}{n},$$

siendo μ_t la media en el instante t, r_t el nuevo dato de la relevancia, y n el número de padres actualizado. Este es otro valor numérico entre -1 y 1, ya que la media de números que están entre -1 y 1, está también entre esos límites.

- Relevancia media de los padres relevantes. La calculamos de manera análoga al caso anterior, pero esta vez, teniendo en cuenta sólo aquellos padres que son relevantes al topic.

- Distancia con la última página ancestro relevante. Definimos una distancia máxima para evitar distancias demasiado grandes. En nuestro caso, definimos una distancia máxima de 9 unidades. Para calcular esta distancia, tenemos en cuenta si la página actual es relevante. Si lo es, la distancia es 0, si no, utilizamos las distancias de sus padres. Si no es relevante, y no tiene padres, la distancia es la máxima. Si tiene padres, escogemos la menor de sus distancias. Esta característica también ayuda a orientar al crawler hacia páginas relevantes, pues cuanto más cerca estemos de otras páginas que ya se han descubierto como relevantes, más probabilidad habrá de llegar a otras también relevantes.

- Cambio de relevancia con respecto al topic. La relevancia de la página actual se compara con la relevancia media ponderada de sus padres. Esta relevancia media se calcula de nuevo de manera incremental, aunque esta vez utilizamos el método *exponential smoothing*. Este método suaviza los pesos de las observaciones pasadas, haciendo que pierdan importancia con respecto a las más recientes. En este método, la media μ en el instante t se calcula de la siguiente manera:

$$\mu_t = \beta x_i + (1 - \beta)\mu_{t-1},$$

donde β es un factor de suavizado comprendido entre 0 y 1. Aplicando este método a nuestro cálculo de la relevancia media ponderada:

$$wrl(x) = \beta * rl(x) + (1 - \beta) * \max_{x' \to x} wrl(x'),$$

donde $rl(x)$ es la relevancia de la página x con respecto al topic, y $wrl(x)$ es la media ponderada de la página x. Si la página tiene múltiples padres, se escoge la relevancia media ponderada máxima de entre todos ellos. Si no tiene padres, la media debe ser igual a la relevancia de la propia página, es decir:

$$wrl(x) = rl(x)$$

Con este cálculo realizado, podemos definir el cambio en la relevancia con respecto al topic como:

$$\nabla rl(x) = rl(x) - \max_{x' \to x} wrl(x'),$$

siendo $\nabla rl(x)$ el cambio en la relevancia de la página x. Si la página no tiene padres, este cambio será nulo.

De nuevo, el uso de esta característica es interesante pues ayuda a orientar al crawler hacia páginas relevantes. Si la relevancia de la página es mucho menor que la de sus padres, estaremos alejándonos de las páginas que nos interesan, y el crawler podrá detectarlo, ya que recibirá números muy pequeños (negativos) en esta característica.

En cuanto a los enlaces, calculamos sus características como un subconjunto de las anteriores. Con ellos, en cuanto a su contenido, no contamos con un texto HTML, sino que utilizamos la URL del enlace, su texto ancla (*anchor text*), y el texto que lo rodea. En este caso, estamos calculando realmente unas características que buscan predecir la relevancia de la página a la que apunta el enlace, antes de obtenerla. En concreto, para los enlaces calculamos las siguientes características: Relevancia con respecto al topic, Relevancia con respecto a las categorías, Relevancia media de todos los padres, Relevancia media de los padres relevantes. Para ello, construimos un texto juntando los 3 campos que ya hemos mencionado, y lo vectorizamos utilizando los vectores W2V de las palabras que lo forman (Apartado 3.3.2). Es este vector el que usamos para calcular la relevancia con respecto al topic y a las categorías.

También cabe destacar la forma en la que se computan estas características a lo largo del proceso de crawling, ya que la optimización es necesaria para ajustar los tiempos de ejecución. Hay que distinguir entre características estáticas y dinámicas. Las primeras, dependen solo de la información de la página o enlace, y por tanto, solo necesitamos calcularlas una vez, pues su valor no va a variar aunque el crawler obtenga más conocimiento sobre el contexto de la página. Las segundas, sí dependen de este contexto, y por tanto, su valor cambia a medida que el crawler descubre nuevos ancestros para estas páginas. Este es el caso de las medias, tanto para todos los padres como los padres relevantes, como para el cambio en la relevancia, y la distancia hasta la última página relevante. En estos casos, se actualizan sus valores de manera incremental con el último ancestro descubierto. Así, minimizamos los cálculos, y obtenemos una extracción de características optimizadas.

Discretización de variables

Todas estas características que hemos definido son números continuos (a excepción de la distancia a la última página relevante, que puede interpretarse como una variable discreta). Es por ello que, tras calcularlos, a la hora de ser usados en el crawler LFA son discretizados. En concreto, la función que usamos para discretizar aquellos valores que son relevancias (relevancia del topic, de las categorías, y las medias de los padres) es la siguiente:

$$discretize(x) = \begin{cases} 0, & \text{if } x < 0, \\ 9, & \text{if } x = 1, \\ int(10x) & \text{otherwise} \end{cases}$$

Es decir, los números negativos se discretizan todos con 0 (para reducir el número de valores que la variable pueda tomar), y para los números positivos se toma la parte entera de su valor multiplicado por 10. Así, el rango $[0,0{,}1)$ se discretiza con un 0, el $[0{,}1,0{,}2)$ con un 1, etc. El caso especial para $x = 1$ se incluye para evitar que el valor 1 se discretize con un 1, en vez de con un 9, que es el asociado al rango $[0{,}9,1)$, aunque se puede eliminar este supuesto, ya que prácticamente es imposible que la similitud sea 1, pues hay mucha variabilidad en los vectores de las páginas, y es muy poco probable que haya una igualdad exacta entre los vectores W2V de la página y el topic.

En cuanto a la distancia a la última página relevante, utilizamos la siguiente función para discretizar su valor:

$$discretize(d) = d_{\max} - d$$

Aquellas páginas que estén muy lejos de las páginas relevantes tienen una distancia $d = d_{\max}$. Sin embargo, en las características que son relevancias, un mayor valor significa una mayor probabilidad de relevancia. Sin embargo, aquí es justo al revés: cuanto menor sea la distancia, mayor es la probabilidad de que esa página sea relevante o pueda llevarnos a otras relevantes. Es por ello que invertimos la escala: hacemos que las distancias más pequeñas se discreticen con los valores más altos.

Por último, necesitamos discretizar también el cambio en la relevancia. Recordemos que este cambio es $\nabla rl(x) = rl(x) - \text{máx}_{x' \to x}\, wrl(x')$. Ambos términos de la resta están comprendidos entre -1 y 1, y por tanto, el resultado está comprendido entre -2 y 2. Por tanto, lo dividimos en intervalos de 0,5 unidades, y asignamos valores de 0 a n. Es decir, rango $[-2,-1{,}5)$ tienen el valor 0, $[-1{,}5,-1)$ el valor 1, hasta $[1{,}5,2]$ que recibe el valor 7. De nuevo la justificación de esta discretización es asignar valores discretos más grandes a aquellos cambios en la relevancia que indiquen que vamos en la buena dirección: si el cambio es positivo y grande, significa que la página actual es más relevante que sus padres, y por tanto, vamos en buena dirección; si por el contrario, el cambio es pequeño y negativo, la página es actual es mucho menos relevante que sus padres, y por tanto, nos estamos alejando de la relevancia y estamos yendo en mala dirección.

3.3.2. Extracción de vectores W2V de las páginas

Los vectores de características los usamos como entrada a los algoritmos de resolución de MDPs que vamos a utilizar. Sin embargo, a la hora de calcular la recompensa de las páginas, lo que queremos es utilizar una heurística que nos indique cómo de relevante es la página con respecto a un topic. Para ello, queremos usar una función de similitud ampliamente conocida en el campo de la Recuperación de Información y la Minería Web: la similitud coseno. Si tenemos el vector de la página, y el vector del topic, podemos definir una recompensa para el agente de RL de manera trivial: el coseno entre ambos vectores. Además, para este caso, podemos utilizar un modelo Word2Vec (Sección 2.2.3). Para ello,

partimos de unos vectores W2V ya pre-entrenados[2], extraídos con el algoritmo GloVe [30]. Se incluyen versiones de 50, 100 y 200 dimensiones para cada vector. Para calcularlos, los autores utilizaron una versión del Dump de Wikipedia en inglés de 2014, por lo que su uso en nuestro crawler está justificado. Además, las palabras que tienen definido un vector W2V componen nuestro vocabulario, dejando fuera a aquellas que no nos van a aportar información (signos de puntuación, números demasiado largos, stopwords, etc.).

Por tanto, definimos el vector del topic como el vector W2V correspondiente a la palabra que lo define. Para las páginas, necesitamos utilizar un modelo que extraiga el vector de la página a partir de los vectores de las palabras que contiene el propio documento. Hay diversas formas de agregar los vectores de las palabras, aunque el más lógico es utilizar la media de todos ellos. Sin embargo, necesitamos ponderar más alto aquellas palabras que aparezcan más veces en el documento, y bajar la ponderación de aquellas palabras que no aporten demasiada información (por ejemplo, aquellas palabras que aparecen en todos los documentos de la colección). Estos supuestos ya se incluyen en el famoso modelo **tf-idf**, (*term frequency - inverse document frequency*, en inglés).

Normalmente, este modelo se estudia en el ámbito de los buscadores web, donde se parte de una colección que es previamente conocida. Sin embargo, en el contexto de una tarea de crawling no conocemos todas las páginas que el crawler se va a encontrar a lo largo del tiempo. Es importante comprender que aunque nosotros tengamos en local el dump entero de Wikipedia, el crawler no lo conoce por completo de partida. Lo correcto es simular el conocimiento que el crawler va adquiriendo con el tiempo: en los primeros pasos apenas ha visto unas pocas páginas, y descubre una página más en cada paso. Esto tiene que verse reflejado en el cálculo de los vectores de las páginas, pues no sería correcto que estuviéramos filtrando información para calcular esos vectores que el crawler todavía desconoce.

Podemos ejemplificar este problema: supongamos que el crawler parte de una página semilla, que contiene el siguiente texto: "la mineria web es muy interesante". Cuando el crawler se encuentra este documento, necesita vectorizarlo, y por tanto, necesita calcular los pesos tf-idf de cada palabra que aparece en él. Si usáramos unos pesos tf-idf calculados con el dump entero, el crawler estaría haciendo uso del contenido de todas las páginas del dump, unos documentos que todavía no se ha encontrado y, por tanto, no puede conocer. En general, podemos asumir que hasta que el crawler no vea un gran número de páginas, no podrá tener un cálculo fiable de los pesos tf-idf. Para resolver este problema, podríamos utilizar dos aproximaciones: actualizar el cálculo de los tf-idf de las palabras de manera incremental, agregando el nuevo conocimiento que el crawler obtiene cada vez que ve un nuevo documento, o utilizar un conjunto de datos diferente para pre-entrenar estos pesos tf-idf de las palabras. En nuestro caso, utilizamos esta segunda aproximación, pues nos facilita el trabajo, y podemos esperar que obtengamos buenos resultados. Además, no estamos filtrando información que el crawler aún no puede conocer, pues estamos usando un conjunto de datos diferente (es razonable asumir que un crawler enfocado haya sido pre-entrenado con una colección de menor tamaño). Para ello, usamos una partición aleatoria del 10 % de las páginas de nuestro dump, y utilizamos esas páginas para pre-calcular los idfs de todas las palabras de nuestro vocabulario. Pre-calculamos solo los idfs,

[2] `https://nlp.stanford.edu/projects/glove/`

pues los tf dependen únicamente del documento para el que se están calculando, y por tanto, el crawler puede calcularlos en tiempo real cuando obtiene una página que deseamos vectorizar.

Una vez aclarado cómo calculamos los tf-idf de las palabras de nuestro vocabulario, podemos ya definir el vector W2V de una página, d, como:

$$V_d = \sum_{w_i \in d} tfidf(w_i, d) * V_{w_i},$$

siendo V_{w_i} el vector de la palabra en la posición i en nuestro vocabulario. Estos vectores se calculan una sola vez, guardándolos en memoria en el momento en el que se visita una página por primera vez. Cuando en posteriores pasos el crawler necesite recurrir al vector de esa página, se carga directamente desde memoria, sin tener que volver a calcularlo.

3.4. Implementación de la frontera

Típicamente, los crawlers enfocados usan una estructura de datos específica para representar la frontera: una cola con prioridad, (PQ, de *priority queue*, en inglés). Implementar este tipo de datos es trivial en Python, utilizando la librería heapq, que ya resuelve eficientemente las operaciones de inserción y extracción de elementos en la cola. Para el crawler básico, con esta estructura es suficiente. Sin embargo, para los crawlers basados en RL, utilizamos una política ϵ-greedy para seleccionar una acción en cada instante de tiempo. Esta política, extrae una acción de la frontera cuyo valor sea máximo con probabilidad $1-\epsilon$, o una acción aleatoria con probabilidad ϵ. Esto nos suscita el primer problema de diseño: las colas con prioridad no incorporan ningún mecanismo para escoger elementos aleatoriamente, ya que el propio diseño de la estructura lo hace imposible. Además, en nuestro crawler, actualizaremos las prioridades de los enlaces según nuestro MDP vaya aprendiendo y ajustando mejor la función Q, lo que nos lleva a otro problema, no es trivial actualizar las prioridades de aquellos elementos que ya han sido añadidos a la cola. Es por esto, que tenemos que diseñar e implementar una cola con prioridad que incorpore al diseño original estas dos nuevas funcionalidades: extracción aleatoria de elementos, y actualizaciones de prioridades de elementos ya insertados.

Updatable and Randomized Priority Queue (URPQ)

Llamamos *Updatable and Randomized Priority Queue* (URPQ), a la implementación de una cola con prioridad que incorpore mecanismos para la actualización de prioridades de elementos ya encolados, y la selección aleatoria de ellos. Partimos de una cola con prioridad ya implementada con la librería heapq de Python. En la documentación oficial de la librería, ya se incluye una posible implementación que permite la actualización de prioridades. Como no hay manera de buscar eficientemente un elemento en la cola, ellos proponen mantener un diccionario de elementos, para poder accederlos en O(1).

Así, a la hora de querer actualizar un elemento ya insertado, lo que realmente hacen es marcar el elemento ya insertado como eliminado en la cola, eliminándolo del diccionario, e insertando uno nuevo con la prioridad actualizada. A la hora de recorrer la cola para extraer un elemento, se ignoran aquellos que han sido marcados como eliminados.

Además, gracias a este diccionario, podemos implementar la selección aleatoria de manera directa. Nos basta con escoger una de las claves del diccionario de manera aleatoria. Otro problema de esta selección aleatoria era eliminarlo efectivamente de la cola. Pero gracias al mecanismo de marcar eliminados, podemos extraer una clave aleatoria del diccionario (los elementos son las claves), y marcarlo como eliminado. Así, a la hora de extraer ese elemento, no se tendrá en cuenta, y a efectos prácticos, se habrá eliminado de la cola.

Gracias a estas modificaciones, ahora contamos con una frontera que puede ser utilizada con políticas ϵ-greedy de manera trivial.

3.5. Implementación de un crawler básico

El objetivo de este trabajo es integrar técnicas de Aprendizaje por Refuerzo en el ámbito del crawling web, y poder realizar un análisis de eficacia en términos de descubrimiento de relevancia. Es por ello que el primer paso es implementar un crawler sencillo que nos sirva como punto de partida tanto en implementación, pudiendo añadir mayor complejidad en su diseño para incluir el uso del Aprendizaje por Refuerzo, como en resultados y evaluación, marcando el rendimiento de este crawler como el punto de partida a batir.

Lo primero es definir de qué se compone nuestro crawler, para después explicar como utiliza estas piezas para hacerlo funcionar. Nuestro crawler tiene 3 componentes básicos:

- Un entorno web, donde el crawler busca las páginas. En nuestro caso, utilizamos la clase Fetcher, que implementa 2 funcionalidades: extraer el contenido de una página a partir de una URL, y extraer los enlaces de una página a partir de su contenido. Este Fetcher es una clase general, que puede implementarse tanto para la web, como para nuestro dump.
- Una frontera. En nuestro caso, como estamos implementando un crawler enfocado, utilizamos una cola de prioridad. Los enlaces que el crawler va descubriendo reciben una prioridad, basada en una estimación de la relevancia de la página con respecto a un topic.
- Una condición de parada. Puede basarse en un simple límite de páginas obtenidas, número de páginas relevantes, etc.

Con estos elementos, podemos implementar directamente un crawler básico. Sin embargo, la complejidad de los crawlers enfocados reside en cómo calcular la prioridad que asignamos a las páginas. Además, esta prioridad hay que asignársela antes de poder

ver su contenido, ya que es cuando se descubre el nuevo enlace el momento en el que se añade a la frontera. Hay muchas propuestas sobre cómo calcular esta prioridad. Algunas se basan en calcular una similitud del topic con el contenido disponible en el momento de descubrir un enlace, (el anchor text del enlace, la propia URL, el texto cercano a la URL) [31] [32], mientras que otras tienen en cuenta métricas sobre los enlaces, como PageRank [33] [34].

Para el caso de nuestro crawler básico, nuestra prioridad va a ser la similitud coseno entre el vector W2V del topic, y el del texto que construimos a partir de los enlaces (utilizando sus URL, textos ancla, y el texto que rodea al enlace en la página). En el Algoritmo 2 detallamos como funciona nuestro crawler básico.

Algorithm 2: Crawler Enfocado Simple

```
Data: Dump D, topic t, seed S, limit L
Result: Path of crawled pages, r
Initialize number of crawled pages, N = 0.;
Initialize fetcher with dump, fetcher = Fetcher(D);
Initialize frontier with PQ, frontier = PriorityQueue();
foreach page ∈ S do
    outlinks = fetcher.process_links(page);
    foreach link ∈ outlinks do
        frontier.add(link, link.priority())
while N ≤ L do
    url ← frontier.pop();
    page ← fetcher.fetch(url);
    outlinks ← fetcher.process_links(page);
    foreach link ∈ outlinks do
        frontier.add(link, link.priority())
    Save page in storage;
    N ← N + 1;
```

En este algoritmo se usan métodos de 2 clases: Fetcher y Frontier. El Fetcher es una interfaz, o clase abstracta, que define 2 métodos:

- Fetch. Realizar el fetch de la página, a partir de un elemento identificador, como es típicamente la URL. Si el Fetcher trabaja con la web, haría un fetch con una petición HTTP. Si trabaja con el dump, va a disco a buscar la página web.
- Process_urls. A partir del texto de la página, extrae los enlaces.

Por otro lado, la frontera es una PriorityQueue, que debe implementar tres funcionalidades: un método *push*, que añade un elemento a la cola, con una prioridad pasada como parámetro; un método *pop*, que extrae un elemento de la cola, eliminándolo de ella; y un método *reset*, que reinicia la cola entera, vaciándola por completo.

3.6. Crawling como MDP

Una vez definido el crawler básico, el siguiente paso es formalizar nuestra tarea de Crawling como un Proceso de Decisión de Markov, para poder implementar efectivamente los crawlers basados en RL. La interacción natural entre un crawler y un entorno web puede ser modelada a través de la interacción entre el entorno y un agente en RL. Un MDP es una tupla, (S, A, P, R), donde se incluyen el espacio de estados S, el espacio de acciones A, la función de transición P, y la función de recompensa R. En nuestro caso, definimos las páginas como estados s, y los enlaces presentes en ellas como acciones a. Cuando el crawler selecciona un enlace, se está produciendo una transición desde la última página que había obtenido (el estado actual) hacia la página apuntada por el enlace (estado siguiente). Por tanto, la función de transición T es la probabilidad de transitar de una página a otra, siguiendo un enlace. Además, como estamos diseñando un crawling enfocado, necesitamos escoger una función de recompensa que premie a aquellas páginas más relevantes al topic. La elección más directa es escoger una función de relevancia, que depende de la página y el topic $R = f(s, topic)$, siendo s el estado asociado a la página en cuestión.

Nosotros vamos a implementar dos crawlers: uno basado en Aproximación Lineal de Funciones (ALF), y otro basado en Deep Q-Learning (DQN). Para el primero, los estados y acciones van a ser modelados con las características definidas en el Apartado 3.3.1. Para el caso del crawler DQN, como veremos más adelante, tan solo necesitamos modelar los estados, así que solo usaremos la representación de características de las páginas. En ambos casos, usaremos como función de relevancia la similitud coseno entre el vector de la página y del topic. Para calcular esta recompensa no utilizamos los vectores de características, utilizamos los vectores W2V definidos en el Apartado 3.3.2.

Típicamente, en problemas de MDPs solemos encontrar un problema recurrente: el inmenso tamaño del espacio de estados y el de acciones. En nuestra tarea de crawling efectivamente contamos con una infinidad de páginas y enlaces, (en el orden de más de 300 mil páginas y más de 5 millones de enlaces), pero ya hemos planteado la solución al problema: el uso de vectores de características. Gracias a esto, en vez de contar con un estado para cada página, y una acción para cada enlace, estamos mapeando páginas similares a un mismo estado, y acciones similares a un mismo estado, por lo que reducimos el tamaño de ambos espacios y se vuelven manejables.

Una vez definido el crawling como un MDP de manera general, vamos a pasar a definir el crawler basado en Deep Q-Learning y el basado en Aproximación Lineal de Funciones por separado, comentando los problemas que puedan surgir en cada uno de ellos, y especificando cómo funcionan.

3.7. Crawler basado en RL: Deep Q-Learning

3.7.1. Definición de la tarea

Planteamos el primero de los crawlers basados en RL como un crawler enfocado que utiliza Deep Q-Learning para predecir la prioridad de los enlaces. En el Deep Q-Learning se utiliza una red neuronal profunda para estimar el valor de una función Q, que asigna un valor a cada par estado-acción, e informa al agente RL, en este caso, nuestro crawler, sobre la conveniencia de tomar una acción u otra según el estado en el que se encuentre. Para poder evitarnos los detalles técnicos de la implementación de la red neuronal profunda, y no tener que implementar desde cero el algoritmo de retropropagación de la red, utilizamos la librería Keras-RL [29], que ya incluye métodos para utilizar una red neuronal profunda como Q-Network en un problema de Deep Q-Learning. Con el uso de esta librería, la Q-Network tiene que estar estructurada de tal manera que reciba a la entrada el estado actual, y a la salida cuente con tantas neuronas como acciones posibles haya, ya que cada neurona indicará el valor de la función Q para la acción correspondiente, dado el estado recibido a la entrada.

El uso de Keras-RL es necesario, ya que el objetivo de este trabajo no es centrarse en los detalles de implementación de una red neuronal compleja, como es nuestra Q-Network. Sin embargo, la estructura de Q-Network que acabamos de comentar nos plantea un gran problema. Toda red neuronal tiene que ser inicializada antes de ser entrenada, definiendo su estructura: número de capas, número de neuronas por capa, etc. Como nuestra capa de salida tiene tantas neuronas como acciones vaya a tener disponibles el agente de RL, estamos decidiendo a priori, antes de comenzar el entrenamiento, cuántas acciones va a haber disponibles. Esto va en contra de la tarea del crawling, donde el programa parte solo con una semilla, y no sabe cuántos enlaces va a tener que añadir a la frontera. Sin embargo, la motivación de investigar la integración de este área con el algoritmo de Deep Q-Learning nos obliga a hacer una gran concesión: imponer un límite en el número de enlaces que el crawler puede añadir a la frontera.

Esta restricción puede esquivarse de manera natural de varias formas. Podemos decidir el número máximo de enlaces que el agente podrá añadir a la frontera. Por ejemplo: decidimos que nuestra frontera solo puede albergar 10.000 enlaces, y todos aquellos que se descubran una vez cubierto el cupo, son descartados directamente. Otra forma es conocer de antemano con cuántas páginas vamos a trabajar. Esto es imposible cuando accedemos directamente a la web, pero si utilizamos un conjunto de datos (un dump, por ejemplo), sí que podemos conocer cuántas páginas contiene. Esta opción va a ser nuestro caso.

Cabe destacar que esta limitación, como viene motivada por la estructura de la Q-Network, podría evitarse cambiando directamente esta misma estructura. Podríamos utilizar una Q-Network que reciba en la entrada el estado y la acción, (por ejemplo, representando también las acciones como vectores de características, e introduciendo en la entrada los vectores del estado y la acción concatenados), y a la salida tenga una sola neurona, cuyo valor esté asociada al valor Q de tomar esa acción, en ese estado. Así, no tendríamos que definir a priori cuantas acciones vamos a utilizar, simplemente tenemos que

decidir el tamaño de ambos vectores. Sin embargo, como ya hemos comentado, desligarnos de la librería nos impediría evitar el trabajo tan costoso de implementar el Deep Q-Learning desde cero. Por como está construida la librería, no es posible utilizar este modelo de Q-Network. Además, esta orientación no funcionaría con datos muy dispersos, donde haya combinaciones de estados y acciones que no aparezcan apenas, y por tanto la red no aprenda lo suficiente. Es decir, necesitaríamos unos datos con una granularidad muy alta, y puede que no sea posible conseguirlos en este contexto, por lo que esta aproximación podría no funcionar computacionalmente. Es por ello que dejamos esta idea como trabajo futuro, para poder ser investigada más adelante.

Concluyendo, hemos propuesto una solución para poder utilizar Keras-RL e integrar de manera sencilla el Deep Q-Learning con nuestra tarea de crawling web. En definitiva, vamos a realizar un crawling sobre un entorno controlado, utilizando un conjunto de datos que nos permita acotar de antemano el número de páginas con el que vamos a trabajar. Aunque perdamos la posibilidad de utilizar nuestro crawling en un entorno real, accediendo a la web sin limitaciones en el número de páginas a descubrir, esta implementación nos sirve a modo de simulación, para explorar la eficacia de su integración.

3.7.2. Funcionamiento del crawler

El crawler cuenta con varias piezas independientes que son todas necesarias para su correcto funcionamiento. La primera es el dump, que lo gestionamos como definimos en la Sección 3.2. La segunda es el vectorizador de páginas, que usamos para calcular los vectores W2V de cada documento, que sigue los principios que definimos en el Apartado 3.3.2. Utilizamos como frontera una URPQ (Sección 3.4), y como política una ϵ-greedy, que trabaja directamente con esta frontera. Como Q-Network, utilizamos un modelo de Keras, donde a la entrada hay tantas neuronas como número de características tengamos para los estados, introducimos tantas capas ocultas como queramos, y a la salida tiene que haber tantas neuronas de salida como páginas haya en el dump que utilicemos. Además, esta red utiliza las modificaciones del algoritmo Dueling y Double (Apartados 2.6.3 y 2.6.2). Extraemos las características que definimos en el Apartado 3.3.1 con dos elementos: un "featurizer" para las páginas, y otro para los enlaces. Con todos estos elementos creamos un entorno de OpenAI Gym [35], una librería que estandariza la implementación de éstos dentro del paradigma del RL. Nosotros definimos el agente RL como un agente Keras-RL, que se comunica con el entorno automáticamente. Todo el peso de nuestro desarrollo está en el lado del entorno, que es donde podemos personalizar qué operaciones se realizan entre paso y paso. En concreto, uno de los métodos de la clase general y abstracta "Env" es el método *step*, que recibe como entrada una acción, y debe devolver el siguiente estado, la recompensa asociada al mismo, la comprobación de la condición de parada, y un diccionario con parámetros de depuración. Además del método step, nuestro entorno debe implementar un método reset, que no recibe parámetros, y debe realizar las operaciones necesarias para reiniciar el entorno a un estado inicial.

Por simplicidad a la hora de presentar el algoritmo, vamos a definir las operaciones realizadas como si todo el proceso de crawling se realizara en una sola función, presentada en el Algoritmo 3, aunque en realidad este código está distribuido entre el constructor del

entorno y diferentes métodos.

En el algoritmo se han ignorado ciertos detalles incluidos en la implementación real para facilitar su lectura en pseudo-código. Estos detalles incluyen la gestión del descubrimiento de ancestros para cada página, la extracción de las características, etc.

Algorithm 3: Crawler Enfocado basado en Deep Q-Learning

Data: Dump D, Vectorizer V, Q-network, topic t, seed $Seed$, limit L, W2V first threshold δ_1, W2V second threshold δ_2
Result: Storaged pages

```
F ← ∅ # Initialize frontier;
Initialize number of crawled pages, N ← 0;
Initialize number of relevant pages discovered, N_R ← 0;
Initialize W2V topic vector V_T using V;
foreach page ∈ Seed do
    S ← Fetch page and extract features from it;
    outlinks ← Extract links from page;
    foreach link ∈ outlinks do
        A ← Extract features from link;
        Add pair (S, A) to frontier with initial Q-value;
while N ≤ L do
    (S, A) ← Get state-action pair from policy ϵ-greedy;
    if (S, A) ∈ visited_links then
        continue
    Add (S, A) to visited_links;
    S' ← Fetch and extract features of page pointed by A;
    L' ← Extract links of S', ignoring those already visited;
    A' ← Extract features of each link ∈ L';
    tf ← number of ocurrences of topic in S';
    if tf > 0 then
        r ← 30;
        N_R ← N_R + 1 # Relevant page discovered
    else
        V_S' ← Extract W2V vector of S' using V;
        sim ← Compute cosine sim between V_S' and V_T;
        if sim > δ_1 then
            r ← 20;
        else if sim > δ_2 then
            r ← 10;
        else
            r ← −1;
    Q ← current Q-values extracted by forward propagating S';
    foreach (S', .) pair ∈ L' do
        Update (S', A') pair in F with value Q[A'];
    Apply back-propagation step in the network, with S' and r;
    Save page in storage;
    N ← N + 1;
```

3.8. Crawler basado en RL: Aproximación Lineal de Funciones

La Aproximación Lineal de Funciones es un método de resolución de MDPs, que mediante una aproximación lineal estima iterativamente el valor de la función Q. Una de las pocas propuestas existentes en la literatura sobre crawlers web que usen técnicas de RL se basa en este método [2], y es precisamente el artículo que utilizamos como base para este crawler. De él tomamos la definición de las características de los estados y las acciones, y seguimos sus indicaciones para implementar este crawler desde cero, sin usar ninguna librería (en este crawler no hacemos uso de la librería Keras-RL).

Este Trabajo Fin de Máster se hace dentro de un programa de posgrado orientado a la Ingeniería Informática, no a la Investigación. Es por ello que, aunque esta propuesta no es original, desde el punto de vista de los objetivos del trabajo, tiene sentido elegir implementar una propuesta ya realizada en la literatura, pues buscamos adquirir conocimientos sobre resolución de problemas de Ingeniería Informática, no sobre investigaciones de enfoques novedosos.

Funcionamiento del crawler

Al igual que con el crawler anterior, el funcionamiento general del crawler depende de diversas piezas que pueden funcionar por separado. En este caso, ya no usamos una red neuronal (Q-Network) como aproximador funcional, ahora realizamos una aproximación lineal de funciones. Ya definimos en el Apartado 2.5 cómo funciona el algoritmo, así que basta con saber que la pieza que sustituye ahora a la red neuronal es un aproximador LFA, que utiliza la librería numpy para manejar las operaciones con vectores. Las demás piezas pueden verse de manera análoga al crawler anterior: usamos un dump a modo de entorno web, un vectorizador para vectorizar las páginas con W2V, una UPRQ (Apartado 3.4) como frontera, una política ϵ-greedy que trabaja con esa frontera, y dos "featurizer" que se encargan de extraer características de las páginas y los enlaces. En el Algoritmo 4 se detalla el funcionamiento del crawler.

Cabe destacar las dos formas en las que podemos actualizar las prioridades de las fronteras. Según avanza el crawling, nuestro agente aprende mejores aproximaciones de la función Q, que la utilizamos como función de prioridad para los enlaces que añadimos en las fronteras. Sin embargo, debido a este desfase temporal, estas prioridades se van quedando obsoletas según avanza el crawling. En [2] los autores proponen 3 formas de actualizar la frontera, pero nosotros nos hemos quedado con las dos principales: el método asíncrono y el síncrono.

Con el método asíncrono, actualizamos las prioridades de los enlaces de la última página que hemos visitado. Por otro lado, con el método síncrono, actualizamos las prioridades de todos los enlaces de la frontera. El primer método, teóricamente, sacrifica exactitud en el valor de las prioridades a cambio de una carga computacional mucho más ligera. Por el contrario, el segundo método asume una carga mucho mayor a cambio de

ser más estrictos con el cálculo de las prioridades. Aunque en el Algoritmo 4 se incluyen ambos métodos, en la práctica, en cada ejecución solo se aplica uno de ellos en todo el proceso de crawling.

Por último, también debemos mencionar que la regla de actualización de los pesos utilizada la hemos extraído del desarrollo presentado en la Sección 2.5. El último término de la regla es el gradiente de $\hat{q}(s, a, w)$, que como vimos en esa Sección, no es más que el vector de características del estado y la acción.

Algorithm 4: Crawler Enfocado basado en Aproximación Lineal de Funciones

Data: Dump D, Vectorizer V, LFA approximator LFA, topic t, seed $Seed$, limit L, W2V first threshold δ_1, W2V second threshold δ_2
Result: Storaged pages

F $\leftarrow \emptyset$ # Initialize frontier;
Initialize number of crawled pages, $N \leftarrow 0$;
Initialize number of relevant pages discovered, $N_R \leftarrow 0$;
Initialize W2V topic vector V_T using V;
foreach *page* $\in Seed$ **do**
    $S \leftarrow$ Fetch page and extract features from it;
    outlinks $\leftarrow$ Extract links from page;
    foreach *link* $\in$ *outlinks* **do**
        $A \leftarrow$ Extract features from link;
        Add pair (S, A) to frontier with initial Q-value;
while $N \leq L$ **do**
    $(S, A) \leftarrow$ Get state-action pair from policy ϵ-greedy;
    if $(S, A) \in$ *visited_links* **then**
        continue
    Add (S, A) to visited_links;
    $S' \leftarrow$ Fetch and extract features of page pointed by A;
    $L' \leftarrow$ Extract links and its features of S', ignoring those already visited;
    $tf \leftarrow$ number of ocurrences of topic in S';
    if $tf > 0$ **then**
        $r \leftarrow 30$;
        $N_R \leftarrow N_R + 1$ # Relevant page discovered
    else
        $V_{S'} \leftarrow$ Extract W2V vector of S' using V;
        $sim \leftarrow$ Compute cosine sim between $V_{S'}$ and V_T;
        if $sim > \delta_1$ **then**
            $r \leftarrow 30$;
        else if $sim > \delta_2$ **then**
            $r \leftarrow 20$;
        else
            $r \leftarrow -1$;
    $A' \leftarrow$ Select next action from L' with ϵ-greedy and S', and extract its features;
    Update LFA weights with the update rule:
    $w \leftarrow w + \alpha[r + \gamma\hat{q}(s', a', w) - \hat{q}(s, a, w)]\nabla\hat{q}(s, a, w)$;
    # Synchronous method
    foreach *(., .) pair* $\in F$ **do**
        Update (S', A') pair in F with value $Q[S', A']$ extracted with LFA;
    # Asynchronous method
    foreach *(S', .) pair* $\in L'$ **do**
        Update (S', A') pair in F with value $Q[S', A']$ extracted with LFA;
    $N \leftarrow N + 1$;

4

Resultados

En esta Sección se describirán en detalle los experimentos realizados para probar el funcionamiento de los tres crawlers implementados y descritos en la sección anterior. Estos experimentos tienen el objetivo de evaluar los sistemas, enfrentando a los crawlers entre sí, realizando comparativas entre las opciones basadas en RL y las simples.

Primero, vamos a detallar la estructura de los conjuntos de datos utilizados, (por ejemplo, la distribución de las páginas en nuestro dump de Wikipedia), para poder entender mejor los resultados que obtendremos más adelante. Después, vamos a definir una tarea experimental clara y concisa. Tras esto, especificaremos la configuración de los parámetros utilizados para ejecutar los algoritmos. Por último, mostraremos los resultados obtenidos tras poner en marcha la tarea experimental definida.

4.1. Detalles de los conjuntos de datos

Para poder comprender mejor los resultados que vamos a exponer en los siguientes apartados y dar una perspectiva más rica a los experimentos, es importante que analicemos el contenido de nuestras diferentes versiones de dumps de Wikipedia, y la distribución de las páginas en el mismo. Nosotros usamos este dump a modo de entorno web. Es decir, el crawler interactúa con él como podría interactuar con la propia web. En la Tabla 4.1 se muestra la comparativa del número de páginas, enlaces y el promedio de enlaces por página en 3 versiones del dump de "simple-english". La primera versión, que nombramos seng-dump, se corresponde a un dump obtenido tras preprocesar solo el formato de los datos. En la siguiente versión (norm-seng-dump) se realiza un filtrado y preprocesado también en el contenido del dump: se eliminan autoenlaces, se dejan fuera aquellos enlaces que tienen origen o destino en páginas que no están en el dump, etc. La última versión

(norm-seng-dump(0.1)) es una partición aleatoria del 10 % del dump anterior. Es decir, tomamos el 10 % de las páginas aleatoriamente, y nos quedamos con los enlaces que parten desde ellas, asegurándonos de nuevo de que todos los enlaces tienen como origen y destino páginas que pertenecen a este nuevo dump. Un dato muy interesante a tener en cuenta para comprender mejor los resultados de los experimentos es el bajo promedio de enlaces por página que tiene nuestro dump reducido. Esto afectará en gran medida a la hora de descubrir páginas relevantes, pues hay muchos caminos hacia ellas que desaparecen en esta colección.

Versión del dump	**Nº de páginas**	**Nº de enlaces**	**Enlaces por página (media)**
seng-dump	323198	+10 millones	31,12 enlaces/página
norm-seng-dump	323198	+5 millones	21,41 enlaces/página
norm-seng-dump(0.1)	32325	$\approx$ 63000	4 enlaces/página

Tabla 4.1: Comparativa del número de páginas y enlaces en las diferentes versiones del dump de Wikipedia "simple-english"

Una vez que hemos dimensionado los dumps, otro aspecto importante que podemos visualizar es la distribución de enlaces por páginas, para poder identificar si están distribuidos de manera uniforme, es decir, de media, todas las páginas tienen más o menos el mismo número de enlaces, o hay algún sesgo. Visualizamos la distribución de la versión norm-seng-dump en la Figura 4.1 y la versión reducida norm-seng-dump(0.1) en la Figura 4.2. Como podemos ver, la distribución de enlaces sí que está sesgada: hay un sesgo por popularidad. Se puede ver que hay un pequeño número de páginas con muchos enlaces (tanto de entrada como de salida, representados en las gráficas con los títulos "inlinks" y "outlinks"), y una gran cola de páginas con muy pocos enlaces. Esta es una distribución muy estudiada en el campo de la Recuperación de Información, la distribución **power-law**. También cabe destacar que hemos incluido solo las 1000 páginas con más enlaces en las curvas para poder visualizar bien las curvas, ya que la cola de páginas con pocos enlaces comprende casi a la totalidad de ellas.

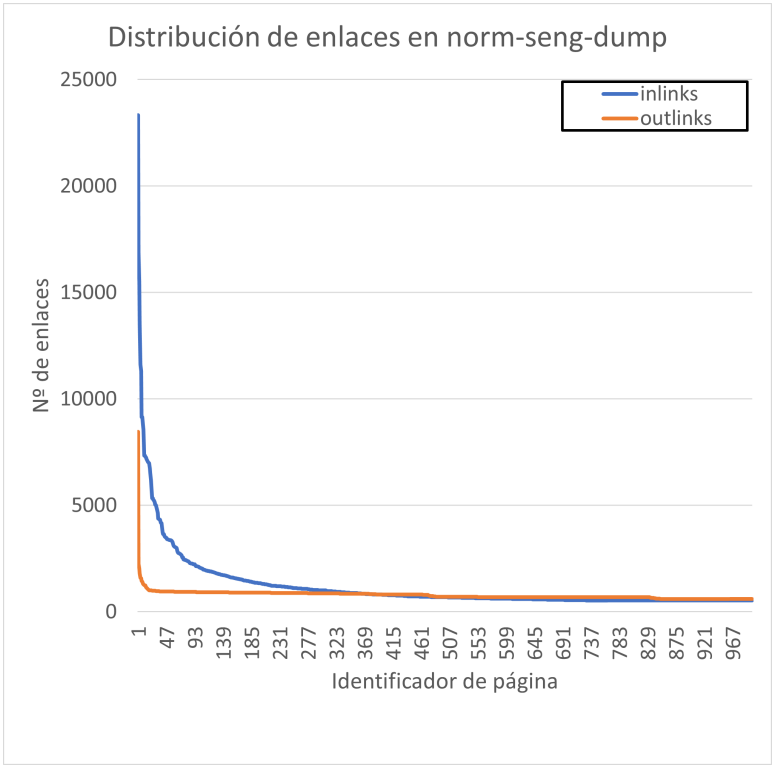


Figura 4.1: Distribución de enlaces en el dump normalizado (norm-seng-dump).

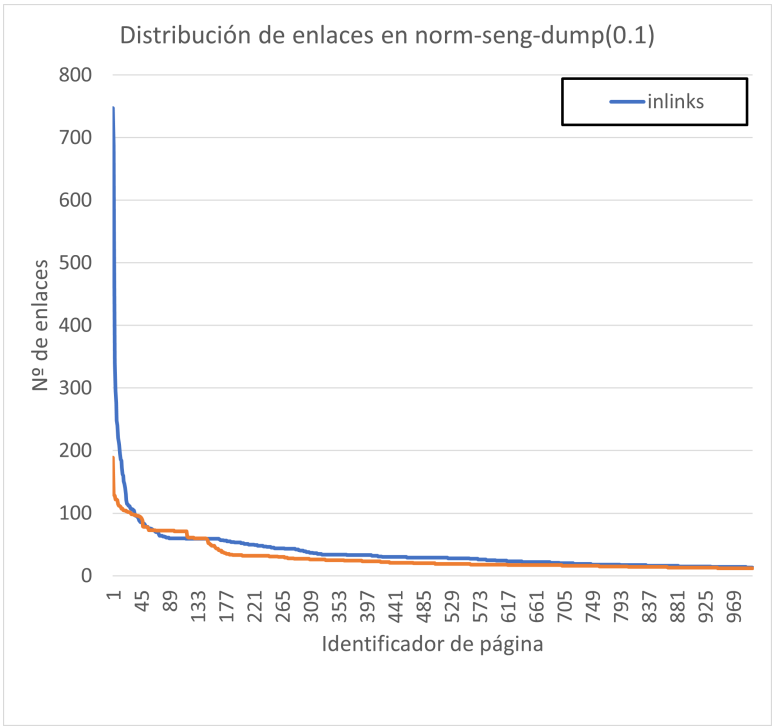


Figura 4.2: Distribución de enlaces en el dump normalizado y reducido (norm-seng-dump(0.1)).

4.2. Definición de la tarea

Para poder realizar un estudio completo sobre el rendimiento de nuestros crawlers explicados en la sección anterior, vamos a definir una tarea básica que aplicaremos en los experimentos. Esta tarea consiste en simular de manera offline la interacción que ocurre entre un crawler y un entorno web. Para ello, nuestros crawlers interactúan con el dump de wikipedia, extrayendo páginas de ellos y analizando su contenido para extraer nuevos enlaces y calcular las prioridades que asignarán a cada enlace en la frontera. Vamos a realizar dos experimentos diferenciados. El primero, utilizando la versión reducida de nuestro dump normalizado, vamos a comparar el rendimiento del crawler enfocado basado en Deep Q-Learning (**Crawler DQN**) con uno básico que no utiliza ningún aprendizaje basado en RL. Utilizamos la versión reducida del conjunto de datos ya que, por las restricciones derivadas del uso de la librería de Keras-RL, se vuelve inviable ejecutar el crawler si nuestro dump es demasiado grande. Después, utilizando el dump normalizado total, compararemos el crawler enfocado basado en aproximación lineal de funciones (**Crawler ALF**) con la versión básica. Ya veremos cuando presentemos estos resultados que la comparativa entre el crawler ALF y el DQN no será necesaria, debido a la gran diferencia de rendimiento que hay entre ellos.

Nuestro crawler parte de una semilla que contiene a páginas representativas del topic escogido. Por tanto, nuestra configuración no es representativa de un arranque en frío, donde el crawler podría partir de una semilla aleatoria, y mediríamos el tiempo medio que tarda cada crawler en comenzar a descubrir relevancia. Aunque pueda parecer que este arranque en frío no es representativo de una situación real, donde el diseñador del sistema define un topic y un entorno web sobre el que realizar el crawling, pero a la vez no conoce una página que sirva de punto de partida, sí que tendría sentido a la hora de trabajar con un entorno web que es desconocido para el experto. Un ejemplo directo que podría adaptarse a esta situación es la propia Deep Web, donde el crawler podría no contar con mecanismos para obtener una página representativa para cualquier topic. Sin embargo, este no es nuestro caso, pues estamos trabajando con artículos de Wikipedia, una enciclopedia online en la que se engloban prácticamente todos los conceptos universales, y por tanto, nos basta con buscar el artículo dedicado al propio concepto que conforma nuestro topic.

Además, en nuestros experimentos vamos a trabajar con un concepto clave: las páginas relevantes. El objetivo de un crawler enfocado es encontrar páginas relevantes a un tema, que nosotros llamamos topic. Hay muchas formas de definir qué es una página relevante, pero nosotros utilizamos una definición muy simple: una página es relevante a un topic dado, si la página contiene el término que define ese mismo topic. Es decir, si el topic es "minería", toda página que contenga ese término será relevante en nuestros experimentos. Nosotros vamos a medir la relevancia acumulada a lo largo del proceso de crawling. Ejecutaremos el crawler durante un número de pasos establecidos, y mediremos cuánta relevancia y cuándo se descubre. Además, en esa relevancia acumulada no incluimos páginas repetidas, ya que lo natural es que si un crawler ha obtenido ya una página, no vuelva a descargarla de la Web ni a procesarla de nuevo más adelante.

4.3. Configuración y parámetros

Debido a la complejidad del sistema, hay multitud de hiperparámetros que pueden modificar el rendimiento del mismo. Para escoger sus valores, hemos tenido en cuenta la configuración que proponen los autores en [2] para el crawler LFA, realizando ciertas modificaciones donde sea necesario.

Para el sistema de vectorización W2V, utilizamos unos vectores W2V pre-entrenados para cada palabra, de 100 dimensiones cada uno. Para calcular los valores IDF de cada una, hemos utilizado el dump reducido (norm-seng-dump(0.1)) para pre-calcularlas. Recorremos todas las páginas del dump contando el número de documentos en los que aparece cada palabra del vocabulario, y almacenamos sus valores para ser cargados directamente a la hora de ejecutar los crawlers.

Para la Q-Network del crawler DQN, tenemos que escoger una estructura de red. Si utilizamos el dump total (norm-seng-dump), necesitamos utilizar una capa de salida de más de 300 mil neuronas, lo cual hace que el número de pesos crezca desmesuradamente, y vuelve inviable entrenar la red. Es por ello que este crawler vamos a ejecutarlo con el dump reducido, teniendo así una capa de salida de alrededor de 30 mil neuronas. Incluimos 2 capas ocultas, de 128 y 256 neuronas, y regularización con una capa de dropout entre la última capa oculta y la de salida.

Para el aproximador lineal, utilizamos un factor de aprendizaje $\alpha = 0{,}001$, ya que necesitamos que sea muy pequeño para garantizar la convergencia [1].

Con respecto a la política, utilizamos una ϵ-greedy, y tomamos un $\epsilon = 0{,}1$ en test, y un $\epsilon = 0{,}3$ en entrenamiento (en el crawler DQN hay una fase de entrenamiento y una de test, pero en el crawler LFA solo hay una ejecución, en este caso, $\epsilon = 0{,}1$).

En cuanto a la extracción de características de los enlaces y las páginas, utilizamos un factor de suavizado $\beta = 0{,}4$ para el método *exponential smoothing* del cambio de relevancia con respecto al topic. Para discretizar este valor, utilizamos 2 parámetros, σ_1 y σ_2, que toman los valores 0,1 y 0,3, respectivamente. Además, para tratar los enlaces, utilizamos el texto alrededor de ellos, que comprende 300 caracteres (los 150 caracteres que preceden al enlace, y los 150 caracteres que aparecen después). El máximo valor que puede tomar la característica de la distancia con el último ancestro relevante es 9, para evitar distancias infinitas. Como topics y categorías, utilizamos los propuestos en [2]:

- Topic “fiction”, con las categorías “literature” y “arts”.
- Topic “cancer”, con las categorías “disease”, “medicine”, “oncology”, “health”
- Topic “olympics”, con la categoría “sports”.
- Topic “cameras”, con las categorías “photography” y “movies’ y “arts”.
- Topic “geology”, con las categorías “earth” y “science”.
- Topic “poetry”, con las categorías “literature” y “arts”.

Por último, necesitamos escoger un factor de descuento γ para ambos crawlers. Para ello, hacemos un barrido de este valor utilizando el crawler LFA y el dump reducido.

Barrido de gamma

Para el agente, necesitamos escoger un factor de descuento γ. Recordemos que este parámetro regula la importancia de las recompensas futuras, siendo estas más importantes cuanto más se aproxime a 1. Aunque podríamos escoger un valor en base a la intuición, es más interesante realizar un barrido sobre diferentes valores y observar cuál obtiene mejores resultados. Para no eternizar la ejecución, utilizamos la versión reducida del dump, y ejecutamos el crawler LFA, ya que es el más rápido. El barrido se realiza sobre $\gamma = [0{,}1, 0{,}2, 0{,}3, 0{,}4, 0{,}5, 0{,}6, 0{,}7, 0{,}8, 0{,}9]$, para el topic "fiction", y sus categorías correspondientes. Ejecutamos el crawler durante 10.000 pasos.

En la Figura 4.3 se muestran los resultados de este barrido. En la propuesta original, [2], proponían el uso de $\gamma = 0{,}9$. Sin embargo, en nuestros resultados, vemos que ese valor es el que menor relevancia acumulada alcanza. Un factor de descuento tan alto hace que el crawler no de apenas importancia a las recompensas actuales, y pondere mucho más las recompensas futuras, aunque nuestra propuesta tiene suficientes piezas de implementación propia y diferenciadas del artículo original que pueden justificar esta disonancia en los resultados. Como en la gráfica no se aprecia del todo bien donde acaba cada curva, se incluye en la Tabla 4.2 los valores finales de cada curva.

En la tabla y en la gráfica se observa un fenómeno interesante: la gran diferencia de rendimiento entre valores altos de γ (desde 0,7 hasta 0,9, ambos incluidos), y valores bajos e intermedios (desde 0,1 hasta 0,6, ambos incluidos). Una posible explicación es que los valores altos priorizan demasiado la exploración, desatendiendo la explotación necesaria para obtener buenos resultados. Sin embargo, dentro de estos valores bajos e intermedios, no hay diferencias evidentes, ya que todos rondan una relevancia acumulada de 1600 páginas, por lo que las diferencias podrían deberse perfectamente al factor aleatorio que introducimos con el uso de la política ϵ-greedy. Por tanto, podemos concluir que dentro de estos valores no hay diferencias apreciables de rendimiento, y en principio, cualquier γ que esté en el rango $[0{,}1, 0{,}6]$ es una buena elección. Nosotros hemos usado un $\gamma = 0{,}3$, por ser un valor intermedio en el rango anterior.

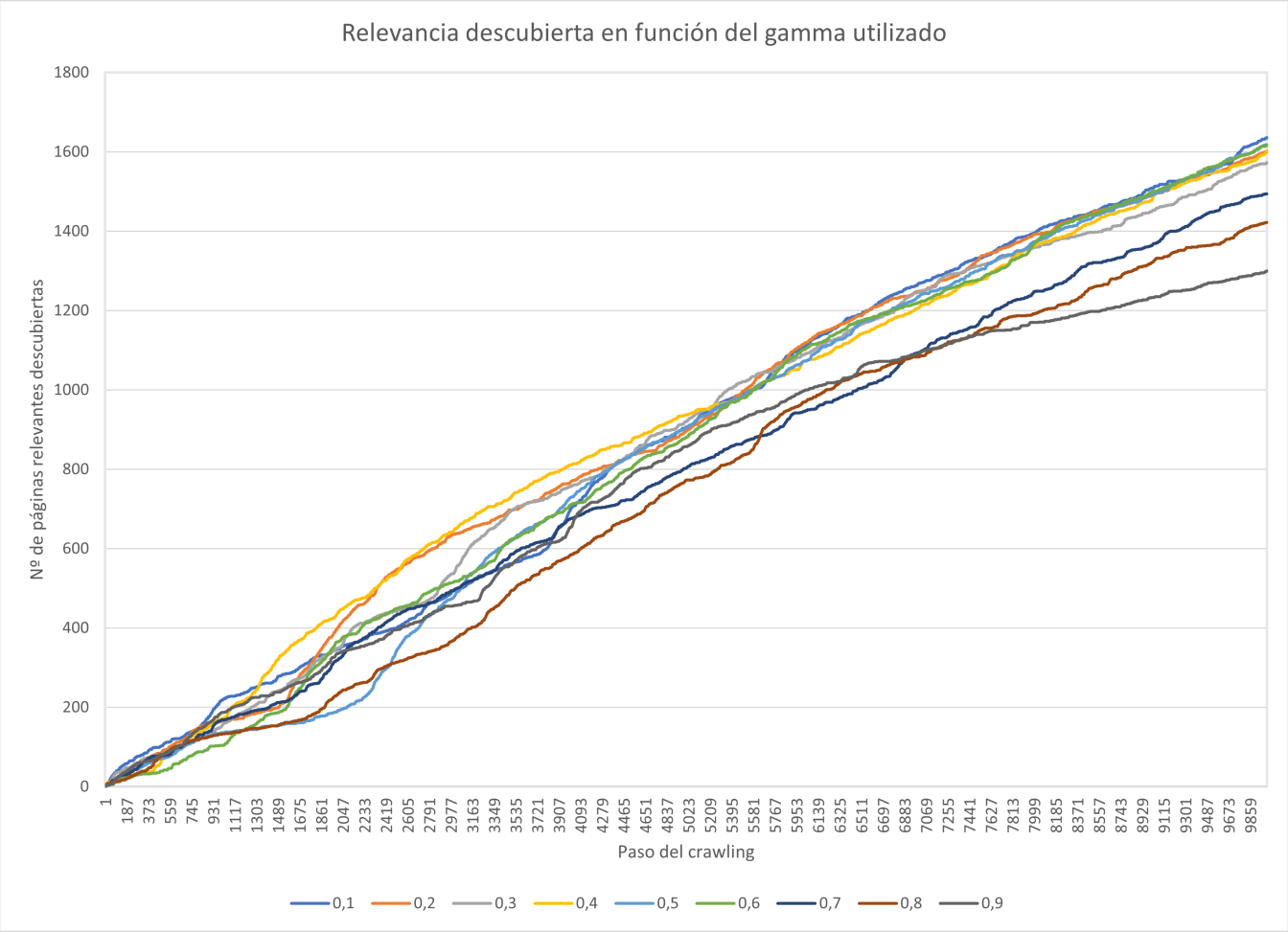


Figura 4.3: Relevancia acumulada descubierta para cada valor de γ.

Gamma	Relevancia acumulada final
0.1	1636
0.2	1600
0.3	1573
0.4	1603
0.5	1618
0.6	1617
0.7	1494
0.8	1422
0.9	1300

Tabla 4.2: Relevancia acumulada final para cada valor de γ.

4.4. Análisis de resultados: crawler DQN

Una vez definida la configuración de nuestros crawlers, pasamos a evaluar su rendimiento. Para ello, necesitamos definir una métrica que evalúe nuestros sistemas. Nosotros estamos enfocando nuestros experimentos hacia el análisis del descubrimiento de relevancia. Por ello, vamos a usar el número de objetos relevantes descubiertos, a lo largo del proceso de crawling. Estos objetos no pueden repetirse, es decir, solo cuentan aquellos objetos relevantes que no habían sido descubiertos ya. Nos referiremos a este cálculo como **relevancia acumulada** a partir de ahora. Podríamos utilizar también como métrica el Recall, que calcula el porcentaje de objetos relevantes descubiertos:

$$Recall = \frac{\text{nº de relevantes descubiertos}}{\text{nº de relevantes totales}}$$

Sin embargo, como lo que vamos a hacer es realizar una comparativa entre crawlers, no es necesario complicar demasiado la métrica, pues lo que nos interesa es ver si nuestro crawler supera al básico, y por cuánto.

En cuanto a los resultados obtenidos por el crawler DQN, podemos visualizar en la Figura 4.4 los resultados obtenidos tanto en la fase de entrenamiento como de test. Hemos entrenado la red durante cinco mil pasos, y luego hemos ejecutado la fase de test durante otros cinco mil pasos. En la gráfica se incluye también los resultados del crawler básico. Lo primero que podemos comentar a la vista de los resultados es directo: el crawler DQN no alcanza el mínimo rendimiento para considerarlo una alternativa viable. Nuestra propuesta de red neuronal no aprende lo suficiente como para alcanzar al crawler enfocado básico, por lo que no tiene sentido utilizar el crawler DQN, ya que es una propuesta mucho más compleja, y mucho más lenta en términos de tiempos de ejecución. El crawler básico no realiza ningún tipo de aprendizaje especial, únicamente calcula la similitud coseno entre dos vectores Word2Vec: el del topic, y el del texto construido a partir de la URL del enlace y el texto que lo rodea, y aún así, casi cuadriplica los resultados del crawler DQN.

El motivo de este bajo rendimiento puede ser múltiple. En nuestra implementación hemos usado la librería Keras-RL [29] para implementar el agente de Deep Q-Learning. Sin embargo, con el uso de esta librería nos vemos obligados a usar un aproximador (la red neuronal) que tenga tantas neuronas de salida como acciones diferentes, ya que Keras-RL asume que cada neurona de salida está asociada siempre con la misma acción. Esto hace que, en problemas donde el espacio de acciones es enorme, como es nuestro caso, la red neuronal se haga demasiado grande, complicando el aprendizaje si queremos mantener un número de pesos que no desborde la capacidad de nuestro equipo. En nuestro caso, para poder entrenar la red en un equipo de 8 GB de memoria RAM, nos vimos obligados a utilizar el dump reducido, (que tiene alrededor de 32.000 enlaces). Sin embargo, este dump sigue teniendo demasiadas acciones posibles, y nos vemos obligados a utilizar unas capas ocultas de pocas neuronas, por lo que al final son unas pocas neuronas las que tienen que tomar una decisión demasiada compleja (nuestra red tenía 2 capas ocultas, de 128 y 256 neuronas respectivamente, como se indicó en el Apartado 4.3). Además, aún utilizando un conjunto de datos pequeño, la red tarda demasiado tiempo en aprender,

por lo que nos dificulta muchísimo la tarea de encontrar una estructura de red adecuada (para dimensionar la diferencia, la red tarda 20 horas en entrenar con el dump original, y entre 4 y 5 horas con el dump reducido), ya que hay demasiados factores que podemos variar, resultando en una combinatoria de configuraciones demasiado grande (podemos variar la propia estructura de la red, el uso de dropouts, el valor de cada capa dropout, regularización L2 con diferentes parámetros, etc.).

Una posible solución para resolver este problema de raíz sería cambiar por completo el funcionamiento de la red. Nuestra red recibe el estado a la entrada, y a la salida da tantos valores como acciones disponibles (sus valores Q). Si por el contrario utilizásemos una red que toma el estado y la acción a la entrada, y un solo valor a la salida, podríamos reducir drásticamente la complejidad tanto estructural como computacional. Sin embargo, no es sencillo implementar esta opción, pues la librería Keras-RL no cuenta con una implementación en este sentido, por lo que habría que implementar la red Q-Network desde cero, utilizando las librerías de Keras y Tensorflow. Como el objetivo del TFM no era entrar a tan bajo nivel con las redes neuronales, sino realizar una implementación más general en el ámbito de los crawlers, esta opción se ha dejado como una posible línea de trabajo futuro.

Por tanto, como nuestro crawler DQN no ha sido capaz de batir al crawler básico, vamos a dejarlo fuera de la comparativa más general que vamos a analizar a continuación, con el uso de diferentes topics.

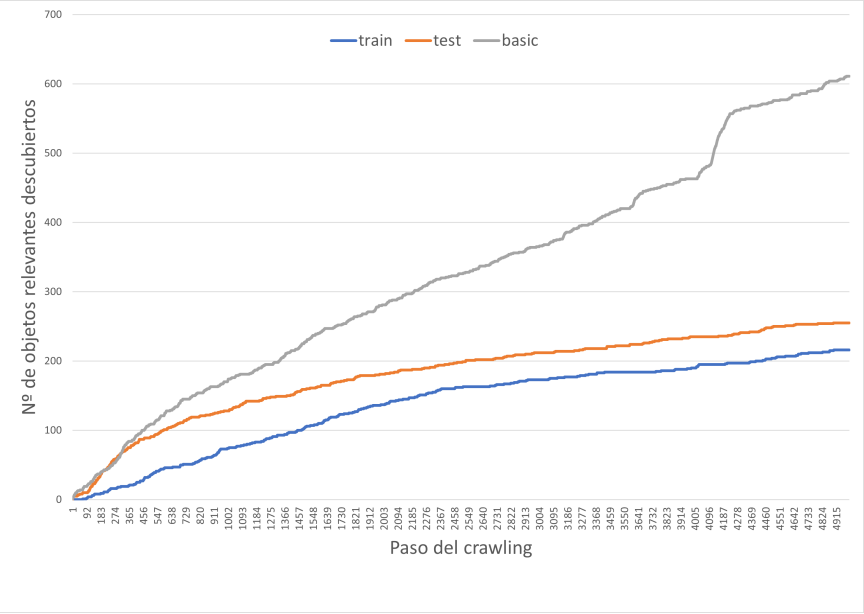


Figura 4.4: Comparativa de relevancia acumulada entre el crawler DQN y el básico para el topic “geology”.

4.5. Análisis de resultados: crawler LFA

Una vez analizados los resultados del crawler DQN y haber concluido que no hemos alcanzado el rendimiento mínimo del crawler básico, vamos a analizar por separado el crawler LFA. En este crawler, distinguimos dos modos de ejecución: uno en el que que usamos una actualización asíncrona de la frontera, y otra con la actualización síncrona (Apartado 3.8). Primero, vamos a analizar brevemente el coste computacional de cada método, en términos de tiempos de ejecución. El motivo para realizar este análisis es simple: justificar el uso de un dump completo o uno reducido para cada caso, y contextualizar la configuración de cada experimento realizado. Después, vamos a mostrar los resultados de cada experimento por separado.

4.5.1. Análisis de rendimiento: Asíncrono vs Síncrono

El método asíncrono es el método más rápido de todos. En cada paso, tan solo actualizamos los enlaces de la frontera que parten de la última página obtenida por el crawler. Por tanto, en cada paso tan solo hay que actualizar pocos enlaces, y el número de actualizaciones por paso no crece con el tiempo, manteniéndose constante alrededor de la media del número de enlaces por página. Estas medias pueden consultarse en la Tabla 4.1. Además, realizamos pocas actualizaciones, pero cada actualización a su vez es una operación muy ligera en términos computacionales. Con el método de ALF, calcular el valor Q de un estado-acción supone realizar un producto escalar de 2 vectores, el de características del estado y la acción, y el de pesos. Ambos tienen que tener las mismas dimensiones, y en nuestro caso, por el diseño de características que hemos utilizado (Apartado 3.3.1), estamos utilizando dos vectores de 23 dimensiones. Por tanto, este método podemos ejecutarlo sin problemas tanto en el dump completo de Wikipedia, donde diez mil pasos tardan alrededor de 4 minutos en completarse, como en el dump reducido.

Por otro lado, el modo síncrono es una alternativa mucho más costosa computacionalmente. En cada paso tenemos que actualizar las prioridades de todos los enlaces en la frontera. Esto hace que la ejecución sea muy lenta, ya que aunque cada actualización es muy rápida como acabamos de mencionar, ahora tenemos que realizar un gran número de actualizaciones por paso. De media en cada paso vamos a añadir a la frontera el número promedio de enlaces. Esto hace que el número de actualizaciones crezca exponencialmente con el número de pasos. Al principio habrá que procesar unos cientos de enlaces por paso, pero cuando ejecutamos miles de pasos, en la frontera llegarán a haber decenas e incluso cientos de miles de enlaces (el promedio de enlaces por página en el dump completo está alrededor de los 20 enlaces por página). Es por ello que realizar experimentos con el dump completo se vuelve inviable si utilizamos el método síncrono. En este caso, vamos a utilizar por tanto el dump reducido, para poder realizar los experimentos en un tiempo prudencial.

Por tanto, vamos a estructurar los experimentos de la siguiente manera: vamos a ejecutar el crawler LFA asíncrono sobre el dump completo de Wikipedia (norm-seng-dump), y lo vamos a comparar con el crawler básico. Después, vamos a ejecutar los 3

crawlers en el dump reducido de Wikipedia: LFA asíncrono, LFA síncrono y el básico.

4.5.2. Comparativa de resultados: dump completo de Wikipedia

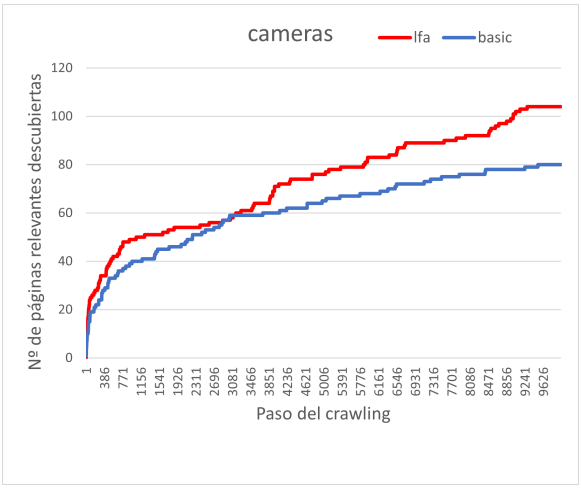
El primero de los experimentos que vamos a analizar es el de la comparativa entre el crawler LFA asíncrono y el básico sobre el dump completo de Wikipedia. En la Figura 4.5 se muestran los resultados de ambos, mostrando una gráfica diferente para cada topic. Para ambos, ejecutamos el crawler durante diez mil pasos, y llevamos la cuenta de las páginas relevantes que descubre cada uno a lo largo del proceso. En este caso, hemos utilizado los topics de "cameras", "cancer", "fiction", "geology" y "poetry". Hemos dejado fuera el topic "olympics" ya que, por la propia estructura de enlaces del dump, a los pocos pasos el crawler se queda sin enlaces en la frontera, y no nos sirve para visualizar una curva de relevancia descubierta durante diez mil pasos. A la vista de los resultados, podemos empezar a extraer conclusiones. La primera y más importante, es que en todos los casos el crawler LFA ha obtenido un mejor rendimiento que el crawler básico. En la Tabla 4.3 podemos observar el porcentaje de mejora que se ha obtenido en cada caso. La mejora mínima es de un 20 % (cancer), y la máxima de un 87 % (fiction). Estos resultados demuestran que podemos aplicar métodos de Aprendizaje por Refuerzo para mejorar la toma de decisiones del crawler enfocado, y extraer una función de prioridad que asigne mayores prioridades a los enlaces que tienen más probabilidad de llevar a páginas relevantes, sin haber obtenido la página todavía. Además, este método de RL tiene una ventaja muy grande con respecto al resto de métodos que hemos desarrollado en este trabajo: es computacionalmente muy ligero. El tiempo de ejecución promedio de diez mil pasos está alrededor de los 4 minutos. A su vez, el tiempo medio del crawler básico está alrededor de los 2 minutos y medio. Por tanto, el crawler LFA asíncrono se muestra como una gran alternativa a un crawler enfocado simple, mejorando el ratio de obtención de páginas relevantes, sin tener que incurrir en un coste computacional muy alto para ello.

Topic utilizado	***Relevantes (Básico)***	**Relevantes (LFA)**	**Porcentaje de mejoría**
Cameras	80	104	30 %
Cancer	538	648	20 %
Fiction	850	1590	87 %
Geology	597	823	38 %
Poetry	250	409	64 %

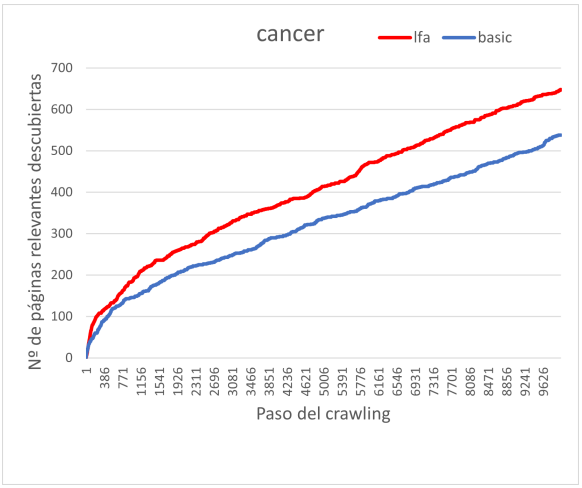
Tabla 4.3: Comparativa de número de páginas relevantes descubiertas entre el crawler LFA en su versión asíncrona y el crawler básico para todos los topics.

4.5.3. Comparativa de resultados: dump reducido de Wikipedia

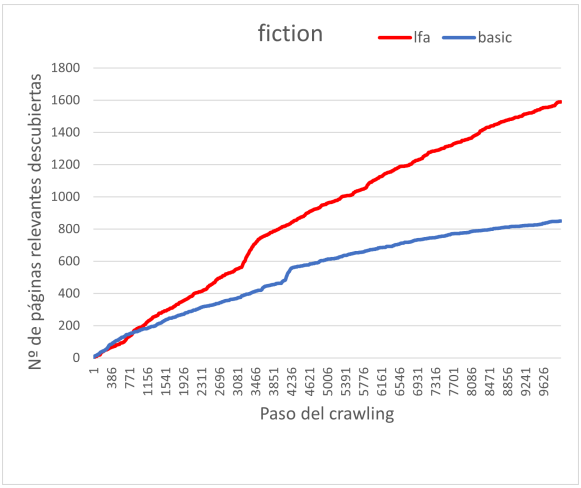
Como hemos comentado, el uso del modo síncrono del crawler LFA requiere una gran carga computacional, y por tanto, necesitamos ejecutarlo en conjuntos de datos con un número de enlaces moderado. Este es el caso de nuestro dump reducido de Wikipedia, que como vimos en la Sección 4.1 tiene unos 63 mil enlaces, frente a los más de 5 millones



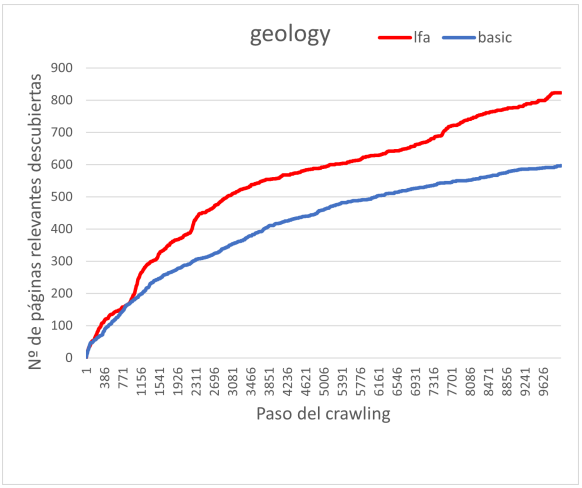
(a) Topic “cameras”



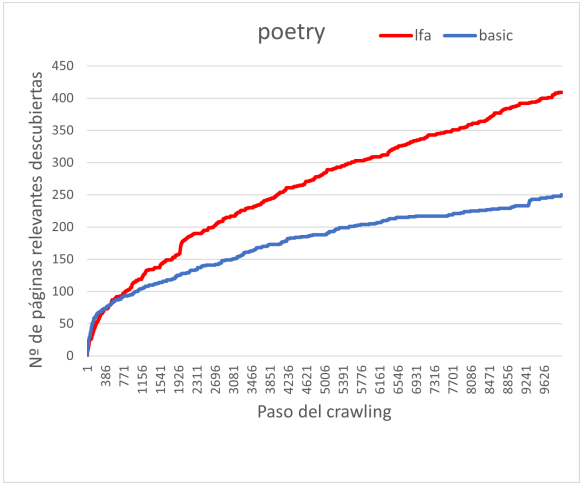
(b) Topic “cancer”



(c) Topic “fiction”



(d) Topic “geology”

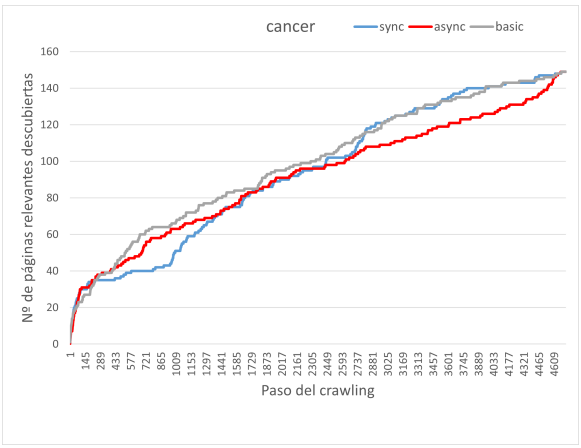


(e) Topic “poetry”

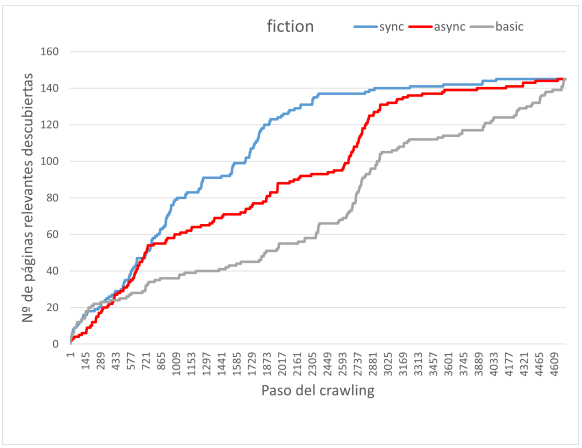
Figura 4.5: Comparativa de resultados entre el crawler LFA asíncrono y el básico para los topics: a) Cameras. b) Cancer. c) Fiction. d) Geology. e) Poetry.

que había en el dump completo. Sin embargo, esta colección cuenta con una media de 4 enlaces por página, y por tanto, hay muchos caminos hacia páginas relevantes que hemos perdido al realizar la reducción. A su vez, el número de páginas relevantes a cada topic que tenemos en total por descubrir es mucho más bajo también. Vamos a visualizar los resultados en cuatro topics, dejando fuera los otros dos por motivos similares a los que mencionamos en el apartado anterior: con los topics "cameras" y "olympics" la frontera del crawler se agota demasiado rápido, ya que partiendo desde los artículos con títulos homónimos a esos topics encontramos muy pocas páginas. En la Gráfica 4.6 se muestra la comparativa para las ejecuciones de los crawlers básico, LFA asíncrono y LFA síncrono con el dump reducido de Wikipedia. Ejecutamos todos los crawlers durante diez mil pasos. Sin embargo, en las gráficas solo se muestran alrededor de 4700 pasos, ya que las fronteras se vacían alrededor de los cuatro mil pasos, debido a los pocos enlaces de salida que tiene cada página en esta colección. Esto explica también por qué todas las curvas acaban en el mismo punto, y es que todos parten de la misma semilla, y al haber tan pocas páginas disponibles a partir de esa página origen, todas van a ver el mismo número de páginas, y a su vez, van a encontrar el mismo número de páginas relevantes una vez terminado todo el proceso de crawling. Sin embargo, unos métodos descubren relevancia más rápido que otros. Podemos observar que, excepto en el topic "cancer", en todos los casos el crawler básico es el que más tarda en descubrir la relevancia. En el topic "cancer" las tres curvas están más o menos igualadas, destacando el básico al principio, y el síncrono a medio y largo plazo. Además, tanto en "fiction" como en "poetry", se puede visualizar claramente cómo el método síncrono es el que más rápido descubre la relevancia, estando su curva claramente por encima del resto desde pasos tempranos del crawling. En estos casos, el método asíncrono es un punto intermedio en rendimiento entre el síncrono y el básico.

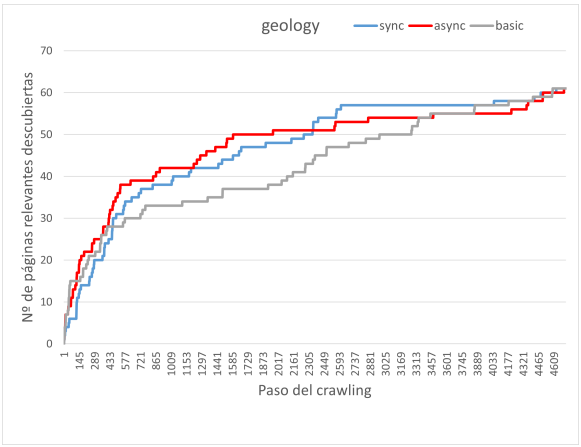
En definitiva, en los experimentos realizados sí observamos una mejora apreciable utilizando el método síncrono en comparación con el asíncrono. Sin embargo, por las condiciones del propio experimento, no podemos afirmar rotundamente que el método síncrono sea la mejor alternativa. Para ciertos topics, el método síncrono acelera el descubrimiento de relevancia, a costa de una mayor carga computacional, así que la decisión de qué método es más deseable dependerá del equilibrio que queramos conseguir entre rendimiento (tiempo de ejecución) y eficacia (relevancia acumulada).



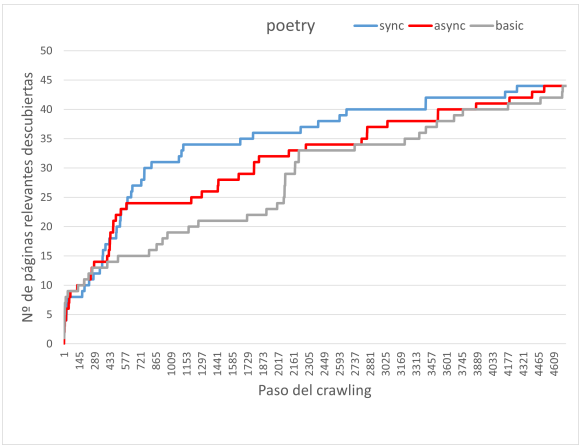
(a) Topic “cancer”



(b) Topic “fiction”



(c) Topic “geology”



(d) Topic “poetry”

Figura 4.6: Comparativa de resultados entre el crawler LFA asíncrono, LFA síncrono y el básico para los topics: a) Cancer. b) Fiction. c) Geology. d) Poetry.

5

Conclusiones y trabajo futuro

5.1. Conclusiones

Durante este trabajo hemos realizado un estudio del campo del Reinforcement Learning, y hemos analizado como integrarlo en el área del Crawling Web. A partir de este estudio, hemos propuesto tres alternativas de crawlers enfocados: uno básico que no usa técnicas de RL, uno novedoso basado en Deep Q-Learning (crawler DQN), y uno basado en Aproximación Lineal de Funciones (crawler ALF), que ya ha sido propuesto en la literatura [2]. Para poder implementar estos crawlers, ha sido necesario realizar un desarrollo extenso de diferentes subsistemas. En concreto, se ha implementado desde cero la gestión de los datos, utilizando colecciones de artículos de Wikipedia; la representación de las páginas como vectores, para poder calcular similitudes utilizando vectores Word2Vec; la representación de las páginas y enlaces con vectores de características, para poder ser utilizados como entrada en la Q-Network y en la Aproximación Lineal de Funciones; el modelado de la frontera como una cola con prioridad, incorporando mecanismos adicionales de extracción aleatoria de elementos y actualización de prioridades, para poder integrarlo con el uso de políticas ϵ-greedy; y la integración de todas estas partes en, primero, el crawler DQN, donde usamos la librería Keras-RL [29] para implementar el Deep Q-Learning, y después, en el crawler ALF, en sus dos modos de actualización de la frontera: asíncrono y síncrono. La implementación de todas estas funcionalidades ha sido el principal esfuerzo de este trabajo, ya que el enfoque se ha orientado más hacia el ámbito de la Ingeniería, y no tanto de la Investigación.

Una vez desarrollados todos los sistemas, hemos procedido a realizar un análisis experimental que nos ha permitido extraer varias conclusiones. La primera, y más general de todas, es que el campo del Reinforcement Learning puede ser aplicado de manera exitosa al campo del Crawling Web, conformando así un área totalmente novedosa

y prometedora para poder ser aplicada a problemas de Crawling Web. Además, aún centrándonos en un solo método de RL, la combinatoria de parámetros y diferentes configuraciones es enorme, por lo que se pueden realizar un gran número de pruebas variando la configuración, habiendo así un gran margen de mejora sobre nuestros resultados.

Centrándonos en el análisis de las tres propuestas de crawlers que hemos realizado, hemos podido observar que la opción de implementar un Deep Q-Learning con una red neuronal, que a la salida tenga tantas neuronas como acciones posibles no es una buena opción. La implementación del Deep Q-Learning es muy compleja, y es por ello que decidimos recurrir al uso de la librería Keras-RL, una de las pocas que actualmente ofrece una implementación suficiente de diferentes métodos de RL. Sin embargo, esta librería solo ofrece la posibilidad de usar un Deep Q-Learning con esa estructura de red, y es probablemente por esto que no hemos obtenido resultados satisfactorios con el crawler DQN, si bien la complejidad de la combinatoria experimental no nos ha permitido realizar la cantidad de experimentos necesarios para afirmar con rotundidad que es un método que no funciona satisfactoriamente en nuestro contexto.

Por otro lado, el análisis de los resultados del crawler LFA ha sido más interesante. Hemos analizado dos formas de ejecutar el crawler: con una actualización asíncrona de la frontera, y una actualización síncrona. Hemos concluido que la versión asíncrona conforma un método de crawling enfocado muy interesante, ya que ofrece muy buenos resultados sin tener que recurrir a tiempos de ejecución extensos. A su vez, el crawler síncrono parece ofrecer un descubrimiento de relevancia más rápido, pero su gran carga computacional hace que su propuesta sea menos interesante, pues su aplicación es demasiado lenta para obtener pequeñas mejoras con respecto a la versión asíncrona. Ambos métodos han superado con creces a la opción del crawler básico en nuestros experimentos, alcanzando la versión asíncrona hasta un 82 % de mejora en la relevancia acumulada de algunas de las pruebas.

5.2. Trabajo Futuro

Como hemos mencionado anteriormente, queda pendiente la implementación de un crawler basado en Deep Q-Learning que alcance unos resultados satisfactorios. Como principal línea de trabajo futuro, queda la propuesta de un crawler que use una Q-Network como aproximador, que a la entrada reciba el estado y la acción, y a la salida haya una sola neurona, cuyo valor sea el valor Q de tomar esa acción en ese estado. Con esta red estaríamos realizando una aproximación no lineal de la función Q, y resolveríamos los problemas de rendimiento derivados de la estructura de la Q-Network que hemos utilizado. Además, podríamos realizar multitud de experimentos adicionales, que nos permitan explorar mejor el gran espacio experimental que nos ha quedado pendiente en este trabajo.

A la hora de realizar experimentos, en el futuro podrían usarse diferentes conjuntos de datos, que exploren otros ámbitos fuera de las colecciones de Wikipedia. En concreto,

podríamos variar los experimentos con el uso de diferentes colecciones, la variación de los parámetros (diferentes usos de ϵ en la política, diferentes valores de γ en los agentes RL, etc.), el uso de nuevas características para modelar los estados y las acciones, la exploración de otras métricas en los experimentos (como la distribución de las similitudes Word2Vec en las páginas relevantes descubiertas), etc. Las posibilidades son enormes, y un análisis más complejo gracias al uso de todas estas diferentes configuraciones podría ayudarnos a llegar a nuevas conclusiones sobre los sistemas desarrollados.

Por último, podríamos profundizar en la forma en la que modelamos las páginas y los enlaces, y en cómo decidimos si una página es relevante o no. En [36] se proponen multitud de algoritmos que podríamos incorporar a la hora de modelar las características de páginas y enlaces: reducción de dimensionalidad, extracción de enlaces artificiales, extracción de características con la información de los vecinos, etc. También existen propuestas de clasificadores de textos, aplicables a páginas web, utilizando redes neuronales [37] [38]. Se podrían integrar estos clasificadores en nuestros sistemas, para mejorar la toma de decisiones sobre qué páginas son relevantes a un tema concreto.

Bibliografía

[1] Richard S. Sutton y Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[2] Miyoung Han, Pierre-Henri Wuillemin y Pierre Senellart. «Focused Crawling Through Reinforcement Learning». En: ene. de 2018, págs. 261-278. ISBN: 978-3-319-91661-3. DOI: 10.1007/978-3-319-91662-0_20.

[3] Xiangyu Zhao y col. *Deep Reinforcement Learning for List-wise Recommendations*. 2017. arXiv: 1801.00209 [cs.LG].

[4] Yu Lei y Wenjie Li. «Interactive Recommendation with User-Specific Deep Reinforcement Learning». En: *ACM Trans. Knowl. Discov. Data* 13.6 (oct. de 2019). ISSN: 1556-4681. DOI: 10.1145/3359554. URL: https://doi.org/10.1145/3359554.

[5] Bing Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*. Ene. de 2007. ISBN: 978-3-540-37881-5. DOI: 10.1007/978-3-642-19460-3.

[6] Filippo Menczer, Gautam Pant y Padmini Srinivasan. «Topical Web Crawlers: Evaluating Adaptive Algorithms». En: *ACM Trans. Internet Technol.* 4.4 (nov. de 2004), 378–419. ISSN: 1533-5399. DOI: 10.1145/1031114.1031117. URL: https://doi.org/10.1145/1031114.1031117.

[7] Padmini Srinivasan y col. «A General Evaluation Framework for Topical Crawlers». En: *Information Retrieval* 8 (ene. de 2005), págs. 417-. DOI: 10.1007/s10791-005-6993-5.

[8] Houqing Lu y col. «An Improved Focused Crawler: Using Web Page Classification and Link Priority Evaluation». En: *Mathematical Problems in Engineering* 2016 (ene. de 2016), págs. 1-10. DOI: 10.1155/2016/6406901.

[9] Lu Jiang y col. «Efficient Deep Web Crawling Using Reinforcement Learning». En: jun. de 2010, págs. 428-439. ISBN: 978-3-642-13656-6. DOI: 10.1007/978-3-642-13657-3_46.

[10] Brian D. Davison. «Topical Locality in the Web». En: SIGIR '00. Athens, Greece: Association for Computing Machinery, 2000, 272–279. ISBN: 1581132263. DOI: 10.1145/345508.345597. URL: https://doi.org/10.1145/345508.345597.

[11] R. Sabbatini. *Neurons and synapses: the history of its dicovery*. 2003. URL: http://www.cerebromente.org.br/n17/history/neurons3_i.htm.

[12] Warren S. McCulloch y Walter Pitts. «A Logical Calculus of the Ideas Immanent in Nervous Activity». En: *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, 1988, 15–27. ISBN: 0262010976.

[13] J. C. Cuevas-Tello. *Apuntes de Redes Neuronales Artificiales*. 2018. arXiv: 1806.05298 [cs.NE].

[14] David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams. «Learning Representations by Back-Propagating Errors». En: *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, 1988, 696–699. ISBN: 0262010976.

[15] Vinod Nair y Geoffrey E. Hinton. «Rectified Linear Units Improve Restricted Boltzmann Machines». En: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. Haifa, Israel: Omnipress, 2010, 807–814. ISBN: 9781605589077.

[16] Kaiming He y col. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. arXiv: 1502.01852 [cs.CV].

[17] Y. Bengio y X. Glorot. «Understanding the difficulty of training deep feed forward neural networks». En: *International Conference on Artificial Intelligence and Statistics* (ene. de 2010), págs. 249-256.

[18] Andrew Y. Ng. «Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance». En: *Proceedings of the Twenty-First International Conference on Machine Learning*. ICML '04. Banff, Alberta, Canada: Association for Computing Machinery, 2004, pág. 78. ISBN: 1581138385. DOI: 10.1145/1015330.1015435. URL: https://doi.org/10.1145/1015330.1015435.

[19] Nitish Srivastava y col. «Dropout: A Simple Way to Prevent Neural Networks from Overfitting». En: *Journal of Machine Learning Research* 15 (jun. de 2014), págs. 1929-1958.

[20] F.L. Lewis y Draguna Vrabie. «Reinforcement Learning and Adaptive Dynamic Programming for Feedback Control». En: *Circuits and Systems Magazine, IEEE* 9 (ene. de 2009), págs. 32 -50. DOI: 10.1109/MCAS.2009.933854.

[21] Alessandro Lazaric, Marcello Restelli y Andrea Bonarini. «Reinforcement Learning in Continuous Action Spaces through Sequential Monte Carlo Methods». En: *Proceedings of the 20th International Conference on Neural Information Processing Systems*. NIPS'07. Vancouver, British Columbia, Canada: Curran Associates Inc., 2007, 833–840. ISBN: 9781605603520.

[22] Lixin Zou y col. *Reinforcement Learning to Optimize Long-term User Engagement in Recommender Systems*. 2019. arXiv: 1902.05570 [cs.IR].

[23] Marc G. Bellemare y col. *Distributional reinforcement learning with linear function approximation*. 2019. arXiv: 1902.03149 [cs.LG].

[24] Francisco S. Melo, Sean P. Meyn y M. Isabel Ribeiro. «An Analysis of Reinforcement Learning with Function Approximation». En: *Proceedings of the 25th International Conference on Machine Learning*. ICML '08. Helsinki, Finland: Association for Computing Machinery, 2008, 664–671. ISBN: 9781605582054. DOI: 10.1145/1390156.1390240. URL: https://doi.org/10.1145/1390156.1390240.

[25] Ruosong Wang y col. *On Reward-Free Reinforcement Learning with Linear Function Approximation*. 2020. arXiv: 2006.11274 [cs.LG].

[26] Hado van Hasselt, Arthur Guez y David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: 1509.06461 [cs.LG].

[27] Scott Fujimoto, Herke van Hoof y David Meger. *Addressing Function Approximation Error in Actor-Critic Methods*. 2018. arXiv: 1802.09477 [cs.AI].

[28] Ziyu Wang y col. *Dueling Network Architectures for Deep Reinforcement Learning*. 2015. arXiv: 1511.06581 [cs.LG].

[29] Matthias Plappert. *keras-rl*. https://github.com/keras-rl/keras-rl. 2016.

[30] Jeffrey Pennington, Richard Socher y Christopher D. Manning. «GloVe: Global Vectors for Word Representation». En: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, págs. 1532-1543. URL: http://www.aclweb.org/anthology/D14-1162.

[31] Soumen Chakrabarti, Martin Berg y Byron Dom. «Focused crawling: A new approach to topic-specific Web resource discovery». En: *Computer Networks* 31 (abr. de 2000), págs. 1623-1640. DOI: 10.1016/S1389-1286(99)00052-3.

[32] Michelangelo Diligenti y col. «Focused Crawling Using Context Graphs». En: *Proceedings of the International Conference on Very Large Databases (VLDB-00), 2000* (dic. de 2000).

[33] George Almpanidis, C. Kotropoulos y Ioannis Pitas. «Combining text and link analysis for focused crawling–an application for vertical search engines Information Systems». En: *Inf. Syst.* 32 (sep. de 2007), págs. 886-908. DOI: 10.1016/j.is.2006.09.004.

[34] Michael Chau y Hsinchun Chen. «A Machine Learning Approach to Web Page Filtering Using Content and Structure Analysis». En: *Decis. Support Syst.* 44.2 (ene. de 2008), 482–494. ISSN: 0167-9236. DOI: 10.1016/j.dss.2007.06.002. URL: https://doi.org/10.1016/j.dss.2007.06.002.

[35] Greg Brockman y col. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[36] Xiaoguang Qi y Brian D. Davison. «Web Page Classification: Features and Algorithms». En: *ACM Comput. Surv.* 41.2 (feb. de 2009). ISSN: 0360-0300. DOI: 10.1145/1459352.1459357. URL: https://doi.org/10.1145/1459352.1459357.

[37] Chunting Zhou y col. *A C-LSTM Neural Network for Text Classification*. 2015. arXiv: 1511.08630 [cs.CL].

[38] Siwei Lai y col. «Recurrent Convolutional Neural Networks for Text Classification». En: *Proceedings of the AAAI Conference on Artificial Intelligence* 29.1 (2015). URL: https://ojs.aaai.org/index.php/AAAI/article/view/9513.

Apéndices

A

Código desarrollado

A.1. Definiendo el sistema de representación de páginas (dumps de Wikipedia)

```
import random
import os
import re
import string

from src.crawler.util.link import Link
from src.crawler.util.page import Page
from src.crawler.util.util import get_substring_positions,
↪  get_surrounding_texts

"""
In this file, we expect to receive a dump structured in 3 main
files and directories:
    * An "index.txt", where each line has the id and title of each
    page. Indexes must be normalized between 0 and n_pages.
    * A "links_n.txt", where each line is in the form:
            -> id_from \t id_to \n
    * An xml directory, where there is a file for each page. Each file
    is named as "id".xml, and contains an XML in the form as specified
    in the page.py module (method save).
"""
```

```
class DumpIndex:
    def __init__(self):
        """
        Initializes an Index object, with 2 components:

            idx2title: dict with idx (int) as key, and title (String)
            as value.
            title2idx: dict with title (String) as key, and idx (int)
            as value.
            n_pages: number of pages.
        """
        self.idx2title = {}
        self.title2idx = {}
        self.n_pages = 0

    def add_entry(self, idx, title):
        normalized_title = title.lower()
        self.idx2title[int(idx)] = normalized_title
        self.title2idx[normalized_title] = int(idx)
        self.n_pages += 1

    def is_title_in(self, title):
        return title.lower() in self.title2idx

    def is_idx_in(self, idx):
        return idx in self.idx2title

    def get_random_idx(self, n_idx):
        return random.choices(list(self.idx2title.keys()), k=n_idx)

    def get_random_titles(self, n_titles):
        return random.choices(list(self.title2idx.keys()), k=n_titles)

    def save(self, path):
        index_path = os.path.join(path, "index.txt")
        with open(index_path, "w", encoding='utf-8') as f:
            pass

        for idx in self.idx2title.keys():
            title = self.idx2title[idx]
            with open(index_path, "a", encoding='utf-8') as f:
                f.write(str(idx) + "\t" + title + "\n")
```

```
class DumpLinks:
    def __init__(self):
        """
        Initializes a Links object, with 2 components:

            outlinks: dict with idx (int) as key, and list of idx as
            value. It contains the pages that the key-page is linked
            to.
            inlinks: dict with idx (int) as key, and list of idx as
            value. It contains the pages that points to the key-page.
        """
        self.outlinks = {}
        self.inlinks = {}
        self.n_links = 0

    def add_entry(self, idx_from, idx_to):
        if idx_from not in self.outlinks:
            self.outlinks[idx_from] = []
        self.outlinks[idx_from].append(idx_to)

        if idx_to not in self.inlinks:
            self.inlinks[idx_to] = []
        self.inlinks[idx_to].append(idx_from)

        self.n_links += 1

    def save(self, path):
        links_path = os.path.join(path, "links_n.txt")
        with open(links_path, "w", encoding='utf-8') as f:
            pass

        for idx_from in self.outlinks:
            for idx_to in self.outlinks[idx_from]:
                with open(links_path, "a", encoding='utf-8') as f:
                    f.write(str(idx_from) + "\t" + str(idx_to) + "\n")

    def get_avg_links_per_page(self):
        return self.n_links / len(self.outlinks.keys())


class WebDump:
    def __init__(self, dump_path, index_path, links_path):
        self.dump_path = dump_path
        self.xml_path = os.path.join(dump_path, "xml")
        self.preprocessed_xml_path = os.path.join(dump_path,
        ↪  "preprocessed_xml")
```

```
        self.index_path = index_path
        self.links_path = links_path
        self.index = DumpIndex()
        self.links = DumpLinks()

        with open(self.index_path, "r", encoding='utf-8') as f:
            lines = f.read().splitlines()
            for line in lines:
                idx, title = line.split("\t")
                self.index.add_entry(idx, title)

        with open(self.links_path, "r", encoding='utf-8') as f:
            lines = f.read().splitlines()
            for line in lines:
                tokens = line.split("\t")
                idx_from = int(tokens[0])
                idx_to = int(tokens[1])
                # No queremos autoenlaces
                if idx_from == idx_to:
                    continue
                # Solo tenemos en cuenta el enlace si ambas paginas
                # existen en nuestro dump
                if self.index.is_idx_in(idx_from):
                    if self.index.is_idx_in(idx_to):
                        self.links.add_entry(idx_from, idx_to)

    def get_all_doc_idx(self):
        return self.index.idx2title.keys()

    def idx2title(self, idx):
        return self.index.idx2title[idx]

    def title2idx(self, title):
        if title not in self.index.title2idx:
            return None
        return self.index.title2idx[title]

    def get_outlinks_from_idx(self, idx):
        if idx in self.links.outlinks:
            return self.links.outlinks[idx]
        else:
            return []

    def get_pages_outlinks_from_idx(self, idx):
        if idx in self.links.outlinks:
            children_idx = self.links.outlinks[idx]
```

```
            pages = []
            for child in children_idx:
                pages.append(self.get_page(child))
            return pages
        else:
            return []

    def get_inlinks_from_idx(self, idx):
        if idx in self.links.inlinks:
            return self.links.inlinks[idx]
        else:
            return []

    def get_outlinks_from_title(self, title):
        if self.index.title2idx[title] in self.links.outlinks:
            return self.links.outlinks[self.index.title2idx[title]]
        else:
            return []

    def get_inlinks_from_title(self, title):
        if self.index.title2idx[title] in self.links.inlinks:
            return self.links.inlinks[self.index.title2idx[title]]
        else:
            return []

    def get_preprocessed_html(self, idx):
        page_path = os.path.join(self.preprocessed_xml_path, str(idx) +
        ↪  ".txt")
        with open(page_path, "r", encoding='utf-8') as f:
            html = f.read()
        return html

    def get_html(self, idx, check_redirects=True):
        page_path = os.path.join(self.xml_path, str(idx) + ".xml")
        with open(page_path, "r", encoding='utf-8') as f:
            html = f.read().lower()

        # Algunas paginas, en el campo text, tienen un redirect a otra
        ↪  pagina.
        # En estos casos, devolvemos el xml de esa pagina.
        if check_redirects:
            if "#redirect" in html:
                result = re.search('\[\[(.*)\]\]', html).group(1)
                new_idx = self.title2idx(result)
                if new_idx is not None and new_idx != idx:
                    return self.get_html(new_idx)
```

```
        return html

    def get_random_seed(self, n_pages):
        return self.index.get_random_idx(n_pages)

    def get_random_seed_with_links(self, n_pages):
        candidates = list(self.links.outlinks.keys())
        return random.sample(candidates, n_pages)

    def get_topic_seed(self, topic):
        if topic in self.index.title2idx:
            return [self.index.title2idx[topic]]
        return None

    def n_pages(self):
        return self.index.n_pages

    def get_all_pages_idx(self):
        return list(self.index.idx2title.keys())

    def save_links(self, path):
        self.links.save(path)

    def save_index(self, path):
        self.index.save(path)

    def save(self, path):
        self.save_index(path)
        self.save_links(path)

def is_absolute(url):
    """
    is_absolute: It returns True if the URL is absolute

    SYNTAX:
        b = is_absolute(url)

    INPUTS:
        url: URL to be checked

    OUTPUTS:
        b: Boolean. True if the url is absolute

    """
    return "//" in url or "www." in url
```

```
def normalize_url(url, base=None):
    """
    normalize_url: It returns an URL in its canonical form

    SYNTAX:
        url = normalize_url(url, base)

    INPUTS:
        url: URL to transform in its canonical form
        base: Default None. Base URL which url cames from

    OUTPUTS:
        url: URL in its canonical form
    """
    if len(url) <= 0:
        return base

    if not is_absolute(url):
        if url[0] == '#':
            return base

        url = urljoin(base, url)

    if url[0] == " ":
        url = url[1:]
    # Procesar urls a canonicas
    url = url.replace(" ", "%20")

    return url


def url_base(url):
    """
    url_base: It returns the base URL from the url

    SYNTAX:
        base = url_base(url)

    INPUTS:
        url: URL which we want to extract its base

    OUTPUTS:
        base: Base URL from url
    """
    split_url = urlsplit(url)
```

```
    return "http://" + split_url.netloc + "/"


def does_url_exist(url):
    """
    does_url_exist: It returns if the URL exists

    SYNTAX:
        b = does_url_exist(url)

    INPUTS:
        url: URL to be checked

    OUTPUTS:
        b: Boolean. True if url returns code 200

    """
    try:
        r = requests.head(url)
        return r.status_code == 200
    except requests.exceptions.RequestException as e:
        print(e)
        return False
    except:
        return False


def extension_from_url(url):
    """
    extension_from_url: It returns the extension from the url checking
↪   its header. We trust header is well formed.

    Siempre tiene la forma "text/extension; charset=charset"
    Por tanto, parseamos primero por espacios, para quedarnos con
↪   "text/extension;", y despues parseamos con
    el caracter /, y eliminamos el ; del final. Le añadimos el . que
↪   precede a la extension por conveniencia.


    SYNTAX:
        ext = extension_from_url(url)

    INPUTS:
        url: URL to be checked

    OUTPUTS:
```

```
        ext: Extension in format .ext (.html, .php, .plain...)
    """
    response = requests.get(url)
    content_type = response.headers['content-type']
    return "." + content_type.split(";")[0].split("/")[1]


def word_contains_token(word, list_of_tokens):
    """
        Cabecera: salida = word_contains_token(word, list_of_tokens)

        Definición:
          Comprueba si una cadena contiene un token, dentro de una lista
↪   de tokens.

        Inputs:
          word: string, palabra donde comprobar si contiene uno de los
↪   token.
          list_of_tokens: list of strings, lista de tokens a comprobar.

        Outputs:
          salida: boolean indicando si la palabra contenía uno de los
↪   token.
    """
    return any(token in word for token in list_of_tokens)


def delete_html_tags(html):
    """

    Limpia las etiquetas HTML de una cadena.

    :param html: cadena conteniendo el HTML a limpiar.
    :return: cadena sin etiquetas HTML.
    """
    cleanr = re.compile('<.*?>')
    try:
        cleantext = re.sub(cleanr, '', html)
    except TypeError:
        print(html)
        return None
    return cleantext
```

A.2. Definiendo el sistema de vectorización de páginas

```
import os
from abc import ABC

import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer

import abc
import math


class AbstractDumpVectorizer(ABC):
    """
    Class that transforms dump pages into vectors. For representing
    pages as vector representations:

        1. It receives a file, containing pre-trained Word2Vec vecs,
        one line for each word.
        2. It computes the tf-idf of each word, using the dump for it.
        3. It computes the Doc2Vec representation of each page:
        doc2vec = (tf-idf) * avg(Word2Vec)

        In other words:
        It computes the average of the Word2vec vectors of the words
        that the page contains, and this vector is weighted by the
        tf-idf score of each word.

    Certainly, this class is an abstract definition, that only
    initializes the basic attributes needed, and preloads the idf
    score of each word. For that purpose, it uses a pre-train phase
    dump. In other words, one dump is used only for idf pre-compute
    phase, and another one is used later for the train phase.

    A child class must be implemented defining how are document
    vectors computed.
    """

    def _load_word_vecs(self, word2vec_path, n_dims):
        """
        Reads Word2Vec pre-trained vectors, and returns the vocabulary
        and vectors.
        :param word2vec_path: (String) path to the Word2Vec file
        with vectors pre-trained.
        :param n_dims: (int) number of dimensions of Word2Vec vectors.
```

```
        :return: (list, np.2darray), a list with all the words present
        in the file; and a matrix with all the vecs of each word.
        """
        with open(word2vec_path, "r", encoding='utf-8') as f:
            lines = f.readlines()

        vocabulary = []
        n_words = len(lines)
        vecs = np.empty([n_words, n_dims])
        words_position = {}

        words_idx = 0
        for line in lines:
            tokens = line.split()
            vocabulary.append(tokens[0])
            vecs[words_idx] = tokens[1:]
            words_position[tokens[0]] = words_idx
            words_idx += 1

        return vocabulary, vecs, words_position

    def _precompute_idfs_by_scikit(self, pretrain_dump, xml_path,
                                   save_path=None):
        files_names = [os.path.join(xml_path, str(i) + ".xml") for i
                       in range(0, pretrain_dump.n_pages())]
        tfidf_vectorizer = TfidfVectorizer(input='filename',
                                           vocabulary=self.vocabulary,
                                           smooth_idf=True,
                                           use_idf=True)
        tfidf_vectorizer_vectors = tfidf_vectorizer.fit(files_names)

        idfs = tfidf_vectorizer.idf_

        # Por la forma que tiene scikit de calcular los idfs, cuando una
        ↪ palabra
        # no aparece en la coleccion, le otorga el maximo idf. Puede que
        ↪ nosotros
        # queramos no tener en cuenta esos idfs a la hora de calcular
        ↪ vectores.
        """null_positions = np.where(idfs == self.max_idf)
        idfs[null_positions] = 0"""

        if save_path is not None:
            np.save(save_path, idfs)
        return idfs
```

```
    def _precompute_idfs(self, dump, save_path=None):
        """

        It precomputes the idf scores of all the words present in the
        vocabulary. For that purpose, it uses a pre-train dump, only
        used for this task.

        :param pretrain_dump: (Dump), dump to compute the idf scores.
        :return: (np.array), with the idfs for each word in the
        vocabulary.
        """

        freqs = np.zeros([self.n_words])
        document_indexes = dump.get_all_doc_idx()
        count = 0
        for idx in document_indexes:
            text = dump.get_html(idx, check_redirects=False)
            unique_words = set(text.split())
            for word in unique_words:
                if word in self.vocabulary:
                    freqs[self.words_position[word]] += 1
            count += 1
        idfs = np.log(dump.n_pages() + 1 / freqs + 1)
        return idfs

    def _load_idfs(self, load_path):
        return np.load(load_path)

    def __init__(self,
                 pretrain_dump,
                 word2vec_path,
                 word2vec_n_dims,
                 load_idfs=False,
                 load_path=None):
        """
        Initializes the abstract vectorizer. It loads the Word2Vec
        vectors, and calculates the idf score for each word.

        :param pretrain_dump: (Dump), dump used for calculating
        idf scores.
        :param word2vec_path: (String), path to a file with Word2Vec
        vectors.
        :param word2vec_n_dims: (String), number of dimensions of
        each vector.
        """
        self.vec_dims = word2vec_n_dims
```

```
        self.vocabulary, \
        self.word_vecs, \
        self.words_position = self._load_word_vecs(word2vec_path,
                                                   word2vec_n_dims)
        # Este es el valor idf que scikit le da a las palabras que no
        # estan presentes en ningun documento
        self.max_idf = math.log((1 + pretrain_dump.n_pages())) + 1

        self.n_words = self.word_vecs.shape[0]
        if load_idfs:
            if load_path is None:
                load_path = os.path.join(pretrain_dump.dump_path,
                                         "idfs_matrix.npy")
            self.idfs = np.load(load_path)
        else:
            save_path = os.path.join(pretrain_dump.dump_path,
                                     "idfs_matrix")
            xml_path = os.path.join(pretrain_dump.dump_path,
                                    "xml")
            self.idfs = self._precompute_idfs_by_scikit(pretrain_dump,
                                                        xml_path,
                                                        save_path)
            np.save(save_path, self.idfs)

    @abc.abstractmethod
    def get_doc_vec(self, idx):
        """
        Get the vector representation of the document which index is
        idx.
        :param idx: int, index.
        :return: np.array, with vector of the document.
        """
        pass


class OnlineDoc2Vec(AbstractDumpVectorizer):

    def __init__(self,
                 dump,
                 pretrain_dump,
                 word2vec_path,
                 word2vec_n_dims,
                 load_idfs=False,
                 load_path=None):
        """
        Dump Vectorizer that calculates the vector representation of
```

```
        each document in the moment the agent sees it.

        :param dump: (Dump), dump used during train.
        :param pretrain_dump: (Dump), dump used for calculating
        idf scores.
        :param word2vec_path: (String), path to a file with Word2Vec
        vectors.
        :param word2vec_n_dims: (String), number of dimensions of
        each vector.
        """
        super().__init__(pretrain_dump,
                         word2vec_path,
                         word2vec_n_dims,
                         load_idfs,
                         load_path)
        self.dump = dump
        #self.tfidf_computer = TFIDFDumpComputer(dump)
        self.doc_vecs = {}

    def tf(self, text, word):
        """
        Returns the term frequency of a word, in a document.
        :param text: text where to count.
        :param word: word to look for.
        :return: term frequency.
        """
        freq = text.count(word)
        if freq > 0:
            return 1 + math.log(freq, 2)
        else:
            return 0

    def get_doc_vec(self, doc_idx):
        """
        If the document is already seen, it only returns the
        already calculated document vector. If not, the document is
        first calculated and then returned.

        :param doc_idx: (int), index of the document.
        :return: (np.array), vector representation of the document.
        """
        if doc_idx not in self.doc_vecs:
            html = self.dump.get_preprocessed_html(doc_idx)
            self.doc_vecs[doc_idx] = self.text2vec(html)
        return self.doc_vecs[doc_idx]
```

```
    def get_word_vec(self, word):
        """
        Gets the vector representation of the word. Can be used for
        the topic.

        :param word: word to transform by Word2Vec.
        :return: (np.array), vector representation.
        """
        return self.word_vecs[self.words_position[word]]

    def text2vec(self, text):
        text_vec = np.zeros([self.vec_dims])
        unique_words = set(text.split())
        n_words_in_doc = len(unique_words)

        for word in unique_words:
            if word in self.words_position:
                i = self.words_position[word]
                tf = self.tf(text, word)
                if tf > 0:
                    text_vec += (tf * self.idfs[i]) * self.word_vecs[i]

        text_vec /= n_words_in_doc
        return text_vec
```

A.3. Gestionando la extracción de características de las páginas

```
class Page:

    def __init__(self, idx, url, html, vec, parents=None,
    ↪  tokens_blacklist=None):
        """

        It initializes a page, preprocessing the HTML (deleting the
        tags) and initializing the parents array.

        :param id: idx of the page.
        :param url: URL of the page.
        :param html: plain HTML of the page.
        """
        self.url = url
        self.idx = idx
```

```
        self.html = html
        self.vec = vec
        if parents is not None:
            self.parents = parents
        else:
            self.parents = []

        # Símbolos y palabras que ignorar en el procesamiento de texto
        if tokens_blacklist is None:
            symbols = ['!', '"', '(', ')', '[', ']', '{', '}', '|',
                       '<', '>', '/', '.', '\n', ';', '?', '&', '_',
                       ',', ':', '=']
            html_specific_words = ['px', 'rem', 'undefined', 'if',
                                   'typeof', '@']
            tokens_blacklist = symbols + html_specific_words

        self.preprocessed_html = preprocess_html(self.html,
                                                 tokens_blacklist)

    def add_parent(self, page):
        """
        Adds a parent to the page.
        :param page: parent page.
        :return: True if the parent was effectively added, False if
        the parent was already present in the current page.
        """
        if page not in self.parents:
            self.parents.append(page)
            return True
        return False

    def n_parents(self):
        return len(self.parents)

    def save(self, path):
        """
        It generates an XML representation of the page, and stores it
        at the path file.
        :param path: path of the file where to store the page.
        :return: void.

        Example of XML:
        <page>
            <id>1</id>
            <title>Example</title>
            <text>This is only an example of page</text>
```

```
        </page>
        """
        # create XML
        root = etree.Element('page')
        id_element = etree.Element('id')
        id_element.text = str(self.idx)
        root.append(id_element)
        title_element = etree.Element('title')
        title_element.text = str(self.url)
        root.append(title_element)
        # another child with text
        text_element = etree.Element('text')
        text_element.text = str(self.html)
        root.append(text_element)

        # pretty string
        s = etree.tostring(root, encoding='unicode',
                           pretty_print=True)
        with open(path, "w", encoding='utf-8') as f:
            f.write(s)

    def __eq__(self, other):
        return self.idx == other.idx

    def __hash__(self):
        return self.idx

class OptimizedPageFeaturizer:
    def __init__(self,
                 topic,
                 topic_vec,
                 smoothing_factor,
                 categories=None,
                 max_distance=9,
                 discretize=False,
                 discretize_sigma1=0.1,
                 discretize_sigma2=0.3):
        self.topic = topic
        self.topic_vec = topic_vec
        self.smoothing_factor = smoothing_factor
        self.max_distance = max_distance
        self.discretize = discretize
        self.sigma1 = discretize_sigma1
        self.sigma2 = discretize_sigma2
        # This categories must be given in vector form
        if categories is None:
```

```
            self.categories = []
        else:
            self.categories = categories

        self.topic_relevances = {}
        self.parents_avg_dict = {}
        self.rel_parents_avg_dict = {}
        self.rel_parents_count = {}
        self.rel_distances = {}
        self.categories_relevances = {}
        self.max_wrl_dict = {}

    def reset(self):
        self.topic_relevances = {}
        self.parents_avg_dict = {}
        self.rel_parents_avg_dict = {}
        self.rel_parents_count = {}
        self.rel_distances = {}
        self.categories_relevances = {}
        self.max_wrl_dict = {}

    def is_relevant(self, page):
        return self.topic in page.html

    def _update_all_parents_avg(self, page, parent):
        new_rel = self.topic_relevances[parent]
        old_avg = self.parents_avg_dict[page]
        new_avg = old_avg + ((new_rel - old_avg) / page.n_parents())
        self.parents_avg_dict[page] = new_avg

    def _update_rel_parents_avg(self, page, parent):
        if self.is_relevant(parent):
            new_rel = self.topic_relevances[parent]
            old_avg = self.rel_parents_avg_dict[page]
            self.rel_parents_count[page] += 1
            new_avg = old_avg + ((new_rel - old_avg) /
             ↪  self.rel_parents_count[page])
            self.rel_parents_avg_dict[page] = new_avg

    def _update_rel_distance(self, page, parent):
        if not self.is_relevant(page):
            if self.rel_distances[parent] + 1 < self.rel_distances[page]:
                self.rel_distances[page] = self.rel_distances[parent] + 1

    def _update_max_wrl(self, page, parent):
        if self.max_wrl_dict[parent] > self.max_wrl_dict[page]:
```

```
            self.max_wrl_dict[page] = self.max_wrl_dict[parent]

    def add_parent(self, page, parent):
        # Actualizamos las metricas solo si el parent no estaba
        # ya en la lista de parents de la pagina.
        if page.add_parent(parent):
            if parent not in self.rel_distances:
                self._initialize_page(parent)
            self._update_all_parents_avg(page, parent)
            self._update_rel_parents_avg(page, parent)
            self._update_rel_distance(page, parent)
            self._update_max_wrl(page, parent)

    def topic_relevance(self, page):
        return cosine_sim(page.vec, self.topic_vec)

    def all_parents_avg(self, page):
        if page.n_parents() <= 0:
            return 0.0

        avg_relevance = 0
        for parent in page.parents:
            avg_relevance += self.topic_relevance(parent)
        return avg_relevance / page.n_parents()

    def rel_parents_avg(self, page):
        avg_relevance = 0
        n_relevant_parents = 0
        for parent in page.parents:
            if self.is_relevant(parent):
                n_relevant_parents += 1
                avg_relevance += self.topic_relevance(parent)
        if n_relevant_parents <= 0:
            return avg_relevance, n_relevant_parents
        return avg_relevance / n_relevant_parents, n_relevant_parents

    def distance_to_last_rel_page(self, page):
        if self.is_relevant(page):
            return 0
        elif page.n_parents() <= 0:
            return self.max_distance
        else:
            min_distance = float('inf')
            for parent in page.parents:
                # Si da error aqui porque el parent no esta en
                ↪  rel_distances,
```

```
                # hay que inicializar los parents en LFACrawler llamando
                ↪ al
                # metodo add_parent de este modulo.
                parent_dist = self.rel_distances[parent]
                if parent_dist + 1 < min_distance:
                    min_distance = parent_dist + 1

            if min_distance > self.max_distance:
                return self.max_distance
            return min_distance

    def categories_relevance(self, page):
        relevances = []
        for category in self.categories:
            if self.discretize:
                i, j = discretize_relevance(cosine_sim(page.vec,
                ↪ category))
                relevances.append(i)
                relevances.append(j)
            else:
                relevances.append(cosine_sim(page.vec, category))
        return relevances

    def wrl(self, page):
        a = self.smoothing_factor * self.topic_relevances[page]
        b = (1 - self.smoothing_factor) * self.max_wrl_dict[page]
        return a + b

    def max_wrl(self, page):
        if page.n_parents() <= 0:
            return self.topic_relevances[page]
        else:
            max_wrl = -1
            for parent in page.parents:
                wrl = self.wrl(parent)
                if wrl > max_wrl:
                    max_wrl = wrl
            return max_wrl

    def relevance_change(self, page):
        return self.topic_relevances[page] - self.wrl(page)

    def _initialize_page(self, page):
        self.topic_relevances[page] = self.topic_relevance(page)
        self.parents_avg_dict[page] = self.all_parents_avg(page)
        rel_parents_avg, n_rel_parents = self.rel_parents_avg(page)
```

```
        self.rel_parents_avg_dict[page] = rel_parents_avg
        self.rel_parents_count[page] = n_rel_parents
        self.rel_distances[page] = self.distance_to_last_rel_page(page)
        self.categories_relevances[page] =
        ↪  self.categories_relevance(page)
        self.max_wrl_dict[page] = self.max_wrl(page)

    def _discretize_relevance_change(self, rel_change):
        if 0 <= rel_change < self.sigma1:
            return 0
        elif self.sigma1 <= rel_change < self.sigma2:
            return 1
        elif rel_change >= self.sigma2:
            return 2
        elif -self.sigma2 <= rel_change < 0:
            return 3
        elif rel_change < -self.sigma2:
            return 4

    def to_feat(self, page):
        # Es la primera vez que vemos la pagina
        if page not in self.topic_relevances:
            self._initialize_page(page)
        if self.discretize:
            topic_relevance_ij =
            ↪  discretize_relevance(self.topic_relevances[page])
            avg_all_parents_ij =
            ↪  discretize_relevance(self.parents_avg_dict[page])
            avg_rel_parents_ij =
            ↪  discretize_relevance(self.rel_parents_avg_dict[page])
            rel_change =
            ↪  self._discretize_relevance_change(self.relevance_change(page))
            return [topic_relevance_ij[0],
                    topic_relevance_ij[1],
                    avg_all_parents_ij[0],
                    avg_all_parents_ij[1],
                    avg_rel_parents_ij[0],
                    avg_rel_parents_ij[1],
                    self.discretize_distance(self.rel_distances[page]),
                    rel_change] + self.categories_relevances[page]
        else:
            return [self.topic_relevances[page],
                    self.parents_avg_dict[page],
                    self.rel_parents_avg_dict[page],
                    self.rel_distances[page],
                    self.relevance_change(page)] \
```

```
                + self.categories_relevances[page]

    def discretize_distance(self, distance):
        return self.max_distance - distance

    def n_feat(self):
        # Topic rel, rel change, avg of all parents, avg of rel
        # parents, distance from last rel page, and a rel for each
        # category
        if self.discretize:
            return 8 + 2*len(self.categories)
        else:
            return 5 + len(self.categories)


class SimpleOptimizedPageFeaturizer(OptimizedPageFeaturizer):

    def categories_relevance(self, page):
        relevances = []
        for category in self.categories:
            if self.discretize:

                ↪  relevances.append(discretize_relevance_simple(cosine_sim(page.vec,
                ↪  category)))
            else:
                relevances.append(cosine_sim(page.vec, category))
        return relevances

    def to_feat(self, page):
        # Es la primera vez que vemos la pagina
        if page not in self.topic_relevances:
            super()._initialize_page(page)
        if self.discretize:
            return
             ↪  [discretize_relevance_simple(self.topic_relevances[page]),

                    ↪  discretize_relevance_simple(self.parents_avg_dict[page]),

                    ↪  discretize_relevance_simple(self.rel_parents_avg_dict[page]),

                    ↪  self._discretize_relevance_change(self.relevance_change(page)),
                   self.discretize_distance(self.rel_distances[page])] +
                    ↪  self.categories_relevances[page]
        else:
            super().to_feat(page)
```

```
    def n_feat(self):
        return 5 + len(self.categories)

def discretize_relevance(value):
    i = 0
    j = 0
    if value < 0.2:
        i = 0
    elif 0.2 <= value < 0.4:
        i = 1
    elif 0.4 <= value < 0.6:
        i = 2
    elif 0.6 <= value < 0.8:
        i = 3
    elif 0.8 <= value <= 1.0:
        i = 4

    if value < 0.1:
        j = 0
    elif 0.1 <= value < 0.3:
        j = 1
    elif 0.3 <= value < 0.5:
        j = 2
    elif 0.5 <= value < 0.7:
        j = 3
    elif 0.7 <= value < 0.9:
        j = 4
    elif 0.9 <= value <= 1.0:
        j = 5

    return i, j


def discretize_relevance_simple(value):
    if value < 0:
        return 0
    return int(value * 10)

def cosine_sim(x, y):
    # If x or y are all zeros, the sim is 0
    if all(v == 0 for v in x) or all(v == 0 for v in y):
        return 0
    return (x @ y.T) / (norm(x)*norm(y))
```

A.4. Gestionando la extracción de características de los enlaces

```
class Link:
    def __init__(self,
                 idx_to,
                 url,
                 vec,
                 parent=None):
        """

        It initializes a link, containing its URL and its anchor
        text.

        :param url: (string), URL of the link.
        :param anchor_text: (string); anchor text of the link.
        :param surrounding_texts: (list of strings), text near each
                                    appearance of the url in the page.
        """
        self.parents = []
        self.idx_to = idx_to
        self.url = url
        self.vec = vec

        if parent:
            self.add_parent(parent)

    def add_parent(self, parent):
        if parent not in self.parents:
            self.parents.append(parent)
            return True
        return False

    def n_parents(self):
        return len(self.parents)

    def get_vec(self):
        return self.vec

    def __eq__(self, other):
        return self.idx_to == other.idx_to

    def __hash__(self):
        return self.idx_to
```

```
from src.crawler.util.util import discretize_relevance,
↪  discretize_relevance_simple
from src.dump.sim import cosine_sim


class OptimizedLinkFeaturizer:
    def __init__(self,
                 topic,
                 topic_vec,
                 page_featurizer,
                 categories=None,
                 discretize=False):
        self.topic = topic
        self.topic_vec = topic_vec
        self.page_featurizer = page_featurizer
        self.discretize = discretize
        if categories is not None:
            self.categories = categories
        else:
            self.categories = []

        self.topic_relevances = {}
        self.parents_avg_dict = {}
        self.rel_parents_avg_dict = {}
        self.rel_parents_count = {}
        self.categories_relevances = {}

    def reset(self):
        self.topic_relevances = {}
        self.parents_avg_dict = {}
        self.rel_parents_avg_dict = {}
        self.rel_parents_count = {}
        self.categories_relevances = {}

    def is_relevant(self, page):
        return self.page_featurizer.is_relevant(page)

    def _update_all_parents_avg(self, link, parent_relevance):
        new_rel = parent_relevance
        old_avg = self.parents_avg_dict[link]
        new_avg = old_avg + ((new_rel - old_avg) / link.n_parents())
        self.parents_avg_dict[link] = new_avg

    def _update_rel_parents_avg(self, link, parent, parent_relevance):
        if self.is_relevant(parent):
```

```
            new_rel = parent_relevance
            old_avg = self.rel_parents_avg_dict[link]
            self.rel_parents_count[link] += 1
            new_avg = old_avg + ((new_rel - old_avg) /
            ↪  self.rel_parents_count[link])
            self.rel_parents_avg_dict[link] = new_avg

    def add_parent(self, link, parent):
        # Actualizamos las metricas solo si el parent no estaba
        # ya en la lista de parents de la pagina.
        if link.add_parent(parent):
            if link not in self.topic_relevances:
                self._initialize_link(link)
            parent_relevance =
            ↪  self.page_featurizer.topic_relevance(parent)
            self._update_all_parents_avg(link, parent_relevance)
            self._update_rel_parents_avg(link, parent, parent_relevance)

    def topic_relevance(self, link):
        return cosine_sim(link.vec, self.topic_vec)

    def all_parents_avg(self, link):
        if link.n_parents() <= 0:
            return 0.0

        avg_relevance = 0
        for parent in link.parents:
            avg_relevance += self.page_featurizer.topic_relevance(parent)
        return avg_relevance / link.n_parents()

    def rel_parents_avg(self, link):
        avg_relevance = 0
        n_relevant_parents = 0
        for parent in link.parents:
            if self.is_relevant(parent):
                n_relevant_parents += 1
                avg_relevance +=
                ↪  self.page_featurizer.topic_relevance(parent)
        if n_relevant_parents <= 0:
            return avg_relevance, n_relevant_parents
        return avg_relevance / n_relevant_parents, n_relevant_parents

    def categories_relevance(self, link):
        relevances = []
        for category in self.categories:
            if self.discretize:
```

```
                i, j = discretize_relevance(cosine_sim(link.vec,
                ↪  category))
                relevances.append(i)
                relevances.append(j)
            else:
                relevances.append(cosine_sim(link.vec, category))
        return relevances

    def _initialize_link(self, link):
        self.topic_relevances[link] = self.topic_relevance(link)
        self.parents_avg_dict[link] = self.all_parents_avg(link)
        rel_parents_avg, n_rel_parents = self.rel_parents_avg(link)
        self.rel_parents_avg_dict[link] = rel_parents_avg
        self.rel_parents_count[link] = n_rel_parents
        self.categories_relevances[link] =
        ↪  self.categories_relevance(link)

    def to_feat(self, link):
        # Es la primera vez que vemos el enlace
        if link not in self.topic_relevances:
            self._initialize_link(link)
        if self.discretize:
            topic_relevance_ij =
            ↪  discretize_relevance(self.topic_relevances[link])
            avg_all_parents_ij =
            ↪  discretize_relevance(self.parents_avg_dict[link])
            avg_rel_parents_ij =
            ↪  discretize_relevance(self.rel_parents_avg_dict[link])
            return [topic_relevance_ij[0],
                    topic_relevance_ij[1],
                    avg_all_parents_ij[0],
                    avg_all_parents_ij[1],
                    avg_rel_parents_ij[0],
                    avg_rel_parents_ij[1]] +
                    ↪  self.categories_relevances[link]
        else:
            return [self.topic_relevances[link],
                    self.parents_avg_dict[link],
                    self.rel_parents_avg_dict[link]] \
                   + self.categories_relevances[link]

    def n_feat(self):
        if self.discretize:
            return 6 + 2*len(self.categories)
        else:
            return 3 + len(self.categories)
```

```
class SimpleOptimizedLinkFeaturizer(OptimizedLinkFeaturizer):

    def categories_relevance(self, link):
        relevances = []
        for category in self.categories:
            if self.discretize:

                ↪ relevances.append(discretize_relevance_simple(cosine_sim(link.vec,
                ↪ category)))
            else:
                relevances.append(cosine_sim(link.vec, category))
        return relevances

    def to_feat(self, link):
        # Es la primera vez que vemos la pagina
        if link not in self.topic_relevances:
            super()._initialize_link(link)
        if self.discretize:
            return
             ↪ [discretize_relevance_simple(self.topic_relevances[link]),

                    ↪ discretize_relevance_simple(self.parents_avg_dict[link]),

                    ↪ discretize_relevance_simple(self.rel_parents_avg_dict[link])]
                    ↪ + self.categories_relevances[link]
        else:
            super().to_feat(link)

    def n_feat(self):
        return 3 + len(self.categories)
```

A.5. Implementando la frontera

```
import heapq
import itertools
import random


class UpdatablePQ:
    def __init__(self):
        """
        It initializes a PriorityQueue, that uses heapq. It is
```

```
        updatable, and randomized. To update a value, it only needs
        to call the method add_task with a new priority. To get a
        random element, it only needs to call the method pop_random.
        """
        # PriorityQueue, in form of a sorted list.
        self.pq = []
        # Dict of tasks: entries.
        self.entry_finder = {}
        # Placeholder of removed elements.
        self.REMOVED = '<removed-task>'
        # Counter to decide what to choose in case of ties.
        self.counter = itertools.count()
        # Number of elements added
        self.n_elements = 0

    def add(self, task, priority=0):
        'Add a new task or update the priority of an existing task'
        if task in self.entry_finder:
            self.remove(task)
        count = next(self.counter)
        entry = [priority, count, task]
        self.entry_finder[task] = entry
        heapq.heappush(self.pq, entry)
        self.n_elements += 1
        #print("Adding state: {}, action: {}, score: {}".format(task[0],
        ↪  task[1], priority))

    def remove(self, task):
        'Mark an existing task as REMOVED.  Raise KeyError if not found.'
        entry = self.entry_finder.pop(task)
        entry[-1] = self.REMOVED
        self.n_elements -= 1

    def pop(self):
        'Remove and return the lowest priority task. Return None if
        ↪  empty.'
        while self.pq:
            priority, count, task = heapq.heappop(self.pq)
            if task is not self.REMOVED:
                del self.entry_finder[task]
                self.n_elements -= 1
                return task
        return None

    def sample(self):
        'Samples an element without deleting it from the queue'
```

```
        while self.pq:
            priority, count, task = heapq.heappop(self.pq)
            if task is not self.REMOVED:
                del self.entry_finder[task]
                self.n_elements -= 1
                self.add(task, priority)
                return task
        raise KeyError('pop from an empty priority queue')

    def pop_random(self):
        'Pop a random element from the heap.'
        random_task = random.choice(list(self.entry_finder.keys()))
        self.remove(random_task)
        return random_task

    def sample_random(self):
        'Samples a random element without deleting it from the queue'
        random_task = random.choice(list(self.entry_finder.keys()))
        return random_task

    def reset(self):
        'Reset entirely the heap.'
        self.pq = []
        self.entry_finder = {}
        self.counter = itertools.count()
        self.n_elements = 0

    def is_empty(self):
        return self.n_elements <= 0

    def clean_removed_elements(self):
        new_pq = []
        new_entry_finder = {}
        n_elements = 0

        while self.pq:
            priority, count, task = heapq.heappop(self.pq)
            if task is not self.REMOVED:
                # Adds it to the new PQ
                entry = [priority, count, task]
                heapq.heappush(new_pq, entry)
                new_entry_finder[task] = entry
                n_elements += 1

        self.pq = new_pq
        self.entry_finder = new_entry_finder
```

```
        self.n_elements = n_elements

    def is_queued(self, task):
        return task in self.entry_finder

    def n_queued_elements(self):
        return self.n_elements
```

A.6. Implementando el crawler básico

```
import abc

from src.crawler.util.link import Link
from src.crawler.util.page import Page
from src.crawler.util.util import get_substring_positions,
↪  get_surrounding_texts


class AbstractDumpFetcher:
    def __init__(self, dump):
        self.dump = dump

    def _fetch_idx(self, idx):
        return self.dump.get_page(idx)

    def _fetch_url(self, url):
        """
        It fetches a document from a dump.

        :param url: url of the page.
        """
        idx = self.dump.title2idx(url)
        return self.dump.get_page(idx)

    @abc.abstractmethod
    def fetch(self, **kwargs):
        pass

    @abc.abstractmethod
    def process_links(self, idx):
        pass


class SimpleDumpFetcher(AbstractDumpFetcher):
```

```
    def fetch(self, idx):
        return self._fetch_idx(idx)

    def process_links(self, idx):
        if idx in self.dump.links.outlinks:
            return self.dump.links.outlinks[idx]
        else:
            return []


class DumpFetcher(AbstractDumpFetcher):

    def __init__(self, dump, vectorizer):
        super().__init__(dump)
        self.vectorizer = vectorizer
        self.links = {}
        self.pages = {}

    def reset(self):
        self.links = {}
        self.pages = {}

    def fetch(self, idx, parents=None):
        if idx not in self.pages:
            url = self.dump.idx2title(idx)
            html = self.dump.get_html(idx)
            preprocessed_html = self.dump.get_preprocessed_html(idx)
            vec = self.vectorizer.text2vec(preprocessed_html)
            self.pages[idx] = Page(idx, url, html, vec, parents)
        return self.pages[idx]

    def process_links(self, idx):
        idx_to_list = self.dump.get_outlinks_from_idx(idx)
        links = []
        page_from = self.fetch(idx)
        for idx_to in idx_to_list:
            # Ignore autolinks
            if idx_to != idx:
                if idx_to not in self.links:
                    self.links[idx_to] = self.get_link(page_from, idx_to)
                links.append(self.links[idx_to])
        return links

    def fetch_link(self, idx_to):
        if idx_to in self.links:
```

```
            return self.links[idx_to]
        return None

    def get_link(self, page_from, idx_to):
        html = page_from.html
        url = self.dump.idx2title(idx_to)
        positions = get_substring_positions(html, "[[" + url)
        surrounding_texts = get_surrounding_texts(html,
                                                  url,
                                                  positions,
                                                  distance=150)
        if len(surrounding_texts) > 0:
            text = "\n".join(surrounding_texts)
            punctuation = r'!"#$%&\'()*+ [],-./:;<=>?@\\^_`{|}~'
            text = text.translate(str.maketrans(punctuation, '
            ↪  '*len(punctuation)))
        else:
            text = url
        vec = self.vectorizer.text2vec(text)
        return Link(idx_to, url, vec, page_from)

from src.crawler.fetcher import DumpFetcher
from src.crawler.util.pq import UpdatablePQ
from src.dump.sim import cosine_sim


class DumpFocusedCrawler:

    def __init__(self, dump, vectorizer, topic):
        self.dump = dump
        self.frontier = UpdatablePQ()
        self.fetcher = DumpFetcher(dump, vectorizer)
        self.vectorizer = vectorizer
        self.topic = topic
        self.topic_vec = self.vectorizer.get_word_vec(topic)
        self.queued_pages = []
        self.history = []

    def _process_page(self, page):
        outlinks = self.fetcher.process_links(page.idx)
        for link in outlinks:
            if link.url not in self.queued_pages:
                q = cosine_sim(link.vec, self.topic_vec)
                self.frontier.add(link.idx_to, -q)
                self.queued_pages.append(link.url)
```

```
    def crawl(self, seed, limit=500, verbose=False):
        # Initialize seed
        for idx in seed:
            page = self.fetcher.fetch(idx)
            self._process_page(page)

        # Start the crawl
        time_steps = 0
        n_relevant_pages = 0
        visited_pages = []
        while time_steps < limit and self.frontier.n_elements > 0:
            idx = self.frontier.pop()
            page = self.fetcher.fetch(idx)
            self._process_page(page)
            tf = self.vectorizer.tf(page.html, self.topic)
            if page.idx not in visited_pages:
                visited_pages.append(page.idx)
                if tf > 0:
                    n_relevant_pages += 1
                    if verbose:
                        print("{} relevant pages discovered in {}
                        ↪  steps".format(n_relevant_pages, time_steps))
            self.history.append(n_relevant_pages)
            time_steps += 1
```

A.7. Definiendo las políticas de RL

```
from src.crawler.util.pq import UpdatablePQ
import numpy as np

from abc import ABC, abstractmethod
import rl.policy

class Policy(ABC):
    @abstractmethod
    def select_action(self):
        pass


class FrontierEpsGreedyPolicy(Policy):
    """
    Epsilon-greedy policy that uses an UpdatablePQ as
    frontier.
```

```
    """
    def __init__(self,
                 frontier,
                 eps=0.1):
        """
        Initializes the policy.

        :param frontier: UpdatablePQ, frontier.
        :param eps: epsilon.
        """
        self.frontier = frontier
        self.eps = eps

    def select_action(self):
        """
        Selects an action, and deletes it from the frontier.

        :return: action selected.
        """
        if np.random.uniform() < self.eps:
            return self.frontier.pop_random()
        else:
            return self.frontier.pop()

    def select_action_from_values(self, q_values):
        """
        Selects an action, but instead of using the frontier,
        it uses a list of q_values passed by argument.
        :param q_values: (np.ndarray) values to make the decission.
        :return: action chosen.
        """
        assert q_values.ndim == 1
        nb_actions = q_values.shape[0]

        if np.random.uniform() < self.eps:
            action = np.random.randint(0, nb_actions)
        else:
            action = np.argmax(q_values)
        return action

    def sample_action(self):
        """
        It selects an action, but without deleting it effectively
        from the frontier.

        :return: action sampled.
```

```
        """
        if np.random.uniform() < self.eps:
            return self.frontier.sample_random()
        else:
            return self.frontier.sample()


class KerasFrontierEpsGreedyPolicy(rl.policy.Policy):
    """
        Epsilon-greedy policy that uses an UpdatablePQ as
        frontier.

        """

    def __init__(self,
                 frontier,
                 eps=0.1):
        """
        Initializes the policy.

        :param frontier: UpdatablePQ, frontier.
        :param eps: epsilon.
        """
        self.frontier = frontier
        self.eps = eps
        self.last_q_values = None

    def select_action(self, q_values):
        """
        Selects an action, and deletes it from the frontier.

        :return: action selected.
        """
        assert q_values.ndim == 1
        self.last_q_values = q_values

        if np.random.uniform() < self.eps:
            action = self.frontier.pop_random()
        else:
            action = self.frontier.pop()
        return action

    def q_estimated(self, idx):
        return self.last_q_values[idx]
```

```
class PageLinkPQ(UpdatablePQ, Policy):
    def __init__(self, epsilon):
        """
        This PQ expects tasks in form (score, page, link).
        """
        super().__init__()
        self.eps = epsilon
        self.last_q_values = []
        # Nested dict: page -> {link -> score, link -> score...}
        self.page_entries = {}
        # Entries expected to be in form (score, page, link)
        self.PAGE_IDX = 0
        self.LINK_IDX = 1

    def add(self, task, priority=0):
        self.page_entries[task[self.PAGE_IDX]] = task[self.LINK_IDX]
        super().add(task, priority)

    def get_page_entries(self, page):
        return self.page_entries[page]

    def pq_asynchrounous_update(self, state, actions):
        for action in actions:
            q_value = self.last_q_values[action]
            self.add((state, action), -q_value)

    def select_action(self, q_values):
        if np.random.uniform() < self.eps:
            page, link = self.pop_random()
        else:
            page, link = self.pop()

        self.last_q_values = q_values
        return link


class VariableEpsPolicy(FrontierEpsGreedyPolicy):
    def __init__(self, frontier, max_eps, min_eps, nb_steps):
        """
        Initializes the policy.

        :param frontier: UpdatablePQ, frontier.
        :param eps: epsilon.
        """
        super().__init__(frontier, max_eps)
        self.value_max = max_eps
```

```
        self.value_min = min_eps
        self.nb_steps = nb_steps
        self.step = 0

    def get_current_value(self):
        """Return current epsilon value

        # Returns
            epsilon to use
        """

        # Linear annealed: f(x) = ax + b.
        a = -float(self.value_max - self.value_min) /
        ↪  float(self.nb_steps)
        b = float(self.value_max)
        value = max(self.value_min, a * float(self.step) + b)

        return value

    def select_action(self):
        """Choose an action to perform

        # Returns
            Action to take (int)
        """
        self.eps = self.get_current_value()
        self.step += 1
        return self.select_action()
```

A.8. Definiendo el crawler basado en Deep Q-Learning

```
import tensorflow as tf
import os
import gym
import numpy as np
import random

from keras.layers import Dense, Flatten, Dropout
from keras.models import Sequential
from keras.optimizers import Adam
from rl.agents import DQNAgent
from rl.memory import SequentialMemory
from gym.spaces import Box
from src.dump.sim import cosine_sim
```

```
def build_model(n_hidden_layers, n_hidden_neurons, n_inputs,
                n_outputs):
    """
    It constructs the NN model.

    :param n_hidden_layers: integer indicating the number of hidden
↪   layers of the NN.
    :param n_hidden_neurons: list in witch each i-th position indicates
↪   the number of neurons of the i-th hidden layer.
    :param n_inputs: number of input neurons.
    :param n_outputs: number of output neurons (n_actions).

    :return: tensorflow NN model
    """
    model = Sequential()

    model.add(Flatten(input_shape=(1, n_inputs)))

    counter = 0
    while counter < n_hidden_layers:
        model.add(
            Dense(n_hidden_neurons[counter], activation='tanh'))
        counter += 1

    model.add(Dropout(0.5))

    model.add(Dense(n_outputs,
                    activation='linear',
                    activity_regularizer=tf.keras.regularizers.l2(0.01)))

    model.compile(loss='mse', optimizer='adam', metrics=['mae'])
    model.summary()
    return model


class QNetwork:
    def __init__(self,
                 model,
                 policy,
                 test_policy,
                 nb_actions,
                 dqn_mode,
                 gamma=0.2,
                 weights_file=None):
```

```
        self.model = model
        self.policy = policy
        self.test_policy = test_policy
        memory = SequentialMemory(limit=50000, window_length=1)

        enable_double_dqn = False
        enable_dueling_network = False
        dueling_type = None
        if dqn_mode == "t":
            enable_double_dqn = True
        elif dqn_mode == "d":
            enable_double_dqn = True
            enable_dueling_network = True
            dueling_type = "avg"

        self.dqn = DQNAgent(model=model,
                            nb_actions=nb_actions,
                            memory=memory,
                            nb_steps_warmup=100,
                            enable_double_dqn=enable_double_dqn,

                            ↪  enable_dueling_network=enable_dueling_network,
                            dueling_type=dueling_type,
                            gamma=gamma,
                            target_model_update=1e-2,
                            policy=policy,
                            test_policy=test_policy)

        if weights_file is not None:
            self.dqn.load_weights(weights_file)
        self.dqn.compile(Adam(lr=1e-3), metrics=['mae'])

    def train(self,
              env,
              nb_steps=50000,
              visualize=True,
              verbose=2,
              callbacks=None):
        return self.dqn.fit(env,
                            nb_steps,
                            visualize=visualize,
                            verbose=verbose,
                            callbacks=callbacks)

    def test(self,
             env,
```

```
                nb_episodes=1,
                visualize=False,
                verbose=2,
                callbacks=None,
                nb_max_episode_steps=None):
            return self.dqn.test(env,
                                 nb_episodes=nb_episodes,
                                 visualize=visualize,
                                 verbose=verbose,
                                 callbacks=callbacks,
                                 nb_max_episode_steps=nb_max_episode_steps)

        def save(self, filename):
            self.dqn.save_weights(filename, overwrite=True)

        def load(self, filename):
            self.dqn.load_weights(filename)

        def compute_q_values(self, state):
            return self.dqn.compute_q_values(state)

        def forward(self, state):
            return self.dqn.forward(state)


class OptimizedCrawlerEnv(gym.Env):

        def to_feat(self, page, link):
            page_feat = self.page_featurizer.to_feat(page)
            link_feat = self.link_featurizer.to_feat(link)
            return np.array(page_feat + link_feat)

        def _initialize_frontier_with_seed(self, seed):
            for idx in seed:
                page = self.fetcher.fetch(idx)
                outlinks = self.fetcher.process_links(idx)
                for link in outlinks:
                    feat = self.page_featurizer.to_feat(page)
                    q_e = self.qnetwork.compute_q_values([feat])[link.idx_to]
                    self.frontier.add((idx, link.idx_to), -q_e)

        def __init__(self,
                     vectorizer,
                     page_featurizer,
                     link_featurizer,
                     frontier,
```

```
            fetcher,
            policy,
            qnetwork,
            seed,
            topic,
            limit,
            results_path,
            n_pages,
            verbose=True,
            visualize=False):

        self.vectorizer = vectorizer
        self.page_featurizer = page_featurizer
        self.link_featurizer = link_featurizer
        self.frontier = frontier
        self.fetcher = fetcher
        self.policy = policy
        self.qnetwork = qnetwork
        self.topic = topic
        self.topic_vector = self.vectorizer.get_word_vec(self.topic)
        self.results_path = results_path
        self.seed = seed
        self.limit = limit
        self.verbose = verbose
        self.visualize = visualize

        # We initialize the check lists and relevance history
        self.relevance_history = [0]
        self.visited_links = []
        self.rel_visited_pages = []
        self.n_visited_pages = 0
        self.time_step = 0
        self.inlinks_to_update = {}

        self.observation_space = Box(low=0,
                                     high=9,
                                     shape=[page_featurizer.n_feat()],
                                     dtype=int)
        # 2 ints, one for the state idx, another for action idx
        self.action_space = Box(low=0,
                                high=n_pages,
                                shape=[2],
                                dtype=int)

        self.reset()
```

```
    def reset(self):
        # We empty the frontier
        self.frontier.reset()
        # And initialize it with the seed
        self._initialize_frontier_with_seed(self.seed)

        # We initialize the check lists and relevance history
        self.relevance_history = [0]
        self.visited_links = []
        self.rel_visited_pages = []
        self.n_visited_pages = 0
        self.time_step = 0
        self.inlinks_to_update = {}

        idx = random.choice(self.seed)
        state = self.fetcher.fetch(idx)
        state_feat = self.page_featurizer.to_feat(state)
        return state_feat

    def reward(self, page):
        """
        It defines the reward of a page, p, as:

        If the page contains the word of the topic, it is relevant
        to the topic, and receives a reward of 30.
        If the page doesnt contain the word, but it has a word2vec
        similarity greater than 0.5, it receives a reward of 30. If it
        is greater than 0.4, the reward is 20.
        Otherwise, the reward is -1.
        :return: reward of current page, and topic.
        """
        if self.page_featurizer.is_relevant(page):
            return 30
        page_vec = self.vectorizer.get_doc_vec(page.idx)
        word2vec_sim = cosine_sim(page_vec, self.topic_vector)
        if word2vec_sim > 0.6:
            return 30
        elif word2vec_sim > 0.5:
            return 20
        else:
            return -1

    def step(self, action):
        if action is None:
            return (None,
                    -1,
```

```
                True,
                {})
        self.time_step += 1
        state_idx = action[0]
        action_idx = action[1]
        state = self.fetcher.fetch(state_idx)
        state_feat = self.page_featurizer.to_feat(state)
        if action_idx in self.visited_links:
            return (state_feat,
                    self.reward(state),
                    self.done(),
                    {})

        self.visited_links.append(action_idx)
        # Si es la primera vez que visitamos una pagina, hay que
        ↪  inicializarla
        # con los parents que hemos ido guardando en inlinks_to_update.
        parents = None
        if action_idx in self.inlinks_to_update:
            parents = self.inlinks_to_update[action_idx]
        next_state = self.fetcher.fetch(action_idx, parents)
        outlinks = self.fetcher.process_links(next_state.idx)

        selectable_outlinks = []
        for outlink in outlinks:
            # Si la página ya ha sido visitada, podemos añadir el parent
            # directamente a traves del metodo del page_featurizer.
            if outlink.idx_to in self.visited_links:
                page = self.fetcher.fetch(outlink.idx_to)
                self.page_featurizer.add_parent(page, next_state)
            else:
                self.link_featurizer.add_parent(outlink, next_state)
                selectable_outlinks.append(outlink)
                # Si no hemos visitado la pagina aun, almacenamos estos
                # enlaces en la variable inlinks_to_update. Cuando
                # hagamos el fetch por primera vez de esa página, la
                # inicializaremos con los parents almacenados
                ↪  previamente.
                if outlink.idx_to not in self.inlinks_to_update:
                    self.inlinks_to_update[outlink.idx_to] = []
                self.inlinks_to_update[outlink.idx_to].append(next_state)

        #action_feat =
        ↪  self.link_featurizer.to_feat(self.fetcher.fetch_link(action_idx))
        if self.visualize:
            print("State: {}".format(state_feat))
```

```
            #print("Action: {}".format(action_feat))

        if self.page_featurizer.is_relevant(next_state):
            if next_state.idx not in self.rel_visited_pages:
                self.rel_visited_pages.append(next_state.idx)
                self.relevance_history.append(self.relevance_history[-1]
                ↪  + 1)
                if self.verbose:
                    print("\n{} relevantes descubiertas en {}
                    ↪  pasos.".format(self.relevance_history[-1],
                    ↪  self.n_visited_pages))
            else:
                self.relevance_history.append(self.relevance_history[-1])
        else:
            self.relevance_history.append(self.relevance_history[-1])

        # Update the frontier
        self._asynchronous_update(next_state, selectable_outlinks)
        self.n_visited_pages += 1
        next_state_feat = self.page_featurizer.to_feat(next_state)

        return (next_state_feat,
                self.reward(next_state),
                self.done(),
                {})

    def done(self):
        return self.n_visited_pages >= self.limit or
        ↪  self.frontier.is_empty()

    def _asynchronous_update(self, next_state, next_actions):
        next_state_feat = self.page_featurizer.to_feat(next_state)
        for next_action in next_actions:
            # q value estimated
            q_e =
            ↪  self.qnetwork.compute_q_values([next_state_feat])[next_action.idx_t
            self.frontier.add((next_state.idx, next_action.idx_to), -q_e)
```

A.9. Definiendo el crawler basado en Aproximación Lineal de Funciones

```
import numpy as np
```

```
from src.crawler.util.pq import UpdatablePQ
from src.dump.sim import cosine_sim


class LinearFunctionApproximator:
    """
    It implements a Linear Function Approximation. It uses
    the gradient descent method to minimize the error function.

    This is used for a problem of Reinforcement Learning, where
    we want to estimate the value of the function Q. We define the
    estimation as the inner product of the transpose of the weights
    and the features of state-action:

    q_e = wT * x(s,a)

    Then, we set the update rule of the weights in the direction of
    the negative gradient, where the error is minimized:

    w <- w + alpha * delta * x(s,a),

    where alpha is the learning rate, delta is the temporal-difference
    error, and x(s,a) is the gradient of q_e, with respect to w. We
    define delta as:

    delta = reward + gamma * q_e(s_t+1, a_t+1, w) - q_e(s_t, a_t, w),
    where gamma is a discount factor 0 <= gamma < 1.
    """
    def __init__(self,
                 n_feat,
                 learning_rate=0.1,
                 discount_factor=0.2):
        self.n_feat = n_feat
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor

        self.weights = np.random.rand(n_feat)

    def reset(self):
        self.weights = np.random.rand(self.n_feat)

    def q_estimated(self, features):
        assert features.shape == self.weights.shape
        w_t = self.weights.transpose()
        return np.inner(w_t, features)
```

```
    def update(self,
               reward,
               features,
               next_features):
        # temporal difference error
        tde = self.discount_factor*self.q_estimated(next_features) -
        ↪  self.q_estimated(features)
        delta = reward + tde
        self.weights += self.learning_rate*delta*features

    def update_relevant_state(self,
                              reward,
                              features):
        delta = reward - self.q_estimated(features)
        self.weights += self.learning_rate * delta * features

    def save(self, path):
        np.savetxt(path, self.weights)

    def load(self, path):
        self.weights = np.loadtxt(path)


class FrontierMode:
    SYNCHRONOUS = 1,
    ASYNCHRONOUS = 2


class LFACrawler:
    """
        Focused Crawler that uses Reinforcement Learning,
        by Linear Function Approximator.
    """

    def to_feat(self, page, link):
        page_feat = self.page_featurizer.to_feat(page)
        link_feat = self.link_featurizer.to_feat(link)
        return np.array(page_feat + link_feat)

    def __init__(self,
                 vectorizer,
                 page_featurizer,
                 link_featurizer,
                 lf_approximator,
                 frontier,
                 fetcher,
```

```
                policy,
                topic,
                results_path,
                frontier_mode=FrontierMode.ASYNCHRONOUS):
        """

        It initializes a blind crawler, based on RL. This crawler
        starts from the seedUrls, and crawls a page at a time,
        using Q-Learning to maintain the score of the pages, based
        on similarities with a topic.

        :param seed: (list of strings), array of starting titles.
        :param topic: np.array, containing the topic vector.
        :param n_pages: (int), number of maximum pages to be queued in
        the frontier.
        :param fetcher: (Fetcher), object that implements the fetch
        and process_urls methods.
        :param tokens_blacklist: (list of strings), list of tokens to
        exclude in HTMls.
        """
        self.vectorizer = vectorizer
        self.page_featurizer = page_featurizer
        self.link_featurizer = link_featurizer
        self.lf_approximator = lf_approximator
        self.frontier = frontier
        self.fetcher = fetcher
        self.policy = policy
        self.topic = topic
        self.topic_vector = self.vectorizer.get_word_vec(self.topic)
        self.results_path = results_path
        self.frontier_mode = frontier_mode
        self.time_steps = 0

    def _initialize_frontier_with_seed(self, seed):
        for idx in seed:
            page = self.fetcher.fetch(idx)
            outlinks = self.fetcher.process_links(idx)
            for link in outlinks:
                feat = self.to_feat(page, link)
                q_e = self.lf_approximator.q_estimated(feat)
                self.frontier.add((page, link), -q_e)

    def reward(self, page):
        """
        It defines the reward of a page, p, as:
```

```
        If the page contains the word of the topic, it is relevant
        to the topic, and receives a reward of 30.
        If the page doesnt contain the word, but it has a word2vec
        similarity greater than 0.5, it receives a reward of 30. If it
        is greater than 0.4, the reward is 20.
        Otherwise, the reward is -1.
        :return: reward of current page, and topic.
        """
        if self.page_featurizer.is_relevant(page):
            return 30
        page_vec = self.vectorizer.get_doc_vec(page.idx)
        word2vec_sim = cosine_sim(page_vec, self.topic_vector)
        if word2vec_sim > 0.6:
            return 30
        elif word2vec_sim > 0.5:
            return 20
        else:
            return -1

    def crawl(self, seed, limit=1000, verbose=False, visualize=False):
        # We empty the frontier
        self.frontier.reset()
        self.policy.frontier = self.frontier
        # And initialize it with the seed
        self._initialize_frontier_with_seed(seed)

        # We initialize the check lists and relevance history
        relevance_history = [0]
        visited_links = []
        rel_visited_pages = []
        n_visited_pages = 0
        self.time_steps = 0
        inlinks_to_update = {}

        while n_visited_pages < limit and not self.frontier.is_empty():
            self.time_steps += 1
            (state, action) = self.policy.select_action()
            if action.idx_to in visited_links:
                continue
            visited_links.append(action.idx_to)
            # Si es la primera vez que visitamos una pagina, hay que
            ↪ inicializarla
            # con los parents que hemos ido guardando en
            ↪ inlinks_to_update.
            parents = None
            if action.idx_to in inlinks_to_update:
```

```
            parents = inlinks_to_update[action.idx_to]
        next_state = self.fetcher.fetch(action.idx_to, parents)
        reward = self.reward(next_state)
        outlinks = self.fetcher.process_links(next_state.idx)

        selectable_outlinks = []
        for outlink in outlinks:
            # Si la página ya ha sido visitada, podemos añadir el
            ↪ parent
            # directamente a traves del metodo del page_featurizer.
            if outlink.idx_to in visited_links:
                page = self.fetcher.fetch(outlink.idx_to)
                self.page_featurizer.add_parent(page, next_state)
            else:
                self.link_featurizer.add_parent(outlink, next_state)
                selectable_outlinks.append(outlink)
                # Si no hemos visitado la pagina aun, almacenamos
                ↪ estos
                # enlaces en la variable inlinks_to_update. Cuando
                # hagamos el fetch por primera vez de esa página, la
                # inicializaremos con los parents almacenados
                ↪ previamente.
                if outlink.idx_to not in inlinks_to_update:
                    inlinks_to_update[outlink.idx_to] = []
                inlinks_to_update[outlink.idx_to].append(next_state)

        #features = self.to_feat(state, action)
        state_feat = self.page_featurizer.to_feat(state)
        action_feat = self.link_featurizer.to_feat(action)
        if visualize:
            print("State: {}".format(state_feat))
            print("Action: {}".format(action_feat))
        features = np.array(state_feat + action_feat)

        if len(selectable_outlinks) > 0:
            next_action = self._select_next_action(next_state,
            ↪ selectable_outlinks)
            next_features = self.to_feat(next_state, next_action)
            self.lf_approximator.update(reward,
                                        features,
                                        next_features)

        if self.page_featurizer.is_relevant(next_state):
            if next_state.idx not in rel_visited_pages:
                rel_visited_pages.append(next_state.idx)
                relevance_history.append(relevance_history[-1] + 1)
```

```
                    if verbose:
                        print("{} relevantes descubiertas en {}
                        ↪  pasos.".format(relevance_history[-1],
                        ↪  n_visited_pages))
                else:
                    relevance_history.append(relevance_history[-1])
            else:
                relevance_history.append(relevance_history[-1])

            # Update the frontier
            if self.frontier_mode == FrontierMode.SYNCHRONOUS:
                # Do synchronous update
                self._synchronous_update(next_state, selectable_outlinks)
            else:
                # Do asynchronous update
                self._asynchronous_update(next_state,
                ↪  selectable_outlinks)

            n_visited_pages += 1
        return relevance_history

    def _select_next_action(self, next_state, next_actions):
        q_values = []

        next_state_feat = self.page_featurizer.to_feat(next_state)
        for i in range(0, len(next_actions)):
            next_action_feat =
            ↪  self.link_featurizer.to_feat(next_actions[i])
            feat = np.array(next_state_feat + next_action_feat)
            q_values.append(self.lf_approximator.q_estimated(feat))

        return
        ↪  next_actions[self.policy.select_action_from_values(np.array(q_values))]

    def _synchronous_update(self, next_state, next_actions):
        self._asynchronous_update(next_state, next_actions)
        new_frontier = UpdatablePQ()
        while not self.frontier.is_empty():
            state, action = self.frontier.pop()
            feat = self.to_feat(state, action)
            q_e = self.lf_approximator.q_estimated(feat)
            new_frontier.add((state, action), -q_e)
        self.frontier = new_frontier
        self.policy.frontier = new_frontier

    def _asynchronous_update(self, next_state, next_actions):
```

```
        next_state_feat = self.page_featurizer.to_feat(next_state)
        for next_action in next_actions:
            next_action_feat = self.link_featurizer.to_feat(next_action)
            feat = np.array(next_state_feat + next_action_feat)
            # q value estimated
            q_e = self.lf_approximator.q_estimated(feat)
            self.frontier.add((next_state, next_action), -q_e)
```