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, y AO3 tiene una base de datos muy grande y accesible. Además, las comunidades de fanfic 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.

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 han vuelto 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* (Jacob Eisenstein, MIT Press). 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.

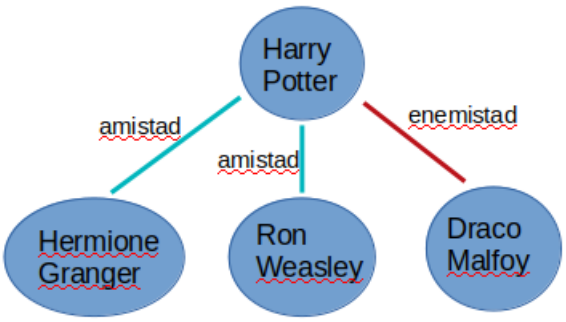
Por ejemplo, es evidente al leer los libros de la saga *Harry Potter* que el texto quiere que pienses que Ron Weasly, el mejor amigo del protagonista, es un chico un poco torpe y bocazas pero con buen corazón, y un buen amigo de Harry. Sin embargo, muchos fans no interpretaron a Ron como torpe y bocazas, 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.

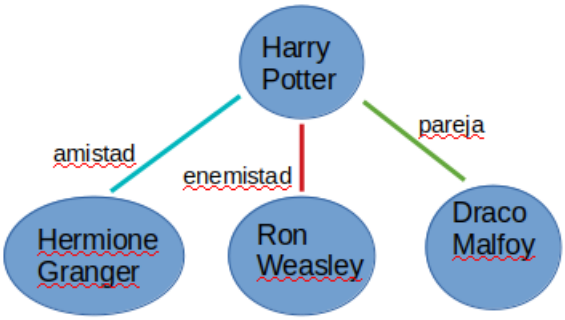
Cuando un fandom alcanza un cierto nivel de madurez, algunas teorías se consolidan y el fandom acaba formando, a nivel de comunidad, una interpretación propia de la obra original. Para distinguir la perspectiva del fandom de la que realmente pretende transmitir la obra original, en los fandoms se distingue 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,



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*

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).

continua parejas y fandom

No todas las parejas son igual de populares en el fandom, sino que por lo general hay una o dos, como mucho tres parejas que monopolizan la atención y creatividad de los fans, y la popularidad de cada una suele ser independiente de si la pareja es canon o no en el material original.

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*. A día de hoy, la

mayoría de fanfics se publican en sitios web de toda índole, destacando entre ellos Archive Of Our Own, que es un archivo open-source y sin ánimo de lucro creado expresamente para alojar obras creadas por fans. Según sus datos de mayo de 2020, tiene más de dos millones de usuarios registrados y más de seis millones de trabajos alojados.

En particular elegí descargar los fanfics basados en *Good Omens*, un libro de Terry Pratchett y Neil Gaiman, tanto por la cantidad de relatos existente como por mi familiaridad con esa comunidad.

Además de la cantidad y variedad de relatos que aloja, los motivos por el que elegí extraer los datos de Archive Of Our Own son su herramienta de búsqueda y filtrado, su sistema de etiquetas y el hecho de que permite descargar el archivo en HTML. Archive Of Our Own permite filtrar fanfics según varios parámetros y genera un enlace a ese subconjunto de relatos particular, muy útil para descargar una gran cantidad de datos.

por qué escogí AO3 sobre FF.net y Wattpad

3. INTELIGENCIA ARTIFICIAL Y ANÁLISIS DE TEXTO

Proyecto que usa NLTK para extraer relaciones de tipo DateOfBirth y HasParents de Rojit Jose [jos17] Stanford Natural Language Processing Group Kernel para extracción de relaciones [Zel03] Resolución de correferencias mediante un modelo basado en entidades [Wic09] Canonicalización [Cul07]

mirar intro del libro de eisenstein

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).

explicar bag of words + función que calcula peso en función de vector características

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 etiquetas. AO3 permite a los autores añadir tantas etiquetas como quieran para que los posibles lectores puedan saber más de su obra a simple vista: temá-

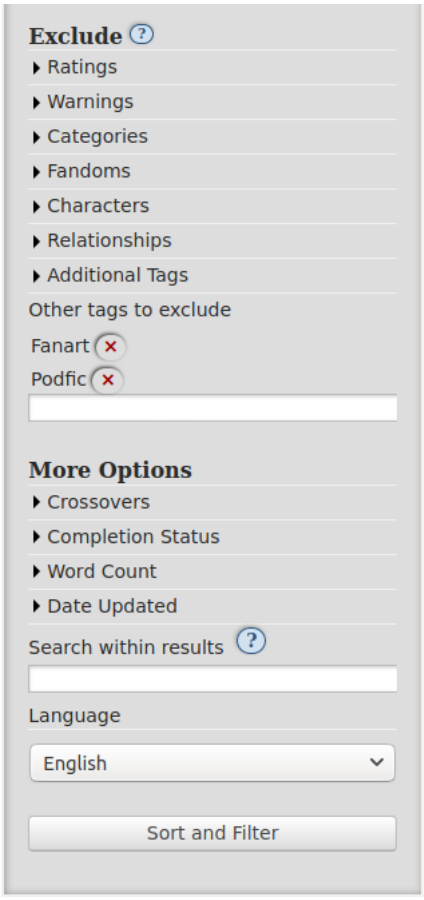


Figura 1: Herramienta de filtrado de AO3. Permite excluir (o incluir) obras que contengan etiquetas específicas, así cómo aquellas no escritas en un idioma particular

tica, 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 *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

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.

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.

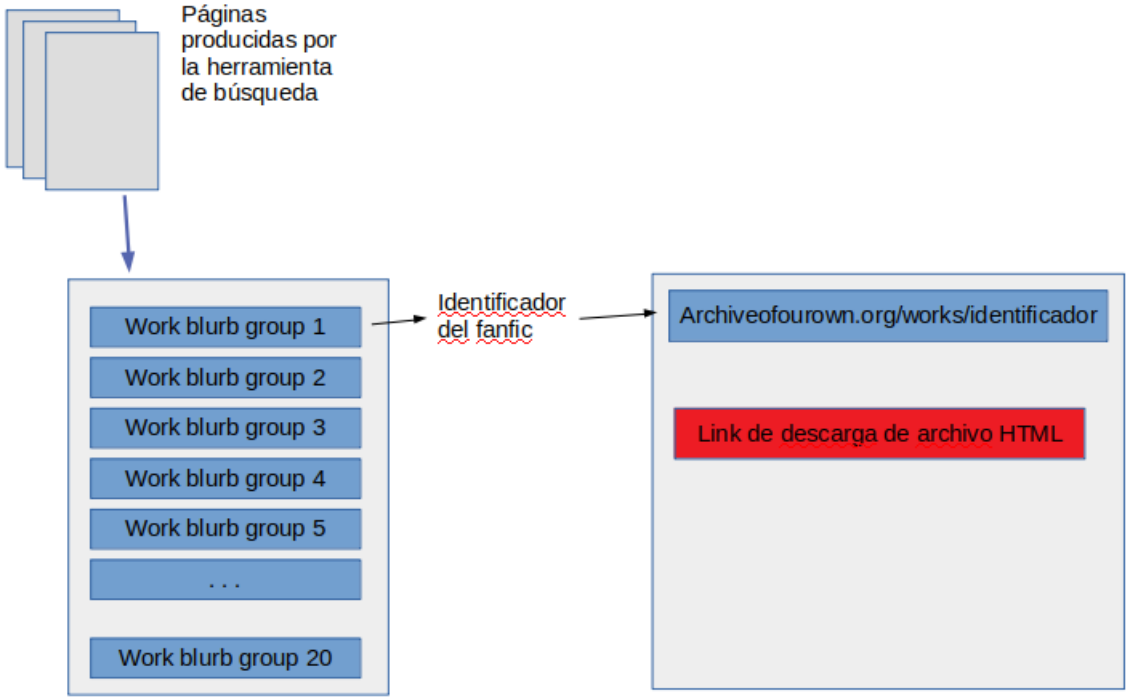


Figura 3: Concepto para el *scraper*. El objetivo es obtener los links de descarga navegando las páginas de búsqueda.

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:

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.

```
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
```

Display line numbers
Display right margin
Highlight current line
Text wrapping

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 textit'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).

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 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

```
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).

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 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.

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

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*'.

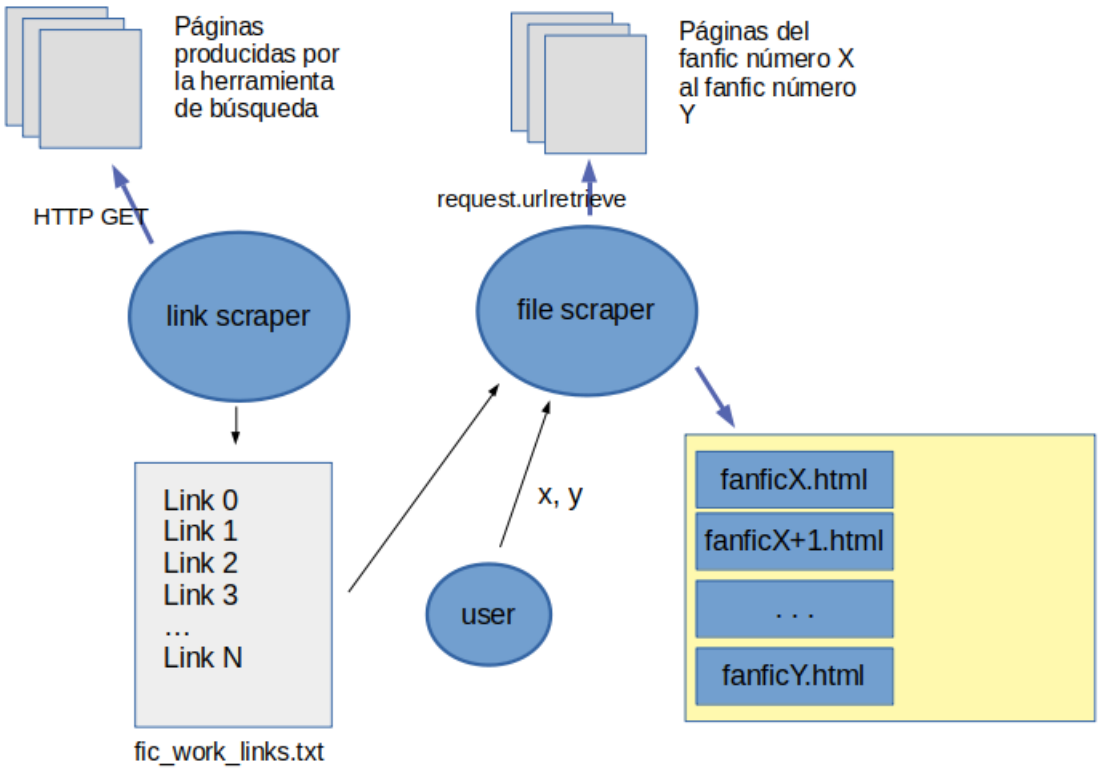


Figura 7: Proceso de descarga de fanfics de AO3 utilizando los programas 'ao3_link_scraper.py' y 'ao3_file_scraper.py'.

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 va almacenando 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 descargando 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 realizar un segundo filtrado de forma sencilla 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 conjunta de estos *scrapres* es una carpeta con 818,8 MB de archivos HTML.

4.2. LIMPIEZA DE DATOS

Los fanfics descargados en HTML vienen con metadatos relacionados con la historia, el autor y cuándo fueron publicados, entre otros. Además, la mayoría de fanfics tienen notas del autor, comentarios y sinopsis que, aunque forman parte del cuerpo del texto, no son relevantes para el análisis que voy a realizar. Me interesa poder recuperar fácilmente los metadatos útiles de cada fanfic, además de convertirlo a texto plano eliminando todos los metadatos y los comentarios del autor, por lo que decidí reunir todas estas funciones en dos clases, FanficCleaner y

FanficHTMLHandler, encapsuladas en un programa fanfic_util.py

- FanficCleaner tiene métodos para extraer el cuerpo del texto de un archivo HTML, sin metadatos, comentarios ni notas del autor. También tiene la función de guardar el texto en un archivo txt en un *path* a elegir.
- FanficHTMLHandler tiene métodos que permiten extraer metadatos del archivo HTML de un fanfic, como por ejemplo los personajes principales, el número de capítulos y su clasificación.

Debido a que los archivos HTML están organizados en carpetas, utilizo listas con sus *path* para identificarlos y acceder a ellos. Los métodos de FanficCleaner y FanficHTML reciben tramos de estas listas para acceder y manejar los archivos deseados. Para ello, el programa utiliza las librerías BeautifulSoup y html2text.

5. EXTRACCIÓN DE DATOS A PARTIR DE TEXTO

Explicar los programas que hice para familiarizarme con NLTK

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.

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.

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.

En horizontal queda así:

(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

La tarea de extracción de relaciones es compleja, y la naturaleza de los datos del proyecto la complica aún más, puesto que la ficción literaria se nutre de la ambigüedad, la heterogeneidad y la experimentación. No es razonable esperar que un conjunto de obras artísticas represente un

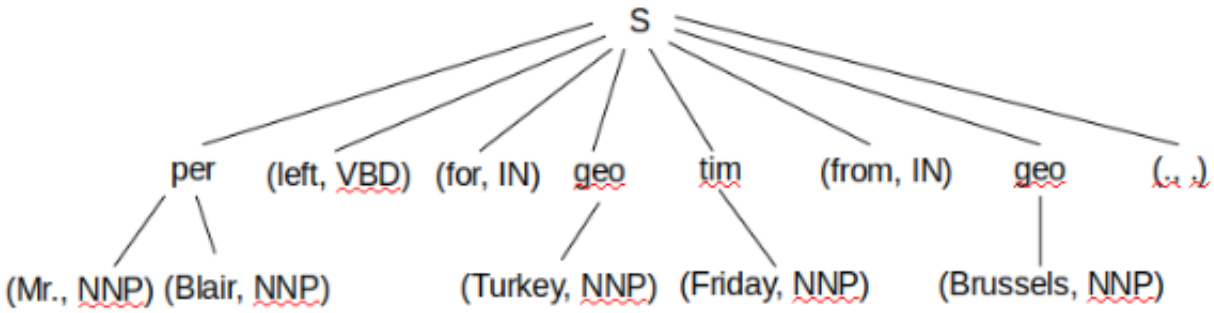


Figura 9: Frase etiquetada por NLTK. Las etiquetas de entidad se encuentran en los nodos, encontrándose todas las palabras pertenecientes a una secuencia de entidad en la profundidad 2 del árbol. Las palabras que no pertenecen a ninguna secuencia de entidad se encuentran en la profundidad 1. Cada hoja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

mismo tema de forma uniforme, mucho menos relaciones sentimentales (incluso si es la misma relación entre los mismos personajes).

Encontrar una estrategia válida para detectar relaciones sociales entre personajes requirió 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, es necesario buscar más opciones fuera de la librería. El Stanford Natural Language Processing Group publicó un extractor de relaciones accesible a través 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. 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. Decidiendo dejar de lado NLTK por el momento, decidí explorar soluciones usando clustering y modelado de temas.

5.2.1. Primeras estrategias: Clustering y LDA

Para este algoritmo decidí crear un algoritmo LDA, que es un algoritmode aprendizaje no supervisado que se utiliza típicamente en identificación de temas. Para entrenar este algoritmo, seleccioné un conjunto de fanfics que sirviesen de modelo para la relación que quería que aprendiera a detectar. Por ejemplo, para entrenar un modelo LDA que detecte situaciones sexuales, preparé un conjunto de fanfics de un único capítulo en los que sucedía sexo, excluyendo otros que tenían escenas sexuales pero eran sólo una pequeña parte de una historia más larga.

mejorar esto

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 10.

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

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 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 10.

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

Figura 10: 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%

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.

[Cul07] Aron Culotta. Canonicalization of database records using adaptive similarity measures. 2007.

[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.

[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.

[Wic09] Michael Wick. An entity based model for coreference resolution. 2009.

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