UCLM
UNIVERSIDAD DE CASTILLA-LA MANCHA

Escuela Superior de Informática

(TITLE)

Autor: María González Gutiérrez

Asignatura:

Grado en Ingeniería Informática

2020, (MONTH)

Índice

1. INTRODUCCIÓN Y OBJETIVOS

RELLENAR PLACEHOLDERS

Empecé el proyectoporque me gusta el fanfiction y el análisis de datos. Existen en internet numerosas comunidades de fan que son muy activas y producen una vasta cantidad de contenido muy detallado; son básicamente una gran discusión entre los fans de una obra sobre qué es lo que dicha obra significa para ellos, y sobretodo, qué es lo que a ellos les hubiese gustado llegar a ver realizado en el texto de la misma. A veces, estas comunidades crean enormes proyectos de calidad profesional de forma totalmente gratuita, simplemente para mejorar el espacio y las experiencias del resto de miembros. Uno de los mayores ejemplos de este tipo de 'trabajo fan' es la Organizaton for Transformative Works, una organización sin ánimo de lucro creado 'por fans y para fans' que crea y mantiene proyectos como FanLore, una wiki sobre la cultura fan, o su comité legal, que se encarga tanto de educar sobre leyes de propiedad intelectual y el *fair use* del mismo como de involucrarse en los procesos jurídicos sobre copyright de diversos gobiernos (especialmente Estados Unidos) para defender el derecho del público general a crear obras derivadas.

Uno de sus proyectos más famosos es Archive of our Own, comúnmente acortado a AO3, un sitio web que aloja principalmente fanfiction y cuyo objetivo es, por un lado, facilitar la tarea de encontrar fanfics para aquellos que quieran leerlos y, por otro, funcionar como un archivo que clasifique y documente el fenómeno fanfic a nivel global. En sus datos de mayo de 2020, aparecen más de dos millones de usuarios registrados, y más de seis millones de trabajos alojados. AO3 se ha convertido en una parte fundamental de la cultura fan, especialmente la dedicada al fanfiction, y en este proyecto lo utilizaré como fuente de información.

En literatura comparada se suelen tener en cuenta dos perspectivas a la hora de analizar una obra: la de la teoría del autor, que tiene en cuenta lo que el autor quería comunicar y plasmar en esa obra, y la de la "muerte del autor"[Bar68], que tiene en cuenta el mensaje con el que los lectores se quedan tras leer la obra (independientemente de si coincide con el que el autor quería comunicar). Para entender el mensaje de una obra de forma plena[Ell18], es necesario tener en cuenta tanto la intención comunicativa del autor, como el mensaje que al final los lectores acaban entendiendo. Y como cada lector es hijo de su padre y de su madre, acaban surgiendo muchas posibles interpretaciones distintas a partir de un único texto.

Tradicionalmente, los académicos solamente han tenido en cuenta las opiniones de un grupo reducido de personas (compuesto principalmente por otros académicos) a la hora de analizar

una obra desde la perspectiva de la muerte del autor, ya que el lector común no suele tener a su disposición las herramientas necesarias para difundir sus interpretaciones. Sin embargo, desde que Internet y los foros como *LiveJournal* se volvieron accesibles a grandes partes de la población, miles de comunidades fan empezaron a organizarse justamente con la intención de poner en común sus interpretaciones, de expresar sus críticas y opiniones. No todas estas discusiones tienen lugar en forma de fanfiction, pero es un género muy popular en las comunidades fans, y yo personalmente estoy muy familiarizada con sus estructuras y códigos.

Estas comunidades de Internet están generando una cantidad inmensa de opiniones y perspectivas en torno a un tema común en foros públicamente accesibles, y me pareció interesante la idea de crear un sistema que sea capaz de recoger y procesar toda esta información para crear un .ªbanico"de las distintas interpretaciones que existen en una comunidad fan, especialmente aquellas sobre los personajes y las relaciones entre ellos. El resultado final se podría utilizar como herramienta dentro de la propia comunidad fan, para observar cómo tienden a interpretar a ciertos personajes a nivel de comunidad y cómo estas interpretaciones cambian a lo largo del tiempo, o en distintas subsecciones dentro de la comunidad en general. También se podría utilizar como herramienta general de análisis literario, aplicándola primero a la obra original y luego a un conjunto de fanfics representativos, y observando cuáles son las diferencias entre la perspectiva del autor original y la de los lectores (convertidos en autores fan).

Al empezar el proyecto, no sabía mucho sobre análisis de texto, por lo que empecé a estudiar sobre análisis de texto natural y extracción de información usando el libro *Natural Language Processing*, de Jacob Eisenstein[Eis18]. Tras la fase de recogida y limpieza de datos (explicado en detalle en las secciones 4.1 y 4.2), el proceso de extracción de información que tenía que seguir consistía en:

1. Identificación de entidades
2. Identificación de relaciones entre entidades
3. Identificación de eventos

2. FANFICTION Y ARCHIVE OF OUR OWN

Fanfiction (del inglés *fan fiction*, 'ficción del fan', y abreviado como 'fanfic') es el nombre que recibe un texto basado en una historia ya existente (normalmente con copyright), en particular

cuando el autor es fan de la obra de la cual su texto deriva. Son, por lo tanto, textos de ficción sin ánimo de lucro que los fans escriben como expresión de su creatividad.

El concepto detrás del fanfiction es, en esencia, una ausencia percibida en la historia original. Uno se termina un libro o un videojuego y siente que le falta algo: el pasado de un protagonista, una perspectiva distinta de un conflicto, una relación que acabó o nunca empezó, qué sucede después del final, o quizás que a la historia le hacían falta doscientas páginas más, o incluso que tendría que haber sido de un género literario distinto... Hay algo en la historia que está ausente. El lector se queda con ganas de explorar más a fondo el mundo y los personajes que el autor ha creado, y de aquí nace el impulso de crear historias propias en las que se exploran dichas ausencias. Por tanto, no es sorprendente descubrir que hay muchos fanfics en los que se cambia el destino de tal o cuál personaje, que exploran qué sucede tras el final, o que llevan a cabo exploraciones exhaustivas de los conflictos, los personajes y sus motivaciones desde perspectivas distintas a las de la obra original.

Todos estos motivos hacen que el fanfiction se considere una obra derivada[Swi98], y está en su naturaleza el reflejar las opiniones y críticas que el autor tiene de la obra original: qué es lo que le gusta, qué temas siente que faltan en la obra, qué cosas tendrían que haberse explicado desde una perspectiva distinta, etc.

Por ejemplo, es evidente al leer los libros de la saga *Harry Potter* que el texto quiere que pienses que Ron Weasly, el mejor amigo del protagonista, es un chico un poco torpe y bocazas pero con buen corazón, y un buen amigo de Harry. Sin embargo, muchos fans no interpretaron a Ron como torpe y bocazas, sino como egocéntrico e insensible, y hay no pocos fanfics en los que Ron y Harry discuten y dejan de ser amigos, o en los que Ron es directamente un villano aliado con Voldemort.

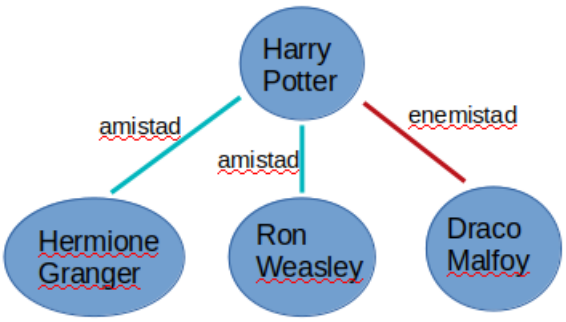
Cuando los fans de una misma obra se reúnen y organizan, se crean comunidades fan llamadas "fandoms", que suelen crear foros donde intercambiar sus impresiones, teorías y, por supuesto, fanfiction y otras formas de arte fan. Es evidente que existe un intercambio de ideas en foros de discusión y otras comunidades online explícitamente creadas para conversar, pero ya que es totalmente posible inferir las opiniones de un autor a partir de sus fanfics, tanto escribir como leer fanfiction son actividades que contribuyen al discurso general del fandom, ayudando a popularizar algunas teorías y generando las suyas propias.

Cuando un fandom alcanza un cierto nivel de madurez, algunas teorías se consolidan y el fandom acaba formando, a nivel de comunidad, una interpretación propia de la obra original. Para distinguir la perspectiva del fandom de la que realmente pretende transmitir la obra original, en los fandoms se distingue entre el *fanom* y el canon. Siguiendo el ejemplo de *Harry Potter*, el Ron Weasly del *fanon* es una persona egoísta que sólo es amigo de Harry por interés, mientras que el Ron Weasly del canon tiene una amistad sincera con Harry. *Fanon*, por tanto, es el 'conjunto de teorías basadas en el material original que, aunque generalmente parecen ser la interpretación 'obvia' o 'única' de los hechos canónicos, no son realmente parte del canon'[Stu17].

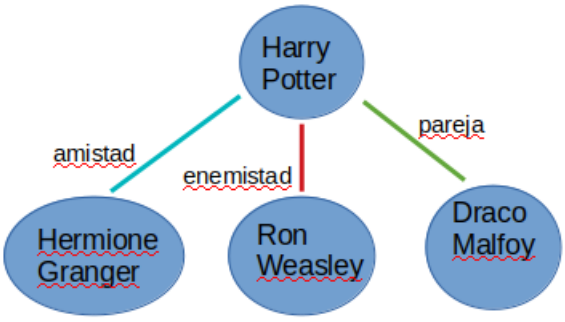
En resumen, las comunidades fan tienen una interpretación propia de la obra original llamada *"fanon"*, que influencia los fanfics que los miembros de dicha comunidad van a escribir y, a su vez, los escritores de fanfic también crean y popularizan interpretaciones que se acaban convirtiendo en parte del *fanon*.

Como se ve en el ejemplo de *Harry Potter*, las relaciones entre personajes son una de las mayores fuentes de especulación entre los fans, especialmente las relaciones románticas. En general, los personajes a los cuales los fans les tienen manía acaban convertidos en villanos (o, como mínimo, enemigo de los protagonistas) en los fanfics, incluso aunque en la obra original sean aliados. Naturalmente, lo mismo sucede a la inversa: los fans tienden a convertir en amigos y aliados a los personajes que les gustan, incluso aunque en la obra original sean los villanos de la historia. Por tanto, simplemente contrastando las relaciones presentes en un fanfic con las relaciones de la obra original podemos tener una buena idea de cuál es la interpretación del autor del fanfic.

Las relaciones románticas entre personajes son una parte enorme de la especulación fan. El romance es uno de los temas más populares, y aunque las relaciones canónicas atraen naturalmente la atención de muchos fans, 'inventar' parejas en el *fanon* no sólo es común, sino una de las principales actividades de un fandom. Los fans ven parejas y conflictos amorosos tanto entre amigos como enemigos, y son felices de ignorar todos y cualquiera de los obstáculos que existan en el canon con tal de tener el escenario necesario para que su pareja preferida pueda estar junta, llegando incluso al extremo de sacar a los personajes del universo al que pertenecen para meterlos en otro más amistoso. Un villano que es muy popular entre los fans tiene garantizados fanfics en los que cambia de bando, convirtiéndose en aliado y pareja del protagonista (no necesariamente en ese orden).



A) Grafo de relaciones entre personajes en los libros de la saga *Harry Potter*



B) Grafo de relaciones entre personajes que se encuentra fácilmente en los fanfics de *Harry Potter*

Como se ha dicho anteriormente, los fans se organizan en comunidades según la obra que es el objeto de su admiración, y tienen sitios como AO3, dedicados a alojar y compartir sus creaciones fan. Evidentemente, analizar las más de seis millones de obras existentes en AO3 es una tarea imposible con los recursos a mi alance, de modo que elegí utilizar únicamente los fanfics basados en *Good Omens*, un libro de Terry Pratchett y Neil Gaiman, en parte por mi familiaridad con esa comunidad, pero también porque tenía una cantidad de fanfics extensa pero manejable.

AO3 no es el único sitio web popular para alojar fanfiction, pero a lo largo de esta década ha desplazado a sitios como Fanfiction.net y Wattpad, y la característica que condicionó mi decisión es su extensivo sistema de etiquetas, que como se verá más adelante fue fundamental para filtrar y gestionar los datos del proyecto.

En términos legales y de derechos de autor, la mayoría de legislaciones considera el fanfiction como un tipo de obra derivada [Swi98] y por tanto entra dentro del *fair use*.

3. EXTRACCIÓN DE INFORMACIÓN EN OTROS TRABAJOS

La identificación de entidades y relaciones son problemas que se abordan en el análisis de lenguaje natural, y en las últimas décadas han surgido diversas técnicas de *machine learning* que han dado muy buenos resultados, sobretodo teniendo en cuenta la amplitud del tema. Es una solución atractiva por su capacidad de resumir y sacar conclusiones a partir de grandes volúmenes de textos ya existentes sin tener que dedicar grandes cantidades de tiempo y esfuerzos a leerlos manualmente, especialmente en campos como la medicina, en el que cada año se publica tal cantidad de estudios que es difícil mantenerse al día con los avances de ciertos campos.

La extracción de relaciones tradicionalmente se ha realizado tratando de identificar relaciones

binarias, bien frase a frase o teniendo en cuenta dos o tres frases consecutivas[Zel03] [Cra99], pero recientemente se han creado algoritmos que identifican una relación n-aria a partir de todas las menciones en las que aparece, como el de Peng et al[Pen17], que utiliza grafos LSTM para modelar el contexto e información de cada frase y las conexiones entre ellas. Este proyecto se centrará en relaciones binarias específicamente entre entidades del tipo 'Persona', y usando correferencia para acceder a la información de otras frases relevantes.

4. RECOGIDA Y LIMPIEZA DE DATOS

4.1. Creando un scraper para Archive of our Own

En el momento en el que decidí utilizar los fanfics de *Good Omens* para el proyecto, dicho libro tenía unos 22000 fanfics en Archive Of Our Own (AO3 para abreviar). Sin embargo, de todos esos relatos sólo me interesaban los que están en inglés y los que realmente contuvieran texto (puesto que, aunque AO3 se centra en relatos, permite alojar todo tipo de archivos multimedia).

Por suerte, AO3 fue creado con la intención específica de funcionar como archivo, por lo que tiene una herramienta de búsqueda y filtrado muy completa y sencilla de usar. Esta herramienta permite filtrar por características como título, autor, idioma y cantidad de palabras, pero su mayor utilidad viene de su sistema de etiquetado. AO3 permite a los autores añadir tantas etiquetas como quieran para que los posibles lectores puedan saber más de su obra a simple vista: temática, personajes principales, parejas en las que se centra, qué medio utiliza, si hay ilustraciones, si trata sobre un evento de la historia original particular... Las etiquetas añaden una gran cantidad de información sobre las historias a las que acompañan, y aunque no es obligatorio poner ninguna, en general los autores se preocupan de etiquetar correctamente sus obras.

AO3 tiene etiquetas específicas para indicar que una obra no es principalmente texto: *'Fanart'*, para ilustraciones, y *'Podfic'* para archivos de audio, así que aproveché la herramienta de búsqueda para llevar a cabo un primer filtrado que eliminara todas las obras que las contuvieran, además de todas las que no estuviesen en inglés. El resultado fue un subconjunto de 20190 fanfics, todos en inglés y cuyos autores no habían incluido ninguna etiqueta que indicara que no fuera puro texto. La herramienta además genera un link permanente que siempre lleva a este subconjunto particular, por lo que no es necesario utilizar esta herramienta nada más que una vez.

Una vez localizado el conjunto de textos y el link a los mismos, viene la parte de crear el

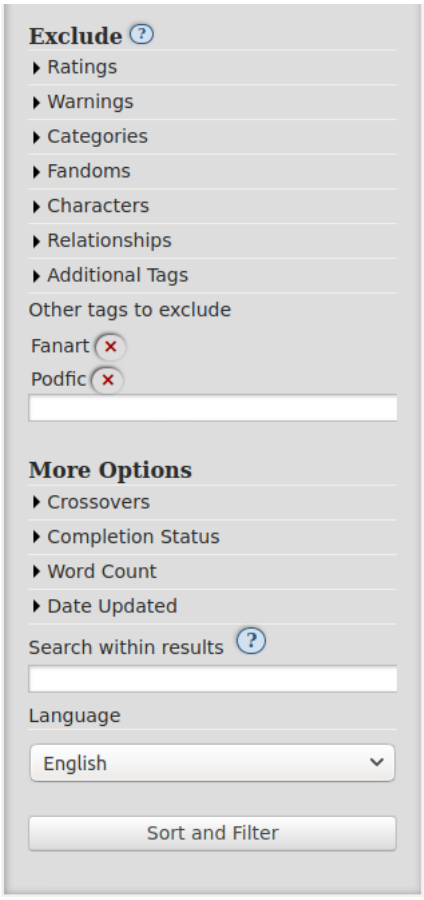


Figura 1: Herramienta de filtrado de AO3. Permite excluir (o incluir) obras que contengan etiquetas específicas, así cómo aquellas no escritas en un idioma particular

scraper en sí. Utilizando la herramienta de inspeccionar elemento de *Firefox* para explorar la estructura del sitio, y enseguida se hizo obvio que los fanfics estaban organizados en páginas con un máximo de 20 fanfics cada una. En el HTML de la página, cada fanfic se presenta dentro de una clase llamada *'work blurb group'*. No se puede extraer un link de descarga directamente de ésta clase, pero sí el identificador del fanfic.

En AO3, cada fanfic tiene un número que lo identifica de forma única. Es posible acceder a la página de cualquier fanfic simplemente añadiendo ese número al final de 'https://www.archiveofourown.org/works/' en la barra de direcciones, y en esa página sí que se pueden encontrar links de descarga.

Por tanto la idea básica para el *scraper* es utilizar las librerías *requests* y *BeautifulSoup* de python para explorar los veinte *'work blurb group'* de cada página, localizar el identificador de cada uno, utilizarlo para acceder a la página del fanfic y extraer el link de descarga. Y así con cada página del listado, hasta llegar a la última. La figura 3 ilustra el proceso con un esquema.

El proceso de descarga de archivos, en principio, tendría estos pasos:

Figura 2: Exploración de la estructura de la página AO3 usando la herramienta 'Inspeccionar elemento' de *Firefox*. Se puede ver que el sitio utiliza una clase HTML llamada *'work blurb group'* para mostrar cada obra.

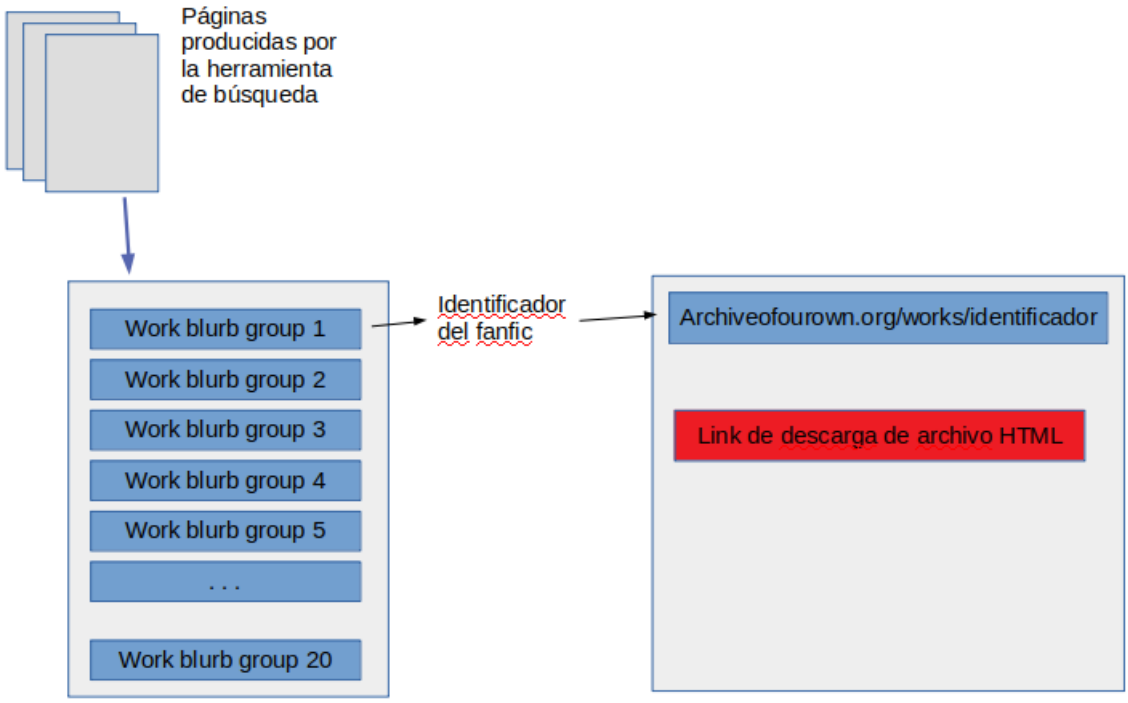


Figura 3: Concepto para el *scraper*. El objetivo es obtener los links de descarga navegando las páginas de búsqueda.

1. Enviar una petición HTTP GET al link permanente del conjunto de datos, generado por la herramienta de búsqueda de AO3.

2. Iterar entre los 20 *'work blurb group'* y extraer el identificador de cada uno.

3. Usar el identificador para acceder a la página de cada fanfic, extraer el link de descarga de la página, y descargar el fanfic como archivo HTML. Hacer esto con los 20 identificadores.

4. Pasar a la siguiente página y repetir, hasta llegar a la última.

Utilizando la librería *requests* de python, el primer paso es trivial, y se puede observar en la figura 5.

Encontrar los identificadores tampoco es complicado. Se puede apreciar en 3 que el identificador del fanfic también es el ID del objeto *'work blurb group'* al que pertenece, y expandiendo la clase se puede ver que el identificador completo se puede encontrar dentro del objeto, como un objeto de tipo *h4*. Por tanto, usando *BeautifulSoup* para manejar los datos resultantes de la petición HTTP GET como objeto HTML, se pueden obtener todos los objetos *'work blurb*

```
40        current_page = 1
41        while current_page < number_of_pages:
42                blurbs = soup.find_all(class_='work blurb group')
43                #print('current page: ',current_page) #debug
44
45                for blurb in blurbs:
46                        #filter out fics that don't contain text
47                        contains_text = check_for_text(blurb)
48
49                        work_id = (blurb.find('h4')).find('a')
50                        if contains_text: work_links.append('https://archiveofourown.org'+work_id['href'])
51                        else:
52                                discarded_links.append('https://archiveofourown.org'+work_id['href'])
53                                #print('out:', work_id['href'])
54                #end 'for blurb' loop
55
56                current_page +=1
57                next_page_link = page_link.replace('&page=1&','&page='+str(current_page)+'&')
58                while True: #wait out if too many requests
59                        page = requests.get(next_page_link)
60
61                        if page.status_code == 429:  #Too Many Requests
62                                print('Sleeping...')
63                                time.sleep(120)
64                                print('Woke up')
65
66                        else: break
67
68                soup = BeautifulSoup(page.content, 'html.parser')
69
70        #end while loop
```


Figura 4: Código perteneciente al *scraper* 'ao3_link_scraper'. Utiliza un bucle *while* para iterar entre las páginas de la búsqueda, y en cada página, usa la librería *BeautifulSoup* para extraer los objetos 'work blurb group' en una lista llamada 'blurbs' (línea 42). De cada 'blurb' extrae el identificador del fanfic y comprueba si tiene texto (líneas 46-49), y si lo contiene forma el enlace a la página del fanfic y lo añade a una lista llamada 'work_links' (línea 50). Si no contiene texto, se añade a otra lista llamada 'discarded_links' (línea 52).

```
29 def get_work_links(page_link):
30         page = requests.get(page_link) #get first page of the archive
31         soup = BeautifulSoup(page.content, 'html.parser')
32 
33         #figure out how many pages in total there are
34         page_list = (soup.find(class_='pagination actions')).find_all('li')
35         number_of_pages = int(page_list[len(page_list)-2].text) #there are number_of_pages pages in total
36 
```

Figura 5: Código perteneciente al *scraper* 'ao3_link_scraper'. Utiliza la librería *requests* para enviar una petición HTTP GET al link permanente del conjunto de datos (línea 30), y *BeautifulSoup* para navegar el resultado como un objeto HTML del que poder extraer datos útiles, como la cantidad total de páginas (líneas 33-35).

group' usando la función *find(class_=<nombre clase>)*, cuyo resultado es una lista con los 20 objetos, sobre los cuales se itera para encontrar los identificadores usando de nuevo la función *find()*. En la figura 4 se puede observar un fragmento del código que realiza este trabajo; el código completo se puede consultar en [placeholder ref].

Las complicaciones empiezan una vez se tienen los identificadores. Para formar la dirección completa, hay que añadir el identificador al final de *'https://www.archiveofourown.org'*, mandar otra petición HTTP GET a dicha dirección, buscar ahí el link de descarga, solicitarla, esperar a que la descarga termine, y repetir todo esto otras 19 veces hasta tener descargados todos los fanfics de la página. Esto significa que por cada iteración del bucle que explora cada página es necesario introducir otro bucle que haga las descargas.

La última parte, la de pasar a la página siguiente, es más complicada de explicar que de ejecutar. Todas las páginas de resultados de búsqueda de AO3 contienen botones para avanzar, retroceder y saltar a páginas concretas. Es posible saber cuántas páginas en total tiene la búsqueda simplemente observando el texto del botón de la última, tal y como se ve en la figura 6. No se aprecia, pero la clase HTML a la que pertenece dicho botón se llama *'pagination actions'*, y es posible extraerla gracias a la función *find(class_=<nombre_clase>)* de *BeautifulSoup*. Y ya con ese objeto, se puede volver a utilizar la función *find()* para buscar todos los objetos hijos de la clase *'pagination actions'* que sean de tipo *li*. El último será el que contenga la cantidad total de páginas, y solicitar la siguiente consiste simplemente en sustituir la referencia en el link a la página 1 por una referencia a la última página. En la figura 5 se ve parte del código que realiza este proceso; el código completo se puede consultar en el anexo [placeholder ref].

Es evidente que la parte de solicitar las descargas en un bucle anidado ralentiza el programa, enturbia el código y además, hace que sea complicado parar o interrumpir el programa si hay algún error de red, pues para reanudar la ejecución por donde se quedó sería necesario almacenar

Figura 6: Navegación de páginas de búsqueda de AO3. Todos los botones vienen con su número de página, y se puede ver cuál es la última

en alguna parte el número de página por el que iba y el número del fanfic dentro de esa página, y programar los bucles para que salten directamente a la iteración deseada.

Ninguna de estas cosas me convenía, ya que descargar más de 20000 archivos ya iba a ser lento de por sí y hacerlo de una sentada sería prácticamente imposible, de modo que decidí dividir el programa en dos: uno que llamé 'link *scraper*' y otro '*file scraper*'.

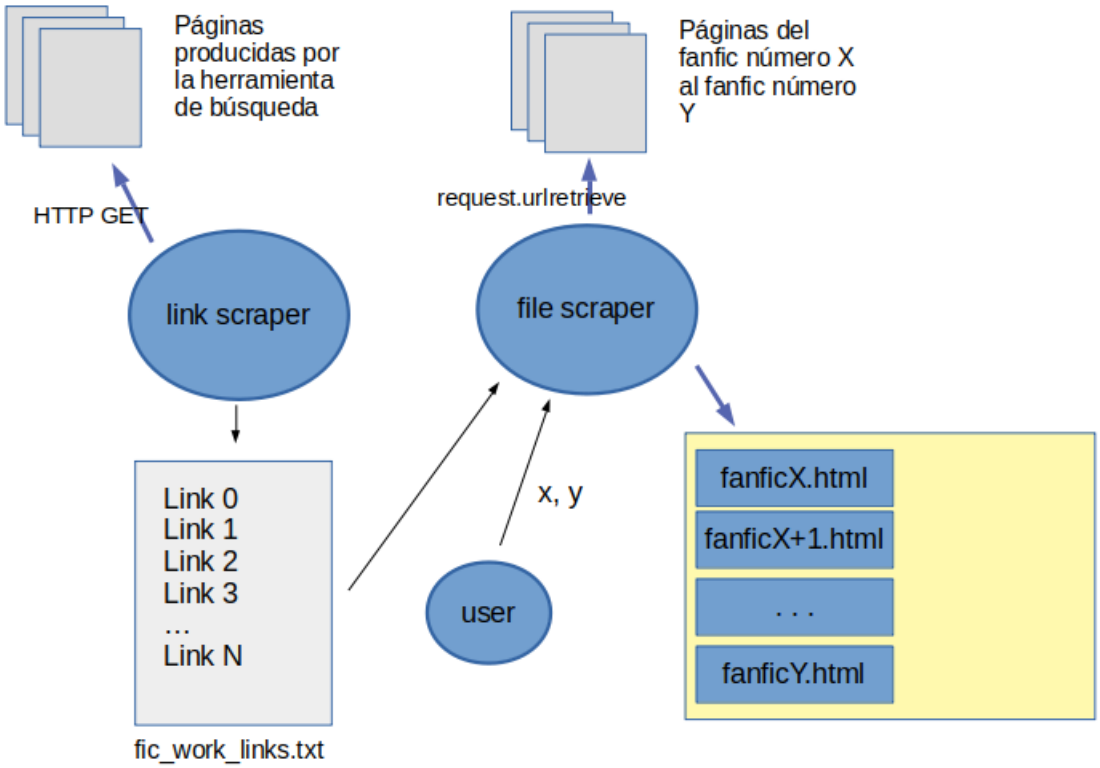


Figura 7: Proceso de descarga de fanfics de AO3 utilizando los programas 'ao3_link_scraper.py' y 'ao3_file_scraper.py'.

El link *scraper* se ejecutaría una vez y exploraría todas las páginas de búsqueda, extrayendo los links a los fanfics de cada una, y los almacenaría en un archivo de texto. Por tanto, al terminar su ejecución este *scraper* ha generado un archivo llamado '*fic_work_links.txt*' que almacena los enlaces a cada fanfic. El *file scraper* utiliza esta lista para saber dónde buscar las descargas, y el usuario le indica en la línea de comando los índices que acotan el tramo de la lista a descargar, tal y como se ilustra en el esquema de la figura 7. De este modo, es posible indicarle al programa que descargue desde el link 0 al link 1000 de la lista, permitiendo descargar los 20000 archivos en porciones manejables. Además, el programa anuncia en pantalla qué link está siendo descargado en cada momento, por lo que si sucede un error de red mientras descargaba el link número 866, es posible reanudar el programa fácilmente e indicarle que continúe desde el 866 al 1000 [placeholder ref].

Esta división del trabajo en dos programas además me daba la oportunidad de introducir con sencillez un segundo filtrado durante el proceso de exploración que realiza el link *scraper*. Si el primer filtrado se encargaba de cribar los fanfics que habían sido etiquetados por sus autores como imágenes o audio, este segundo filtrado pretende detectar los fanfics que tampoco contienen texto, pero no han sido etiquetados como tal por sus autores. Para ello usé el criterio de la relación palabras/capítulo de cada fanfic: si una obra tiene menos de 40 palabras por capítulo, se considera como fanfic "sin texto", y se elimina. Escogí 40 palabras como umbral tras investigar un poco con la herramienta de búsqueda de AO3, que como se puede ver en la figura 1, tiene una opción para filtrar por cantidad total de palabras. Tras probar varios umbrales, 40 parecía ser el que descartaba todas las obras sin texto sin sacrificar muchos microrrelatos en el proceso.

Introducir este filtrado en el *scraper* fue sencillo, puesto que el número de palabras y capítulos de la obra es información que se puede extraer de la clase *work blurb group* de cada fic. Todo esto se realiza desde la función *check_for_text*, y en la figura 4 se puede ver cómo el bucle llama a dicha función; el código completo se puede consultar en [placeholder ref]. Por tanto, el link *scraper* realiza estos pasos:

1. Enviar una petición HTTP GET mediante la librería *requests* al link permanente del conjunto de datos, generado por la herramienta de búsqueda de AO3.
2. Iterar entre los 20 *'work blurb group'*, comprobar si contienen texto, y descartar los identificadores de los que no.
3. Utilizar cada identificador para generar el link de la página de cada fanfic y almacenarlos en un archivo de texto.
4. Pasar a la siguiente página y repetir, hasta llegar a la última.

fig: pantallazo ejecución file scraper

Por su parte, el *file scraper* realiza estos pasos:

1. Abrir el archivo *fic_work_links.txt* y extraer la lista de links.
2. Mediante la librería *requests*, realizar una petición HTTP GET al primer link, saltando al siguiente si devuelve un código 404.
3. Extraer el link de descarga HTML de cada página.
4. Solicitar la descarga mediante *request.urlretrieve*. Guardar el archivo resultante en la carpeta adecuada en el sistema.
5. Repetir con todos los links de la lista.

El manejo del código de error 404 (Page Not Found) es bastante importante en este *scraper*, puesto que entre el momento en el que se almacenó el link del fanfic mediante el primer *scraper* y el momento en el que el segundo *scraper* lo utiliza para la descarga pueden haber pasado varios días. En ese tiempo, el autor del fanfic puede haber decidido borrar el fanfic de AO3, o haberlo hecho privado, y de ahí que el *scraper* reciba un 404. Un simple *try-catch* detecta el código 404 y simplemente pasa al siguiente link, como se puede consultar en el anexo [placeholder ref].

El otro error que ambos *scrapers* necesitaban manejar es, naturalmente, el error 429 (Too Many Requests). En las líneas 58-66 de la figura 4 se puede ver cómo se utiliza un *try-catch* que envuelve la petición HTTP GET para detectar el status 429 y, en vez de pasar al siguiente link, se lanza una espera de dos minutos tras la cual vuelve a solicitar la página. Antes de incorporar este código a los *scrapers* creé un pequeño programa de prueba, para ver cuánto tardaba AO3 en enviar un 429 y cuánto tiempo de espera requería antes de volver a aceptar solicitudes; dicho programa se puede consultar en [placeholder ref].

El resultado de la ejecución de estos *scrapers* es una carpeta con 818,8 MB de archivos HTML.

4.2. Limpieza de datos y creación de datasets

Al terminar el proceso de descarga, acabé con un conjunto de archivos HTML y un archivo TXT con una lista de los *path* de todos ellos.

La principal tarea de limpieza de datos es, por tanto, convertir los archivos HTML a texto. Para ello utilicé la librería *HTML2Text*, que como cuyo nombre dice sirve para eso mismo. Sin embargo, los fanfics en HTML no contienen sólo el texto del fic en sí, sino que también

contienen todos los metadatos del mismo: etiquetas, *rating*, resumen, comentarios del autor, entre otros, con lo que tras limpiar el archivo HTML con *HTML2Text* el resultado no era el texto puro del fanfic. Tuve que crear varias funciones y ayudarme de *Beautiful Soup* para limpiar todos estos metadatos y dejar únicamente el texto en sí, sin llevarme por delante parte del texto ni dejarme notas del autor entre capítulos.

Al principio puse estas funciones dentro de cada programa que necesitaba manejar los textos, pero obviamente enseguida se volvió muy aparatoso, por lo que consolidé todas las funciones de limpieza en un único archivo, *fanfic_util.py*, y creé dos clases para encapsular el uso de estas funciones:

- FanficGetter, que se encarga de proveer el texto limpio de un fanfic (o lista de fanfics) a demanda. Al principio devolvía los textos como una lista de *string*, pero luego resultó más útil que devolviera una lista de objetos *Fanfic* que pudiera devolver los capítulos por separado o juntos en un mismo *string*, además del identificador del fanfic.
- FanficHTMLHandler se encarga de extraer información de los metadatos del archivo HTML de un fanfic, como por ejemplo los personajes principales, las relaciones, las etiquetas, el número de capítulos y su clasificación.

La creación de la clase *Fanfic* surgió a mediados del proyecto, cuando el límite de 100000 caracteres de *CoreNLP* hizo necesario poder acceder al texto de cada fanfic dividido en capítulos. *Fanfic* por tanto tiene un atributo *chapters*, que es la lista de *string* con los capítulos, y un método *get_string_chapters()* que devuelve todos los capítulos en un único *string*.

Sin embargo, el acceso a los datos sigue basándose en una lista de *paths* guardada en un archivo TXT. Es un sistema muy rudimentario (ilustrado en la figura), pero no vi necesidad de trasladarlo a una base de datos propiamente dicha, puesto que nada en la creación de una base de datos me ahorraba nada del trabajo de crear *fanfic_util*, y desde el punto de vista del resto de programas es igual acceder al texto de un fanfic a través de FanficGetter que de una función que recupere el texto de una base de datos. Únicamente intercambiaría el tiempo de limpiar los textos por el de conectar con la base de datos y extraer su información.

Para la parte de identificación de relaciones del proyecto se necesita filtrar los fanfics originales en tres subconjuntos: fanfics centrado en el romance, en la amistad, y en la enemistad.

5. EXTRACCIÓN DE DATOS A PARTIR DE TEXTO

Explicar los programas que hice para familiarizarme con

5.1. Algoritmo de identificación de entidades

En la identificación de entidades, se considera una entidad a los personajes, los lugares y las instituciones, entre otras cosas, que haya sido nombrada en el texto. Un algoritmo capaz de identificar entidades nombradas tiene que poder dividir un texto en tramos y asignarle una etiqueta de entidad ("Persona", "País", etc) a cada uno. Esta tarea además requiere que las palabras del texto hayan sido previamente etiquetadas con su rol morfológico.

Por estos motivos, la librería NLTK parecía la más idónea para la tarea. Es una librería de python que contiene herramientas básicas para el análisis de texto, y en particular me interesaba que venía con un *part of speech tagger* (es decir, un identificador de rol morfológico) ya programado y entrenado. NLTK también viene con un identificador de entidades ya entrenado, pero quería programar uno que fuera más preciso y adaptado a mi conjunto de datos.

Además del identificador de rol morfológico, NLTK también tiene una clase llamada *ChunkParser* cuyo trabajo es dividir un texto en tramos. Todas las funciones de la librería que se encargan de dividir y/o etiquetar texto (como el identificador de rol morfológico) heredan de alguna versión de la clase *ChunkParser*, de modo que la idea para el algoritmo era modificar la clase *ChunkParserI* para convertirla en un identificador de secuencias basado en características. El código utilizado en este proyecto está basado en el tutorial de Ivanov en *Natural Language Processing for Hackers* [iva]

Un identificador de secuencias basado en características trata de asignar un peso a un tramo concreto, y según el peso, le asigna una etiqueta u otra. Este peso se calcula como una función de las características del propio tramo, así como de los tramos que le preceden. El programador puede elegir las características que considere más importantes, pero hay algunas que son bien conocidas como las más importantes para reconocer entidades, como:

- El rol morfológico de la palabra actual, las anteriores y las siguientes.
- La forma de la palabra, las anteriores y las siguientes (si empiezan por mayúscula, si tienen signos de puntuación, si son siglas, etc)
- Los prefijos y/o sufijos de la palabra actual, las anteriores y las siguientes.
- Si la palabra anterior ha sido identificada como una entidad o no.

El conjunto de características de cada tramo se llama vector de características, y se utiliza para calcular un "peso"que se corresponde con la probabilidad de que un tramo X con un vector de características V tenga una etiqueta Y. El algoritmo al final asigna a cada tramo la etiqueta cuyo

peso sea el más alto.

El cómo se calcula exactamente ese peso depende del modelo matemático a utilizar. A la versión modificada de ChunkParserI para la identificación de entidades la llame NERChunker (NER por Named Entitiy Recognition), y tiene tres versiones:

- NERChunkerv1 y NERChunkerv3 utilizan un modelo de regresión logística (también llamado modelo de entropía máxima), a través de la clase MaxentClassifier de NLTK. Para que NLTK pueda utilizar esta clase correctamente, es necesario tener instalado el módulo Megam para python, que no viene incluído en NLTK. La única diferencia entre la versión 1 y la 3 de este chunker es que la 3 maneja las estructuras de NLTK para oraciones y etiquetas de forma ligeramente más rápida.
- NERChunkerv2, que utiliza un modelo de naïve Bayes a través de la clase ClassifierBasedTagger de NLTK.

Las versiones *v1* y *v3* de *NERChunker* obtuvieron los mejores resultados en la evaluación, y la *v3* es algo más rápida, por lo que es la versión definitiva del identificador de entidades. Todas estas versiones, junto con sus funciones auxiliares, se encuentran encapsuladas en el archivo *NERChunkers.py*, para ser utilizadas donde se las necesite.

fig: evaluaciones de las versiones de NERChunker y el NER de NLTK

Puesto que tanto los clasificadores de regresión logística como los de *naïve* Bayes son algoritmos de aprendizaje supervisado, antes de poder utilizar (o evaluar) cualquiera de las versiones de *NERChunker* era necesario entrenarlas con un conjunto de datos ya etiquetados. El problema aquí es que NLTK, a pesar de incluir un corpus muy extenso en la propia librería, sólo tiene dos conjuntos de datos para identificación de entidades: uno en español y el otro en holandés. Todos los textos a analizar en el proyecto están en inglés, obligándome a buscar un conjunto ajeno a NLTK y finalmente decidiéndome por *Groningen Meaning Bank* (GMB). GMB es un *dataset* para identificación de entidades específicamente en inglés, grande, con una gran variedad de etiquetas de entidad y, sobretodo, con un formato de etiquetado sencillo de entender,cosa importante puesto que al ser ajeno a NLTK, GMB utiliza etiquetas distintas que son necesario adaptar para que *MaxentClassifier* pueda trabajar con ellas.

GMB utiliza la notación IOB para etiquetar entidades, y separa cada palabra de la siguiente por un carácter de nueva línea, y cada frase, por dos. De modo que la frase *"Mr. Blair left for Turkey Friday from Brussels."* en GMB tendrá el aspecto de la figura 8.

Mr.	NNP	B-PER
Blair	NNP	B-PER
left	VBD	O
for	IN	O
Turkey	NNP	B-GEO
Friday	NNP	B-TIM
from	IN	I-TIM
Brussels	NNP	I-TIM
.	.	O

Figura 8: Frase etiquetada por GMB. De izquierda a derecha, las columnas representan la palabra a etiquetar, la etiqueta de rol morfológico, y la etiqueta IOB.

Cuando el programa detecta una entidad de tipo persona, etiqueta como 'B-PER' la primera palabra de la secuencia, mientras que el resto de palabras dentro de la secuencia son etiquetadas como 'I-PER'. Similarmente, si la entidad es de tipo geográfico las etiquetas usadas serán 'B-GEO' y 'I-GEO', si es de tiempo serán 'B-TIM' y 'I-TIM', etc. Si una palabra no forma parte de ninguna secuencia de entidad, se etiqueta como 'O'.

NLTK, por su parte, no utiliza la notación IOB ni caracteres de nueva línea, sino que utiliza una estructura de datos propia de tipo árbol que encapsula cada palabra y cada tramo con su etiqueta. La misma frase etiquetada por NTLK tiene el aspecto de la figura 9.

Como se ve, en vez de usar etiquetas IOB, NLTK organiza las palabras y su etiquetas en una estructura de árbol. La raíz, S, indica el inicio de la frase (Sentence), y las etiquetas de entidad son nodos.

En horizontal queda así:

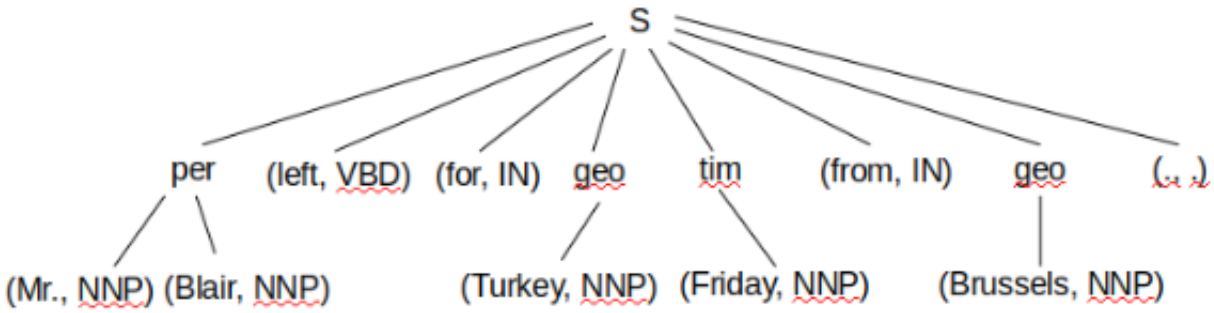


Figura 9: Frase etiquetada por NLTK. Las etiquetas de entidad se encuentran en los nodos, encontrándose todas las palabras pertenecientes a una secuencia de entidad en la profundidad 2 del árbol. Las palabras que no pertenecen a ninguna secuencia de entidad se encuentran en la profundidad 1. Cada hoja del árbol contiene una tupla formada por la palabra y su etiqueta de rol morfológico.

(S, [(per, [(‘Mr.’, NNP), (‘Blair’, NNP)]), (‘left’, VBD), (‘for’, IN), (geo, [(‘Turkey’, NNP)]), (tim, [(‘Friday’, NNP)]), (‘from’, IN), (geo, [(‘Brussels’, NNP)]), (‘.’,.)])

Fue más o menos a estas alturas del proyecto cuando decidí separar el proceso de entrenar el identificador de entidades y el de utilizarlo para etiquetar texto nuevo en dos programas distintos (NERTrainer y NERTagger, respectivamente). Acceder a los textos de GMB y transformar sus etiquetas a un formato que NLTK pueda entender y acceder a los textos de la base de datos de fanfics y preprocesarlos para su posterior etiquetado mediante el programa ya entrenado han resultado serdos procesos muy distintos, y dividirlo parecía la mejor manera de tener un código limpio y claro.

listar en alguna parte las librerías: BeautifulSoup, pandas

5.2. Algoritmo de identificación de relaciones

Las relaciones que definen un fanfic son las amistades, los enemigos y, principalmente, los amantes. Extraer relaciones a partir de texto natural es una tarea compleja de por sí, y tratar de detectar este tipo concreto de relaciones en literatura de género puede presentar un reto mayor, puesto que las relaciones sentimentales tienden a representarse de forma implícita, de modo que el lector aprende quiénes son amigos y quiénes enemigos a través de las acciones de los personajes. Además, la ambigüedad, la heterogeneidad y la experimentación son partes naturales de cualquier proceso creativo, por lo que un conjunto de obras no representará una misma relación de forma uniforme, incluso si es entre los mismos personajes.

Encontrar una estrategia para abordar este problema requiere exploración y creatividad, y ya que había empezado el proyecto con NLTK, me pareció natural comenzar la búsqueda por ahí.

El extractor de relaciones de NLTK funciona mediante reglas: después de extraer las entidades nombradas del texto, se puede utilizar el módulo *relextract* para dividir el texto en listas de fragmentos del texto que contienen dichas entidades, y aplicar reglas basadas en expresiones regulares que definan la relación entre las entidades. La regla puede incluir etiquetas de rol morfológico en la expresión regular, y *relextract* permite filtrar por etiqueta IOB, lo que le da algo más de flexibilidad.

Por ejemplo, para extraer una relación de lugar entre una organización y una localización, se puede crear una expresión regular que busque la palabra clave 'in' en el texto, e indicarle a *relextract* que sólo te interesan los fragmentos de texto que tengan una entidad de tipo 'ORG' seguida de una entidad de tipo 'LOC':

```
IN = re.compile(r'.*\bin\b(?!\b.+ing\b)')

for doc in parsed_docs:
        for rel in nltk.sem.extract_rels('ORG','LOC', doc, pattern=IN):
                print(nltk.sem.show_raw_rtuple(rel))
```

Listado 1: Ejemplo de código que utiliza el módulo *regexp* de NLTK para extraer relaciones de lugar y mostrarlas por pantalla. Adaptado del capítulo 7 de Natural Language Processing with Python[Bir12]

Existen proyectos que utilizan este módulo de NLTK para extraer relaciones como *DateOfBirth* y *HasParent* [jos17], pero es evidente que es un método poco adecuado para el tipo de proyecto que estaba intentando hacer.

Estos programas basados en reglas dependen de localizar palabras claves en el texto, y aunque existen palabras clave para identificar relaciones sociales (*"love"*, *"kiss"*, *"hug"*, *"friend"*, *"kill"*, *"hate"*, etc), lo cierto es que la naturaleza de la expresión literaria hace que este método, incluso a simple vista, parezca bastante ingenuo. No sólo es perfectamente posible expresar amor, amistad y odio sin usar "palabras clave.ªsociadas con dichos sentimientos, sino que en un texto literario raramente se escribe explícitamente *Romeo loved Juliette*, si no que es más normal encontrar estructuras como *'I love you', said Romeo*. En una frase así, no se menciona explícitamente a Julieta, pero pero un lector humano sabe si se refiere a ella por el contexto de la escena. Pero un programa que únicamente se preocupa de las etiquetas IOB de una frase no

será capaz de unir ese *you* con Julieta (ni, ya puestos, el *I* con Romeo).

Descartado el extractor de relaciones de NLTK, empiezo a buscar opciones en otras librerías. El Stanford Natural Language Processing Group publicó un extractor de relaciones como parte de las funciones de CoreNLP, pero las relaciones que está entrenado para detectar (*Live_In, Located_In, OrgBased_In, Work_For, None*) no parecen útiles para el proyecto. Por tanto, entrenar mi propio modelo para relaciones sociales parece la única solución.

Crear un modelo de regresión logística con NLTK, similar al identificador de entidades, requería que el texto ya estuviera etiquetado con las relaciones. Los autores de AO3 usan etiquetas que es posible extraer las relaciones a partir del archivo HTML de cada fanfic, pero es una etiqueta a nivel del texto completo, no a nivel de frase, que es como trabaja NLTK. Dejando de lado NLTK por el momento, decidí explorar soluciones usando clustering y modelado de temas.

5.2.1. Primeras estrategias: Clustering y LDA

Decidir si dos personajes son amigos, enemigos o amantes (sin tener ninguna información previa sobre la obra) puede requerir leer el texto completo, con lo cual es razonable utilizarlo para el análisis y asignar una etiqueta de 'romance', 'amistad' o 'enemistad' al texto en su conjunto, más que etiquetar ciertas palabras y personajes del mismo.

Por tanto, dado un conjunto de textos al azar, podría llegarse a la conclusión de cuáles contienen romance, cuáles amistad y cuáles enemistad observando si hay similitudes entre ellos. Con este enfoque, parece una tarea adecuada para un algoritmo de clustering.

Para comprobar cómo de útil sería esta estrategia, filtré los datos en tres grupos para que en cada uno sólo hubiese fanfics que se centrasen en torno al romance, la amistad y la enemistad respectivamente. Esperaba que teniendo tres conjuntos claros en los que el tema era evidente sirviese para ver cómo de eficaz es el clustering para la tarea general, que sería poder decir si hay romance en un texto incluso si no es el tema central del mismo.

El criterio utilizado para crear estos tres grupos está basado en la longitud de los textos y sus etiquetas. En el caso de las etiquetas es sencillo: los autores casi siempre etiquetan las relaciones románticas en sus relatos, y a menudo también las amistades, para hacer que sus historias sean más fáciles de encontrar por aquellos que quieran leerlas. En AO3, las etiquetas románticas

tienen el formato 'Personaje A/Personaje B', mientras que las etiquetas de amistad son 'Personaje A & Personaje B' o 'Personaje A and Personaje B', con lo que extraer estas etiquetas del archivo HTML es sencillo utilizando expresiones regulares [placeholder ref]. Además de tener una etiqueta que valide la expresión regular, sólo tuve en cuenta aquellos relatos que sólo tenían un capítulo. De esa forma, esperaba poder eliminar historias largas y elaboradas que contienen romance, pero que principalmente son una historia costumbrista o de aventuras. Limitando la longitud de la historia a un capítulo, todo el romance o la amistad queda condensada en dicho capítulo.

Crear un conjunto con relaciones de enemistad u odio es una tarea más complicada, ya que los usuarios de AO3 no tienen un formato oficial para las mismas y no se suelen etiquetar. Al final creé un conjunto de las etiquetas que los autores comúnmente utilizan como aviso de que su historia contiene violencia o abusos, como 'Rape/Non-con', 'Torture', 'Graphic Depictions of Violence' o 'Dead Dove: Do Not Eat'[placeholder ref]. Esperaba que, entre esas etiquetas y la limitación de longitud, pudiese aislar un conjunto de relatos que sirvieran de modelo para la relación de enemistad.

El resultado fueron 12520 relatos en el conjunto de romance, 784 en el de amistad y 155 en el de enemistad. Para equilibrar los *datasets*, reduje el conjunto de romance a 220 y el de amistad a 180.

Una vez creados los conjuntos, utilicé la librería *Scikit-Learn* en conjunto con NLTK para crear el programa de clustering. Para preprocesar el texto se utilizan los métodos de NLTK para *tokenizar* y *lemmatizar* los textos, además de crear un conjunto de *stopwords*, palabras comunes en inglés pero que no aportan mucha información sobre el mismo (preposiciones, pronombres, puntuación, demostrativos, etc). Después de preprocesar el texto se procede a extraer las características relevantes del mismo, para lo cual se utiliza el módulo *TfidfVectorizer* de *Scikit-Learn*. Su trabajo es 'vectorizar' el texto de manera que sus características principales queden expresadas en un formato que el algoritmo de clustering pueda entender, cosa que hace asignando un peso a cada palabra dependiendo del número de ocurrencias de la misma (esto se llama *bag of words*). Hay vectorizadores que asignan más peso cuanta más frecuencia, pero esto hace que palabras muy comunes pero con poco valor informativo roben protagonismo a palabras menos frecuentes pero más interesantes. El vectorizador 'Tf-Idf' (*Term frequency-Inverse document frequency*), en cambio, multiplica la frecuencia de una palabra en un documento por un componente *idf* que, como se ve en la fórmula 1, está basado en la frecuencia de ese término en

$$\text{idf}(t) = \log \frac{1+n}{1+\text{df}(t)} + 1 \tag{1}$$

Figura 10: Componente idf del vectorizador Tf-Idf. *t* se refire al término cuyo peso está determinando, *n* al número total de documentos y *df* a la frecuencia de *t* en este documento.

todos los documentos. Los vectores resultantes se normalizan usando la norma de Euclides; más información está disponible en la guía de usuario de *Scikit-Learn* [skl].

Con los textos ya preprocesados y convertidos en vectores *tf-idf* se puede crear un modelo de clustering, en este caso utilizando el módulo *KMeans* de *Scikit-Learn*. KMeans es un algoritmo que crea clusters de tal forma que cada uno tenga la misma varianza, minimizando la suma de los cuadrados de las distancias entre los miembros de cada grupo (fórmula 2).

$$\sum_{i=0}^{n} \min_{\mu_j \in C} ||x_i - \mu_j||^2 \tag{2}$$

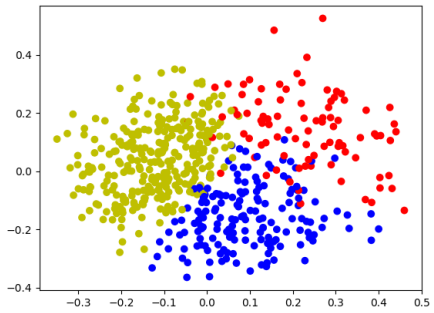
Figura 11: Criterio de la suma de cuadrados. *n* es el total de textos, con *x* perteneciendo a *n*. *C* es el número de clusters, con mu siendo la media de las muestras *x* de cada cluster. El algoritmo KMeans reduce esta suma todo lo posible.

Además del código necesario para crear el modelo KMeans y procesar los daños, añadí código para evaluar el modelo e imprimir un diagrama de puntos con los clusters. Tras probar dos tokenizers distintos y varias combinaciones con los parámetros del vectorizador y el modelo, los resultados se pueden ver en la figura 12.

Los resultados no son muy buenos. Aunque los términos de cada cluster parecen prometedores, ninguna métrica sube del 0.1, lo que indica que las categorías del clustering son sólo un poco mejores que haberlas asignado al azar, y que hay mucho solapamiento entre clusters.

Puesto que para crear el modelo he utilizado datos filtrados a propósito para modelar cada categoría tan bien como fuera posible, con la esperanza de poder usarlo como posible semilla para un sistema más general, esto supone un gran problema.

Busqué entonces otra estrategia, utilizando un modelo de temas más que uno de clustering. El modelado de temas con el algoritmo LDA parece también una buena opción para este problema, puesto que al contrario que el clustering clásico, LDA asigna a cada documento una distribución de temas en cada uno. Esto se ajusta a este análisis, puesto que aunque he intentado crear *datasets* 'perfectos' que traten un único tipo de relación en cada uno, lo cierto es que lo normal



(a) El código utilizado para crear este diagrama fue escrito por Matt L en StackOverflow.

```
maria@maria-P7815:~/Documents/Fanfic_ontology$ python3 toy_relex_kmeans.py
Fetching fic texts...
...fics fetched. Elapsed time:  0.5578843951225281  mins
Creating stopwords and removing character names from texts...
...done in  0.032831732432047525  mins
Creating vectorizer, transforming texts to tf-idf coordinates...
...done in  3.929908561706543  mins
Creating the K-Means model and fitting data...
...done in  0.0027207175890604654 mins
n_samples: 555, n_features: 3

 === Metrics ===
Homogeneity: 0.140
Completeness: 0.150
V-measure: 0.145
Adjusted Rand-Index: 0.115
Silhouette Coefficient: 0.233

Top terms per cluster:
Cluster 0:
kiss, skin, bed, tongue, press, neck, touch, lean, slide, morning,

Cluster 1:
man, child, friend, boy, wine, god, bookshop, young, understand, sat,

Cluster 2:
wings, pain, water, blood, bed, heaven, floor, black, man, form,
```

Figura 12: A la derecha, ejecución de *toy_relex_kmeans*, mostrando su evaluación. La homogeneidad se cumple cuando ningún cluster contiene miembros que pertenezcan a categorías distintas en los datos reales. La completitud se satisface si todos los miembros de una de las categorías reales pertenecen al mismo cluster. A la izquierda, el diagrama creado por el programa.

en un relato es que estén mezcladas.

LDA es un algoritmo que descubre temas de forma no supervisada. Trabaja bajo la asunción de que cada documento es un conjunto de temas, y que cada tema es un conjunto de palabras. Empieza asignando cada a palabra a un conjunto al azar de temas, y en cada iteración mejora la asignación.

Al igual que en el programa anterior, LDA requiere un preprocesado del texto, con lo que utilizo los dos *tokenizers* del algoritmo de clustering. *Scikit-Learn* carece de modelo LDA, por lo que utilizo el de la librería *gensim*. LDA también trata los textos como un *bag of words*, pero no es necesario vectorizarlos antes de usarlos para entrenar el modelo, que devolverá lista de palabras por tema, junto con la probabilidad de que esa palabra pertenezca a dicho tema.

Tras preprocesar los textos de forma similar a como se hizo con clustering y entrenar el modelo LDA, se observan los resultados en la figura 13.

Los resultados son un poco decepcionantes, pues ninguno de los 10 términos más relevantes por tema tiene siquiera un 0.1 % de probabilidad de pertenecer a su tema. Tampoco es sorprende que no sean muy relevantes, pues aparecen muchos pronombres, determinantes e incluso algún número. Aprovechando el etiquetado de rol morfológico de NLTK, se retiran esas palabras y se crea un nuevo modelo, cuya ejecución está en la figura 14.

Figura 13: Ejecución de *toy_relex_lda*, filtrando sólo la puntuación. Se muestra los 10 términos más relevantes de cada tema, y su probabilidad de pertenecer a dicho tema.

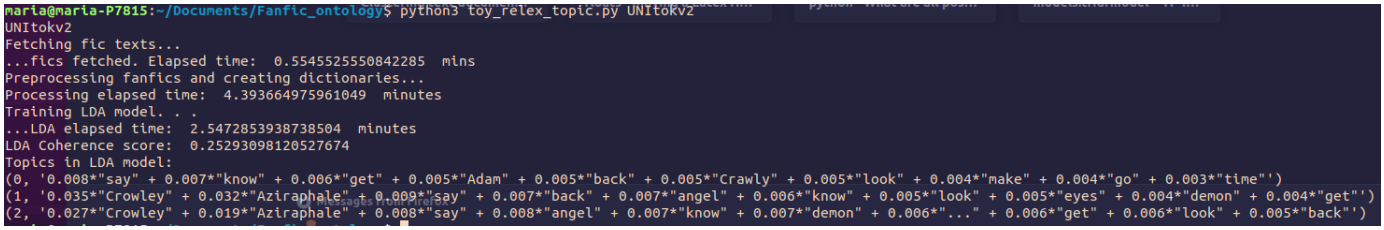


Figura 14: Ejecución de *toy_relex_lda*, filtrando puntuación, pronombres, determinantes, etc. Se muestra los 10 términos más relevantes de cada tema, y su probabilidad de pertenecer a dicho tema.

Para buscar el modelo LDA más eficiente, creé tres modelos que tenían en cuenta diferentes categorías morfológicas. Primero utilizaba el *tagger* de NLTK para identificar categorías morfológicas, y el algoritmo sólo tenía en cuenta aquellas que resultaran relevantes.

- El modelo B tenía en cuenta sustantivos, adverbios y verbos.
- El modelo C tenía en cuenta sustantivos, adjetivos y verbos.
- El modelo D tenía en cuenta sustantivos, adverbios, adjetivos y verbos.

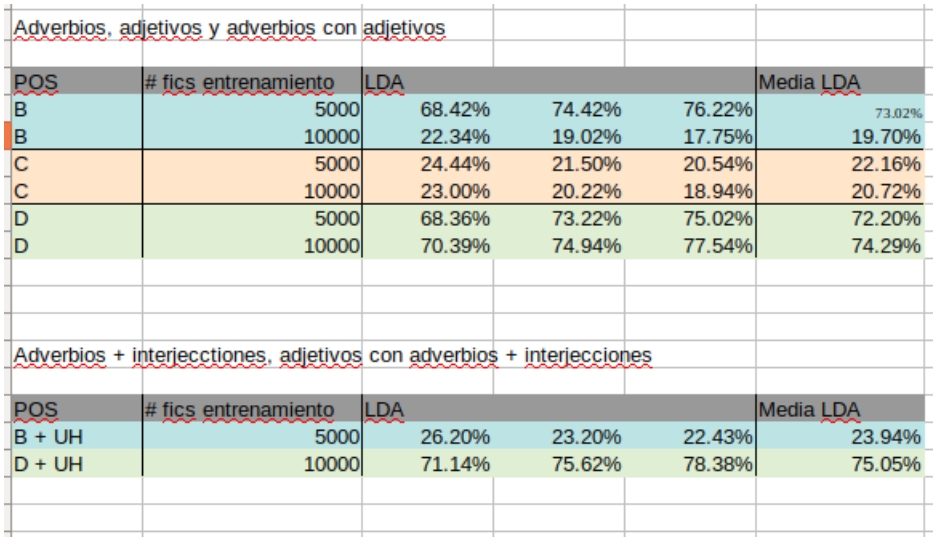
Además de utilizar distintas categorías morfológicas, también probé distintos tamaños para el set de entrenamiento, de modo que cada modelo tiene dos versiones: una entrenada con 5000 fanfics y otra entrenada con 10000.

Para la evaluación de los modelos, simplemente los puse a clasificar textos nuevos, contando la cantidad de aciertos de cada uno y calculando el *hit ratio* de cada uno. Los resultados aparecen en la primera tabla de la figura 15.

$$hit\_ratio = \frac{correct\_guesses}{total\_number\_of\_guesses}$$

Las primeras pruebas mostraron que los modelos B y D eran los que arrojaban mejores resultados. Observando qué otras categorías morfológicas el *tagger* de NLTK puede identificar, pensé

Figura 15: Porcentaje de aciertos de cada modelo. Cada prueba se realizó tres veces.



Adverbios, adjetivos y adverbios con adjetivos

POS	# fics entrenamiento	LDA			Media LDA
B	5000	68.42%	74.42%	76.22%	73.02%
B	10000	22.34%	19.02%	17.75%	19.70%
C	5000	24.44%	21.50%	20.54%	22.16%
C	10000	23.00%	20.22%	18.94%	20.72%
D	5000	68.36%	73.22%	75.02%	72.20%
D	10000	70.39%	74.94%	77.54%	74.29%

Adverbios + interjecctiones, adjetivos con adverbios + interjecciones

POS	# fics entrenamiento	LDA			Media LDA
B + UH	5000	26.20%	23.20%	22.43%	23.94%
D + UH	10000	71.14%	75.62%	78.38%	75.05%

que añadir interjecciones a los modelos podría aumentar su precisión. Llamé B+UH y D+UH a los modelos resultantes, y repetí las pruebas. Los resultados están en la segunda tabla de la figura 15.

Curiosamente, el modelo D fue mejorado ligeramente teniendo en cuenta las interjecciones, pero el modelo B empeoró considerablemente.

5.2.2. Correferencia con CoreNLP

6. EVALUACIÓN DEL SISTEMA

7. REFERENCIAS

Referencias

[Bar68] Ronald Barthes. La mort de l'auteur. *Manteia*, (5), 1968.

[Bir12] Steven Bird. Natural language processing with python. https://www.nltk.org/book_1ed/ch07.html, October 2012.

[Cra99] Mark Craven. Constructing biological knowledge bases by extracting informationfrom text sources. *SMB-99 Proceedings*, 1999.

[Eis18] Jacob Eisenstein. *Natural Language Processing*. MIT Press, Nov 2018.

[Ell18] Lindsay Ellis. Death of the author. https://www.youtube.com/watch?v=MGn9x4-Y_7A, December 2018. Youtube.

[iva] Complete guide to build your own named entity recognizer with python. https://nlpforhackers.io/named-entity-extraction/. NLP for Hackers.

[jos17] Information extraction. https://github.com/rohitjose/InformationExtraction, 2017.

[Pen17] Nanyun Peng. Cross-sentence n-ary relation extraction with graph lstms. *Arxiv*, 2017.

[skl] Feature extraction: Tf–idf term weighting. https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction. Scikit Learn User Guide.

[Stu17] Alasdair Stuart. Dean winchester and commander shepard walk into a bar: Why fanon matters. *Uncanny Magazine*, July/August 2017.

[Swi98] Jonathan Swift. Copyright 101: A brief introduction to copyright for fan fiction writers. http://www.whoosh.org/issue25/lee1a.html#41, October 1998. Woosh Magazine, Birthplace of the International Association of Xena Studies.

[Zel03] Dimitri Zelenko. Kernel methods for relation extraction. *Journal of Machine Learning Research*, (3):1083–1106, 2003.