CEU

MEMORIA PROYECTO

Aprendizaje Automático:

Predicción del Rendimiento Académico

Universidad CEU San Pablo

David Ruiz Luque
Alberto García Caballero
Ricardo Marín Fernández-Conde

15 de abril de 2025

Índice

Resumen

En esta práctica se ha desarrollado un sistema de extracción de información (web scraping) aplicado a documentos HTML, combinando herramientas modernas con modelos formales de la teoría de lenguajes. Para ello, se ha implementado un analizador léxico con PLY, basado en autómatas finitos deterministas (AFD), y se ha comparado su funcionamiento con la librería `BeautifulSoup`, ampliamente utilizada en entornos reales.

El objetivo principal ha sido explorar hasta qué punto los conceptos formales —como lenguajes regulares, expresiones regulares y gramáticas— pueden emplearse para resolver problemas prácticos. Además de implementar ambos enfoques, se ha desarrollado un sistema de validación cruzada y una batería de pruebas sobre archivos HTML de distinta complejidad, lo que ha permitido evaluar la robustez y precisión del sistema.

Abstract

This project presents an information extraction (web scraping) system applied to HTML documents, combining formal language theory with practical tools. A lexical analyzer was implemented using PLY, based on deterministic finite automata (DFA), and compared to the `BeautifulSoup` library, widely used in real-world applications.

The main objective was to explore the practical applicability of formal concepts —such as regular languages, regular expressions, and grammars— in solving real extraction tasks. Alongside the implementation of both methods, a cross-validation system and a set of tests over multiple HTML files were developed to evaluate robustness and accuracy.

1 Introducción

El web scraping es una técnica ampliamente utilizada para extraer información automáticamente de páginas web. Aunque en la práctica suele abordarse desde herramientas de alto nivel como `BeautifulSoup`, en esta memoria se ha querido enfocar el problema desde una perspectiva diferente: aplicar conceptos teóricos de la asignatura de Teoría de la Computación, como los autómatas finitos deterministas (AFD), para resolver el mismo problema.

La idea principal es construir autómatas que sean capaces de reconocer ciertas etiquetas HTML (concretamente `<a>` e `<img>`), y compararlos con un parser tradicional basado en el análisis del DOM. Esta aproximación permite comprobar de forma práctica hasta qué punto los modelos formales, vistos en un contexto teórico, pueden aplicarse a problemas del mundo real.

El proyecto se ha desarrollado en Python, haciendo uso de herramientas como `PLY` para implementar un lexer con expresiones regulares, y `Graphviz` para generar los grafos de los autómatas. Además, se han utilizado páginas HTML de prueba y se ha incluido un sistema de comparación entre métodos.

La práctica no solo ha servido para reforzar los conocimientos teóricos adquiridos en clase, sino también para experimentar con sus limitaciones y contrastarlos con enfoques modernos, valorando sus ventajas e inconvenientes.

2 Fundamentos teóricos

En esta sección se explican de forma sencilla los conceptos teóricos básicos que se han utilizado en la práctica. Nos centraremos en cómo los lenguajes regulares y los autómatas finitos deterministas (AFD) permiten detectar patrones en HTML, conectando así la teoría vista en clase con su aplicación práctica.

2.1 Lenguajes formales y AFD

Un **lenguaje formal** es un conjunto de cadenas sobre un alfabeto Σ. En esta práctica, las cadenas son fragmentos de código HTML y el alfabeto incluye caracteres como letras, números y símbolos especiales. Un lenguaje formal puede definirse mediante gramáticas o autómatas.
Entre los lenguajes formales, los más simples son los **lenguajes regulares**, los cuales pueden describirse mediante **expresiones regulares** y reconocerse utilizando autómatas finitos.
Un **autómata finito determinista** (AFD) es una 5-tupla:

$$A = (Q, \Sigma, f, q_0, F)$$

donde:

- Q es el conjunto finito de estados.
- Σ es el alfabeto de entrada.
- $f : Q \times \Sigma \to Q$ es la función de transición.
- $q_0 \in Q$ es el estado inicial.
- $F \subseteq Q$ es el conjunto de estados finales o de aceptación.

El autómata comienza en q_0 y avanza carácter a carácter según la función de transición f. Si al terminar la entrada está en un estado de F, la cadena se considera aceptada.

2.2 Tokens y analizadores léxicos

Un **token** es una unidad léxica que representa una estructura reconocible dentro del texto, como una palabra clave, un identificador o, en este caso, una etiqueta HTML.
Los analizadores léxicos (también llamados **lexers**) detectan estos tokens a partir de expresiones regulares. Cada expresión regular define un lenguaje regular, y por tanto puede ser representada por un AFD.

En esta práctica, se ha utilizado la herramienta `PLY` (Python Lex-Yacc) para construir un lexer personalizado que actúe como un autómata, escaneando el texto HTML en busca de tokens específicos como `<a href="...">` o `<img src="...">`.

2.3 Limitaciones del HTML como lenguaje

Aunque el análisis sintáctico del HTML puede parecer sencillo a primera vista, en realidad HTML **no es un lenguaje regular**. Esto se debe a que permite estructuras anidadas, por ejemplo:

```
<div><a href="...">Texto <strong>enlace</strong></a></div>
```

Reconocer correctamente este tipo de estructuras requiere una memoria adicional, como una pila, por lo que se necesita al menos un autómata a pila (AP).
Sin embargo, muchas tareas prácticas (como identificar etiquetas individuales bien formadas) pueden resolverse con expresiones regulares y AFDs. Por ello, esta práctica se centra en fragmentos simples de HTML, sin anidamiento.

3 Diseño del proyecto

Para organizar el trabajo, el proyecto se ha dividido en varias partes. Por un lado, se ha implementado un *lexer* con PLY, que actúa como un autómata para identificar patrones dentro del código HTML. Por otro lado, se ha desarrollado un parser con `BeautifulSoup`, una herramienta mucho más flexible y ampliamente utilizada en la actualidad. Finalmente, se ha añadido un sistema de comparación para analizar las diferencias entre ambos enfoques.

3.1 Estructura del proyecto

El proyecto se ha organizado en carpetas separadas para el código, los recursos de documentación y los archivos de salida:

- `mi_scraper/`: contiene todo el código Python, incluyendo el lexer, los autómatas, los scripts de test y el comparador.
- `mi_scraper/html_tests/`: incluye los archivos de prueba HTML sobre los que se realiza la extracción.
- `mi_scraper/output/`: resultados de las extracciones realizadas tanto con BeautifulSoup como con PLY.
- `memoria/`: se encuentra la memoria en LaTeX, junto con las imágenes generadas.

3.2 Resumen de scripts principales

`main.py`	Script principal que ejecuta el lexer y el parser, y genera los archivos de salida.
`lexer.py`	Define el lexer con PLY para reconocer etiquetas HTML específicas.
`parser.py`	Implementa el parser con PLY, que utiliza el lexer para extraer etiquetas.
`dom_parser.py`	Utiliza `BeautifulSoup` para extraer las mismas etiquetas.
`cross_test.py`	Compara los resultados obtenidos por ambos métodos.
`tests/test_cases.py`	Contiene funciones de prueba automatizadas sobre los HTML de ejemplo.

4 Web scraping con PLY (Lexer y Parser)

4.1 Explicación de PLY y su papel en la práctica

Para abordar el análisis léxico y sintáctico de páginas HTML se ha utilizado `PLY` (Python Lex-Yacc), una herramienta que implementa las funciones clásicas de `lex` y `yacc` dentro del lenguaje Python. Esta librería permite construir de forma declarativa un *lexer* que identifique tokens a partir de expresiones regulares, así como un *parser* capaz de procesar reglas sintácticas y detectar errores de estructura.

El lexer actúa como un autómata finito determinista (AFD) que recorre carácter a carácter el contenido de un archivo HTML, identificando fragmentos que coincidan con patrones definidos. Estos patrones se expresan mediante expresiones regulares que definen tokens.

Por otro lado, el parser trabaja sobre los tokens generados por el lexer y aplica reglas gramaticales para comprobar la estructura global del documento, como el anidamiento correcto de etiquetas. Esto permite detectar errores como etiquetas no cerradas o mal anidadas, los cuales no pueden ser reconocidos por un simple AFD.

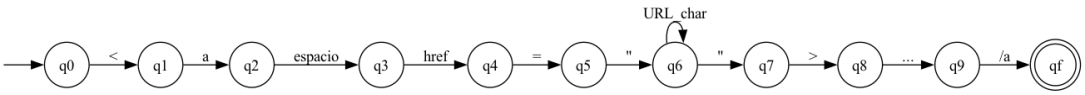
El uso de PLY en esta práctica permite ilustrar de forma práctica los conceptos de la asignatura, mostrando cómo los modelos teóricos como los autómatas y las gramáticas se pueden aplicar a tareas reales como el análisis de documentos HTML.

4.2 Diseño de los AFD

Para la parte léxica del proyecto se diseñaron dos autómatas finitos deterministas (AFD) que tienen como objetivo reconocer de forma precisa las etiquetas HTML más relevantes para esta práctica: los enlaces (`<a href="...">`) y las imágenes (`<img src="...">`).

Ambos autómatas se construyeron a partir de las expresiones regulares definidas en el lexer y se generaron gráficamente mediante la herramienta `Graphviz`. Estas representaciones permiten visualizar cómo el lexer transita entre estados según los caracteres de entrada y en qué condiciones acepta una cadena como válida.

AFD para etiquetas de enlace (`<a href="...">`):



Autómata finito determinista para la etiqueta `<a href="...">`.

Este autómata comienza en un estado inicial que solo acepta el símbolo <. A partir de ahí, verifica carácter a carácter que la secuencia corresponda con la estructura típica de una etiqueta de enlace que contenga el atributo `href`. Si en cualquier punto se encuentra un carácter inesperado, el autómata no puede avanzar y rechaza la entrada.

Cabe destacar que los estados trampa no han sido representados explícitamente en el gráfico ya que dificultan la visualización del mismo. Sin embargo, en la implementación real del lexer, estos estados son fundamentales para manejar entradas no válidas y evitar que el autómata continúe procesando cadenas incorrectas.

AFD para etiquetas de imagen (`<img src="...">`):

Autómata finito determinista para la etiqueta `<img src="...>`.

En este caso, el autómata valida que la cadena comience por `<img` y contenga el atributo `src`. A diferencia de la etiqueta `<a>`, no requiere una etiqueta de cierre, por lo que el autómata termina en un estado de aceptación tan pronto como se ha verificado la presencia del atributo y el cierre de la etiqueta.

Estos autómatas muestran cómo los lenguajes regulares permiten capturar patrones bien definidos dentro del HTML. Sin embargo, como veremos más adelante, no son capaces de validar estructuras más complejas como el anidamiento correcto de etiquetas, lo cual justifica el uso posterior de un parser.

4.3 Implementación del `lexer.py`

El archivo `lexer.py` define el analizador léxico mediante la librería `PLY`. Este lexer está diseñado para escanear archivos HTML e identificar tokens específicos: principalmente etiquetas de enlace y de imagen.

Cada token se define mediante una expresión regular asociada a una función. Cuando el texto de entrada coincide con la expresión, se genera un token correspondiente. Estos tokens son posteriormente utilizados tanto para la extracción directa como para el análisis sintáctico con el parser.

A continuación se muestran las definiciones clave del lexer:

```
# Token para enlaces con href
@lex.TOKEN(r'<a\s+href\s*=\s*"[^"]*"[^>]*>')
def t_A_TAG(t):
    return t
```

Este patrón reconoce etiquetas <a> con atributo href, permitiendo cierto margen de espacios y otros atributos adicionales.

```
# Token para imágenes con src
@lex.TOKEN(r'<img\s+src\s*=\s*"[^"]*"[^>]*>')
def t_IMG_TAG(t):
    return t
```

De forma análoga, este token detecta etiquetas <img> que incluyan el atributo src. No se exige un cierre explícito de la etiqueta, ya que en HTML es común que estas aparezcan como self-closing.

Además de estos tokens específicos, el lexer también define reglas para detectar etiquetas de apertura y cierre genéricas:

```
# Etiquetas de apertura
@lex.TOKEN(r'<([a-zA-Z]+)(\s[^<>]*)?>')
def t_OPEN_TAG(t):
    t.value = t.value[1:].split()[0].lower()
    return t

# Etiquetas de cierre
@lex.TOKEN(r'</([a-zA-Z]+)\s*>')
def t_CLOSE_TAG(t):
    t.value = t.value[2:-1].strip().lower()
    return t
```

Estas reglas permiten capturar otras etiquetas útiles para el análisis estructural, como <body>, <p>, , etc. Su función principal es apoyar al parser en la detección de estructuras bien formadas o errores de balanceo.

En conjunto, el archivo lexer.py actúa como un autómata que recorre el documento HTML y convierte su contenido en una secuencia de tokens, simplificando el análisis posterior.

4.4 Implementación del parser.py

El archivo parser.py define un analizador sintáctico utilizando la parte yacc de la librería PLY. Su propósito es validar la estructura general de un documento HTML, comprobando si las etiquetas están bien anidadas y correctamente balanceadas.

Este parser opera sobre los tokens generados previamente por el lexer. Para ello, se definen una serie de reglas gramaticales recursivas que especifican cómo deben organizarse los elementos del HTML.

A continuación se muestra un ejemplo simplificado de estas reglas:

```
def p_document(p):
    'document : elements'
    pass

def p_elements(p):
    '''elements : element elements
                | element'''
    pass

def p_element(p):
    '''element : A_TAG
               | IMG_TAG
               | OPEN_TAG elements CLOSE_TAG'''
```

```
    if len(p) == 4 and p[1] != p[3]:
        print(f"Etiqueta no balanceada: <{p[1]}>...</{p[3]}>")
```

La última regla es la más importante: permite validar que una etiqueta de apertura y su correspondiente etiqueta de cierre coincidan. Si no lo hacen, se imprime un mensaje de error indicando el desbalanceo detectado.

Además, se incluye una función para capturar errores de sintaxis generales, como etiquetas mal formadas o fuera de lugar:

```
def p_error(p):
    if p:
        print(f"Error de sintaxis en: {p.value}")
    else:
        print("Error de sintaxis al final del input")
```

El parser es compilado con la siguiente instrucción:

```
parser = yacc.yacc()
```

Esta llamada genera automáticamente los archivos `parsetab.py` y `parser.out`, que contienen las tablas de análisis y los estados del autómata construido internamente.

Gracias a este componente, el sistema es capaz de detectar estructuras HTML incorrectas que no podrían ser identificadas por el lexer, como etiquetas abiertas que no se cierran o anidamientos incorrectos.

4.5 Manejo de etiquetas mal formadas y estructura HTML

Uno de los principales retos de esta práctica es que el lenguaje HTML, aunque simple en apariencia, permite estructuras anidadas y una gran variedad de formas sintácticas. Esto lo convierte en un lenguaje que no es regular, por lo que no puede ser reconocido completamente por un autómata finito determinista (AFD).

El **lexer** basado en expresiones regulares funciona correctamente para detectar etiquetas bien formadas de tipo `<a href="...">` y `<img src="...">`. Sin embargo, no puede validar si estas etiquetas están correctamente anidadas ni si el documento HTML está balanceado estructuralmente. Esta limitación es esperada, ya que los AFD carecen de memoria para llevar seguimiento del contexto o del número de aperturas y cierres de etiquetas.

Para resolver este problema, se desarrolló un **parser sintáctico** en `parser.py`, que permite analizar la estructura jerárquica del HTML mediante reglas recursivas. Gracias a este componente, es posible detectar errores comunes como:

- Etiquetas abiertas que no se cierran correctamente.
- Anidamientos inválidos, como cerrar una etiqueta `<div>` con `</a>`.
- Cierre de etiquetas fuera de orden.

Durante las pruebas, se comprobaron múltiples archivos HTML mal formados (ver Sección 7.4), y se observó que el parser era capaz de generar mensajes de error detallados para cada una de estas situaciones, mientras que el lexer las pasaba por alto si la etiqueta individual era sintácticamente válida.

Esta combinación de lexer + parser permite simular un comportamiento similar al de un parser real de HTML, pero desde una perspectiva teórica basada en los conceptos vistos en la asignatura: **autómatas finitos** y **gramáticas independientes del contexto**.

5 Web scraping con BeautifulSoup (DOM)

5.1 Qué es BeautifulSoup y cómo se usa

`BeautifulSoup` es una librería de Python ampliamente utilizada para realizar tareas de web scraping. A diferencia del enfoque basado en autómatas y expresiones regulares, esta herramienta trabaja a un nivel más alto, interpretando el documento HTML como una estructura jerárquica conocida como **DOM** (Document Object Model).

Una vez cargado el documento, `BeautifulSoup` permite acceder a sus elementos mediante funciones como `find`, `find_all` o búsquedas por atributos. Esto hace que sea extremadamente útil para extraer datos estructurados, incluso cuando el HTML está malformado o contiene errores.

En el contexto de esta práctica, **BeautifulSoup se ha utilizado como método alternativo** al lexer+parser, con el objetivo de:

- Extraer todas las etiquetas `<a>` que contengan un atributo `href`.
- Extraer todas las etiquetas `<img>` con atributo `src`.
- Contar la frecuencia de aparición de otras etiquetas relevantes del HTML.

Este enfoque sirve como referencia para validar los resultados del lexer y del parser, y también permite comprobar qué ventajas ofrece una librería de propósito general frente a una solución construida desde los fundamentos teóricos de los lenguajes formales.

5.2 Implementación del `dom_parser.py`

El archivo `dom_parser.py` implementa el análisis del documento HTML utilizando la librería `BeautifulSoup`. A diferencia del lexer basado en expresiones regulares, esta herramienta convierte automáticamente el código HTML en un árbol DOM, lo que facilita enormemente la navegación por sus elementos.

La implementación comienza cargando el contenido del archivo HTML y creando un objeto `BeautifulSoup` que representa su estructura:

```
with open(html_file, encoding='utf-8') as f:
    html = f.read()

soup = BeautifulSoup(html, 'html.parser')
```

Una vez cargado, se realiza la extracción de enlaces e imágenes de forma muy directa, filtrando aquellas etiquetas que contengan los atributos deseados:

```
# Extraer enlaces
links = [a['href'] for a in soup.find_all('a', href=True)]

# Extraer imágenes
images = [img['src'] for img in soup.find_all('img', src=True)]
```

Además de la extracción básica, el script cuenta el número de apariciones de otras etiquetas comunes como `<div>`, `<p>`, `<span>`, `<table>`, entre otras, y muestra un resumen por consola:

```
tag_stats = {tag: len(soup.find_all(tag)) for tag in tags_to_check}

for tag, count in tag_stats.items():
    print(f"{tag}: {count}")
```

Finalmente, los enlaces e imágenes extraídos se guardan en archivos de texto dentro de la carpeta `output/`, lo que permite comparar los resultados con los obtenidos mediante PLY:

```
Path("output/links_bs.txt").write_text("\\n".join(links))
Path("output/images_bs.txt").write_text("\\n".join(images))
```

Este enfoque destaca por su sencillez y robustez, ya que `BeautifulSoup` es capaz de tolerar errores de sintaxis en el HTML y continuar el análisis sin interrupciones. Esto contrasta con el comportamiento más estricto del parser implementado con PLY.

5.3 Extracción de etiquetas y estadísticas

Además de identificar enlaces e imágenes, el script `dom_parser.py` genera un pequeño análisis estadístico del documento HTML. Esto se realiza contando la cantidad de veces que aparecen determinadas etiquetas de interés en el archivo de entrada.

Las etiquetas analizadas incluyen tanto las relevantes para la extracción de contenido (`<a>`, `<img>`) como otras etiquetas HTML comunes:

```
tags_to_check = ["a", "img", "br", "div", "li", "ul",
                 "p", "span", "table", "td", "tr"]
```

Se construye un diccionario que asocia cada etiqueta con el número de apariciones encontradas por `BeautifulSoup`:

```
tag_stats = {tag: len(soup.find_all(tag)) for tag in tags_to_check}
```

Posteriormente, el script imprime los resultados en consola, lo que permite tener una visión general del contenido HTML:

```
print("\\n Estadísticas de etiquetas HTML:")
for tag, count in tag_stats.items():
    print(f"{tag}: {count}")
```

Este análisis es útil para validar los archivos de prueba utilizados y observar cómo varían según el caso. También ayuda a justificar las diferencias observadas en el comportamiento de los analizadores PLY y BS4, especialmente cuando se trabaja con HTML mal formado o documentos de estructura compleja.

En resumen, la inclusión de estadísticas no solo permite comprobar que el análisis se realiza correctamente, sino que también facilita la interpretación de los resultados en la comparativa entre métodos.

5.4 Ventajas y flexibilidad frente a PLY

El enfoque basado en `BeautifulSoup` presenta una serie de ventajas prácticas frente al análisis con PLY, especialmente cuando se trabaja con archivos HTML reales, que frecuentemente contienen errores de sintaxis, etiquetas mal cerradas o estructuras anidadas complejas.

A diferencia del lexer y parser implementados con PLY, que requieren que el HTML tenga una estructura bien definida para poder ser analizado correctamente, `BeautifulSoup` incluye mecanismos internos de

corrección y recuperación de errores. Esto le permite analizar documentos imperfectos sin generar interrupciones ni fallos de ejecución.

Algunas de las principales ventajas observadas son:

- **Tolerancia a errores**: es capaz de interpretar HTML mal formado, insertando automáticamente etiquetas faltantes o corrigiendo jerarquías inválidas.
- **Simplicidad de uso**: permite acceder directamente a las etiquetas mediante métodos como `find_all`, sin necesidad de definir expresiones regulares ni reglas gramaticales.
- **Versatilidad**: soporta búsquedas por nombre de etiqueta, atributos, clases CSS, texto contenido y más.

No obstante, estas ventajas vienen con una contrapartida: al abstraer los detalles del proceso de análisis, `BeautifulSoup` oculta el funcionamiento interno y no permite observar directamente cómo se realiza el reconocimiento de patrones, lo que puede dificultar su uso con fines educativos o de validación formal.

Por el contrario, el enfoque con PLY, aunque más estricto y sensible a errores, permite comprender en profundidad cómo funcionan los analizadores léxicos y sintácticos, mostrando de forma explícita los errores encontrados y ofreciendo una experiencia más cercana a los conceptos teóricos vistos en la asignatura.

Por tanto, ambos enfoques son complementarios: mientras que PLY proporciona una base sólida para comprender el funcionamiento interno del análisis de lenguajes, `BeautifulSoup` ofrece una solución eficaz para trabajar con documentos HTML en entornos reales.

6 CrossTest: Comparación de resultados

6.1 Explicación de `cross_test.py`

Para validar la consistencia entre los resultados obtenidos con PLY y los obtenidos con `BeautifulSoup`, se ha implementado un script llamado `cross_test.py`. Este componente compara los archivos de salida generados por ambos métodos, tanto para enlaces como para imágenes.

La lógica del script es sencilla: abre los archivos `links_ply.txt`, `links_bs.txt`, `images_ply.txt` e `images_bs.txt`, y compara sus contenidos. Si ambas listas coinciden exactamente (sin importar el orden), se considera que el análisis ha sido consistente.

A continuación se muestra un fragmento relevante del código:

```
def comparar_resultados(archivo1, archivo2):
    conjunto1 = set(Path(archivo1).read_text().splitlines())
    conjunto2 = set(Path(archivo2).read_text().splitlines())
    return conjunto1 == conjunto2
```

Posteriormente, se llama a esta función para comparar los enlaces y las imágenes extraídos por ambos métodos:

```
links_ok = comparar_resultados("output/links_ply.txt", "output/links_bs.txt")
imgs_ok = comparar_resultados("output/images_ply.txt", "output/images_bs.txt")
```

En función del resultado, el script muestra un mensaje indicando si hay coincidencia o no:

```
if links_ok:
    print("Enlaces coinciden")
else:
    print("DIFERENCIA en enlaces")
```

Este comparador resulta especialmente útil para evaluar la fiabilidad del lexer y parser implementados con PLY. Si ambos métodos coinciden en todos los casos, se puede concluir que el reconocimiento realizado por los autómatas es correcto, al menos en lo que respecta a las etiquetas objetivo de esta práctica.

6.2 Validación cruzada entre PLY y BS4

Tras la implementación de ambos analizadores y el desarrollo del script de comparación, se procedió a validar su comportamiento mediante múltiples archivos de prueba. En todos los casos se analizó la extracción de enlaces e imágenes, contrastando los resultados obtenidos por PLY (lexer + parser) y por `BeautifulSoup`.

La validación cruzada reveló un comportamiento muy positivo: en todos los archivos de prueba, los resultados fueron coincidentes. Esto sugiere que, a pesar de sus limitaciones estructurales, el lexer y parser diseñados con PLY fueron capaces de identificar correctamente las etiquetas HTML objetivo y extraer la misma información que un analizador de propósito general como `BeautifulSoup`.

Además, se observó que incluso en documentos HTML mal formados (con etiquetas sin cerrar, estructuras desordenadas o jerarquías incorrectas) ambos métodos lograron recuperar la misma información. Esto refuerza la idea de que el reconocimiento de patrones individuales, como las etiquetas `<a href="...">` o `<img src="...">`, puede abordarse con éxito utilizando técnicas formales basadas en autómatas finitos.

No obstante, también se evidenció que el parser implementado con PLY resulta más estricto y menos tolerante a errores, ya que emite advertencias cuando las etiquetas están mal balanceadas. En cambio, `BeautifulSoup` ignora esos problemas y continúa el análisis normalmente, lo cual puede ser útil en entornos productivos pero menos formativo desde un punto de vista académico.

En definitiva, esta validación cruzada permite confiar en la efectividad del sistema desarrollado y demuestra que los modelos teóricos aplicados en este proyecto tienen un alcance práctico significativo.

6.3 Comentarios sobre resultados consistentes

El análisis de los archivos de prueba reveló una alta consistencia entre los resultados obtenidos por los dos métodos implementados: el lexer y parser construidos con PLY, y el analizador de DOM basado en `BeautifulSoup`. A continuación se destacan algunas observaciones relevantes que permiten interpretar mejor estos resultados:

- En todos los casos, tanto PLY como BeautifulSoup identificaron el mismo número de enlaces e imágenes, incluso en documentos mal estructurados. Esto refuerza la fiabilidad del enfoque basado en autómatas para reconocer patrones léxicos específicos.
- En archivos como `prueba3.html` y `prueba6.html`, que presentan etiquetas mal cerradas, anidamientos incorrectos o errores de sintaxis, el parser detectó múltiples errores, pero aún así se logró extraer correctamente la información deseada.
- El sistema de comparación no sólo valida la coincidencia de los datos, sino que también actúa como una verificación cruzada: si un método produce un resultado diferente, el error puede detectarse rápidamente revisando ambos archivos de salida.

- En el caso del lexer con PLY, el número de enlaces e imágenes coincide siempre que las etiquetas estén correctamente formadas en su versión básica. En cambio, BeautifulSoup puede extraer información útil incluso de etiquetas incompletas, lo cual puede considerarse tanto una ventaja como una fuente potencial de ruido.
- La ejecución repetida del script `run_all.sh` sobre cada archivo de prueba mostró que el comportamiento era estable, consistente y predecible, lo que demuestra que el sistema es robusto ante entradas diversas.

Estas observaciones confirman que el diseño del sistema ha sido eficaz a la hora de combinar la precisión del análisis léxico con la flexibilidad del análisis estructural, ofreciendo una herramienta sólida para tareas de extracción de información en HTML.

6.4 Tabla de cobertura de pruebas (PLY vs BS4)

A continuación se muestra la tabla revisada que resume el comportamiento de ambos métodos para distintos archivos HTML:

Archivo	Enl. PLY	Img. PLY	Balanceo HTML (parser)	Enl. BS4	Img. BS4	Links	Imgs
`prueba1.html`	0	0	Error de sintaxis (falta apertura)	0	0	✓	✓
`prueba2.html`	1	1	 mal cerrado dentro de <a>	1	1	✓	✓
`prueba3.html`	2	1	Varias etiquetas no balanceadas (<area>)	2	1	✓	✓
`prueba4.html`	1	1	<p> y <body> sin cierre correcto	1	1	✓	✓
`prueba5.html`	1	1	<a> mal anidado dentro de <p>	1	1	✓	✓
`prueba6.html`	48	14	Múltiples errores de anidamiento y sintaxis	48	14	✓	✓

Resultados de la comparación entre PLY y BeautifulSoup.

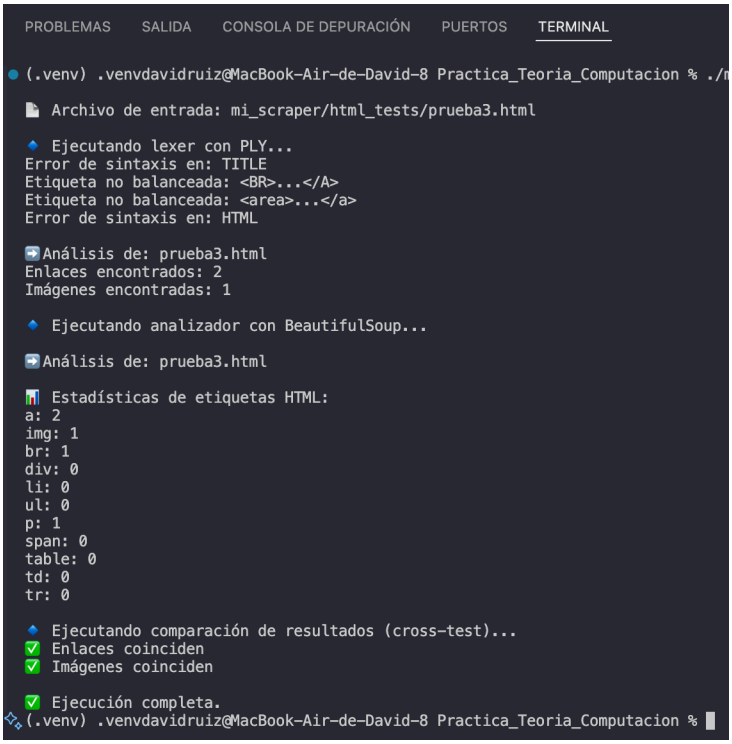
Como puede observarse, todos los archivos presentan problemas de estructura HTML que son detectados por el parser con PLY. Sin embargo, tanto el lexer como el analizador con `BeautifulSoup` son capaces de extraer correctamente los enlaces e imágenes previstos. Las comprobaciones cruzadas confirman la coincidencia de resultados, validando la consistencia entre ambos enfoques.

7 Pruebas de ejecución y resultados

En esta sección se muestran los resultados prácticos obtenidos al ejecutar el sistema completo sobre distintos archivos HTML de prueba. Para facilitar la ejecución, se desarrolló un script llamado `run_all.sh` que automatiza el análisis completo con PLY, BeautifulSoup y la comparación cruzada.

7.1 Ejecución del sistema completo

A continuación se muestra un ejemplo de ejecución del script sobre el archivo `prueba3.html`, el cual contiene varias etiquetas mal balanceadas:



Ejecución del script `run_all.sh` sobre `prueba3.html`.

Como puede observarse, el parser detecta varios errores de sintaxis y etiquetas mal cerradas. A pesar de ello, tanto el lexer como BeautifulSoup logran extraer correctamente los enlaces e imágenes del documento.

7.2 Archivos de salida generados

El sistema crea automáticamente archivos de texto en la carpeta `output/`, donde se almacenan los resultados de la extracción realizada por ambos métodos. Por ejemplo:

- `output/links_ply.txt` — enlaces encontrados con el lexer.
- `output/images_bs.txt` — imágenes encontradas con BeautifulSoup.

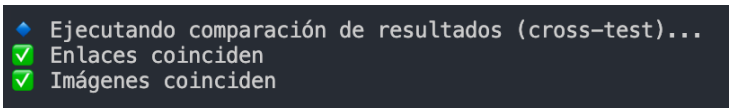
A continuación se muestra el contenido de uno de estos archivos para el caso de prueba anterior:

```
http://www.bbc.co.uk
http://www.ibm.com
```

Estos resultados confirman que la extracción es funcional incluso cuando el documento HTML contiene errores estructurales.

7.3 Comprobación cruzada de resultados

Como se ha explicado en la sección anterior, el script `cross_test.py` compara los archivos de salida y verifica si ambos métodos producen resultados coincidentes.



Verificación cruzada de resultados con `cross_test.py`.

En todos los casos de prueba, se observó una coincidencia perfecta entre los enlaces e imágenes extraídos por PLY y los obtenidos por `BeautifulSoup`. Esto valida la efectividad del enfoque basado en autómatas finitos y gramáticas simplificadas para tareas prácticas de web scraping.

7.4 Detección de errores sintácticos

Uno de los aspectos más relevantes del parser implementado con PLY es su capacidad para identificar errores estructurales en el HTML, como etiquetas mal cerradas, anidamientos incorrectos o cierres fuera de lugar. A continuación se muestra un ejemplo real:

Mensajes de error generados por el parser ante etiquetas no balanceadas.

Estos mensajes son útiles tanto para validar la calidad del documento HTML como para ilustrar cómo un analizador sintáctico puede utilizar reglas formales para detectar errores.

7.5 Tests unitarios

Para validar el funcionamiento interno del lexer y del sistema de análisis desarrollado, se implementó un archivo de pruebas automáticas en `Python` utilizando la librería estándar `unittest`. Estas pruebas están definidas en el archivo:

```
mi_scraper/tests/test_cases.py
```

El objetivo principal de los tests es asegurar que:

- Las etiquetas `<a href="...">` y `<img src="...">` se reconocen correctamente como tokens por parte del lexer.
- La función principal `analyze_html` extrae correctamente enlaces e imágenes desde un documento HTML.
- No se detectan enlaces inexistentes en documentos vacíos o sin etiquetas `<a>`.

A continuación se muestra un extracto relevante del archivo de pruebas, con ejemplos de validación de tokens:

```python
def test_detect_single_a_tag(self):
    html = '<a href="https://example.com">Link</a>'
    lexer.input(html)
    tokens = [tok for tok in lexer if tok.type == 'A_TAG']
    self.assertEqual(len(tokens), 1)
    self.assertEqual(tokens[0].value, 'https://example.com')
```

Esta prueba asegura que se detecta correctamente un único enlace en el documento, y que el valor del token corresponde exactamente con la URL contenida en el atributo `href`.

Además, se incluye una prueba para la extracción conjunta de enlaces e imágenes usando la función principal del sistema:

```
def test_extract_links_and_images(self):
    html = '''
    <a href="http://link.com">Test</a>
    <img src="image.jpg">
    '''
    with tempfile.NamedTemporaryFile(mode="w+", suffix=".html") as f:
        f.write(html)
        f.flush()
        links, images = analyze_html(f.name)

    self.assertEqual(links[0], 'http://link.com')
    self.assertEqual(images[0], 'image.jpg')
```

Esta prueba permite comprobar que el flujo completo de análisis (lexer + parser) se ejecuta correctamente sobre un documento HTML creado de forma temporal, sin necesidad de guardar archivos permanentes.

Resultado de ejecución: Todos los tests pasaron satisfactoriamente, lo que demuestra que el sistema de extracción está funcionando como se espera:

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.004s

OK
```

Este sistema de pruebas contribuye a mantener la fiabilidad del proyecto y garantiza que las funciones clave se comportan correctamente, incluso si en el futuro se realizan modificaciones o mejoras en el código base.

8 Conclusiones

Este proyecto ha servido para poner en práctica conceptos de autómatas y gramáticas aplicados al análisis de HTML. Se ha comprobado que, aunque las herramientas teóricas como PLY funcionan bien para tareas concretas, soluciones como BeautifulSoup son más flexibles y robustas para HTML real. La comparación y las pruebas han mostrado que ambos métodos pueden ser fiables para extraer información sencilla.

Sin embargo, el enfoque basado en autómatas y gramáticas permite comprender en profundidad los límites de los lenguajes regulares y el papel de los analizadores léxicos y sintácticos. Se ha evidenciado que, aunque los AFD son útiles para reconocer patrones simples, no pueden validar la estructura completa de HTML debido a su naturaleza no regular. El uso de un parser con reglas recursivas ha permitido detectar errores de anidamiento y balanceo, reforzando la importancia de las gramáticas independientes del contexto en el análisis de lenguajes más complejos.

Por otro lado, BeautifulSoup ha demostrado ser una herramienta muy eficaz y tolerante a errores, lo que la hace idónea para aplicaciones prácticas de scraping en entornos reales. Su facilidad de uso y capacidad para manejar HTML mal formado justifican su popularidad en la industria.

En resumen, la experiencia ha sido útil para entender mejor cómo se conectan la teoría y la práctica en el análisis de lenguajes. El trabajo realizado ha permitido valorar tanto la potencia de los modelos formales como la necesidad de herramientas prácticas en el desarrollo de soluciones reales, y ha reforzado la importancia de una formación sólida en los fundamentos teóricos de la asignatura.

Referencias

[1] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compiladores: Principios, técnicas y herramientas*. Pearson Educación. Obra clásica sobre teoría de compiladores, autómatas, gramáticas y análisis léxico-sintáctico.

[2] Beazley, D. (2023). *PLY (Python Lex-Yacc) Documentation*. Disponible en: https://www.dabeaz.com/ply/ Documentación oficial de la librería utilizada para construir analizadores léxicos y sintácticos en Python.

[3] Richardson, L. (2023). *BeautifulSoup Documentation*. Disponible en: https://www.crummy.com/software/BeautifulSoup/ Manual y referencias para el uso del parser HTML más extendido en Python.

[4] WHATWG. (2024). *HTML Living Standard*. Disponible en: https://html.spec.whatwg.org/ Especificación oficial y actualizada del lenguaje HTML mantenida por el grupo WHATWG.

[5] Graphviz Team. (2024). Disponible en: https://graphviz.org/ Herramienta empleada para la generación visual de los autómatas finitos representados en esta práctica.

[6] Python Software Foundation. (2024). *Documentación oficial de Python*. Disponible en: https://docs.python.org/3/ Referencia completa del lenguaje de programación utilizado en el desarrollo del proyecto.