TIPOLOGÍA Y CICLO DE VIDA DE LOS DATOS

PRÁCTICA 1

¿CÓMO PODEMOS CAPTURAR LOS DATOS DE LA WEB?

CÁCERES BERRAQUERO, ALMIR
HERNÁNDEZ SALMERÓN, ALEJANDRO

MÁSTER EN CIENCIA DE DATOS
2022/2023 - A

Índice

1. Contexto

El sitio web escogido para la recopilación de información es la página web de la plataforma de Steam: https://store.steampowered.com

Steam es una **plataforma de distribución digital de videojuegos, software y demás contenido multimedia** perteneciente a la compañía Valve Corporation. Steam permite la compra de videojuegos así como su uso en línea a través de la propia plataforma.En los últimos años Steam se ha alzado como una de las más importantes plataformas de juegos en línea, con cifras del orden de millones de jugadores activos en todo el mundo. De hecho, Steam dispone de juegos en más de 25 idiomas, y es capaz de mantener a miles de usuarios continuos en línea, siendo su récord de casi 25 millones de jugadores simultáneos.

Esta popularidad ha despertado el interés en los desarrolladores de videojuegos de todo el mundo, consiguiendo que lancen sus productos en la plataforma.
Esto además, **ha favorecido en la última década al auge de los videojuegos "Indies"**, los cuale son juegos producidos por uno o varios desarrolladores. La plataforma ha facilitado su distribución entre los usuarios consiguiendo un mayor alcance para este tipo de producto, que por otro lado, no sería capaz de darse a conocer sin una gran inversión previa. Esto ha llevado a la plataforma a disponer de más de 50000 juegos y aplicaciones en su catálogo.

Esta gran cantidad de juegos significa una gigantesca cantidad de información disponible en la plataforma, perfecta para realizar tareas de scraping en su página web. De esta página es posible **extraer información referente a los videojuegos** como puede ser título, género, descripción, fecha de lanzamiento, precio y ofertas entre otros. También es posible obtener información sobre las reseñas que los jugadores hacen sobre los videojuegos e incluso información de los propios usuarios, como historial de compras o juegos favoritos.

Esta información puede utilizarse para hacer análisis de mercado y tendencias, siendo una fuente de información muy importante para los desarrolladores, ya que posibilita el enfoque del desarrollo en tendencias concretas del mercado.
Otra gran utilidad del scraping en esta plataforma está destinada a los jugadores. Hoy en día los consumidores están muy influenciados por la opinión de otros clientes a la hora de comprar un producto. Como es posible **extraer información de las reseñas, y puntuaciones de los clientes**, se puede valorar la compra de un videojuego con dependencia del resultado que refleje este tipo de información.

Se puede decir que el uso del scraping en esta plataforma tiene un gran valor informativo, ya sea para los consumidores de contenido de la plataforma como para los interesados en la venta de estos.

2. Título del dataset

El título del dataset hace referencia al contenido de este.

El contenido de este contiene información extraída de la página web de Steam, siendo un repositorio de información sobre videojuegos para PC. Por ello se ha escogido un título que hace referencia al contenido de este.

Título → steam_games.csv

---

3. Descripción del dataset

El dataset se ha creado con el objetivo de **describir la información disponible en cada videojuego** que existe en la página web de Steam. Como ya se ha mencionado, Steam es la plataforma online de videojuegos más grande que existe actualmente, por lo que la información extraída de esta página proporciona una gran base de datos acerca de los videojuegos. Dispone de la información pertinente a estos, y los datos que lo describen, como es el título, desarrollador, editor y género. También están los **datos técnicos** de este como son los idiomas en los que está disponible, el precio, las ofertas e incluso la puntuación que los usuarios ponen al videojuego, puntuación de metacritic.

Básicamente, consiste en una base de datos de videojuegos existentes en la plataforma Steam con información descriptiva sobre estos.

---

4. Representación gráfica

Se ha utilizado scrapy como framework de scraping para el desarrollo de la práctica.
Los componentes principales que participan en esta interacción son los siguientes:

1. **Spyder:** Es el componente principal de Scrapy. Se encarga de definir el método de extracción de los datos de una página web. Esto define las URLs que se han de procesar, los datos que se van a extraer.

2. **Scrapy Engine:** El *engine* o motor es el encargado de coordinar el flujo de datos entre todos los componentes en el proceso de la extracción de los datos de las solicitudes que recibe de spyder. Este se encarga de enviar estas solicitudes que

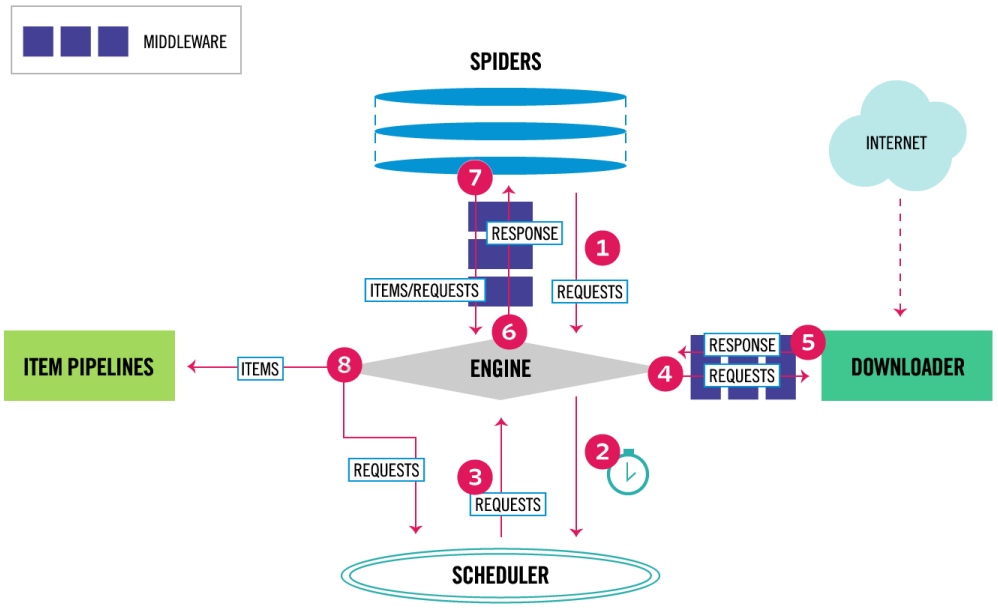
recibe de spyder a los downloaders y gestionar los eventos que se generan en el proceso de descarga.

3. **Downloader:** Se encarga de descargar las las páginas en función al orden definido en las request a través del motor y enviarlas de nuevo a este.

4. **Item Pipelines:** Se encarga de procesar los elementos una vez que han sido extraídos (o scrapeados) por las arañas. Las tareas típicas incluyen la limpieza, la validación y la persistencia (como almacenar el elemento en una base de datos).

5. **Schedule:** Se encarga de gestionar las peticiones del motor y ponerla en cola para su posterior utilización, asegurando que estas peticiones se procesen en un orden adecuado evitando el envío de solicitudes duplicadas.

A continuación se muestra la representación gráfica del proceso que lleva scrapy para la extracción de los datos.



En esta representación se puede ver el flujo de Scrapy, que es controlado por el Engine de ejecución. A continuación se detallan los pasos del proceso que sigue Scrapy:

1. El Engine recibe las solicitudes iniciales de rastreo de las Spiders.
2. El Engine programa las peticiones en el Scheduler y solicita las siguientes peticiones que se van a rastrear.
3. El Scheduler devuelve las siguientes peticiones en orden al Engine.

4. El Motor envía las peticiones al Downloader. Aquí ha de pasar por los middleware del spider.
5. Una vez que la página termina de descargarse, el downloader genera una respuesta (con esa página) y la envía al Engine, pasando por los Middlewares del Downloader.
6. El Engine recibe la Respuesta del Downloader y la envía al Spider para su procesado, pasando a través del Middleware del Spider.
7. El Spider procesa la respuesta y devuelve los ítems raspados y las nuevas solicitudes (a seguir) al motor, pasando por el middleware de la araña (véase process_spider_output()).
8. El Engine envía los ítems procesados a los Item Pipelines , luego envía las solicitudes procesadas al Schedulelr y pregunta por posibles próximas solicitudes a rastrear.

Desde el Item Pipelines, se guardan los Items procesados.
El proceso se repite (desde el paso 3) hasta que no hay más peticiones del Scheduler.

---

5. Contenido

Ahora se va a describir el contenido que refleja el dataset. Este contenido está obtenido como ya se ha mencionado de las páginas principales de cada videojuego.

'is_dlc' → Si el juego es una expansión de contenido o extra de otro juego.
'img_src' → URL de la imágen de la portada del videojuego
'short_description' → Breve descripción de videojuego
'recent_reviews' → Valoraciones más recientes de los usuarios
'recent_reviews_count' → Número total de valoraciones recientes
'all_reviews' → Total de valoraciones de los usuarios
'all_reviews_count' → Número total de valoraciones
'reviews_anomally' → Indica si existen anomalías en las reviews del videojuego
'release_date' → Fecha de lanzamiento del videojuego
'developer' → Empresa o usuario desarrollador del videojuego
'developer_url' → Url del desarrollador del videojuego
'publisher' → Editor del videojuego
'publisher_url' → URL del editor del videojuego
'tags' → Géneros relacionados con el videojuego
'discount_original_price' → Precio base del videojuego
'discount_final_price' → Precio final del videojuego con el descuento aplicado
'discount' → Descuento del videojuego
'price' → Precio base del videojuego en caso de no tener descuento
'game_content' → Contenido del videojuego
'name' → Título del videojuego
'genre' → Género principal del videojuego

'website' → URL de la página oficial del videojuego
'metacritic_score' → Puntuación de metacritic
'metacritic_url' → URL de la página del videojuego en metacritic

---

6. Propietario

El propietario del conjunto de datos que se ha obtenido de la web de steam a través de la técnica de scraping es la persona o entidad que ha realizado la extracción, siempre y cuando esta persona/entidad cuente con el **permiso explícito del propietario del sitio web**. En este caso, al no contar con el permiso explícito de "Valve Corporation", se podría asumir que el propietario del conjunto de datos es la propia entidad Valve Corporation.

A la hora de hacer scraping en un sitio web, es necesario tener en cuenta que la extracción de datos puede ser ilegal y no ética si no se cuenta con el permiso explícito del propietario. Para ello es necesario **revisar las políticas de privacidad y términos de uso de estos sitios web** para confirmar si se prohíbe explícitamente la extracción automática de estos datos. Referente al sitio web de Steam, ni la política de privacidad ni los términos de uso indican nada específico en términos de uso de robots automáticos para la extracción de datos.

Además, en las páginas existe el archivo "robots.txt", que gestiona el tráfico de los motores de búsqueda y a los robots de rastreo automático que secciones de la página web no deben ser rastreadas.

El archivo robots.txt de Steam incluye las siguientes instriucciones:

Disallow: /actions/ → Los robots de rastreo no tienen permitido acceder a la sección actions, ya que esta sección puede contener acciones de usuarios como comentarios o publicaciones

Disallow: /linkfilter/ → Los robots de rastreo no tienen permitido acceder a la sección linkfilter ya que esta sección puede contener información sobre enlaces compartidos por los usuarios.

Disallow: /tradeoffer/ → Los robots de rastreo no tienen permitido acceder a la sección tradeoffer ya que esta sección puede contener información personal sobre las ofertas comerciales que ofrecen los usuarios en la plataforma.

Disallow: /trade/ → Los robots de rastreo no tienen permitido acceder a la sección tradeoffer ya que esta sección puede contener información personal sobre las transacciones comerciales que ofrecen los usuarios en la plataforma.

Disallow: /email/ → Los robots de rastreo no tienen permitido acceder a la sección tradeoffer ya que esta sección puede contener información personal de los usuarios como el correo electrónico.

Teniendo en cuenta las limitaciones comentadas anteriormente, podemos decir que **el desarrollo de la práctica se ha hecho cumpliendo tanto las políticas legales como los principios éticos** sobre el uso del scraping.

Podemos encontrar en internet diferentes trabajos relacionados con el scraping en la página web de Steam. El artículo "Scraping All Game In Steam Using Python" donde se crea un programa para buscar todos los datos del juego que se busca. Otro ejemplo de programa para obtener información de la página web de Steam es el "Steam Scraper" disponible en Github, donde se puede obtener información de las reviews que dejan los usuarios en los videojuegos.

---

7. Inspiración

El conjunto de datos obtenido contiene información sobre los videojuegos alojados en la web de Steam. Podemos decir que actualmente Steam es la plataforma más grande de videojuegos para PC, donde se puede tanto comprar como jugar a estos videojuegos. Esto la hace una muy buena base de datos para este tipo de contenido.
Por ello, podríamos decir que el conjunto de datos que se puede obtener es como **una base de datos de videojuegos**. Además, el código implementado es capaz de localizar todos los juegos que existen en la plataforma ya que utiliza los códigos numéricos de las URLs de los videojuegos para ello, permitiendo almacenar los datos de todos los videojuegos.
También se ha implementado un sistema de gestión de, ya que se ha implementado un **sistema de almacenamiento de URLs scrapeadas**, lo que permite lanzar el programa periódicamente para actualizar la base de datos con las nuevas incorporaciones de videojuegos.

Con la información que se ha comentado sobre el juego de datos, se puede decir que las principales preguntas que puede responder este dataset son los siguientes:

- Base de datos de videojuegos como principal uso.
- Búsqueda de los juegos por valoración, ya sea por número, valoraciones recientes…
- Búsqueda de videojuegos por género.
- Búsqueda de videojuegos por nota de metacritic.
- Clasificar los juegos por precio u ofertas.
- Próximos videojuegos en salir a la venta.

Dada la extensión de los atributos que se han obtenido, es posible encontrar una gran cantidad de usos para este juego de datos.

en comparación con los trabajos analizados sobre el scraping en Steam, mientras que los otros trabajos encontrados buscan una función concreta (ya sea búsqueda de datos de un videojuego concreto, o la búsqueda de ofertas), este ejercicio se enfoca más en **una gestión global de la información de los datos de la web** para abrir las posibilidades a la hora del análisis posterior del juego de datos.

---

8. Licencia

Entre los términos y condiciones de Steam se lee:

"Por el presente acuerdo, Valve concede, y usted acepta, una licencia no exclusiva y el derecho de utilizar los contenidos y servicios para su uso personal sin fines comerciales"

Además la directiva de la UE sobre bases de datos del 1996, protege los elementos que existen en este dataset ya que "se ha realizado una inversión sustancial, tanto cualitativa como cuantitativa, para la obtención, verificación o presentación de los resultados".
Entendemos por lo tanto que, no habiendo ánimo de lucro en este proyecto, no existe ninguna problemática en otorgar a nuestro dataset una licencia Creative Commons 4.0, ya que es la más global de entre las que oferta la página de Zenodo.

Las propiedades principales de una licencia Creative Commons 4.0 son:

1. Reconocimiento: La licencia requiere que se dé crédito al creador original del trabajo.
2. No comercial: La licencia prohíbe el uso del trabajo para fines comerciales sin el permiso del titular de los derechos de autor.
3. Sin obras derivadas: La licencia prohíbe la creación de obras derivadas del trabajo original.
4. Herencia de la licencia: La licencia requiere que cualquier obra derivada se comparta bajo la misma licencia que el trabajo original.
5. Excepciones y limitaciones: La licencia permite ciertas excepciones y limitaciones, como el uso justo y la cita de fragmentos del trabajo, dependiendo de las leyes de cada país.

Además de la licencia del dataset también hemos desarrollado un código que sí que nos pertenece ya que el desarrollo ha sido realizado por nosotros. Por esta razón y ya que no tenemos ningún interés en que nuestro código no sea aprovechado por otras personas, hemos otorgado a éste una licencia GNU General Public License 3.0. Esta licencia da al usuario final la **libertad de usar, estudiar, compartir y modificar el software**. La versión 3.0 de la GPL, en particular, proporciona las siguientes garantías y condiciones:

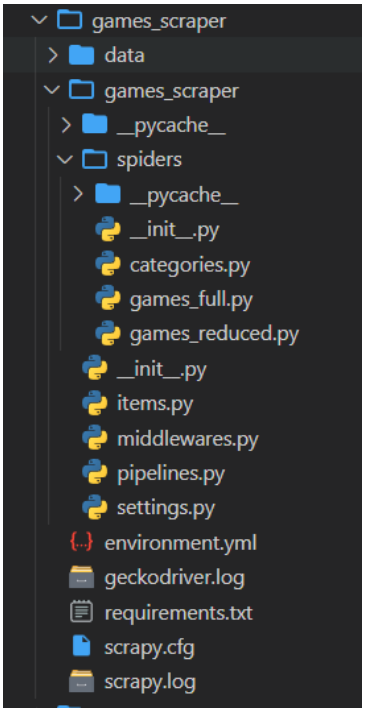
1. Libertad para usar, copiar y distribuir el software.
2. Libertad para estudiar el código fuente del software y adaptarlo a sus necesidades.
3. Libertad para redistribuir copias modificadas del software.
4. La obligación de compartir los cambios y mejoras en el software con otros usuarios, publicando la versión modificada bajo la misma licencia.
5. La obligación de proporcionar una copia de la licencia junto con el software.
6. La prohibición de cualquier restricción adicional sobre el uso, copia o distribución del software.

---

9. Código

Nota: El código completo se puede encontrar en la carpeta /source del repositorio de esta práctica. En esta sección se presentarán las partes del código que consideramos que son de mayor interés.

Tras un breve periodo de investigación decidimos realizar el proyecto con scrapy, es por ello que la estructura utilizada en el código coincide con la que se utiliza en esta librería de python. En nuestro caso contamos con lo siguiente:



- ***settings.py:*** Parámetros de configuración del proyecto

- ***middlewares.py:*** Middleware es el objeto que procesa la request que queremos hacer. En nuestro caso utilizamos un middleware instalado en otra librería (scrapy-selenium).

- ***pipelines.py:*** Aquí se encuentran los objetos encargados de tratar los ítems que llegan de los spiders.

- ***spiders:*** en esta carpeta tenemos que alojar nuestros spiders, que son los objetos que se encargan de recibir la respuesta del middleware, procesarla y devolver los ítems que llegarán al pipeline.

Si bien la arquitectura habitual de un proyecto de scrapy cuenta con un solo spider, en la mayor parte de ellos se trata de un proyecto que hace crawling a una web completa o a una única página. En nuestro caso dentro de la web de la cual estamos obteniendo la información hemos detectado tres estructuras diferentes de las que queríamos obtener los datos.

Además, hemos planteado el proyecto no como un método de obtención de datos para un momento puntual, sino como un proceso que pueda pasarse periódicamente para seguir completando y actualizando el dataset actual. Estas son las razones de la división de nuestros spiders en 3 módulos.

A continuación se entrará en más detalle en las modificaciones realizadas en cada uno de los archivos.

9.1. Settings.py

En este archivo establecemos las variables de funcionamiento del propio scrapy y creamos nuevas variables que utilizaremos en nuestros spiders, pipelines, middleware o cualquier otro módulo del proyecto.

Las dos primeras variables a las que hemos dado valor son *ROBOTSTXT_OBEY,* que hace que scrapy respete en todo momento las restricciones que aparecen en este archivo.

```
ROBOTSTXT_OBEY = True
```

Además hemos puesto el tiempo mínimo entre requests a 2.5 segundos, con esto evitamos que la carga al servidor pueda ser muy elevada (lo cual no será un problema dada la importancia de la página) o que nos bloqueen por realizar demasiadas peticiones a la vez.

```
DOWNLOAD_DELAY = 2.5
```

Para la utilización de selenium dentro de scrapy tenemos que indicar el navegador a utilizar, la ubicación del ejecutable y de sus drivers. Además podemos utilizar el argumento headless para que el navegador no se muestre. En nuestro caso hemos observado que en función de la versión de firefox utilizar este argumento puede provocar que uno de los spiders no funcione correctamente. Por último tenemos que añadir el middleware de selenium para que este sea el que realice las requests.

```
SELENIUM_DRIVER_NAME = 'firefox'
SELENIUM_DRIVER_EXECUTABLE_PATH = "C:\\geckodriver\\geckodriver.exe"
SELENIUM_BROWSER_EXECUTABLE_PATH = "C:\\Program Files\\MozillaFirefox\\firefox.exe"
SELENIUM_DRIVER_ARGUMENTS=[]
# SELENIUM_DRIVER_ARGUMENTS=['-headless']
DOWNLOADER_MIDDLEWARES = {
     'scrapy_selenium.SeleniumMiddleware': 800
     }
```

Podemos activar o desactivar la variable JOBDIR para que scrapy guarde el estado de nuestras *spiders* y sea capaz de seguir con una ejecución después de haberla parado. En nuestro caso hemos observado que no funciona cuando utilizamos el middleware de selenium. Esto probablemente se deba a que, según la documentación de scrapy, esta implementación solo funciona cuando las requests que devuelve el middleware son serializables por pickle.

```
# JOBDIR = "crawls/test"
```

En el caso de que creemos algún pipeline debemos declararlo en la siguiente variable:

```
ITEM_PIPELINES = {'games_scraper.pipelines.GamesScraperPipeline': 300}
```

Configuramos el log:

```
LOG_FILE = "scrapy.log" # output file
LOG_FILE_APPEND = True # If True it will appends to file, if False LOG_FILE will be
overwritten
LOG_LEVEL = 'DEBUG'
```

Ponemos el user agent de mozilla para que scrapy lo utilice en todas las requests. Para las requests hechas con selenium no es necesario poner el user agent ya que utiliza el del propio navegador (que es el que hemos copiado y pegado en la siguiente variable):

```
USER_AGENT="Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101
Firefox/112.0"
```

Scrapy también nos da la posibilidad de añadir headers de manera automática, para hacer esto debemos utilizar la variable que se muestra a continuación. En nuestro caso, como se utiliza alguna técnica dependiente del idioma (para detectar la variable is_dlc del dataset), hemos de pasar la cabecera para que utilice el español:

```
DEFAULT_REQUEST_HEADERS = {'Accept-Language':'es-ES,es;q=0.8,en-US;q=0.5,en;q=0.3'}
```

9.2. categories.py

En esta *spider* hacemos scraping únicamente a la landing page de steam, para obtener las categorías en las que se separan los juegos y sus correspondientes urls. Será en estas páginas de categoría donde encontraremos un datatable con los juegos.

Muy probablemente la funcionalidad de este spider debería estar implementada dentro del *game_reduced*, sin embargo decidimos dejarlo ya que representa bien la estructura de esta clase de scrapy.

Dentro de un spider tenemos sus atributos, en este caso declaramos únicamente el atributo *name*. El uso de atributos es importante ya que nos puede servir para modificar algunos parámetros del comportamiento de nuestro *spider* al lanzar la ejecución.

```
class Categories(scrapy.Spider):
    name="categories"
```

Para conocer las urls que debe visitar, debemos incluir la función *start_requests* que debe devolver una a una las requests, en las cuales se debe indicar como callback la función a la que se le pasará la respuesta:

```
    def start_requests(self):
        urls = [self.settings.attributes['LINKS'].value['index']]
        for url in urls:
            yield scrapy.Request(url=url, callback=self.parse)
```

Una vez la request a la url es realizada por el middleware, el resultado de ésta es devuelto al spider para que aplique la función parse (o cualquier otra que hayamos definido como callback en la función *start_requests.*

En esta función parse añadimos una línea para loguear el user agent (ya que pide en el enunciado de la práctica que se verifique que se está modificando) y después utilizamos beautifulsoup para obtener los datos que queremos (en este caso el nombre de cada categoría y su url) y devolverlos en forma de diccionario para que así puedan ser interpretados por el *item_pipeline* y enviados a un output.

```
    def parse(self, response):
        self.logger.info(f"Request user agent: {response.request.headers}")

        soup = BeautifulSoup(response.text, features="lxml")
        genre_selector = soup.find(id="genre_flyout")
        categories_dict = {}
        for el in genre_selector.find_all(class_="popup_menu_item"):
            try:
                categories_dict[el.text.strip()] = el.attrs['href']
            except KeyError as e:
                pass
            except Exception as e:
                print(e)
                print(el)

        res = {
             k:clean_url(v)
             for k,v in categories_dict.items()
             if clean_url(v) is not None
        }
        yield res
```

9.3. games_reduced.py

Este es el *spider* más complejo ya que es el que utiliza selenium. Comenzamos declarando sus atributos. Tendremos:

- El número de páginas máximo que queremos analizar (existen una cantidad inmensa de juegos en steam, es por esto que decidimos poner un límite).

- Dirección de almacenamiento del estado (recordemos que no es posible utilizar la gestión de estados de scrapy utilizando el middleware de selenium).

- Dónde exportar un archivo con las ids obtenidas (este archivo servirá como input del tercer spider).

- Aunque no aparezca entre los parámetros también hay que pasarle en el comando de ejecución la variable input_file indicando dónde está la salida del spider categories.

```
    n_pages_per_cat = 4
    state_file = "data/games_reduced_state.json"
    ids_file = "data/games_ids.json"
    state = {}
```

En la función *__init__* comprobamos si existe un estado y lo cargamos:

```
with open(self.state_file) as state_file:
      self.state = json.load(state_file)
```

Para la función *start_requests*, generamos las urls por medio de la url base de la categoría (llamando a la función *go_to_page* que nos genera la url de la página dada), siempre que ésta no almacene el valor -1 en el estado, ya que es el método que utilizamos para indicar que en esa categoría hemos alcanzado una página que ya no incluye juegos. Además, almacenamos en el estado la página actual en la que estamos para esa categoría.

```
for cat, url in cat_dict.items():
    count = 0
    while count < self.n_pages_per_cat:
        if(self.state.get(cat, 0)) < 0:
            self.logger.warning(f"Category {cat} limited reached)
            break
        self.state[cat] = self.state.get(cat, 0) + 1
        url = self.go_to_page(url, self.state[cat])
```

Tras esto devolvemos una request de selenium. Además de la url y el callback, tenemos que pasar:

- **script:** un script a ejecutar en javascript. En este caso es muy simple, solamente hace un scroll hacia abajo de 2600 unidades, de no hacer esto el contenido de la tabla que queremos extraer no se genera en la web.
- **wait_until:** aquí indicamos que debemos esperar hasta que un elemento con la clase *salepreviewwidgets_SaleItemBrowserRow_y9MSd* aparezca en el html de la página. Esta es la clase correspondiente a una fila de la tabla, la cual representa un juego.
- **wait_time:** en caso de no encontrar el elemento indicado en la variable anterior, éste será el tiempo que esperará antes de devolver un error 404.

```
        yield SeleniumRequest(
            url=url,
            callback=self.parse,
            script="scroll(0, 2600)",
            wait_time = 15,
            wait_until=EC.presence_of_element_located((
                  By.CLASS_NAME,
                  "salepreviewwidgets_SaleItemBrowserRow_y9MSd")),
        )
```

En la función parse, además de la extracción de los datos habitual realizada con beautifulsoup, detectamos si se ha llegado a una página vacía (que ya no contenga juegos) para así modificar el estado, dándole a la categoría un valor -10 y así la función que obtiene las requests deje de tenerla en cuenta.

```
games = soup.find_all(class_="salepreviewwidgets_SaleItemBrowserRow_y9MSd")
if soup.select_one(".saleitembrowser_EmptyResults_3_IxA")and len(games) == 0:
    cat = self.get_cat_from_url(response.url)
    self.logger.warning(f"Category {cat} limited reached")
    self.state[cat] = -10
    return
```

También almacenamos en el estado las ids de los juegos obtenidos para saltarnos un juego en el caso de que ya lo hayamos visitado y para exportarlas al final y ser usadas por el siguiente spider.

```
if id in self.state.get('ids', []):
    return
self.state['ids'] = self.state.get('ids', []) + [id]
```

El último punto importante de este archivo es que, para asegurar la permanencia del estado y el guardado de las ids se ha interceptado la señal de scrapy de cierre del spider, para de esta manera realizar estas operaciones antes de que el spider termine u obtenga un error.

```
@classmethod
def from_crawler(cls, crawler, *args, **kwargs):
    spider = super(GamesReduced, cls).from_crawler(crawler, *args, **kwargs)
    crawler.signals.connect(spider.spider_closed, signal=signals.spider_closed)
    return spider

def spider_closed(self, spider):
    self.logger.info(f"Saving state in {self.state_file}")
    with open(self.state_file, "w") as state_file:
        json.dump(self.state, state_file)

    self.logger.info(f"Saving ids in {self.ids_file}")
    with open(self.ids_file, "w") as ids_file:
        ids_file.write(str(self.state.get("ids", [])))
```

En la primera función, *from_crawler*, conectamos la señal *signals.spider_closed* con la función *spider_closed* de nuestro spider. En ésta función es en la que realizamos la lógica de guardado de nuestro estado y de las ids.

9.4. games_full.py

En este caso declaramos, además del nombre, una variable state que utilizaremos para gestionar el estado del *spider*, así en caso de interrupción de la ejecución podremos continuar más adelante.

También inicializamos una variable donde almacenamos las cookies que vamos a usar. La necesidad de usar la cookie reside en que steam, para mostrar juegos con contenido adulto, o con contenido desagradable, primero pide autorización al usuario. Al saltar la página de autorización el spider no conseguía obtener los datos. El problema lo hemos solucionado copiando y pegando los pares clave-valor nuevos que aparecían en la cookie tras rellenar manualmente los datos en el navegador.

Por último, al igual que para *games_reduced* también debemos introducir el atributo *input_file* apuntando al archivo que contenga las ids en el comando de ejecución.

```
name="games_full"
cookie = {
    'wants_mature_content': 1,
    'birthtime': "786254401",
    'lastagecheckage': "1-0-1995"
    }
state = {}


def __init__(self, *args, **kwargs):
    super(GamesFull, self).__init__(*args, **kwargs)
    if not hasattr(self, 'input_file'):
        self.logger.warning("No input file given, unable to crawl games IDs")
```

En la función *start_requests* hacemos gestión del estado para añadir solamente las ids de cuyos juegos no hayan sido obtenidos aún, y generamos las urls como *https://store.steampowered.com/app/{game_id/*

```
def start_requests(self):
    ids = [
          id
          for id in self.get_input_file_ids()
          if id not in self.state.get("crawled_ids", [])
    ]
    for id in ids:
        self.state["crawled_ids"] = self.state.get("crawled_ids", []) + [id]
        url = self.get_url(id)
        yield scrapy.Request(url=url, callback=self.parse, cookies=self.cookie)
```

En la función parse simplemente hemos sacado la información con *beautifulsoup*. No existe nada más a destacar en este archivo.

9.5. Modificación de scrapy_selenium

Uno de los mayores problemas que hemos encontrado ha sido la gestión de contenido dinámico. En la página web de cada categoría necesitamos cargar el contenido dinámicamente ya que cuenta con un data-table muy complejo que además no carga hasta haber hecho un poco de scroll hacia abajo. La página es demasiado compleja para seguir las trazas de javascript y poder realizar la ejecución de todas las funciones por medio de python.

Scrapy cuenta con Scrapy-Playwright, sin embargo no es compatible con plataformas windows, que es desde donde estamos desarrollando el proyecto. Al final nos hemos decantado por usar scrapy_selenium. Esta librería lleva sin mantenimiento desde 2019 y no cuenta con demasiadas funcionalidades.

Ya hemos descrito una de las problemáticas encontradas, que es la imposibilidad de que scrapy gestione el estado de un spider que utilice scrapy_selenium. Otro problema que hemos encontrado es sobre la implementación de las requests. Una request de scrapy_selenium se estructura como sigue:

```
yield SeleniumRequest(
     url=url,
     callback=self.parse,
     script="scroll(0, 2600)",
     wait_time = 15,
     wait_until=EC.presence_of_element_located((
           By.CLASS_NAME,
           "salepreviewwidgets_SaleItemBrowserRow_y9MSd")),
 )
```

La resolución de las esperas y el script se realiza de manera secuencial:

1. Esperar wait_time o hasta que los elementos de la variable wait_until se hayan cargado
2. Ejecutar el javascript del atributo script.

Esto lo podemos ver en el código de la librería (middleware.py):

```
 if request.wait_until:
     WebDriverWait(self.driver, request.wait_time).until(
         request.wait_until
     )

 if request.screenshot:
     request.meta['screenshot'] = self.driver.get_screenshot_as_png()

 if request.script:
     self.driver.execute_script(request.script)
```

Es por esto que lo hemos modificado para que el javascript se ejecute antes, quedando así:

```
        if request.script:
            self.driver.execute_script(request.script)

        if request.wait_until:
            WebDriverWait(self.driver, request.wait_time).until(
                request.wait_until
            )

        if request.screenshot:
            request.meta['screenshot'] = self.driver.get_screenshot_as_png()
```

Nota: Para encontrar el archivo middleware.py que tenemos que modificar podemos importarlo y observar su atributo *__file__*, desde el mismo entorno de python en el que estemos desarrollando nuestro proyecto:

```
from scrapy_selenium import middlewares
print(middlewares.__file__)
```

9.6. Ejecución

Hemos estructurado el proyecto para que la obtención de los datos se pueda parametrizar bastante. En función del spider que utilicemos y los parámetros que les enviemos obtendremos unos resultados diferentes, que era la idea original del proyecto. En términos generales:

- categories: obtiene el json de {categoría: url} con todas las categorías disponibles en la landing page de steam.
- games_reduced: obtiene la información reducida de los juegos por medio de las páginas de categorías.
- games_full: obtiene la información completa de los juegos a partir de una lista de ids.

Para realizar una ejecución con scrapy tenemos que lanzar el siguiente comando:

```
scrapy crawl {spider_name} -o {output_file} -s {seting_variable}={value} -a
{attribute}={value}
```

- Con -a podemos modificar los atributos del spider.
- Con -s podemos modificar las variables que existen en los settings
- Con -o indicamos el archivo y formato de salida (-0 hace lo mismo pero sobrescribiendo en caso de que exista)

De esta manera realizaríamos una ejecución de un ciclo completo de scrapping (podemos enlazar los 3 comandos con “&” para que se encadenen solos:

```
scrapy crawl categories
       -s LOG_FILE=data\output-1\categories.log
       -O data\output-1\categories.json

scrapy crawl games_reduced
       -a input_file=data\output-1\categories.json
       -a n_pages_per_cat=20
       -a state_file=data\output-1\games_reduced_state.json
       -a ids_file=data\output-1\games_ids.txt
       -a use_test_dict=False
       -s LOG_FILE=data\output-1\games_reduced.log
       -o data\output-1\games_reduced.json

scrapy crawl games_full
       -a input_file=data\output-1\games_ids.txt
       -o data\output-1\games_full.csv
       -s JOBDIR=data\output-1\crawl_games_full
       -s LOG_FILE=data\output-1\games_full.log
```

En la carpeta *data\output-1* tendríamos los logs de los 3 spiders, el archivo con las categorías y sus urls, el archivo con las ids de los juegos, y el *output* del *games_reduced* y el del *games_full* (que sería el dataset final que se entrega junto a la práctica)

Con esta configuración podemos parametrizar la ejecución para poder continuar con una ejecución anterior, hacer crawling de sólo algunas categorías u obtener solo la información reducida de manera más rápida para así, por ejemplo, actualizar la información de los precios.

9.7. Rendimiento

En una primera ejecución que hemos realizado, con una configuración de 20 páginas por categoría y 61 categorías, según los cálculos teóricos podríamos obtener un máximo de 15640 juegos (61 categorías, 20 páginas, 12 juegos por página), pero *games_reduced* solo ha conseguido 5660 juegos diferentes. Esto se debe a que un mismo juego puede (y suele) aparecer en múltiples categorías diferentes y cada categoría los ordena más o menos por popularidad, por lo que suelen ocupar posición en páginas similares. En el apartado de mejoras se proponen algunas soluciones para este problema.

El tiempo que han tardado en obtener la información ha sido de 1 hora y 7 minutos para el *games_reduced* y 4 horas y 52 minutos para el *games_full.*

Después hemos realizado una segunda ejecución partiendo del estado en el que habían quedado las ejecuciones anteriores y hemos conseguido otros 5337 juegos diferentes en 1 hora y 9 minutos para el *games_reduced* y 4 horas 31 minutos para el *games_full*

9.8. Mejoras

Entre las mejoras que se nos han ocurrido están las siguientes:

- La estructura habitual de un proyecto de scrapy cuenta con un solo spider. Ahora que hemos aprendido a estructurar bien este objeto, podríamos juntar los 3 spiders

en uno solo y lanzarlo con el comportamiento deseado utilizando los atributos deseados utilizando el parámetro -a en la ejecución del spider. Así podríamos aprovechar al 100% el trabajo en paralelo que realiza scrapy, en lugar de hacerlo de manera “semi-secuencial” como ahora.

- Incluir el middleware de selenium. En lugar de utilizar la librería que tantos problemas nos ha dado podríamos incluir el middleware de selenium, mejorando los siguientes comportamientos:

    - Ejecución del navegador si no se usan las requests de selenium.
    - Flexibilidad a la hora de ejecutar javascript y las esperas. Utilizando las cadenas de acciones de selenium sería posible parametrizar una request al 100%.
    - Compatibilidad con la gestión de estado de scrapy. Como ya hemos comentado, scrapy da problemas a la hora de gestionar el estado de un spider que utilice SeleniumRequest. Probablemente se pueda solucionar consiguiendo que las requests sean serializables con pickle, como indica la documentación de scrapy

- Mejorar la eficiencia de la obtención de información de las categorías. Si utilizaramos los géneros en lugar de las categorías (otra forma que tiene la página de Steam de organizar los juegos) ganaríamos eficiencia ya que el número de géneros es mucho menor por lo que el número de juegos que podemos encontrarnos repetidos se reduce drásticamente. Este cambio es bastante sencillo ya que los géneros son un tipo de categoría por lo que simplemente habría que reducir las categorías a aquellas que son consideradas géneros.

  Otra forma de ganar eficiencia sería hacer un algoritmo que escoja qué categoría “scrapear” en función del número de juegos del que ya dispongamos en cada categoría.

---

10. Dataset

https://doi.org/10.5281/zenodo.7860328

11. Video

El video se adjunta con la documentación

https://drive.google.com/drive/folders/1VWopidWMsSG19LuNNujoyiZ6IOAxDsIe?usp=share_link

12. Bibliografía

Acuerdo sobre la Política de Privacidad. (n.d.). Steampowered.com. Retrieved April 24, 2023, from https://store.steampowered.com/privacy_agreement/?l=spanish

Architecture overview — Scrapy 2.8.0 documentation. (n.d.). Scrapy.org. Retrieved April 24, 2023, from https://docs.scrapy.org/en/latest/topics/architecture.html

Open Data Commons. (n.d.). Opendatacommons.org. Retrieved April 24, 2023, from https://opendatacommons.org/licenses/dbcl/

Parameswara, S. (2022, March 9). *Scraping all game in Steam using python*. Medium. https://medium.com/@senchooo/scraping-all-game-in-steam-using-python-e9f0ad206add

4Perunicic, A. (n.d.). *steam-scraper: A pair of spiders for scraping product data and reviews from Steam*.