# Actividad 3 (2da parte) - Escarbando a lo loco

ExactasPrograma - Datos

Invierno 2020

Ahora que ya aprendimos algunas cosas básicas de cómo traernos una página, cómo hacer para extraer el texto y, quizás, lidiamos con algunos temas de codificación de caracteres *raros*, estamos en condiciones de iniciar nuestra segunda etapa del viaje: *crawlear* páginas a escala Superman.

Esta actividad se basa en la siguiente motivación:

> *Vamos a suponer que queremos explorar una nueva área de trabajo porque alguien nos dijo que es "lo que se viene". Da la casualidad que sabemos poco y nada de esa área y queremos ver de qué se trata. Lo que se nos ocurre hacer es empezar aprendiendo cuáles son las palabras más utilizadas en el área en cuestión para entender por dónde viene la mano.*

Dado que somos expertos (o estamos aprendiendo a serlo) en el arte de traer cosas de Internet, el procedimiento que vamos a hacer es el siguiente:

1. Elegir una o dos *keywords* que representen el área en la que me quiero sumergir. Si no se les ocurre ninguna, pueden usar algunas de éstas:

   - `Biodiversity`
   - `Nanotechnology`
   - `Vaccine`
   - `Robot`
   - `Climate change`
   - `Drug delivery`
   - `Parallel`

2. Para que podamos realizar esta tarea desde nuestras propias casas, vamos a utilizar un buscador que esté accesible, nos permita traer los resultados y que admita indicarle que solo queremos resultados que sean tipo *open access* ya que sino tendríamos que tener acceso a las publicaciones desde casa y eso *no* se puede hacer *sin pagar*. El sitio que vamos a usar para hacer la búsqueda es bastante conocido: `https://pubmed.ncbi.nlm.nih.gov`.

   Vamos a probar realizar una búsqueda en esa página con la *keyword* `baldness`: al poner esa palabra en el cuadro de búsqueda devuelve una página que dice que se encontraron 23404 resultados (en mi caso el primero es *Male baldness* de Clarke en la revista *Australian Family Physician*).

   Si todo fue bien, deberíamos tener diez resultados de esta búsqueda en el navegador y en la barra de direcciones algo similar a esto: `https://pubmed.ncbi.nlm.nih.gov/?term=baldness`. Pueden ver que en la dirección de la página está puesta la *keyword* que indicamos originalmente (la palabra `baldness`). Esta es la manera que tiene el sitio de indicar los parámetros que el usuario ingresa en la página.

3. Ahora vamos a cambiar la cantidad de resultados que trae la página por vez. Esto se hace en `display options` que despliega un menú con distintas opciones de visualización, vamos a cambiar para que se vean 50 resultados por página (la opción es `Per page`). Una vez que hicimos esto, la dirección de la página web cambia a: `https://pubmed.ncbi.nlm.nih.gov/?term=baldness&size=50` y si vamos

al fondo de la página, veremos que efectivamente, hay 50 resultados. Nuevamente, el parámetro que elegimos cómo usuario, se codifica en la dirección como `size=50`. Podemos cambiar este valor entre los disponibles (no podemos poner cualquier cosa, por que no funciona, seguro lo van a comprobar) y vemos que podemos cambiar la cantidad de resultados sin entrar al menú, solamente alcanza con cambiar el número que está en la dirección de la página web.

Para terminar de ver cómo voy a bajar los trabajos que me interesan, solo resta activar el filtro de trabajos disponibles de forma gratuita / *open access*. Dentro de la página, simplemente hay que tildar la opción `Free full text`, con lo que la dirección de la página quedaría algo así: https://pubmed.ncbi.nlm.nih.gov/?term=baldness&filter=simsearch2.ffrft&size=50 y encuentra 5498 resultados.

Este *link* correspondería con nuestro primer paso en la exploración de la nueva área de trabajo, ya que si reemplazo *baldness* por otra palabra clave que me interese, directamente podría obtener la lista de los primeros 50 trabajos dentro del área de interés que son de acceso gratuito.

4. No queremos hacer click a mano en los 5498 resultados, sino que queremos que esto sea un proceso automático. Nuestro siguiente paso es preparar un código para poder traernos, sin intervención nuestra, los 5498 trabajos completos para poder analizar y extraer las 20 palabras más frecuentes distintas a la *keyword* usada.

   Desde el directorio donde queremos guardar los trabajos y los archivos que generemos, vamos a escribir el siguiente comando:

```
scrapy startproject sherlock
```

   Si esto va bien, debería informar que creó el proyecto *sherlock* y que ya podríamos empezar a escribir el código del programa que traerá los archivos de los papers. Deberían poder ver que hay un nuevo directorio que se llama, justamente, `sherlock` que contiene un archivo llamado `scrapy.cfg` y otro directorio que también se llama `sherlock` (sí que molesto que tenga el mismo nombre, es para confundirlos).

   Dentro del segundo directorio `sherlock` hay una carpeta que se llama `spiders`, que es el lugar donde vamos a poner los programas que van a ir conectándose a las páginas donde están los artículos y descargándolos. Estos programas se conocen, en la jerga de scrapy, como `spiders` (sí, igual que el nombre del directorio, es para confudirlos). Exploren los distintos directorios y vean que hay varios archivos, por ahora, no vamos a tocar nada de eso.

5. El siguiente paso es crear el programa que efectivamente trae las páginas que nos interesan. Como dijimos antes, estos programas se llaman `spiders`. `scrapy` nos ofrece la opción de crearnos una base para un `spider` de manera de poder modificarlo luego. Desde el directorio base de nuestro proyecto (el primer nivel dentro de *sherlock*), tipeamos el siguiente comando:

```
scrapy genspider -t basic messi prueba.com
```

   Este comando nos crea un archivo llamado `messi.py` en el directorio `spiders` con el siguiente contenido (¡verificarlo!):

```
import scrapy

class MessiSpider(scrapy.Spider):
    name = 'messi'
    allowed_domains = ['prueba.com']
    start_urls = ['http://prueba.com/']

    def parse(self, response):
        pass
```

   Breve explicación de cada parte:

- `name`: es el nombre que le ponemos al `spider`. Debe ser único en el contexto del proyecto (en nuestro caso, del proyecto *sherlock*).
- `allowed_domains`: esta variable es optativa y permite restringir los dominios (i.e. páginas web) que nuestro `spider` puede visitar. Ya vamos a ver que nuestro `spider` puede visitar otros sitios en base a la información que vaya obteniendo, así que esta variable es útil para indicar dónde **no queremos ir**. En nuestros ejemplos, no va a ser necesaria, pero la podrían usar para evitar traer información de revistas de dudosa calidad.
- `start_urls`: es la página por la que vamos a empezar a *crawlear* información.
- `parse`: esta función es **importante**, se trata de la función que se llama cada vez que nuestro `spider` recupera una página con información. En este ejemplo no hace nada, pero la idea será usar esta función para guardar los artículos que nos interesan.

6. El siguiente paso es modificar el `spider` que recién generamos como se indica a continuación:

   I) Comentar la línea que incluye la definición de `allowed_domains`.

   II) Cambiar el link inicial (`start_urls`) con el comando de búsqueda que habíamos definido antes: https://pubmed.ncbi.nlm.nih.gov/?term=baldness&filter=simsearch2.ffrft&size=10 (para que traiga 10 resultados, vamos a empezar de manera humilde).

   III) Modificar la función `parse` para que convierta a texto la página recuperada que queda en la variable `response` y guardarla en un archivo (usar un nombre de archivo *razonable*). Recordar agregar el `import` de `html2text` para usar la conversión a texto.

   IV) Ejecutar el `spider` (ver cómo en el siguiente párrafo, que no te gane la ansiedad), abrir el archivo que guardaron y explorar su contenido.

   Para **ejecutar** un `spider` hay que poner el siguiente comando **desde un directorio dentro del proyecto** en la consola:

```
scrapy crawl messi
```

   La salida de este comando tendría que ser más o menos parecida a cuando hicimos el `fetch` de una página desde dentro del `shell` de `scrapy`. Si algo no anduvo bien, el mensaje de error debería indicarles para donde correr... eso esperamos. A veces no es tan fácil y puede quedar mezclado en medio de muchos mensajes propios del `scrapy`.

> La forma de trabajo propuesta consiste en tener el código del `spider` abierto en un editor de texto o en el entorno de desarrollo `spyder`. Cada vez que tengamos que probar si funciona, vamos a ejecutar el `spider` desde la consola para poder ver los mensajes que nos de.

7. En este paso, vamos a procesar la respuesta que nos trajo el `spider` para obtener las direcciones de los artículos que tenemos que bajar. Si miran el archivo que guardaron en el punto anterior, pueden encontrar el *string* "`doi: `" que indica que a continuación está el identificador digital único de un artículo. Por medio del *doi* uno puede acceder directamente a la página de la revista donde está publicado. Por ejemplo: el primer artículo de nuestra búsqueda no tiene *doi*, por lo que no podremos recuperarlo... así es la vida.

   El segundo artículo es *He H, Xie B, Xie L. He H, et al. Medicine (Baltimore). 2018 Jul;97(28):e11379. doi: 10.1097/MD.0000000000011379. Medicine (Baltimore). 2018. PMID: 29995779 Free PMC article. Review.* El *doi* de este artículo es `10.1097/MD.0000000000011379` (noten que está en el medio de la cita) y para accederlo, tendríamos que ir al *link* https://dx.doi.org/10.1097/MD.0000000000011379.

   La tarea que nos espera en este punto es generar una lista de *links* en base a la búsqueda realizada. Tenemos que implementar la función `extraer_links(texto)` que recibe el texto de la búsqueda que hicimos en pubmed (el mismo que convertimos y guardamos en una archivo en el punto anterior)

y devuelve una lista en la que están todos los *dois* pero en formato link para que podamos ir directamente a la página (revisar bien cómo aparecen para poder extraerlos). Esta función debe estar en el archivo `messi.py` antes de la definición de la clase (después de los imports). Por ejemplo:

```
In [24]: texto = "He H, Xie B, Xie L. He H, et al. Medicine (Baltimore). 2018 Jul;97(28):
    ...: e11379. doi: 10.1097/MD.0000000000011379. Medicine (Baltimore). 2018. PMID: 2999
    ...: 5779 Free PMC article. Review."

In [25]: mis_links = extraer_links(texto)

In [26]: print(mis_links)
['https://dx.doi.org/10.1097/MD.0000000000011379']
```

Si usamos el texto obtenido de la búsqueda que hicimos recién, la lista debería tener 8 *links* correspondiente a todos los artículos que tenían *doi*.

8. Nuestro spider, empieza a tener un poco de color, ya tenemos una función definida que permite extraer los *links* a partir de los *doi* de la búsqueda que estoy haciendo. Vamos a definir, ahora, una función que procese el artículo que bajamos y lo guarde en un archivo. La vamos a llamar `parse_paper(self, response)`. Tiene los mismos parámetros que la función `parse` que viene definida en la estructura básica del `spider` y tiene que estar definida a continuación (dentro de la clase `MessiSpider`). El parámetro `response` va a tener la estructura que incluye el artículo que queremos bajar. Lo que debemos hacer es similar a la prueba de la actividad anterior: convertir el artículo a texto y guardarlo en un archivo (recordar importar el módulo `html2text`) de manera similar a cómo hicimos antes.

   **Warning!**: Esta función se va a llamar varias veces, si dejamos fijo el nombre del archivo, me lo va a escribir todas las veces encima, tenemos que pensar una manera de generar un nombre de archivo distinto para cada vez que llamemos a esta función.

   **Warning! bis**: Puede ocurrir que el artículo que querramos traer no esté como una página html sino que sea un archivo pdf (binario). Si intentamos convertir a texto una `response` que es un archivo, vamos a generar un error. Es posible preguntar a la variable `response` si lo que tiene adentro es texto o no por medio de la función `hasattr(response, "text")` que devuelve `True` si lo que nos trajimos es un texto y `False` en caso contrario (ponemos un `if` y nos ahorramos mala sangre).

9. Ahora vamos a combinar las dos funciones que hicimos en el `spider` que estamos definiendo para que realice la búsqueda inicial y luego haga la descarga de cada uno de los *links* que encontremos. La función `parse` se llama automáticamente para procesar la respuesta al *link* que pusimos en la variable `start_urls`. En nuestro caso, esta función la va a llamar `scrapy` cuando haya obtenido el resultado de la búsqueda en pubmed. Lo que tenemos que hacer es usar la función `extraer_links` para obtener la lista con todos los links de los papers con *doi* que estaban la búsqueda. Una vez que tengamos esa lista, tenemos que hacer que `scrapy` obtenga cada una de las páginas y las procese con la función `parse_paper` que definimos en el punto anterior.

   `Python` tiene la operación `yield` que nos va a ayudar con eso. `yield` tiene un efecto similar al `return`, solo que en lugar de cortar la ejecución, la función continúa. Va devolviendo (generando en la jerga) los resultados de uno. Estos resultados son tomado por *scrapy* que los interpreta como pedidos (`requests`) a servidores web y arma todo lo necesario para que nuestro `spider` luego las obtenga. En código, se escribe así:

```
    for link in mi_lista_links:
        yield scrapy.Request(link, callback=self.parse_paper)
```

   `scrapy` va a usar el elemento `i` de la lista como *link* y va a armar el pedido, el segundo parámetro le indica que la respuesta la procese con esa función (que nosotros implementamos para que guarde cada artículo científico como texto en un archivo diferente). `scrapy` se encarga de ir haciendo el seguimiento de los pedidos a los servidos web y de obtener la información de todas las páginas y, al final, termina llamando a la función `parse_paper` para guardar la información que nos interesa.

10. Para poder bajar información de manera de que los servidores no nos insulten o nos lo prohíban, hay un par de detalles que podemos agregar. El primero es el `user_agent` que es una variable que usan los *browsers* para identificarse, nosotros podemos setear esa variable en la configuración de nuestro proyecto para que nuestro *spider* se haga pasar por una persona. Pero hay una solución mucho más poderosa, vamos a instalar el paquete `scrapy-user-agents` que tiene alrededor de 2200 identificadores distintos de *browsers*. `scrapy` puede usar estos identificadores no solo para hacerse pasar por un humano, sino para engañar al sitio haciéndole creer que cada pedido viene de un *browser* distinto.

    Para instalarlo en Linux y en Windows, deberían usar algunas de las dos siguientes opciones, varía de una distribución a otra: `pip install scrapy-user-agents --user` o `pip3 install scrapy-user-agents --user`. Este paquete no está dentro de Anaconda.

    Para que nuestro `spider` use este paquete, tenemos que agregar una configuración en el archivo `settings.py`:

    ```
    DOWNLOADER_MIDDLEWARES = {
        'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': None,
        'scrapy_user_agents.middlewares.RandomUserAgentMiddleware': 400,
    }
    ```

    Nos tenemos que dar cuenta que está funcionando bien porque, al mirar el log de funcionamiento, tiene que indicar que está dando distintos valores de `user_agent`.

11. El otro detalle de configuración que se puede cambiar para bajar más artículos deshabilitar seguir la reglas de convivencia entre `spiders` y servidores web. Dentro del archivo `settings.py`, podemos poner la variable `ROBOTSTXT_OBEY = False` (esto debería estar comentado, así que simplemente es descomentar).

    Al hacer esto, nuestro `spider` deja de acatar las recomendaciones de los administradores del sitio web y baja todo lo que le pedimos sin medir las consecuencias.

    Lo peor que puede pasar es que nuestra dirección IP sea marcada como perteneciente a *gente mala* y ese servidor deje de responder nuestros pedidos (en general por una determinada cantidad de tiempo, tipo quince minutos).

    Si realmente **necesitan** bajar esa información, tienen a disposición esa herramienta. En el contexto de nuestra actividad, podemos probar la diferencia, pero recomendamos seguir las reglas ya que nos podemos arreglar igual.

12. Llegamos al punto en el que deberíamos haber corrido nuestro `spider` y tener los archivos con el texto de los artículos. Ahora tenemos que procesarlos. Para eso, vamos a crear **otro** archivo `Python` en donde vamos a implementar el procesamiento de la información obtenida. En este nuevo archivo que llamaremos `analisis.py` vamos a tener que hacer un programa que lea los archivos que generamos con el `spider` y haga una nube de palabras por cada artículo (almacenándola como pdf en un archivo diferente cada una) incluyendo las 20 palabras más frecuentes que no sean triviales y una nube de palabras general para todos los artículos.

    Deberán nombrar los archivos generados con las nubes de palabras de manera razonable (les dejamos que piensen qué forma convendría), pero a la nube de palabras final **le deben poner el nombre** `nube_final.pdf`.

Optativo 1. Hasta ahora, hicimos un ejemplo de baja escala para ver si todo funciona. Vamos a empezar a darle más intensidad. Nuestra primer búsqueda incluyó solo los 10 primeros artículos que cumplían el criterio de búsqueda querido (en nuestro ejemplo, usamos la palabra *baldness*).

    Vamos a repetir el procesamiento que hicimos, pero el link de búsqueda será, en cada caso:

    *a*) https://pubmed.ncbi.nlm.nih.gov/?term=baldness&filter=simsearch2.ffrft&size=50

*b*) https://pubmed.ncbi.nlm.nih.gov/?term=baldness&filter=simsearch2.ffrft&size=100

*c*) https://pubmed.ncbi.nlm.nih.gov/?term=baldness&filter=simsearch2.ffrft&size=200

Solamente deberán generar la nube de palabras obtenida en base a las 20 palabras más frecuentes no triviales que se obtengan globalmente de la búsqueda. En cada caso, los archivos se deberán llamar: `nube_final_050.pdf`, `nube_final_100.pdf` y `nube_final_200.pdf`.

Optativo 2. Nos queda el último paso antes de volvernos un agujero negro de artículos científicos. En nuestros ejemplos anteriores, siempre nos quedamos con la primera página de la búsqueda (los primeros diez resultados en el primer ejemplo y 50, 100 y 200 en el punto anterior). Podemos hacer que `scrapy` siga recorriendo otras páginas adicionales y repitiendo el proceso de bajar la información en cada una de estas páginas.

El *link* para traerse la segunda página de la búsqueda sería https://pubmed.ncbi.nlm.nih.gov/?term=baldness&filter=simsearch2.ffrft&size=50&page=2. La parte nueva es "`&page=2`", si cambiamos el número dos por otro número, podríamos ir pidiendo las distintas páginas.

La tarea de este punto es modificar nuestro `spider` para que avance a la siguiente página, una vez que la procesó (es decir, que extrajo los *links* e hizo el `yield` por cada uno). Al final deberían agregar otro `yield` que ponga como dirección (url) el mismo link de la búsqueda, pero incrementando la página requerida. Para el parámetro `callback`, en este caso usaremos la misma función `parse` que estamos modificando.

Entonces, la estructura de nuestro `spider` quedaría así:

- Función `parse`: será la encargada de procesar una página de resultado de la búsqueda. Extrae los links que encuentra y hace un `yield` por cada uno usando a la función `parse_paper` como *callback*. Antes de terminar, debe hacer otro `yield` a la siguiente página siempre y cuando corresponda de acuerdo al criterio de parada que establezcan (alguno tiene que tener, sino se va a colgar intentando traer cosas).
- Función `parse_paper`: se invoca con cada uno de los artículos obtenidos. Debe bajar a disco en un archivo diferente el contenido en texto de cada una de las páginas.

**Warning**: en algún momento se acaban los resultados o no queremos colapsar nuestra máquina, así que antes de hacer el `yield` a la nueva página, conviene que establezcan algún criterio de corte (se recomienda una combinación de cantidad máxima de páginas visitadas con haber conseguido al menos un *doi* para procesar en la página actual).