Marketing decision engine

<Versión 1.0>

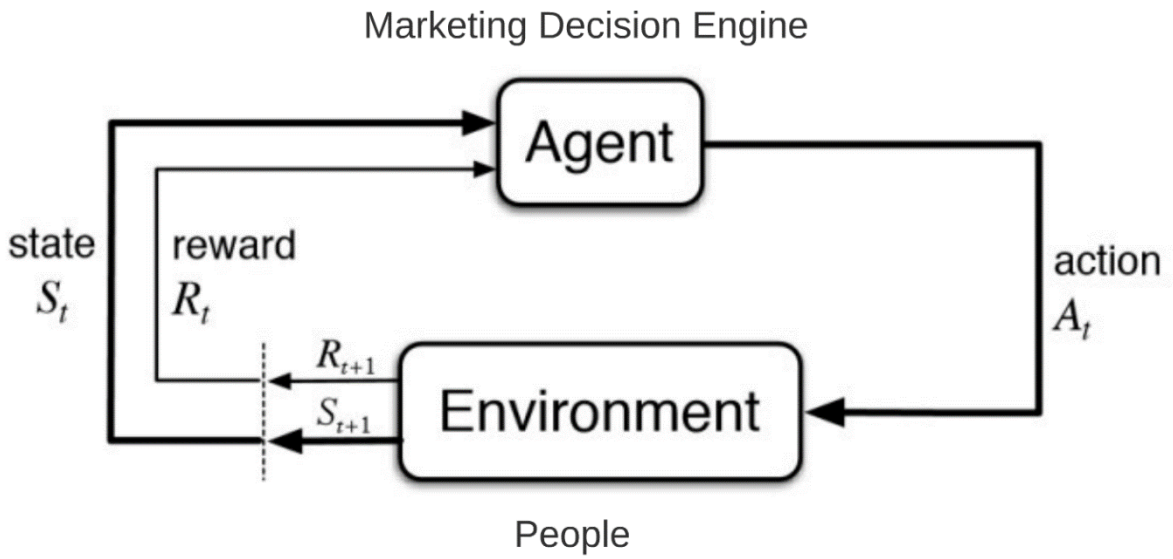
Índice

Control de Ediciones

Edición	Fecha	Cambio	Autor del cambio
V 1.0	21/05/2019	Primera versión	Fernando Fuentes

1. Propósito

En este proyecto vamos a plantear un modelo de Reinforcement Learning donde el agente va a ser un motor de decisión de marketing que va a intentar vender productos a una población de clientes en concreto (entorno).



2. Tecnologías

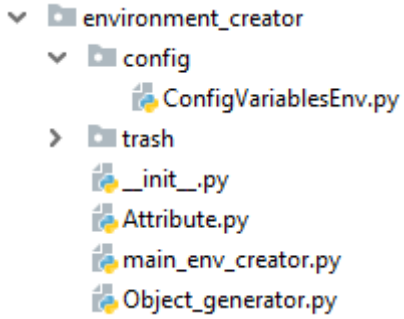
Para realizar prototipos de manera rápida y sencilla mi elección de tecnologías es:

- Backend: Python 3.0
- Capa de persistencia: Mongo DB
- Frontend: R (Shiny)

3. Generación de los datos

Para el propósito de este proyecto, en vez de conseguir información de clientes, vamos crear un software parametrizable que pueda crear diferentes entidades con los atributos que definamos para las mismas.

La estructura específica de este módulo es la siguiente:



3.1. Atributos (Attribute.py)

Cuando definamos un tipo de objeto (ej: persona) vamos a darle sentido configurando para el mismo una serie de atributos (ej: nombre, DNI, edad,…) que lo definan.

Para ello primero vamos a definir que tipos de atributos necesitamos y como deben generarse. Aunque se nos ocurren más, de momento vamos a generar el mínimo de tipos necesarios, que van a ser 3, un tipo numérico, un tipo texto y finalmente un tipo fecha.

La clase `Attribute` acepta como parámetro en la creación un objeto que represente una conexión a la BBDD (en este caso Mongo DB). Esta clase va a tener un único punto de entrada que va a ser el método `create_attribute()` que recibe dos parámetros:

- Attdic: Un diccionario que contiene la descripción de como debe crearse el atributo.
- Attnum: Cuantos atributos de este tipo debe crear.

Esta clase devolverá una lista con la cantidad de atributos solicitados.

Como indicamos cada atributo se define con un diccionario, que tiene como claves comunes a los distintos tipos:

- OBJ_NAME: Nombre del objeto que posee el atributo.
- ATT_NAME: Nombre del atributo.
- ATT_CONTR: Si esta clave tiene como valor "UNIQUE", significa que este atributo no puede contener valores repetidos en la lista devuelta. Es decir, no puede haber dos objetos que compartan el mismo valor de atributo. (Ej: DNI)
- ATT_TYPE: Puede tomar 3 valores: NUMBER, TEXT o DATE.

El atributo de tipo **NUMBER** será procesado por la función `create_numbers()` y esperará en el diccionario una serie de claves específicas:

- PRECISION: Cantidad de decimales que contendrá la variable numérica.
- RANGE: Numero máximo y mínimo entre los cuales se generará el número.
- GENERATION: La generación de la lista de números podrá realizarse de dos maneras:
  - RANDOM: Generación aleatoria.
  - SEQUENTIAL: Generación secuencial.
- HOP: En caso de que la generación sea secuencial, en el valor del HOP indicaremos el salto entre números.

El atributo de tipo **TEXT** será procesado por la función `create_text()` y esperará en el diccionario una serie de claves específicas:

- LENGTH: Numero máximo de caracteres del texto.
- GENERATION: La generación de la lista de textos podrá realizarse de dos maneras:
  - RANDOM: Generación aleatoria.
  - BBDD: Texto aleatorio escogido de una base de datos.
  - CTE: Texto constante.
- BBDD_SOURCE: En caso de que la generación sea por base de datos, este campo indicará la tabla/colección de donde se obtendrá el texto.
- BBDD_FIELD: En caso de que la generación sea por base de datos, este campo indicará la columna/clave de donde se obtendrá el texto.
- CTE_STR: En caso de que la generación sea constante, este campo contendrá el texto.

El atributo de tipo **DATE** será procesado por la función `create_date()` y esperará en el diccionario una serie de claves específicas:

- GENERATION: La generación de la lista de fechas podrá realizarse de dos maneras:
  - RANDOM: Generación aleatoria.
  - CTE: Generación constante.
- MIN_DATE: Fecha mínima que puede tomar la generación aleatoria.
- MAX_DATE: Fecha máxima que puede tomar la generación aleatoria.

- CTE_DATE: Fecha constante.

3.2. Objetos (Object_generator.py)

Como hemos comentado vamos a representar un objeto (Ej: persona) como entidad que posee una serie de atributos (Ej: nombre, apellidos, peso, altura,…).

Para el propósito de nuestro proyecto, vamos a crear el objeto PERSONA con los siguientes atributos:

- NAME: Nombre de la persona, que será un atributo de tipo texto que generaremos seleccionándolos de manera aleatoria desde una BBDD.
- SURNAME1: Primer apellido, que será un atributo de tipo texto que generaremos seleccionándolos de manera aleatoria desde una BBDD.
- SURNAME2: Segundo apellido, que será un atributo de tipo texto que generaremos seleccionándolos de manera aleatoria desde una BBDD.
- ID_DOC: Identificador único que representa de manera unívoca a cada persona. Se generará como una cadena aleatoria de texto de 8 caracteres.
- BORN_DATE: Fecha de nacimiento, que será un atributo fecha generado de manera aleatoria entre 1960-01-01 y el 2000-01-01.
- WEIGHT: El peso será un atributo numérico con dos decimales entre 50 y 120.
- HEIGHT: La altura será un atributo numérico sin decimales entre 130 y 210.
- GENDER: Género de la persona, que será un atributo de tipo texto que generaremos seleccionándolos de manera aleatoria desde una BBDD.
- CITY: Ciudad de residencia de la persona, que será un atributo de tipo texto que generaremos seleccionándolos de manera aleatoria desde una BBDD.
- ACCOUNT_BALANCE: Poder adquisitivo con el que cuenta la persona. Será un atributo numérico con dos decimales entre 1000 y 120000.

Esta configuración del objeto persona la dejamos cargada en la BBDD en la colección "`conf_objects`" como:

```
{"_id":1, "OBJ_NAME":"PERSON", "ATT_NAME":"NAME", "ATT_CONSTR":"NON_UNIQUE",
"ATT_TYPE":"TEXT", "GENERATION":"BBDD", "BBDD_SOURCE":"data_person_names",
"BBDD_FIELD":"name"}
{"_id":2, "OBJ_NAME":"PERSON", "ATT_NAME":"SURNAME1", "ATT_CONSTR":"NON_UNIQUE",
"ATT_TYPE":"TEXT", "GENERATION":"BBDD", "BBDD_SOURCE":"data_person_surnames",
"BBDD_FIELD":"surname"}
{"_id":3, "OBJ_NAME":"PERSON", "ATT_NAME":"SURNAME2", "ATT_CONSTR":"NON_UNIQUE",
"ATT_TYPE":"TEXT", "GENERATION":"BBDD", "BBDD_SOURCE":"data_person_surnames",
"BBDD_FIELD":"surname"}
{"_id":4, "OBJ_NAME":"PERSON", "ATT_NAME":"ID_DOC", "ATT_CONSTR":"UNIQUE",
"ATT_TYPE":"TEXT", "GENERATION":"RANDOM", "LENGTH":"8"}
{"_id":5, "OBJ_NAME":"PERSON", "ATT_NAME":"BORN_DATE",
"ATT_CONSTR":"NON_UNIQUE", "ATT_TYPE":"DATE", "GENERATION":"RANDOM",
"MIN_DATE":"1960-01-01", "MAX_DATE":"2000-01-01"}
```

```
{"_id":6, "OBJ_NAME":"PERSON", "ATT_NAME":"WEIGHT", "ATT_CONSTR":"NON_UNIQUE",
"ATT_TYPE":"NUMBER", "PRECISION":"2", "RANGE":"50-120", "GENERATION":"RANDOM"}
{"_id":7, "OBJ_NAME":"PERSON", "ATT_NAME":"HEIGHT", "ATT_CONSTR":"NON_UNIQUE",
"ATT_TYPE":"NUMBER", "PRECISION":"0", "RANGE":"130-210", "GENERATION":"RANDOM"}
{"_id":8, "OBJ_NAME":"PERSON", "ATT_NAME":"GENDER", "ATT_CONSTR":"NON_UNIQUE",
"ATT_TYPE":"TEXT", "GENERATION":"BBDD", "BBDD_SOURCE":"data_person_gender",
"BBDD_FIELD":"gender"}
{"_id":9, "OBJ_NAME":"PERSON", "ATT_NAME":"CITY", "ATT_CONSTR":"NON_UNIQUE",
"ATT_TYPE":"TEXT", "GENERATION":"BBDD", "BBDD_SOURCE":"data_person_city",
"BBDD_FIELD":"city"}
{"_id":10, "OBJ_NAME":"PERSON", "ATT_NAME":"ACCOUNT_BALANCE",
"ATT_CONSTR":"NON_UNIQUE", "ATT_TYPE":"NUMBER", "PRECISION":"2", "RANGE":"1000-
120000", "GENERATION":"RANDOM"}
```

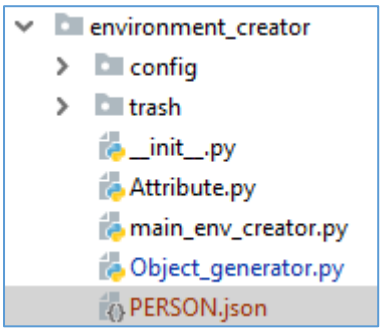

La clase `Object_generator` contiene una función `create_json_with_objects()` que recibe como primer parámetro el nombre del tipo de objeto a crear y de segundo el número de objetos de ese tipo que queremos crear.

3.3. Creacion de los objetos

Creamos un main que genere 10.000 objetos del tipo persona:

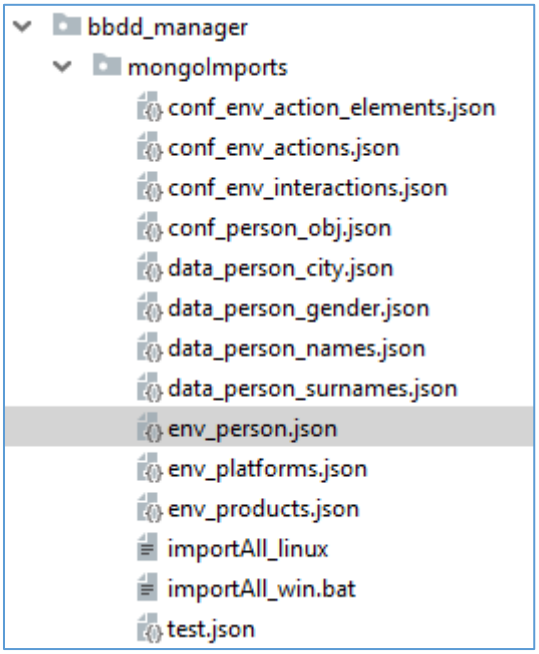
```
from bbdd_manager.Mongo_manager import Mongo_manager
from bbdd_manager import ConfigVariablesBbdd
from environment_creator.Object_generator import Object_generator

# Creamos una conexion a la BBDD
bbdd_connec = Mongo_manager(ConfigVariablesBbdd.env_database)
# Creamos el objeto
ogl = Object_generator(bbdd_connec)
ogl.create_json_with_objects("PERSON",10000)
```

Esto genera un fichero `PERSON.json` dentro del paquete de creación del entorno:



Este fichero los renombramos como `env_person.json` y nos lo llevamos a la ruta:

- bbdd_manager/mongoImports/

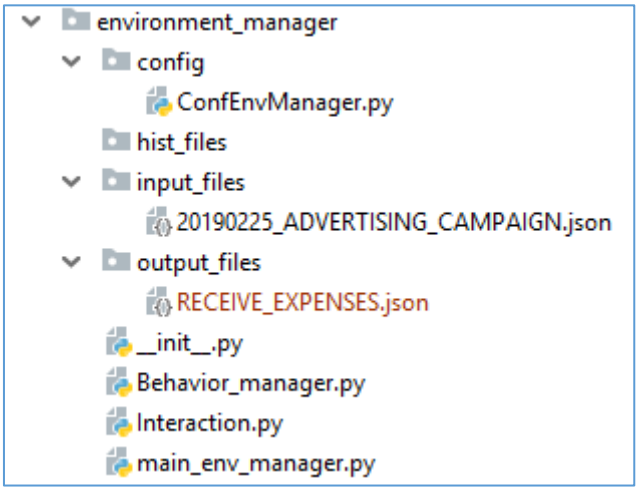


Desde donde lo cargaremos en la BBDD con la siguiente sentencia:

```
C:\MongoDB\Server\3.4\bin\mongoimport --db environmentBBDD --collection
env_person --drop --file
C:\FERNANDO\MasterDS\Proyecto\marketing_de\bbdd_manager\mongoImports\env_
person.json
```

4. Creación de un entorno vivo

La dificultad de trabajar con RL es que requiere de un entorno que responda a las acciones del agente que creemos. El módulo que va a simular el entorno tiene la siguiente estructura:



4.1. Acciones e interacciones (Interaction.py)

Vamos a definir la **acción** del agente de marketing como la suma de un producto a anunciar más la plataforma mediante la que va a ser anunciado. Por lo tanto:

- ACTION = PRODUCT + PLATFORM

Las plataformas disponibles van a ser:

- RADIO
- TV
- NEWSPAPER
- SOCIAL_NETWORKS

Los productos disponibles:

- PARAGUAS
- BAÑADOR
- BIKINI
- GIMNASIO
- DIETISTA
- BALON_BALONCESTO

- DESCAPOTABLE
- BALON_FUTBOL
- PORSCHE

Ahora vamos definir las interacciones con el entorno, que no es más que una breve descripción de como va a ser la comunicación entre el entorno y el agente. Por lo tanto los objetos de tipo `Interaction`, no son más que beans que contienen información. Tenemos dos tipos de interacciones:

- INPUT_INTERACTION = OBJECT_ID + ACTION
- OUTPUT_INTERACTION = OBJECT_ID + REGUARD

Definimos ambas interacciones en la BBDD con los siguientes diccionarios:

La interacción de entrada, la vamos a llamar "`SEND CAMPAIGN`", en ella vamos a indicar que la comunicación se realiza mediante ficheros en formato JSON que se dejarán en el directorio "input_files". También indicamos el nombre del objeto (PERSON) sobre el que se realiza y como referenciar a cada elemento (ID_DOC):

```
{"_id":1, "int_type": "INPUT", "int_name": "SEND_CAMPAIGN"
,"communication_type":"FILE","communication_format":"JSON","communication
_ext":"json","communication_location":"input_files",
"obj_name": "PERSON", "obj_id": "ID_DOC", "obj_source":
"env_person","ACT_NAME": "ADVERTISING_CAMPAIGN", "INTER_CAPACITY": 10000}
```

La interacción de salida, la vamos a llamar "`RECEIVE EXPENSES`", en ella vamos a indicar que la comunicación se realiza mediante ficheros en formato JSON que se dejarán en el directorio "output_files":

```
{"_id":2, "int_type": "OUTPUT", "int_name": "RECEIVE_EXPENSES"
,"communication_type":"FILE","communication_format":"JSON","communication
_ext":"json","communication_location":"output_files"}
```

4.2. Gestor del comportamiento del entorno

El comportamiento del entorno es gestionado por un objeto de la clase `Behaviour_manager`, que a su vez llamará al método que gestiona la comunicación mediante ficheros `file_action_behaviour_manager()`. El comportamiento se va a

harcodear de la siguiente manera, el método `behaviour()` va a desglosar la acción en el producto y la plataforma para validarlos por separado contra el objeto.

El comportamiento de los clientes con respecto a la plataforma va ser el siguiente:

- RADIO: Acierto para aquellos clientes nacidos entre 1960 y 1970.
- TV: Acierto para aquellos clientes nacidos entre 1970 y 1980.
- NEWSPAPER: Acierto para aquellos clientes nacidos entre 1980 y 1990.
- SOCIAL_NETWORKS: Acierto para aquellos clientes nacidos entre 1990 y 2000.

El comportamiento de los clientes con respecto a los productos va ser el siguiente:

- PARAGUAS: Acierto para aquellos clientes que vivan en Santiago, Oviedo o Santander.
- BAÑADOR: Acierto para aquellos clientes del género masculino que vivan en Cádiz, Marbella o Málaga.
- BIKINI: Acierto para aquellos clientes del género femenino que vivan en Cádiz, Marbella o Málaga.
- GIMNASIO: Acierto para aquellos clientes con un peso inferior a 90 kilos y que residan en Salamanca o Segovia.
- DIETISTA: Acierto para aquellos clientes con un peso superior a 90 kilos y que residan en Salamanca o Segovia.
- BALON_BALONCESTO: Acierto para aquellos clientes con una altura mayor a 190 con ingresos inferiores a 90.000 y residentes en Madrid.
- DESCAPOTABLE: Acierto para aquellos clientes con una altura mayor a 190 con ingresos superiores a 90.000 y residentes en Madrid.
- BALON_FUTBOL: Acierto para aquellos clientes con una altura menor a 190 con ingresos inferiores a 90.000 y residentes en Madrid.
- PORSCHE: Acierto para aquellos clientes con una altura menor a 190 con ingresos superiores a 90.000 y residentes en Madrid.

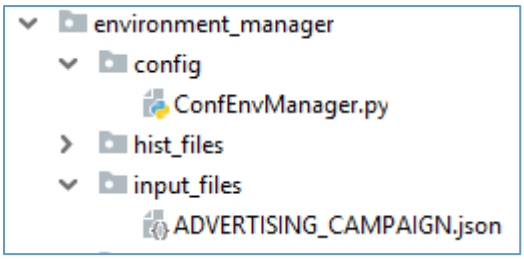
4.3. Prueba de ejecución

Creamos un fichero de entrada con 9 casos de prueba:

```
{"PERSON":{"ID_DOC": "UGQrzdAL"},"ADVERTISING_CAMPAIGN":
{"PRODUCT":{"PRODUCT_NAME": "PARAGUAS","PRICE": 10}
,"PLATFORM":{"PLATFORM_NAME": "RADIO"}}}
{"PERSON":{"ID_DOC": "OHlEECpP"},"ADVERTISING_CAMPAIGN":
{"PRODUCT":{"PRODUCT_NAME": "BANADOR","PRICE": 10}
,"PLATFORM":{"PLATFORM_NAME": "TV"}}}
```

```
{"PERSON":{"ID_DOC": "vwwwJvMV"},"ADVERTISING_CAMPAIGN":
{"PRODUCT":{"PRODUCT_NAME": "BIKINI","PRICE": 10}
,"PLATFORM":{"PLATFORM_NAME": "NEWSPAPER"}}}
{"PERSON":{"ID_DOC": "BqfvsoIX"},"ADVERTISING_CAMPAIGN":
{"PRODUCT":{"PRODUCT_NAME": "GIMNASIO","PRICE": 10}
,"PLATFORM":{"PLATFORM_NAME": "SOCIAL_NETWORKS"}}}
{"PERSON":{"ID_DOC": "ghuSvoqL"},"ADVERTISING_CAMPAIGN":
{"PRODUCT":{"PRODUCT_NAME": "DIETISTA","PRICE": 10}
,"PLATFORM":{"PLATFORM_NAME": "RADIO"}}}
{"PERSON":{"ID_DOC": "wqZRfVzn"},"ADVERTISING_CAMPAIGN":
{"PRODUCT":{"PRODUCT_NAME": "BALON_BALONCESTO","PRICE": 10}
,"PLATFORM":{"PLATFORM_NAME": "TV"}}}
{"PERSON":{"ID_DOC": "OstlKWUm"},"ADVERTISING_CAMPAIGN":
{"PRODUCT":{"PRODUCT_NAME": "DESCAPOTABLE","PRICE": 10}
,"PLATFORM":{"PLATFORM_NAME": "NEWSPAPER"}}}
{"PERSON":{"ID_DOC": "gLTvYabg"},"ADVERTISING_CAMPAIGN":
{"PRODUCT":{"PRODUCT_NAME": "BALON_FUTBOL","PRICE": 10}
,"PLATFORM":{"PLATFORM_NAME": "SOCIAL_NETWORKS"}}}
{"PERSON":{"ID_DOC": "hiykIWgZ"},"ADVERTISING_CAMPAIGN":
{"PRODUCT":{"PRODUCT_NAME": "PORSCHE","PRICE": 10}
,"PLATFORM":{"PLATFORM_NAME": "RADIO"}}}
```

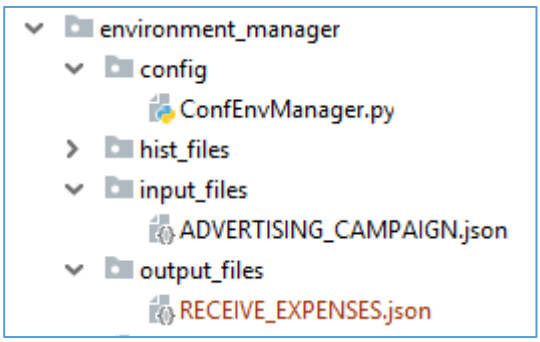

Dejamos el fichero de entrada en la ruta "environment_manager /input_files":



Ejecutamos el siguiente script de Python:

```
from bbdd_manager.Mongo_manager import Mongo_manager
from bbdd_manager import ConfigVariablesBbdd
from environment_manager.Interaction import Interaction
from environment_manager.Behavior_manager import Behavior_manager

# Creamos una conexion a la BBDD
bbdd_connec = Mongo_manager(ConfigVariablesBbdd.env_database)
# Ejemplo de uso behaviour
bm1 = Behavior_manager(bbdd_connec)
bm1.action_behavior_manager("SEND_CAMPAIGN","RECEIVE_EXPENSES")
```

Al finalizar el script, se genera el fichero de respuesta en el directorio "environment_manager/output_files":



Si lo abrimos, podemos ver el resultado obtenido:


```
{"ID_DOC": "UGQrzdAL", "EXPENSE": 10}
{"ID_DOC": "OHlEECpP", "EXPENSE": 10}
{"ID_DOC": "vwwwJvMV", "EXPENSE": 10}
{"ID_DOC": "BqfvsoIX", "EXPENSE": 0}
{"ID_DOC": "ghuSvoqL", "EXPENSE": 0}
{"ID_DOC": "wqZRfVzn", "EXPENSE": 10}
{"ID_DOC": "OstlKWUm", "EXPENSE": 10}
{"ID_DOC": "gLTvYabg", "EXPENSE": 10}
{"ID_DOC": "hiykIWgZ", "EXPENSE": 10}
```


De los 9 clientes con los que interactuado:

- 7 de ellos han reaccionado positivamente comprando el producto (gasto = 10).
- 2 de ellos no han reaccionado (gasto = 0).

5. Marketing decision engine (agent)

Mi propuesta va a ser resolver este problema mediante Reinforcement Learning. En vez de emplear alguna funcionalidad ya implementada en alguna librería, vamos a estudiar algún algoritmo, e implementarlo desde cero.

5.1. Reinforcement Learning

RL se basa en el aprendizaje mediante la interacción con un entorno. Por lo tanto está relacionado con el aprendizaje natural que conocemos, como aprenden los animales. La conexión con un entorno nos permite obtener grandes cantidades de información sobre la "causa" y el "efecto". Esto nos enseña las consecuencias de la acciones que podemos gestionar luego para alcanzar nuestros propósitos.

Por lo tanto en un problema de RL, tenemos un entorno y agente que aplicará acciones sobre el mismo y recibirá recompensas. El agente deberá aprender que acciones maximizan las recompensas. Uno de los mayores retos en RL es el de balancear correctamente la exploración vs explotación. La explotación implica que el agente utiliza acciones que ya conoce para obtener recompensa y la exploración implica probar acciones nuevas en busca de mejores recompensas.

Dentro de los múltiples algoritmos que se estudian en RL, vamos estudiar el más básico y sencillo de ellos, el "Multi-arm Bandits".

5.2. Multi-arm Bandits

Estamos hablando de un problema sencillo de RL en el que consideramos un feedback evaluativo (el feedback depende enteramente de la acción tomada) y un entorno no varía (sólo permite una situación).

En el problema "n-Armed Bandit", te enfrentas de manera repetida con una elección a tomar sobre las "n" posibles. Después de cada elección se recibe una recompensa numérica que depende de la acción tomada. Como siempre, nuestro objetivo es el de maximizar la recompensa a lo largo de una serie de intentos.

La manera más sencilla de estimar el valor de una acción, es mediante el cálculo de una media:

$$Q_t(\mathrm{a}) = \frac{R_1 + R_2 + \cdots + R_{Nt(a)}}{N_t(a)}$$

Donde $Q_t(\mathrm{a})$ es el valor estimado en la interacción "t", $N_t(a)$ es el número de veces que la acción "a" ha sido seleccionado antes de la interacción "t" y R_x las recompensas recibidas hasta ese momento.

Una vez que hemos visto como estimar las acciones, ahora hay que ver como influye esto en la selección de las mismas. La norma más sencilla es escoger la acción con el mayor valor estimado:

$$A_t = arg\max_a Q_t(a)$$

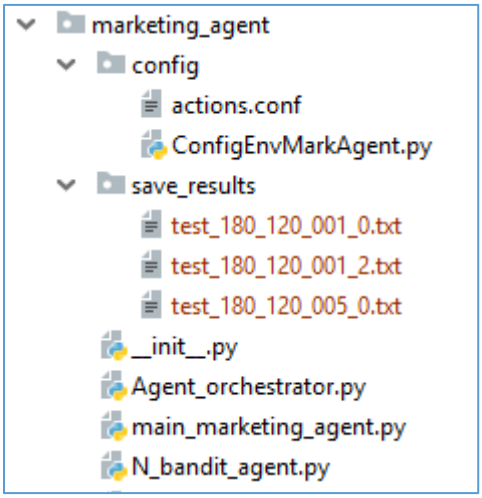
A esto es lo que se llama "método de selección avaricioso", que siempre explota el conocimiento actual para recibir la máxima recompensa inmediata.

Una alternativa mejor y sencilla consiste simplemente en comportarse de manera avariciosa la mayor parte del tiempo, pero de vez en cuando una pequeña probabilidad "ε" se realiza una selección aleatoria de acción. A este método se le llama "ε-greedy".

Con estos simples conceptos vamos a intentar implementar nuestro agente de marketing.

6. Creacion del Decisión Engine

El módulo que va a simular el motor de decisión de marketing es el siguiente:



6.1. Agente (N_bandit_agent.py)

Tenemos que analizar como vamos a implementar la estimación del valor de una acción, que como definimos anteriormente se calcula como:

$$Q_t(\mathrm{a}) = \frac{R_1 + R_2 + \cdots + R_{Nt(a)}}{N_t(a)}$$

Implementar esa fórmula tal y como está expresada, no tienen mucho sentido desde el punto de vista computacional, dado que nos exigiría guardar todas las recompensas recibidas ($R_1 + R_2 + \cdots + R_{Nt(a)}$) en las k iteraciones. Por ello vamos a desarrollar un poco más esta fórmula:

$$Q_{k+1} = \frac{R_1 + R_2 + \cdots + R_k}{\mathrm{k}} = \frac{1}{k}\sum_{i=1}^{k} R_i = \frac{1}{k}\left(R_k + \sum_{i=1}^{k-1} R_i\right) = \frac{1}{k}(R_k + (k-1)Q_k)$$

$$= \frac{1}{k}\left(R_k + (kQ_k - Q_k)\right) = Q_k + \frac{1}{k}(R_k - Q_k)$$

Lo que significa, que para calcular el nuevo valor de estimación, solamente necesitamos guardar dos valores, la última estimación y la iteración por la que vamos.

Por lo tanto en la inicialización de agente le vamos a pasar lo siguientes atributos:

- Environment: Objeto del entorno con el que debe interactuar. En este caso, una persona.
- N_actions: Una lista con las acciones disponibles.
- Error: Porcentaje de error en el cual obligamos al agente a explorar nuevas acciones.

En la inicialización, vamos a generar una tabla que recoja para cada acción su estimación de valor (Q_k) cuantas veces se ha empleado (k) y una variable (alpha) para un posible futuro uso. Dado que puede haber un gran número de agentes, no vamos a utilizar una BBDD para mantener esta tabla, porque si no el performance podría llegar a ser muy pobre. En sustitución a una BBDD, nuestra tabla va a ser un DataFrame de Pandas con la siguiente estructura:

Index	Action	Est. Val. (Q_k)	Iter. (k)	alpha
1	Action 1	0	1	0
2	Action 2	0	1	0
3	Action 3	0	1	0
…	…	…	…	…

Antes de la primera iteración, no tenemos valor estimado, y debemos definir por lo tanto como vamos a inicializar nuestras variables. Vamos a inicializar todos los valores estimados como cero (Q_k=0) en una primera iteración (k=1) imaginaria.

Creamos un método "select_action()" que escoge una acción y la devuelve en formato json junto con el ID del entorno asociado (una persona). Este método con probabilidad (1-ε) tenderá un comportamiento avaricioso (**explotación**) y por lo tanto escogerá la acción que tenga un mayor valor estimado:

$$A_t = arg\max_a Q_t(a)$$

En caso de encontrarnos con varias posibles acciones que empaten por tener todas ellas el valor máximo del conjunto, seleccionaremos una al azar. Este método con probabilidad (ε) tenderá un comportamiento totalmente aleatorio (**exploración**) en la selección de la acción.

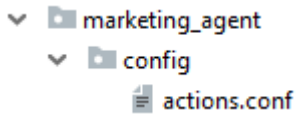
Creamos otro método “receive_feedback(reward)” que se emplea para informarle al agente la recompensa recibida tras procesar su acción. Este método actualiza dos variables:

- $k = k + 1$
- $Q_{k+1} = Q_k + \frac{1}{k}(R_k - Q_k)$

6.2. Orquestador (Agent_orchestrator.py)

Necesitamos un orquestador que gestione la interacción entre los agentes y el entorno.

Lo primero que va a hacer este orquestador es cargar las posibles acciones en su inicialización. El listado de acciones lo hemos dejado en un fichero de configuración:



Y contiene todas las combinaciones de producto y plataforma. Es decir 9 productos con 4 plataformas posibles nos generan 36 acciones posibles :

```
{"PRODUCT":{"PRODUCT_NAME": "PARAGUAS","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "RADIO"}}}
{"PRODUCT":{"PRODUCT_NAME": "BANADOR","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "RADIO"}}}
{"PRODUCT":{"PRODUCT_NAME": "BIKINI","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "RADIO"}}}
{"PRODUCT":{"PRODUCT_NAME": "GIMNASIO","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "RADIO"}}}
{"PRODUCT":{"PRODUCT_NAME": "DIETISTA","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "RADIO"}}}
{"PRODUCT":{"PRODUCT_NAME": "BALON_BALONCESTO","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "RADIO"}}}
{"PRODUCT":{"PRODUCT_NAME": "DESCAPOTABLE","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "RADIO"}}}
{"PRODUCT":{"PRODUCT_NAME": "BALON_FUTBOL","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "RADIO"}}}
{"PRODUCT":{"PRODUCT_NAME": "PORSCHE","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "RADIO"}}}
{"PRODUCT":{"PRODUCT_NAME": "PARAGUAS","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "TV"}}}
{"PRODUCT":{"PRODUCT_NAME": "BANADOR","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "TV"}}}
{"PRODUCT":{"PRODUCT_NAME": "BIKINI","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "TV"}}}
{"PRODUCT":{"PRODUCT_NAME": "GIMNASIO","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "TV"}}}
{"PRODUCT":{"PRODUCT_NAME": "DIETISTA","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "TV"}}}
{"PRODUCT":{"PRODUCT_NAME": "BALON_BALONCESTO","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "TV"}}}
{"PRODUCT":{"PRODUCT_NAME": "DESCAPOTABLE","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "TV"}}}
{"PRODUCT":{"PRODUCT_NAME": "BALON_FUTBOL","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "TV"}}}
{"PRODUCT":{"PRODUCT_NAME": "PORSCHE","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "TV"}}}
{"PRODUCT":{"PRODUCT_NAME": "PARAGUAS","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "NEWSPAPER"}}}
{"PRODUCT":{"PRODUCT_NAME": "BANADOR","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "NEWSPAPER"}}}
{"PRODUCT":{"PRODUCT_NAME": "BIKINI","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "NEWSPAPER"}}}
{"PRODUCT":{"PRODUCT_NAME": "GIMNASIO","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "NEWSPAPER"}}}
```

```
{"PRODUCT":{"PRODUCT_NAME": "DIETISTA","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "NEWSPAPER"}}}
{"PRODUCT":{"PRODUCT_NAME": "BALON_BALONCESTO","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "NEWSPAPER"}}}
{"PRODUCT":{"PRODUCT_NAME": "DESCAPOTABLE","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "NEWSPAPER"}}}
{"PRODUCT":{"PRODUCT_NAME": "BALON_FUTBOL","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "NEWSPAPER"}}}
{"PRODUCT":{"PRODUCT_NAME": "PORSCHE","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "NEWSPAPER"}}}
{"PRODUCT":{"PRODUCT_NAME": "PARAGUAS","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "SOCIAL_NETWORKS"}}}
{"PRODUCT":{"PRODUCT_NAME": "BANADOR","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "SOCIAL_NETWORKS"}}}
{"PRODUCT":{"PRODUCT_NAME": "BIKINI","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "SOCIAL_NETWORKS"}}}
{"PRODUCT":{"PRODUCT_NAME": "GIMNASIO","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "SOCIAL_NETWORKS"}}}
{"PRODUCT":{"PRODUCT_NAME": "DIETISTA","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "SOCIAL_NETWORKS"}}}
{"PRODUCT":{"PRODUCT_NAME": "BALON_BALONCESTO","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME":
"SOCIAL_NETWORKS"}}}
{"PRODUCT":{"PRODUCT_NAME": "DESCAPOTABLE","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME":
"SOCIAL_NETWORKS"}}}
{"PRODUCT":{"PRODUCT_NAME": "BALON_FUTBOL","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME":
"SOCIAL_NETWORKS"}}}
{"PRODUCT":{"PRODUCT_NAME": "PORSCHE","PRICE": 10} ,"PLATFORM":{"PLATFORM_NAME": "SOCIAL_NETWORKS"}}}
```


Esta clase va a tener un único método principal que se va a llamar "run()" y va a realizar los siguientes pasos:

1. El primer paso consiste en asociar una persona del entorno a cada uno de los "n" agentes.
2. Luego por cada iteración:
    a. Pedimos a cada agente su selección de acción y la guardamos en un fichero.
    b. Le pedimos al entorno que procese el fichero de acciones.
    c. Le comunicamos a cada agente la recompensa recibida por parte del entorno tras su acción.

6.3. Primera ejecución y análisis

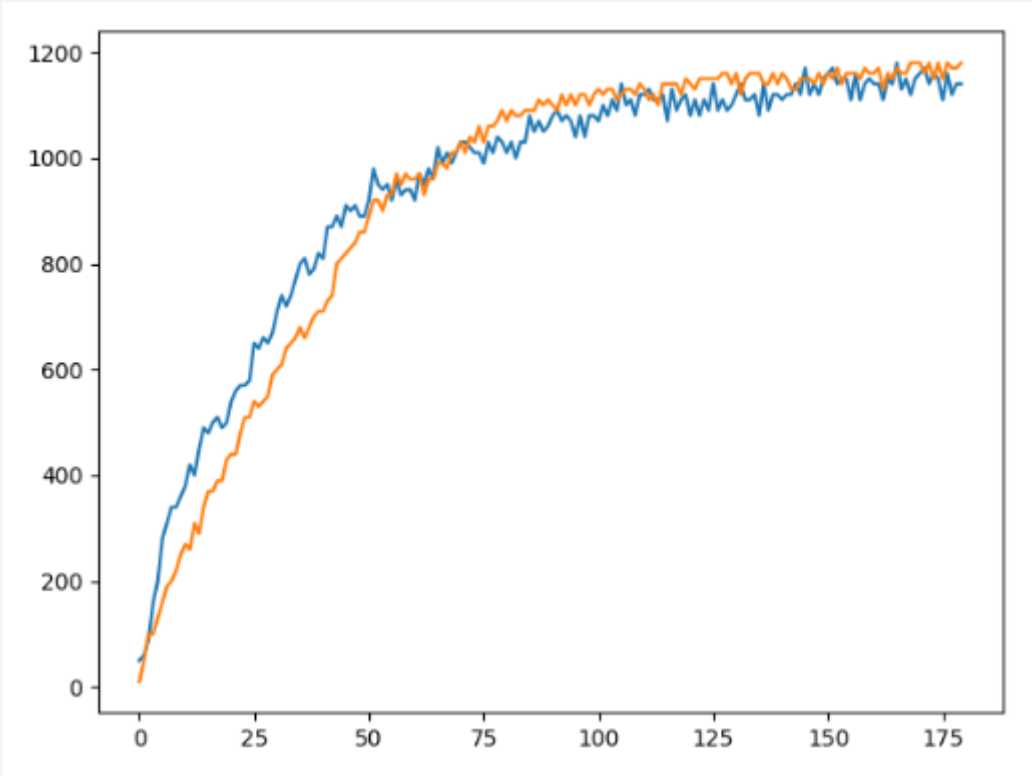
Preparamos dos casos de prueba, ambos los establecemos en 180 iteraciones con 120 agentes, pero uno de ellos lo configuramos un error del 1% y al otro del 5%:

```
# Creamos una conexion a la BBDD
bbdd_connec = Mongo_manager(ConfigVariablesBbdd.env_database)

# Creamos una accion
test_orch = Agent_orchestrator(bbdd_connec)
# CASE 1 - 180 Iterations, 120 number of agents, 0.05 error
list_of_results = test_orch.run(180,120,0.05)
save.from_list_to_file(list_of_results,'save_results/test_180_120_005_0.t
xt')
x_axis = range(len(list_of_results))
plt.plot(list_of_results)
# CASE 2 - 180 Iterations, 120 number of agents, 0.01 error
list_of_results1 = test_orch.run(180,120,0.01)
save.from_list_to_file(list_of_results,'save_results/test_180_120_001_0.t
xt')
x_axis1 = range(len(list_of_results1))
```

```
plt.plot(list_of_results1)
```

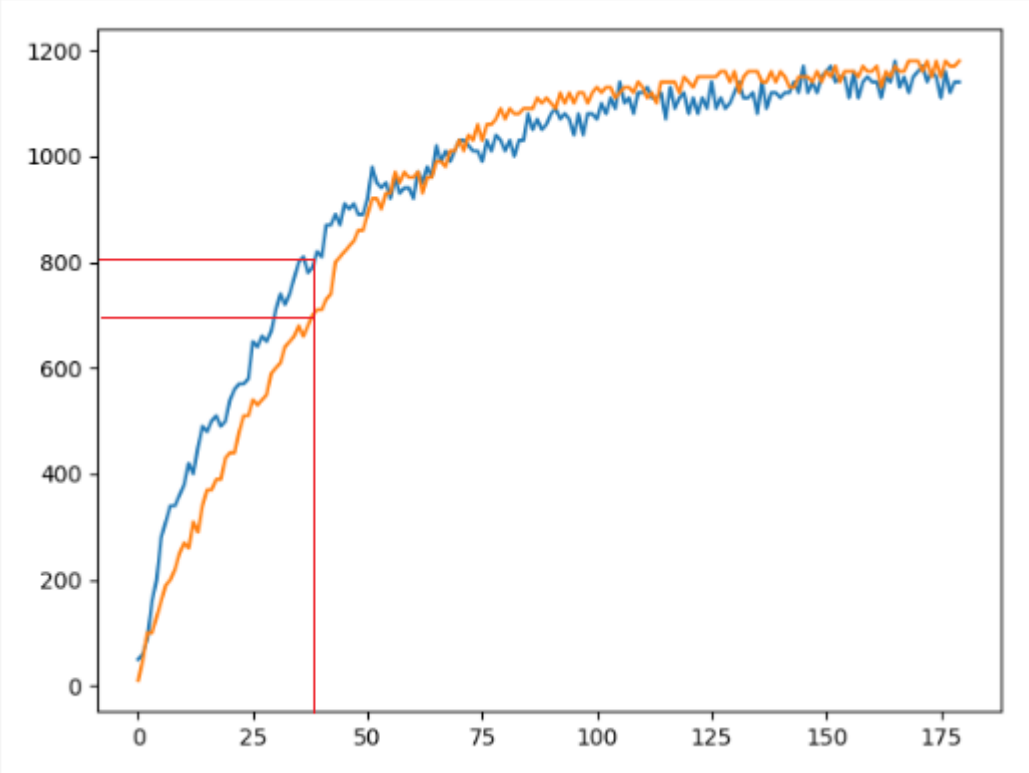
Si imprimimos el total de las recompensas recibidas en cada iteración podemos ver un hecho lógico que es el siguiente:

- El caso de test con agentes cuyo error es del 5% (curva azul), como exploran más, al principio llegan antes a obtener una mayor recompensa.
- Sin embargo, el caso de test con error de 1% (curva naranja) aunque tarda más en llegar al máximo de recompensa, como explora menos finalmente se mantiene en mayores niveles.



6.4. Resultado inesperado y mejora

Hay un detalle que llama la atención de las gráficas, tal y como hemos visto disponemos de 36 posibles acciones, y dado que estamos empleando un entorno estacionario (siempre responde igual) la gran mayoría de agentes debieran encontrar la acción correcta este número de acciones. Sin embargo si nos fijamos en la interacción 36, a penas hemos conseguido un poco más de la mitad de aciertos del total posible:



Analizamos porqué está ocurriendo esto y vemos que se debe a la inicialización a cero del valor estimado (Q_k=0). Al haber escogido este valor, equivocarse no penaliza, si no que devuelve la acción escogida de nuevo con Q_k=0 y por lo tanto puede volver a ser elegible hasta encontremos una acción que genere recompensa.

Vamos a basarnos en siguiente ejemplo, si todas las estimaciones son cero, cuando escogemos una acción y nos equivocamos, la devolvemos de nuevo con Q_k=0 y por lo tanto al seleccionar en la siguiente interacción las acciones con máximo valor estimado, vuelve a poder ser elegible (a pesar de que sabemos que no es acertada):

Index	Action	Est. Val. (Q_k)	Iter. (k)	alpha
1	Action 1	0	1	0
2	Action 2	0	1	0
3	Action 3	0	1	0
…	…	…	…	…

Esto nos hace ver la importancia de elegir un valor positivo para inicializar el valor estimado (Q_k). Si en vez de cero empleamos por ejemplo Q_k=2 entonces al seleccionar una acción equivocada (reward = 0) el recalculo de la estimación:

- $Q_{k+1} = Q_k + \frac{1}{k}(R_k - Q_k) = 2 + \frac{1}{1}(0 - 2) = 0$

reduce su valor con respecto al inicial, y por lo tanto ya no es seleccionable con respecto al resto de acciones que se mantienen con el valor inicial.

Vamos suponer que partimos de un valor estimado positivo por ejemplo Q_k=2:

Index	Action	Est. Val. (Q_k)	Iter. (k)	alpha
1	Action 1	2	1	0
2	Action 2	2	1	0
3	Action 3	2	1	0
…	…	…	…	…

Si escogemos por ejemplo la "Action 2" y fallamos (reward = 0), vamos a recalcular el valor estimado a cero (Q_k=0):

Index	Action	Est. Val. (Q_k)	Iter. (k)	alpha
1	Action 1	2	1	0
2	Action 2	0	2	0
3	Action 3	2	1	0
…	…	…	…	…

Y por lo tanto en la siguiente iteración cuando seleccionemos las acciones de mayor valor estimado (Q_k), vamos a descartar esa acción "Action 2".

Por ello vamos a incluir tanto en el agente, como en el orquestador la posibilidad de establecer el valor inicial que toma Q_k.

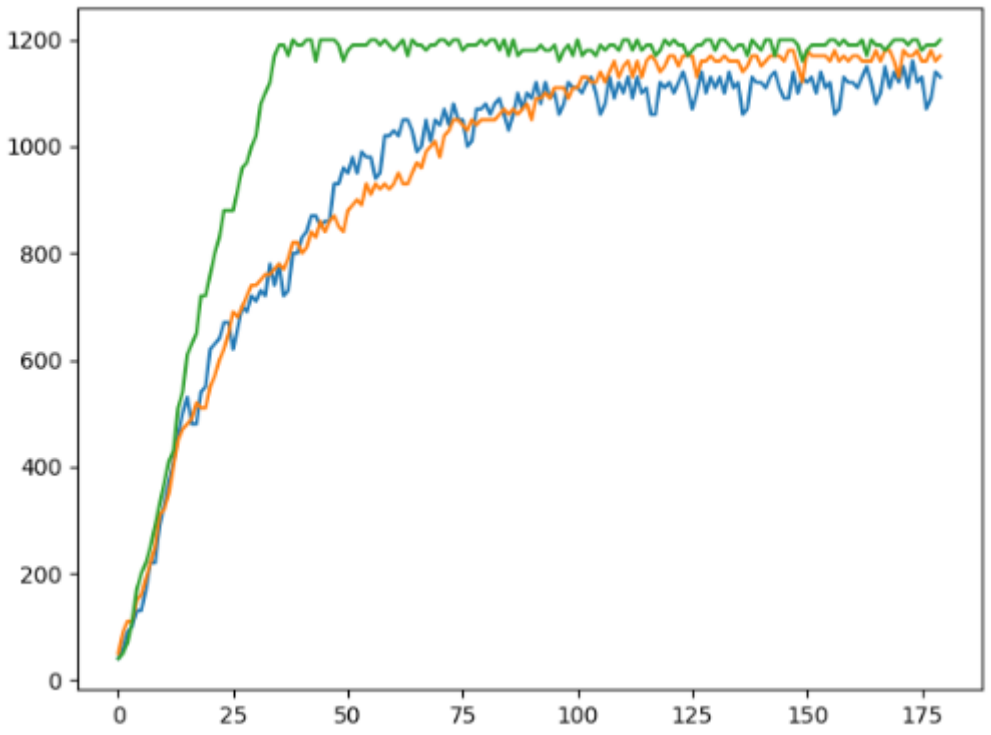
6.5. Segunda ejecución y análisis

Junto a los dos casos anteriores, preparamos un tercero en el cual establecemos el valor de estimación inicial a 2:

```
# Creamos una conexion a la BBDD
bbdd_connec = Mongo_manager(ConfigVariablesBbdd.env_database)

# Creamos una accion
test_orch = Agent_orchestrator(bbdd_connec)
```

```
# CASE 1 - 180 Iterations, 120 number of agents, 0.05 error, 0
initial_estimate
list_of_results = test_orch.run(180,120,0.05,0)
save.from_list_to_file(list_of_results,'save_results/test_180_120_005_0.t
xt')
x_axis = range(len(list_of_results))
plt.plot(list_of_results)
# CASE 2 - 180 Iterations, 120 number of agents, 0.01 error, 0
initial_estimate
list_of_results1 = test_orch.run(180,120,0.01,0)
save.from_list_to_file(list_of_results,'save_results/test_180_120_001_0.t
xt')
x_axis1 = range(len(list_of_results1))
plt.plot(list_of_results1)
# CASE 3 - 180 Iterations, 120 number of agents, 0.05 error, 2
initial_estimate
list_of_results2 = test_orch.run(180,120,0.01,2)
save.from_list_to_file(list_of_results,'save_results/test_180_120_001_2.t
xt')
x_axis2 = range(len(list_of_results2))
plt.plot(list_of_results2)
```

Efectivamente, si revisamos ahora el la nueva curva de recompensas generada tras el cambio en la inicialización (curva verde), podemos comprobar como ahora tras 36 iteraciones ya se ha alcanzado el máximo de recompensa:

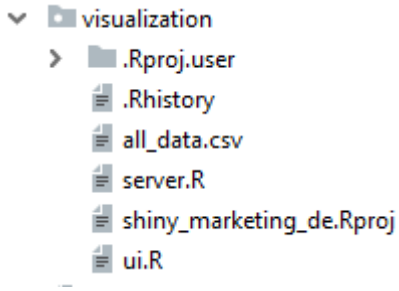


Con esto damos por correcta la implementación del algoritmo n-bandit.

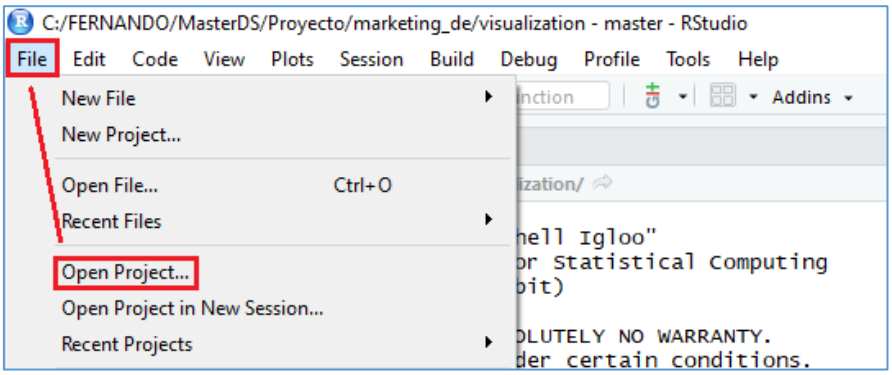
7. Módulo de visualización

Dado el enfoque de este proyecto, se trata más bien de un proceso batch con cierto coste computacional, por lo que no le veo mucho sentido la inclusión de un front (que encajaría más en una aplicación online con posibilidad de interacción con el usuario, requiriendo por lo tanto un rápida respuesta). En este caso para cubrir este apartado, lo que vamos a hacer, es generar un dataset con varias curvas pre-calculadas y montar un front en shiny que premita visionar varias a la vez.

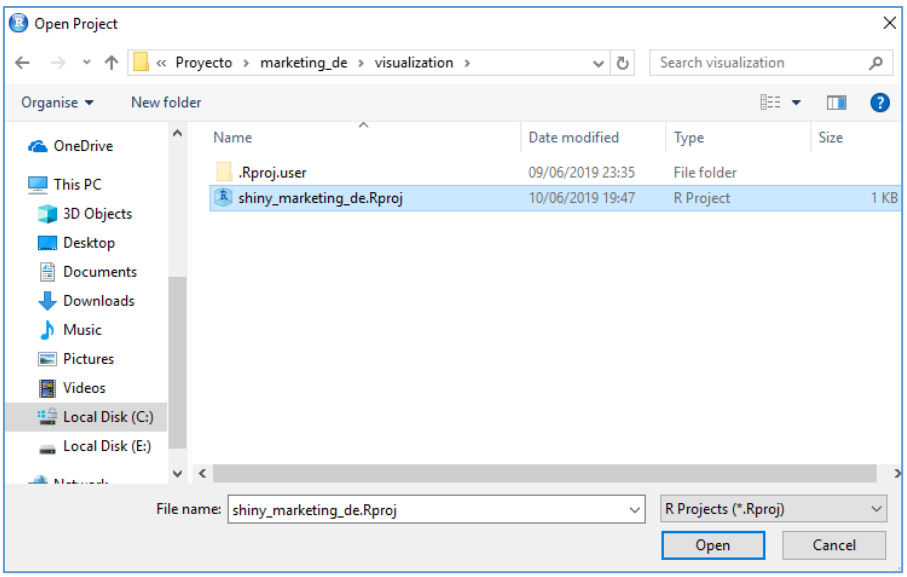
Este módulo se encuentra en el directorio "visualization" del proyecto:



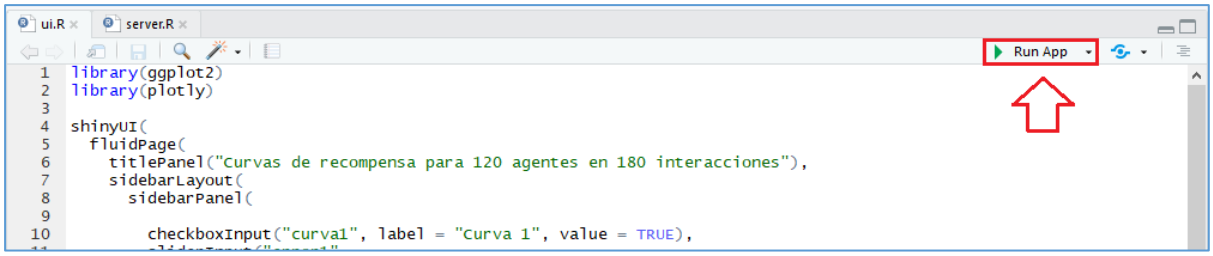
Podemos abrirlo por ejemplo con un R-Studio, hacemos click en la barra principal primero sobre "File" y luego sobre "Open Project…":



Nos vamos a la ruta de "visualization" y abrimos el fichero "shiny_marketing_de.Rproj":

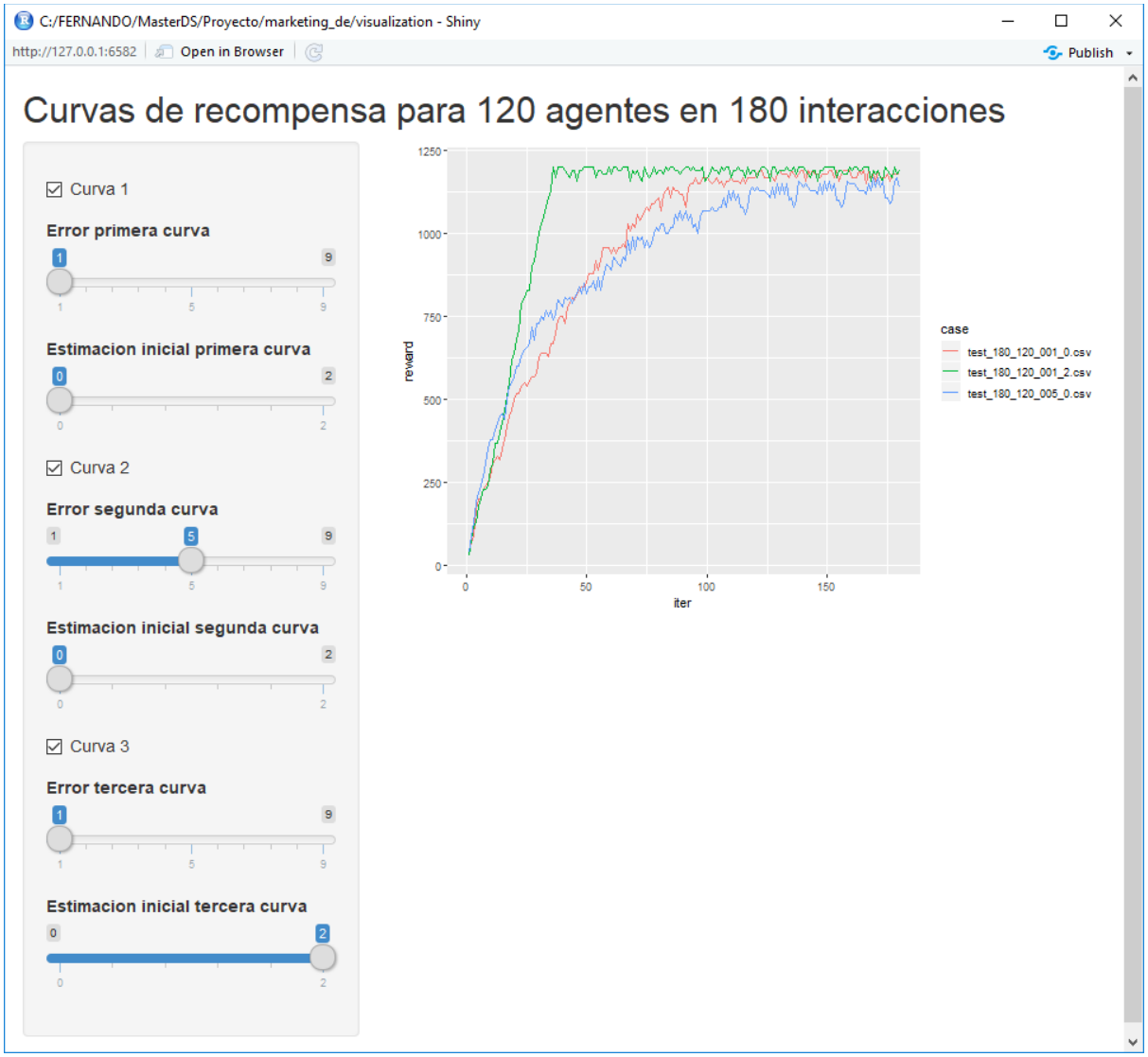


Hacemos click sobre el icono "Run App":



Esto nos abre una ventana en la que tenemos un menú interactivo a la izquierda y una gráfica a la derecha. Podemos visualizar un máximo de 3 curvas a la vez (mínimo de 0) empleando el check-box y por cada curva podemos establecer su configuración en función de 2 variables:

- El error permitido (exploración) que puede ser 1, 5 o 9%.
- Valor de la estimación inicial (Q_k) que puede ser 0 o 2.



Esta aplicación nos permite comparar 2 o 3 curvas de nuestra elección entre las posibilidades dadas.