

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

El minado web surge como solución al costoso y tradicional proceso de extracción. Este nuevo procedimiento es adoptado por múltiples empresas e instituciones con el objetivo de automatizar y optimizar el proceso de obtención de información. ¿Qué herramientas existen en el mercado y como funcionan? ¿Qué diferencias existen con otras ya existentes? El objetivo del trabajo es realizar un análisis profundo de los paquetes de programación más importantes dentro del *web scraping*, con el objetivo posterior de efectuar una evaluación cuantitativa de los mismos.

En respuesta a todas estas preguntas, se inicia un proceso de estudio. En el mismo se confecciona un método de evaluación basado, por un lado, en la calidad de la información extraída de cada herramienta, y por otro en el empleo de recursos para cada extracción. La cantidad y el contenido *boilerplate* de la información extraída, así como el uso de CPU/RAM o el tiempo de ejecución serán variables sometidas a prueba.

Los resultados obtenidos reflejan que aquellos algoritmos cuya heurística es más sofisticada suelen presentar mejores soluciones. Se ha detectado, además, que tanto el lenguaje, como el analizador empleado tienen una importancia menor con respecto a la eficacia de la extracción, pero no en la eficiencia. Como conclusión se puede determinar que tanto el objetivo del paquete como su implementación, son fundamentales para lograr correctos resultados.

Palabras clave: Minado Web, analizador, heurística, paquete, evaluación.

Abstract

Web mining has emerged as a solution to the costly and traditional mining process. This new procedure is adopted by many companies and institutions with the aim of automating and optimizing the process of obtaining information. What tools are available on the market and how do they work? What are the differences with other existing tools? What are the differences with other existing ones? The objective of this work is to carry out an in-depth analysis of the important programming packages within web scraping, with the subsequent objective of making a quantitative evaluation of them.

In response to all these questions, a study process is initiated. In this process, an evaluation method is evaluation method based, on the one hand, on the quality of the information extracted from each tool, on the other hand, on the use of resources for each extraction. The quantity and boilerplate content of the information extracted, as well as CPU/RAM usage or execution time will be variables tested.

The results obtained show that those algorithms whose heuristics are more sophisticated tend to present better solutions. It has also been found that both the language and the parser used are of lesser importance with respect to the heuristics used. It has also been found that both the language and the analyzer used are of minor importance with respect to the effectiveness of the extraction, but not in the efficiency. In conclusion, it can be determined that both the objective of the package and its implementation are fundamental to achieve correct results.

Keywords: Web scraping, parser, heuristic, package, evaluation.

Resumen extendido

Cuando cualquier usuario entra en un sitio web, lo que busca es obtener una determinada información lo más rápido y preciso posible. El *web scraping* trata, de precisamente eso, de extraer y recopilar datos de uno o varios sitios web y disponerlos de manera ordenada en una cierta estructura de datos.

Dada su naturaleza, y con el avance y desarrollo de nuevas herramientas, la minería web ha ido cobrando importancia sobre todo en el ámbito de la ciencia de los datos. Tanto multinacionales como instituciones financieras hacen uso de este tipo de software con el objetivo de analizar a la competencia y para mejorar su situación dentro del mercado.

Existen múltiples herramientas desarrolladas para este propósito, ya sea a través del uso de *frameworks* o bibliotecas de programación. El trabajo se centra en la minería web basada en herramientas de programación, donde el objetivo de cada paquete y su heurística tomarán un papel fundamental.

¿Qué herramientas de programación existen y como funcionan? ¿Qué diferencias hay con otras ya existentes? A lo largo del trabajo se realiza un análisis sobre las diferentes bibliotecas de *web scraping* disponibles, con el objetivo de conocer y comparar el funcionamiento de las mismas.

Puesto que se espera que estas herramientas de minado puedan ser una solución fiable y de calidad con respecto a la extracción tradicional, se realiza un proceso de estudio donde la información extraída sea comparada con la original. Los algoritmos a prueba serán evaluados en términos de cantidad y el contenido *boilerplate* de la información extraída, uso de recursos del entorno y tiempo de ejecución empleado.

Como posible metodología con las que poder comparar múltiples textos sin perjudicar la integridad de la herramienta se destaca la comparación basada en n-gramas. Esta división permite minimizar la probabilidad de fallo de manera de la posible repetición de palabras a lo largo del texto, o la posición de las mismas no sea un factor perjudicial. La eficacia en la exclusión de contenido *boilerplate*, la capacidad de captación de contenido principal y la proporción de predicciones correctas del texto extraído, marcan la calidad del texto extraído.

Por otro lado, además del análisis de la calidad del texto extraído de cada herramienta, se realiza un análisis de la optimización de la misma. El uso de recursos del entorno de ejecución, y el tiempo empleado para ejecutar el algoritmo determinan como de buenas son las herramientas en términos de eficiencia.

Los resultados obtenidos reflejan que la heurística de ciertas herramientas están muy elaboradas. En términos de calidad y optimización, bibliotecas como **Trafilatura** o **Readability** están muy cerca de la solución tradicional. Ambos algoritmos han sido capaces de detectar el 94 % del contenido principal de 101 documentos en tan solo 4 segundos y con un uso de recursos mínimo.

Sin embargo, no todas las herramientas evaluadas han ido capaces de obtener los mismos resultados. Algunas como **rvest** o **Rcrawler** presentan resultados bastante pobres. Esto refleja que tanto el objetivo de cada paquete, como la implicación en el desarrollo del mismo, son fundamentales para obtener una herramienta software de calidad.

Índice general

Índice general	xi
Índice de figuras	xv
Índice de tablas	xvii
Índice de listados de código fuente	xx
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	5
2.1 En que consiste realmente el web scraping	5
2.1.1 Extracción de los datos	5
2.1.2 Herramientas software disponibles	6
2.1.2.1 XPath	6
2.1.2.2 Selectores CSS	7
2.1.3 Tipologías	7
2.2 Posibilidades prácticas del web scraping	8
2.3 Retos del web scraping	8
2.4 Aspectos ético-legales del web scraping	9
3 Bibliotecas de programación orientadas al web scraping	11
3.1 Búsqueda de bibliotecas destinadas al web scraping	11
3.2 Bibliotecas de Python encontradas durante el proceso de búsqueda	11
3.2.1 inscriptis	12
3.2.1.1 Estructura de la solución	12
3.2.1.2 Reglas de anotación	13

3.2.1.3	Postprocesamiento	14
3.2.2	Beautiful Soup	14
3.2.2.1	Multiplicidad de analizadores	14
3.2.2.2	Sopa de objetos	16
3.2.3	jusText	16
3.2.3.1	Preprocesamiento	16
3.2.3.2	Segmentación	16
3.2.3.3	Clasificación de bloques	17
3.2.3.4	Reclasificación de bloques	18
3.2.3.5	Bloques de cabecera	19
3.2.4	news-please	19
3.2.4.1	Web crawling en news-please	20
3.2.4.2	Web scraping en news-please	20
3.2.5	Libextract	21
3.2.5.1	eatihit como algoritmo de partición	21
3.2.6	html_text	22
3.2.6.1	Normalización de espacio	22
3.2.7	html2text	23
3.2.8	Readability	23
3.2.8.1	Heurística de nodo candidato	23
3.2.9	Trafilatura	24
3.2.9.1	Delimitación de contenido	24
3.2.9.2	Algoritmos de respaldo	25
3.2.9.3	Extracción base	25
3.2.10	Dragnet	26
3.2.10.1	Extracción enfocada al aprendizaje automatico	26
3.2.10.2	Características del modelo	26
3.2.11	Goose3	27
3.2.11.1	Cálculo del mejor nodo	27
3.2.12	Newspaper3k	28
3.2.12.1	Heurística según el idioma	28
3.2.13	BoilerPy3/BoilerpipeR	29
3.2.13.1	Multiplicidad de extractores	29
3.3	Paquetes de R encontrados durante el proceso de búsqueda	29
3.3.1	rvest	30
3.3.1.1	Elementos a nivel de bloque vs elementos en línea	30
3.3.2	Rcrawler	30

3.3.2.1	Arquitectura y proceso de extracción en Rcrawler	31
3.3.3	htm2txt	32
3.3.4	scrapeR	32
3.3.5	RSelenium	32
3.4	Paquetes descartados para el proceso de evaluación	32
4	Selección de variables de análisis y proceso de estudio	35
4.1	Introducción al proceso de evaluación	35
4.1.1	Aspectos generales a considerar	36
4.1.2	Cuerpo principal del sitio web como objetivo de la evaluación	37
4.1.3	Recopilación del conjunto de datos	39
4.2	Análisis de la herramienta de evaluación	40
4.2.1	Aspectos específicos a considerar	41
4.2.2	Heurística basada en n-gramas	41
4.2.2.1	Comparación de texto empleando el método palabra por posición	42
4.2.2.2	Comparación de texto empleando el método palabra por detección	43
4.2.2.3	Comparación de texto empleando la creacion de n-gramas	44
4.2.3	Cálculo y puntuación de n-gramas	45
4.2.4	Introducción a las métricas de evaluación	48
4.2.4.1	Métricas independientes del entorno de ejecución	48
4.2.4.2	Métricas dependientes del entorno de ejecución	51
5	Análisis y comparativa de paquetes	55
5.1	Evaluación individual de paquetes	55
5.2	Comparación de paquetes	79
5.2.1	Comparación de resultados entre paquetes que usan expresiones XPath	80
5.2.2	Comparación de resultados entre paquetes según el analizador	81
5.2.3	Comparación de paquetes según la heurística empleada	85
5.2.3.1	Comparación entre Readability y Goose3	85
5.2.3.2	Comparación entre Trafilatura, Readability y jusText	86
5.2.3.3	Comparación entre Beautiful Soup y rvest	87
5.2.4	Comparación genérica de paquetes	89
6	Conclusiones y futuras líneas de trabajo	91
	Bibliografía	93

Apéndice A	Funcionamiento básico de un web scraper	97
A.1	Fase de búsqueda	97
A.2	Fase de extracción	98
A.3	Fase de transformación	99
Apéndice B	Analizadores empleados en los paquetes de web scraping	101
B.1	lxml	101
B.1.1	lxml.html	102
B.1.2	soupparser	102
B.2	html5lib	103
B.3	html.parser	103
B.4	XML	103
B.5	xml2	104
Apéndice C	Métricas descartadas del proceso de evaluación	105
C.1	<i>Perfect accuracy</i>	105
C.2	<i>Specificity</i>	106

Índice de figuras

2.1	Fases del web scraping	6
2.2	Fases del web crawling	7
3.1	inscriptis - Postprocesamiento html	15
3.2	Arbol de objetos	16
3.3	jusText - Reclasificación de bloques	18
3.4	news-please - Proceso de scraping y crawling	19
3.5	rvest - Elementos a nivel de bloque	30
3.6	Rcrawler - Arquitectura y componentes principales de Rcrawler	31
4.1	Estructura genérica del proceso de evaluación	36
4.2	Artículo web tradicional	37
4.3	Contenido <i>boilerplate</i> de un artículo web tradicional	38
4.4	Diagrama de Venn: Texto base vs texto extraído	40
4.5	Estructura específica del proceso de evaluación	41
4.6	Comparación de textos empleando el método palabra por posición	42
4.7	Comparación de textos empleando el método palabra por posición	42
4.8	Comparación de textos empleando el método palabra por detección	43
4.9	Comparación de textos empleando el método palabra por detección	43
4.10	Comparación de textos empleando el método palabra por detección	43
4.11	Creación de n-gramas	44
4.12	Comparación de textos empleando n-gramas	44
4.13	Comparación de textos empleando n-gramas mejorados	47
4.14	Matriz de confusión de las instancias calculadas	48
4.15	Jerarquía de métricas	49
4.16	Estructura del entorno de ejecución	52
5.1	Gráfica - Resultados de la evaluación de inscriptis	56
5.2	Gráfica - Comparación de resultados entre las tres variantes de Beautiful Soup	58
5.3	Gráfica - Comparación de resultados entre las tres variantes de Beautiful Soup 2	58

5.4	Gráfica - Resultados de la evaluación de jusText	60
5.5	Gráfica - Resultados de la evaluación de html_text	61
5.6	Gráfica - Resultados de la evaluación de html2text	62
5.7	Gráfica - Resultados de la evaluación de Readability	64
5.8	Gráfica - Resultados de la evaluación de Trafilatura	65
5.9	Gráfica - Resultados de la evaluación de Trafilatura (<i>precision</i>)	66
5.10	Gráfica - Resultados de la evaluación de Trafilatura (<i>recall</i>)	67
5.11	Gráfica - Comparación de resultados entre los tres algoritmos de Trafilatura	68
5.12	Gráfica - Comparación de resultados entre los tres algoritmos de Trafilatura 2	68
5.13	Gráfica - Resultados de la evaluación de Goose3	70
5.14	Gráfica - Resultados de la evaluación de Boilerpy	71
5.15	Gráfica - Comparación de resultados entre Boilerpy & BoilerpipeR	73
5.16	Gráfica - Comparación de resultados entre Boilerpy & BoilerpipeR 2	73
5.17	Gráfica - Resultados de la evaluación de rvest	75
5.18	Gráfica - Resultados de la evaluación de Rcrawler	76
5.19	Gráfica - Resultados de la evaluación de htm2txt	78
5.20	Gráfica - Resultados de la evaluación de XPath	79
5.21	Gráfica - Comparación de resultados entre paquetes que usan expresiones XPath	80
5.22	Comparación de la eficacia en los paquetes que usan expresiones XPath	81
5.23	Gráfica - Comparación de resultados entre paquetes que usan <i>html.parser</i> como analizador	82
5.24	Gráfica - Comparación de resultados entre paquetes que usan <i>lxml</i> como analizador	83
5.25	Gráfica - Comparación de resultados entre paquetes que usan <i>XML</i> o <i>xml2</i> como analizador	84
5.26	Comparación de la eficacia en los paquetes según el analizador empleado	85
5.27	Figura - Comparación de resultados entre Readability & Goose3	86
5.28	Figura - Comparación de resultados entre Trafilatura & Readability & jusText	87
5.29	Figura - Comparación de resultados entre Beautiful Soup & rvest	88
5.30	Comparación del tiempo de ejecución en los paquetes de <i>web scraping</i>	89
5.31	Comparación de la eficacia en los paquetes de <i>web scraping</i>	90
5.32	Comparación de la eficiencia en los paquetes de <i>web scraping</i>	90
B.1	Estructura basica de un analizador	101
C.1	Jerarquía de métricas descartadas	105

Índice de tablas

3.1	Emparejamientos biblioteca-analizador	12
3.2	inscriptis - Perfil de anotaciones	13
3.3	Beautiful Soup - Analizadores disponibles	15
3.4	jusText - Clasificación de bloques long & medium size	18
3.5	Emparejamientos paquete-analizador	29
4.1	Ejemplos de instancias positivas y negativas	46
4.2	Ejemplos de True Negative	46
4.3	Características del entorno de ejecución	52
5.1	Tabla - Resultados de la evaluación de inscriptis	56
5.2	Tabla - Resultados de la evaluación de Beautiful Soup (<i>html.parse</i>)	57
5.3	Tabla - Resultados de la evaluación de Beautiful Soup (<i>lxml</i>)	57
5.4	Tabla - Resultados de la evaluación de Beautiful Soup (<i>html5lib</i>)	57
5.5	Tabla - Resultados de la evaluación de jusText	59
5.6	Tabla - Resultados de la evaluación de html_text	61
5.7	Tabla - Resultados de la evaluación de html2text	62
5.8	Tabla - Resultados de la evaluación de Readability	63
5.9	Tabla - Comparación de resultados entre Readability & Goose3	63
5.10	Tabla - Resultados de la evaluación de Trafilatura	65
5.11	Tabla - Resultados de la evaluación de Trafilatura (<i>precision</i>)	66
5.12	Tabla - Resultados de la evaluación de Trafilatura (<i>recall</i>)	67
5.13	Tabla - Comparación de resultados entre los tres algoritmos de Trafilatura	67
5.14	Tabla - Resultados de la evaluación de Goose3	69
5.15	Tabla - Resultados de la evaluación de Boilerpy	71
5.16	Tabla - Resultados de la evaluación de BoilerpipeR	72
5.17	Tabla - Comparación de resultados entre Boilerpy & BoilerpipeR	72
5.18	Tabla - Resultados de la evaluación de rvest	74
5.19	Tabla - Resultados de la evaluación de Rcrawler	76

5.20	Tabla - Resultados de la evaluación de htm2txt	77
5.21	Tabla - Resultados de la evaluación de XPath	79
5.22	Tabla - Comparación de resultados entre paquetes que usan expresiones XPath	80
5.23	Tabla - Comparación de resultados entre paquetes que emplean <i>html.parser</i> como analizador	82
5.24	Tabla - Comparación de resultados entre paquetes que emplean <i>lxml</i> como analizador . .	83
5.25	Tabla - Comparación de resultados entre paquetes que emplean <i>XML</i> o <i>xml2</i> como analizador	84
5.26	Tabla - Comparación de resultados entre Readability & Goose3	86
5.27	Tabla - Comparación de resultados entre Trafilatura & Readability & jusText	87
5.28	Tabla - Comparación de resultados entre Beautiful Soup & rvest	88
5.29	Tabla - Comparación de resultados entre algoritmos de <i>web scraping</i>	89

Índice de listados de código fuente

3.1	inscriptis - Uso de reglas de anotación	14
3.2	jusText - Algoritmo de clasificación	17
3.3	news-please - Extracción de contenido relevante	20
3.4	Libextract - Funcionamiento de eatiht	21
3.5	Readability - Selección del nodo candidato	23
3.6	Trafilatura - Readability como algoritmo de respaldo	25
3.7	Trafilatura - jusText como algoritmo de respaldo	25
3.8	Goose3 - Cálculo del mejor nodo 1	27
3.9	Goose3 - Cálculo del mejor nodo 2	27
3.10	Newspaper3k - Heurística en el árabe	28
4.1	Proceso de tokenización	44
4.2	Creación de n-gramas a partir de los tokens resultantes	45
4.3	Cálculo de true positives	46
4.4	Cálculo de false positives	46
4.5	Cálculo de false negatives	46
4.6	Cálculo de puntuaciones	47
4.7	Cálculo de la métrica <i>precision</i>	49
4.8	Cálculo de la métrica <i>recall</i>	50
4.9	Cálculo de la métrica <i>accuracy</i>	50
4.10	Función de ejecución de Boilerpy	51
4.11	Función de ejecución de BoilerpipeR	51
5.1	Función de ejecución de inscriptis	55
5.2	Función de ejecución de BeautifulSoup	57
5.3	Función de ejecución de jusText	59
5.4	Función de ejecución de html_text	60
5.5	Función de ejecución de html2text	62
5.6	Función de ejecución de Readability	63
5.7	Función de ejecución de Trafilatura	64
5.8	Función de ejecución de Trafilatura (<i>precision</i>)	65
5.9	Función de ejecución de Trafilatura (<i>recall</i>)	66
5.10	Función de ejecución de Goose3	69
5.11	Función de ejecución de Boilerpy	70
5.12	Ejecución de BoilerpipeR desde Python	71
5.13	Función de ejecución de BoilerpipeR	72
5.14	Ejecución de rvest desde Python	74
5.15	Función de ejecución de rvest	74
5.16	Función de ejecución de Rcrawler	75

5.17 Ejecución de Rcrawler desde Python	76
5.18 Función de ejecución de htm2txt	77
5.19 Ejecución de htm2txt desde Python	77
5.20 Función de ejecución de XPath	78
A.1 Solicitud del documento <i>robots.txt</i>	97
A.2 Acceso y descarga del archivo HTML	98
A.3 Extracción de datos de interés del documento	99
A.4 Transformación de datos en un data frame	99
C.1 Cálculo de la métrica <i>perfect accuracy</i>	106
C.2 Cálculo de la métrica <i>specificity</i>	106

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 obtenida de un sitio web [1].

Las formas en las que se extraen datos de internet pueden ser muy diversas. 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 el modo en la que se almacenan. Durante los siguientes apartados se realizará una especificación más 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 empleo 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 propia colección de páginas.

Además de ser un ámbito interesante, el minado web también es un área muy requerida. Algunos de los 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 usada 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 un correcto minado web.

¿Qué solución es la más rentable para el minado web? ¿Cuál 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 y eficaz.

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 su uso 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 y objetivos.**

En la introducción se especifica tanto el contexto como la motivación para desarrollo del 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: Bibliotecas de programación orientadas al web scraping.**

Se procede con 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 explica 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 procede 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 6: Conclusiones y futuras líneas trabajos.**

Una vez realizado todo el proceso heurístico se analizan los resultados obtenidos. Además, se determinan una serie de aspectos a reforzar en futuros trabajos

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. Las diferentes herramientas empleadas 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 JSON, 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.

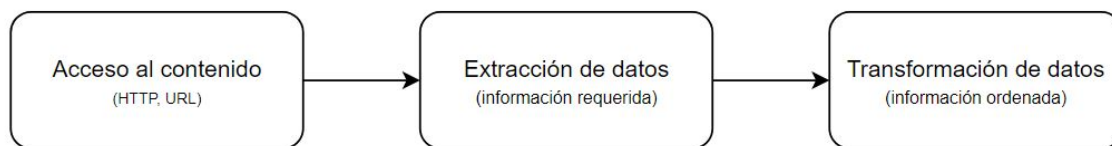


Figura 2.1: Fases del web scraping

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 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 usado 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*.

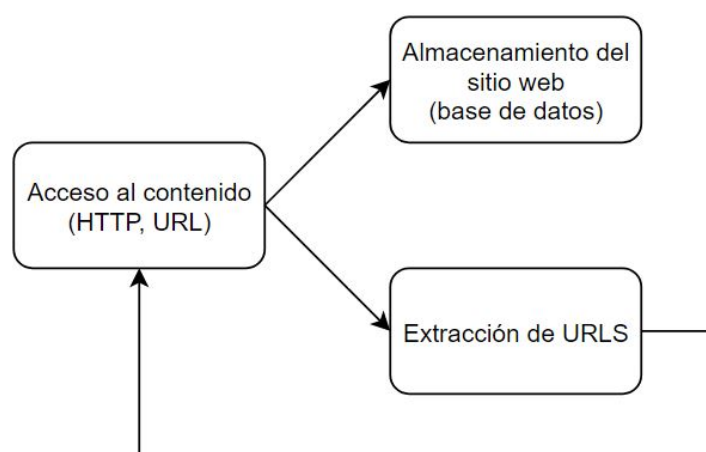


Figura 2.2: Fases del web crawling

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 interesantes, 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. En la mayoría de casos 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, 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 sea analizada, puede que no sea correcta. La gramática y la ortografía en ocasiones suponen un problema en la fase de análisis, ya que la 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. Además, 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 el equilibrio perfecto 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 manera 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 al 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 este especificado en el documento, si esto sucede debemos determinar un número de solicitudes razonable, por ejemplo, una solicitud por segundo.

¹Archivo 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 ilustración 3.1 un ejemplo de la salida obtenida.

3.2.2 BeautifulSoup

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.

The screenshot shows a web page with the following structure:

- heading**: Politics [[edit](#)]
- subheading**: Coat of arms [[edit](#)]
- text**: Blazon: Argent, a city gate gules with three [merlons](#), within which a capricorn rampant sable, langued and viriled of the
- subheading**: Administrative divisions [[edit](#)]
- subheading**: Government [[edit](#)]
- text**: The City Council (Stadtrat) constitutes the [executive](#) government of the City of Chur and operates as a [collegiate authori](#)
- text**: As of 2017, Chur's City Council is made up of one representative of the FDP ([FDP.The Liberals](#)), who is also the mayor), or
- text**: Stadtrat of Chur[20]
- table**:

City Councillor	Party	Head of Department (Leitung, since) of	elected since
(Stadtrat/ Stadträtin)			
Urs Marti [CC 1]	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

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: BeautifulSoup - 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 múltiples 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

El proceso de extracción que se emplea en este algoritmo es sencillo. En primer lugar, el documento ya sea HTML o XML se convierte por 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.

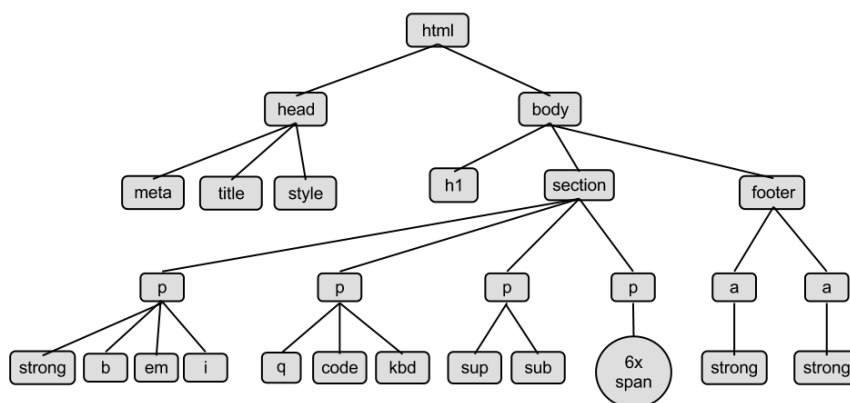


Figura 3.2: Árbol de objetos

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. Por eso **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 $\langle a \rangle$, 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.

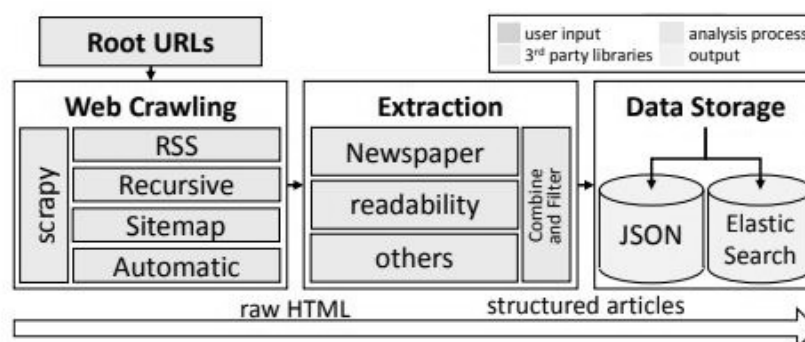


Figura 3.4: news-please - Proceso de scraping y crawling

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 **Goose3** 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 **Goose3** 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**. Este se encarga de hacer el mismo trabajo que 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.

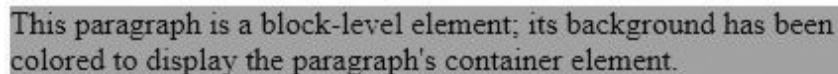
En el apéndice A se puede comprobar el funcionamiento del paquete durante la extracción de parte de un documento HTML. Ademas, se observa que el paquete actúa como *wrapper* entre los paquetes *xml2* y *httr*, facilitando así la descarga y manipulación del fichero HTML.

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].

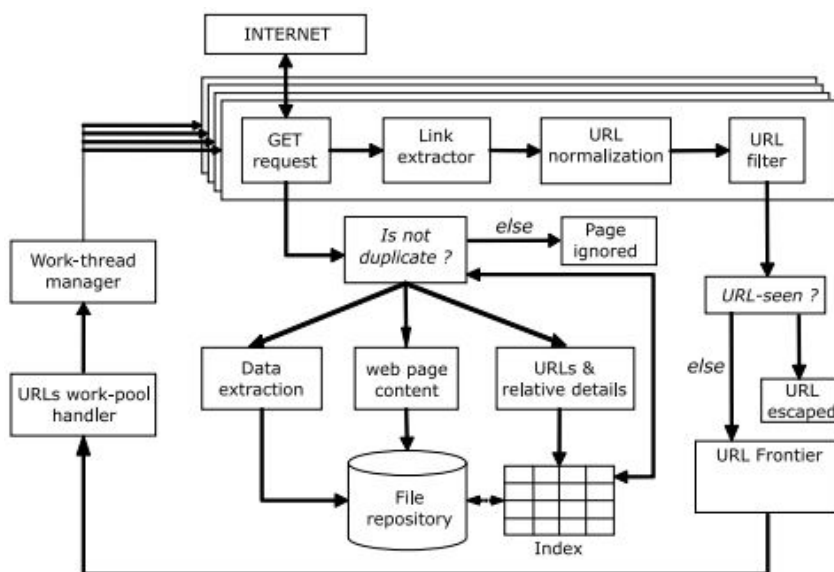


Figura 3.6: Rcrawler - Arquitectura y componentes principales de Rcrawler

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 el 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 usando expresiones regulares. También ofrece las funciones *gettext()* 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 analizado 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 uso 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 propia. 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

Tras la sinopsis de herramientas de *web scraping* de código abierto, se pretende realizar una introducción al proceso de evaluación de las mismas. El capítulo se dividirá en tres secciones claramente diferenciadas las cuales se especifican con más detalle a continuación:

1. En primer lugar se realiza una introducción sobre el proceso de evaluación a realizar. ¿Cuáles son los aspectos más importantes en la evaluación de herramientas de minería web? ¿Es posible realizar una valoración objetiva entre bibliotecas de diferentes lenguajes de programación?
2. La segunda sección trata sobre el proceso de comparación, así como una introducción al código desarrollado para este. El proceso de tokenización y la creación/comparación de n-gramas o bloques son algunos de los aspectos a destacar. ¿Cuál es el tamaño óptimo de cada n-grama o bloque? ¿Cómo se determina si la extracción ha sido exitosa?
3. Por último, se especifican las variables de comparación y los test preparados de cada algoritmo. Se determina el proceso seguido para calcular la precisión, velocidad de extracción, uso de memoria y demás características de cada algoritmo.

Cabe destacar que la evaluación será conjunta, las herramientas de los diferentes lenguajes de programación serán sometidas a los mismos test con el fin de ver aquellas bibliotecas o paquetes más involucrados en el proceso de minado web.

A lo largo del capítulo será posible comprobar en los diferentes fragmentos de código como se ha realizado la integración de las diversas herramientas, y sobre como es posible construir una valoración objetiva para cualquier herramienta de minado web desarrollada sobre cualquier lenguaje de programación convencional.

4.1 Introducción al proceso de evaluación

Cuando cualquier usuario entra en un sitio web lo que busca es obtener la información requerida lo más rápido y preciso posible. La calidad del texto extraído es prioritaria, para ello el uso de heurísticas o la eliminación de contenido *boilerplate* es crucial en cualquier algoritmo de *web scraping*.

Muchos de los algoritmos descartados para el proceso de evaluación como **scrapeR**, o ni si quiera mencionados en la sinopsis de paquetes como **selectr** [36], emplean únicamente expresiones regulares para la extracción de contenido. Esto provoca que la extracción de texto no sea 'limpia', puesto que siempre van a existir resultados donde se extraiga contenido no deseado.

El objetivo de este proyecto concierne la evaluación del proceso heurístico de las diferentes herramientas de minado web. La inclusión de algoritmos que emplean únicamente expresiones *XPath* o selectores CSS no tiene sentido para la herramienta desarrollada, pues cualquier analizador de los ya mencionados en el apéndice B podría realizar el mismo trabajo.

4.1.1 Aspectos generales a considerar

La gran mayoría de los *web scrapers* son capaces de extraer fragmentos de información de un sitio web, ya sea el precio de un determinado producto en venta, el autor de un ensayo o el resultado de un partido de fútbol. Pero además, se espera que estas herramientas de minado puedan ser una solución fiable y de calidad con respecto a la extracción tradicional, ¿es posible medir de algún modo la calidad del texto extraído de estas herramientas?

La respuesta es si, y para ello, los algoritmos se centrarán en la extracción de artículos de prensa en su mayoría, con el objetivo de que estos sean comparados con un texto base. Se define como texto base aquellos fragmentos de texto principal de cualquier sitio web que el usuario visualizaría al entrar. Se muestra en la figura 4.1 la estructura general del proceso de evaluación.

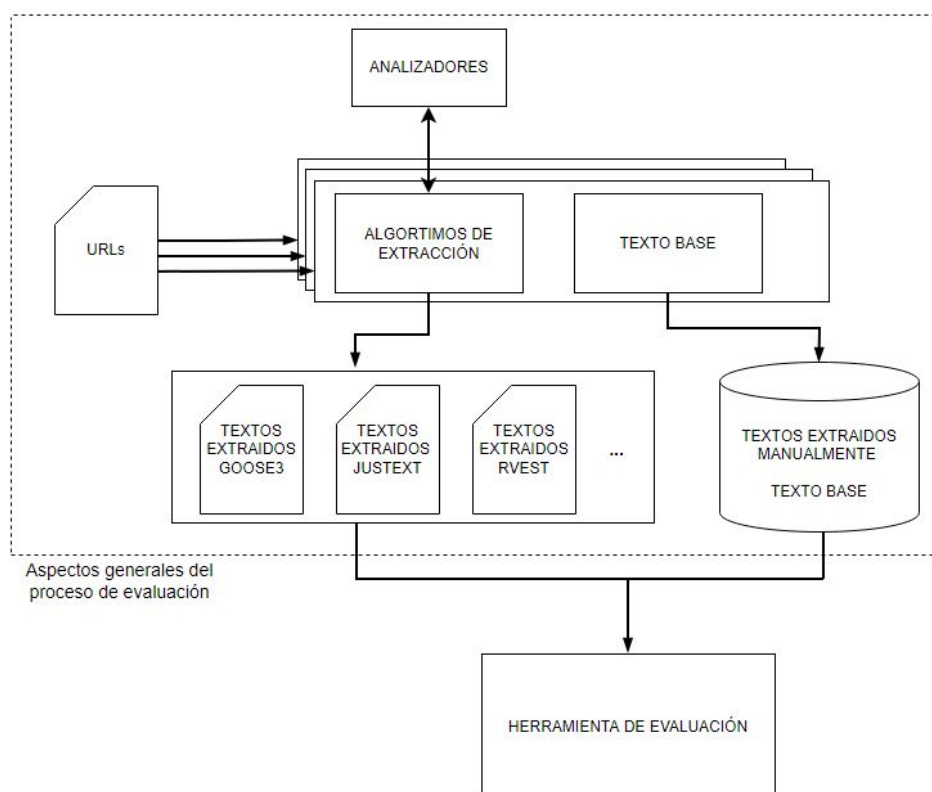


Figura 4.1: Estructura genérica del proceso de evaluación

Tanto los textos obtenidos por los algoritmos de *web scraping*, como los textos producidos por la extracción manual, se almacenan de forma ordenada en archivos JSON. Esto se hace con el objetivo de que su acceso sea sencillo para cualquier lenguaje de programación.

4.1.2 Cuerpo principal del sitio web como objetivo de la evaluación

Como ya se ha mencionado anteriormente, una de las características más importantes de un *web scraper* es la capacidad de extraer de texto de calidad. Por lo tanto, comparar el texto extraído de diferentes algoritmos será el cometido de la herramienta de evaluación. Ahora la principal pregunta es, ¿qué fragmentos de texto son considerados como principales?

Imaginemos un artículo web de noticias tradicional o una entrada de blog, algo parecido a lo mostrado en la figura 4.2, del que se pretende extraer información de valor. En la imagen se reflejan varias secciones claramente diferenciadas, anuncios de contenidos relacionados, información de autor e incluso elementos de navegación.

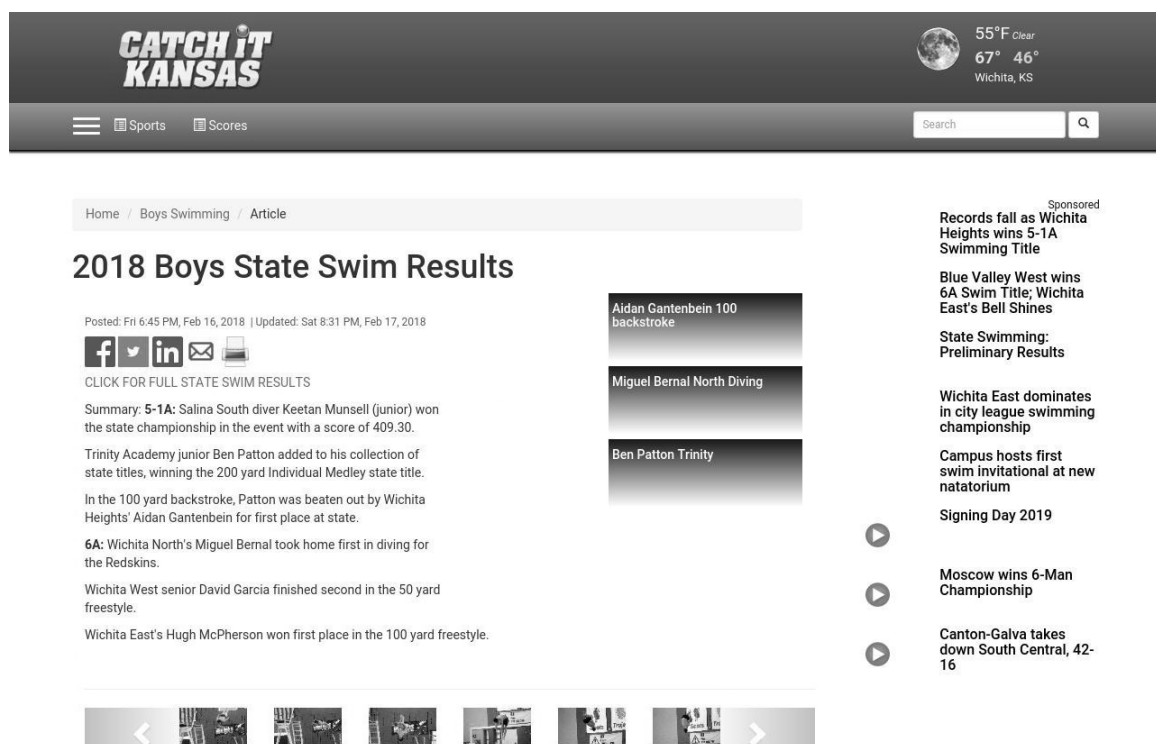


Figura 4.2: Artículo web tradicional

La tarea de extracción puede parecer sencilla, pero dependiendo del sitio web puede ser sorprendentemente complicada y llena de matices [37]. La selección del cuerpo principal del sitio web será el objetivo de cada algoritmo. Para conseguir un cuerpo de artículo no solo será necesario saber dónde empieza o termina este, también se deberán conocer las partes a excluir.

Con el fin de la que la evaluación sea lo más justa posible, se debe definir qué compone el cuerpo del artículo, y estar seguros de que todas las herramientas siguen los mismos objetivos. El principio subyacente es que el cuerpo del artículo debe ser un texto limpio, sin campos adicionales, elementos de navegación o anuncios. En la siguiente lista se determina el conjunto de elementos o secciones de un sitio web no contemplados en la evaluación:

- Campos de información sobre autor/es, fechas de publicación, palabras clave o títulos de imágenes y vídeos.
- Botones para compartir y sugerencias para compartir un artículo, artículos relacionados, enlaces 'leer a continuación', 'recomendado para usted'...

- Comentarios e interfaz de usuario relacionada con los mismos. Elementos de navegación propios del sitio web.
- Elementos de control alrededor de las imágenes que producen texto innecesario, número de imágenes en una galería, botones superpuestos a una imagen/vídeo...
- Código JavaScript. Aunque esta no sea una sección específica de un sitio web, muchas herramientas extraen estos fragmentos de código pensando que pertenece como parte del contenido principal del propio artículo.

Se muestra a continuación la lista de elementos que serán extraídos por las diferentes herramientas como parte del contenido principal, además del propio contenido principal.

- Título principal de artículo.
- Enlaces para leer con más detalle, a la fuente u otros contenidos directamente relacionados. Estos enlaces pueden requerir un conocimiento mucho más profundo y complejo del contenido principal del sitio web.
- Avisos de copyright.

En definitiva, todo fragmento que pueda ser extraído y comparado con el original y del que se puedan obtener conclusiones firmes, será incluido como parte del contenido principal. Véase la figura 4.3 en la que se determinan las secciones descartadas y definidas como contenido *boilerplate* o 'basura'.

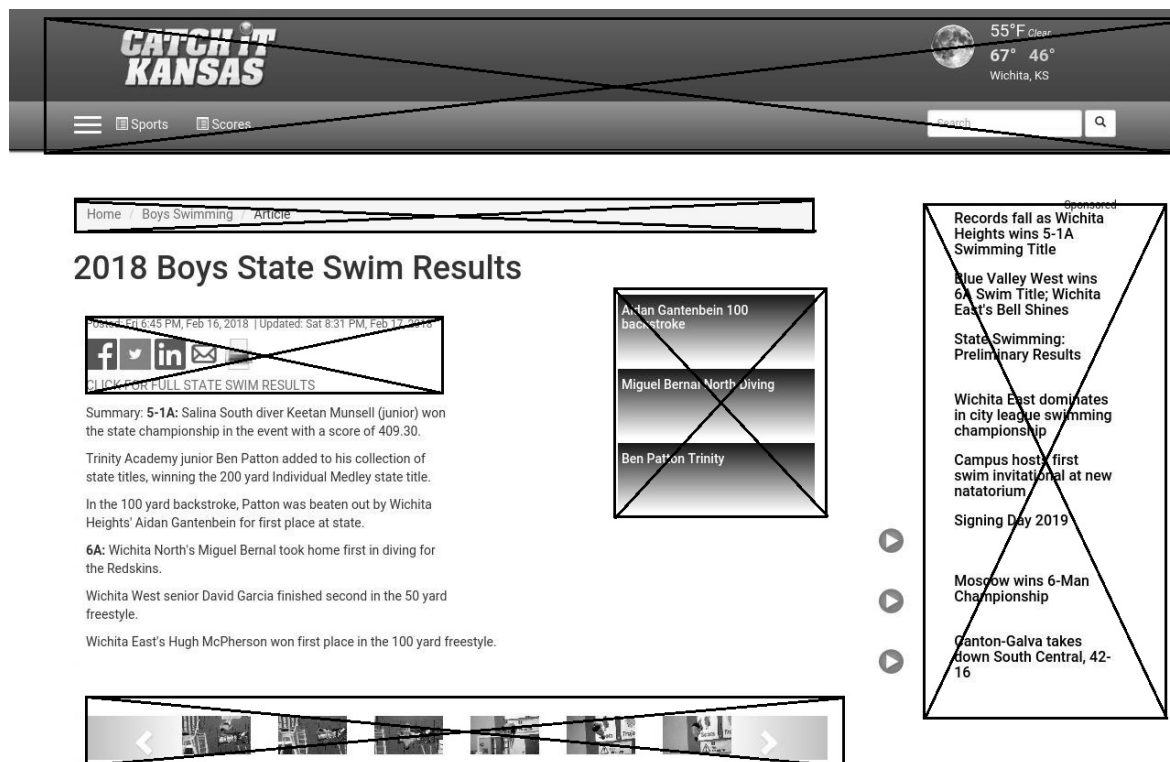


Figura 4.3: Contenido *boilerplate* de un artículo web tradicional

4.1.3 Recopilación del conjunto de datos

Una vez definido en que consiste el cuerpo principal de cada artículo, el siguiente paso consiste en la recolección del conjunto de datos de prueba para la herramienta. Se pretende que este conjunto sea diverso, representativo e imparcial. Es por ello que en el *dataset* no solo se compone artículos de noticias relevantes, también hay un conjunto muy variado de artículos no informativos como entradas de blog, comunicados de prensa...

El proceso de recolección del conjunto de datos consiste en tomar una muestra aleatoria del millón de sitios web escritos en inglés. Para cada sitio, se comprueba si contiene algún artículo, en cuyo caso, se añaden dos de ellos al azar. En una primera instancia, el conjunto que compone el *dataset* suponía cerca de 200 sitios web listos para analizar.

Antes de comenzar con el proceso de evaluación se deben asegurar una serie de conceptos relacionados con posibles errores de acceso, análisis o extracción de la información de algunas de las páginas incluidas:

1. Se debe asegurar que las páginas no cambian durante el análisis. Minimizar las posibilidades de obtener un artículo actualizado durante la extracción.
2. Se deben excluir aquellas páginas que las herramientas no sean capaces de descargar.
3. Se deben descartar aquellas paginas cuyo contenido no sea explícitamente escrito en inglés.

Tras tener en cuenta todas estas consideraciones, el grupo inicial de 200 páginas se redujo a 101 casos listos para el análisis. Aún realizada la reducción, el *dataset* es lo suficientemente amplio como para sacar conclusiones claras de los algoritmos a analizar.

Por último, como ya se ha mencionado anteriormente, tanto la información extraída de forma manual, como aquella extraída por los algoritmos de minado web, es almacenada en archivos JSON. Esto hace mucho más cómodo el análisis, pues el acceso a cada texto es mucho más sencillo. Se muestra a continuación un ejemplo de como se almacenan los diferentes textos extraídos por **BeautifulSoup**.

```
{
  "0000test": {
    "texto": "Nadal keeps Spain alive against Russia in Davis Cup Finals..."
  },
  ...
  "0100test": {
    "texto": "CNN - Breaking News, Latest News and Videos CNN | 11/25/2021..."
  }
}
```

De la misma forma se almacenan los textos extraídos de forma manual. En este caso, se guarda información adicional del sitio web. Como se puede observar, el orden de almacenamiento es clave, pues de él depende que el análisis sea correcto.

```
{
  "0000test": {
    "texto": "MADRID -- Rafael Nadal kept Spain's hopes alive, then Marcel...",
    "url": "https://www.sportsnet.ca/tennis/..."
  },
  ...
}
```

```
...
"0100test": {
  "texto": "Groups of thieves target two high-end stores in California...",
  "url": "http://lite.cnn.com/en/article/..."
}
```

Esta forma de almacenamiento es aplicada a todos los algoritmos de minado, no solo a aquellos desarrollados en Python, las herramientas codificadas en R también lo aplican. Esto permite incluir en la evaluación a cualquier algoritmo de minado web desarrollado sobre un lenguaje de programación que permita trabajar con documentos JSON.

4.2 Análisis de la herramienta de evaluación

Una vez se dispone del *dataset* al completo, el objetivo es comparar cada uno de los textos obtenidos por las herramientas de extracción con el texto base. La exposición de las diferentes soluciones aportadas será clave para comprender el resultado final.

Uno de los aspectos fundamentales de la herramienta de evaluación es su heurística. Se debe pensar en un algoritmo que sea capaz de comparar diferentes tipos de textos minimizando la probabilidad de fallo. A lo largo de la sección se mostrará como el proceso no es sencillo, pues muchos algoritmos de extracción presentan diferentes soluciones a un mismo problema.

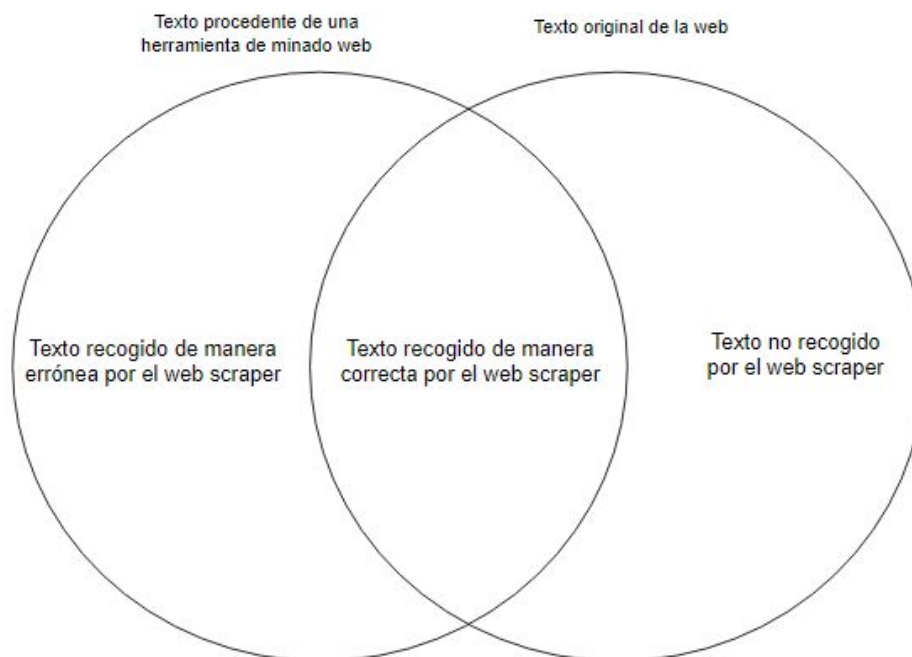


Figura 4.4: Diagrama de Venn: Texto base vs texto extraído

La principal preocupación subyace por la forma en la que los diferentes algoritmos consideran como parte de contenido principal de un sitio web. En algunas ocasiones es posible configurar la manera de extracción de los mismos, pero en otra no. Es por ello que se debe pensar en un método lo suficientemente justo como para comparar múltiples textos de diversas procedencias que pueden ser ligeramente diferentes.

4.2.1 Aspectos específicos a considerar

En la figura 4.1 se definía de forma genérica como se gestionaba la información recogida por los diferentes algoritmos de *web scraping*. Veamos ahora como se trata y se compara dicha información con la extraída manualmente, con el objetivo de conocer aquellos algoritmos que más se acercan a un resultado real.

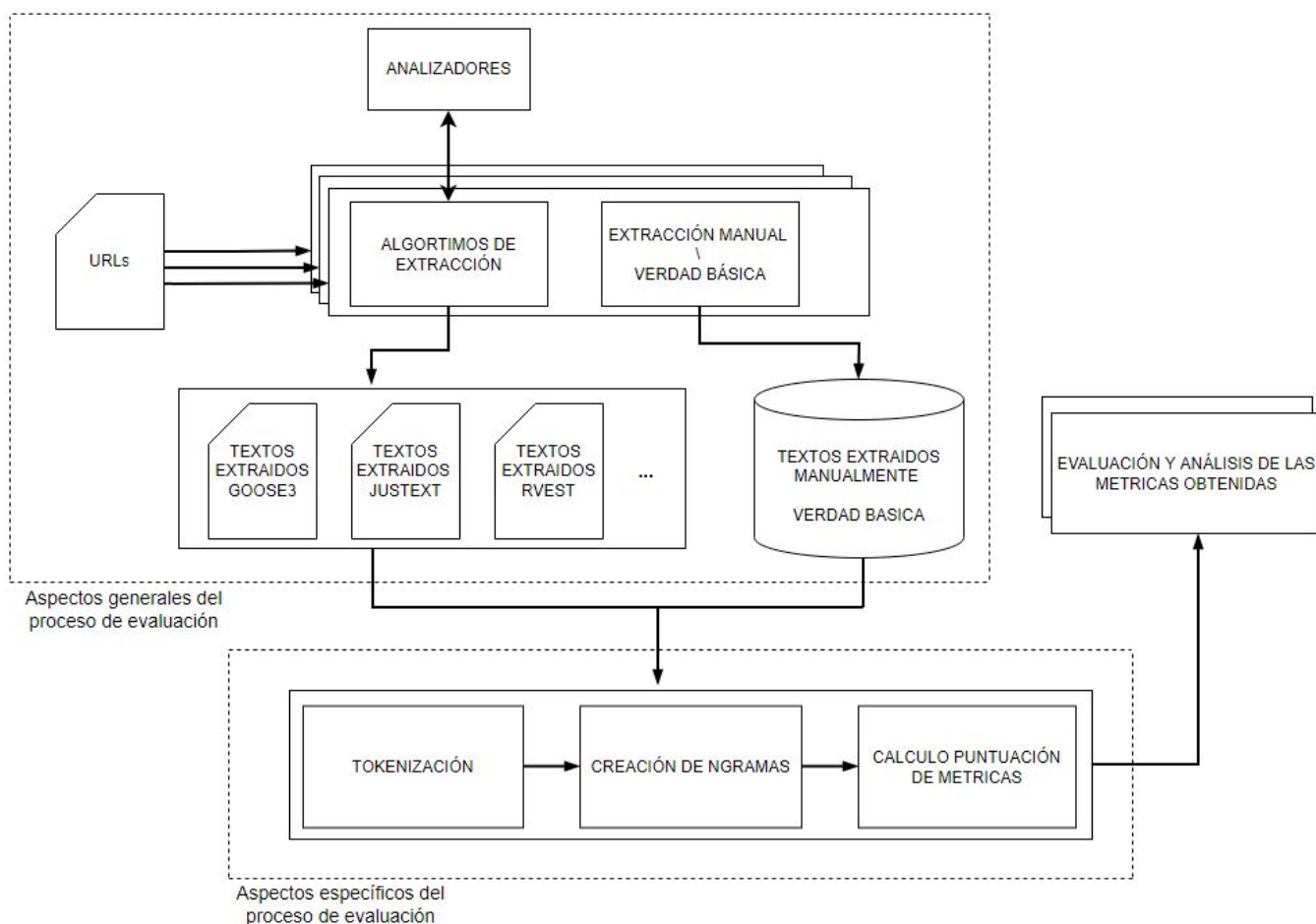


Figura 4.5: Estructura específica del proceso de evaluación

Se puede observar en la figura 4.5 la solución adoptada por la herramienta de evaluación. Tras la obtención de texto base y texto extraído, se pretende realizar un proceso de tokenización y creación de n-gramas, clave a la hora de realizar el posterior cálculo de métricas. En cuanto a la evaluación y análisis de métricas obtenidas, se abordará en el próximo capítulo.

4.2.2 Heurística basada en n-gramas

A lo largo de la sección se aborda el problema de comparar diferentes fragmentos de texto. Se presentan diferentes soluciones, una de ellas podría ser la comparación de palabras en la misma posición, otra podría ser la búsqueda de palabras pertenecientes a ambos textos. Veamos porque la división en n-gramas es un buen método para este propósito.

4.2.2.1 Comparación de texto empleando el método palabra por posición

Imaginemos que se disponen dos textos a comparar como los siguientes. El primero de ellos se considera como texto base, pues ha sido obtenido de forma manual del propio sitio web. El segundo surge como resultado de la ejecución de cualquiera de las herramientas de *web scraping* expuestas en el capítulo anterior.

```
MADRID -- Top-ranked Rafael Nadal has arrived in Madrid to
lead Spain in the new-look Davis Cup Finals.\n\n"It's a
new competition and we must be focused,\" Nadal said Sunday.
```

```
MADRID \u2014 Top-ranked Rafael Nadal has arrived in Madrid to
lead Spain in the new-look Davis Cup Finals.\"It\u2019s a
new competition and we must be focused,\" Nadal said Sunday.
```

Veamos el funcionamiento de los diferentes métodos de comparación pensados. En primer lugar, se aplica la comparación palabra por posición, donde se prepara un fragmento de código que recorra ambos textos al mismo tiempo y determine si las palabras dispuestas en la misma posición son idénticas.

MADRID	--	Top-ranked	Rafael	Nadal	has	arrived	in	Madrid	to
lead	Spain	in	the	new-look	Davis	Cup	Finals.\n\n"	a	
new	competition	and	we	must	be	focused,\"	Nadal	said	Sunday.
MADRID	\u2014	Top-ranked	Rafael	Nadal	has	arrived	in	Madrid	to
lead	Spain	in	the	new-look	Davis	Cup	Finals.\"It\u2019s	a	
new	competition	and	we	must	be	focused,\"	Nadal	said	Sunday.

Figura 4.6: Comparación de textos empleando el método palabra por posición

Como se muestra en la figura 4.6 la comparación no ha sido tan desastrosa como se esperaba. Los únicos errores han sido producidos por espacios en blanco y caracteres Unicode que la herramienta de minado web no ha sido capaz de detectar.

El problema con esta solución viene cuando la disposición de los mismos no es exacta, o incluso cuando ambos fragmentos no tienen el mismo tamaño. Imaginemos que la herramienta de minado no ha sido capaz de detectar la palabra **MADRID** como parte del contenido principal. En este caso, ninguna palabra coincidiría con la original pues las posiciones no están dispuestas del mismo modo.

MADRID	--	Top-ranked	Rafael	Nadal	has	arrived	in	Madrid	to
lead	Spain	in	the	new-look	Davis	Cup	Finals.\n\n"	a	
new	competition	and	we	must	be	focused,\"	Nadal	said	Sunday.
\u2014	Top-ranked	Rafael	Nadal	has	arrived	in	Madrid	to	lead
Spain	in	the	new-look	Davis	Cup	Finals.\"It\u2019s	a	new	competition
and	we	must	be	focused,\"	Nadal	said	Sunday.		

Figura 4.7: Comparación de textos empleando el método palabra por posición

Esta forma de comparación no funcionaría, y menos en artículos donde los fragmentos de textos son amplios y las posibilidades de fallo aumentan. Además, el *dataset* es de gran volumen por lo que la posibilidad de que existan fragmentos de texto con disposiciones totalmente distintas a la original tiene una alta probabilidad.

4.2.2.2 Comparación de texto empleando el método palabra por detección

Es necesario encontrar entonces una nueva heurística que minimice la probabilidad de fallo, donde la posición de cada palabra no sea un aspecto a tener en cuenta. Una solución podría darse con el recorrido de uno de los fragmentos de texto, donde se busquen palabras coincidentes con el otro. Se selecciona una palabra pivote, por ejemplo **Spain**, y se recorre el segundo texto hasta encontrar la palabra indicada.

MADRID	--	Top-ranked	Rafael	Nadal	has	arrived	in	Madrid	to
lead	Spain	in	the	new-look	Davis	Cup	Finals.\n\n"It's	a	
new	competition	and	we	must	be	focused,\"	Nadal	said	Sunday.

MADRID	\u2014	Top-ranked	Rafael	Nadal	has	arrived	in	Madrid	to
lead	Spain	in	the	new-look	Davis	Cup	Finals.\"It\u2019s	a	
new	competition	and	we	must	be	focused,\"	Nadal	said	Sunday.

Figura 4.8: Comparación de textos empleando el método palabra por detección

En realidad, si se realiza un análisis profundo sobre esta solución, surgen nuevas problemáticas. ¿Qué ocurriría si apareciesen dos palabras idénticas sobre el mismo texto? Imaginemos que durante el recorrido del primer texto, se debe analizar una palabra pivote repetida varias veces a lo largo del segundo texto. La heurística anterior no sería correcta, veamos un ejemplo.

Comienza el algoritmo seleccionando pivotes y realizando la heurística correspondiente para cada uno, primero con **MADRID**, luego con **'--'**, y así sucesivamente hasta llegar a **Nadal** que es el que nos interesa en este ejemplo. Se recorre el segundo fragmento buscando dicha palabra y se encuentran dos apariciones de la misma. Para ambas apariciones se calcula y se suma una puntuación determinada.

MADRID	--	Top-ranked	Rafael	Nadal	has	arrived	in	Madrid	to
lead	Spain	in	the	new-look	Davis	Cup	Finals.\n\n"It's	a	
new	competition	and	we	must	be	focused,\"	Nadal	said	Sunday.

MADRID	\u2014	Top-ranked	Rafael	Nadal	has	arrived	in	Madrid	to
lead	Spain	in	the	new-look	Davis	Cup	Finals.\"It\u2019s	a	
new	competition	and	we	must	be	focused,\"	Nadal	said	Sunday.

Figura 4.9: Comparación de textos empleando el método palabra por detección

El algoritmo sigue realizando el bucle y nuevamente emplea la palabra **Nadal** como pivote, pero en este caso en su segunda aparición. Se volvería a efectuar un calculo y la posterior suma de su puntuación. La implementación de esta heurística forzaría un resultado irreal, pues este error se acrecentaría sobre *datasets* extensos que contengan fragmentos de texto prolongados, como es el caso.

MADRID	--	Top-ranked	Rafael	Nadal	has	arrived	in	Madrid	to
lead	Spain	in	the	new-look	Davis	Cup	Finals.\n\n"It's	a	
new	competition	and	we	must	be	focused,\"	Nadal	said	Sunday.

MADRID	\u2014	Top-ranked	Rafael	Nadal	has	arrived	in	Madrid	to
lead	Spain	in	the	new-look	Davis	Cup	Finals.\"It\u2019s	a	
new	competition	and	we	must	be	focused,\"	Nadal	said	Sunday.

Figura 4.10: Comparación de textos empleando el método palabra por detección

4.2.2.3 Comparación de texto empleando la creación de n-gramas

Se debe pensar entonces en una nueva heurística que nuevamente minimice la probabilidad de fallo, la cual tenga en cuenta que la posición de cada palabra debe ser irrelevante y que cuide la posible repetición de palabras. Se utilizan bloques como estructuras de datos, veamos como funciona.

Los bloques o n-gramas se consideran conjuntos de secuencias alfanuméricas, en las que se descartan los espacios en blanco y se separan los signos de puntuación [38]. Esto funciona bien en la mayoría de los casos, ya que normalmente el texto a excluir se encuentra en bloques separados por nuevas líneas o espacios en blanco. Siguiendo con el ejemplo anterior, la división de texto en n-gramas donde $n = 3$ sería:

MADRID	--	Top-ranked	Rafael	Nadal	has	arrived	in	Madrid
to	lead	Spain	in	the	new-look	Davis	Cup	Finals.
It's	a	new	competition	and	we	must	be	focused,\"
Nadal	said	Sunday.						

Figura 4.11: Creación de n-gramas

El objetivo ahora es separar ambos textos en n-gramas de tamaño n y realizar la comparación entre ellos. La posición ya no es relevante pues la comparación no se va a ejecutar en torno a la posición de cada n-grama. Además, la repetición de n-gramas no es preocupante pues existe una baja probabilidad de que esto ocurra a lo largo del texto.

MADRID	Top-ranked	Rafael	Nadal	has	arrived	in	Madrid	
to	lead	Spain	in	the	new-look	Davis	Cup	Finals.
It's	a	new	competition	and	we	must	be	focused,\"
Nadal	said	Sunday.						

MADRID	Top-ranked	Rafael	Nadal	has	arrived	in	Madrid	
to	lead	Spain	in	the	new-look	Davis	Cup	Finals.
It's	a	new	competition	and	we	must	be	focused,\"
Nadal	said	Sunday.						

Figura 4.12: Comparación de textos empleando n-gramas

El algoritmo procede del mismo modo que antes, se selecciona un n-grama pivote y se buscan n-gramas coincidentes en el texto extraído. Tras el recorrido completo del texto se determina una cierta puntuación. En la figura 4.12 se muestra el resultado de ejecución del algoritmo.

Si observamos de nuevo la figura anterior, los n-gramas en séptima y primera posición del texto base no coincide con los n-gramas del texto extraído, a pesar de que la mayor parte de las palabras sí que son coincidentes. Es posible realizar una mejora dividiendo el texto en el mayor número de n-gramas posibles. Este aumento de divisiones se efectúa con el objetivo de recuperar la mayor cantidad de palabras mejorando así la precisión de la herramienta desarrollada.

Listado 4.1: Proceso de tokenización

```
def tokenizar(text):
    '''se separa el texto en tokens/palabras y lo devuelve como una lista'''
    token = re.compile(r'\w+', re.UNICODE| re.MULTILINE| re.IGNORECASE| re.DOTALL)
    return token.findall(text or '')
```


En los fragmentos de código 4.1 y 4.2 se muestra el proceso de tokenización y creación de n-gramas usado en la herramienta de evaluación. Para el desarrollo de la herramienta se ha decidido emplear n-gramas donde $n = 4$ con el objetivo de aumentar la precisión del resultado dado.

Listado 4.2: Creación de n-gramas a partir de los tokens resultantes

```
def crear_ngramas(text, n):
    '''division del texto en ngramas donde cada ngrama contendra n tokens,
    el resultado se devolvera como una tupla de ngramas'''
    tokens = tokenizar(text)
    ngramas = []
    for i in range(0, max(1, len(tokens) - n + 1)):
        #ngrama = tupla de 4 elementos, desde la posicion i hasta i + n
        ngrama = tuple(tokens[i: i + n])
        if ngrama:
            ngramas.append(ngrama)
    return ngramas
```

Aplicando esta metodología al ejemplo anterior se obtienen una cantidad bastante mayor de n-gramas. Este mismo proceso se realiza tanto en el texto base, como en el texto extraído, para poder realizar el cálculo posterior de la puntuación.

```
(MADRID -- Top-ranked Rafael), (-- Top-ranked Rafael Nadal),
(Top-ranked Rafael Nadal has), (Rafael Nadal has arrived),
(Nadal has arrived in), (has arrived in Madrid), (arrived in Madrid to),
(in Madrid to lead), (Madrid to lead Spain), (to lead Spain in),
(lead Spain in the), (Spain in the new-look), (in the new-look Davis),
(the new-look Davis Cup), (new-look Davis Cup Finals.), (Davis Cup Finals. It's),
(Cup Finals. It's a), (Finals. It's a new), (It's a new competition),
(a new competition and), (new competition and we), (competition and we must),
(and we must be), (we must be focused,), (must be focused, Nadal),
(be focused, Nadal said), (focused, Nadal said Sunday.)
```

4.2.3 Cálculo y puntuación de n-gramas

Una vez que ambos fragmentos de texto han sido convertidos en n-gramas, se debe realizar el cálculo del texto extraído para poder conocer la calidad del mismo. Para comprender como se ha desarrollado dicho cálculo, debemos recordar la figura 4.4, donde el diagrama expuesto representa los tres posibles tipos de puntuaciones.

Para realizar el cálculo y puntuación de n-gramas, se desarrolla un clasificador binario, encargado de realizar una división de instancias positivas y negativas:

- Positivo: La instancia se clasifica como miembro de la clase que el clasificador está tratando de identificar. Por ejemplo, un clasificador que busque fotos de gatos clasificará las fotos con gatos como positivas cuando sean correctas.
- Negativo: La instancia se clasifica como no perteneciente a la clase que se intenta identificar. Por ejemplo, un clasificador que busque fotos de gatos debería clasificar las fotos con perros como negativas.

Las bases de las métricas independientes del entorno de ejecución especificadas en la sección 4.2.4.1, provienen de los conceptos de **True Positive**, **False Positive** y **False Negative**. La tabla 4.1 ilustra estos conceptos, donde se considera el valor '1' como una predicción positiva.

Predicción	Valor actual	Tipo	Explicación
1	1	True Positive	La predicción coincidió con el valor resultante
1	0	False Positive	La predicción no coincidió con el valor resultante
0	1	False Negative	La predicción no coincidió con el valor resultante

Tabla 4.1: Ejemplos de instancias positivas y negativas

Una instancia que no se tiene en cuenta en el clasificador desarrollado para el cálculo de métricas es **True Negative**. Esta instancia determinaría la capacidad de la herramienta para clasificar contenido que no se ha predicho. Se muestran algunos ejemplos en la tabla 4.2.

Predicción	Valor actual	Tipo	Explicación
0	0	True Negative	La predicción coincidió con el valor resultante
No Gato	No Gato	True Negative	Se predijo que no había gato y no era un gato
(Nadal Rafael has)	(Nadal Rafael has)	True Negative	Se predijo n-grama erróneo y era erróneo

Tabla 4.2: Ejemplos de True Negative

Los n-gramas que coinciden en el texto base y en el texto extraído se conoce como **True Positives**. Se muestra en el fragmento de código 4.3 como se realiza dicho cálculo.

Listado 4.3: Cálculo de true positives

```
def calcular_tp(contador_base, contador_extraido):
    '''Numero de ngramas que coinciden en el texto base y en el extraido'''
    return min(contador_base, contador_extraido)
```

Por otro lado, los n-gramas que aparecen en el texto extraído, pero no en el texto base se conoce como **False Positives**. Se muestra en el fragmento de código 4.4 como se realiza dicho cálculo.

Listado 4.4: Cálculo de false positives

```
def calcular_fp(contador_base, contador_extraido):
    '''Numero de ngramas que aparecen en el texto extraido pero no en el base'''
    return max(0, contador_extraido - contador_base)
```

Por ultimo, los n-gramas que aparecen en el texto base pero no en el texto extraído se conoce como **False Negatives**. Se muestra en el fragmento de código 4.5 como se realiza dicho cálculo.

Listado 4.5: Cálculo de false negatives

```
def calcular_fn(contador_base, contador_extraido):
    '''Numero de ngramas que aparecen en el texto base pero no en el extraido'''
    return max(0, contador_base - contador_extraido)
```

Una vez conocidas estas tres posibles puntuaciones, lo único que se debe hacer es recorrer el texto base y texto extraído, haciendo una comparación de n-gramas. El código fragmento de código mostrado en 4.6 se encarga de retornar una tupla con las puntuaciones calculadas.

Listado 4.6: Cálculo de puntuaciones

```
def calculo_puntuacion(texto_base, texto_extraido, num_ngramas = 4):
    '''recorre ngramas de texto base/extraido y devuelve las metricas obtenidas'''
    ngramas_texto_base = get_diccionario_ngramas(texto_base, num_ngramas)
    ngramas_texto_extraido = get_diccionario_ngramas(texto_extraido, num_ngramas)

    tp, fp, fn = 0
    for key in (set(ngramas_texto_base) | set(ngramas_texto_extraido)):
        contador_base = ngramas_texto_base.get(key, 0)
        contador_extraido = ngramas_texto_extraido.get(key, 0)
        tp += calcular_tp(contador_base, contador_extraido)
        fp += calcular_fp(contador_base, contador_extraido)
        fn += calcular_fn(contador_base, contador_extraido)

    tp_fp_fn = [tp, fp, fn]
    sumatorio = sum(tp_fp_fn)

    if sumatorio > 0:
        tp_fp_fn = [metrica / sumatorio for metrica in tp_fp_fn]

    return tuple(tp_fp_fn)
```

Con el objetivo de esclarecer como funciona el cálculo de la puntuación, se emplea el ejemplo anterior como sujeto de pruebas. Se muestra a continuación el texto base y extraído convertidos en n-gramas donde $n = 4$.

MADRID	--	Top-ranked	Rafael	--	Top-ranked	Rafael	Nadal	Top-ranked	Rafael	Nadal	has
Rafael	Nadal	has	arrived	Nadal	has	arrived	in	has	arrived	in	Madrid
arrived	in	Madrid	to	in	Madrid	to	lead	Madrid	to	lead	Spain
to	lead	Spain	in	lead	Spain	in	the	Spain	in	the	new-look
in	the	new-look	Davis	the	new-look	Davis	Cup	new-look	Davis	Cup	Finals.
Davis	Cup	Finals.	It's	Cup	Finals.	It's	a	Finals.	It's	a	new
It's	a	new	competition	a	new	competition	and	new	competition	and	we
competition	and	we	must	and	we	must	be	we	must	be	focused,\"
must	be	focused,\"	Nadal	be	focused,\"	Nadal	said	focused,\"	Nadal	said	Sunday.

MADRID	\u2014	Top-ranked	Rafael	\u2014	Top-ranked	Rafael	Nadal	Top-ranked	Rafael	Nadal	has
Rafael	Nadal	has	arrived	Nadal	has	arrived	in	has	arrived	in	Madrid
arrived	in	Madrid	to	in	Madrid	to	lead	Madrid	to	lead	Spain
to	lead	Spain	in	lead	Spain	in	the	Spain	in	the	new-look
in	the	new-look	Davis	the	new-look	Davis	Cup	new-look	Davis	Cup	Finals.
Davis	Cup	Finals.	\u201cIt\u2019s	Cup	Finals.	\u201cIt\u2019s	a	Finals.	\u201cIt\u2019s	a	new
\u201cIt\u2019s	a	new	competition	a	new	competition	and	new	competition	and	we
competition	and	we	must	and	we	must	be	we	must	be	focused,\"
must	be	focused,\"	Nadal	be	focused,\"	Nadal	said	focused,\"	Nadal	said	Sunday.

Figura 4.13: Comparación de textos empleando n-gramas mejorados

Una vez aplicada la heurística habitual del algoritmo, se procede con el cálculo de la puntuación. Se realiza el recorrido de n-gramas de ambos textos y se va realizando el cálculo. Se muestran las puntuaciones calculadas la matriz de confusión 4.14.

		Actual (True) Values	
		Positive	Negative
Predicted Values	Positive	TP = 21.0	FP = 6.0
	Negative	FN = 6.0	TN = N/A

Figura 4.14: Matriz de confusión de las instancias calculadas

Una vez realizado el cálculo de las diferentes instancias, se procede con las métricas. Algunos términos básicos son **Precision**, **Recall** y **F1-Score**. Estos términos están relacionados con lo bien que funciona un clasificador, en lugar de limitarse a observar la precisión general.

4.2.4 Introducción a las métricas de evaluación

Después de realizar todo el proceso heurístico y el posterior cálculo de puntuaciones, se debe medir como de buenos son los diferentes algoritmos de extracción. Se propone efectuar una división de métricas, por un lado, se determinarán métricas que dependan del entorno en el que se ejecute el programa, por otro lado, se definirán otras métricas no dependientes de dicho entorno.

Es conveniente revisar el apéndice C, donde se especifican métricas adicionales que finalmente han sido descartadas. Su descarte, es debido a su poca implicación en la evaluación o por el reemplazo de otras métricas más desarrolladas.

4.2.4.1 Métricas independientes del entorno de ejecución

En esta sección, se determinan las diferentes métricas que definirán la calidad de extracción de cada algoritmo. Se definen como métricas independientes del entorno, porque no dependen del medio de ejecución, ni del lenguaje de programación usado para dicho procedimiento.

En la figura 4.15 se muestra la jerarquía existente entre el cálculo de instancias y métricas. A primera vista, es una red un poco desordenada. Las métricas forman una jerarquía que comienza con las negativas/positivas, y que llega hasta la puntuación F1 para unir las todas. Estas métricas son dependientes de las instancias previamente calculadas.

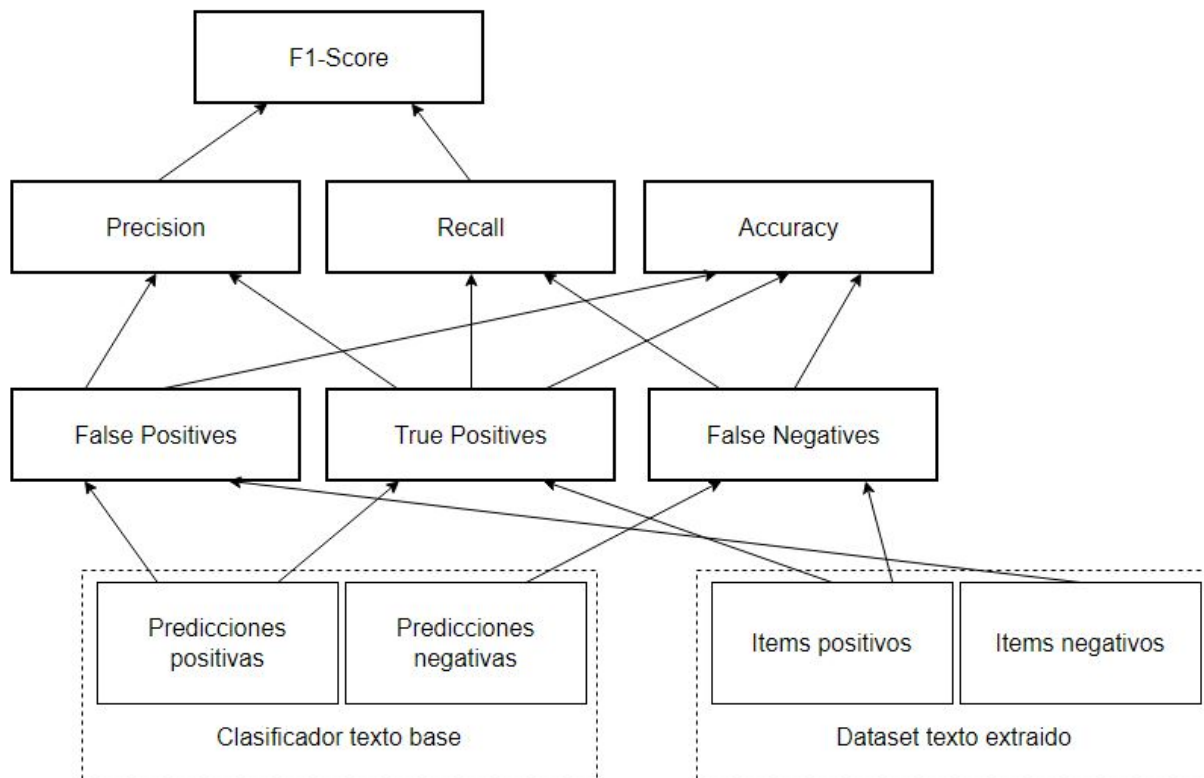


Figura 4.15: Jerarquía de métricas

La primera métrica se conoce como *precision*, y mide el número de predicciones positivas realizadas que son correctas. En otras palabras, mide la eficacia de los sistemas para excluir contenido *boilerplate* de un sitio web. A continuación se determina tanto la formula para su cálculo, como el fragmento de código incluido en la herramienta de evaluación.

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} = \frac{N.\ of\ Correctly\ Predicted\ Positive\ Instances}{N.\ of\ Total\ Positives\ Predictions\ you\ Made}$$

Listado 4.7: Cálculo de la métrica *precision*

```

def calcular_precision(tp, fp, fn):
    '''mide la eficacia de como los algoritmos excluyen contenido boilerplate'''
    if fp == 0 and fn == 0:
        return 1
    if tp == 0 and fp == 0:
        return 0
    return tp / (tp + fp)
  
```

Otra de las métricas obligadas a calcular es lo que se conoce como *recall*. Esta característica mide el número de casos positivos que el clasificador predijo correctamente, sobre todos los casos positivos del *dataset*. A veces también se denomina *sensitivity*. En términos generales, mide la eficacia de los sistemas para captar las partes deseadas del cuerpo del artículo. En el fragmento de código 4.8 se muestra como se realiza el cálculo.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} = \frac{\text{N. of Correctly Predicted Positive Instances}}{\text{N. of Total Positives Instances in the Dataset}}$$

Listado 4.8: Cálculo de la métrica *recall*

```
def calcular_recall(tp, fp, fn):
    '''mide la eficacia de como el algoritmo capta contenido principal'''
    if fp == 0 and fn == 0:
        return 1
    if tp == 0 and fn == 0:
        return 0
    return tp / (tp + fn)
```

Tanto la *precision* como *recall* son métricas importantes de analizar, puesto que miden como de bien diferencian el contenido importante del contenido *boilerplate* los diferentes algoritmos de *web scraping*. Estas métricas definen la manera en la que los algoritmos siguen lo especificado en la sección 4.1.2.

Combinando ambas métricas es posible determinar la calidad en general de la extracción. Esta característica se denomina F1, y funciona bien en los casos en los que los conjuntos de datos están desequilibrados, ya que requiere que tanto las métricas de *precision* como *recall* tengan un valor razonable.

$$F1 = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

Por último, se realiza el cálculo de lo que se conoce como *accuracy*, la cual mide la proporción de predicciones correctas sobre el número total de predicciones. A continuación se muestra tanto la fórmula de cálculo, como su implementación en el entorno de evaluación.

$$\text{Accuracy} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives} + \text{False Positives}} = \frac{\text{N. of Correct Predictions}}{\text{N. of All Predictions}}$$

Listado 4.9: Cálculo de la métrica *accuracy*

```
def calcular_accuracy(tp, fp, fn):
    '''mide la proporción de predicciones correctas de texto extraído'''
    if tp == 0:
        return 0
    if tp == 0 and fp == 0 and fn == 0:
        return 1
    return tp / (tp + fp + fn)
```

4.2.4.2 Métricas dependientes del entorno de ejecución

A lo largo de esta sección, se especificarán aspectos relativos al rendimiento a la hora de utilizar *web scraping* como un posible sustituto de la extracción tradicional. Aspectos como la gestión de recursos no se tienen en cuenta en un principio, pero pueden ser cruciales a la hora de realizar minado web sobre múltiples documentos [39].

Antes de comenzar con el cálculo de métricas, es conveniente asegurar la integridad de la evaluación. Al trabajar con múltiples lenguajes de programación pueden existir conflictos, ya sean relacionados con la complejidad de los algoritmos o con las bibliotecas empleadas.

El primer aspecto a considerar tiene que ver con el diseño de los algoritmos. Aquellos algoritmos escritos en diferentes lenguajes de programación deben presentar similitudes estructurales en su código. En los fragmentos 4.10 y 4.11 se refleja claramente este aspecto.

Listado 4.10: Función de ejecución de Boilerpy

```
from boilerpy3 import extractors

def run_boilerpy():
    '''extraccion de texto empleando boilerpipe'''
    output = {} #diccionario salida

    for path in Path('archivos_html').glob('*.html'):
        with open(path, 'r', encoding = "utf-8") as file:
            html_to_string = file.read()

            extractor = extractors.ArticleExtractor()
            output[path.stem] = {'texto': extractor.get_content(html_to_string)}

    return output
```

Listado 4.11: Función de ejecución de BoilerpipeR

```
library(boilerpipeR)

run_boilerpipeR <- function() {
    ###extraccion de texto empleando boilepipeR###
    output <- c() #diccionario salida

    files <- list.files(path = "./archivos_html", pattern = ".html")
    index <- 0
    for (key in fromJSON(file = "./documento_base.json")) {
        html_to_str <- stri_enc_toutf8(ArticleExtractor(key$url, asText = FALSE))
        lista_texto <- list('test' = list("texto" = html_to_str))

        names(lista_texto) <- str_remove(files[index + 1], ".html")
        output <- c(output, lista_texto)
    }
}
```

En términos referentes a la complejidad, ambos algoritmos presentan una complejidad lineal. El número de ejecuciones de ambos fragmentos de código dependerá del conjunto de elementos del *dataset* de prueba, pero siempre será finito.

Otro aspecto importante corresponde con el uso de herramientas referentes al cálculo de métricas. El empleo de múltiples bibliotecas para abordar este aspecto afectaría negativamente a la integridad de la evaluación. Por ello se propone la utilización de una única interfaz entre lenguajes que unifique el proceso.

Para abordar esta problemática se pretende emplear la biblioteca **rpy2** [40] como interfaz entre ambos lenguajes. Esto permitirá que el compilador de Python ejecute líneas de código escritas en R. De esta forma, una misma biblioteca podrá monitorear algoritmos escritos en ambos lenguajes.

Resueltas ambas cuestiones, se procede con el cálculo. Para este propósito se emplean dos bibliotecas, por un lado, **psutil** [41] encargada de la supervisión del sistema y limitación de recursos, entre otros, y por otro **time** que permite el registro del tiempo de ejecución, pues retorna el número de segundos transcurridos desde un cierto instante.

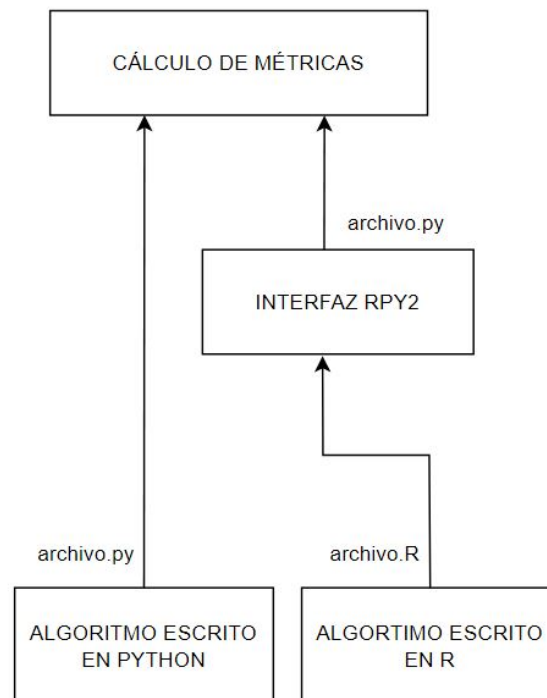


Figura 4.16: Estructura del entorno de ejecución

Determinadas las métricas dependientes del entorno de ejecución, los resultados diferirán según el medio en el que se realicen las pruebas. Se muestra en la tabla 4.3 las características del entorno empleado.

Procesador (CPU)	Memoria RAM	Tarjeta Gráfica (GPU)
AMD Ryzen 7 2700 Eight-Core 3.20 GHz	16,0 GB	NVIDIA GeForce RTX 2060 SUPER

Tabla 4.3: Características del entorno de ejecución

Tiempo de ejecución:

La primera de las mediciones tiene que ver con el tiempo en realizar el proceso completo de minado. En el momento en el que el algoritmo comienza a extraer información se activa el contador, y este para cuando finaliza el proceso.

Cabe destacar que el cálculo de dicho tiempo abarca el proceso de minado al completo, desde el acceso al contenido, hasta la transformación del mismo.

Uso de CPU(%):

El uso de CPU es la cantidad total cantidad de procesador que consume la tarea a realizar. Permite valores por encima del 100 %, lo que indica que el proceso se ejecuta en múltiples hilos en diferentes núcleos de la CPU. La máquina que realizó las pruebas tiene un total de ocho núcleos, por lo que el 800 % significaría que todos los núcleos trabajan al máximo en el proceso.

Para el cálculo del uso de CPU se emplea la función *cpu_percent(interval)*, que devuelve un número que representa la utilización actual de la CPU en todo el sistema en forma de porcentaje. Cuando la variable *interval* es mayor que cero, compara los tiempos de CPU del sistema transcurridos antes y después de dicho intervalo.

Uso de de memoria física o RAM(%):

La memoria real también conocida como memoria física o RAM, es la tercera métrica registrada. Simplemente mide la cantidad de memoria física que el proceso está empleando. Como recordatorio, la máquina en la que se realiza la evaluación tiene un total de 16GB de memoria física.

Para el cálculo del uso de memoria RAM se emplea la función *virtual_memory()*, que devuelve estadísticas sobre el uso de la memoria física total.

Capítulo 5

Análisis y comparativa de paquetes

Tras la exposición de las diferentes variables y métricas que conformaran la evaluación, se realiza el correspondiente análisis y comparación de paquetes. Inicialmente, el análisis será individual de cada paquete, señalando aspectos relativos a otros paquetes si fuese necesario. En la segunda sección, se realizará una comparación conjunta de todos los paquetes analizados.

5.1 Evaluación individual de paquetes

Tal y como se ha especificado en la introducción, se procede con el análisis individual de cada paquete. A lo largo de la misma, se introducirán aspectos relativos a ciertos paquetes, como el tipo de analizador y los resultados obtenidos empleando distintos tipos de herramientas. Para cada paquete se especificará una tabla de resultados y una ilustración de los mismos resultados dispuestos gráficamente.

inscriptis

El primer paquete sometido a análisis será **inscriptis**, el cual debemos instalar e importar en el fragmento de código respectivo. En cuanto a su instalación, es muy sencilla, simplemente se debe ejecutar la siguiente instrucción en la línea de comandos: *\$ pip install inscriptis*.

Listado 5.1: Función de ejecución de inscriptis

```
from inscriptis import get_text

def run_inscriptis():
    '''extraccion de texto empleando inscriptis'''
    output = {} #diccionario salida

    for path in Path('archivos_html').glob('*.html'):
        with open(path, 'r', encoding = "utf-8") as file:
            html_to_string = file.read()

            output[path.stem] = {'texto': get_text(html_to_string)}

    return output
```

En cuanto al código mostrado en 5.1, es muy simple. Se itera sobre los archivos HTML a analizar, y se emplea la función `get_text()` para obtener la información deseada. Dicha información se almacena posteriormente de forma ordenada sobre un diccionario.

Para conservar la información obtenida y no ejecutar el algoritmo múltiples veces, el diccionario obtenido se almacena en un archivo *json* propio del paquete. De esta forma, el archivo *inscriptis.json* conserva todos los fragmentos de texto obtenidos del minado web anterior. Este mismo procedimiento se realiza para el resto de paquetes.

Una vez ejecutado el algoritmo, se realizan todos los cálculos pertinentes y se determinan las puntuaciones obtenidas. En el caso de **inscriptis**, se muestran en la tabla 5.1 los resultados obtenidos.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
inscriptis	0.5414	0.5404	0.9875	0.6985	45.0	0.2	2.1005

Tabla 5.1: Tabla - Resultados de la evaluación de inscriptis

En forma de gráfica, se muestra en la figura 5.1 los cálculos obtenidos en referencia a las métricas independientes del entorno de ejecución. Analizando estas métricas, y sabiendo que *f1* determina la calidad en general de la extracción, podemos concluir que **inscriptis** realiza un equilibrado trabajo en el minado web.

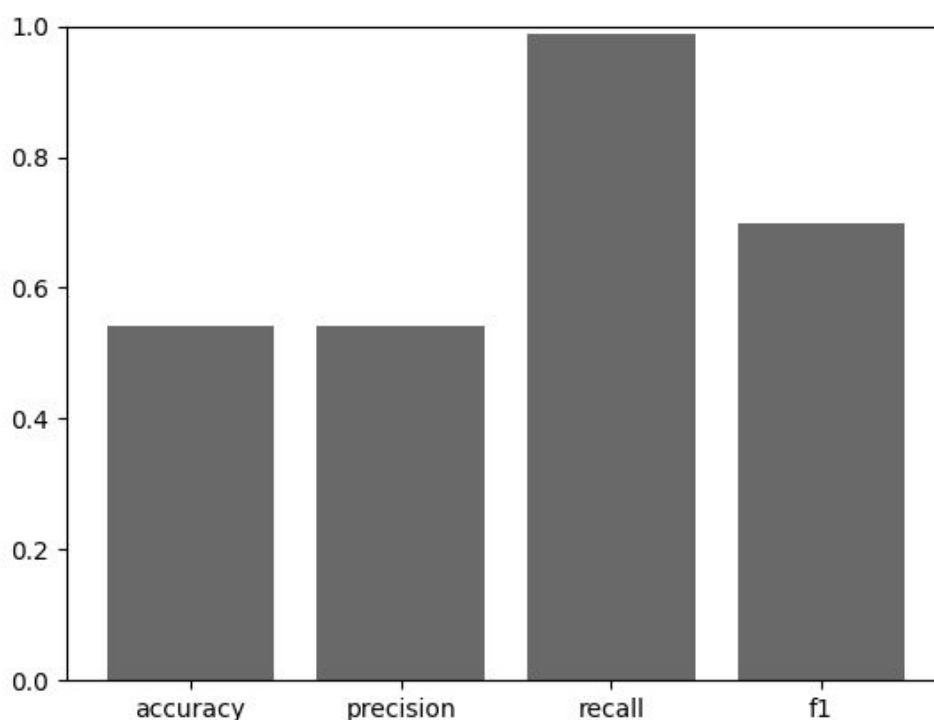


Figura 5.1: Gráfica - Resultados de la evaluación de inscriptis

Por otro lado, respecto a las métricas dependientes del entorno, al solo tener una medición de referencia no podemos decir demasiado. El uso de memoria RAM parece elevado dadas las especificaciones del entorno. Por el contrario, el uso de CPU es mínimo. Algo sorprendente es que **inscriptis** haya sido capaz de analizar 101 documentos HTML en únicamente dos segundos.

Beautiful Soup

El segundo paquete sometido a análisis será **Beautiful Soup**, el cual debemos instalar e importar en el fragmento de código respectivo. En cuanto a su instalación, simplemente se debe ejecutar la siguiente instrucción en la línea de comandos: `$ pip install beautifulsoup4`.

Como se observa en el fragmento de código 5.2, la función es muy similar a la vista con **inscriptis**. Se itera sobre los documentos HTML a analizar y se aplica la función `get_text()` para extraer la información.

Listado 5.2: Función de ejecución de Beautiful Soup

```
from bs4 import BeautifulSoup

def run_beautifulsoup():
    '''extraccion de texto empleando beautifulsoup'''
    output = {} #diccionario salida

    for path in Path('archivos_html').glob('*.html'):
        with open(path, 'r', encoding = "utf-8") as file:
            html_to_string = file.read()

            bs = BeautifulSoup(html_to_string, 'html.parser')
            output[path.stem] = {'texto': bs.get_text(separator=' ', strip=True)}

    return output
```

Debemos recordar que **Beautiful Soup** permite seleccionar entre varios tipos de analizadores, ya sea *html.parse*, *lxml* o *html5lib*. Tras la ejecución de **Beautiful Soup** empleando *html.parse* como analizador, se procede con el cálculo de métricas. Los resultados obtenidos se muestran en la tabla 5.2.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Beau. Soup	0.5165	0.5129	0.9928	0.6764	47.0	4.3	4.0882

Tabla 5.2: Tabla - Resultados de la evaluación de Beautiful Soup (*html.parse*)

Se emplea ahora *lxml* como analizador estándar y se procede con el cálculo de métricas. Los resultados obtenidos se muestran en la tabla 5.3.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Beau. Soup	0.5165	0.5129	0.9928	0.6764	38.1	3.4	3.2183

Tabla 5.3: Tabla - Resultados de la evaluación de Beautiful Soup (*lxml*)

Se emplea ahora *html5lib* como analizador estándar y se procede con el cálculo de métricas. Los resultados obtenidos se muestran en la tabla 5.4.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Beau. Soup	0.1327	0.1315	0.9921	0.2323	39.1	3.4	9.8604

Tabla 5.4: Tabla - Resultados de la evaluación de Beautiful Soup (*html5lib*)

En forma de gráfica, se muestra en 5.2 los cálculos obtenidos en referencia a las métricas independientes del entorno de ejecución. Realizando un análisis, podemos determinar que el uso de *html.parse* o *lxml* no altera la calidad de la información obtenida. Por el contrario cuando se emplea *html5lib* tanto la cantidad de predicciones correctas, como la exclusión de contenido *boilerplate* se reduce drásticamente.

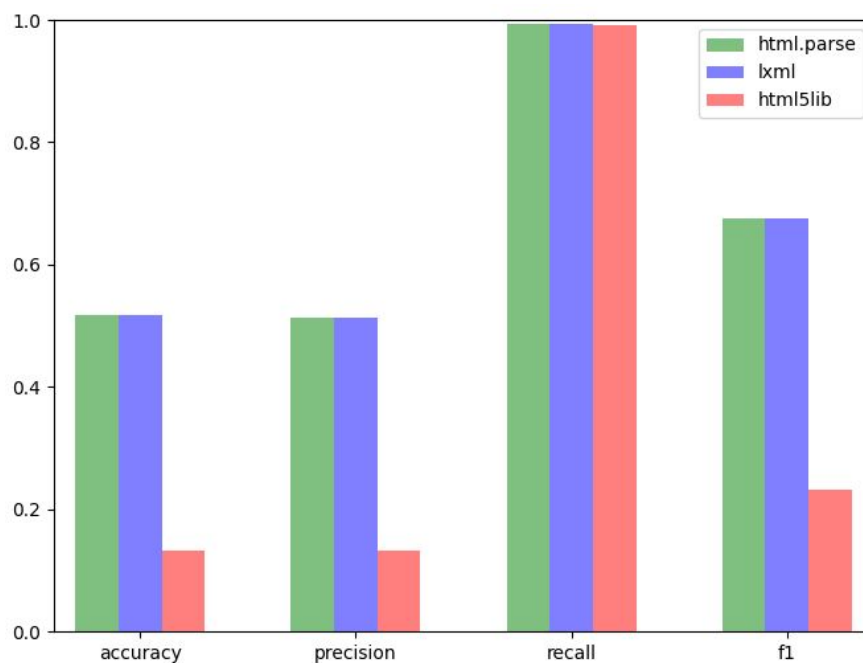


Figura 5.2: Gráfica - Comparación de resultados entre las tres variantes de BeautifulSoup

Por otro lado, respecto a las métricas dependientes del entorno, el uso de memoria RAM, al igual que el uso de CPU, es similar en todos los analizadores. Véase 5.3.

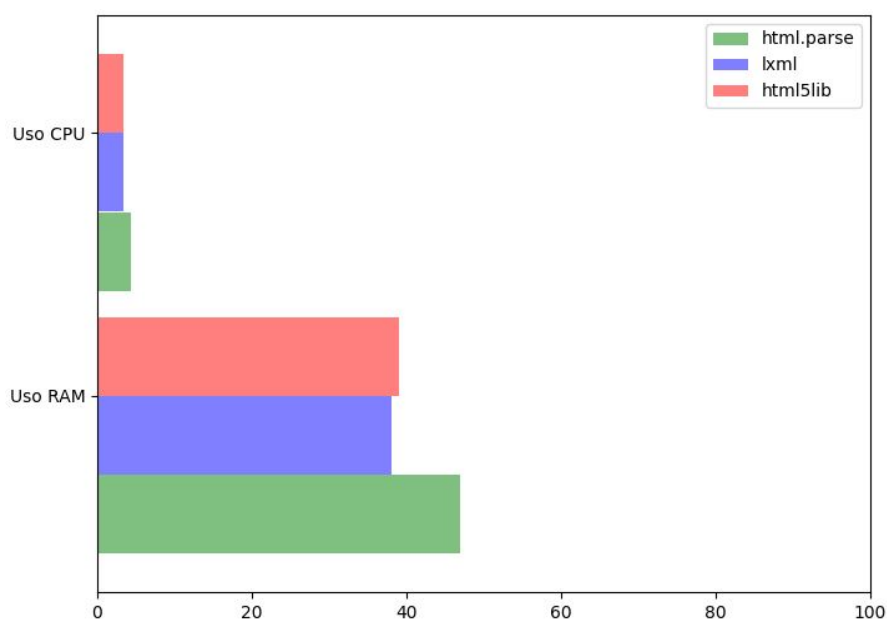


Figura 5.3: Gráfica - Comparación de resultados entre las tres variantes de BeautifulSoup 2

jusText

Continuamos con el análisis de **jusText**, el cual debemos instalar e importar en el fragmento de código respectivo. En cuanto a su instalación, simplemente se debe ejecutar la siguiente instrucción en la línea de comandos: *\$ pip install justext*.

Como se observa en el fragmento de código 5.3, la forma en la que se extrae texto de documentos HTML es muy similar a las vistas anteriormente. La principal diferencia es que en este caso **jusText**, previamente a la extracción, realiza las siguientes acciones:

- Clasificación de bloques.
- Preprocesamiento de bloques de cabecera.
- Reclasificación de bloques.
- Postprocesamiento de bloques de cabecera.

Listado 5.3: Función de ejecución de jusText

```
import justext

def run_jusText():
    '''extraccion de texto empleando justext'''
    output = {} #diccionario salida

    for path in Path('archivos_html').glob('*.html'):
        with open(path, 'r', encoding = "utf-8") as file:
            html_to_string = file.read()

            pargphs = justext.justext(html_to_string, justext.get_stoplist("English"))
            valid = [pargph.text for pargph in pargphs if not pargph.is_boilerplate]
            output[path.stem] = {'texto': ' '.join(valid)}

    return output
```

Como el resto de algoritmos, para conservar la información obtenida y no ejecutar el código múltiples veces, el diccionario obtenido se almacena en un archivo *json* propio del paquete. De esta forma, el *justext.json* conserva todos los fragmentos de texto obtenidos del minado web anterior.

Una vez ejecutado el algoritmo, se realizan todos los cálculos pertinentes y se determinan las puntuaciones obtenidas. En el caso de **jusText**, los resultados obtenidos se muestran en la tabla 5.5.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
jusText	0.7668	0.8649	0.8573	0.8610	45.1	0.5	2.9546

Tabla 5.5: Tabla - Resultados de la evaluación de jusText

En forma de gráfica, se muestra en la figura 5.4 los cálculos obtenidos en referencia a las métricas independientes del entorno de ejecución. Con un simple vistazo, es posible determinar que el algoritmo de **jusText** realiza un buen trabajo en la selección de contenido relevante.

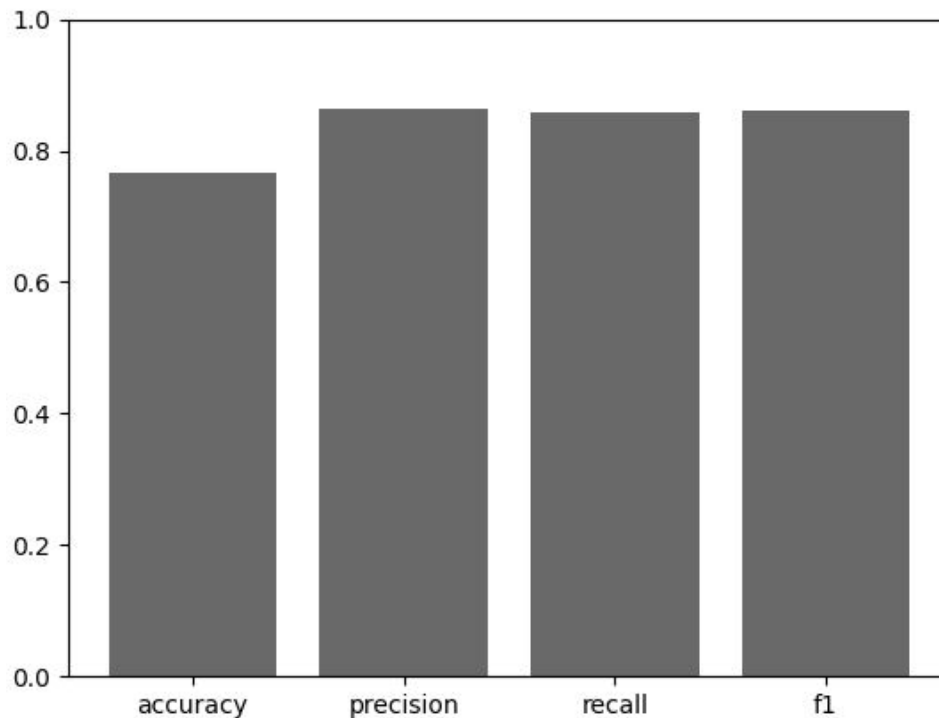


Figura 5.4: Gráfica - Resultados de la evaluación de jusText

Por otro lado, con respecto a las métricas dependientes del entorno de ejecución, se debe remarcar que **jusText** es un paquete de minado web muy bien optimizado. A pesar de realizar todo el trabajo previo a la extracción, mantiene un buen tiempo de ejecución y escaso uso de la CPU.

html_text

El siguiente paquete sometido a análisis será **html_text**, el cual debemos instalar e importar en el fragmento de código respectivo. En cuanto a su instalación, simplemente se debe ejecutar la siguiente instrucción en la línea de comandos: *\$ pip install html-text*.

Listado 5.4: Función de ejecución de html_text

```
import html_text

def run_html_text():
    '''extraccion de texto empleando html_text'''
    output = {} #diccionario salida

    for path in Path('archivos_html').glob('*.html'):
        with open(path, 'r', encoding = "utf-8") as file:
            html_to_string = file.read()

            output[path.stem] = {'texto': html_text.extract_text(html_to_string)}

    return output
```


En el fragmento de código 5.4 se muestra como **html_text** extrae texto de los diferentes documentos HTML. La función `extract_text()` emplea expresiones *XPath* y normalizaciones de espacio para obtener una mejor calidad en los resultados.

Una vez ejecutado el algoritmo, se realizan todos los cálculos pertinentes y se determinan las puntuaciones obtenidas. En el caso de **html_text**, los resultados obtenidos se muestran en la tabla 5.6.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
html_text	0.5166	0.5130	0.9928	0.6765	44.9	0.5	1.1800

Tabla 5.6: Tabla - Resultados de la evaluación de `html_text`

Tanto en la tabla 5.6 como en su correspondiente gráfica 5.5, podemos observar que el algoritmo cumple asequiblemente con unos requisitos mínimos con respecto a aquellas métricas no dependientes del entorno de ejecución.

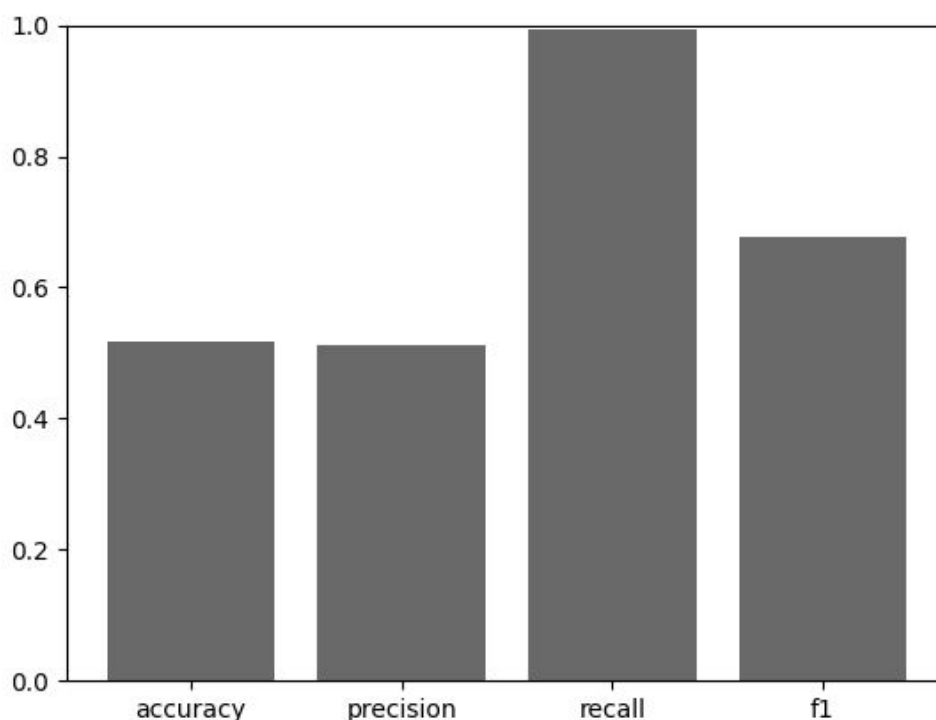


Figura 5.5: Gráfica - Resultados de la evaluación de `html_text`

En cuanto a las métricas dependientes de entorno, es cierto que el algoritmo emplea pocos recursos, y su tiempo de ejecución es reducido. Esto se debe a que su heurística es simple, el uso único de expresiones *XPath* optimiza los recursos. ¿Compensa sobre la calidad de la extracción? Lo veremos en la siguiente sección, cuando comparemos la *performance* de todos los paquetes.

html2text

Otro de los paquetes sometidos a análisis será **html2text**, el cual debemos instalar e importar en el fragmento de código respectivo. En cuanto a su instalación, simplemente se debe ejecutar la siguiente instrucción en la línea de comandos: `$ pip install html2text`.

Listado 5.5: Función de ejecución de html2text

```

from html2text import HTML2Text

def run_html2_text():
    '''extraccion de texto empleando html2_text'''
    output = {} #diccionario salida

    for path in Path('archivos_html').glob('*.html'):
        with open(path, 'r', encoding = "utf-8") as file:
            html_to_string = file.read()

            HTML2Text().ignore_links , HTML2Text().ignore_images = True
            output[path.stem] = {'texto': HTML2Text().handle(html_to_string)}

    return output

```

En el fragmento de código 5.5 se muestra como **html2text** extrae texto de los diferentes documentos HTML. Si recordamos, **html2text** no se caracteriza por disponer de una heurística compleja, el algoritmo simplemente hace uso de *html.parse* para analizar el documento y envolver todos los párrafos del texto proporcionado.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
html2text	0.5105	0.5107	0.9804	0.6715	44.6	1.8	4.4020

Tabla 5.7: Tabla - Resultados de la evaluación de html2text

Los resultados mostrados, tanto en la tabla 5.7 como en la gráfica 5.6, hacen reflejo de la simplicidad de su heurística. A pesar de todo ello, la calidad en general de la extracción no está muy lejos de la media.

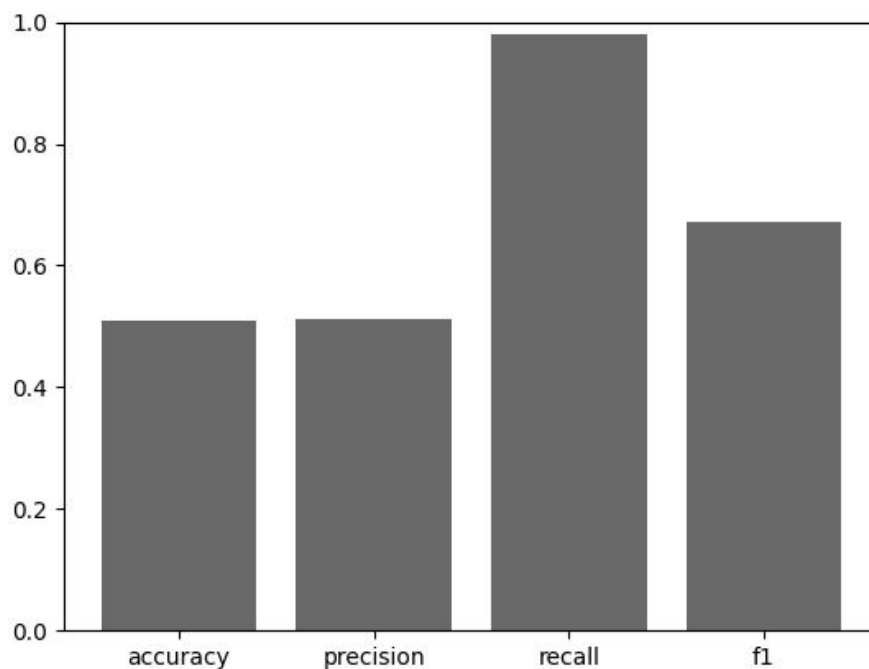


Figura 5.6: Gráfica - Resultados de la evaluación de html2text

Readability

El siguiente paquete sometido a análisis será **Readability**, el cual debemos instalar e importar en el fragmento de código respectivo. En cuanto a su instalación, simplemente se debe ejecutar la siguiente instrucción en la línea de comandos: `$ pip install readability-lxml`.

Listado 5.6: Función de ejecución de Readability

```
import html_text
from readability import Document

def run_readability():
    '''extraccion de texto empleando readability'''
    output = {} #diccionario salida

    for path in Path('archivos_html').glob('*.html'):
        with open(path, 'r', encoding = "utf-8") as file:
            html_to_string = file.read()

            doc = Document(html_to_string).summary(html_partial=True)
            output[path.stem] = {'texto': html_text.extract_text(doc)}

    return output
```

En el fragmento de código 5.6 se muestra como **Readability** extrae texto de los diferentes documentos HTML. El algoritmo es muy simple y similar a los algoritmos de otros paquetes. Veamos los resultados de las métricas calculadas.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Readability	0.8880	0.9101	0.9370	0.9233	45.3	1.6	3.5952

Tabla 5.8: Tabla - Resultados de la evaluación de Readability

Por otro lado, si recordamos la heurística del paquete, se empleaba la selección del nodo candidato para encontrar nodos de valor. Tras una primera selección se buscaban nodos adyacentes con el mismo propósito. Se puede observar tanto en la tabla 5.8 como en la gráfica 5.7, que la heurística parece funcionar. El algoritmo presenta una buena calidad en la extracción, los cálculos de *accuracy* y *precision* son muy buenos con respecto a los paquetes ya evaluados.

Además de **Readability**, **Goose3** empleaba la misma técnica de nodo candidato para encontrar nodos de valor. Se muestra en la tabla 5.9 una pequeña comparación entre ambos paquetes, donde los resultados referentes a la calidad de la extracción, son muy parejos.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Readability	0.8880	0.9101	0.9370	0.9233	45.3	1.6	3.5952
Goose3	0.8658	0.9273	0.8913	0.9089	32.2	6.1	25.9731

Tabla 5.9: Tabla - Comparación de resultados entre Readability & Goose3

Además de la comparación vista, más adelante se realizará una evaluación individual de **Goose3** el cual hasta entonces, al igual que **Readability**, parece un algoritmo bastante completo.

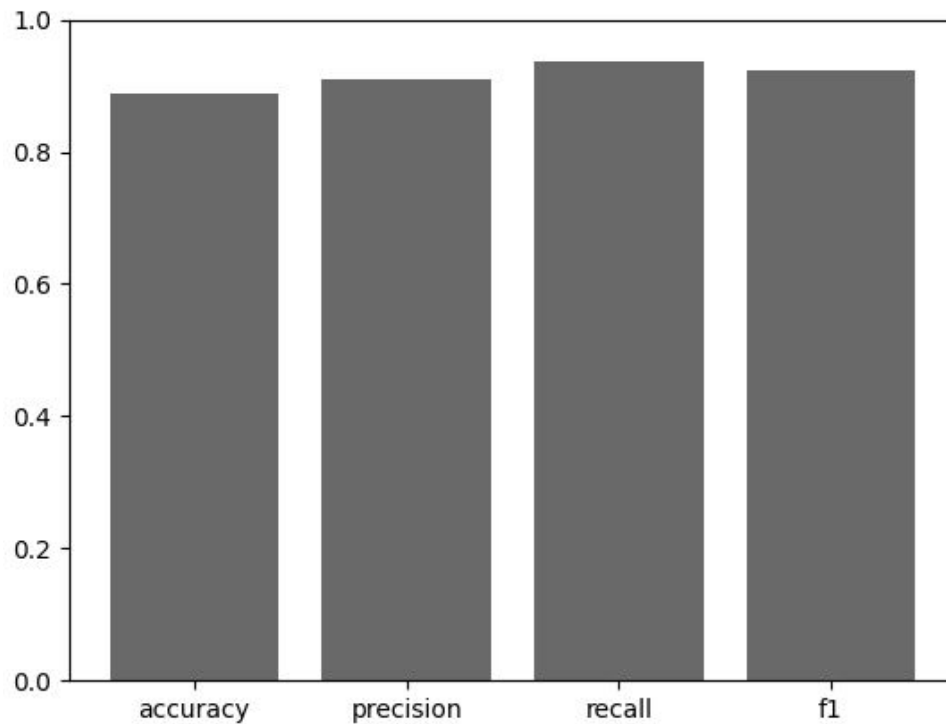


Figura 5.7: Gráfica - Resultados de la evaluación de Readability

En cuanto a las métricas dependientes del entorno, tanto los resultados de uso de memoria RAM, como el uso de CPU y el tiempo de ejecución del algoritmo entran dentro de lo estándar. La calidad de la información extraída y los resultados obtenidos en tiempo de ejecución y empleo de recursos, reflejan una muy buena optimización del paquete.

Trafilatura

Otro paquete sometido a análisis será **Trafilatura**, el cual debemos instalar e importar en el fragmento de código respectivo. En cuanto a su instalación, simplemente se debe ejecutar la siguiente instrucción en la línea de comandos: *\$ pip install trafilatura*.

Listado 5.7: Función de ejecución de Trafilatura

```
import trafilatura as tf

def run_trafilatura():
    '''extraccion de texto empleando trafilatura'''
    output = {} #diccionario salida

    for path in Path('archivos_html').glob('*.html'):
        with open(path, 'r', encoding = "utf-8") as file:
            html_to_string = file.read()

            output[path.stem] = {'texto': tf.extract(html_to_string)}

    return output
```

En cuanto a la función de extracción, debemos recordar que **Trafilatura** dispone de diferentes configuraciones con las que poder determinar la forma en la que se va a realizar el minado. En el fragmento de código 5.7 se muestra la función más genérica.

Tanto en la tabla 5.10 como en la correspondiente gráfica 5.8 se pueden observar los resultados de ejecutar **Trafilatura** en su versión más genérica. Las métricas calculadas determinan que la heurística empleada por el algoritmo es muy completa.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Trafilatura	0.9124	0.9196	0.9707	0.9444	45.7	1.4	4.3919

Tabla 5.10: Tabla - Resultados de la evaluación de Trafilatura

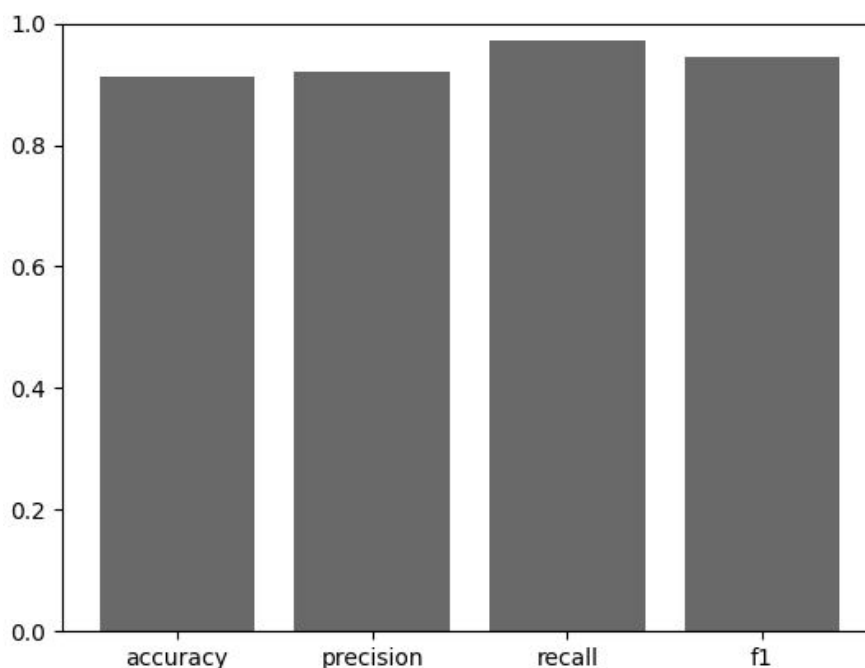


Figura 5.8: Gráfica - Resultados de la evaluación de Trafilatura

Listado 5.8: Función de ejecución de Trafilatura (*precision*)

```
import trafileatura as tf

def run_trafileatura_precision():
    '''extraccion de texto empleando trafileatura tipo precision'''
    output = {} #diccionario salida

    for path in Path('archivos_html').glob('*.html'):
        with open(path, 'r', encoding = "utf-8") as file:
            html_to_string = file.read()

            output[path.stem] = {'texto': tf.extract(html_to_string,
                no_fallback = False, favor_precision = True)}

    return output
```

Veamos las diferentes configuraciones de **Trafilatura** para potenciar la calidad del texto extraído. En primer lugar, vamos a mejorar la precisión del algoritmo. Como se puede observar en el fragmento de código 5.8, se activan además los algoritmos de respaldo.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Tra. precision	0.9129	0.9208	0.9699	0.9447	45.9	1.4	4.4590

Tabla 5.11: Tabla - Resultados de la evaluación de Trafilatura (*precision*)

Podemos observar en la tabla 5.11, que los resultados obtenidos reflejan una mínima mejora en lo referido a la *precision*. Algo sorprendente es que la inclusión de los algoritmos de respaldo no aumentan el uso de recursos ni el tiempo de ejecución.

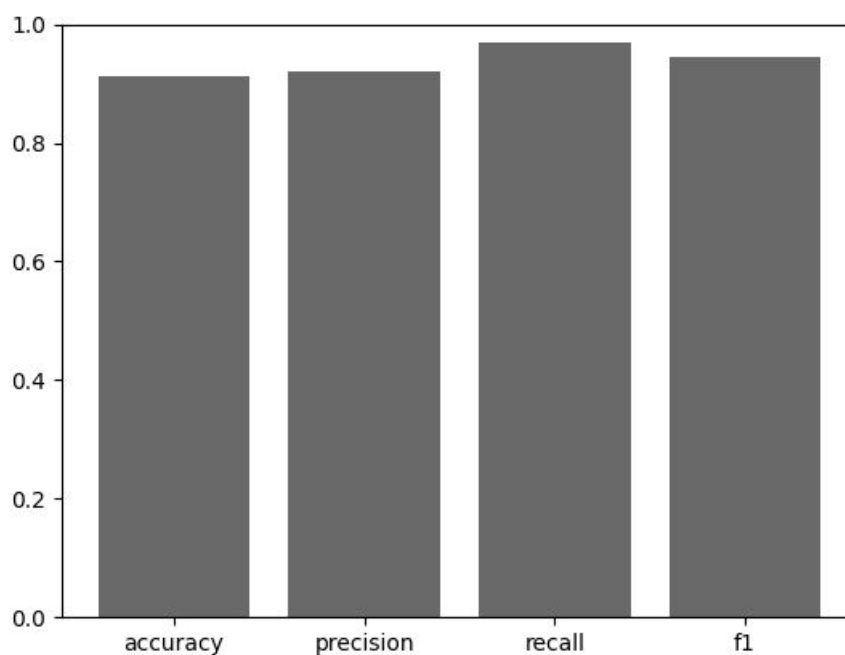


Figura 5.9: Gráfica - Resultados de la evaluación de Trafilatura (*precision*)

Listado 5.9: Función de ejecución de Trafilatura (*recall*)

```
import trafileatura as tf

def run_trafileatura_recall():
    '''extraccion de texto empleando trafileatura tipo recall'''
    output = {} #diccionario salida

    for path in Path('archivos_html').glob('*.html'):
        with open(path, 'r', encoding = "utf-8") as file:
            html_to_string = file.read()

            output[path.stem] = {'texto': tf.extract(html_to_string,
                                                    no_fallback = False, favor_recall = True)}

    return output
```

Otra de las configuraciones permite mejorar la capacidad de captar las partes deseadas del documento, es decir, permite aumentar el valor de la métrica *recall*. Como podemos observar en el fragmento de código 5.9 los algoritmos de respaldo siguen activos. Veamos los resultados obtenidos.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Tra. recall	0.9188	0.9287	0.9776	0.9525	46.3	0.5	3.0888

Tabla 5.12: Tabla - Resultados de la evaluación de Trafilaturation (*recall*)

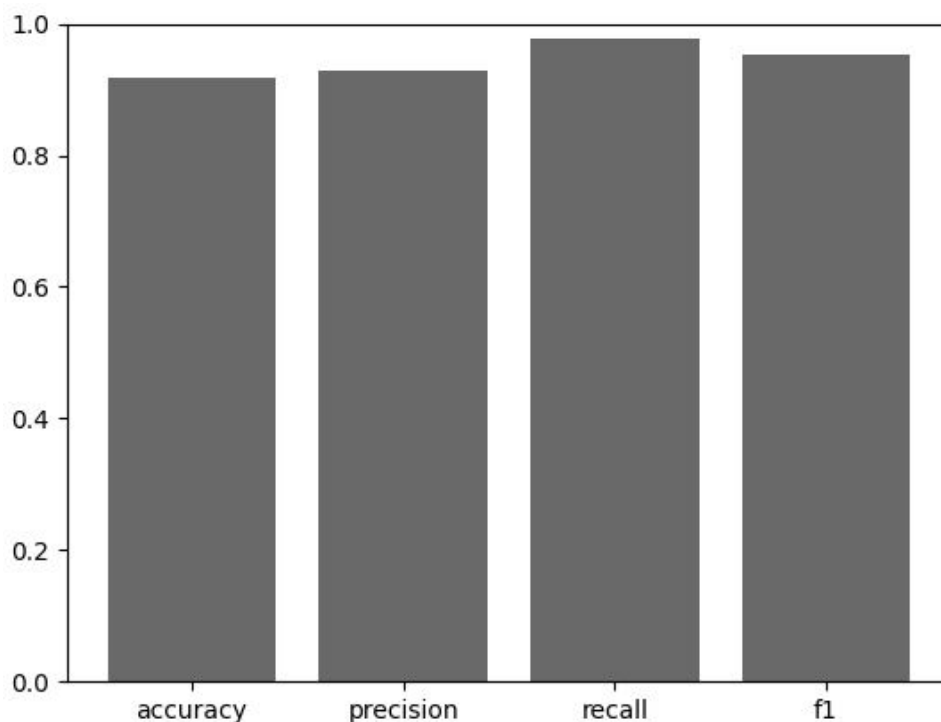


Figura 5.10: Gráfica - Resultados de la evaluación de Trafilaturation (*recall*)

En los resultados mostrados en la tabla 5.12 y en la gráfica 5.10, se puede observar que nuevamente la mejora es mínima. La calidad de la información extraída es prácticamente la misma que la ofrecida por la heurística anterior.

Haciendo una comparación más amplia entre los tres tipos de configuraciones para **Trafilaturation**, podemos observar que la heurística dedicada a seleccionar las partes deseadas de un documento es la más refinada. Esta supera incluso el valor de la métrica *precision* de la propia heurística dedicada a ello. Véase 5.11.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Trafilaturation	0.9124	0.9196	0.9707	0.9444	45.7	1.4	4.3919
Tra. precision	0.9129	0.9208	0.9699	0.9447	45.9	1.4	4.4590
Tra. recall	0.9188	0.9287	0.9776	0.9525	46.3	0.5	3.0888

Tabla 5.13: Tabla - Comparación de resultados entre los tres algoritmos de Trafilaturation

Por otra parte, los valores de referidos al uso de recursos del entorno de ejecución también se ven reducidos en la configuración para *recall*. Al parecer la ejecución de los algoritmos de respaldo no afecta al consumo de recursos del entorno de ejecución. Véase 5.12.

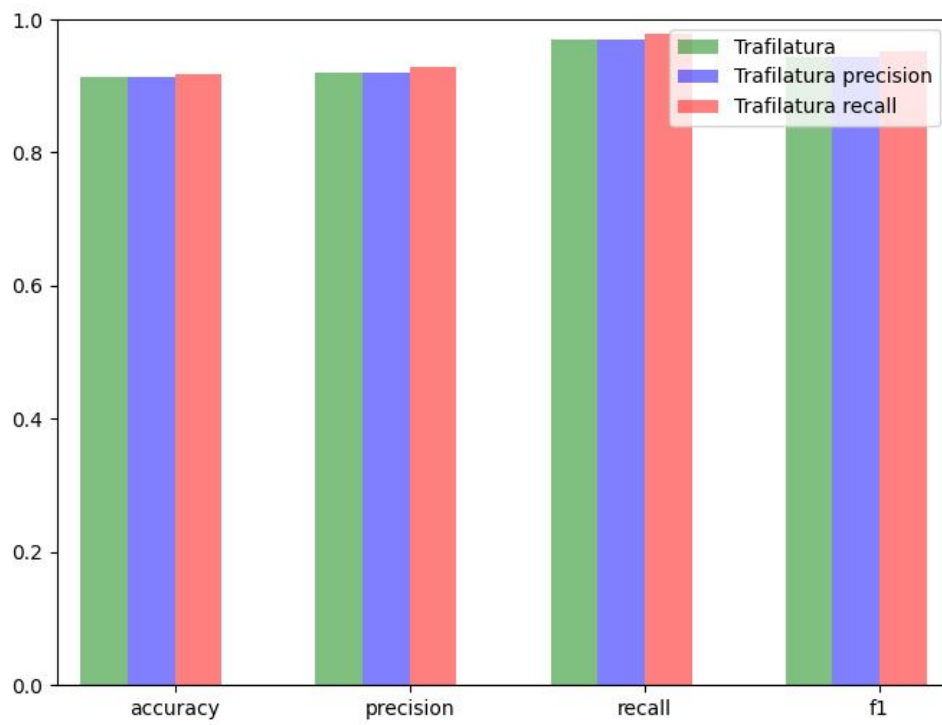


Figura 5.11: Gráfica - Comparación de resultados entre los tres algoritmos de Trafilatura

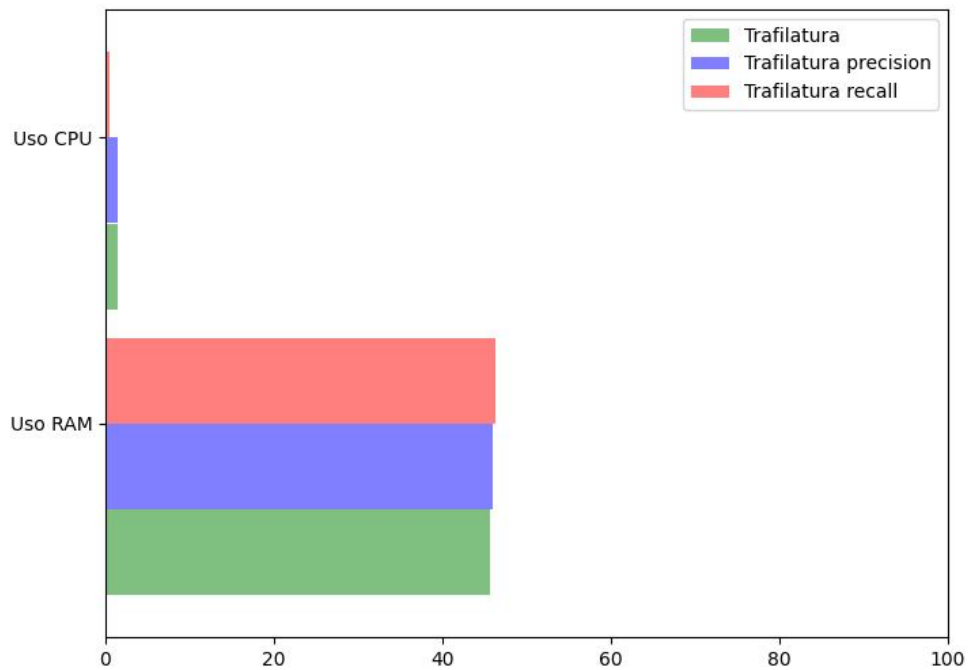


Figura 5.12: Gráfica - Comparación de resultados entre los tres algoritmos de Trafilatura 2

Goose3

El siguiente paquete sometido a análisis es **Goose3**, el cual debemos instalar e importar en el fragmento de código respectivo. En cuanto a su instalación, simplemente se debe ejecutar la siguiente instrucción en la línea de comandos: *\$ pip install goose3*.

Listado 5.10: Función de ejecución de Goose3

```
from goose3 import Goose

def run_goose3():
    '''extraccion de texto empleando goose3'''
    output = {} #diccionario salida

    for path in Path('archivos_html').glob('*.html'):
        with open(path, 'r', encoding = "utf-8") as file:
            html_to_string = file.read()

            g = Goose()
            article = g.extract(raw_html = html_to_string)

            output[path.stem] = {'texto': article.cleaned_text}

    return output
```

Al igual que otros algoritmos ya vistos, **Goose3** permite modificar parte de su configuración con el objetivo de personalizar la salida obtenida. Hay dos formas de pasar la configuración. La primera consiste en pasarle al algoritmo un objeto *Configuration()*, la segunda es a través de un diccionario de configuración.

```
>>> g = Goose({'browser_user_agent': 'Mozilla'})
>>> g = Goose({'browser_user_agent': 'Mozilla', 'parser_class': 'soup'})
...
>>> g = Goose({'strict': False})
```

En el fragmento de código 5.10 se muestra las líneas de código empleadas para que **Goose3** pueda extraer la información deseada de documentos HTML. Se observa que la forma de actuar es la misma que en el resto de algoritmos ya vistos.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Goose3	0.8658	0.9273	0.8913	0.9089	32.2	6.1	25.9731

Tabla 5.14: Tabla - Resultados de la evaluación de Goose3

En cuanto a calidad de extracción, en la tabla 5.14 podemos observar que los valores son buenos. La mayor parte del texto extraído son partes deseadas y la exclusión de contenido *boilerplate* también es muy buena.

En contrapartida, los valores referentes a la optimización de recursos y tiempo de ejecución, dejan algo que desear. Además del exagerado tiempo que tarda el algoritmo en ejecutar, el uso de CPU es elevado respecto a otros algoritmos.

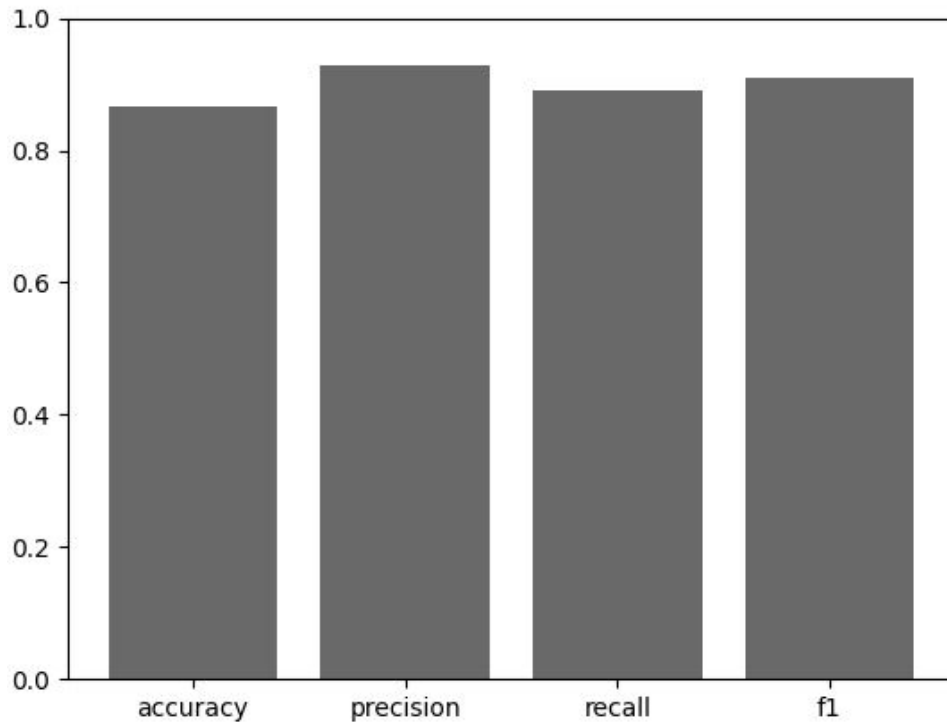


Figura 5.13: Gráfica - Resultados de la evaluación de Goose3

Boilerpy

Otro de los paquetes sometidos a análisis es **Boilerpy**, el cual debemos instalar e importar en el fragmento de código respectivo. En cuanto a su instalación, simplemente se debe ejecutar la siguiente instrucción en la línea de comandos: *\$ pip install boilerpy3*.

Listado 5.11: Función de ejecución de Boilerpy

```
from boilerpy3 import extractors

def run_boilerpy():
    '''extraccion de texto empleando boilerpipe'''
    output = {} #diccionario salida

    for path in Path('archivos_html').glob('*.html'):
        with open(path, 'r', encoding = "utf-8") as file:
            html_to_string = file.read()

            extractor = extractors.ArticleExtractor()
            output[path.stem] = {'texto': extractor.get_content(html_to_string)}

    return output
```

En el fragmento de código 5.11 se muestra como **Boilerpy** extrae texto de diferentes documentos HTML. Recordemos que el algoritmo disponía de múltiples extractores que se usaban dependiendo del tipo de información a obtener. En este caso se emplea *ArticleExtractor()* el cual extrae el texto de artículos.

En cuanto a los resultados obtenidos, entran dentro de lo estándar. La calidad del texto que se extrae es buena y el uso de recursos es reducido. Véase 5.15 y 5.14.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Boilerpy	0.7947	0.8544	0.8803	0.8672	43.9	1.9	2.5412

Tabla 5.15: Tabla - Resultados de la evaluación de Boilerpy

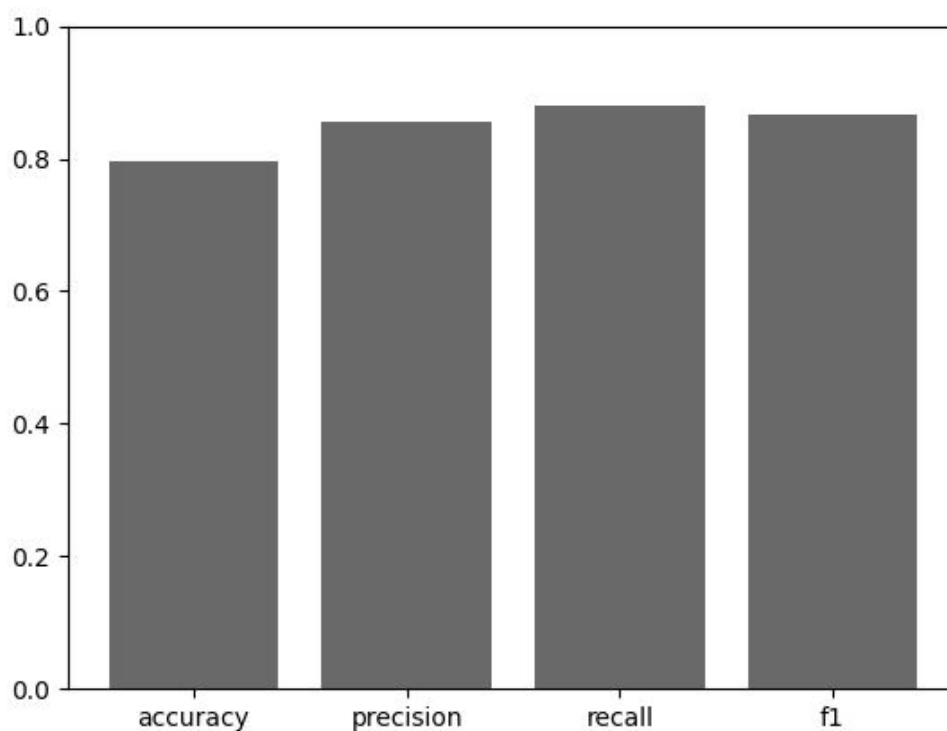


Figura 5.14: Gráfica - Resultados de la evaluación de Boilerpy

BoilerpipeR

La version en **R** de **Boilerpy** es **BoilerpipeR**, el cual debemos instalar e importar en el fragmento de código respectivo. En cuanto a su instalación, simplemente se debe ejecutar la siguiente instrucción en la línea de comandos de R: `install.packages("boilerpipeR")`.

Listado 5.12: Ejecución de BoilerpipeR desde Python

```
from rpy2.robjects import r

def run_boilerpipeR():
    '''ejecucion de boilerpipeR, codigo fuente en ./run_boilerpipeR.R'''
    r('source("./algoritmos_extraccion/run_boilerpipeR.R")')
```

Al ser este un paquete propio de R, es necesaria una interfaz que nos permita ejecutarlo en Python. En el fragmento de código 5.12 se muestra el proceso.

Por otro lado, en cuanto a la propia función de ejecución, se puede observar en el fragmento de código 5.13 como está programado el algoritmo. Podemos observar que la forma de proceder es similar a la ya propuesta por los algoritmos de otras herramientas de minado.

Listado 5.13: Función de ejecución de BoilerpipeR

```
library(boilerpipeR)

run_boilerpipeR <- function() {
  ###extraccion de texto empleando boilerpipeR###
  output <- c() #diccionario salida

  files <- list.files(path = "./archivos_html", pattern = ".html")
  index <- 0
  for (key in fromJSON(file = "./documento_base.json")) {
    html_to_str <- stri_enc_toutf8(ArticleExtractor(key$url, asText = FALSE))
    lista_texto <- list('test' = list("texto" = html_to_str))

    names(lista_texto) <- str_remove(files[index + 1], ".html")
    output <- c(output, lista_texto)
  }
}
```

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
BoilerpipeR	0.7830	0.8486	0.8696	0.8590	47.9	2.6	39.9543

Tabla 5.16: Tabla - Resultados de la evaluación de BoilerpipeR

Realizando una apreciación general sobre los datos obtenidos, podemos observar que **BoilerpipeR** realiza un buen trabajo de minado, véase 5.16. Comparemos ahora los resultados obtenidos, con los calculados anteriormente para **Boilerpy**.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Boilerpy	0.7947	0.8544	0.8803	0.8672	43.9	1.9	2.5412
BoilerpipeR	0.7830	0.8486	0.8696	0.8590	47.9	2.6	39.9543

Tabla 5.17: Tabla - Comparación de resultados entre Boilerpy & BoilerpipeR

Se muestra en la tabla 5.17 los datos obtenidos para ambas herramientas. En ambos algoritmos se ha empleado *ArticleExtractor* como extractor estándar. Podemos observar que apenas existen claras diferencias entre ambos resultados, respecto a la calidad del texto extraído, las dos herramientas reflejan unos resultados muy parejos.

En contrapartida, los resultados obtenidos con respecto a empleo de recursos del entorno de ejecución son algo diferentes. El tiempo que tarda en ejecutarse **BoilerpipeR** se aleja demasiado de la media de algoritmos visto hasta ahora. Veamos una comparación gráfica.

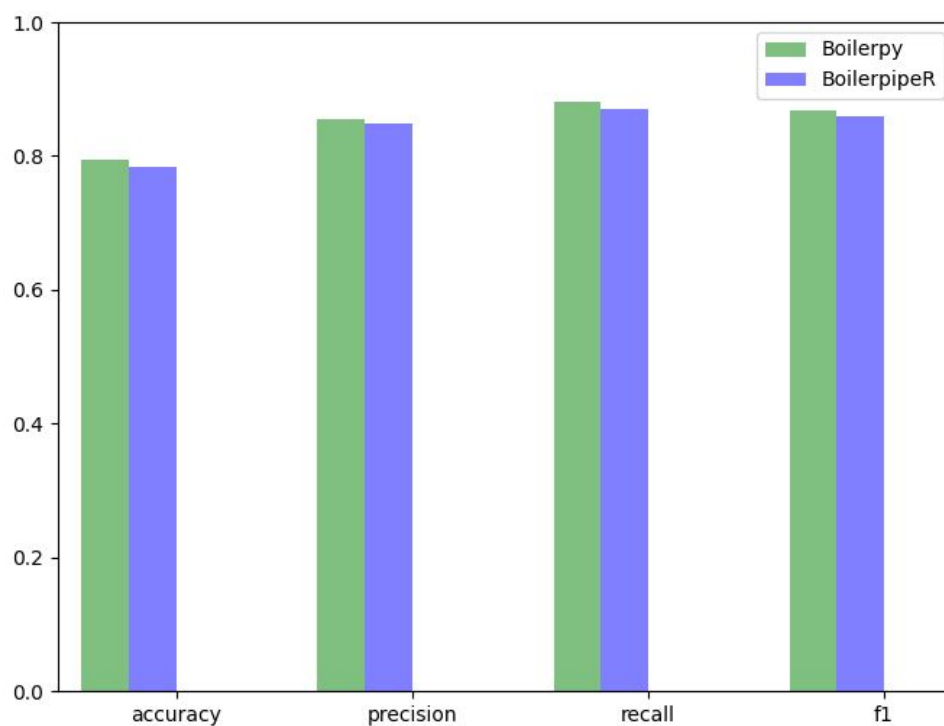


Figura 5.15: Gráfica - Comparación de resultados entre Boilerpy & BoilerpipeR

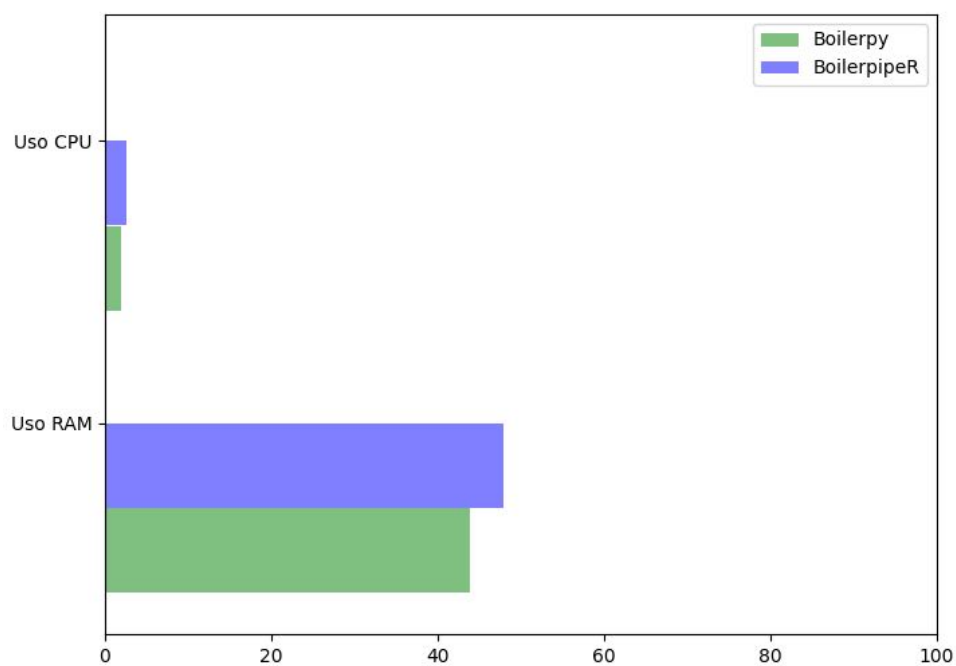


Figura 5.16: Gráfica - Comparación de resultados entre Boilerpy & BoilerpipeR 2

rvest

El siguiente paquete sometido a análisis es **rvest**, el cual debemos instalar e importar en el fragmento de código respectivo. En cuanto a su instalación, simplemente se debe ejecutar la siguiente instrucción en la línea de comandos de R: `install.packages("rvest")`.

Listado 5.14: Ejecución de rvest desde Python

```
from rpy2.robjects import r

def run_rvest():
    '''ejecucion de rvest, codigo fuente en ./run_rvest.R'''
    r('source("./algoritmos_extraccion/run_rvest.R")')
```

Listado 5.15: Función de ejecución de rvest

```
library(rvest)

run_rvest <- function() {
    ###extraccion de texto empleando rvest###
    output <- c() #diccionario salida

    files <- list.files(path = "./archivos_html", pattern = ".html")
    index <- 0
    for (key in fromJSON(file = "./documento_base.json")) {
        html_to_string <- read_html(key$url, encoding = "UTF-8") %%
            html_nodes(xpath = '//body') %% html_text()
        lista_texto <- list('test' = list("texto" = html_to_string))

        names(lista_texto) <- str_remove(files[index + 1], ".html")
        output <- c(output, lista_texto)
    }
}
```

En el fragmento de código 5.15 se muestra la función empleada para ejecutar **rvest**. Se puede observar el empleo del operador tubería `%>%` el cual se encarga de facilitar la transferencia de resultados de las diferentes funciones implicadas.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
rvest	0.1347	0.1371	0.8974	0.2378	44.1	8.9	60.3245

Tabla 5.18: Tabla - Resultados de la evaluación de rvest

En cuanto a las métricas, los datos reflejados en 5.18 dejan bastante claro que la calidad de **rvest** en la extracción de contenido no es buena. Uno de los principales motivos es el uso necesario de expresiones *XPath*. En este caso `xpath = //body` permite al algoritmo trabajar con la etiqueta *body* del documento HTML en cuestión.

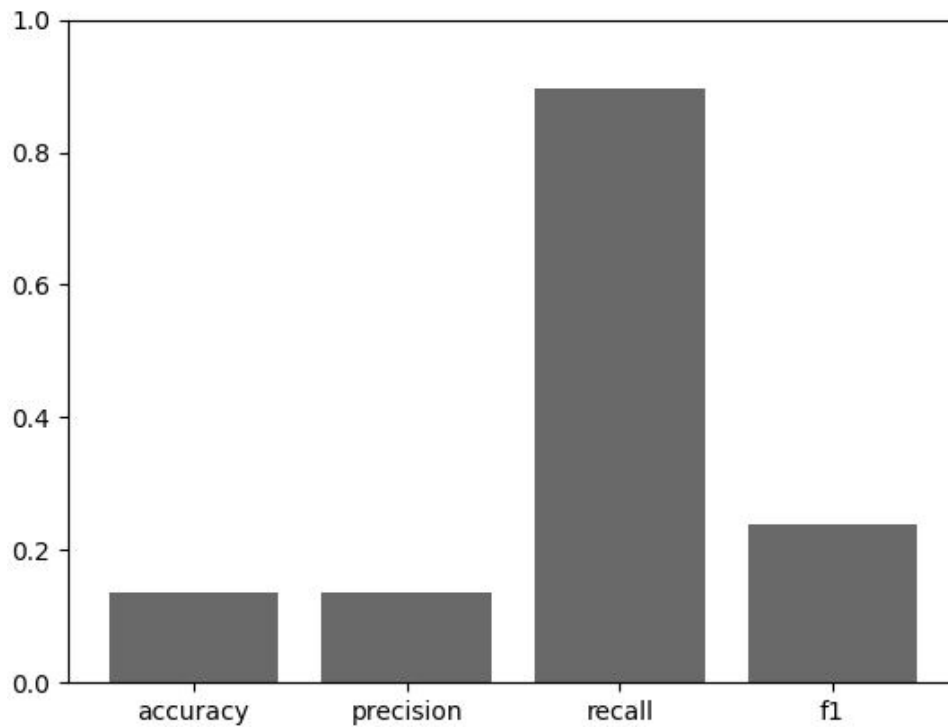


Figura 5.17: Gráfica - Resultados de la evaluación de rvest

Rcrawler

Otro paquete propio de R sometido a análisis será **Rcrawler**, el cual debemos instalar e importar en el fragmento de código respectivo. En cuanto a su instalación, simplemente se debe ejecutar la siguiente instrucción en la línea de comandos: *install.packages(Rcrawler", dependencies = TRUE)*.

Listado 5.16: Función de ejecución de Rcrawler

```
library(Rcrawler)

run_rcrawler <- function() {
  ###extraccion de texto empleando rcrawler###

  output <- c() #diccionario salida
  files <- list.files(path = "./archivos_html", pattern = ".html")
  index <- 0
  for (key in fromJSON(file = "./documento_base.json")) {
    html_to_string <- ContentScraper(Url = key$url, XpathPatterns = "//body")
    lista_texto <- list('test' = list("texto" = html_to_string))

    names(lista_texto) <- str_remove(files[index + 1], ".html")
    output <- c(output, lista_texto)
  }
}
```

En referencia al fragmento de código mostrado en 5.16, es muy similar al resto de códigos ya mostrados anteriormente. En este caso se deben emplear nuevamente expresiones *XPath* para trabajar sobre la etiqueta HTML requerida, *XpathPatterns = //body*.

Listado 5.17: Ejecución de Rcrawler desde Python

```
from rpy2.robjects import r

def run_rcrawler():
    '''ejecucion de rcrawler, codigo fuente en ./run_rcrawler.R'''
    r('source("./algoritmos_extraccion/run_rcrawler.R")')
```

Sobre los datos obtenidos, se muestran en la tabla 5.19. Estos reflejan una pobre calidad de la información extraída por parte de **Rcrawler**. Al igual que sucedía con **rvest**, el empleo de expresiones *XPath* limita la información a extraer. Por otro lado, ni el empleo de recursos, ni el tiempo de ejecución son buenos.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Rcrawler	0.4540	0.4628	0.9310	0.6181	46.7	3.4	158.0663

Tabla 5.19: Tabla - Resultados de la evaluación de Rcrawler

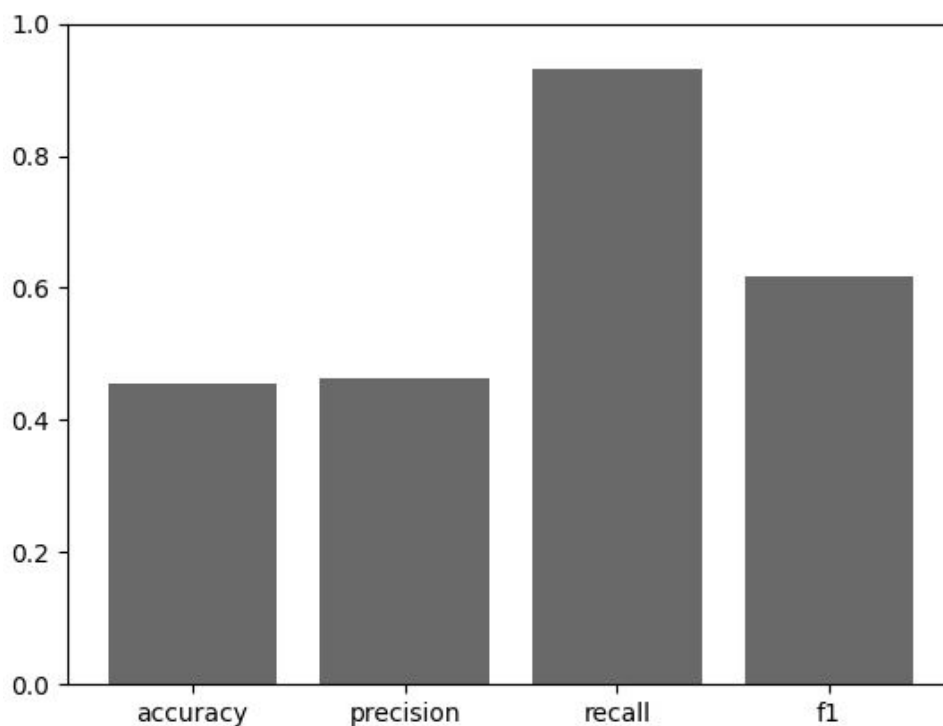


Figura 5.18: Gráfica - Resultados de la evaluación de Rcrawler

htm2txt

Veamos como funciona el paquete **htm2txt**, para ello primero debemos realizar la instalación e importación en el fragmento de código respectivo. En cuanto a su instalación, simplemente se debe ejecutar la siguiente instrucción en la línea de comandos de R: `install.packages('htm2txt')`.

Listado 5.18: Función de ejecución de htm2txt

```
library(htm2txt)

run_htm2txt <- function() {
  ###extraccion de texto empleando htm2txt###

  output <- c() #diccionario salida
  files <- list.files(path = "./archivos_html", pattern = ".html")
  index <- 0
  for (key in fromJSON(file = "./documento_base.json")) {
    html_to_string <- stri_enc_toutf8(gettxt(key$url, encoding = "latin1"))
    lista_texto <- list('test' = list("texto" = html_to_string))

    names(lista_texto) <- str_remove(files[index + 1], ".html")
    output <- c(output, lista_texto)
  }
}
```

Listado 5.19: Ejecución de htm2txt desde Python

```
from rpy2.robjects import r

def run_htm2txt():
    '''ejecucion de htm2txt, codigo fuente en ./run_htm2txt.R'''
    r('source("./algoritmos_extraccion/run_htm2txt.R")')
```

En cuanto al fragmento de código mostrado en 5.18, es muy similar al resto de códigos ya vistos anteriormente. En este caso no entran en juego expresiones *XPath*, sino que es directamente la propia heurística del algoritmo la que se encarga de seleccionar el contenido válido.

Si recordamos la heurística de **htm2txt**, esta se encargaba de utilizar expresiones regulares para eliminar las etiquetas de los documentos HTML pertinentes. En esta heurística tan simple, funciones como `gsub()` tomaban mucha relevancia.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
htm2txt	0.4547	0.4714	0.8885	0.6160	46.7	2.0	80.5288

Tabla 5.20: Tabla - Resultados de la evaluación de htm2txt

Los resultados mostrados en la tabla 5.20 reflejan la simplicidad del algoritmo. El uso único de expresiones regulares hace que la calidad del texto extraído no sea buena. Por otro lado, el tiempo de ejecución es demasiado alto.

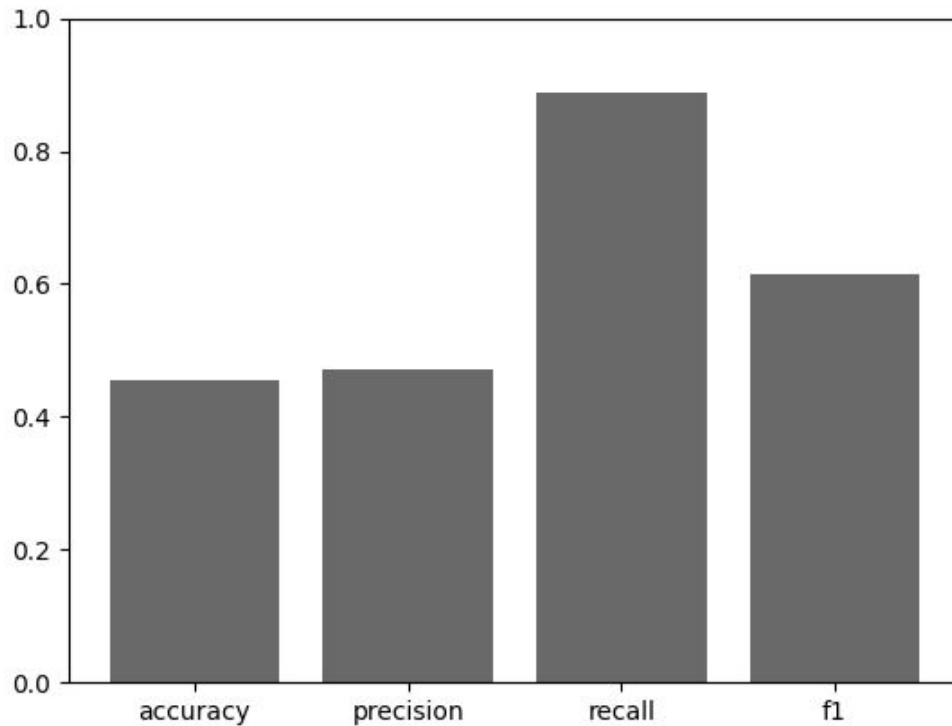


Figura 5.19: Gráfica - Resultados de la evaluación de htm2txt

Expresiones XPath

¿Qué ocurre si solo empleamos expresiones *XPath* sin ningún tipo de heurística? Lo obvio sería que los resultados obtenidos fuesen peores de los ya calculados. Por otro lado, para analizar el documento se usara *lxml*, veamos.

Listado 5.20: Función de ejecución de XPath

```
import lxml.html

def run_xpath_text():
    '''extraccion de texto empleando xpath_text'''
    output = {} #diccionario salida

    for path in Path('archivos_html').glob('*.html'):
        with open(path, 'r', encoding = "utf-8") as input_file:
            html_to_string = input_file.read()

            root = lxml.html.fromstring(html_to_string)
            bodies = root.xpath('//body')
            if bodies: root = bodies[0]

            output[path.stem] = {'texto': ' '.join(root.xpath('//*[text()]'))
                                .replace('\r', '').replace('\n', '')
                                .replace('\r', '').replace('\t', '')}

    return output
```

En cuanto al fragmento de código 5.20, lo único a destacar es la expresión *XPath* usada. Para cada artículo queremos obtener el cuerpo del mismo, por ello se accede a la etiqueta *body* del mismo. Por otro lado, para refinar la solución obtenida, eliminamos las expresiones propias de la estructuración del texto.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
XPath	0.2421	0.2401	0.9915	0.3866	44.4	2.0	0.7476

Tabla 5.21: Tabla - Resultados de la evaluación de XPath

Si observamos la tabla 5.21 podemos comprobar que los resultados obtenidos reflejan unas métricas muy pobres. La calidad del texto extraído es muy inferior a la media del resto de algoritmos. Por otro lado, como era de esperar, el tiempo de ejecución es muy bajo, pues al no disponer de heurística propia la cantidad de ejecuciones a realizar es menor.

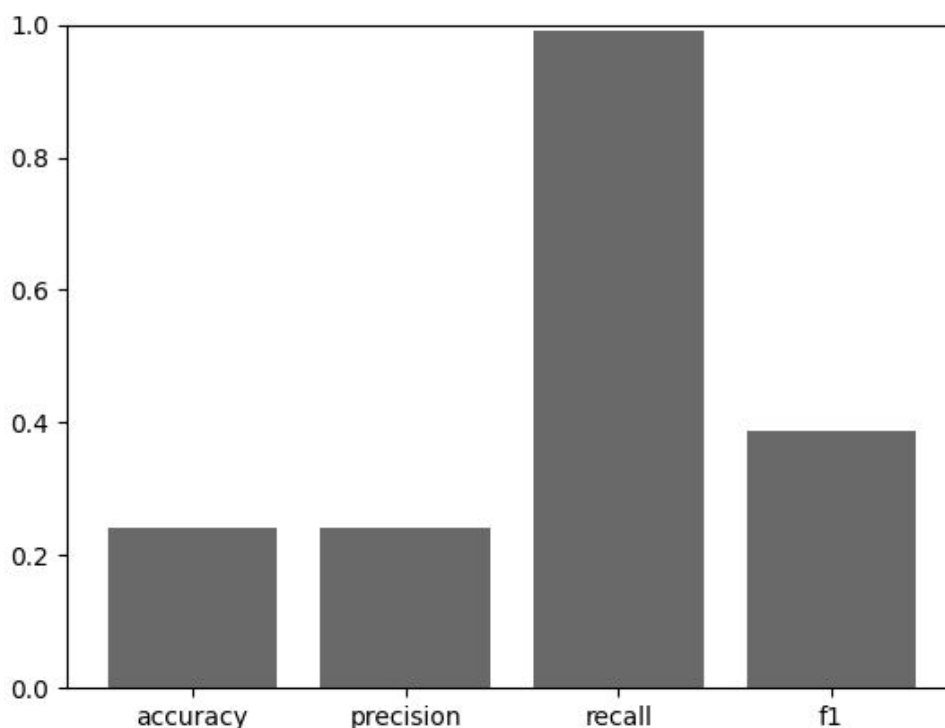


Figura 5.20: Gráfica - Resultados de la evaluación de XPath

Este tipo de paquetes o algoritmos que emplean expresiones *XPath*, ya sea como parte de su heurística, o como única fuente de obtención de información, se emplean mayoritariamente para obtener partes reducidas de un documento HTML.

5.2 Comparación de paquetes

Una vez vistos los resultados individuales de cada paquete, veamos una comparación de los mismos desde diferentes puntos de vista. El objetivo de esta comparación es obtener de una manera más clara cuáles son los causantes de los diferentes resultados obtenidos. En muchos casos, el analizador empleado, la heurística, o el propio lenguaje de programación se convierten en un factor fundamental.

5.2.1 Comparación de resultados entre paquetes que usan expresiones XPath

El uso de expresiones regulares es una técnica conocida que puede utilizarse en el proceso de extracción, se determina un cierto patrón y se buscan etiquetas a lo largo del documento que coincidan. Sin embargo, pueden existir problemas cuando el número de etiquetas internas es ambiguo [42]. Para una regla de extracción, existen dos técnicas. Se puede buscar en el documento todos los registros encontrados de la expresión regular, o el proceso de extracción puede finalizar con la búsqueda del primer registro [43].

En este caso, la comparación abordará aquellos paquetes que basen su heurística a partir del uso de este tipo de expresiones. Paquetes como **Rcrawler** y **rvest** serán los seleccionados, entre otros. Cabe mencionar que la comparación se realizará a partir de las métricas calculadas en la sección anterior. Además, será interesante ver una comparación con el simple empleo de expresiones *XPath*.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
rvest	0.1347	0.1371	0.8974	0.2378	44.1	8.9	60.3245
Rcrawler	0.4540	0.4628	0.9310	0.6181	46.7	3.4	158.0663
htm2txt	0.4547	0.4714	0.8885	0.6160	46.7	2.0	80.5288
XPath	0.2421	0.2401	0.9915	0.3866	44.4	2.0	0.7476

Tabla 5.22: Tabla - Comparación de resultados entre paquetes que usan expresiones XPath

Podemos observar en la tabla 5.22 los resultados obtenidos por los paquetes que emplean expresiones *XPath*. De manera general podemos decir que los resultados dejan bastante que desear. Resultados muy pobres para el tiempo de ejecución empleado. Veamos ahora una comparación más minuciosa, observando las diferencias entre las métricas de forma individual.

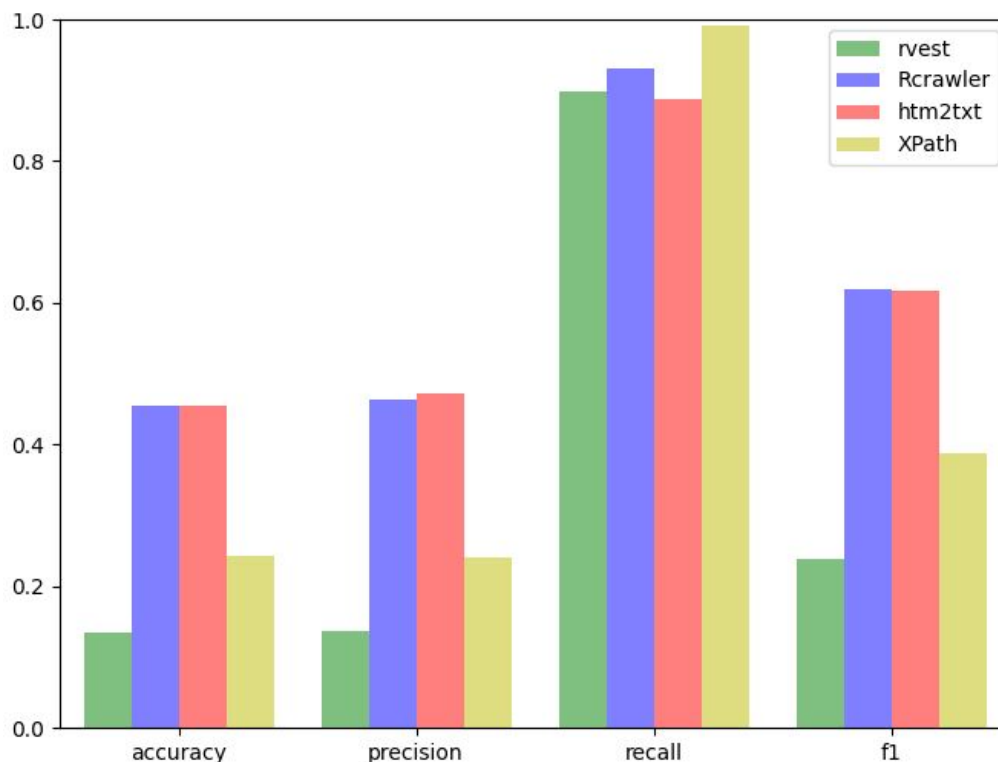


Figura 5.21: Gráfica - Comparación de resultados entre paquetes que usan expresiones XPath

En la gráfica 5.21 podemos observar de manera más visual los resultados obtenidos. Sorprende que **rvest** tenga unos resultados tan pobres, siendo superado incluso por expresiones regulares. Por otro lado, **Rcrawler** sin tener resultados buenos, es el único que realiza un decente proceso de *web scraping*.

¿Cómo es posible que se haya captado la mayor parte del contenido principal y los resultados del resto de métricas sean tan malos? La razón es que además de captar contenido principal, el algoritmo ha captado contenido no deseado, esto empeora la calidad de la solución. Si observamos la relación entre la variable *recall* y la variable *precision*, nos damos cuenta de que captación de contenido principal es buena, pero la exclusión de contenido *boilerplate* no lo es tanto.

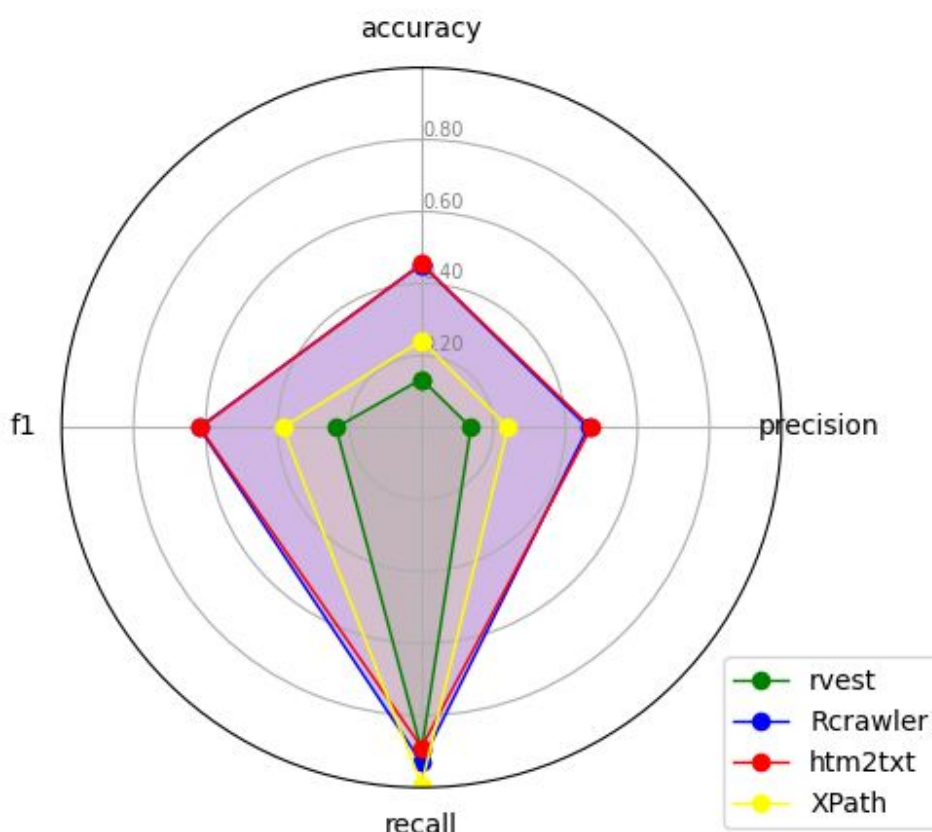


Figura 5.22: Comparación de la eficacia en los paquetes que usan expresiones XPath

Vistos los diferentes resultados obtenidos, y observando la imagen 5.22 podemos concluir que los algoritmos sometidos a evaluación que emplean expresiones *XPath* no realizan un minado web de calidad. Por otro lado, los resultados referentes al rendimiento y optimización de los mismos tampoco son buenos.

Si estos algoritmos no ofrecen un resultado de calidad y su rendimiento no es bueno, ¿por qué son tan utilizados? La respuesta es que este tipo de algoritmos son buenos cuando deseas encontrar determinadas etiquetas dentro de un documento HTML y la calidad de la solución no es importante.

5.2.2 Comparación de resultados entre paquetes según el analizador

El uso de expresiones regulares es una técnica conocida que puede utilizarse en el proceso de extracción. Sin embargo, puede causar problemas cuando el número de etiquetas internas es ambiguo. En esta situación el empleo de analizadores pueden usarse como una posible solución.

A lo largo del apéndice B, ya se realizó una breve introducción de los analizadores empleados en los algoritmos de *web scraping*. Veamos ahora la importancia de cada uno en la eficacia de los mismos.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Boilerpy	0.7947	0.8544	0.8803	0.8672	43.9	1.9	2.5412
Goose3	0.8658	0.9273	0.8913	0.9089	32.2	6.1	25.9731
html2text	0.5105	0.5107	0.9804	0.6715	44.6	1.8	4.4020
Beau. Soup	0.5165	0.5129	0.9928	0.6764	47.0	4.3	4.0882

Tabla 5.23: Tabla - Comparación de resultados entre paquetes que emplean *html.parser* como analizador

Empecemos primero con aquellos algoritmos de que emplean *html.parse* como parte de su heurística. Podemos observar en la tabla 5.23 los resultados obtenidos para los cuatro diferentes algoritmos. Si recordamos como funcionaba *html.parse*, trabaja con etiquetas, además no es un algoritmo excesivamente rápido y es indulgente en su análisis.

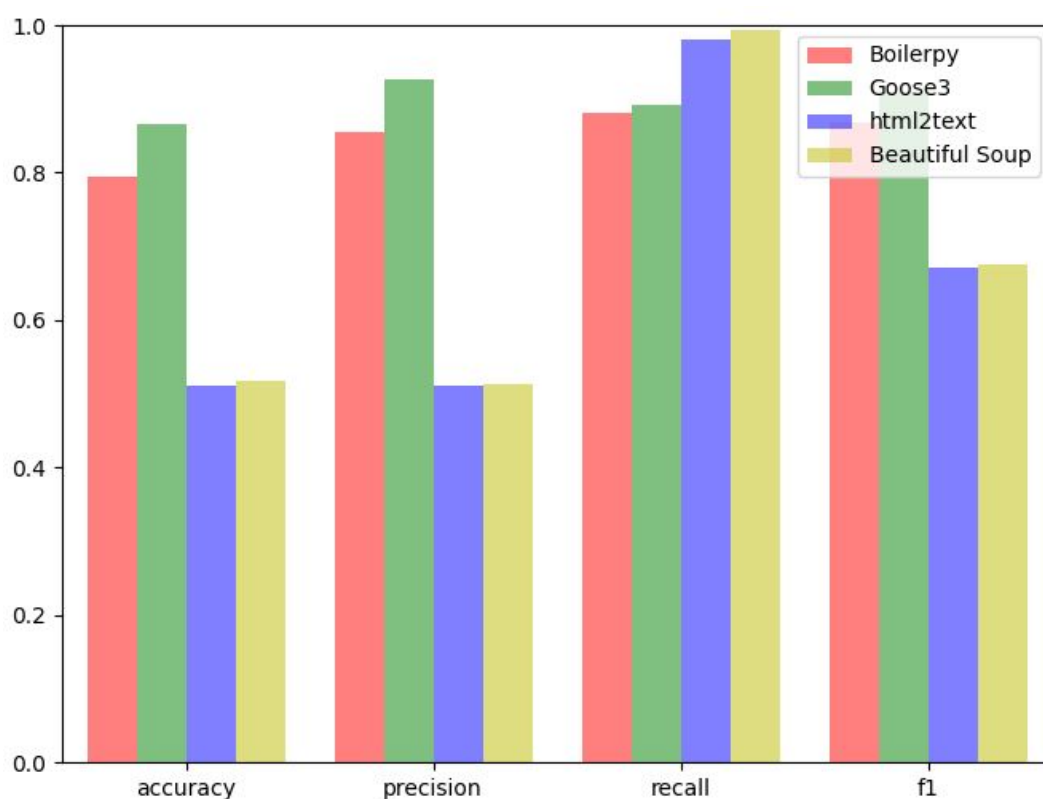


Figura 5.23: Gráfica - Comparación de resultados entre paquetes que usan *html.parser* como analizador

Si observamos la gráfica 5.23, y ponemos atención en las variables *accuracy* y *precision*, nos damos cuenta de que los valores mostrados son muy dispares. Al tratar con un analizador tan permisivo, es necesario crear una heurística elaborada para excluir la mayor cantidad de contenido *boilerplate*.

Siguiendo este aspecto, en cuanto a la captación de contenido principal, todos los algoritmos retornan buenos resultados. Sorprende incluso ver que los datos mostrados para **Beautiful Soup** y **html2text**. Entonces, ¿a qué pueden deberse estas diferencias en las métricas? La explicación es que **Boilerpy** y **Goose3** presentan una heurística más compleja y elaborada.

En cuanto a las métricas dependientes del entorno de ejecución, los valores relativos al uso de memoria RAM, CPU y tiempo de ejecución empleado son bastante estándar. El único paquete que presenta valores que difieren del resto es **Goose3**.

Acabamos de ver el rendimiento de aquellos paquetes que emplean *html.parse* como analizador. Veamos ahora como se comportan los paquetes que usan *lxml* como parte de su heurística. Podemos observar en la tabla 5.24 los resultados obtenidos para los cuatro diferentes algoritmos.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
inscriptis	0.5414	0.5404	0.9875	0.6985	45.0	0.2	2.1009
Beau. Soup	0.5165	0.5129	0.9928	0.6764	42.0	1.2	3.1778
html_text	0.5166	0.5130	0.9928	0.6765	44.9	0.5	1.1800
jusText	0.7668	0.8649	0.8573	0.8610	45.1	0.5	2.9546
Readability	0.8880	0.9101	0.9370	0.9233	45.3	1.6	3.5952
Trafilatura	0.9124	0.9196	0.9928	0.9444	45.7	1.4	4.3919

Tabla 5.24: Tabla - Comparación de resultados entre paquetes que emplean *lxml* como analizador

Recordemos que para analizar los diferentes documentos, *lxml* convertía el texto de entrada en una estructura de tipo árbol. Esto le permite navegar y encontrar la información requerida de forma más sencilla. Al igual que *html.parse*, este es un analizador permisivo, por lo que el trabajo de eliminación de contenido *boilerplate* será delegado a la heurística determinada.

En cuanto a la comparación de algoritmos, si observamos las variables de *accuracy* y *precision* en la gráfica 5.24, nos damos cuenta de la disparidad de resultados entre los algoritmos. Mientras que algoritmos como **inscriptis**, **Beautiful Soup** o **html_text** presentan unos resultados mediocres, el resto se acercan un poco más a lo que un usuario final pretendería obtener.

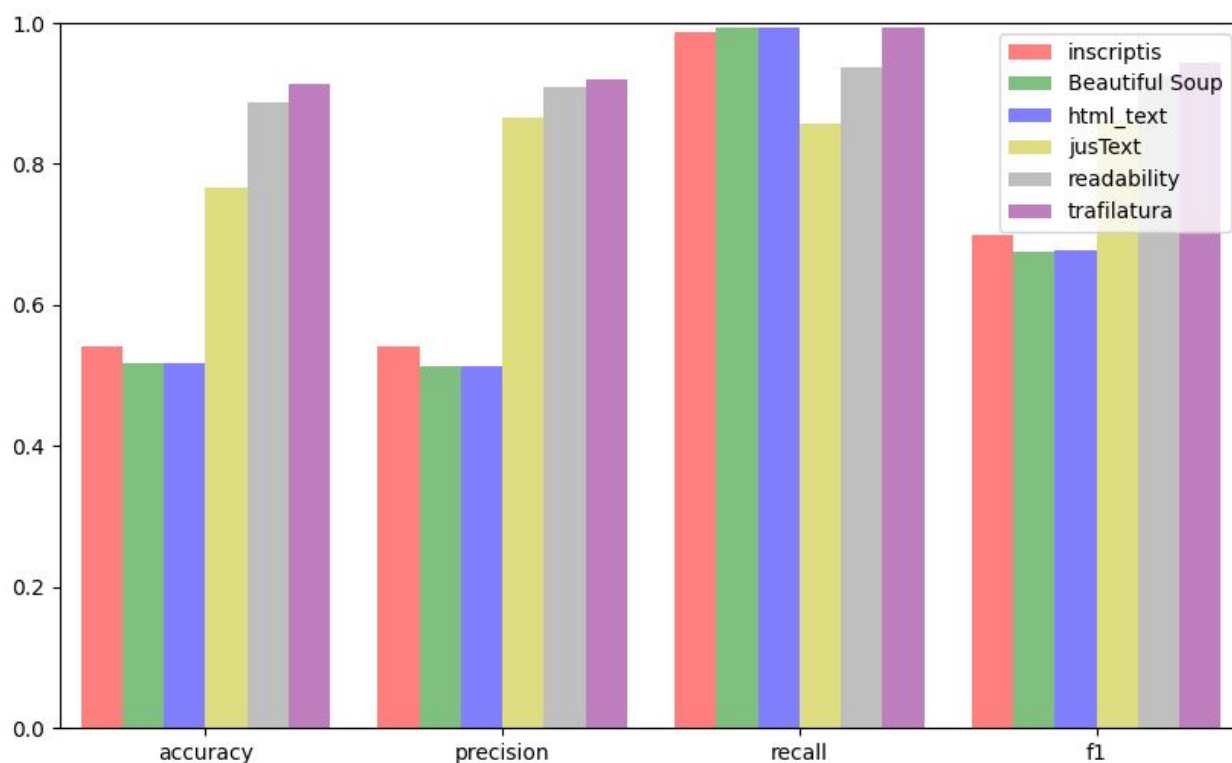


Figura 5.24: Gráfica - Comparación de resultados entre paquetes que usan *lxml* como analizador

La complejidad de cada algoritmo, también, se ve reflejada en el uso de recursos del medio de ejecución. Los dos algoritmos más complejos, computacionalmente, son los que más uso hacen de la CPU y que más tardan en obtener el resultado final.

Veamos por último aquellos paquetes que emplean *XML* o *xml2* como analizador. Ambas librerías emplean la jerarquía de clases para realizar un análisis. La principal diferencia entre los dos tiene que ver con la gestión de memoria.

Obsérvese la tabla 5.25, donde se aprecia la inmensa diferencia entre los tres paquetes. Mientras que **rvest** y **Rcrawler** presentan datos muy pobres, **boilerpipeR** retorna buenas métricas. Debemos mencionar también la multiplicidad de extractores que este último presenta, parte de su objetivo es el análisis de este tipo de documentos.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
rvest	0.1347	0.1371	0.8974	0.2378	44.1	8.9	60.3245
Rcrawler	0.4540	0.4628	0.9310	0.6181	46.7	3.4	158.0663
boilerpipeR	0.7830	0.8486	0.8696	0.8590	47.9	2.6	39.9543

Tabla 5.25: Tabla - Comparación de resultados entre paquetes que emplean *XML* o *xml2* como analizador

Aun recopilando menos cantidad de contenido principal que **rvest** o **Rcrawler**, **boilerpipeR** es capaz de obtener la información requerida de forma precisa. De hecho, la efectividad del algoritmo es asombrosa, solo hace falta ver la poca diferencia entre puntuación entre *recall* con el resto de métricas.

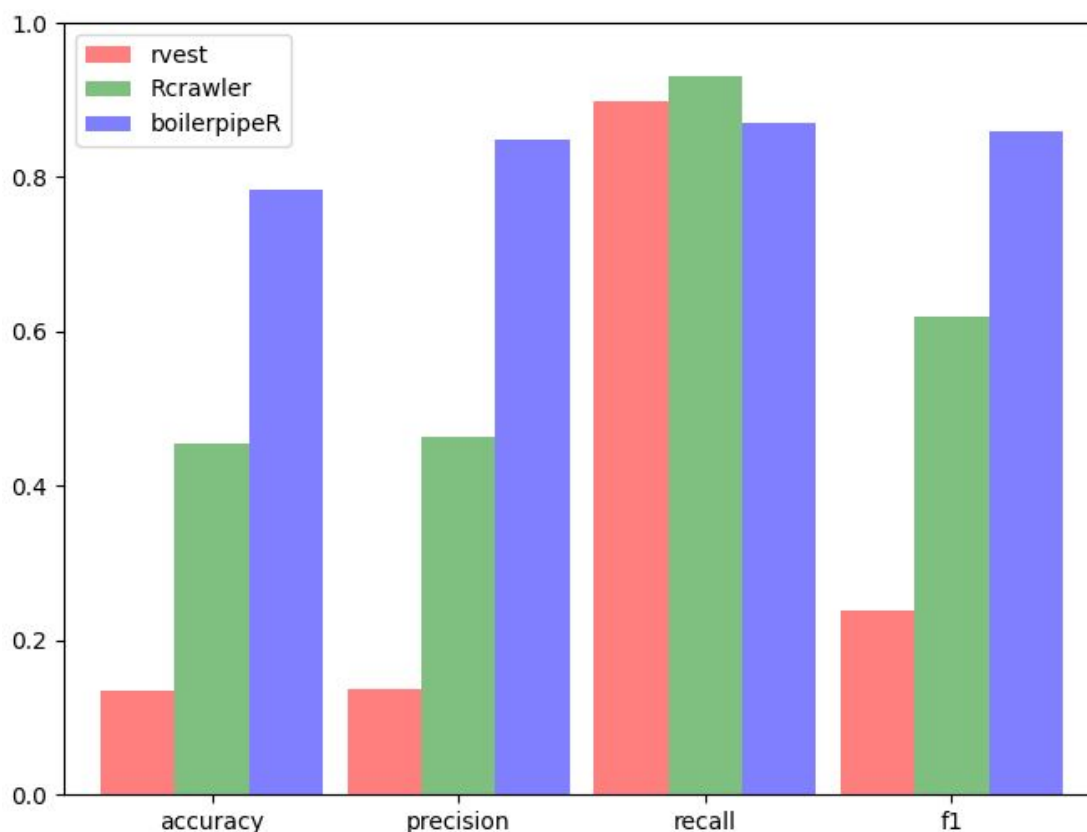


Figura 5.25: Gráfica - Comparación de resultados entre paquetes que usan *XML* o *xml2* como analizador

En cuanto al uso de recursos, *boilerpipeR* también muestra un uso más óptimo de los mismos. Aunque el tiempo de ejecución sigue siendo alto para lo que se pretende, es mucho menor que el de **rvest** o **Rcrawler**.

En la figura 5.26, se puede observar una comparación de paquetes sobre las variables *recall* y *precision*. Esta gráfica nos muestra la importancia de la heurística a la hora de eliminar contenido *boilerplate*. En este sentido, **Trafilatura** es capaz de extraer la mayor cantidad posible de información valiosa del texto principal de un documento HTML.

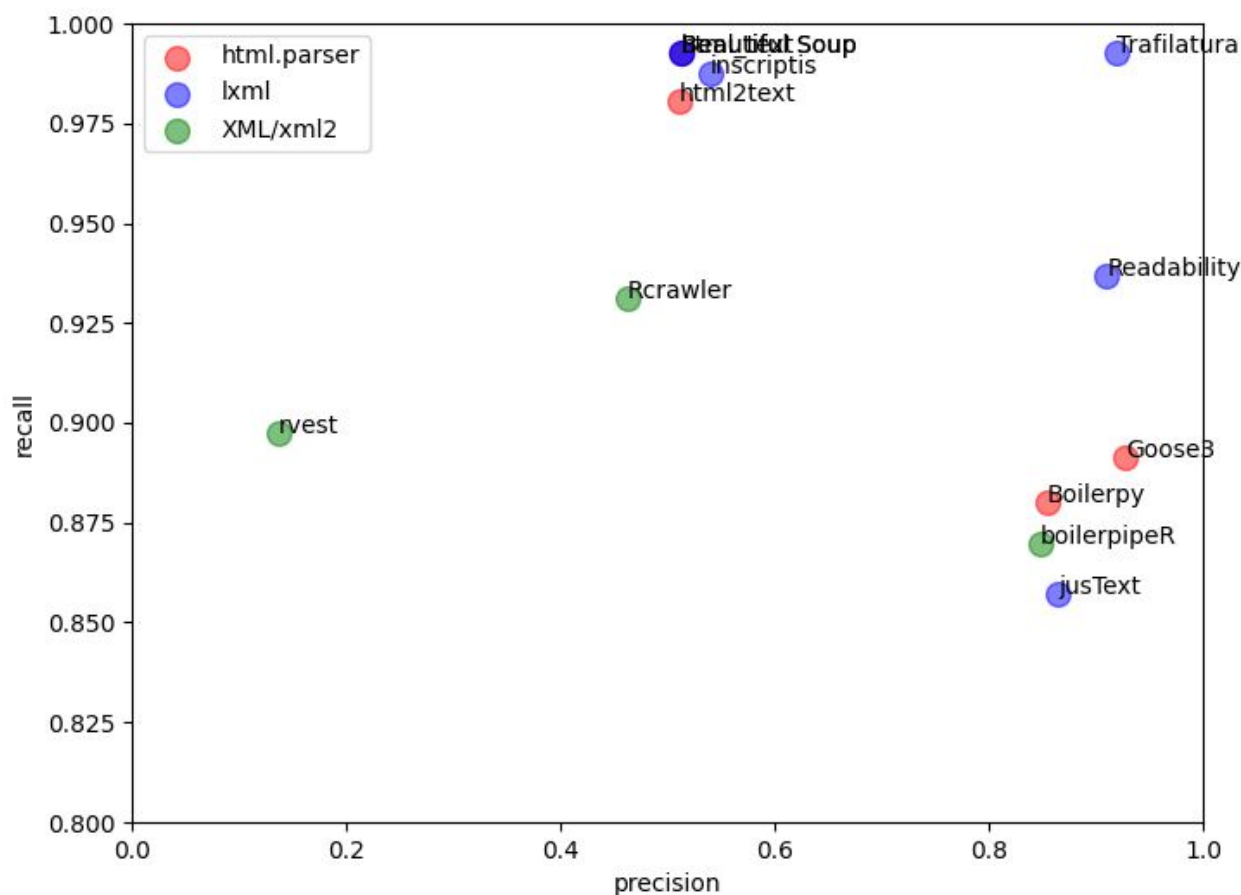


Figura 5.26: Comparación de la eficacia en los paquetes según el analizador empleado

A modo de conclusión, podemos determinar que el analizador es una herramienta fundamental en los algoritmos de *web scraping*, pero es necesaria una idea que los respalde. Que un paquete ofrezca mejores resultados que otro no dependerá del analizador empleado, sino del objetivo del propio paquete y de la implicación de su heurística en el mismo.

5.2.3 Comparación de paquetes según la heurística empleada

A lo largo del capítulo 3 vimos que algunas de las bibliotecas basaban parte de su heurística en otras bibliotecas relacionadas. En esta sección se realizará una comparación entre aquellos paquetes cuyo proceso de *web scraping* esté relacionado.

5.2.3.1 Comparación entre Readability y Goose3

Tanto **Readability** como **Goose3** emplean la heurística del nodo candidato para obtener la mayor cantidad de información posible. Ambos algoritmos recorren un árbol y puntúan cada nodo del mismo según la información contenida. Una vez puntuados los nodos, se seleccionan aquellos cuya puntuación es máxima.

El objetivo es evaluar y comparar los dos paquetes de *web scraping* de forma que podamos determinar aquel cuyo rendimiento es más sólido. Recordemos que ambos paquetes emplean diferentes analizadores, deberá ser algo a tener en cuenta.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Readability	0.8880	0.9101	0.9370	0.9233	45.3	1.6	3.5952
Goose3	0.8658	0.9273	0.8913	0.9089	32.2	6.1	25.9731

Tabla 5.26: Tabla - Comparación de resultados entre Readability & Goose3

En la tabla 5.26 se pueden observar los datos obtenidos para ambos paquetes. Los datos referentes a la calidad del texto extraído son muy similares, **Readability** aun capturando mayor cantidad de contenido principal, presenta mejores resultados que **Goose3**.

Por otro lado, en cuanto a las métricas dependientes del entorno de ejecución, **Readability** refleja unos datos mucho mejores que **Goose3**. La optimización de **Readability** es mucho mejor en términos de uso de CPU y tiempo de ejecución. Esto es sorprendente pues usan la misma heurística.

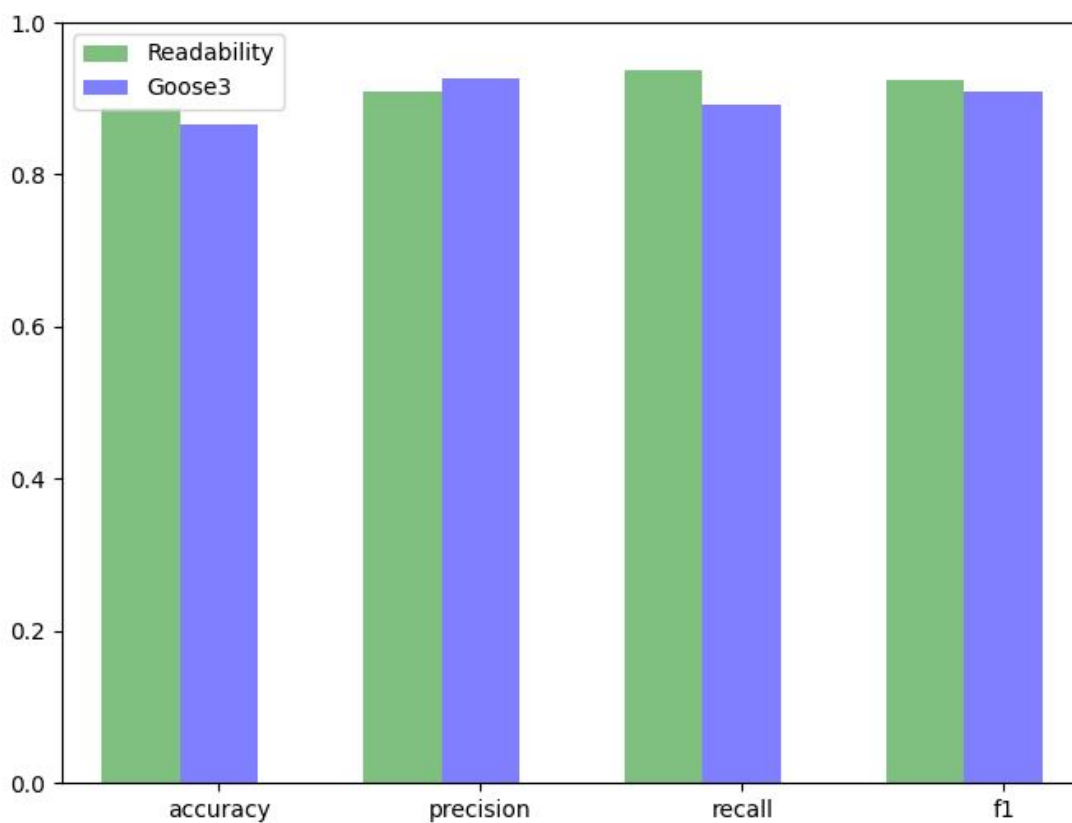


Figura 5.27: Figura - Comparación de resultados entre Readability & Goose3

5.2.3.2 Comparación entre Trafilatura, Readability y jusText

En una de las fases de ejecución de **Trafilatura**, este empleaba tanto **Readability** como **jusText** como algoritmos de respaldo. Ambos se encargaban de servidor de redes de seguridad y *fallbacks* en caso de la extracción previa resultase errónea. Una vez ejecutados los algoritmos, el resultado se comparaba con el inicial con el objetivo de determinar una solución final.

Para poder realizar una comparación justa, se ejecutará el algoritmo de **Trafilatura** con la opción de *fallbacks* activada. Esto forzará a que el algoritmo ejecute los algoritmos de respaldo.

Por otro lado, lo más lógico es pensar que **Trafilatura** va a mostrar mejores resultados tanto que **Readability** como **jusText** por el simple hecho de incluirlos dentro de su heurística. Se muestra en la tabla 5.27 los resultados obtenidos.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Trafilatura	0.9129	0.9208	0.9699	0.9447	45.9	1.4	4.4590
Readability	0.8880	0.9101	0.9370	0.9233	45.3	1.6	3.5952
jusText	0.7668	0.8649	0.8573	0.8610	45.1	0.5	2.9546

Tabla 5.27: Tabla - Comparación de resultados entre Trafilatura & Readability & jusText

Los resultados muestran como efectivamente **Trafilatura** presenta una mejor solución en calidad que el resto de algoritmos. Tanto la cantidad de contenido principal extraída, como la eliminación de contenido *boilerplate* es ligeramente mejor.

Por otro lado, lo más normal sería pensar en el gran costo de recursos de **Trafilatura** durante su ejecución. Lejos de ser así, presenta unos resultados francamente buenos con respecto a **jusText** o **Readability**. Tanto el tiempo de ejecución, como el uso de memoria, RAM y CPU son óptimos.

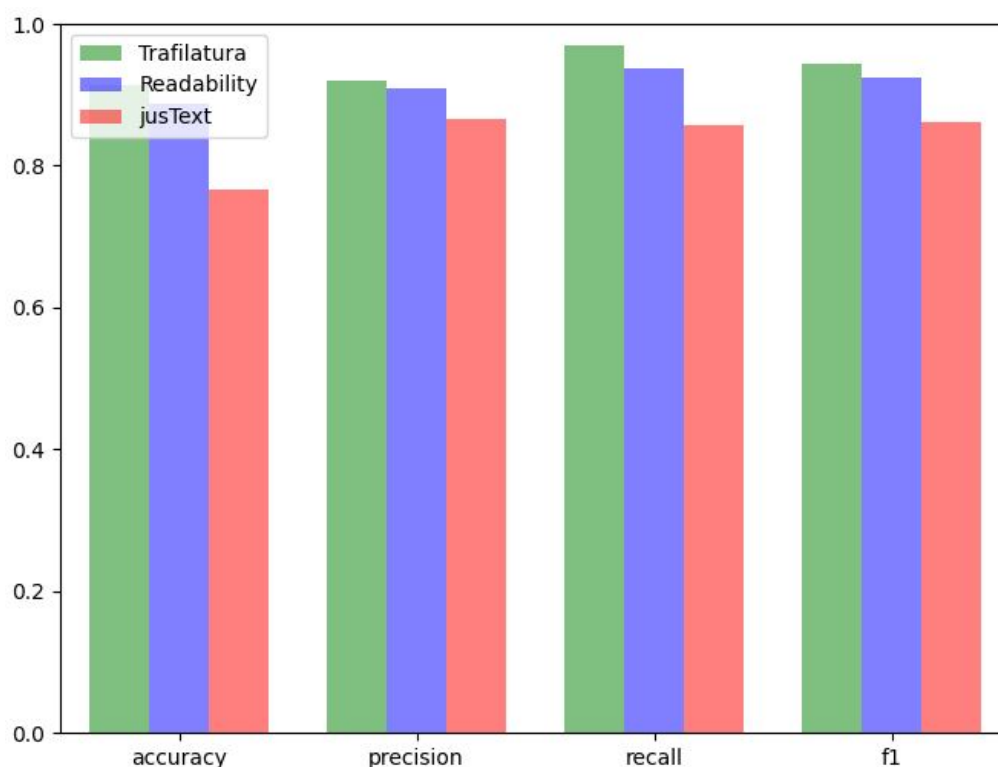


Figura 5.28: Figura - Comparación de resultados entre Trafilatura & Readability & jusText

A modo de conclusión, podemos determinar que **Trafilatura** es el mejor de los tres algoritmos. Tanto por la calidad ofrecida, como por la optimización del mismo.

5.2.3.3 Comparación entre BeautifulSoup y rvest

Tanto **rvest** como **Beautiful Soup** son dos de las herramientas de minado web más comunes en este ámbito. Es la propia heurística de **rvest** la que está basada en **Beautiful Soup** pues se incluyen aspectos

como la división de elementos a nivel de bloque y en línea. Será interesante ver los distintos resultados retornados por ambos paquetes.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Beau. Soup	0.5165	0.5129	0.9928	0.6764	42.0	1.2	3.1778
rvest	0.1347	0.1371	0.8974	0.2378	44.1	8.9	60.3245

Tabla 5.28: Tabla - Comparación de resultados entre Beautiful Soup & rvest

Se observa tanto en la tabla 5.28 como en la gráfica 5.29, los resultados retornados por ambos paquetes. Aunque ambos presentan una calidad del texto extraído que deja bastante que desear, los resultados no son nada parecidos.

Por un lado, respecto a las métricas independientes del entorno, ambos paquetes son capaces de recoger una gran cantidad de contenido principal. El problema viene cuando la heurística que presentan no es capaz de eliminar el contenido '*basura*' del mismo. Aunque los resultados de **Beautiful Soup** no son tan malos, siguen estando muy alejados de lo que pretende ser una herramienta de *web scraping* de calidad.

En cuanto a las métricas referentes a la optimización, si bien **Beautiful Soup** presenta unos datos bastante estándar, **rvest** es todo lo contrario. La optimización en cuanto a tiempo de ejecución y uso de CPU es bastante pobre en el paquete de R.

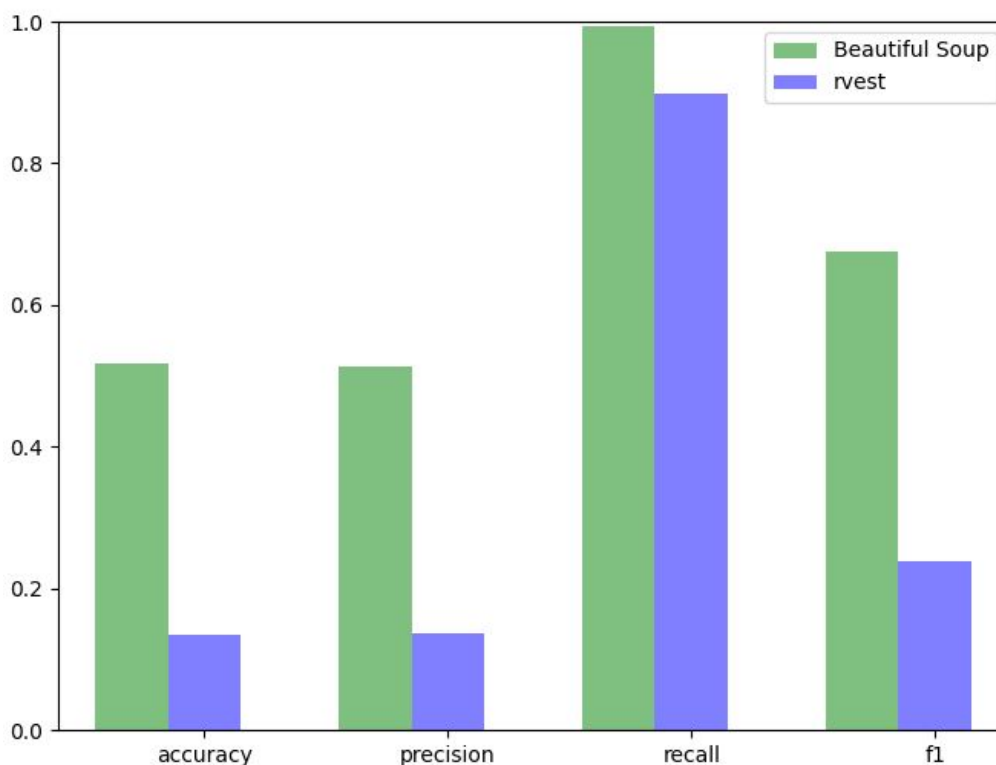


Figura 5.29: Figura - Comparación de resultados entre Beautiful Soup & rvest

A modo de conclusión, podemos destacar que aunque la idea de realizar una heurística similar por parte de **rvest** no es del todo mala, la ejecución de la misma deja bastante que desear. El propósito de ambos paquete es totalmente distinto y eso se ve reflejado en los resultados.

5.2.4 Comparación genérica de paquetes

Veamos por último una comparación general entre los resultados de todos los paquetes. Al finalizar la misma podremos determinar aquella librería más preparada para el proceso de *web scraping*. En primer lugar, podemos observar la tabla 5.29 con los resultados conjuntos de todos los paquetes.

Nombre	Accuracy	Precision	Recall	F1	RAM(%)	CPU(%)	Time Exec.(s)
Trafilatura	0.9129	0.9208	0.9699	0.9447	45.9	1.4	4.4590
Readability	0.8880	0.9101	0.9370	0.9233	45.3	1.6	3.5952
Goose3	0.8658	0.9273	0.8913	0.9089	32.2	6.1	25.9731
Boilerpy	0.7947	0.8544	0.8803	0.8672	43.9	1.9	2.5412
jusText	0.7668	0.8649	0.8573	0.8610	45.1	0.5	2.9546
boilerpipeR	0.7830	0.8486	0.8696	0.8590	47.9	2.6	39.9543
inscriptis	0.5414	0.5404	0.9875	0.6985	45.0	0.2	2.1009
html_text	0.5166	0.5130	0.9928	0.6765	44.9	0.5	1.1800
Beau. Soup	0.5165	0.5129	0.9928	0.6764	42.0	1.2	3.1778
html2text	0.5105	0.5107	0.9804	0.6715	44.6	1.8	4.4020
Rcrawler	0.4540	0.4628	0.9310	0.6181	46.7	3.4	158.0663
htm2txt	0.4547	0.4714	0.8885	0.6160	46.7	2.0	80.5288
XPath	0.2421	0.2401	0.9915	0.3866	44.4	2.0	0.7476
rvest	0.1347	0.1371	0.8974	0.2378	44.1	8.9	60.3245

Tabla 5.29: Tabla - Comparación de resultados entre algoritmos de *web scraping*

Se ha ordenado la tabla según la calidad del texto extraído resultante. En este caso, **Trafilatura** es el paquete cuya extracción resultante presenta una similitud más cercana a lo que un usuario vería en el documento HTML convencional. Véase 5.31.

Por otro lado, la optimización del mismo es relativamente buena comparada con la del resto. El tiempo de ejecución del mismo es relativamente corto y el uso de recurso es reducido. Relativo a este aspecto, los algoritmos propios de R generalmente presentan unos datos muy inferiores a los de Python en cuanto a tiempo de ejecución y empleo de CPU. Véase 5.32 y 5.30.

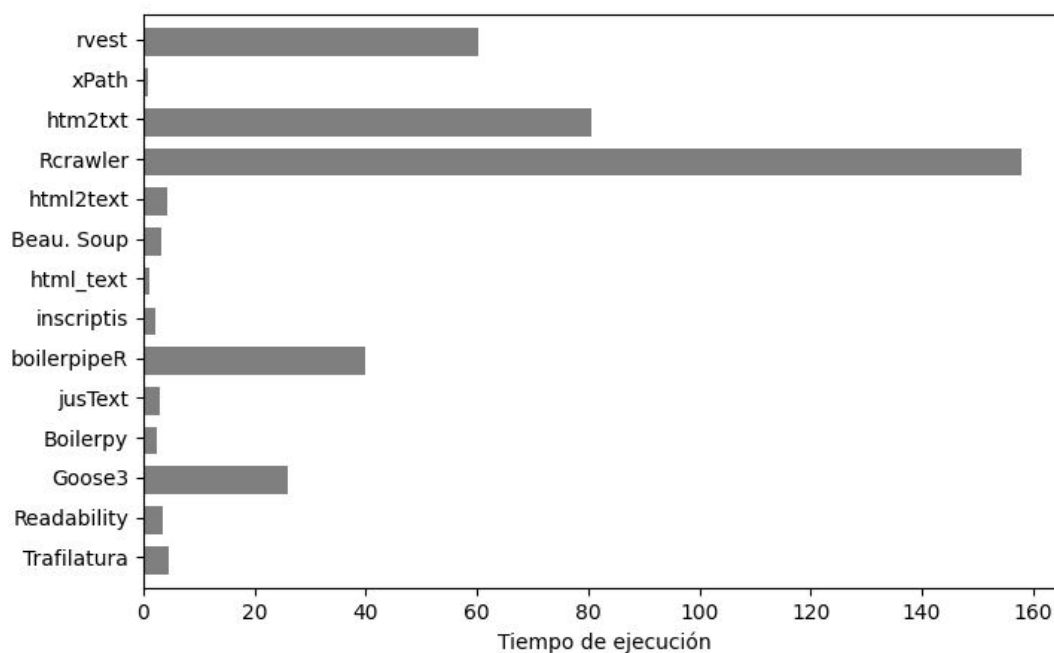


Figura 5.30: Comparación del tiempo de ejecución en los paquetes de *web scraping*

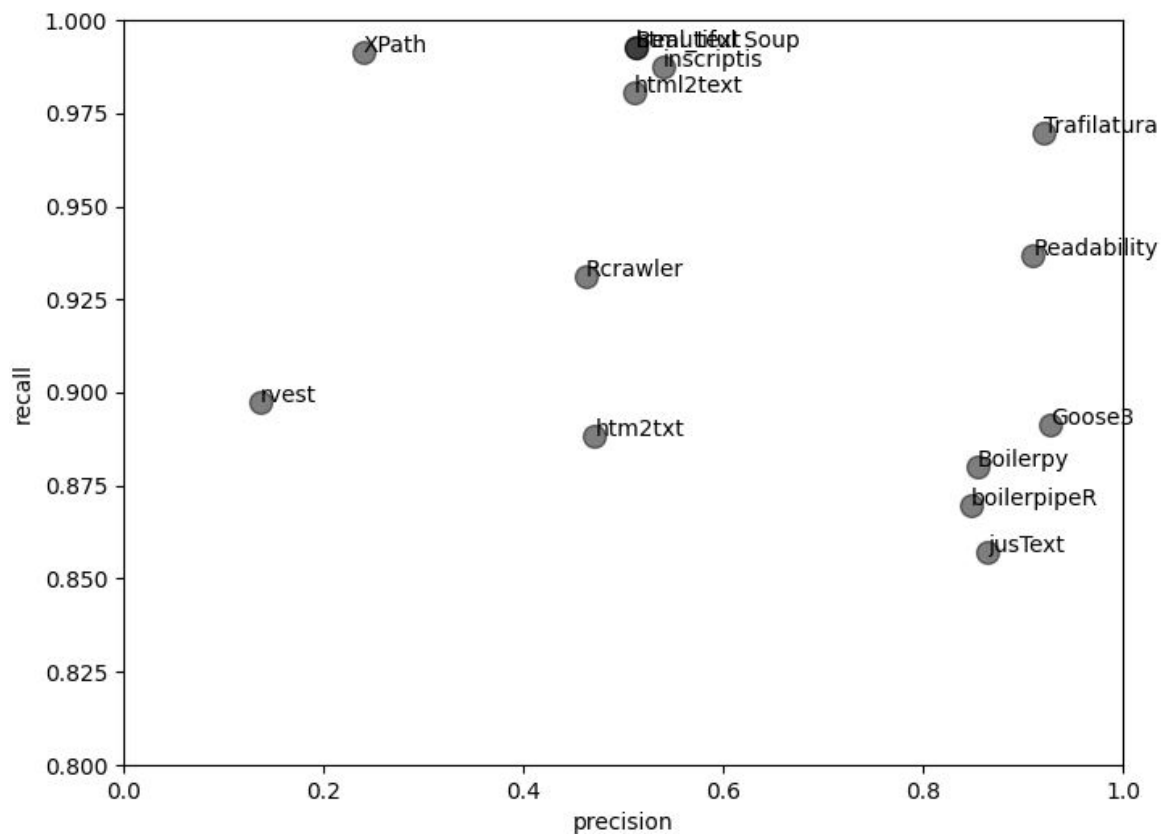


Figura 5.31: Comparación de la eficacia en los paquetes de *web scraping*

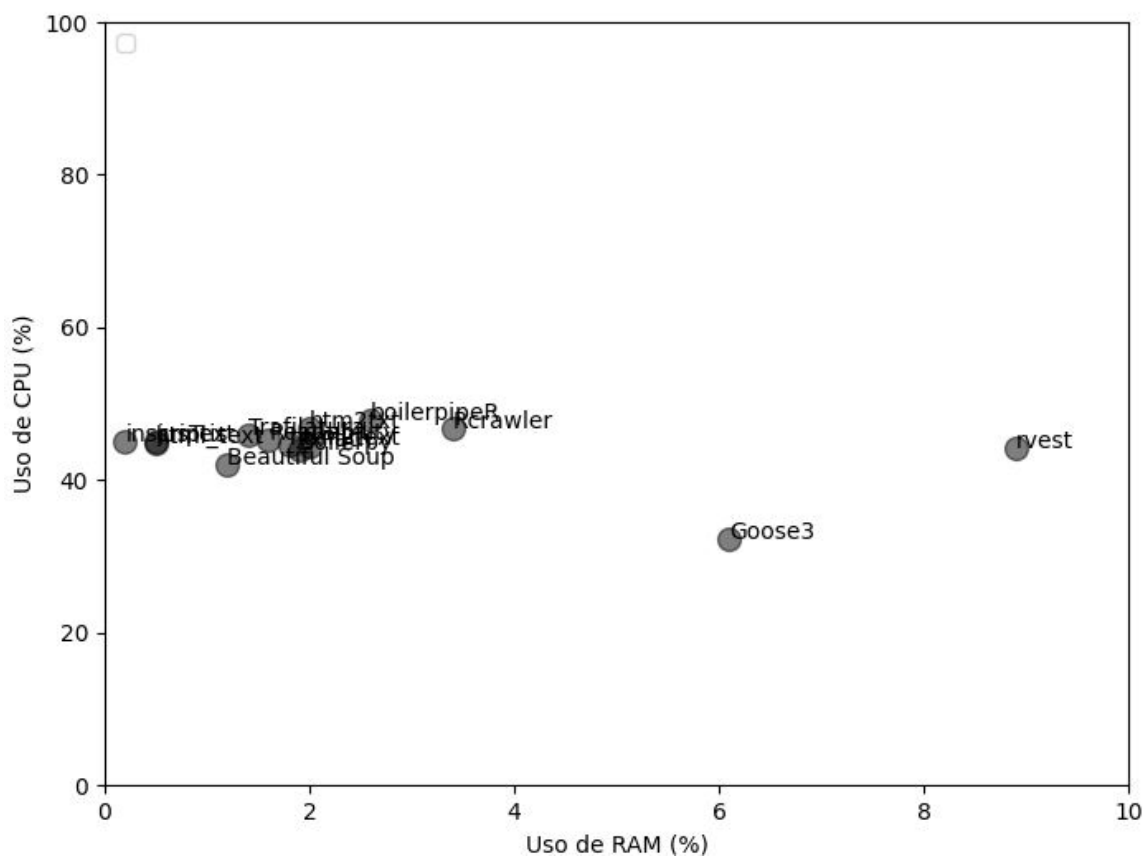


Figura 5.32: Comparación de la eficiencia en los paquetes de *web scraping*

Capítulo 6

Conclusiones y futuras líneas de trabajo

Una vez realizado el proceso de evaluación, es posible determinar una serie de conclusiones firmes sobre los resultados obtenidos:

- Aquellas librerías o paquetes cuya heurística está más desarrollada presentan unos resultados mejores que herramientas menos cuidadas. El simple uso de expresiones *XPath* dificulta un proceso de extracción de calidad.
- La calidad de la extracción, no dependen del analizador ni del tipo de lenguaje. La heurística y optimización del mismo son los únicos aspectos relevantes en este sentido.
- Además de la heurística, el objetivo que se quiera conseguir con el paquete es muy importante para obtener buenos resultados. Paquetes como **rvest** demuestran que el objetivo de algunas herramientas es simplemente ser capaces de acceder a ciertas etiquetas de documentos HTML.
- **Trafilatura** es el paquete cuya extracción resultante presenta una similitud más cercana a lo que un usuario vería en el documento HTML convencional.
- El uso de *web scraping* facilita y optimiza el uso tradicional de extracción y almacenado de información. Conseguir extraer el 94 % del contenido principal de 101 documentos HTML en tan solo 4.4590 segundos, hace que el empleo de herramientas similares a **Trafilatura** sean buenas opciones para sustituir la extracción tradicional.

Como futuras líneas de trabajo a seguir, podrían añadirse nuevas librerías o paquetes de minado web. Sería interesante la ampliación del proyecto a otros lenguajes de programación como Go, NodeJS, JavaScrip... Además, esto permitiría descubrir nuevas heurísticas que sean comparadas a las ya analizadas.

Otra posible mejora que interesante a introducir, sería la de ampliar la evaluación hacia nuevos idiomas en los documentos de prueba. Hasta ahora, las herramientas han sido evaluadas sobre documentos HTML escritos en inglés. Podría ser un aspecto positivo para la integridad de la evaluación, evaluar librerías de minado que extraigan texto de documentos escritos en diferentes idiomas.

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, “eatih, a simple tool used to extract an article’s text in html documents.” <http://rodricios.github.io/eatih/>. [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] S. Potter, “Introducing the selectr package,” The University of Auckland, Auckland, New Zealand, Tech. Rep., 2012. [Online]. Available: <http://stattech.wordpress.fos.auckland.ac.nz/2012-10-introducing-the-selectr-package/>
- [37] P. E. San, “Boilerplate removal and content extraction from dynamic web pages,” *International Journal of Computer Science*, vol. 4, no. 6, December 2014. [Online]. Available: <https://ssrn.com/abstract=3882476>
- [38] P. Jan, “Removing boilerplate and duplicate content from web corpora,” Master’s thesis, Masaryk University, Faculty of Informatics, 2011.
- [39] E. Persson, “Evaluating tools and techniques for web scraping,” Master’s thesis, School of Electrical Engineering and Computer Science, 2019.
- [40] L. Gautier, *rapy2-doc-zh CN Documentation*, 2011, 2.3.0-dev. [Online]. Available: https://buildmedia.readthedocs.org/media/pdf/rapy2-doc-zh_cn/latest/rapy2-doc-zh_cn.pdf
- [41] G. Rodola, *psutil Documentation*, 2022, release 5.9.1. [Online]. Available: https://psutil.readthedocs.io/_/downloads/en/latest/pdf/
- [42] E. Uzun, T. Yerlikaya, and O. Kirat, “Comparison of python libraries used for web data extraction,” vol. 24, pp. 87–92, 01 2018.
- [43] B. Indig, Á. Knap, Z. Sárközi-Lindner, M. Timári, and G. Palkó, “The elte.dh pilot corpus - creating a handcrafted gigaword web corpus with metadata,” in *WAC*, 2020.
- [44] 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>
- [45] 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>
- [46] 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>
- [47] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.
- [48] S. Behnel, M. Faassen, and I. Bicking, “lxml: Xml and html with python,” 2005.
- [49] J. Graham, S. Sneddon, and contributors, *html5lib Documentation*, 2020, release 1.1. [Online]. Available: https://html5lib.readthedocs.io/_/downloads/en/stable/pdf/
- [50] Python, “html parser - simple html and xhtml parser,” <https://docs.python.org/3/library/html.parser.html>. [Ultimo acceso 20/diciembre/2021].
- [51] 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* [44] 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* [45] 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* [46] 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, '.list-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 [47]. 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.

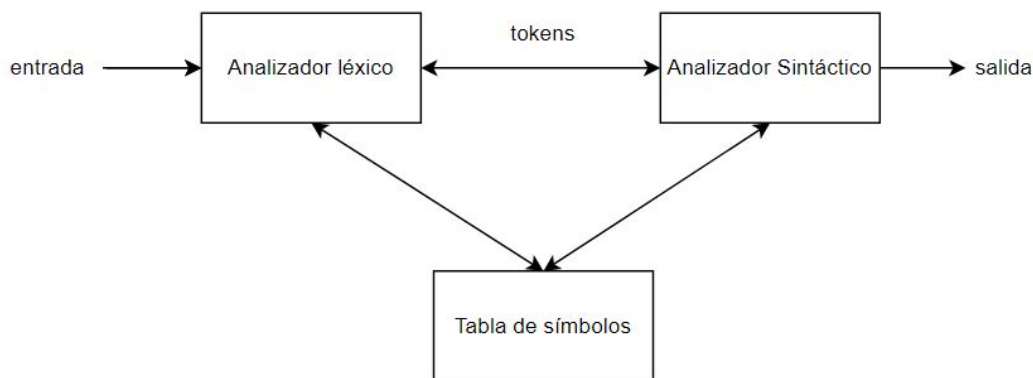


Figura B.1: Estructura básica de un analizador

B.1 lxml

Uno de los analizadores más comunes en todos los algoritmos de minado web es **`lxml`** [48]. 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** [48]. 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** [48] 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 [49] 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 [50] 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 [51] 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 examinar 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** [45]. 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.

Apéndice C

Métricas descartadas del proceso de evaluación

A lo largo del capítulo 4 se han definido diferentes métricas capaces de comparar algoritmos de minado web. En este apéndice se definen aquellos parámetros no incluidos en el clasificador. En la figura C.1 se muestra la relación entre variables y métricas. Se destacan, aquellas que han sido descartadas del proceso de evaluación.

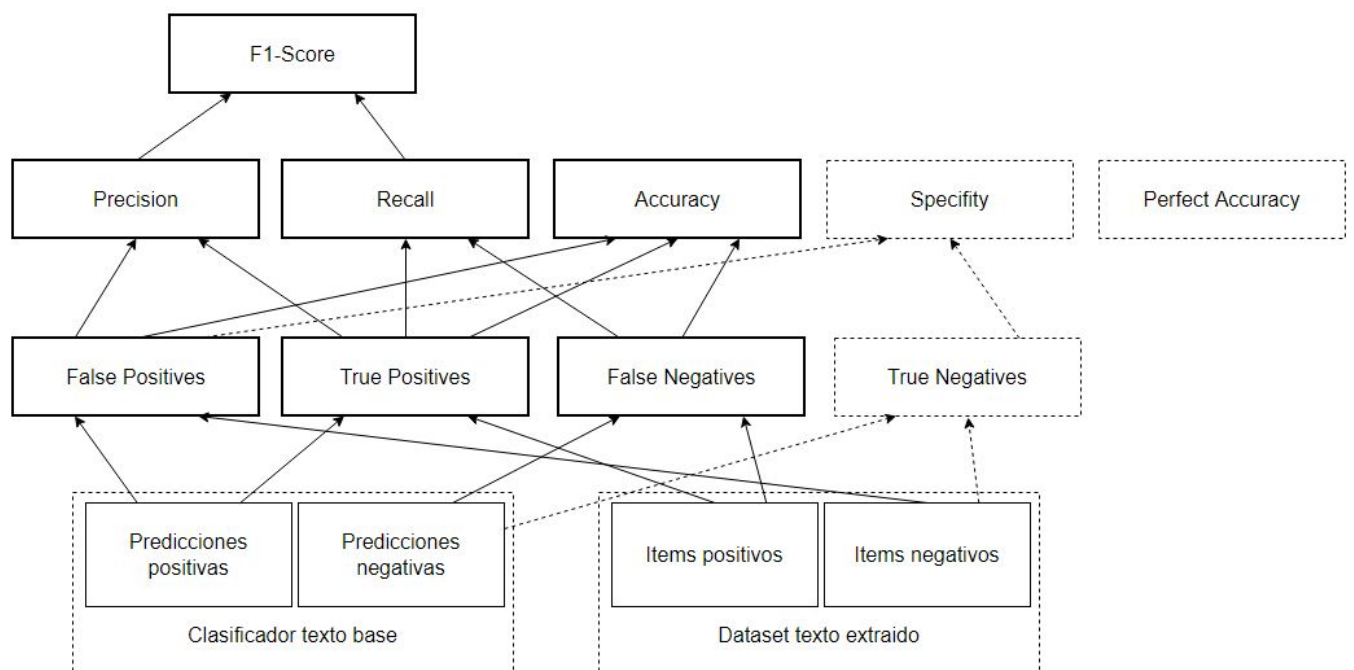


Figura C.1: Jerarquía de métricas descartadas

C.1 *Perfect accuracy*

Ya es sabido que la métrica *accuracy*, mide la proporción de predicciones correctas sobre el número total de predicciones, véase 4.9. Durante la búsqueda de una característica que calculase este número, se pensó en una idea feliz en la que se comparaba burdamente los tokens de los diferentes fragmentos de texto.

Esta métrica es fácil de entender pero bastante ruidosa, ya que una pequeña diferencia hace que toda la página sea un fracaso. Veamos un pequeño fragmento de código, para comprobar la forma en la que se podría incluir en nuestra evaluación.

Listado C.1: Cálculo de la métrica *perfect accuracy*

```
def calcular__perfect_accuracy(texto_base, texto_extraido):
    '''mide la proporción de predicciones perfectas de texto extraido'''
    return float(tokenizar(texto_base) == tokenizar(texto_extraido))
```

Se observa que la función convierte en tokens los fragmentos de texto correspondientes, para más tarde realizar la comparación pertinente. Este cálculo es muy similar al ya explicado en la sección 4.2.2.2, donde se comparaban textos a partir de los tokens de los mismos.

C.2 *Specificity*

La métrica conocida como *Specificity* mide la cantidad de predicciones negativas realizadas que son correctas. En otras palabras, mide como de mal a efectuado el minado web dicho algoritmo. La fórmula para su cálculo es:

$$Specificity = \frac{True\ Negatives}{True\ Negatives + False\ Positives} = \frac{N.\ of\ Correctly\ Predicted\ Negative\ Instances}{N.\ of\ Total\ Negative\ Instances\ in\ the\ Dataset}$$

Como se puede observar en el fragmento de código C.2, es necesario contemplar los verdaderos negativos como nueva variable. Los verdaderos negativos se producen cuando el clasificador ha predicho un resultado negativo, y el resultado real fue negativo.

Listado C.2: Cálculo de la métrica *specificity*

```
def calcular_specificity(tp, fp, fn, tn):
    '''mide la cantidad de predicciones negativas que en realidad son correctas'''
    if fp == 0 and fn == 0:
        return 1
    if tn == 0 and fp == 0:
        return 0
    return tp / (tn + fp)
```


Universidad de Alcalá
Escuela Politécnica Superior



ESCUELA POLITECNICA
SUPERIOR



Universidad
de Alcalá