

Universidad de Alcalá

Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Análisis y comparacion de paquetes para el desarrollo de web
scraping

Autor: David Márquez Mínguez

Tutor: Juan José Cuadrado Gallego

2022

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Trabajo Fin de Grado

**Análisis y comparacion de paquetes para el desarrollo de web
scraping**

Autor: David Márquez Mínguez

Tutor: Juan José Cuadrado Gallego

Tribunal:

Presidente:

Vocal 1º:

Vocal 2º:

Fecha de depósito: de de

Resumen

Resumen..... correo de contacto: David Márquez Mínguez <david.marquez@edu.uah.es>.

Palabras clave: Trabajo fin de /grado, L^AT_EX, soporte de español e inglés, hasta cinco....

Abstract

Abstract..... contact email: David Márquez Mínguez <david.marquez@edu.uah.es>.

Keywords: Bachelor final project , L^AT_EX, English/Spanish support, maximum of five....

Resumen extendido

Con un máximo de cuatro o cinco páginas. Se supone que sólo está definido como obligatorio para los TFGs y PFCs de UAH.

Índice general

Resumen	v
Abstract	vii
Resumen extendido	ix
Índice general	xi
Índice de figuras	xv
Índice de tablas	xvii
Índice de listados de código fuente	xix
1 Introducción y objetivos	1
1.1 Contexto	1
1.2 Motivación	1
1.3 Objetivo y limitaciones	2
1.4 Estructura del documento	2
2 Web scraping, extracción de datos en la web	3
2.1 En que consiste realmente el web scraping	3
2.1.1 Extracción de los datos	3
2.1.2 Herramientas software disponibles	4
2.1.2.1 XPath	4
2.1.2.2 Selectores CSS	5
2.1.3 Tipologías	5
2.2 Posibilidades prácticas del web scraping	6
2.3 Retos del web scraping	6
2.4 Aspectos ético-legales del web scraping	7

3	Bibliotecas de programación orientadas al web scraping	9
3.1	Búsqueda de bibliotecas destinadas al web scraping	9
3.2	Bibliotecas de Python encontradas durante el proceso de búsqueda	9
3.2.1	inscriptis	10
3.2.1.1	Estructura de la solución	10
3.2.1.2	Reglas de anotación	11
3.2.1.3	Postprocesamiento	12
3.2.2	Beautiful Soup	13
3.2.2.1	Multiplicidad de analizadores	13
3.2.2.2	Sopa de objetos	14
3.2.3	jusText	14
3.2.3.1	Preprocesamiento	14
3.2.3.2	Segmentación	14
3.2.3.3	Clasificación de bloques	15
3.2.3.4	Reclasificación de bloques	16
3.2.3.5	Bloques de cabecera	17
3.2.4	news-please	17
3.2.4.1	Web crawling en news-please	18
3.2.4.2	Web scraping en news-please	18
3.2.5	Libextract	19
3.2.5.1	eatihit como algoritmo de partición	19
3.2.6	html_text	20
3.2.6.1	Normalización de espacio	20
3.2.7	html2text	21
3.2.8	Readability	21
3.2.8.1	Heurística de nodo candidato	21
3.2.9	Trafilatura	22
3.2.9.1	Delimitación de contenido	22
3.2.9.2	Algoritmos de respaldo	23
3.2.9.3	Extracción base	23
3.2.10	Dragnet	24
3.2.10.1	Extracción enfocada al aprendizaje automatico	24
3.2.10.2	Características del modelo	24
3.2.11	Goose3	25
3.2.11.1	Cálculo del mejor nodo	25
3.2.12	Newspaper3k	26
3.2.12.1	Heurística según el idioma	26

3.2.13	BoilerPy3/BoilerpipeR	27
3.2.13.1	Multiplicidad de extractores	27
3.3	Paquetes de R encontrados durante el proceso de búsqueda	27
3.3.1	rvest	28
3.3.1.1	Elementos a nivel de bloque vs elementos en línea	28
3.3.2	Rcrawler	28
3.3.2.1	Arquitectura y proceso de extracción en Rcrawler	29
3.3.3	htm2txt	30
3.3.4	scrapeR	30
3.3.5	RSelenium	30
3.4	Paquetes descartados para el proceso de evaluación	30
4	Selección de variables de análisis y proceso de estudio	33
5	Presupuesto	35
	Bibliografía	37
	Apéndice A Funcionamiento básico de un web scraper	41
A.1	Fase de búsqueda	41
A.2	Fase de extracción	42
A.3	Fase de transformación	43
	Apéndice B Analizadores empleados en los paquetes de web scraping	45
B.1	lxml	45
B.1.1	lxml.html	46
B.1.2	soupparser	46
B.2	html5lib	47
B.3	html.parser	47
B.4	XML	47
B.5	xml2	48

Índice de figuras

2.1	Fases del web scraping	4
2.2	Fases del web crawling	5
3.1	inscriptis - Postprocesamiento html	12
3.2	Arbol de objetos	14
3.3	jusText - Reclasificación de bloques	16
3.4	news-please - Proceso de scraping y crawling	17
3.5	rvest - Elementos a nivel de bloque	28
3.6	Rcrawler - Arquitectura y componentes principales de Rcrawler	29
B.1	Estructura basica de un analizador	45

Índice de tablas

3.1	Emparejamientos biblioteca-analizador	10
3.2	inscriptis - Perfil de anotaciones	11
3.3	Beautiful Soup - Analizadores disponibles	13
3.4	jusText - Clasificación de bloques long & medium size	16
3.5	Emparejamientos paquete-analizador	27

Índice de listados de código fuente

3.1	inscriptis - Uso de reglas de anotación	12
3.2	jusText - Algoritmo de clasificación	15
3.3	news-please - Extracción de contenido relevante	18
3.4	Libextract - Funcionamiento de eatiht	19
3.5	Readability - Selección del nodo candidato	21
3.6	Trafilatura - Readability como algoritmo de respaldo	23
3.7	Trafilatura - jusText como algoritmo de respaldo	23
3.8	Goose3 - Cálculo del mejor nodo 1	25
3.9	Goose3 - Cálculo del mejor nodo 2	25
3.10	Newspaper3k - Heurística en el árabe	26
A.1	Solicitud del documento <i>robots.txt</i>	41
A.2	Acceso y descarga del archivo HTML	42
A.3	Extracción de datos de interés del documento	43
A.4	Transformación de datos en un data frame	43

Capítulo 1

Introducción y objetivos

1.1 Contexto

La *World Wide Web* o lo que comúnmente se conoce como la web, es la estructura de datos más grande en la actualidad, y continúa creciendo de forma exponencial. Este gran crecimiento se debe a que el proceso de publicación de dicha información se ha ido facilitando con el tiempo.

Tradicionalmente el proceso de inserción y extracción de la información se realizaba a través del *copy-paste*. Aunque este método en ocasiones pueda ser la única opción, es una técnica muy ineficiente y poco productiva, pues provoca que el conjunto final de datos no esté bien estructurado. El *web scraping* o minado web trata precisamente de eso, de automatizar la extracción y almacenamiento de información extraída de un sitio web [1].

La forma en la que se extraen datos de internet puede ser muy diversa, aunque comúnmente se emplea el protocolo HTTP, existen otras formas de extraer datos de una web de forma automática [2]. Este proyecto, se centra en la metodología existente de obtención de información, de como se tratan los datos y la forma en la que se almacenan. Durante los siguientes apartados se realizará una especificación mas concreta del objetivo del proyecto, así como de la estructura y limitaciones del mismo.

1.2 Motivación

El proceso de extracción y recopilación de datos no estructurados en la web es un área interesante en muchos contextos, ya sea para uso científico o personal. En ciencia por ejemplo, los conjuntos de datos se comparten y utilizan por múltiples investigadores, y a menudo también son compartidos públicamente. Dichos conjuntos de datos se proporcionan a través de una API ¹ estructurada, pero puede suceder que solo sea posible acceder a ellos a través de formularios de búsqueda y documentos HTML. En el uso personal también ha crecido a medida que han comenzado a surgir servicios que proporcionan a los usuarios herramientas para combinar información de diferentes sitios web en su propias colección de páginas.

Además de ser un ámbito interesante, el minado web también es un area muy requerida, algunos de las campos de mayor demanda tienen relación con la venta minorista, mercado de valores, análisis de las redes sociales, investigaciones biomédicas, psicología...

¹Conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizada por otro software como abstracción.

1.3 Objetivo y limitaciones

Existen muchos tipos de técnicas y herramientas para realizar *web scraping*, desde programas con interfaz gráfica, hasta bibliotecas software de desarrollo. El objetivo de esta tesis es realizar un análisis cuantitativo de las diferentes técnicas y paquetes software para el desarrollo de *web scraping*.

¿Cuál es la solución más rentable para el minado web? ¿Cuál de las soluciones tiene un mejor rendimiento? Para responder a esta pregunta, se realizará un estudio comparando las diferentes características de los paquetes software, con el objetivo finalmente de poder determinar cuál es el más óptimo en términos de memoria, rendimiento...

En cuanto a las restricciones para el funcionamiento de un *scraper*, estas pueden ser varias, ya sean legales o por la incapacidad de acceder a una gran parte del contenido no indexado en internet. Aunque el uso de los *scrapers* está generalmente permitido, en algunos países como en Estados Unidos, las cortes en múltiples ocasiones han reconocido que ciertos usos no deberían estar autorizados [1]. El desarrollo de este proyecto no se verá perjudicado por este tipo de cuestiones, pues solo se limitará al estudio y análisis de los mismos.

1.4 Estructura del documento

Para poder facilitar la composición de la memoria se detalla a continuación la estructura de la misma:

1. Bloque I: Introducción.

- **Capítulo 1: Introducción.**

En la introducción se especifica tanto el contexto como la motivación a realizar el proyecto, así como las limitaciones esperadas durante la realización del mismo.

2. Bloque II: Marco teórico.

- **Capítulo 2: Web scraping, extracción de datos en la web.**

Durante este capítulo se explica en que consiste el *web scraping*, sus posibilidades prácticas y aspectos más generales.

- **Capítulo 3: Introducción a los paquetes seleccionados y proceso de búsqueda.**

Se realiza la selección de paquetes y se dictamina cuál ha sido la razón por la que los paquetes han sido seleccionados. Además, se especifican las características principales de cada uno, así como una visión general de sus funcionalidades.

3. Bloque III: Marco práctico.

- **Capítulo 4: Selección de variables de análisis y proceso de estudio.**

Durante este capítulo se especifica el proceso de análisis a realizar, cuáles son los test a los que los paquetes serán sometidos y que variables se tomarán a estudio para los mismos.

- **Capítulo 5: Análisis y comparativa de paquetes.**

Una vez introducidos todos los paquetes y el estudio al que van a ser sometidos, se realizará la comparativa de los mismos. Inicialmente, los paquetes serán analizados uno por uno y finalmente se hará una comparativa con los datos obtenidos.

4. Bloque IV: Conclusiones y futuras líneas de trabajo.

Capítulo 2

Web scraping, extracción de datos en la web

2.1 En que consiste realmente el web scraping

En la actualidad el *web scraping* se puede definir como una “*solución tecnológica para extraer información de sitios web, de forma rápida, eficiente y automática, ofreciendo datos en un formato más estructurado y más fácil de usar [3]*”. Sin embargo, esta definición no ha sido siempre así, los métodos de minado web han evolucionado desde procedimientos más pequeños con ayuda humana, hasta sistemas automatizados capaces de convertir sitios web completos en conjuntos de datos bien organizados.

Existen diferentes herramientas de minado, no solo capaces de analizar los lenguajes de marcado o archivos JSON, también capaces de realizar un procesamiento del lenguaje natural para simular cómo los usuarios navegan por el contenido web.

2.1.1 Extracción de los datos

En realidad el minado web es una práctica muy sencilla, se extraen datos de la web y se almacenan en una estructura de datos para su posterior análisis o recuperación. En este proceso, un agente de software, también conocido como robot, imita la navegación humana convencional y paso a paso accede a tantos sitios web como sea necesario [4]. Las fases por las que pasa el agente software en cuestión se determinan a continuación:

1. **Fase de búsqueda:** Esta primera fase consiste en el acceso al sitio web del que se quiere obtener la información. El acceso se proporciona realizando una solicitud de comunicación HTTP. Una vez establecida la comunicación, la información se gestiona a partir de los métodos GET y POST usuales.
2. **Fase de extracción:** Una vez que el acceso ha sido permitido, es posible obtener la información de interés. Se suelen emplear para este propósito expresiones regulares o librerías de análisis HTML. Los diferentes softwares empleados para este propósito se especifican en la sección [2.1.2](#).
3. **Fase de transformación:** El objetivo de esta fase es transformar toda la información extraída en un conjunto estructurado, para una posible extracción o análisis posterior. Los tipos de estructuras más comunes en este caso son soluciones basadas en cadenas de texto o archivos CSV y XML.

Una vez que el contenido ha sido extraído y transformado en un conjunto ordenado, es posible realizar un análisis de la información de una forma más eficaz y sencilla que aplicando el método tradicional. El proceso descrito se resume en la ilustración 2.1.

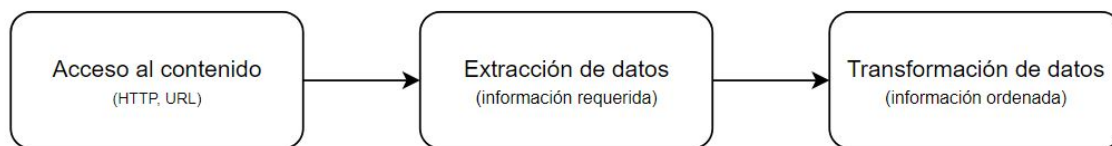


Figura 2.1: Fases del web scraping

Por otro lado, a lo largo del apéndice A, se ilustra el funcionamiento del agente software en cada una de las fases descritas anteriormente, así como de su comportamiento con el servidor web al que se desea acceder.

2.1.2 Herramientas software disponibles

El software disponible que emplean los *web scrapers* puede dividirse en varios enfoques, ya sean bibliotecas de programación de propósito general, *frameworks*, o entornos de escritorio.

Por lo general tanto los *frameworks* como los entornos de escritorio presentan una solución más sencilla e integradora con respecto a bibliotecas de programación. Esto es debido a que ambos, no se ven afectados a los posibles cambios HTML de los recursos a los que accede. Además, estas necesitan la integración de otras múltiples bibliotecas adicionales para el acceso, análisis y extracción del contenido.

Este trabajo se desarrolla sobre bibliotecas de programación, las cuales se implementan como un programa software convencional utilizando estructuras de control y de datos del propio lenguaje. Por lo general, bibliotecas como *curl* [5] conceden acceso al sitio web deseado haciendo uso del protocolo HTTP, mientras que los contenidos extraídos se analizan a través de funciones como la coincidencia de expresiones regulares y la tokenización.

Comprender como las bibliotecas obtienen los datos de los sitios web, pasa por conocer las diferentes formas en las que los documentos HTML se organizan. Existen dos técnicas, dependiendo si se realiza un renderizado previo o no [6]. La primera técnica consiste simplemente en parsear, es decir, realizar un análisis léxico-sintáctico sobre estructuras XML o HTML. Se suelen emplear expresiones XPath o selectores CSS para su realización. Por otro, lado si es necesario que parte de la lógica del sitio web pase al lado del cliente, este deberá pasar por un proceso de renderizado previo.

Con el paso del tiempo, cada vez se extiende más el uso de bibliotecas de desarrollo como JQuery, encargadas de pasar parte de la lógica del lado del servidor al lado del cliente con el objetivo de favorecer la interactividad. Estas páginas no podrán ser analizadas si no se renderizan antes.

2.1.2.1 XPath

XPath es un lenguaje que permite construir expresiones que recorren y procesan un documento XML [7]. Puede utilizarse para navegar por documentos HTML, ya que este es un lenguaje similar en cuanto a estructura a XML. Es comúnmente utilizado en el *web scraping* para extraer datos de documentos HTML, además utiliza la misma notación empleada en las URL para navegar por la estructura del sitio web en cuestión.

2.1.2.2 Selectores CSS

El segundo método de extracción de datos en documento HTML se realiza a través de lo que se conoce como selectores CSS [7]. CSS es el lenguaje utilizado para dar estilo a los documentos HTML, por otro lado, los selectores son patrones que se utilizan para hacer coincidir y extraer elementos HTML basados en sus propiedades CSS.

Hay múltiples sintaxis de selector diferentes, estas se corresponden con la forma en la que el documento CSS está estructurado. En el fragmento de código A.3 se hace uso de los selectores `'text-primary'` y `'li-list-item-header a'` para acceder al contenido web deseado.

2.1.3 Tipologías

Dependiendo de como se acceda y extraiga la información, existen dos técnicas de *web scraping*. Se mencionan los siguientes supuestos a continuación:

- Si la información que se almacena no procede de sitios web concretos, sino que durante el análisis de páginas web se encuentran enlaces que retroalimentan el análisis de otras nuevas, el método se conoce como *web crawling* [1].
- Por el contrario, si la información se extrae de sitios web concretos, donde ya se conoce como extraer y generar un valor por la misma, la técnica se conoce como *web scraping* genérico. Mientras que en el *web crawling* el resultado de ejecución es la obtención de nuevas páginas, en el *web scraping* el resultado es la propia información.

Es decir, la principal diferencia entre ambas, es que mientras los *web scrapers* extraen información de páginas webs concretas, los *web crawlers* almacenan y acceden a las páginas a través de los enlaces contenidos en las mismas. En la figura 2.1 se mostraba la arquitectura en fases de un *web scraper*, veamos a continuación como es la arquitectura de un *web crawler*.

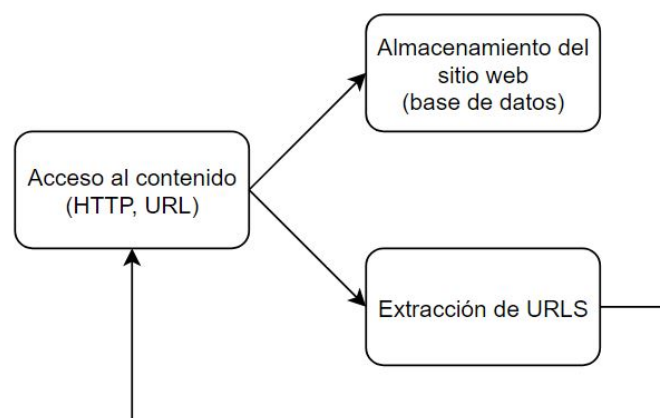


Figura 2.2: Fases del web crawling

Ya sea empleando cualquiera de estas dos tipologías, existen páginas que no pueden ser analizadas o rastreadas. Esto es debido a que algunos sitios web solo están disponibles con una autorización previa, o necesitan información especial para su acceso.

2.2 Posibilidades prácticas del web scraping

Son muchas las aplicaciones prácticas de la minería web, la mayoría de estas entran en el ámbito de la ciencia de los datos. En la siguiente lista se exponen algunos casos de su uso en la vida real [8]:

- Bancos y otras instituciones financieras utilizan minería web para analizar a la competencia. Inversores también utilizan *web scraping*, para hacer un seguimiento de artículos de prensa relacionados con los activos de su cartera.
- En las redes sociales se emplea minería de datos para conocer la opinión de la gente a cerca de un determinado tema.
- Existen aplicaciones capaces de analizar diferentes sitios web y encontrar los mismos productos a precio reducido. Incluso capaces de detectar ofertas de artículos a tiempo récord.
- etcétera

El minado web contiene infinidad de aplicaciones, muy diversas e interesante capaces de automatizar el trabajo y conseguir la información de forma ordenada. No obstante, muchos sitios web ofrecen alternativas como el uso de APIs o ficheros estructuras para el acceso a dichos datos.

En general el *web scraping* es una técnica que consume bastantes recursos, por lo que el desarrollador debe limitar su uso si existen otras alternativas, como las APIs, que proporcionan los mismos resultados.

2.3 Retos del web scraping

La forma en la que se crean los sistemas de extracción web se ha discutido desde diferentes perspectivas a lo largo del tiempo. Se emplean métodos científicos como el aprendizaje automático, la lógica o el procesamiento del lenguaje natural para lograr su implementación.

Uno de los principales retos a afrontar tiene relación con las fuentes cambiantes de información. A menudo, una herramienta de extracción tiene que obtener datos de forma rutinaria de una determinada página o sitio web que puede evolucionar con el tiempo. Estos cambios se producen sin previo aviso, son imprevisibles, por lo que es bastante probable que los raspadores web se vean sometidos a cambios. Por ello, surge la necesidad crear herramientas flexibles capaces de detectar y afrontar modificaciones estructurales de las fuentes web relacionadas.

Otros problemas recaen sobre la información extraída. En primer lugar, uno de los aspectos que se deben tener en cuenta al obtener información trata sobre la fiabilidad de la misma. Aunque la información exista y se pueda ser analizada, esta puede que no sea correcta. La gramática y la ortografía pueden ser un problema en la fase de análisis, ya que información puede perderse o ser falsamente recogida. Por otro lado, tanto aplicaciones que tratan con datos personales, como software de minado deben ofrecer garantías de privacidad. Por lo tanto, los posibles intentos de violar la privacidad del usuario deben ser identificados y contrarrestados a tiempo y de forma adecuada.

Puesto que multitud de técnicas de minería web requieren ayuda humana, un primer reto consiste en proporcionar un alto grado de automatización, reduciendo así al máximo el esfuerzo humano. Sin embargo, la ayuda humana puede desempeñar un papel importante a la hora de elevar el nivel de precisión alcanzado por un sistema de extracción de datos web, por ello la clave está en encontrar un equilibrio entre automatización e intervención humana.

Por último, a pesar de que las herramientas de *web scraping* han evolucionado con el tiempo, los aspectos legales están algo inexplorados pues dependen de los términos y condiciones de cada sitio web en cuestión.

2.4 Aspectos ético-legales del web scraping

Para comprender los aspectos legales del *web scraping*, debemos recordar el robot o agente software definido en el apartado 2.1.1. Este agente software, previamente examinado por el servidor, es el que se encarga de acceder y realizar un recorrido por el contenido web.

Durante el acceso al contenido, se espera que este agente se ajuste a los términos de uso del sitio en cuestión, así como el cumplimiento del archivo '*robots.txt*'¹, con el objetivo de evitar accesos no deseados y sobrecargas en el servidor.

Puesto que el documento '*robots.txt*' no es de obligado cumplimiento, a lo largo de los años la reputación del *web scraping* ha decrecido de forma significativa. Muchos agentes software no siguen las indicaciones determinadas, por lo que definir la cantidad de accesos y archivos a los que se accede dependerá de la ética de cada desarrollador.

Con el objetivo de tener una cierta garantía de que nuestro agente software cumple con los aspectos ético-legales, se deben tener en consideración las siguientes cuestiones [10]:

- Leer los términos de uso de la página web en la que se vaya a realizar el minado.
- Inspeccionar y cumplir con el documento *robots.txt*, para ser capaces de identificar los accesos del servidor.
- Realizar peticiones al servidor de forma controlada. Puede que el índice de solicitudes al servidor no esté especificado en el documento, si esto sucede debemos determinar un número de solicitudes razonable, por ejemplo, una solicitud por segundo.

¹ Archivo alojado en el servidor web, que gestiona el tráfico del mismo e indica los documentos a los que no se debe acceder de forma automática [9].

Capítulo 3

Bibliotecas de programación orientadas al web scraping

3.1 Búsqueda de bibliotecas destinadas al web scraping

Como se especificó en la sección 1.3 este trabajo se limitará a realizar una comparativa de los programas software de minado web más frecuentes. Esta comparativa se efectúa con el fin de conocer cuál o cuáles de estos programas o paquetes software son los más rentables para este propósito.

¿Como saber que paquetes software destinados al minado web son los más comunes? Durante todo este capítulo se procederá a la búsqueda, selección e introducción de programas software empleados para el *web scraping*. Cabe destacar que Python y R serán los lenguajes de programación con los que se trabajará tanto para el desarrollo de la herramienta de comparación, como para el proceso de extracción de datos.

El primer paso consiste en buscar todos los elementos posibles que conforman la población de paquetes software del mercado, ya sean de Python o R. La búsqueda de estos paquetes se ha realizado a través de las distintas fuentes de información mostradas a continuación:

1. GitHub [11]. Gran parte de los desarrolladores de estos programas, publican su trabajo en estos repositorios de código abierto.
2. CRAN [12]. The Comprehensive R Archive Network es una red de servidores ftp y web que almacena versiones idénticas de código y documentación para R.
3. PyPi [13]. Python Package Index es un repositorio de software para Python, útil para la búsqueda de paquetes de un determinado propósito.

3.2 Bibliotecas de Python encontradas durante el proceso de búsqueda

A continuación, se hará una breve sinopsis de las bibliotecas de Python encontradas. Esta sinopsis tiene como objetivo conocer el funcionamiento, funciones principales y proceso de extracción de las mismas.

Es posible que algunos de los paquetes hayan sido desarrollados tanto en R como en Python. En ese caso, por un lado, la introducción se realizará de forma conjunta, sin embargo, será interesante ver el código del mismo en ambos casos y como se comportan estos ante los distintos test de evaluación.

Antes de comenzar con la sinopsis de bibliotecas, es conveniente revisar el apéndice B, donde se realiza una breve introducción de aquellos analizadores empleados en las mismas. Los 'emparejamientos' entre biblioteca-analizador se recogen en la tabla 3.1 a modo de resumen.

inscriptis	dragnet	boilerpy	libextract	news-please	justext	goose3
lxml	lxml	html parser	lxml	lxml	lxml	lxml & html parser

html2text	readability	trafilatura	beautiful soup	newspaper3k	html_text
html parser	lxml	lxml	lxml & html5lib & html	lxml	lxml

Tabla 3.1: Emparejamientos biblioteca-analizador

Ya sea porque determinadas bibliotecas permiten al desarrollador seleccionar entre varios tipos de analizadores, o bien porque múltiples de ellos son necesarios para el correcto funcionamiento de la extracción, muchas de estas bibliotecas hacen uso de más de un analizador en su código.

3.2.1 inscriptis

Además de ser una biblioteca de conversión de HTML a texto basada en Python, **inscriptis** [14] también tiene soporte como línea de comandos o como servicio web para tablas anidadas. A pesar de sus múltiples funcionalidades esta sección se centra en **inscriptis** como biblioteca de programación.

3.2.1.1 Estructura de la solución

A diferencia de otros algoritmos de extracción, **inscriptis** no solo tiene en cuenta la calidad del texto extraído, la estructura del mismo también es muy importante. Esto provoca que el resultado obtenido se acerque más a un posible resultado aplicando el método tradicional a través de cualquier navegador web.

Se muestra a continuación una pequeña comparación entre la extracción de texto de **Beautiful Soup** 3.2.2, con la extracción de texto de **inscriptis**, donde se tiene un fragmento HTML como el siguiente como objeto de prueba.

```
<li>first</li>
<li>second</li>
```

Se emplea en primer lugar el método *get_text()* propio de **Beautiful Soup** sobre el fragmento HTML anterior.

```
# firstsecond
```

Se puede observar que el formato obtenido no es el adecuado, puesto que no se ha respetado la estructura del documento base. Sin embargo, si se aplica **inscriptis** sobre el mismo fragmento HTML, la salida obtenida sería la siguiente:

```
# first
# second
```

El algoritmo no solo admite construcciones tan simples como la anterior, también es posible analizar construcciones mucho más complejas, como las tablas anidadas, y subconjuntos de atributos HTML o CSS donde es esencial una conversión precisa de HTML a texto.

3.2.1.2 Reglas de anotación

La técnica que se emplea en este caso se conoce como reglas de anotación, es decir, mapeos que permiten realizar anotaciones sobre el texto extraído. Estas anotaciones se basan en la información estructural y semántica codificada en las etiquetas y atributos HTML utilizados para controlar la estructura y diseño del documento original. Con el fin de asignar etiquetas y/o atributos HTML a las anotaciones, se utiliza lo que se conoce como perfil de anotaciones, algo parecido a un diccionario.

h1	['heading', 'h1']
h2	['heading', 'h2']
b	['emphasis']
div#class=toc	['table-of-contents']
#class=FactBox	['fact-box']
#cite	['citation']

Tabla 3.2: inscriptis - Perfil de anotaciones

Si observamos el diccionario mostrado en la tabla 3.2, las etiquetas de tipo cabecera producen anotaciones de tipo *hn*, una etiqueta `<div>` con una clase que contiene el valor *toc* da como resultado la anotación *table-of-contents*, y todas las etiquetas con un atributo *cite* se anotan como *citation*.

A modo de ejemplo, y con el fin de mostrar el correcto etiquetado del algoritmo, imaginemos que se dispone el fragmento de un documento HTML como el mostrado a continuación y unas reglas de anotación como las mostradas en la tabla 3.2.

```
<h1>Chur</h1>
<b>Chur</b> is the capital and largest town of the Swiss
canton of the Grisons and lies in the Grisonian Rhine Valley.
```

A partir de este ejemplo, y basándonos en el diccionario anterior, la salida esperada debería ser una etiqueta de cabecera y otra de énfasis, veamos el resultado que proporciona el proceso de asignación.

```
{
  "text": "Chur\n\nChur is the capital and largest town of the Swiss
          canton of the Grisons and lies in the Grisonian Rhine Valley.",
  "label": [[0, 4, "heading"], [0, 4, "h1"], [6, 10, "emphasis"]]
}
```

Como era de esperar la obtención del texto es precisa, pero no solo del texto sino de su estructura. Además, la asignación de etiquetas también se ha realizado de forma correcta. Se muestra en el fragmento de código 3.1 como se pueden emplear las reglas de anotación dentro de un programa.

Listado 3.1: inscriptis - Uso de reglas de anotación

```
import urllib.request
from inscriptis import get_annotated_text, ParserConfig

html = urllib.request.urlopen(url).read().decode('utf-8')

rules = {'h1': ['heading', 'h1'],
        'h2': ['heading', 'h2'],
        'b': ['emphasis'],
        'table': ['table']}

output = get_annotated_text(html, ParserConfig(annotation_rules=rules))
```

3.2.1.3 Postprocesamiento

Además, **inscriptis** da la posibilidad al usuario de realizar una fase de postprocesamiento donde las anotaciones se realizan en un formato determinado. Un primer tipo de postprocesamiento es el que se conoce como *surface*, donde se retorna una lista de mapeos entre la superficie de anotación y su etiqueta.

```
[['heading', 'Chur'],
 ['emphasis': 'Chur']]
```

En segundo lugar, el postprocesamiento tipo xml retorna una etiqueta de versión adicional, propia de un documento XML convencional.

```
<?xml version="1.0" encoding="UTF-8" ?>
<heading>Chur</heading>

<emphasis>Chur</emphasis> is the capital and largest town of the
Swiss canton of the Grisons and lies in the Grisonian Rhine Valley.
```

Por último, el postprocesamiento en tipo html crea un documento HTML que contiene el texto convertido y resalta todas las anotaciones. Se muestra en la figura 3.1 un ejemplo de la salida obtenida.

City Councillor	Party	Head of Department (Leitung, since)	emphasized	elected since
(Stadtrat/ Stadträtin)				
Urs Marti	FDP	Departement 1 (2013)		2012
Tom Leibundgut	FLV	Departement 3 (2013)		2012
Patrik Degiacomi	SP	Departement 2 (2017)		2016

Figura 3.1: inscriptis - Postprocesamiento html

3.2.2 Beautiful Soup

Una de las bibliotecas de Python más comunes en el ámbito del *web scraping* es **Beautiful Soup** [15], la cual está diseñada para extraer datos de documentos XML y HTML. Como se muestra en la tabla 3.1 y a diferencia del resto de bibliotecas, **Beautiful Soup** permite determinar el tipo de analizador que se empleará en la extracción, lo que flexibiliza el proceso de navegación, búsqueda y modificación de documentos.

3.2.2.1 Multiplicidad de analizadores

Beautiful Soup tiene a *html.parse* como analizador estándar de documentos HTML, pero admite varios analizadores de terceros. En la tabla 3.3 se muestran los distintos analizadores disponibles, así como un pequeño resumen de las ventajas y desventajas de estos.

Tipo de analizador	Forma de uso	Ventajas	Desventajas
html	bs(markup, "html.parser")	Notablemente rápido	Mas lento que lxml
lxml html	bs(markup, "lxml")	Muy rapido	Dependencia de C
lxml xml	bs(markup, "xml")	Muy rápido y soporta xml	Dependencia de C
html5lib	bs(markup, "html5lib")	Analiza igual que un buscador	Muy lento

Tabla 3.3: Beautiful Soup - Analizadores disponibles

El empleo de distintos analizadores supondrá una importancia menor si se aplica sobre documentos bien formados, pues la solución aportada presentará la misma estructura que el propio documento original. En caso contrario, el uso de diferentes analizadores creará diferentes soluciones para un mismo documento.

Se emplea el analizador *lxml* sobre un documento HTML sencillo pero con erratas. Vemos como la solución aportada propone la inclusión de nuevas etiquetas `<html>` y `<body>`, sin embargo, ¿qué ha ocurrido con la etiqueta `</p>`?

```
>>> BeautifulSoup("<a></p>", "lxml")
# <html><body><a></a></body></html>
```

En lugar de ignorar la etiqueta `</p>` como lo hace *lxml*, el analizador *html5lib* la empareja con una etiqueta `<p>` de apertura. También añade una etiqueta `<head>` que el analizador *lxml* había obviado.

```
>>> BeautifulSoup("<a></p>", "html5lib")
# <html><head></head><body><a><p></p></a></body></html>
```

Al igual que *lxml*, *html.parse* ignora la etiqueta de cierre `</p>`. Podemos observar que este analizador ni siquiera intenta crear un documento HTML bien formado añadiendo etiquetas `<html>` o `<body>`.

```
>>> BeautifulSoup("<a></p>", "html.parser")
# <a></a>
```

Como vemos diferentes analizadores crearan diferentes soluciones en caso de que el documento a analizar no este bien formado. Por ello, si se desea analizar múltiples documentos de los que no conocemos su origen o estructura, sería conveniente especificar el tipo de analizador con el fin del obtener la solución deseada.

3.2.2.2 Sopa de objetos

En cuanto al proceso de extracción que se emplea en este algoritmo es sencillo. En primer lugar, el documento ya sea HTML o XML se convierte al completo en caracteres unicode. Tras ello se crea un árbol de objetos donde cada uno de ellos representa una etiqueta o tag del propio documento. Finalmente, un analizador especificado por parámetro, recorre el árbol buscando las partes del mismo que se desean.

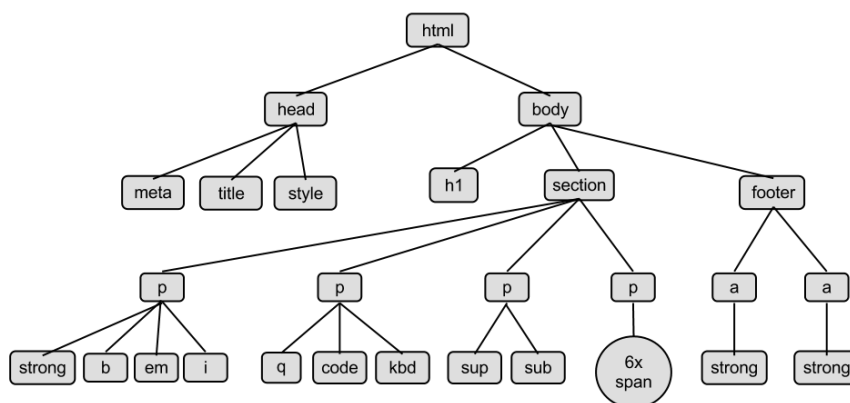


Figura 3.2: Árbol de objetos

Este algoritmo, no solo permite el recorrido automático del árbol en busca del texto del documento al completo, sino que permite además la posibilidad de recorrer el mismo de forma manual, por lo que es posible acceder a todos los objetos del árbol empleando métodos de navegación como *soup.head*, *soup.parent*, *soup.next_sibling*,

3.2.3 jusText

jusText [16] es una herramienta para eliminar el contenido repetitivo, como los enlaces de navegación, encabezados y pies de página de los documentos HTML. Este algoritmo, está diseñado para preservar el texto que contiene frases completas.

3.2.3.1 Preprocesamiento

Previamente a cualquier fase y con el fin de facilitar el trabajo heurístico, se realiza un preprocesamiento del documento HTML. Durante este proceso, se elimina el contenido de ciertas etiquetas como *<header>*, *<style>* y *<script>*. Además, el contenido de etiquetas como *<select>* se clasifica inmediatamente como contenido basura. Lo mismo ocurre con los bloques que contienen ciertos símbolos especiales como el de copyright ©.

3.2.3.2 Segmentación

Tras una previa fase de preprocesamiento, se procede a lo que se conoce como segmentación. La idea es formar bloques de texto dividiendo la página HTML por etiquetas. Una secuencia de dos o más etiquetas como *
*, *<div>*, ..., separaría los bloques.

Para la segmentación de bloques, la clave es que los bloques largos y algunos bloques cortos pueden clasificarse con una confianza muy alta. El resto de bloques cortos pueden clasificarse observando los bloques circundantes.

Aunque no sea habitual, puede ocurrir que el contenido de estos bloques no sea homogéneo, es decir, que dentro de un mismo bloque haya una mezcla de información importante con contenido basura, denominado *'boilerplate'*. Se resumen a continuación algunos aspectos en relación.

- Los bloques cortos que contienen un enlace son casi siempre del tipo *'boilerplate'*.
- Los bloques que contienen muchos enlaces son casi siempre del tipo *'boilerplate'*.
- Tanto los bloques buenos como los bloques de tipo *'boilerplate'* tienden a crear grupos, es decir, un bloque *'boilerplate'* suele estar rodeado de otros bloques de su mismo tipo y viceversa.
- Los bloques largos que contienen texto gramatical forman parte del contenido valioso, mientras que todos los demás bloques largos son casi siempre del tipo *'boilerplate'*.

Con respecto al ultimo punto, decidir si un texto es gramatical o no puede ser complicado, **jusText** emplea una simple heurística basada en el volumen de palabras con sentido gramatical. Mientras que un texto gramatical suele contener un cierto porcentaje de estas palabras, los contenidos de tipo *'boilerplate'* suelen carecer de ellas.

3.2.3.3 Clasificación de bloques

Tras la fase de preprocesamiento y segmentación se procede a la clasificación de bloques, donde a cada uno de estos bloques se le asigna una clase dependiendo de su naturaleza. En el fragmento de código 3.2 se muestra paso a paso como se determina el tipo de clase para cada bloque.

Listado 3.2: jusText - Algoritmo de clasificación

```
if link_density > MAX_LINK_DENSITY:
    return 'bad'

# short blocks
if length < LENGTH_LOW:
    if link_density > 0:
        return 'bad'
    else :
        return 'short'

# medium and long blocks
if stopwords_density > STOPWORDS_HIGH:
    if length > LENGTH_HIGH:
        return 'good'
    else :
        return 'near-good'
if stopwords_density > STOPWORDS_LOW:
    return 'near-good'
else :
    return 'bad'
```

Analizando el algoritmo, se observa que se definen dos tipos de variables, la densidad y la longitud. Mientras que la longitud es el número de caracteres por bloque, la densidad se define como la proporción de caracteres o palabras dentro de una etiqueta de tipo *<a>*, o una lista de parada.

Además de los valores de densidad y longitud, el algoritmo toma como parámetros dos enteros definidos como *LENGTH_LOW* y *LENGTH_HIGH*, y también tres números de coma flotante, *MAX_LINK_DENSITY*, *STOPWORDS_LOW* y *STOPWORDS_HIGH*. Los dos primeros establecen los umbrales para dividir los bloques por su longitud. Los dos últimos dividen los bloques por la densidad de palabras de parada en bajos, medianos y altos.

Si volvemos a observar el algoritmo 3.2, nos damos cuenta de que solo se ha realizado una clasificación real sobre los bloques de tamaño medio y largo. En la tabla 3.4 se muestra a modo de resumen como se actúa sobre este tipo de bloques.

Block size	Stopwords density	Class
medium size	low	bad
long	low	bad
medium size	medium	near-good
long	medium	near-good
medium size	high	near-good
long	high	good

Tabla 3.4: jusText - Clasificación de bloques long & medium size

3.2.3.4 Reclasificación de bloques

¿Qué ocurre entonces con los bloques cortos y los bloques casi buenos? La reclasificación en este caso se realiza en función de los bloques circundantes. Los bloques ya clasificados como buenos o malos sirven como piedras base en esta etapa y su clasificación se considera fiable, por lo que nunca se modifica. Esta reclasificación se puede ver resumida en la figura 3.3.



Figura 3.3: jusText - Reclasificación de bloques

La idea que subyace a la reclasificación es que los bloques *'boilerplate'* suelen estar rodeados de otros bloques *'boilerplate'* y viceversa. Los bloques casi buenos suelen contener datos útiles del corpus si se encuentran cerca de bloques buenos. Los bloques cortos suelen ser útiles sólo si están rodeados de bloques buenos por ambos lados.

3.2.3.5 Bloques de cabecera

En cuanto a los bloques de cabecera, aquellos encerrados en etiquetas del tipo `<h1>`, `<h2>`, ..., son tratados de una manera especial. El objetivo es conservar estos bloques en los textos determinados como buenos.

Para el tratamiento especial de bloques de cabecera se añaden dos etapas. La primera etapa, conocida como preprocesamiento, se ejecuta justamente después de la clasificación y justo antes de la reclasificación de bloques. La segunda etapa, conocida como postprocesamiento, se ejecuta después de la reclasificación.

1. Clasificación de bloques.
2. **Preprocesamiento de bloques de cabecera.**
3. Reclasificación de bloques.
4. **Postprocesamiento de bloques de cabecera.**

Durante esta fase de preprocesamiento, se buscan bloques de cabecera cortos que precedan a bloques buenos, y que al mismo tiempo no haya más caracteres entre el bloque de cabecera y el bloque bueno. El propósito de esto es preservar los bloques cortos entre el encabezado y el texto bueno que, de otro modo, podrían ser seleccionados como malos durante el proceso de reclasificación.

Por otro lado, en el postprocesamiento, se buscan de nuevo bloques de cabecera que precedan a bloques buenos, y que al mismo tiempo no haya más caracteres entre el bloque de cabecera y el bloque bueno. El propósito es que algunas cabeceras cortas y casi buenas puedan clasificarse como buenas si preceden a bloques buenos que, de otro modo, habrían sido clasificadas como malas tras de la reclasificación.

3.2.4 news-please

Otro *web scraper*, y al mismo tiempo *web crawler*, de código abierto es **news-please** [17]. Esta biblioteca está desarrollada para cumplir cinco requisitos: extracción de noticias de cualquier sitio web, extracción completa del sitio web, alta calidad de la información extraída, facilidad de uso y mantenibilidad.

A diferencia de otras bibliotecas ya mencionadas anteriormente, se emplean herramientas ya existentes las cuales se amplían con nuevas funcionalidades con el objetivo de cumplir con los requisitos señalados.

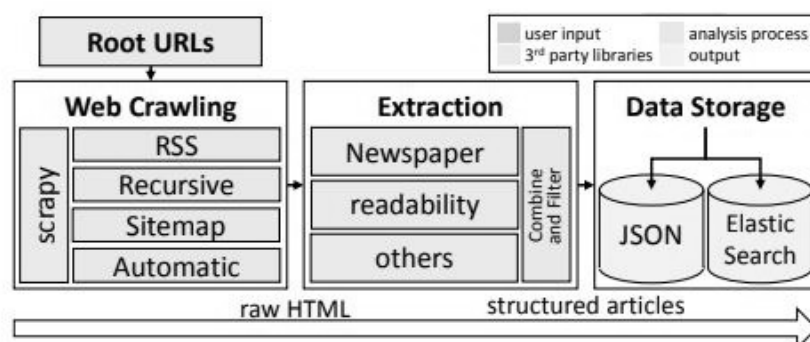


Figura 3.4: news-please - Proceso de scraping y crawling

3.2.4.1 Web crawling en news-please

En cuanto al proceso de *web crawling*, primero se descarga el documento y en segundo lugar, con el objetivo de encontrar todos los artículos publicados en dicho documento, se procede con el *crawling*. Como se muestra en la figura 3.4, para este proceso se dispone de cuatro técnicas diferentes.

La primera técnica se conoce como RSS, en la que se analizan canales RSS con el objetivo de encontrar artículos recientes. Un RSS [18] es un tipo de formato de datos utilizado para proporcionar a los usuarios contenidos actualizados con frecuencia.

En segundo lugar, se emplea un seguimiento recursivo de los enlaces internos en las páginas rastreadas. El uso del *framework scrapy* es fundamental independientemente de la técnica.

La tercera técnica consiste en analizar los *sitemaps* en busca de enlaces a todos los artículos. Un *sitemap* [18] es un archivo que enumera las URL's visibles de un determinado sitio, cuyo objetivo principal es revelar dónde pueden buscar contenido las máquinas.

Finalmente se prueba con el *crawling* a través de los *sitemaps* y en caso de que se produzca cualquier error se vuelve a la técnica de *crawling* recursiva.

Estos cuatro enfoques también pueden combinarse, por ejemplo, iniciando dos instancias en paralelo, una en modo automático para obtener todos los artículos publicados hasta el momento y otra instancia en modo RSS para recuperar los artículos recientes.

3.2.4.2 Web scraping en news-please

Para el proceso de *web scraping*, se emplean herramientas ya existentes en el mercado con el objetivo de obtener la información deseada, título, contenido principal, autor, fecha... Herramientas como **Newspaper3k** 3.2.12 o **Readability** 3.2.8 son utilizadas para este proceso de extracción.

Listado 3.3: news-please - Extracción de contenido relevante

```
def from_html(html, url=None, download_date=None, fetch_images=True):
    '''Extracts relevant information from an HTML page given as a string.'''
    extractor = article_extractor.Extractor(
        (['newspaper_extractor']
         if fetch_images
         else ["newspaper_extractor_no_images", "NewspaperExtractorNoImages"])
    ) + ['readability_extractor', 'date_extractor', 'lang_detect_extractor'])

    item = NewscrawlerItem()
    item['spider_response'] = DotMap()
    item['spider_response'].body = html
    ...
    item['modified_date'] = None
    item = extractor.extract(item)

    tmp_article = ExtractedInformationStorage.extract_relevant_info(item)
    final_article = ExtractedInformationStorage.convert_to_class(tmp_article)
    return final_article
```

Una vez que el contenido relevante ha sido extraído, se almacena de forma estructurada en determinados archivos de forma que este pueda ser consultado posteriormente.

3.2.5 Libextract

Libextract [19] es una biblioteca de extracción de datos con capacidad estadística que funciona con documentos HTML y XML escrita en Python. En cuanto al algoritmo de extracción empleado, **eatih**, funciona en base a una simple suposición en la cual los datos aparecen como colecciones de elementos repetitivos.

3.2.5.1 eatiht como algoritmo de partición

eatih [20] es una posible solución en una línea de muchas soluciones imperfectas a uno o más problemas en algún subcampo de la extracción de datos. En pocas palabras, este algoritmo intenta extraer el texto central de un sitio web determinado.

Se trabaja con dos supuestos, en primer lugar se determina que el texto considerado como valioso es largo, mientras que los textos 'basura' suelen disponer de una menor cantidad de información. Por otro lado, se determina que los textos de valor vienen agrupados.

Imaginemos un documento HTML sencillo en el que aplicando una determinada expresión XPath como la siguiente, `/text()[string-length(normalize-space()) >20]`, se seleccionen todos los nodos de texto que tengan una longitud de cadena superior a veinte.

```
/html/body/div/article
/html/body/div/div
/html/body/div/footer
```

Dada una posible solución como la anterior, se debe contar el numero de nodos hijo de cada uno con el objetivo de crear una distribución de frecuencias de cada conjunto a través de un histograma.

```
/html/body/div/article | -- -- -- --
/html/body/div/div      | --
/html/body/div/footer   | --
```

Tras este proceso y con una simple comprensión de la lista de conjuntos, es posible adquirir algunos, pero no todos los nodos de texto que existen en el subárbol deseado. Veamos el funcionamiento de dicho algoritmo.

Listado 3.4: Libextract - Funcionamiento de eatiht

```
text_node_parents = HtmlTree.execute_xpath_expression('/path/to/text()/..')

for node in text_node_parents:
    partitions = []
    for child in node.children:
        partitions.insert( TextSplitter().split( child ))
    node.children = partitions

histogram = {}
for node in text_node_parents:
    histogram[node.parent.path] = node.children.count

max_path = argmax(histogram)
main_text_nodes = [node for node in text_node_parents if max_path in node.path]
```

3.2.6 `html_text`

`html_text` [21] es una biblioteca de Python encargada de convertir un documento HTML en texto plano. Las principales diferencias frente a algoritmos como **Beautiful Soup**, 3.2.2 donde se emplean expresiones del tipo `.get_text()`, o frente a expresiones XPath como `.xpath('//text())`, son las siguientes:

1. El texto extraído en este caso, no contiene código JavaScript del propio documento, así como comentarios o cualquier contenido que no sea visible por el usuario.
2. La normalización de espacios también se tiene en cuenta, pero de una manera más inteligente que con el uso simple de expresiones XPath, `.xpath('normalize-space())`, ya que añade espacios alrededor de los elementos en línea, y evitar añadir espacios extra para la puntuación.
3. Se añaden nuevas líneas de forma frecuente, por ejemplo después de los encabezados o párrafos, para que el texto de salida se parezca más a cómo se presenta en los navegadores.

En cuanto a la heurística, en primer lugar se limpia el documento empleando como instancia `lxml.html.clean.Cleaner`, y después, se procede con la conversión de árbol de objetos a texto, empleando para ello lo que se conoce como `'chunks'`.

3.2.6.1 Normalización de espacio

El uso habitual del algoritmo se consigue ejecutando la función de extracción de texto principal, de forma que un documento HTML se pasa por parámetro y se devuelve como salida la parte de información que se determina como valiosa.

```
>>> html_text.extract_text('<h1>Hello</h1> world!')
# 'Hello\n\nworld!'
```

En cuanto a la estructura del texto extraído, es posible personalizar cómo se añaden nuevas líneas al mismo utilizando los argumentos `newline_tags` y `double_newline_tags`. De esta forma el algoritmo es capaz de distinguir de forma automática aquellas secciones del documento que deben ser separadas por línea simple o por doble línea.

```
NEWLINE_TAGS = frozenset([
    'article', 'aside', 'br', 'dd', 'details', 'div', 'dt', 'fieldset',
    'figcaption', 'footer', 'form', 'header', 'hr', 'legend', ...
])

DOUBLE_NEWLINE_TAGS = frozenset([
    'blockquote', 'dl', 'figure', 'h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'ol',
    'p', 'pre', 'title', 'ul'
])
```

Imaginemos que se desea ejecutar `html_text` sobre un documento HTML como el siguiente `<div>Hello</div>world!`, donde el texto principal esta separado por una etiqueta de tipo `<div>`. A partir de la gestión de espacio según el tipo de etiqueta, se determina la solución obtenida.

```
>>> newline_tags = html_text.NEWLINE_TAGS - {'div'}
>>> html_text.extract_text('<div>Hello</div> world!',
...                        newline_tags=newline_tags)
# 'Hello world!'
```


3.2.7 html2text

html2text [22] es un script de Python que convierte un documento HTML a texto ASCII, el cual también resulta ser un formato de texto válido a HTML. Imaginemos que se dispone de un documento HTML sencillo como el siguiente, '`<p>Zed's dead baby, Zed's dead.</p>`', veamos la salida obtenida:

```
>>> html2text("<p><strong>Zed's</strong> dead baby, <em>Zed's</em> dead.</p>")
# **Zed's** dead baby, _Zed's_ dead.
```

Si por el contrario se desea que la solución sea limpia, es posible eliminar todos los caracteres de marcado del texto obtenido empleando la función *handle* como método de extracción.

```
>>> handle("<p><strong>Zed's</strong> dead baby, <em>Zed's</em> dead.</p>")
# Zed's dead baby, Zed's dead.
```

En cuanto a la heurística de extracción, el algoritmo simplemente analiza el documento a través de *html.parser*, y a continuación envuelven todos los párrafos del texto proporcionado.

3.2.8 Readability

Dado un documento HTML, **Readability** [23] extrae el cuerpo del texto principal del mismo. Previamente a la extracción de texto se produce una limpieza, donde se eliminan *tags* y atributos del propio documento. Este preprocesamiento se lleva a cabo mediante el uso de expresiones regulares.

3.2.8.1 Heurística de nodo candidato

La heurística de esta biblioteca es similar a la empleada por **Goose3** 3.2.11 para calcular el mejor nodo del árbol. A partir de una puntuación dada para cada nodo se determina aquel cuya puntuación se máxima con el objetivo de obtener el nodo con una mayor cantidad de información.

Listado 3.5: Readability - Selección del nodo candidato

```
def select_best_candidate(self, candidates):
    if not candidates:
        return None

    sorted_candidates = sorted(
        candidates.values(), key=lambda x: x["content_score"], reverse=True)

    best_candidate = sorted_candidates[0]
    return best_candidate
```

Una vez seleccionado dicho nodo, se busca en nodos hermanos con el objetivo de encontrar aquellos cuya información esté relacionada con el mismo, y, por tanto, sean de valor.

3.2.9 Trafilatura

Al igual que muchas otras, **Trafilatura** [24] es una herramienta de búsqueda de texto en la web que descarga, analiza y extrae datos de documentos HTML. El algoritmo de extracción no solo se centra en los metadatos, el cuerpo del texto o comentarios, también trata de conservar parte del formato y la estructura de la página.

En cuanto a la heurística, esta se fundamenta en una cascada de filtros determinados por una serie de reglas y metodología del contenido. En las siguientes secciones se determinan los pasos que realiza el algoritmo para una extracción exitosa.

3.2.9.1 Delimitación de contenido

El objetivo de esta primera fase es separar el contenido con valor del contenido basura o *'boilerplate'*. Para ello el algoritmo hace uso de expresiones XPath dirigidas tanto a atributos HTML, como al contenido principal del documento.

```
DISCARD_XPATH = [
    '''//*[ (self::div or self::item or self::list or self::p or self...) ][
        contains(@id, "related") or contains(@class, "nav") or
        contains(@class, "subnav") or contains(@id, "cookie") or ...]'''
]
```

Inicialmente, el uso de expresiones como las mostradas se emplea para excluir partes o secciones no deseadas del código HTML, por ejemplo la expresión `<div class='nav'>` eliminaría todo el contenido que incluyese cualquier etiqueta `<div>` de tipo *'navigation'*.

Tras esta exclusión de contenido *'boilerplate'*, el algoritmo se centra nuevamente en el uso de expresiones XPath, pero en este caso que abarquen todo el contenido de valor a extraer. Expresiones como `<section id='entry-content'>` o como las mostradas a continuación, buscan partes de contenido potencialmente valioso.

```
BODY_XPATH = [
    '''//*[ (self::article or self::div or self::main or self::section) ][
        contains(@id, "content-main") or
        contains(@class, "entry-content") or
        contains(@class, "content-main") or ...]'''
]
```

Las mismas operaciones se realizan para los comentarios en caso de que estos formen parte de la extracción. Los nodos seleccionados del árbol HTML se procesan, es decir, se comprueba su relevancia y se simplifican en cuanto a su estructura HTML.

```
COMMENTS_XPATH = [
    '''//*[ (self::div or self::section or self::list) ][
        contains(@id, 'commentlist') or
        contains(@class, 'commentlist') or
        contains(@class, 'comment-page') or ...]'''
]
```

3.2.9.2 Algoritmos de respaldo

Si se detecta que la extracción realizada ha sido defectuosa, se ejecuta lo que se conocen como algoritmos de respaldo. Estos algoritmos aplican una heurística basada en la longitud de las líneas, la relación texto/marcado y la posición/profundidad de los elementos en el árbol HTML.

Listado 3.6: Trafilatura - Readability como algoritmo de respaldo

```
def try_readability(htmlinput, url):
    '''Safety net: try with the generic algorithm readability'''
    try:
        doc = LXMLDocument(htmlinput, url, min_text_length=25, retry_length=250)
        return html.fromstring(doc.summary(html_partial=True), parser=HTML_PARSER)
    except (etree.SerialisationError, Unparseable):
        return etree.Element('div')
```

Esta fase de la extracción combina dos bibliotecas para su funcionamiento, tanto **Readability 3.2.8** como **jusText 3.2.3** se encargan de servir como redes de seguridad y *fallbacks* en caso de que la extracción anterior resultase errónea. Se muestra el código fuente de ambos en 3.6 y 3.7.

Listado 3.7: Trafilatura - jusText como algoritmo de respaldo

```
def justext_rescue(tree, url, target_language, postbody, len_text, text):
    '''Try to use justext algorithm as a second fallback'''
    result_bool = False
    temp_post_algo = try_justext(tree, url, target_language)
    if temp_post_algo is not None:
        temp_text = trim(' '.join(temp_post_algo.itertext()))
        len_algo = len(temp_text)
        if len_algo > len_text:
            postbody, text, len_text = temp_post_algo, temp_text, len_algo
            result_bool = True
    return postbody, text, len_text, result_bool
```

Una vez aplicados estos algoritmos, su resultado se compara con la extracción inicial con el objetivo de determinar cual es la más eficaz en términos de longitud e impurezas.

3.2.9.3 Extracción base

En caso de que ni la delimitación de contenido, ni los algoritmos de respaldo funcionen, se ejecuta una extracción base para buscar elementos de texto '*salvajes*' que probablemente se hayan pasado por alto. Esto implica descartar las partes no deseadas, y buscar cualquier elemento que pueda contener contenido textual útil, por ejemplo elementos `<div>` sin párrafos.

Como resultado de toda esta consecución de algoritmos se obtienen los textos principales y los potenciales comentarios de los documentos HTML analizados, con la posibilidad de conservar elementos estructurales como párrafos, títulos, listas, comillas o saltos de línea. En la extracción, también se incluyen metadatos, es decir, título, nombre del sitio, autor, fecha, categorías y etiquetas del propio documento.

3.2.10 Dragnet

Dragnet [25] es una biblioteca de *web scraping* escrita en Python, la cual empleando inteligencia artificial es capaz de extraer el texto principal de documento HTML.

Algoritmos de extracción ya mostrados, como **inscriptis** 3.2.1, trabajan dividiendo el documento HTML en una secuencia de bloques que posteriormente son clasificados empleando la densidad de los mismos como instrumento de medida. **Dragnet** combina esta clasificación de bloques con el aprendizaje automático para minimizar el contenido basura y maximizar la obtención de información valiosa.

3.2.10.1 Extracción enfocada al aprendizaje automatico

El algoritmo comienza dividiendo el documento en una secuencia de bloques, utilizando el DOM ¹ y un conjunto específico de etiquetas como `<div>`, `<p>` o `<h1>`, que modifican la estructura del propio documento. Para crear esta secuencia de bloques el algoritmo itera por el DOM, y cada vez que se encuentra cierto tipo de etiqueta se crea un nuevo bloque.

Una vez dividido el documento en bloques, se asocia un token con cada bloque del mismo. Estos tokens serán útiles en un clasificador para separar el contenido con valor del contenido *'boilerplate'* a nivel de bloque. Cualquier bloque con más del 10% de los tokens extraídos se etiqueta como contenido valioso.

3.2.10.2 Características del modelo

Además de la división por bloques y del aprendizaje automático, esta biblioteca introduce una serie de características diseñadas heurísticamente para capturar información semántica en el código HTML que dejan los programadores.

En primer lugar, muchos atributos como *id* o *class*, y etiquetas HTML incluyen tokens de tipo *comment*, *header* y *nav*. Estos nombres descriptivos son utilizados por los desarrolladores cuando programan código en CSS y JavaScript y, puesto que se eligen con el objetivo de que tengan sentido para el programador, incorporan cierta información semántica sobre el contenido del bloque.

```
attribute_tokens = (
    ('id',
     ('nav', 'ss', 'top', 'content', 'link', 'title', 'comment',...)),
    ('class',
     ('menu', 'widget', 'nav', 'share', 'facebook', 'cat', 'top', 'content',
      'item', 'twitter', 'button', 'title', 'header', 'ss', 'post', ...))
)
```

Otro conjunto de características se corresponde con la densidad de texto y densidad de enlaces. La intuición es que los bloques de contenido tienen una mayor densidad de texto y una menor densidad de enlaces que los bloques denominados *'boilerplate'*, ya que muchos de estos últimos son bloques que consisten en breves fragmentos de palabras o son principalmente texto ancla.

Por último, la tercera característica incluye varias ideas, por un lado, la relación entre la longitud del texto y el número de etiquetas HTML, y en segundo lugar la agrupación de bloques *'boilerplate'*. La idea final es combinar la proporción de etiquetas de contenido valioso y la proporción de etiquetas de contenido basura en un enfoque de agrupación de *k-means* no supervisado, de modo que los bloques sin contenido se agrupen naturalmente cerca del origen.

¹El DOM define la manera en que objetos y elementos se relacionan entre sí en el navegador y en el documento.

3.2.11 Goose3

Otra biblioteca de programación cuyo objetivo es la extracción de texto en artículos es **Goose** [26]. Inicialmente, el proyecto fue desarrollado en Java, pero a lo largo del tiempo han ido surgiendo nuevas versiones en diferentes lenguajes de programación como Scala o Python.

El objetivo del software es tomar cualquier documento de noticias o página web de tipo artículo y además del cuerpo principal, extraer la imagen, videos, descripción y meta tags del mismo.

3.2.11.1 Cálculo del mejor nodo

La técnica que emplea esta biblioteca para la extracción de contenido principal es muy similar a la que se emplea en muchos otros algoritmos de extracción, donde un árbol se recorre con el fin de obtener la mayor cantidad de información posible.

El primer paso consiste en recorrer todo los posibles nodos del árbol para almacenar aquellos que posean texto legible. En el fragmento de código 3.8 se muestra una primera parte de como Goose realiza dicho recorrido.

Listado 3.8: Goose3 - Cálculo del mejor nodo 1

```
def calculate_best_node(self, doc):
    nodes_to_check = self.nodes_to_check(doc)

    for node in nodes_to_check:
        text_node = self.parser.getText(node)
        word_stats = self.stopwords_class(...get_stopword_count(text_node))
        high_link_density = self.is_highlink_density(node)
        if word_stats.get_stopword_count() > 2 and not high_link_density:
            nodes_with_text.append(node)
```

Una vez los nodos han sido almacenados, se determina una puntuación para cada uno de ellos. Esta puntuación se calcula en función del número de palabras con sentido gramatical del texto contenido en cada nodo, más una cierta cantidad denominada *'boost-score'*.

```
upscore = int(word_stats.get_stopword_count() + boost_score)
```

Una vez determinada la puntuación de cada nodo se procede de la manera mostrada en 3.9 donde se compara la puntuación de cada nodo, con el objetivo de encontrar aquel cuya puntuación sea máxima.

Listado 3.9: Goose3 - Cálculo del mejor nodo 2

```
for itm in parent_nodes:
    score = self.get_score(itm)
    if score > top_node_score:
        top_node = itm
        top_node_score = score
    if top_node is None:
        top_node = itm
return top_node
```

3.2.12 Newspaper3k

Inspirada en otras bibliotecas de HTTP e impulsada por el analizador *lxml*, **Newspaper3k** [27] es una biblioteca de Python encargada de realizar minado en la web. Entre muchas de las características que hacen destacar esta biblioteca sobre el resto se puede destacar las siguientes:

- Capacidad de descarga de artículos en paralelo gracias al uso de múltiples hilos.
- Identificación de URL's de artículos.
- Extracción de texto de un documento HTML.
- Extracción de imágenes de un documento HTML.
- Extracción de palabras clave de un texto.
- Resumen del contenido de un documento texto.
- Extracción de términos de tendencia de Google.
- Trabajo con múltiples idiomas.

En cuanto a la heurística de extracción, se usa gran parte del código fuente de **Goose** 3.2.11, por lo que la fuente de cálculo del mejor nodo, así como al selección del mismo a través del número de palabras clave será esencial en este caso también.

3.2.12.1 Heurística según el idioma

Uno de los aspectos importantes del algoritmo de extracción de texto gira en torno al recuento del número de palabras clave o palabras con sentido gramatical presentes en un texto. Estas palabras son algunas de las palabras funcionales cortas más comunes de un idioma, como *the, is, at, which, on...*

Al trabajar con múltiples idiomas, esta heurística debe cambiar, puesto que distintos tipos de palabras aparecen para cada uno. Para idiomas latinos, se proporciona una lista de palabras clave en formato *stopwords-<código de idioma>*. A continuación, se toma un texto de entrada y se convierte en palabras dividiendo los espacios en blanco, para más tarde realizar un conteo y enumerar la cantidad de palabras clave.

Para los idiomas no latinos, es necesario dividir las palabras en tokens de una manera diferente, puesto que la división por espacios en blanco no funciona para idiomas como el chino o el árabe. Para el idioma chino se emplea una nueva biblioteca de código abierto llamada *jieba*, sin embargo, para el árabe se utiliza un tokenizador especial, *nltk* que se encarga de hacer el mismo trabajo como se muestra en el fragmento de código 3.10.

Listado 3.10: Newspaper3k - Heurística en el árabe

```
def candidate_words(self, stripped_input):
    import nltk
    s = nltk.stem.isri.ISRIStemmer()
    words = []
    for word in nltk.tokenize.wordpunct_tokenize(stripped_input):
        words.append(s.stem(word))
    return words
```

3.2.13 BoilerPy3/BoilerpipeR

BoilerpipeR [28] en R, y **BoilerPy3** [29] en Python, son paquetes que proporcionan una interfaz para una biblioteca Java. Soportan la extracción genérica del contenido del texto principal de los archivos HTML y, por tanto, elimina los anuncios, las barras laterales y cabeceras del contenido fuente HTML.

En cuanto a la heurística, el algoritmo original se basa en separar el contenido boilerplate del contenido valioso. Para ello se emplean algoritmos de apoyo con el fin de detectar y eliminar el exceso de desorden alrededor del contenido principal de una página web.

3.2.13.1 Multiplicidad de extractores

El paquete dispone de varios extractores para diferentes situación de extracción, con múltiples heurísticas las cuales se especifican a continuación. Por otro lado, cada extractor es capaz de manejar las excepciones que se produzcan durante la extracción y devolver cualquier texto que haya sido extraído con éxito.

- **DefaultExtractor**: extractor muy simple, sin heurística, bastante genérico pero completo.
- **ArticleExtractor**: gestiona los artículos de noticias. Funciona muy bien para la mayoría de los tipos de HTML tipo artículo.
- **ArticleSentencesExtractor**: se encarga de la extracción de frases en artículos de prensa.
- **LargestContentExtractor**: se centra en extraer el componente de texto más grande de una página.
- **CanolaExtractor**: : extractor de texto entrenado en un *krdwd*.
- **KeepEverythingExtractor**: marca todo como contenido, debería devolver el texto de entrada. Se emplea en caso de error, para comprobar si el error procede de un extractor en particular o de otro lugar.
- **NumWordsRulesExtractor**: se basa únicamente en el número de palabras por bloque.

3.3 Paquetes de R encontrados durante el proceso de búsqueda

Además de la introducción de aquellas bibliotecas de Python destinadas al *web scraping*, se incluye en la evaluación paquetes con la misma funcionalidad desarrollados en R. Por ello, se realizará una sinopsis de aquellos paquetes de R que realicen *web scraping* y puedan ser incluidos en el proceso de evaluación.

rvest	rcrawler	htm2txt	rselenium	scraper	boilerpiper
xml2	xml2	N/A	XML	XML	XML

Tabla 3.5: Emparejamientos paquete-analizador

Antes de comenzar con la sinopsis de paquetes, es conveniente revisar el apéndice B, donde se realiza una breve introducción de aquellos analizadores empleados en los mismos. Los 'emparejamientos' entre paquete-analizador se recogen en la tabla 3.5 a modo de resumen.

Se puede observar que **htm2txt** no emplea ningún tipo de analizador. En contrapartida hace uso de expresiones regulares para eliminar las etiquetas html.

3.3.1 rvest

Inspirada en otras bibliotecas como **Beautiful Soup** 3.2.2, **rvest** [30] es una de las herramientas mas comunes de minado web. Este paquete esta diseñado para trabajar con *magrittr*, para facilitar tareas comunes del *web scraping*, y con *polite*, para garantizar que se respete el archivo *robots.txt* y no saturar el sitio web con multiples peticiones.

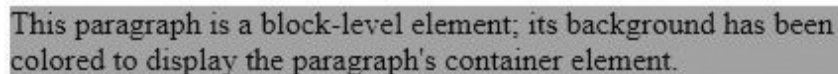
En el apéndice A se puede comprobar el funcionamiento del paquete durante la extracción de parte de un documento HTML. Ademas, se observa que el paquete actúa como *wrapper* entre los paquetes *xml2* y *httr*, facilitando así la descarga y manipulación de un fichero HTML.

3.3.1.1 Elementos a nivel de bloque vs elementos en línea

El objetivo de **rvest** es que a partir de una determinada entrada, ya sea una URL o un documento HTML, extraer el texto principal emulando a un navegador con el fin de que el texto extraído se acerque lo máximo posible a la realidad. Para ello el algoritmo se basa en lo que se conoce como elementos a nivel de bloque.

Los elementos de un documento HTML se clasifican como elementos de nivel de bloque o de nivel de línea. Un elemento de nivel de bloque ocupa todo el espacio horizontal de su contenedor, y un espacio vertical igual a la altura de su contenido, creando así lo que se conoce como bloque. Se muestra un ejemplo a continuación.

```
<p>This paragraph is a block-level element; its background has been
colored to display the paragraph's container element.</p>
```



This paragraph is a block-level element; its background has been colored to display the paragraph's container element.

Figura 3.5: rvest - Elementos a nivel de bloque

Los elementos de nivel de bloque pueden contener elementos en línea y otros elementos de nivel de bloque. Esta distinción estructural lleva implícita la idea de que los elementos de bloque crean estructuras más grandes que los elementos en línea. Ademas, los elementos de nivel de bloque comienzan en líneas nuevas, pero los elementos en línea pueden empezar en cualquier parte de una línea. Los elementos que **rvest** determina a nivel de bloque son los siguientes:

```
block_tag <- c( "address", "article", "aside", "blockquote", "details",
               "dialog", "dd", "div", "dl", "dt", "fieldset", "figcaption", "figure",
               "footer", "form", "h1", "h2", "h3", "h4", "h5", "h6", "header", ...)
```

3.3.2 Rcrawler

Rcrawler [31] tiene como objetivo el rastreo y la extracción de datos estructurados de forma paralela. Está diseñado para rastrear, parsear y almacenar páginas web para producir datos que puede ser utilizados con posterioridad para aplicaciones de análisis.

Entre las características de **RCrawler** se destaca, el rastreo multihilo, la extracción de contenidos y la detección de contenidos duplicados. Además, incluye funcionalidades como el filtrado de URL's y tipos de contenido, el control del nivel de profundidad y un analizador de *robot.txt*.

La principal diferencia entre **Rcrawler** y otros paquetes de *web scraping* como **rvest**, es que mientras **rvest** extrae datos de un documento HTML navegando a través de selectores, **Rcrawler** recorre, analiza y extrae todos los datos de todas las páginas web con un solo comando.

3.3.2.1 Arquitectura y proceso de extracción en Rcrawler

Inspirado en otros paquetes como **Mercator** o **Ubicrawler**, y con el objetivo evitar una sobrecarga del entorno, la arquitectura de **Rcrawler** es lo más óptima y simple posible. En la figura 3.6 se muestra la propia arquitectura de la herramienta [32].

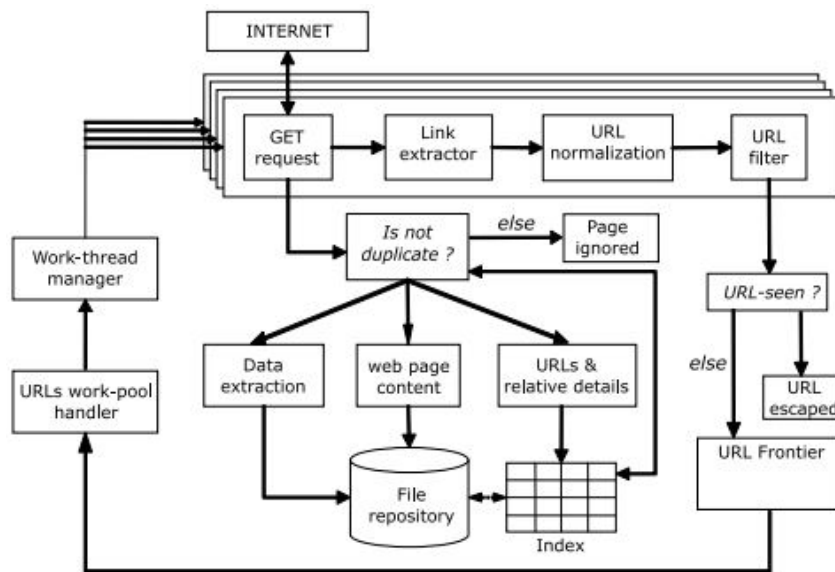


Figura 3.6: Rcrawler - Arquitectura y componentes principales de Rcrawler

En primer lugar, el rastreador pone en marcha el entorno de trabajo, es decir, la estructura del índice, el repositorio de documentos web y los nodos del clúster para la computación en paralelo. Una vez puesto en marcha el entorno de trabajo, se procede con el rastreo el cual es realizado por múltiples hilos de trabajo. Por otro lado, el componente *work-pool-handler* prepara un *pool* de URL's para procesar en paralelo, donde cada nodo ejecuta las siguientes funciones para cada URL:

1. Descargar el documento correspondiente y su cabecera HTTP mediante una petición GET.
2. Analizar y extraer todos los enlaces contenidos en el documento.
3. Proceder a la canonización y normalización de URL's.
4. Aplicar un filtro de URL's, manteniendo sólo aquellas que coincidan con la configuración proporcionada por el usuario, el tipo de archivo y las URL's específicas del dominio.

Tras la descarga del documento a través de la petición GET, se determina si este ya ha sido analizado previamente. En caso de que el documento no esté duplicado se procede con el rastreo y la extracción de datos, con el objetivo de indexar y almacenar los documentos encontrados.

Como resultado, para cada URL cada nodo devuelve su documento HTML, los detalles de la cabecera HTTP y la lista de enlaces externos descubiertos. La función *URL-seen* comprueba si la URL ya ha sido procesada.

3.3.3 htm2txt

Otro de los paquetes propios de R que se encargan de realizar *web scraping* es **htm2txt** [33]. Convierte un documento html en textos simples eliminando todas las etiquetas html. Este paquete utiliza expresiones regulares para eliminar las etiquetas html. También ofrece las funciones *gettxt()* y *browse()*, que permiten obtener o navegar por los textos de una determinada página web.

El uso de estas expresiones regulares se hacen notar durante la limpieza del texto extraído, donde la función *gsub* toma una notable relevancia. Una vez el texto ha sido 'limpiado' se retorna el mismo.

3.3.4 scrapeR

scrapeR [34] es otro paquete software que pretende hacer *web scraping* sobre documentos basados en la web. El paquete en sí es bastante simple, pues consta de una única función encargada de la obtención y creación del árbol DOM para su posterior iteración.

Puesto que no dispone de funciones propias de recorrido del DOM, **scrapeR** recurre a funciones y métodos del paquete *XML* para realizar su labor. Esto tiene una ventaja significativa, y es que permite ejecutar la petición de varias URL's de forma paralela.

3.3.5 RSelenium

RSelenium [35] es un paquete que tiene como objetivo facilitar la conexión a un servidor de Selenium. Además de *web scraping*, **RSelenium** también permite hacer pruebas de unidad y de regresión en las aplicaciones web que se desarrollen utilizando un gran rango de navegadores y de sistemas operativos [6]. Gracias a esto permite realizar entre otras las siguientes tareas:

1. Navegar a través de cualquier documento HTML.
2. Acceder a los elementos del DOM utilizando selectores de id, de clase, de nombre de etiqueta, selectores XPath o CSS.
3. Enviar eventos de teclado y de ratón a los elementos de la página que se determinen.
4. Inyectar JavaScript en la página.
5. Utilizar marcos y ventanas diferenciadas.

A modo de anécdota, cabe destacar que mediante el uso de **RSelenium** es posible automatizar los navegadores tanto de forma local como remota. Permite manejar un navegador web nativamente como lo haría un usuario, lo que supone un salto adelante en términos de automatización del *web scraping*.

3.4 Paquetes descartados para el proceso de evaluación

A lo largo de las secciones 3.2 y 3.3 se han introducido y desarrollado aquellas herramientas de *web scraping* más comunes en el entorno de programación. Con el punto de mira en los siguientes capítulos, se pretende enumerar aquellos paquetes descartados para el proceso de evaluación.

Ya sea por la simpleza de su heurística, por la similitud con otras herramientas, o por la dificultad en la instalación y uso de las mismas, las bibliotecas seleccionadas para el descarte son las siguientes:

- **Dragnet**: Uno de los principales motivos por el que Dragnet a sido descartado es por su complejidad en la instalación. La necesidad de un Docker hace que la mayoría de usuarios no programadores hagan uso de otras herramientas en su lugar.
- **news-please**: El descarte de **news-please** tiene que ver con el uso de otras herramientas para el proceso de *web scraping*. La ausencia de heurística propia y el empleo de otros algoritmos como **Readability** hace que la evaluación de **news-please** no tenga sentido.
- **Libextract**: Durante el desarrollo de la herramienta de evaluación, se ha detectado que en este caso no se cumplían los requisitos mínimos para la misma. La extracción resultaba errónea para ciertos documentos HTML.
- **Newspaper3k**: Tal y como se ha indicado en la sinopsis, el algoritmo emplea gran parte de la heurística de **Goose3** para la extracción de contenido. A falta de heurística propia no tendría sentido realizar una evolución del mismo.
- **scrapeR**: La sencillez de la herramienta, y su falta de heurística hace que el descarte de la misma sea necesario. Su única función necesita herramientas XPath para la búsqueda de información a través del DOM.
- **RSelenium**: Además del necesario emplea de un Docker para su uso, al igual que **scrapeR**, **RSelenium** carece de heurística propia, ya que para la extracción de texto emplea expresiones XPath y selectores CSS únicamente.

Por último, es notable la falta de algoritmos de R en la herramienta de evaluación. Durante el proceso de búsqueda e investigación, la mayoría de ellos no poseen ni siquiera heurística. Trabajar únicamente a partir de un analizador y expresiones XPath, hace que la evaluación de los mismos no tenga sentido alguno.

Capítulo 4

Selección de variables de análisis y proceso de estudio

Capítulo 5

Presupuesto

Blah, blah, blah.

Bibliografía

- [1] A. Mehlführer, “Web scraping, a tool evaluation,” Master’s thesis, Technische Universität Wien, 2009.
- [2] B. Zhao, *Web Scraping*, 05 2017, pp. 1–3.
- [3] O. Castrillo-Fernández, *Web Scraping: Applications and Tools*, ser. Report No. 2015 / 10. European Public Sector Information Platform, 2015.
- [4] D. Glez-Peña, A. Lourenco, H. López-Fernández, M. Reboiro-Jato, and F. Fdez-Riverola, “Web scraping technologies in an API world,” *Briefings in Bioinformatics*, vol. 15, no. 5, pp. 788–797, April 2013.
- [5] J. Ooms, *curl: A Modern and Flexible Web Client for R*, 2019, r package version 4.3. [Online]. Available: <https://CRAN.R-project.org/package=curl>
- [6] D. Francisco López, “Revisión de los paquetes para realizar web scraping en r: Análisis cualitativo y cuantitativo,” Master’s thesis, Universidad de Alcalá Escuela Politécnica Superior, 2018.
- [7] S. L. Nerdal, “Newsenhancer,” Master’s thesis, Department of Informatics, University of Bergen, 2018.
- [8] S. Broucke Vande and B. Baesens, *Web Scraping for Data Science with Python*, 11 2017, pp. 14–16.
- [9] D. Doran and S. Gokhale, “Web robot detection techniques: Overview and limitations,” *Data Mining and Knowledge Discovery*, vol. 22, pp. 183–210, 06 2011.
- [10] M. Beckman, S. Guerrier, J. Lee, R. Molinari, S. Orso, and I. Rudnytskyi, “An introduction to statistical programming methods with r, chapter 10,” <https://smac-group.github.io/ds/index.html>. [Ultimo acceso 27/octubre/2021].
- [11] github, “Github,” 2020. [Online]. Available: <https://github.com/>
- [12] “Cran,” <https://cran.r-project.org/>. [Ultimo acceso 07/diciembre/2021].
- [13] Python package index - pypi. [Online]. Available: <https://pypi.org/>
- [14] A. Weichselbraun, “Inscriptis - a python-based html to text conversion library optimized for knowledge extraction from the web,” *Journal of Open Source Software*, vol. 6, no. 66, p. 3557, 2021. [Online]. Available: <https://doi.org/10.21105/joss.03557>
- [15] L. Richardson, “Beautiful soup documentation,” *April*, 2007.
- [16] J. Pomikálek, “Removing boilerplate and duplicate content from web corpora,” Ph.D. dissertation, Masaryk university, Faculty of informatics, Brno, Czech Republic, 2011.

- [17] F. Hamborg, N. Meuschke, C. Breiteringer, and B. Gipp, “news-please: A generic news crawler and extractor,” in *Proceedings of the 15th International Symposium of Information Science*, March 2017, pp. 218–223.
- [18] H. Bjerkander and E. Karlsson, “Distributed web-crawler,” Master’s thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2005.
- [19] R. Palacios and E. Jun, “Libextract: extract data from websites,” <https://github.com/datalib/libextract>. [Ultimo acceso 18/enero/2022].
- [20] R. Palacios, “eatihit, a simple tool used to extract an article’s text in html documents.” <http://rodricios.github.io/eatihit/>. [Ultimo acceso 18/enero/2022].
- [21] M. Korobov, “Html-text,” <https://github.com/TeamHG-Memex/html-text>. [Ultimo acceso 15/diciembre/2021].
- [22] A. Savand, “html2text,” <https://github.com/Alir3z4/html2text>. [Ultimo acceso 17/enero/2022].
- [23] Y. Baburov, “python-readability,” <https://github.com/buriy/python-readability>. [Ultimo acceso 17/enero/2022].
- [24] A. Barbaresi, “Trafilatura: A Web Scraping Library and Command-Line Tool for Text Discovery and Extraction,” in *Proceedings of the Joint Conference of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 2021, pp. 122–131. [Online]. Available: <https://aclanthology.org/2021.acl-demo.15>
- [25] M. E. Peters and D. Lecocq, “Content extraction using diverse feature sets,” in *Proceedings of the 22nd International Conference on World Wide Web*, ser. WWW ’13 Companion. New York, NY, USA: Association for Computing Machinery, 2013, pp. 89–90. [Online]. Available: <https://doi.org/10.1145/2487788.2487828>
- [26] M. Lababidi, “Goose3,” <https://goose3.readthedocs.io/en/latest/>. [Ultimo acceso 11/diciembre/2021].
- [27] L. OuYang, *newspaper Documentation*, 2018, release 0.0.2. [Online]. Available: <https://buildmedia.readthedocs.org/media/pdf/newspaper/latest/newspaper.pdf>
- [28] S. A. file., *boilerpipeR: Interface to the Boilerpipe Java Library*, 2021, r package version 1.3.2. [Online]. Available: <https://CRAN.R-project.org/package=boilerpipeR>
- [29] J. Riebold, “Boilerpy3,” <https://github.com/jmriebold/BoilerPy3>. [Ultimo acceso 19/enero/2022].
- [30] H. Wickham, *rvest: Easily Harvest (Scrape) Web Pages*, 2020, r package version 0.3.6. [Online]. Available: <https://CRAN.R-project.org/package=rvest>
- [31] S. Khalil, *Rcrawler: Web Crawler and Scraper*, 2018, r package version 0.1.9-1. [Online]. Available: <https://CRAN.R-project.org/package=Rcrawler>
- [32] S. Khalil and M. Fakir, “Rcrawler: An r package for parallel web crawling and scraping,” *SoftwareX*, vol. 6, pp. 98–106, 2017.
- [33] S. Park, *htm2txt: Convert Html into Text*, 2017, r package version 2.1.1. [Online]. Available: <https://CRAN.R-project.org/package=htm2txt>

- [34] R. M. Acton, *scrapeR: Tools for Scraping Data from HTML and XML Documents*, 2010, r package version 0.1.6. [Online]. Available: <https://CRAN.R-project.org/package=scrapeR>
- [35] J. Harrison, *RSelenium: R Bindings for 'Selenium WebDriver'*, 2020, r package version 1.7.7. [Online]. Available: <https://CRAN.R-project.org/package=RSelenium>
- [36] P. Meissner and K. Ren, *robotstxt: A 'robots.txt' Parser and 'Webbot'/'Spider'/'Crawler' Permissions Checker*, 2020, r package version 0.7.13. [Online]. Available: <https://CRAN.R-project.org/package=robotstxt>
- [37] H. Wickham, J. Hester, and J. Ooms, *xml2: Parse XML*, 2020, r package version 1.3.2. [Online]. Available: <https://CRAN.R-project.org/package=xml2>
- [38] S. M. Bache and H. Wickham, *magrittr: A Forward-Pipe Operator for R*, 2020, r package version 2.0.1. [Online]. Available: <https://CRAN.R-project.org/package=magrittr>
- [39] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.
- [40] S. Behnel, M. Faassen, and I. Bicking, “lxml: Xml and html with python,” 2005.
- [41] J. Graham, S. Sneddon, and contributors, *html5lib Documentation*, 2020, release 1.1. [Online]. Available: https://html5lib.readthedocs.io/_/downloads/en/stable/pdf/
- [42] Python, “html parser - simple html and xhtml parser,” <https://docs.python.org/3/library/html.parser.html>. [Ultimo acceso 20/diciembre/2021].
- [43] D. Temple Lang, *XML: Tools for Parsing and Generating XML Within R and S-Plus*, 2021, r package version 3.99-0.8. [Online]. Available: <https://CRAN.R-project.org/package=XML>

Apéndice A

Funcionamiento básico de un web scraper

Siguiendo las directrices determinadas en la sección 2.1.1, se muestra el funcionamiento de un web scraper durante las tres fases definidas. Para ello se realizará un pequeño ejemplo mostrando el comportamiento del mismo y de como interactúa con el servidor web al que se desea acceder.

Para el desarrollo de este ejemplo se han empleado bibliotecas software basadas en el lenguaje de programación R, capaces de extraer datos de cualquier web. En este caso la web sujeta al análisis será, *imdb* encargada de asignar un ranking entre películas y series.

A.1 Fase de búsqueda

Antes de comenzar con la primera de las tres etapas, debemos asegurarnos que nuestro agente software cumple con todos los aspectos ético-legales descritos en la sección 2.4. Los términos y servicios de la página deberán ser leídos, al igual que el documento '*robots.txt*' con el objetivo de conocer cuáles son los accesos disponibles y el índice de solicitudes a realizar.

En el fragmento de código A.1 se muestra la solicitud al servidor realizada. A través de la biblioteca *robotstxt* [36] y haciendo uso de la función *get_robotstxt* se obtiene el documento deseado.

Es posible que la solicitud del documento no se realice correctamente, pues o bien la página no dispone del documento en ese instante, o la función ha fallado durante su solicitud. En cualquiera de estos dos escenarios, es posible realizar una doble comprobación accediendo al mismo a través de la propia URL, *https://www.imdb.com/robots.txt*.

Listado A.1: Solicitud del documento *robots.txt*

```
install.packages("robotstxt")
library("robotstxt")

robots <- get_robotstxt(domain = "https://www.imdb.com/")

head(robots)
```

```
# robots.txt for https://www.imdb.com properties
User-agent: *
Disallow: /OnThisDay
Disallow: /ads/
Disallow: /ap/
Disallow: /mymovies/
Disallow: /r/
Disallow: /register
Disallow: /registration/
...
```

Una vez se ha accedido al documento *'robots.txt'*, conociendo los posibles accesos al servidor, y determinando el número de solicitudes máximas por segundo, es posible acceder y descargar el fichero HTML de forma segura. Para este propósito se empleará la función *read_html* de la biblioteca *xml2* [37] la cual se detalla a continuación.

Listado A.2: Acceso y descarga del archivo HTML

```
install.packages("xml2")

library("xml2")

url <- "imdb.com/search/title/?count=100&release_date=2016,2016&title_type=feature"
html_doc <- read_html(url)
```

El valor que retorna la función *read_html*, se trata del archivo HTML integro, donde se incluyen cabecera, cuerpo y demás etiquetas del mismo. Una vez que se dispone de la página, es posible comenzar con la extracción de datos de interés de la misma.

A.2 Fase de extracción

Para el minado se empleará *rvest* [30], una de las bibliotecas software más comunes en este aspecto, diseñada para trabajar con *magrittr* [38] y facilitar tareas de la extracción. Además, será necesario el uso de funciones como *html_nodes()* y *html_text()*.

Durante esta fase de extracción se obtendrán tanto los títulos como el ranking asignado a cada película o serie. Para realizar la extracción de forma correcta, se deberá conocer la etiqueta HTML que envuelve dicha información.

Listado A.3: Extracción de datos de interés del documento

```
install.packages("rvest")
install.packages("magrittr")

library("rvest")
library("magrittr")

rank_data <- html_nodes(html_doc, '.text-primary') %>%
  html_text()

title_data <- html_nodes(html_doc, '.lister-item-header_a') %>%
  html_text()

head(rank_data)
head(title_data)
```

```
[1] "1." "2." "3." "4." "5." "6."
```

```
[1] "Animales nocturnos" "Train to Busan" "La llegada (Arrival)"
[4] "Escuadrón suicida" "Deadpool" "Hush (Silencio)"
```

A.3 Fase de transformación

Una vez los datos han sido extraídos, la última fase consiste en la transformación de los mismos. Los datos desordenados deberán ser convertidos en estructuras de datos ordenadas y coherentes con el fin su posible almacenamiento en una base de datos.

Listado A.4: Transformación de datos en un data frame

```
movies_df <- data.frame(Rank = rank_data, Title = title_data)

head(movies_df)
```

	Rank	Title
1	1.	Animales nocturnos
2	2.	Train to Busan
3	3.	La llegada (Arrival)
4	4.	Escuadrón suicida
5	5.	Deadpool
6	6.	Hush (Silencio)

Una vez los datos han sido ordenados en una estructura de datos propia, en este caso un data frame, es posible trabajar con ellos de forma más cómoda y sencilla.

Apéndice B

Analizadores empleados en los paquetes de web scraping

Este apéndice tiene como objetivo introducir aquellos aspectos más relevantes relacionados con los analizadores, también conocidos como *parsers*, empleados en el web scraping. A lo largo de esta sección se realizará una pequeña sinopsis de aquellas herramientas empleadas en los algoritmos de extracción de los paquetes definidos en la sección 3.2.

A modo de introducción cabe destacar que un analizador sintáctico, es un programa informático que analiza una cadena de símbolos según las reglas de una gramática formal [39]. Generalmente, los analizadores se componen de dos partes, por un lado, un analizador sintáctico, y por otro un analizador léxico. Mientras que el analizador léxico crea tokens a partir de una secuencia de caracteres de entrada, el analizador sintáctico convierte los tokens en otras estructuras.

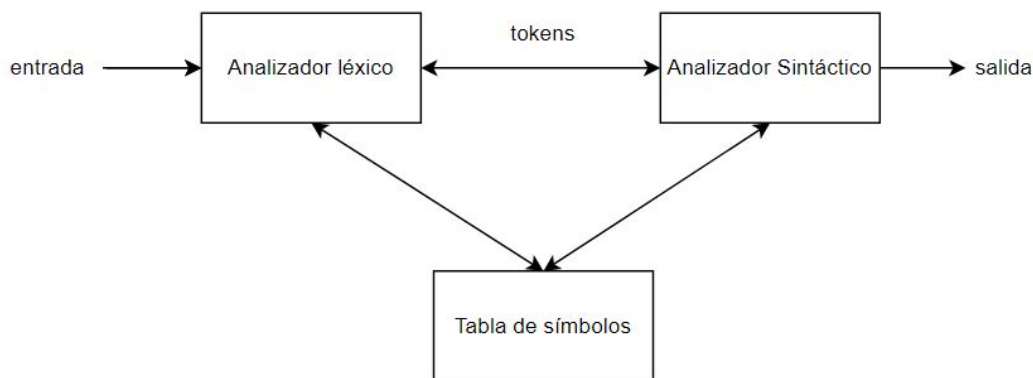


Figura B.1: Estructura básica de un analizador

B.1 lxml

Uno de los analizadores más comunes en todos los algoritmos de minado web es **`lxml`** [40]. Esta biblioteca de Python permite procesar tanto documentos XML como HTML.

Como la gran mayoría de analizadores, **`lxml`** convierte el texto de entrada en una estructura tipo árbol. Esto permite navegar por la propia estructura en busca de la información que se desea de forma sencilla.

En el caso del web scraping, la manera en la que **lxml** es capaz de encontrar información valiosa, es decir texto, es a través de expresiones XPath. Se muestra un pequeño ejemplo a continuación.

```
> html.xpath("string()")
# TEXTTAIL

> html.xpath("//text()")
# ['TEXT', 'TAIL']
```

Cabe destacar que el resultado dado por una expresión XPath es un objeto especial que conoce parte de su estructura. Esto permite ejecutar operaciones sobre este elemento y saber de dónde proviene a través de diferentes métodos.

B.1.1 lxml.html

¿Qué ocurre con los documentos HTML mal formados? Para este propósito, se creó lo que se conoce como **lxml.html** [40]. Un paquete de Python especial para tratar con documentos HTML, el cual a diferencia del analizador base, proporciona una API de elementos propia de HTML, así como una serie de utilidades para tareas comunes de procesamiento de los mismos.

```
> broken_html = "<html><head><body><h1>page title</h3>"
> parser = etree.HTMLParser()
> tree = etree.parse(StringIO(broken_html), parser)
> result = etree.tostring(tree.getroot(), method="html")
# <html><head></head><body><h1>page title</h1></body></html>
```

Como se puede observar, incluso analizando documentos realmente mal formados, el analizador es capaz de crear una estructura propia de un documento HTML. A partir de ahí, es posible aplicar expresiones XPath con el fin de recorrer el árbol generado y recuperar información de valor.

B.1.2 soupparser

Otro de los analizadores propios de **lxml** es **soupparser** [40] propio del paquete **Beautiful Soup**. La forma en la que **lxml** interactúa con **Beautiful Soup**, es a través del módulo **lxml.html.soupparser** el cual proporciona una serie de funciones principales.

Por un lado, tanto *fromstring()* como *parse()* se emplean para analizar ya sea una cadena o un archivo HTML, por otro lado *convert_tree()* se emplea para convertir un árbol **Beautiful Soup** existente en una lista de elementos de nivel superior.

```
> tag_soup = '''<meta/><head></head><body>Hi all<p>'''
> root = fromstring(tag_soup)
# <html><meta/><head></head><body>Hi all<p/></body></html>
```

Imaginemos que se dispone de una cadena HTML mal formada como la mostrada en el ejemplo. Al ser una cadena, se emplea *fromstring()* para realizar un análisis de la misma. Como vemos la salida también puede provocar algún error, puesto que no es exactamente una estructura propia de HTML.

B.2 html5lib

html5lib [41] es un paquete de Python que implementa el algoritmo de análisis sintáctico de HTML5. Proporciona una interfaz similar a la del módulo **lxml.html** B.1.1 con métodos como *fromstring()* o *parse()* que operan de la misma manera que las funciones de análisis de HTML normales.

Al igual que **lxml.html**, este paquete también trabaja con árboles de objetos, pero en este caso normaliza algunos elementos y estructuras a un formato común. Por ejemplo, incluso si una tabla no tiene una etiqueta del tipo *<tbody>*, se inyectará una automáticamente.

```
> tostring(html5parser.fromstring("<table><td>foo"))
# '<table><tbody><tr><td>foo</td></tr></tbody></table>'
```

B.3 html.parser

html.parser [42] es un módulo de Python que define una clase *HTMLParse* como base para analizar archivos HTML y XHTML. Este módulo trabaja alrededor de etiquetas, pues llama a métodos de manejo cuando se encuentran etiquetas de inicio, etiquetas finales, texto, comentarios y otros elementos de marcado.

```
> parser.feed('<p><a class=link href=#main>tag soup</p>></a>')
# Start tag: p
# Start tag: a
# Data      : tag soup
# End tag   : p
# End tag   : a
```

Como se puede observar, el algoritmo es capaz de detectar etiquetas tanto de apertura, como de cierre incluso en documentos HTML mal formados como este. Esto hace que la búsqueda de información valiosa, en este caso texto, sea sencilla.

B.4 XML

XML [43] es un paquete de R que se encarga de realizar el análisis de documentos XML y HTML. Además del proceso de análisis, también proporciona acceso a un intérprete de XPath que permite consultar nodos concretos de los documentos. Como contrapartida, no permite el uso de selectores CSS para consultar XML o HTML analizados.

Adicionalmente, hay que destacar que tiene algunos problemas de liberación de memoria que no puede resolver adecuadamente el recolector de basura de R. Por ello, se han ido creando diferentes soluciones que abordan este problema.

En cuanto al método de análisis, se recorre el árbol buscando la información que desee y se coloca en diferentes formas. Hay dos maneras de hacer esta iteración. Una es recorrerrecursivamente el árbol empezando por el nodo raíz y procesándolo, la otra consiste procesar cada nodo hijo de la misma manera, trabajando en su nombre y atributos y luego en sus hijos, y así sucesivamente.

B.5 xml2

Otro paquete de R que permite trabajar con ficheros XML y HTML es **xml2** [37]. Está basado sobre la biblioteca **libxml2**. El paquete sigue los mismos objetivos que **XML**, por lo que el análisis de ficheros compone la parte principal del mismo.

xml2 tiene una jerarquía de clases muy sencilla que permite no preocuparse por el tipo de objeto se está gestionando. Estas clases son clave y se definen a continuación:

1. `xml_node`: un solo nodo de un documento.
2. `xml_doc`: el documento completo. Actuar sobre un documento suele ser lo mismo que actuar sobre el nodo raíz del documento.
3. `xml_nodeset`: un conjunto de nodos dentro del documento. Las operaciones sobre `xml_nodesets` se vectorizan, aplicando la operación sobre cada nodo del conjunto.

A diferencia del paquete **XML**, **xml2** cuida la gestión de la memoria de forma transparente, liberando aquella que haya sido utilizada por un documento tan pronto como se pierden todas las referencias que apuntan a él. Además, la jerarquía de clases es más simple, por lo que no es necesario pensar exactamente qué tipo de objeto se tiene, **xml2** simplemente hará lo correcto.

Universidad de Alcalá
Escuela Politécnica Superior



ESCUELA POLITECNICA
SUPERIOR



Universidad
de Alcalá