UNIVERSIDAD ALFONSO X EL SABIO

FACULTAD BUSINESS&TECH

ESTUDIO DEL PROBLEMA DEL BANDIDO MULTIBRAZO Y SUS APLICACIONES

Autor: Eduardo López Pérez

Tutor: Javier Jiménez Raboso

TRABAJO FIN DE MÁSTER

Máster Universitario en Análisis de Datos Masivos (Big Data)
para los Negocios

22 de Agosto 2025

Resumen

En este documento se aborda el problema del bandido multibrazo, haciendo énfasis en los principales algoritmos utilizados clásicamente para resolverlo y en las posibles aplicaciones en entornos reales. Los algoritmos tratados son ϵ-Greedy, Upper Confidence Bound, Thompson Sampling, Linear Upper Confidence Bound y Linear Thompson Sampling, y son aplicados al caso práctico de una plataforma de comercio electrónico, las cuales han tomado mayor relevancia con la digitalización de nuestra sociedad. Se analiza la precisión de las recomendaciones realizadas por los algoritmos previamente mencionados en función de si sus usuarios hacen clic o no en los productos sugeridos por el algoritmo, se destaca el aumento de la precisión de las recomendaciones cuando se hace uso del contexto del usuario para realizar las sugerencias y se habla de la posibilidad de utilizar modelos basados en aprendizaje profundo para mejorar aún más las recomendaciones del sistema, lo cual se deja para futuros trabajos.

Palabras clave: bandido multibrazo, ϵ-Greedy, Upper Confidence Bound, Thompson Sampling, Linear Upper Confidence Bound, Linear Thompson Sampling, recompensa, regret, comercio electrónico.

Abstract

This document addresses the multi-armed bandit problem, focusing on the main algorithms traditionally used to solve it and their possible applications in real-world environments. The algorithms discussed are ϵ-Greedy, Upper Confidence Bound, Thompson Sampling, Linear Upper Confidence Bound, and Linear Thompson Sampling, and they are applied to the practical case of an e-commerce platform, which have become increasingly relevant with the digitization of our society. The accuracy of the recommendations made by the aforementioned algorithms is analysed based on whether or not users click on the products suggested by the algorithm. The increase in the accuracy of the recommendations when the user's context is considered to make suggestions is highlighted, and the possibility of using deep learning-based models to further improve the system's recommendations is discussed, which is left for future work.

Keywords: multi-armed bandit, ϵ-Greedy, Upper Confidence Bound, Thompson Sampling, Linear Upper Confidence Bound, Linear Thompson Sampling, reward, regret, e-commerce.

Índice general

Capítulo 1

Introducción

En este trabajo se pretende abordar el problema del bandido multibrazo y su extensión a los bandidos multibrazo contextuales. Este problema es uno de los ejemplos más comunes dentro del aprendizaje automático por refuerzo y cuenta con numerosas aplicaciones en el mundo real. De estas aplicaciones, las más extendidas son: el diseño de sistemas de recomendación (para plataformas de "streaming", comercio electrónico o redes sociales), ensayos clínicos, enrutamiento dinámico y el diseño de pruebas A/B.

En el caso de los sistemas de recomendación, el auge de la cantidad de servicios que hoy en día son realizados por la mayoría de las personas a través de internet ha hecho que estos sistemas tomen una enorme importancia a la hora de maximizar los beneficios para múltiples empresas y, por tanto, la aplicación de los algoritmos para la resolución del problema del bandido multibrazo y el aprendizaje por refuerzo en general han cobrado un mayor grado de relevancia recientemente.

La mayoría de estos servicios ofrecidos a través de internet generan una enorme cantidad de datos sobre como los usuarios interaccionan con una página web, lo cual provoca que este tipo de aplicaciones del problema del bandido multibrazo sean realizadas en entornos de Big Data, conllevando ciertos retos, aunque también ciertas ventajas, ya que los algoritmos de aprendizaje por refuerzo nunca paran de aprender y a medida que se van generando más datos (siempre y cuando estos datos sean de buena calidad) estos sistemas seguirán mejorando más y más cada día.

En lo que respecta a este trabajo, se pretende realizar una introducción teórica al problema del bandido multibrazo y su extensión al bandido multibrazo contextual, y también a los principales algoritmos usados para resolver estos problemas, mostrando además como se utilizarían estos algoritmos para el sistema de recomendación de una plataforma de comercio electrónico.

El documento estará estructurado de la siguiente forma:

- En una primera sección se expondrán la motivación y los objetivos del trabajo.
- A continuación, se hará una revisión de la literatura, en la que se explicará todo el contexto teórico de estos problemas y los principales algoritmos usados para resolverlos.
- Después, se tendrá una sección destinada a la metodología seguida para la implementación de la parte práctica del trabajo, en la que se aplicarán los algoritmos presentados

en el estado del arte a un sistema de recomendación en una web de comercio electrónico.

- Tras la metodología, habrá una sección de resultados y discusión, en la que se explicarán los resultados obtenidos de la aplicación de estos algoritmos al sistema de recomendación.
- Y para finalizar, habrá un apartado de conclusiones en el que se expondrán las conclusiones generales del trabajo.

Capítulo 2

Motivación y objetivos

La razón de la elección del problema del bandido multibrazo como tema de este trabajo fin de master (TFM) es el de realizar un estudio de un campo del aprendizaje automático (el aprendizaje por refuerzo) que normalmente no es tratado con mucho detalle cuando se empieza a estudiar sobre el aprendizaje automático y la inteligencia artificial, ya que el aprendizaje no supervisado, y sobre todo el supervisado, suelen acaparar más el foco debido su mayor número de aplicaciones en diferentes ámbitos.

El principal objetivo de este trabajo consiste en mostrar la utilidad del marco teórico de los bandidos multibrazo para la resolución de problemas del mundo real, como el diseño de un sistema de recomendación para una plataforma de comercio electrónico parecido a lo realizado en Rohde et al. (2018), que se verá en la parte práctica de este trabajo.

Los problemas del bandido multibrazo y el bandido multibrazo contextual son problemas que suelen ser estudiados desde un punto de vista académico como un primer ejemplo de aprendizaje por refuerzo, pero no suele llegar a estudiarse nunca su traslación a entornos del mundo real. Estos problemas, a pesar de su aparente simplicidad, dan lugar a un campo de estudio amplio y complejo, y cuya aplicación en entornos empresariales puede tener un alto valor añadido.

Capítulo 3

Estado del arte

En esta sección se dará una explicación de en qué consiste tanto el problema del bandido multibrazo, como el problema del bandido multibrazo contextual, y se explicarán en detalle los principales algoritmos que han sido usados clásicamente en la literatura para abordar estos problemas.

El problema del bandido multibrazo se da cuando se es presentado con n opciones (o acciones) diferentes reiteradamente y se pretende maximizar la recompensa obtenida por las opciones escogidas a lo largo de un periodo de tiempo estipulado.

Estas recompensas suelen consistir en valores numéricos provenientes de una distribución de probabilidad estacionaria diferente según la acción escogida.

El ejemplo más comúnmente usado para explicar este problema es el de una serie de máquinas tragaperras. Cada máquina tragaperras se traduce en un brazo diferente del bandido multibrazo y el objetivo del problema será decidir qué máquina escoger para maximizar la cantidad de dinero obtenida de los premios de las máquinas. Para maximizar estos premios, se buscará escoger aquellas máquinas tragaperras que tengan una mayor probabilidad de dar un premio.

Cada una de las acciones de un bandido multibrazo tiene una recompensa media esperada, la cual si fuera conocida para cada uno de los brazos convertiría este problema en un problema trivial, ya que siempre se escogería la acción que mejor recompensa media tuviera. Sin embargo, estas recompensas medias esperadas no son conocidas en la mayoría de los casos, o por lo menos no de forma exacta. Por este motivo, para poder realizar una estimación de cuál es la recompensa media de cada una de las acciones se necesitará probar a escoger cada una de estas y evaluar las recompensas obtenidas de forma que se pueda hacer una estimación de la media de cada opción posible.

El obtener un buen balance entre probar las diferentes acciones posibles para mejorar las estimaciones de las recompensas medias de cada una de estas acciones y escoger la acción con la mejor recompensa media según las estimaciones disponibles para maximizar la recompensa obtenida es algo fundamental en este tipo de problemas y es lo que se conoce como el dilema o conflicto de la exploración frente a la explotación.

Este fenómeno de exploración vs. explotación es un concepto ampliamente estudiado en el ámbito del aprendizaje por refuerzo y no existe una solución única que permita resolver este problema para cualquier bandido multibrazo.

Por este motivo, existen un gran número de algoritmos que son usados hoy en día para resolver esta clase de problemas, cada uno de ellos balanceando de una forma distinta este equilibrio entre exploración y explotación.

De forma general, algoritmos con un mayor grado de explotación obtendrán mejores recompensas acumuladas en periodos cortos de tiempo, mientras que algoritmos con un mayor grado de exploración obtendrán mejores recompensas acumuladas en periodos más largos de tiempo, ya que estos últimos es más probable que encuentren acciones con mejores recompensas medias que las acciones que en un principio se creía que tenían las mejores recompensas medias de entre todas las acciones posibles.

Los algoritmos clásicos más utilizados para la resolución de los problemas de bandido multibrazo son el ϵ-greedy, Upper Confidence Bound (UCB) y Thompson Sampling (TS). A continuación, se explicará la matemática detrás de cada uno de estos algoritmos. Para información más detallada se pueden consultar Sutton y Barto (2018) para los dos primeros algoritmos y Russo et al. (2017) para Thompson Sampling.

Algoritmo ϵ-greedy: Este algoritmo o grupo de algoritmos consisten en escoger la acción que se prevé que va a dar la mejor recompensa con probabilidad $(1-\epsilon)$. Desde un punto de vista formal se tiene que:

Dada una acción a, la estimación del valor de dicha acción en un paso temporal t será:

$$Q_t(a) = \frac{r_1 + r_2 + ... + r_{n_t(a)}}{n_t(a)}$$

Donde $r_1, r_2, ..., r_{n_t(a)}$ son las recompensas de los $n_t(a)$ pasos temporales previos en los que la acción a fue escogida. Este método para estimar el valor de las acciones es conocido como el método del promedio muestral.

Cuando $n_t(a) = 0$ se asigna a $Q_t(a)$ un valor por defecto, y cuando $n_t(a) \to$ ínf, por la ley de los grandes números se tiene que $Q_t(a) = q(a)$, donde $q(a)$ es el valor real de la acción a.

Un algoritmo *greedy* escogerá para un tiempo t la acción:

$$a_t = argmax_a Q_t(a)$$

Donde a_t es la acción escogida en el tiempo t. En caso de que haya varias acciones que maximicen Q_t, la acción escogida será una aleatoria de entre todas las que lo maximicen.

El algoritmo ϵ-greedy escogerá esa acción con probabilidad $(1-\epsilon)$, siendo ϵ un valor fijo perteneciente al intervalo $[0, 1]$, y otra acción diferente escogida de forma completamente aleatoria con probabilidad ϵ.

Cuando $\epsilon = 0$, estaremos ante el caso de un algoritmo completamente *greedy*, y cuando $\epsilon = 1$ estaremos ante un algoritmo que escoge acciones de forma completamente aleatoria.

Algoritmo Upper Confidence Bound (UCB): Al contrario que para el caso de los métodos ϵ-greedy, la exploración de las acciones que a priori no son óptimas se hace siguiendo una estrategia diferente a la de escoger una acción aleatoria de entre todas las posibles. En el

caso del algoritmo UCB, la elección de la acción a explorar se hace en función del potencial que tienen cada una de las acciones de llegar a maximizar la recompensa, teniendo en cuenta tanto cuan cerca está la estimación de la media de cada acción de la acción cuya recompensa media es máxima, como la incertidumbre de dichas estimaciones.

La decisión de qué acción escoger se hace en este caso siguiendo la fórmula:

$$a_t = argmax_a \left[Q_t(a) + \alpha \sqrt{\frac{lnt}{n_t(a)}} \right]$$

Donde lnt denota el logaritmo neperiano de t y $\alpha > 0$ es un parámetro ajustable que controla el grado de exploración del algoritmo. Si $n_t(a) = 0$, se considera que a está maximizando la acción, evitando así divisiones por 0.

El término que se encuentra dentro de la raíz cuadrada sirve como una medida de la incertidumbre de la estimación del valor de la recompensa de la acción a.

Algoritmo Thompson Sampling (TS): Este algoritmo tiene un componente más probabilístico que los algoritmos anteriores, que se basan en estimar la acción a escoger de forma empírica directamente desde los resultados de las acciones escogidas en los pasos temporales anteriores.

En este caso se supone que las recompensas de las diferentes acciones a escoger siguen distribuciones de probabilidad definidas a priori y que van actualizándose cada vez que se escoge una acción y se obtiene una recompensa.

Más formalmente: dada una distribución de probabilidad a priori $p_0(a)$, para cada acción a, en cada paso del algoritmo de Thompson Sampling se muestrea aleatoriamente un $\hat{Q}_t(a)$ $p_t(a)$, siendo $p_t(a)$ la distribución de probabilidad a posteriori en el paso t de la acción a. Para ese tiempo t la acción escogida será:

$$a_t = argmax_a \hat{Q}_t(a)$$

La distribución de probabilidad a posteriori se obtiene a partir de $p_0(a)$, aplicando el teorema de Bayes de la siguiente forma:

$$p\left(a|r_1, ..., r_{n_t(a)}\right) = \frac{p\left(r_1, ..., r_{n_t(a)}|a\right) \cdot p_0(a)}{p\left(r_1, ..., r_{n_t(a)}\right)}$$

Donde a es la acción para la que se quiere estimar la distribución de probabilidad, $r_1, ..., r_{n_t(a)}$ son las recompensas de los $n_t(a)$ pasos temporales previos en los que dicha acción fue escogida y $p\left(a|r_1, ..., r_{n_t(a)}\right) = p_t(a)$.

Esta actualización de la distribución a posteriori se puede hacer en cada paso temporal mediante la fórmula recursiva:

$$p_{t+1}(a) \propto p(r_t|a) \cdot p_t(a)$$

Una extensión ampliamente conocida y estudiada del problema del bandido multibrazo es el conocido como bandido multibrazo contextual. En este problema, además de la información disponible en un problema de bandido multibrazo clásico existe también un contexto, el cual es conocido antes de la toma de cualquier decisión y proporciona información relevante sobre el entorno o el usuario, pudiendo ayudar a tomar una mejor decisión en lo que respecta a qué acción escoger.

Este contexto, que se denotará $x_t \in X$, suele venir dado por un vector de características. Los diferentes contextos disponibles son usados para ajustar modelos (un modelo diferente para cada acción disponible) para predecir la recompensa que se obtendrá en cada caso.

Clásicamente, los modelos usados para esta finalidad solían ser modelos lineales, aunque en la actualidad también se usan modelos no lineales más complejos basados en redes neuronales o árboles de decisión.

En este documento serán tratados los algoritmos de Linear Upper Confidence Bound (LinUCB) y Linear Thompson Sampling (LinTS), los cuales son extensiones de los algoritmos presentados previamente a los bandidos multibrazo contextuales mediante el uso de algoritmos lineales para modelar el contexto. Para más información sobre el primer algoritmo se puede consultar Zhou (2015), y para el segundo Russo et al. (2017).

Algoritmo Linear Upper Confidence Bound (LinUCB): Consiste en una extensión del algoritmo UCB mediante el uso de algoritmos lineales para modelar el contexto. De forma más rigurosa, este algoritmo se puede enunciar como:

Dado un contexto $x_{t,a} \in \mathbb{R}^d$ para cada acción a y cada tiempo t, la recompensa esperada se considerará que es una función lineal del contexto de la siguiente forma:

$$Q_t\left(a\right) = x_{t,a}^{\top}\theta^*$$

Donde $\theta^* \in \mathbb{R}^d$ es un vector de parámetros desconocido el cual permite minimizar la función de pérdida

$$R\left(T\right) = \sum_{t=1}^{T}\left(x_{t,a_t^*}^{\top}\theta^* - x_{t,a_t}^{\top}\theta^*\right)$$

Donde $a_t^* = argmax_{a\in A}x_{t,a}^{\top}\theta^*$.

Esta función de pérdida es conocida como el regret acumulado.

En cada paso del algoritmo se tiene una estimación de θ^* proveniente del modelo lineal. Dicha estimación se denotará $\hat{\theta}_t$ y la acción escogida en cada paso será:

$$a_t = argmax_a x_{t,a}^{\top}\hat{\theta}_{t-1} + \alpha \cdot \sqrt{x_{t,a}^{\top}A_{t-1}^{-1}x_{t,a}}$$

Donde $\alpha > 0$ es el mismo parámetro ajustable que controla el grado de exploración del algoritmo de UCB y la matriz A_{t-1} proviene del modelo lineal, el cual consiste en una regresión ridge (regularizada) de la siguiente forma:

$$\hat{\theta}_t = A_t^{-1} b_t$$

Donde

$$A_t = \lambda I + \sum_{s=1}^{t} x_{s,a_s} x_{s,a_s}^{\top} \in \mathbb{R}^{d \times d}$$

$$b_t = \sum_{s=1}^{t} r_s x_{s,a_s} \in \mathbb{R}$$

$$A_0 = \lambda I$$

con λ el parámetro entrenable de la regresión lineal, I la matriz identidad, r_s la recompensa para un tiempo s y x_{s,a_s} el contexto para la acción a_s en el tiempo s.

Algoritmo Linear Thompson Sampling (LinTS): Es una extensión del algoritmo de Thompson Sampling mediante el uso de modelos lineales para tener en cuenta el contexto.

Al igual que para el algoritmo anterior, la recompensa esperada viene dada por la ecuación:

$$Q_t(a) = x_{t,a}^{\top} \theta^*$$

Pero, en este caso, $\hat{\theta}_t$, la estimación del vector de parámetros θ^*, consiste en una muestra aleatoria de una distribución a posteriori de vectores de parámetros, $p_t(\theta)$, la cual se calcula a partir de una distribución a priori, $p_0(\theta)$, aplicando el teorema de Bayes de forma análoga al caso del algoritmo de Thompson Sampling, siempre y cuando esta distribución a priori sea conjugada de la a posteriori, es decir, pertenezcan a la misma familia. En caso de que no sea así, habría que recurrir a métodos numéricos, como, por ejemplo, Markov Chain Monte Carlo (MCMC), para el cálculo de dicha distribución a posteriori.

Por tanto, tomando un $\hat{\theta}_t \sim p_t(\theta)$, la acción escogida será:

$$a_t = argmax_a x_{t,a}^{\top} \hat{\theta}_t$$

Por último, cabe mencionar que el paso de estos modelos lineales para utilizar la información del contexto a modelos no lineales consiste en sustituir la expresión lineal $Q_t(a) = x_{t,a}^{\top} \theta^*$ por una expresión no lineal $Q_t(a) = f^*(x_{t,a})$ donde f^* es una función no lineal.

Sin embargo, este paso de una función lineal a una no lineal conlleva una serie de complicaciones teóricas que no van a ser tratadas en este trabajo.

Capítulo 4

Metodología

Como ya fue mencionado previamente, la parte práctica de este trabajo consistirá en aplicar los bandidos multibrazo a una plataforma de comercio electrónico. En este apartado se detallará el procedimiento seguido para llevar a cabo este caso de uso.

Como es habitual en el caso de las aplicaciones del aprendizaje por refuerzo, en vez de usar directamente un conjunto de datos para entrenar los modelos implementados, se construyó un simulador o bot contra el que usar los algoritmos, es decir, este bot genera para cada paso temporal (o turno) una recompensa para la acción escogida por el algoritmo utilizado. Una vez devuelta la recompensa, se actualiza la política del algoritmo del bandido multibrazo sobre sus preferencias sobre qué acción escoger en cada caso y se avanza al siguiente turno, donde el simulador genera un nuevo usuario, el algoritmo sugiere la acción más adecuada para ese usuario, y el simulador devuelve la recompensa de dicha acción.

En este caso, el usuario corresponderá con un usuario que ha accedido a la página web, las acciones serán los posibles productos que recomendar, y la recompensa será si el usuario clica o no en el producto recomendado para realizar una compra.

El simulador utilizado es de creación propia, y su funcionamiento es el siguiente:

- Primero se utiliza una función para generar un conjunto de usuarios que serán los posibles clientes. Estos usuarios tendrán asociado un id, edad, género y una categoría de productos preferida.
    - El id es un número que permitirá identificar a los usuarios de forma unívoca.
    - La edad se generará como un número entero aleatorio entre 15 y 70.
    - El género será una elección aleatoria entre masculino (M) y femenino (F).
    - Y la categoría preferida será una aleatoria de entre 5 categorías: tecnología, moda, libros, deportes y hogar.
- Se utiliza otra función diferente para generar los productos que habrá disponibles en la plataforma de comercio electrónico. Estos productos tendrán asociados un id, una categoría, una tasa base de clics que recibe el producto en la plataforma y un factor de edad.
    - El id en este caso, también será un número entero que permitirá identificar los productos de forma unívoca.

- La categoría es la sección a la que pertenece el producto. Al igual que se mencionó para el generador de usuarios, las categorías posibles serán: tecnología, moda, libros, deportes y hogar. El simulador está diseñado de forma que todas las categorías tengan el mismo número de productos.
- La tasa base de clics define la probabilidad base (previa a tener ningún tipo de información del cliente o de la hora de acceso a la página web) de que se haga clic en un determinado producto si este es recomendado a un cliente. Estas probabilidades son generadas a partir de una distribución uniforme con límite inferior en 0,15 y límite superior en 0,3, $U\,(0{,}15, 0{,}3)$.
- El factor de edad estima como de atractivo es un producto para los clientes según sus edades. Este factor añade un sumando que sigue una función lineal a la tasa base de clics (más información sobre esta más adelante). Los valores de este factor provienen de muestras aleatorias de una normal $N\,(0{,}1, 0{,}05)$ acotados inferiormente por 0,02, es decir, si alguna de las muestras aleatorias tomadas fuera inferior a 0,02, el valor de esa muestra sería fijado a dicho valor, y superiormente por 0,4.

- El tercer y último elemento del simulador es el simulador de interacciones, el cual se encarga de, para cada interacción usuario-producto, decidir si el usuario clicó o no en el producto recomendado.

Esta decisión de si el usuario clicó o no en el producto recomendado se hace tomando una muestra de una distribución Bernoulli, $Bernoulli\,(p)$, cuya probabilidad se calcula como se explicará a continuación:

Para cada producto, como ya se mencionó previamente, existe una tasa base de clics cuyos valores provienen de muestras de una distribución uniforme continua $U\,(0{,}15, 0{,}3)$.

A esta tasa base se le multiplica un factor según si la categoría de productos preferida del cliente coincide con la categoría del producto. Este factor será de 1,5 en caso de que coincidan y de 0,75 en caso de que no.

También se le multiplica un factor de género según la categoría del producto sugerido. Si el usuario es de género masculino y el producto sugerido es de las categorías de deportes o de tecnología se multiplicará por un factor de 1,1, y si es de las otras 3 categorías el factor será de 0,9. En caso de que el usuario sea de género femenino, los factores serán al revés, 0,9 para los productos de las categorías de deportes y de tecnología, y 1,1 para las otras tres categorías.

A esta tasa de clics también se le añaden dos sumandos adicionales, uno de los cuales sigue una función lineal en función de la edad del usuario y el otro sigue otra función lineal en función de la hora a la que el usuario accedió a la página web.

Por tanto, la tasa de clics final vendrá dada por la expresión:

$$ctr = base \cdot cat \cdot gen + age + hour$$

Donde $base$ es la tasa base de clics, $cat \in \{0{,}75, 1{,}5\}$ es el factor de coincidencia de categorías explicado previamente, $gen \in \{0{,}9, 1{,}1\}$ es el factor de género también previamente

explicado, *age* es el sumando lineal en función de la edad del usuario y *hour* es el sumando lineal en función de la hora a la que se accedió a la página.

El sumando *age* viene dado por la expresión:

$$age = \frac{a \cdot x}{70}$$

Donde $a \in N\,(0{,}1, 0{,}05)$ es el coeficiente de la regresión lineal y x la edad del usuario. El valor de a, además es forzado a estar acotado en el intervalo $[0{,}02, 0{,}4]$.

El sumando *hour* viene dado por la expresión:

$$hour = \frac{0{,}05 \cdot x}{24}$$

donde en este caso x es la hora de acceso a la página web.

Tras el diseño de este simulador, el siguiente paso fue utilizar dicho simulador como el generador de datos para entrenar los algoritmos de aprendizaje por refuerzo presentados en el estado del arte.

Para el caso de los algoritmos del bandido multibrazo que no utilizan el contexto (ϵ-Greedy, UCB y TS) este fue ignorado, mientras que para los algoritmos del bandido multibrazo contextual (LinUCB y LinTS) el contexto sí que fue utilizado. Sin embargo, este contexto, dado que contiene variables categóricas (género, categoría preferida del usuario y categoría del producto), dichas variables tuvieron que transformarse, ya que no pueden usarse directamente para entrenar los modelos lineales subyacentes de estos algoritmos.

Estas variables fueron codificadas utilizando One-Hot encoding, lo cual consiste en crear una variable binaria para cada categoría de una de estas variables categóricas, de forma que esa variable vale 1 si el dato seleccionado pertenece a dicha categoría y 0 en caso contrario.

Hay otras formas alternativas de codificar estas variables categóricas, como, por ejemplo, la codificación ordinal, en la que a cada categoría de una variable se le asigna un número diferente. Sin embargo, se decidió escoger el One-Hot encoding ya que este no introduce ningún sesgo en las diferentes categorías de una variable, como sí lo hace, por ejemplo, la codificación ordinal, la cual asigna un orden a las diferentes categorías.

Antes de realizar una comparativa de los diferentes modelos, los hiperparámetros de los algoritmos de ϵ-Greedy, Upper Confidence Bound, Linear Upper Confidence Bound y Linear Thompson Sampling fueron optimizados individualmente para poder tener la mejor versión posible de cada uno de estos modelos de cara a hacer una comparativa justa.

Notar que los hiperparámetros del algoritmo de Thompson Sampling no fueron optimizados. Esto es debido a que este algoritmo carece de hiperparámetros numéricos similares a los de los otros algoritmos. En este caso, el único hiperparámetro es la elección de la distribución a priori utilizada, la cual se fijó a una distribución beta.

Tras esta optimización de hiperparámetros, todos los modelos fueron entrenados a la vez, de forma que el usuario seleccionado en cada paso temporal del problema fuera el mismo para todos los algoritmos. Para este usuario, se simuló si haría clic en cada uno de los

diferentes productos disponibles en la plataforma si estos le fueran sugeridos, y una vez hecho esto se llamó a los modelos para que realizaran sus sugerencias y se devolviera el resultado del producto sugerido. Con el resultado de la sugerencia de cada algoritmo, estos fueron actualizados (cada uno únicamente con el resultado de su producto sugerido) y se pasó a la siguiente iteración.

Para medir el rendimiento de todos estos algoritmos tanto la recompensa acumulada como el regret acumulado fueron utilizados. Además de estas métricas, la precisión (número de veces que la mejor acción posible fue sugerida / número de pasos temporales) y la proporción de accesos que terminaron en que el usuario clicara en el producto sugerido (tasa de éxito) también fueron analizadas.

Capítulo 5

Resultados y discusión

Con el objetivo de medir el rendimiento de todos los algoritmos introducidos previamente en la práctica, se hizo uso del simulador introducido en la sección anterior.

Para estas pruebas, se generó un conjunto de datos con 200 usuarios diferentes. En cada paso temporal de los algoritmos, uno de estos usuarios fue escogido aleatoriamente como aquel que accedió a la página en dicho momento (este muestreo fue realizado con reposición, es decir, un mismo usuario pudo ser escogido un número indefinido de veces).

Por otro lado, se generó también otro conjunto de datos con 10 productos diferentes a recomendar, dos productos de cada una de las categorías introducidas anteriormente.

Por último, se procedió a utilizar el simulador con estos conjuntos de datos para comprobar si las recomendaciones de los diferentes algoritmos conseguían que el usuario hiciera clic en el producto sugerido. Esto se hizo durante 25.000 pasos temporales con todos los algoritmos al mismo tiempo, es decir, se simularon 25.000 accesos a la página web por alguno de los usuarios generados y se comprobó si las recomendaciones generadas por cada uno de los algoritmos consiguieron que el usuario hiciera clic en dicho producto o no. Este proceso se repitió 5 veces y las métricas de estas iteraciones fueron promediadas y estudiadas.

Antes de la comparación de unos algoritmos con otros, se siguió este mismo procedimiento para la optimización de hiperparámetros de los diferentes algoritmos. Para el caso del algoritmo ϵ-Greedy, se probaron valores del parámetro ϵ desde 0,01 hasta 1. De todos estos valores, $\epsilon = 0{,}1$ fue el valor para el que se obtuvo la mejor recompensa acumulada. Los resultados de estas pruebas con diferentes valores del parámetro ϵ se pueden observar en la figura 5.1.

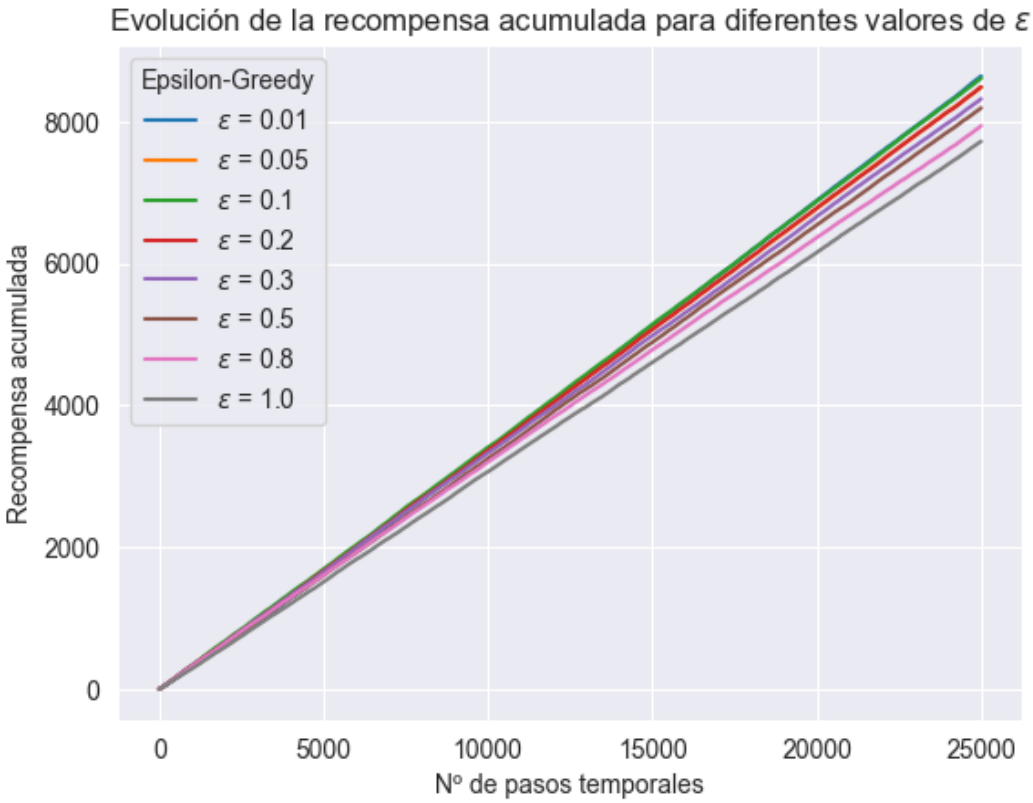


Figura 5.1: Gráfica con la evolución de la recompensa acumulada a lo largo del tiempo para diferentes valores del parámetro ϵ del algoritmo ϵ-Greedy.

Para el algoritmo de UCB, se probaron valores del parámetro α desde 0,05 hasta 10 y los mejores resultados fueron obtenidos para $\alpha = 0{,}25$ y $\alpha = 0{,}5$. De entre estos dos valores, $\alpha = 0{,}25$ fue el que finalmente fue escogido para utilizar en la comparativa entre los diferentes algoritmos. Los resultados de esta optimización de hiperparámetros se pueden ver en la figura 5.2.

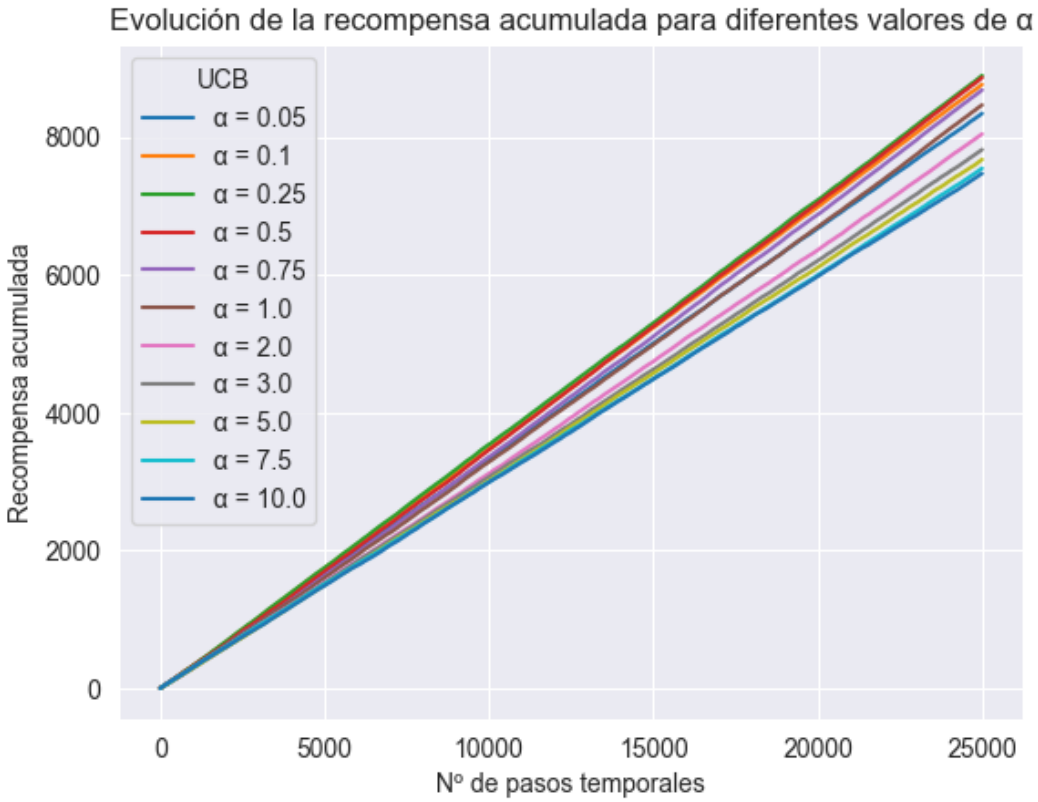


Figura 5.2: Gráfica con la evolución de la recompensa acumulada a lo largo del tiempo para diferentes valores del parámetro α del algoritmo UCB.

Para el algoritmo de LinUCB, se obtuvo que en este caso los valores de α que mejores resultados dieron fueron $\alpha = 0{,}75$ y $\alpha = 1$ de entre todos los probados (se probaron los mismos valores que para UCB). De entre estos dos valores se escogió $\alpha = 0{,}75$ para la comparativa entre los diferentes algoritmos. Los resultados de estas pruebas se pueden ver en la figura 5.3.

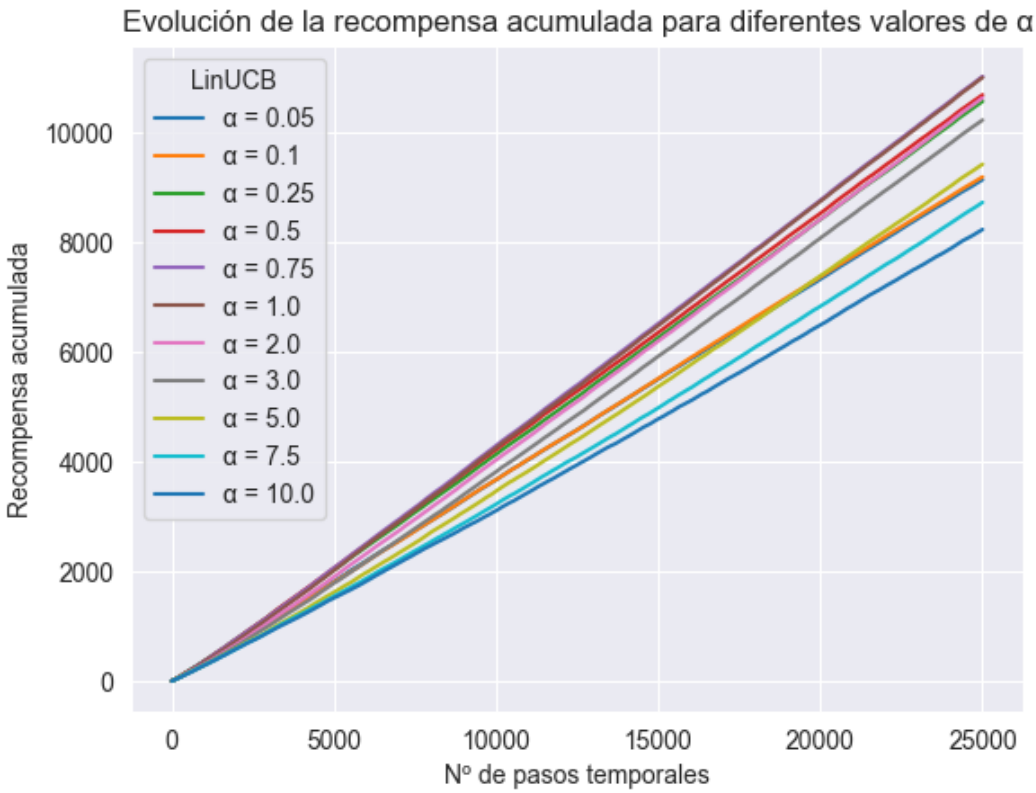


Figura 5.3: Gráfica con la evolución de la recompensa acumulada a lo largo del tiempo para diferentes valores del parámetro α del algoritmo LinUCB.

Y para el algoritmo LinTS, se probaron valores del parámetro ν entre 0,1 y 5, obteniéndose los mejores resultados para $\nu = 0{,}3$. Estos resultados están representados en la figura 5.4.

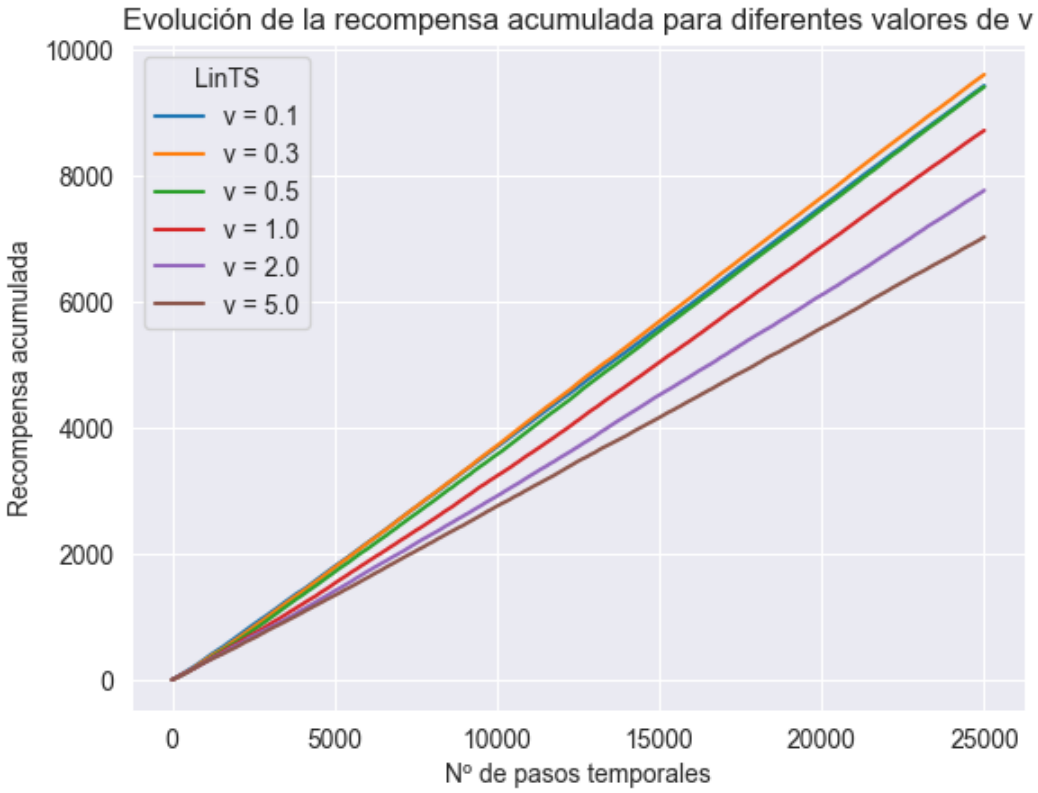


Figura 5.4: Gráfica con la evolución de la recompensa acumulada a lo largo del tiempo para diferentes valores del parámetro ν del algoritmo LinTS.

Una vez optimizados los hiperparámetros de todos los algoritmos, se procedió a la comparativa de los algoritmos entre sí. Para ello, se utilizaron los valores de los parámetros previamente mencionados y se compararon los valores de las métricas obtenidos de cada uno de los algoritmos.

En lo que respecta a la evolución de las recompensas acumuladas, en la gráfica de la figura 5.5 puede apreciarse que, como era de esperar, los algoritmos contextuales son superiores a los que ignoran el contexto en el caso de que este esté disponible, y todos estos algoritmos, con o sin contexto, son claramente mejores que escoger los productos a recomendar de forma completamente aleatoria. En particular, el algoritmo de LinTS obtiene unos resultados ligeramente superiores a LinUCB, siendo por tanto, el mejor de todos los estudiados para este caso concreto.

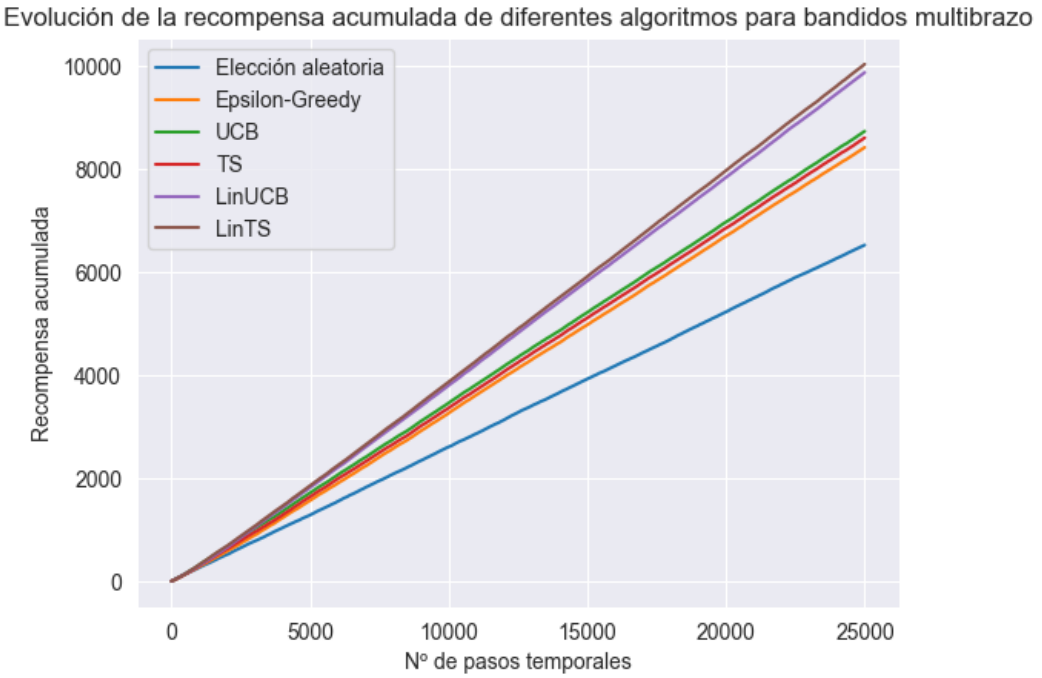


Figura 5.5: Gráfica con la evolución de la recompensa acumulada a lo largo del tiempo para los diferentes algoritmos utilizados.

Por otro lado, si se observa el regret acumulado (figura 5.6), el orden de mejor a peor rendimiento de los algoritmos fue prácticamente el mismo (notar que en el caso del regret acumulado se busca que este sea lo menor posible). El algoritmo con el menor regret acumulado fue también LinTS, seguido de cerca por LinUCB y con una clara ventaja sobre el resto de los algoritmos.

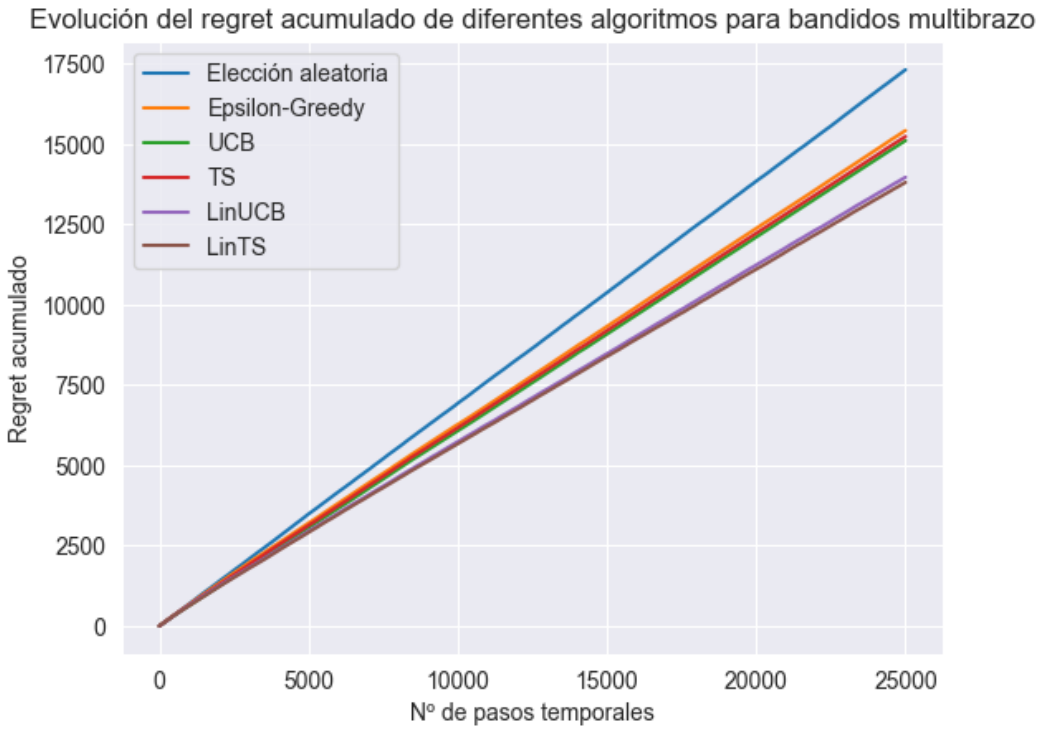


Figura 5.6: Gráfica con la evolución del regret acumulado a lo largo del tiempo para los diferentes algoritmos utilizados.

Además de estas dos métricas, también se midieron la tasa de éxito y la precisión de los algoritmos, cuyos resultados se pueden ver en la tabla 5.1. Para estas métricas, LinTS también fue el algoritmo que obtuvo los mejores resultados.

Algoritmo	**Tasa de éxito**	**Precisión**
Elección aleatoria	0,261	0,099
Epsilon-Greedy	0,336	0,283
UCB	0,349	0,321
TS	0,344	0,307
LinUCB	0,395	0,657
LinTS	0,401	0,744

Tabla 5.1: Resultados de la tasa de éxito y precisión de todos los algoritmos utilizados.

En lo que respecta al regret acumulado, cabe mencionar que en este caso fue calculado a partir de datos empíricos mediante la fórmula:

$$R\left(T\right) = \sum_{t=1}^{T} \left(\hat{r}_t - r_t\right)$$

Donde r_t es la recompensa del algoritmo seleccionado en ese paso temporal y $\hat{r}_t$ es, de todas las posibles recompensas que podrían haberse dado en dicho paso temporal según el

producto recomendado, aquella que fue máxima.

El hecho de usar esta fórmula alternativa para el cálculo del regret acumulado es debido a que la fórmula teórica introducida previamente en el estado del arte no puede ser usada directamente en la práctica en la mayoría de los casos, ya que en algunos casos, la acción cuya recompensa es óptima es desconocida, o como en este caso, al tratarse de un proceso estocástico (al estar muestreando de una distribución Bernoulli) la acción óptima, que es aquella con mayor probabilidad de que dicha sugerencia sea exitosa, puede no ser exitosa, mientras que otra sugerencia con menor probabilidad de éxito sí puede acabar siéndolo. Esto conllevaría que el regret, calculado mediante la fórmula teórica, pudiera ser negativo en algún paso temporal, lo cual no es posible por definición, como es explicado en Loomes y Sugden (1982).

Capítulo 6

Conclusiones

En este documento se ha realizado un estudio del problema del bandido multibrazo y su extensión a los bandidos multibrazo contextuales, explicando con detalle los principales algoritmos usados tradicionalmente para sus resoluciones.

Este tipo de problemas, como se ha podido ver en el trabajo, pueden trasladarse desde el plano teórico al plano práctico en múltiples ámbitos, siendo por tanto de gran utilidad.

En el caso del uso de estos algoritmos para la implementación de sistemas de recomendación, esto supone una clara mejora de las recomendaciones con respecto a la realización de estas de forma completamente aleatoria, de ahí el auge de este tipo de sistemas en plataformas de comercio electrónico, redes sociales, etc.

En lo que respecta a los algoritmos implementados en el sistema de recomendación diseñado, estos o bien ignoran el contexto por completo, o lo modelan como una función lineal, lo cual en muchos casos prácticos no se asemeja a la realidad.

A día de hoy, el estado del arte de los sistemas de recomendación se está moviendo cada vez más hacia el uso de modelos de aprendizaje profundo cada vez más complejos. Sin embargo, el uso de estos modelos clásicos siempre tendrá cabida para ciertas aplicaciones, especialmente aquellas en las que el contexto disponible es bastante escaso o se carece de él, situaciones en las que el aprendizaje profundo sufre para ser capaz de dar buenos resultados. Por este motivo, el estudio de estas técnicas sigue siendo a día de hoy de vital importancia.

En lo que respecta a las técnicas de aprendizaje profundo que pueden ser usadas para resolver el problema del bandido multibrazo, su estudio y comparación de rendimiento con las técnicas estudiadas en este documento se deja para futuros trabajos.

Bibliografía

Loomes, G., & Sugden, R. (1982). Regret Theory: An Alternative Theory of Rational Choice Under Uncertainty. *The Economic Journal*, *92*(368), 805-824. https://doi.org/10.2307/2232669

Rohde, D., Bonner, S., Dunlop, T., Vasile, F., & Karatzoglou, A. (2018). RecoGym: A Reinforcement Learning Environment for the problem of Product Recommendation in Online Advertising. https://arxiv.org/abs/1808.00720

Russo, D., Roy, B. V., Kazerouni, A., & Osband, I. (2017). A Tutorial on Thompson Sampling. *CoRR*, *abs/1707.02038*. http://arxiv.org/abs/1707.02038

Sutton, R., & Barto, A. (2018). *Reinforcement Learning, second edition: An Introduction*. MIT Press. https://books.google.es/books?id=sWV0DwAAQBAJ

Zhou, L. (2015). A Survey on Contextual Multi-armed Bandits. *CoRR*, *abs/1508.03326*. http://arxiv.org/abs/1508.03326

Anexo A

Implementación del código en Python

En este anexo se muestra el código de Python usado para la implementación del simulador, los algoritmos de ϵ-Greedy, UCB, TS, LinUCB y LinTS, y el entrenamiento de estos con los datos generados por el simulador. Todo este código está disponible en el repositorio de GitHub accesible mediante la siguiente URL: https://github.com/Eduard0LP/MABE-Multi-Armed-Bandits-for-E-commerce

La implementación de todo el código relacionado con los modelos y el simulador se realizó utilizando únicamente las librerías de *numpy* y *pandas* como dependencias. Además de estas librerías, también fueron usadas en el código realizado las librerías de *scikit-learn* para la codificación de las variables categóricas en variables numéricas durante el entrenamiento de las regresiones lineales de los algoritmos contextuales, y las librerías de *matplotlib* y *seaborn* para la representación gráfica de los resultados.

En lo que respecta al simulador, la función que genera los usuarios que serán los potenciales clientes de la plataforma de comercio electrónico fue llamada *generate_user* y devuelve los datos de un usuario en un diccionario de Python. Su implementación se puede ver en la figura A.1.

```python
def generate_user(user_id: int, product_categories: list[str]) -> dict[str, Any]:
    """Funtion to generate a user for an e-commerce platform.

    This function can be used to generate a user for an e-commerce platform so that a Multi-Armed Bandit algorithm
    can be trained. It has to be combined with the functions generate_products and simulate_interactions in
    order to generate the final dataset that will be fed to the Multi-Armed Bandit algorithms.

    Args:
        user_id: ID that will be given to the user.
        product_categories: Possible options for the prefered_category field of the user dictionary.

    Returns:
        A dictionary containing the following fields:
            - user_id: The user_id passed as argument to the function.
            - age: Age of the user. It is an integer between 15 and 70 generated randomly.
            - gender: Gender of the user. Randomly chosen between male or female (M or F).
            - preferred_category: Preferred category of products of the user. It can be any given category from the
                product_categories list that is passed as argument to the function.
    """
    return {
        'user_id': user_id,
        'age': np.random.randint(15, 70),
        'gender': np.random.choice(['M', 'F']),
        'preferred_category': np.random.choice(product_categories)
    }
```

Figura A.1: Función en Python usada para generar los usuarios de la página web.

Para generar los productos se utilizó la función *generate_products* (Figura A.2), la cual retorna un *dataframe* de *pandas* con los datos de todos los productos disponibles.

```python
def generate_products(n_products: int, product_categories: list[str]) -> pd.DataFrame:
    """Funtion to generate the products for an e-commerce platform.

    This function can be used to generate the products for an e-commerce platform so that a Multi-Armed Bandit
    algorithm can be trained. It has to be combined with the functions generate_user and
    simulate_interactions in order to generate the final dataset that will be fed to the Multi-Armed Bandit algorithms.

    Args:
        n_products: Number of products that will be generated.
        product_categories: Possible options for the category column of the products dataframe.

    Returns:
        A pandas dataframe containing the following columns:
            - product_id: ID to identify the product.
            - category: Category the product belongs to.
            - base_ctr: Base rate/probability at which the product gets clicked when showed to a user. It is obtained
                from sampling a uniform distribution U(0.15, 0.3).
            - age_aggregate_factor: Coefficient of an additive factor dependant on age that is added to the base_ctr.
                age_aggregate is a linear function with respect to the age of the user that gets added to the base_ctr
                probability. It is obtained from a sample of a gaussian distribution N(0.1, 0.05) and clipping this
                value to a minimum of 0.02.
    """
    products = []
    for i in range(n_products):
        products.append({
            'product_id': i,
            'category': product_categories[int(i//(n_products/len(product_categories)))],
            'base_ctr': np.random.uniform(0.15, 0.3),
            'age_aggregate_factor': np.clip(np.random.normal(0.1, 0.05), 0.02, 0.4)
        })
    return pd.DataFrame(products)
```

Figura A.2: Función en Python usada para generar los productos de la página web.

```python
# Generar usuarios y productos
users = [generate_user(i) for i in range(n_users)]
products_df = generate_products()
```

Figura A.3: Fragmento de código en que las funciones *generate_user* y *generate_products* son llamadas para generar los usuarios y los productos.

El tercer y último elemento del simulador, como se mencionó en la sección de metodología, era el encargado de simular las interacciones entre usuarios y productos, lo cual se implementó con la función *simulate_interactions*, la cual recibe los datos del usuario generado por *generate_user* y el dataframe generado por *generate_products* y simula si para cada producto de dicho *dataframe* dicho usuario haría clic o no en el producto si este le fuera recomendado. Los resultados de todas estas simulaciones son retornados en un nuevo *dataframe*. La implementación de dicha función se puede ver en la figura A.4.

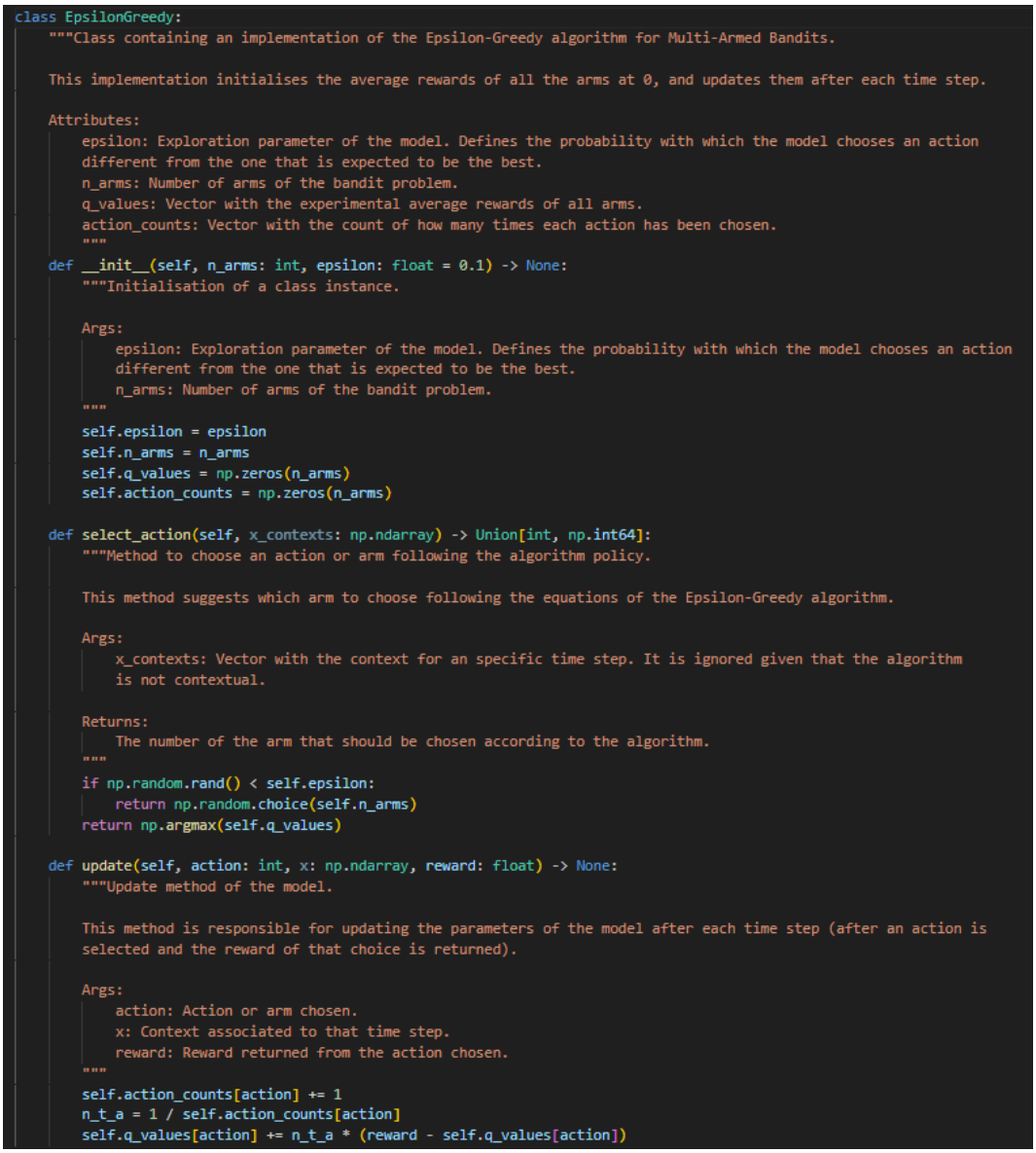
```python
def simulate_interactions(user: dict[str, Any], products_df: pd.DataFrame, timestamp: pd.Timestamp,
                          cat_splits: tuple[list[str], list[str]] | None = None) -> pd.DataFrame:
    """Funtion to simulate the interactions between a user and all available products.

    This function simulates for a given user, if he/she would click in each one of the products of the
    products_df dataframe if each of these products was recommended to him/her by a Multi-Armed Bandit
    algorithm. It has to be combined with the functions generate_user and generate_products to be used as a complete
    simulator.

    The probability of clicking on a product if recommended is given by the following expression:

        base_ctr * category_multiplier * gender_multiplier + age_aggregate + hour_aggregate

    where:
        - base_ctr is the value given in the dataframe generated by generate_products.
        - category_multiplier is a multiplicative factor that has value 1.5 if the category of the product matches
          the preferred category of the user and 0.75 if not.
        - gender_multiplier is a multiplicative factor that takes value 1.1 if the category of the product is between
          the preferred categories of that gender, and 0.9 if not.
        - age_aggregate is an additive factor given by the linear expression age_aggregate_factor * age / 70. The
          age_aggregate_factor is the one given in the dataframe generated by generate_products.
        - hour_aggregate is an additive factor given by the linear expression 0.05 * timestamp.hour / 24.

    Args:
        user: User data. It is given by the output of the generate_user function.
        products_df: Dataframe with all the product characteristics.It is given by the output of the generate_products
        function.
        timestamp: Time at which the user accesed the platform.
        cat_splits: Optional parameter consisting of a tuple with 2 lists. The first list gives the preferred
        categories of the male users and the second list the preferred categories of the female users. If not provided
        the function _split_randomly is called to generate this tuple.

    Returns:
        A pandas dataframe containing the following columns:
            - product_id: ID of the product. It is copied from the column with the same name on products_df.
            - ctr: Probability of the binomial distribution used to decide whether the user clicked or not on a
              product if recommended.
            - click: Binary variable stating whether the user clicked or not on the product with that
              product_id if recommended to the user. It is obtained from a sample of a a binomial
              distribution B(1, ctr).
    """
    age_aggregate = products_df['age_aggregate_factor'].values * user['age'] / 70

    category_multiplier = np.where(products_df['category'].values == user['preferred_category'], 1.5, 0.75)

    if cat_splits is None:
        cat_splits = _split_randomly(products_df['category'].unique())

    if user['gender'] == 'M':
        condition = np.isin(products_df['category'].values, cat_splits[0])
    elif user['gender'] == 'F':
        condition = np.isin(products_df['category'].values, cat_splits[1])
    else:
        condition = np.zeros(len(products_df), dtype=bool)

    gender_multiplier = np.where(condition, 1.1, 0.9)

    hour_aggregate = 0.05 * timestamp.hour / 24

    ctr = products_df['base_ctr'].values * category_multiplier * gender_multiplier + age_aggregate + hour_aggregate

    click = np.random.binomial(1, ctr)

    return pd.DataFrame({'product_id': products_df['product_id'], 'ctr': ctr, 'click': click})
```

Figura A.4: Función en Python para simular las interacciones producto-cliente.

En lo que respecta a las implementaciones de los algoritmos discutidos en el trabajo, estos fueron implementados usando clases de Python con 3 métodos principales: un método *__init__* para inicializar la clase con la parametrización adecuada del algoritmo, un método *select_action* mediante el cual el algoritmo realiza la recomendación de un producto en función del contexto y un método *update* para actualizar los parámetros del algoritmo tras recibir el resultado de si la recomendación realizada tuvo recompensa o no. Es importante destacar que incluso para los algoritmos que no utilizan el contexto, este es pasado a los métodos

select_action y *update* y en esas clases simplemente es ignorado.

Esto está hecho así para mantener exactamente el mismo formato en todos los modelos y así facilitar el entrenamiento de todos ellos a la vez en un solo bucle de entrenamiento con el simulador. Las implementaciones de todos estos modelos se pueden ver en las figuras A.5, A.6, A.7, A.8 y A.9. En lo que respecta al entrenamiento de todos los modelos, este se realizó a la vez en el bucle de entrenamiento que se puede ver en la figura A.10.



```python
class EpsilonGreedy:
    """Class containing an implementation of the Epsilon-Greedy algorithm for Multi-Armed Bandits.

    This implementation initialises the average rewards of all the arms at 0, and updates them after each time step.

    Attributes:
        epsilon: Exploration parameter of the model. Defines the probability with which the model chooses an action
        different from the one that is expected to be the best.
        n_arms: Number of arms of the bandit problem.
        q_values: Vector with the experimental average rewards of all arms.
        action_counts: Vector with the count of how many times each action has been chosen.
        """
    def __init__(self, n_arms: int, epsilon: float = 0.1) -> None:
        """Initialisation of a class instance.

        Args:
            epsilon: Exploration parameter of the model. Defines the probability with which the model chooses an action
            different from the one that is expected to be the best.
            n_arms: Number of arms of the bandit problem.
        """
        self.epsilon = epsilon
        self.n_arms = n_arms
        self.q_values = np.zeros(n_arms)
        self.action_counts = np.zeros(n_arms)

    def select_action(self, x_contexts: np.ndarray) -> Union[int, np.int64]:
        """Method to choose an action or arm following the algorithm policy.

        This method suggests which arm to choose following the equations of the Epsilon-Greedy algorithm.

        Args:
            x_contexts: Vector with the context for an specific time step. It is ignored given that the algorithm
            is not contextual.

        Returns:
            The number of the arm that should be chosen according to the algorithm.
        """
        if np.random.rand() < self.epsilon:
            return np.random.choice(self.n_arms)
        return np.argmax(self.q_values)

    def update(self, action: int, x: np.ndarray, reward: float) -> None:
        """Update method of the model.

        This method is responsible for updating the parameters of the model after each time step (after an action is
        selected and the reward of that choice is returned).

        Args:
            action: Action or arm chosen.
            x: Context associated to that time step.
            reward: Reward returned from the action chosen.
        """
        self.action_counts[action] += 1
        n_t_a = 1 / self.action_counts[action]
        self.q_values[action] += n_t_a * (reward - self.q_values[action])
```

Figura A.5: Clase de Python con la implementación del algoritmo ϵ-Greedy.

```python
class UCB:
    """Class containing an implementation of the Upper Confidence Bound (UCB) algorithm for Multi-Armed Bandits.

    This implementation initialises the average rewards of all the arms at 0, and updates them after each time step.

    Attributes:
        n_arms: Number of arms of the bandit problem.
        q_values: Vector with the experimental average rewards of all arms.
        action_counts: Vector with the count of how many times each action has been chosen.
        total_counts: Total amount of time steps the algorithm has taken.
        alpha: Parameter of the UCB algorithm that controls exploration.
    """
    def __init__(self, n_arms: int, alpha: float = 1.0) -> None:
        """Initialisation of a class instance.

        Args:
            n_arms: Number of arms of the bandit problem.
            alpha: Parameter of the UCB algorithm that controls exploration.
        """
        self.n_arms = n_arms
        self.q_values = np.zeros(n_arms)
        self.action_counts = np.zeros(n_arms)
        self.total_counts = 0
        self.alpha = alpha

    def select_action(self, x_contexts: np.ndarray) -> np.int64:
        """Method to choose an action or arm following the algorithm policy.

        This method suggests which arm to choose following the equations of the UCB algorithm.

        Args:
            x_contexts: Vector with the context for an specific time step. It is ignored given that the algorithm
            is not contextual.

        Returns:
            The number of the arm that should be chosen according to the algorithm.
        """
        self.total_counts += 1
        # The value 1e-5 is a small value used to avoid dividing by 0
        ucb_values = self.q_values + self.alpha * np.sqrt(np.log(self.total_counts) / (self.action_counts + 1e-5))
        return np.argmax(ucb_values)

    def update(self, action: int, x: np.ndarray, reward: float) -> None:
        """Update method of the model.

        This method is responsible for updating the parameters of the model after each time step (after an action is
        selected and the reward of that choice is returned).

        Args:
            action: Action or arm chosen.
            x: Context associated to that time step.
            reward: Reward returned from the action chosen.
        """
        self.action_counts[action] += 1
        n_t_a = 1 / self.action_counts[action]
        self.q_values[action] += n_t_a * (reward - self.q_values[action])
```

Figura A.6: Clase de Python con la implementación del algoritmo Upper Confidence Bound.

```python
class ThompsonSampling:
    """Class containing an implementation of the Thompson Sampling (TS) algorithm for Multi-Armed Bandits.

    This implementation considers a beta distribution B(alpha, beta) as the prior distribution of the rewards.
    The parameters of the distribution are both initialised to 1.

    Attributes:
        n_arms: Number of arms of the bandit problem.
        successes: Vector with the amount of positive rewards of the suggestions of each arm plus one. This corresponds
        to the alpha parameter of the beta distribution.
        failures: Vector with the amount of negative rewards (no reward) of the suggestions of each arm plus one. This
        corresponds to the beta parameter of the beta distribution. .
    """
    def __init__(self, n_arms: int) -> None:
        """Initialisation of a class instance.

        Args:
            n_arms: Number of arms of the bandit problem.
        """
        self.n_arms = n_arms
        self.successes = np.ones(n_arms)
        self.failures = np.ones(n_arms)

    def select_action(self, x_contexts: np.ndarray) -> np.int64:
        """Method to choose an action or arm following the algorithm policy.

        This method suggests which arm to choose following the equations of the TS algorithm.

        Args:
            x_contexts: Vector with the context for an specific time step. It is ignored given that the algorithm
            is not contextual.

        Returns:
            The number of the arm that should be chosen according to the algorithm.
        """
        sampled_means = np.random.beta(self.successes, self.failures)
        return np.argmax(sampled_means)

    def update(self, action: int, x: np.ndarray, reward: float) -> None:
        """Update method of the model.

        This method is responsible for updating the parameters of the model after each time step (after an action is
        selected and the reward of that choice is returned).

        Args:
            action: Action or arm chosen.
            x: Context associated to that time step.
            reward: Reward returned from the action chosen.
        """
        if reward == 1:
            self.successes[action] += 1
        else:
            self.failures[action] += 1
```

Figura A.7: Clase de Python con la implementación del algoritmo Thompson Sampling.

```python
class LinUCB:
    """Class containing an implementation of the Linear Upper Confidence Bound (LinUCB) algorithm for Contextual
    Multi-Armed Bandits.

    This implementation initialises the linear regresion coefficient matrices to the identity matrix and the vectors of
    independent terms to 0, and updates them after each time step.

    Attributes:
        n_arms: Number of arms of the bandit problem.
        d: Dimension of the context vector that the algorithm should expect.
        alpha: Parameter of the LinUCB algorithm that controls exploration.
        A: Linear regresion coefficient matrices.
        b: Vectors of independent terms of the linear regressions.
    """
    def __init__(self, n_arms: int, context_dim: int, alpha: float = 0.1) -> None:
        """Initialisation of a class instance.

        Args:
            n_arms: Number of arms of the bandit problem.
            context_dim: Dimension of the context vector that the algorithm should expect.
            alpha: Parameter of the LinUCB algorithm that controls exploration.
        """
        self.n_arms = n_arms
        self.d = context_dim
        self.alpha = alpha

        self.A = [np.identity(self.d) for _ in range(n_arms)]  # A_a = d x d
        self.b = [np.zeros((self.d, 1)) for _ in range(n_arms)]  # b_a = d x 1

    def select_action(self, x_contexts: np.ndarray) -> np.int64:
        """Method to choose an action or arm following the algorithm policy.

        This method suggests which arm to choose following the equations of the LinUCB algorithm.

        Args:
            x_contexts: Vector with the context for an specific time step. It has shape (d, 1).

        Returns:
            The number of the arm that should be chosen according to the algorithm.
        """
        p_values = []

        for a in range(self.n_arms):
            A_inv = np.linalg.inv(self.A[a])
            theta = A_inv @ self.b[a]
            x = x_contexts[a]
            p = ((x.T @ theta) + self.alpha * np.sqrt((x.T @ A_inv @ x)))[0, 0]
            p_values.append(p)

        return np.argmax(p_values)

    def update(self, action: int, x: np.ndarray, reward: float) -> None:
        """Update method of the model.

        This method is responsible for updating the parameters of the linear regression of the model after each time
        step (after an action is selected and the reward of that choice is returned).

        Args:
            action: Action or arm chosen.
            x: Context associated to that time step.
            reward: Reward returned from the action chosen.
        """
        self.A[action] += x @ x.T
        self.b[action] += reward * x
```

Figura A.8: Clase de Python con la implementación del algoritmo Linear Upper Confidence Bound.

```python
class LinTS:
    """Class containing an implementation of the Linear Thompson Sampling (LinTS) algorithm for Contextual
    Multi-Armed Bandits.

    This implementation initialises the linear regresion coefficient matrices to the identity matrix and the vectors of
    independent terms to 0 and considers a multivariate normal distribution N(mu, sigma) as the prior distribution of
    the rewards. Both the coefficient matrices and vectors of independent terms get updated after each time step
    based on the reward obtained.

    Attributes:
        n_arms: Number of arms of the bandit problem.
        d: Dimension of the context vector that the algorithm should expect.
        v: Parameter of the LinTS algorithm that scales the values of the covariance matrix of the multivariate normal
        distribution. It controls exploration.
        A: Linear regresion coefficient matrices.
        b: Vectors of independent terms of the linear regressions.
    """
    def __init__(self, n_arms: int, context_dim: int, v: float = 1.0) -> None:
        """Initialisation of a class instance.

        Args:
            n_arms: Number of arms of the bandit problem.
            context_dim: Dimension of the context vector that the algorithm should expect.
            v: Parameter of the LinTS algorithm that scales the values of the covariance matrix of the multivariate
            normal distribution. It controls exploration.
        """
        self.n_arms = n_arms
        self.d = context_dim
        self.v = v

        self.A = [np.identity(self.d) for _ in range(n_arms)]  # A_a = d x d
        self.b = [np.zeros((self.d, 1)) for _ in range(n_arms)]  # b_a = d x 1

    def select_action(self, x_contexts: np.ndarray) -> np.int64:
        """Method to choose an action or arm following the algorithm policy.

        This method suggests which arm to choose following the equations of the LinTS algorithm.

        Args:
            x_contexts: Vector with the context for an specific time step. It has shape (d, 1).

        Returns:
            The number of the arm that should be chosen according to the algorithm.
        """
        p_values = []

        for a in range(self.n_arms):
            A_inv = np.linalg.inv(self.A[a])
            mu = A_inv @ self.b[a]
            cov = self.v ** 2 * A_inv

            theta_sample = np.random.multivariate_normal(mu.flatten(), cov).reshape(-1, 1)
            x = x_contexts[a]

            p = (theta_sample.T @ x)[0, 0]
            p_values.append(p)

        return np.argmax(p_values)

    def update(self, action: int, x: np.ndarray, reward: float) -> None:
        """Update method of the model.

        This method is responsible for updating the parameters of the linear regression of the model after each time
        step (after an action is selected and the reward of that choice is returned).

        Args:
            action: Action or arm chosen.
            x: Context associated to that time step.
            reward: Reward returned from the action chosen.
        """
        self.A[action] += x @ x.T
        self.b[action] += reward * x
```

Figura A.9: Clase de Python con la implementación del algoritmo Linear Thompson Sampling.

```python
rewards_runs = []
regrets_runs = []
accuracies = []
for i in range(n_runs):
    policies = {
        "random": EpsilonGreedy(n_arms=n_arms, epsilon=1.),
        "epsilon_greedy": EpsilonGreedy(n_arms=n_arms, epsilon=0.1),
        "ucb": UCB(n_arms=n_arms, alpha=0.25),
        "thompson_sampling": ThompsonSampling(n_arms=n_arms),
        "linucb": LinUCB(n_arms=n_arms, context_dim=10, alpha=0.75),
        "lints": LinTS(n_arms=n_arms, context_dim=10, v=0.3),
    }
    cum_reward = [0] * len(policies)
    cum_regret = [0] * len(policies)
    accuracy = [0] * len(policies)
    rewards = [[0] for _ in range(len(policies))]
    regrets = [[0] for _ in range(len(policies))]
    for i in range(n_interactions):
        user = np.random.choice(users)
        start = pd.Timestamp('2025-01-01 00:00:00')
        end = pd.Timestamp('2025-12-31 23:59:59')
        timestamp = start + pd.to_timedelta(np.random.uniform(0, 1) * (end - start))

        # Create context vectors
        context_df = products_df.assign(**user)
        context_df = context_df.loc[:, ['age', 'gender', 'preferred_category', 'category']]
        # List of categorical columns
        categorical_cols = ['gender', 'preferred_category', 'category']
        categories = [['M', 'F'], product_categories, product_categories]
        # Create the column transformer
        column_transformer = ColumnTransformer(
            transformers=[
                ('onehot', OneHotEncoder(sparse_output=False, categories=categories, drop='first'), categorical_cols)
            ],
            remainder='passthrough'  # or 'passthrough' to keep non-categorical columns
        )
        # Fit and transform the data
        encoded_array = column_transformer.fit_transform(context_df)
        x_contexts = list(encoded_array[:, :, None])

        cat_splits = (['technology', 'sports'], ['fashion', 'books', 'home'])
        interactions = simulate_interactions(user, products_df, timestamp, cat_splits)

        for j, key in enumerate(policies.keys()):
            recommendation = policies[key].select_action(x_contexts)
            product = interactions.iloc[recommendation, 0]
            click = interactions.iloc[recommendation, -1]

            if recommendation == int(interactions.loc[interactions['ctr'].idxmax()]['product_id']):
                accuracy[j] += 1
            cum_reward[j] += click
            cum_regret[j] += interactions.iloc[:, -1].max() - click
            rewards[j].append(cum_reward[j])
            regrets[j].append(cum_regret[j])

            policies[key].update(recommendation, x_contexts[recommendation], click)
    accuracies.append(accuracy)
    rewards_runs.append(rewards)
    regrets_runs.append(regrets)
```

Figura A.10: Entrenamiento simultáneo de todos los algoritmos.