Proyecto de prácticas:
Buscador web

Sistemas de Almacenamiento y Recuperación de Información
(Proyecto coordinado SAR - ALT)
Primera parte

2023-2024, v1.0

Trabajo en grupo (grupos de 4 personas). Se deben apuntar en PoliformaT.

Objetivo

El presente proyecto de prácticas de SAR es la primera parte del proyecto conjunto entre las asignaturas de *Sistemas de Almacenamiento y Recuperación de Información* (semestre 3B) y *Algorítmica* (semestre 4A). Esto implica que el código que desarrolle cada grupo se utilizará, por el mismo grupo, como base para el proyecto de la asignatura de ALT en el próximo curso.

El objetivo del presente proyecto es la implementación en *python* de un sistema de crawling de artículos de la Wikipedia para su posterior indexación y consulta.

En PoliformaT/Lessons se proporcionan los cinco ficheros siguientes con extensión `.py`:

- `SAR_Crawler.py`. Programa principal para el crawling de artículos de la Wikipedia en castellano. Crea un objeto de la clase `SAR_Crawler` para descargar artículos de la Wikipedia y almacenarlos en un directorio local en formato json. No se debe modificar.
- `SAR_Indexer.py`. Programa principal para la indexación de artículos. Crea un objeto de la clase `SAR_Project` para extraer los artículos de una colección alojada en un directorio local, los indexa y guarda en disco la instancia del objeto. No se debe modificar.
- `SAR_Searcher.py`. Programa principal para la recuperación de artículos. Carga de disco una instancia de la clase `SAR_Project` para responder a las consultas que se le realicen. No se debe modificar.
- `SAR_Crawler_lib.py`. Librería para la definición de la clase `SAR_Crawler`. Las funcionalidades de crawling se pueden conseguir modificando algunos métodos de esa clase. Se debe modificar.
- `SAR_lib.py`. Librería para la definición de la clase `SAR_Project`. Las funcionalidades de indexación y recuperación se pueden conseguir completando un número reducido de

métodos de esa clases. Se debe modificar.

Nota: Solamente hay que modificar los ficheros `SAR_Crawler_lib.py` y `SAR_lib.py`, los otros tres (`SAR_Crawler.py`, `SAR_Indexer.py` y `SAR_searcher.py`) ni se modifican ni se entregan).

Entrega y Evaluación

En PoliformaT se abrirá una tarea para que el representante de cada grupo suba:

- El fichero `SAR_Crawler_lib.py` con las funcionalidades implementadas.
- El fichero `SAR_lib.py` con las funcionalidades implementadas.
- Una **memoria** del proyecto en formato pdf. Realizar una buena memoria con un buen seguimiento

El proyecto de prácticas se debe hacer en **grupos de 4 personas**, se aceptarán grupos de 3 integrantes en casos de fuerza mayor, en ningún caso se aceptarán grupos de más de 4 personas. Debéis apuntaros a uno de los grupos "ProyPrac" del Poliformat (*Sar/Información del sitio/Grupos a los que puede unirse*) **antes del día 3 de mayo** (tercera sesión de prácticas del proyecto). Debéis apuntaros a uno de los grupos correspondiente a vuestro grupo de prácticas; los que comienzan por 3CO11 para mañanas, y 3CO21 para tardes. En caso de haber integrantes de mañana y tardes, se deberán inscribir a la sesión a la que pertenezcan la mayoría de componentes. En caso de empate, es indistinto a qué grupo os unáis.

Se permite realizar nuevas entregas mientras la tarea de PoliformaT esté abierta (sólo se considerará la última entrega). Tanto en el fichero fuente como en la memoria deberán identificarse TODAS las funcionalidades implementadas y TODOS los miembros del grupo. La no inclusión de dicho listado (funcionalidad, miembro que la ha realizado) conllevará una penalización de 0.5 puntos sobre los 3 puntos.

Se hará una evaluación del proyecto final en el laboratorio, con un peso del 30% (**3 puntos**). La aplicación deberá contar con unas **funcionalidades mínimas** que se puntuarán en total con un máximo de **2 puntos**. Adicionalmente, se podrán **ampliar las funcionalidades** para obtener mayor nota, hasta un máximo total de **1 punto adicional**. La parte adicional sólo se tendrá en cuenta si la parte obligatoria funciona correctamente.

Esta prueba es recuperable y podrán presentarse tanto los alumnos suspendidos como aprobados. La recuperación podrá variar en función del trabajo inicial realizado. Los alumnos suspendidos que se presenten a la recuperación optarán a una nota máxima de 5 sobre 10 (1.5 puntos). Aquellos alumnos que estando aprobados se presenten para subir nota podrán optar al 10 (3 puntos). En ambos casos la nota final será la de la recuperación, sea más alta o más baja que la nota original.

1 Funcionalidades básicas (hasta 2 puntos)

1.1 Crawler (`SAR_Crawler.py`)

La Figura 1.1 muestra el mensaje de ayuda del programa `SAR_Crawler.py`.
El crawler tiene dos modos básicos de funcionamiento:

- `--initial-url` realiza el crawling a partir de una url inicial. Se parte de una cola con una única página inicial; en cada iteración se extrae una página de la cola, se procesa y los enlaces a artículos de la Wikipedia encontrados se añaden a la cola de páginas a procesar. La profundidad de crawling se puede limitar con el argumento `--max-depth-level`.
- `--urls-filename` realiza el crawling sólo de las páginas contenidas en un fichero de urls. En este caso la cola de páginas incluye unicamente las contenidas en el fichero y no se añaden nuevas, la profundidad de crawling en este modo es 0.

```
$ python SAR_Crawler.py --help

usage: SAR_Crawler.py [-h] --out-base-filename OUT_BASE_FILENAME
                      [--batch-size BATCH_SIZE] [--initial-url INITIAL_URL]
                      [--urls-filename URLS_FILENAME]
                      [--document-limit DOCUMENT_LIMIT]
                      [--max-depth-level MAX_DEPTH_LEVEL]

Capturador de entradas de la Wikipedia

optional arguments:
  -h, --help            show this help message and exit
  --out-base-filename OUT_BASE_FILENAME
                        Ruta donde se guardarán los documentos capturados
                        (.json)
  --batch-size BATCH_SIZE
                        Si se define un tamaño, se guardará en un fichero cada
                        vez que hayamos capturado este número de documentos
  --initial-url INITIAL_URL
                        Ruta de un artículo de la Wikipedia
  --urls-filename URLS_FILENAME
                        Listado de URLs que se desea capturar. Se omitirán
                        todas aquellas que no sean entradas de la Wikipedia
  --document-limit DOCUMENT_LIMIT
                        Número máximo de documentos a capturar
  --max-depth-level MAX_DEPTH_LEVEL
                        Profundidad máxima de captura
```

nombre del archivo de guardado

Cuantos documentos como máximo voy a meter en cada archivo de guardado

Aquí no depende de la profundidad, se recuperarán únicamente los documentos de ls urls que pasemos

cuántos documentos como máximo voy a recuperar

Profundidad máxima del árbol en la que se recuperarán los documentos

Figure 1: Ayuda de `SAR_Crawler.py`.

El programa crea un objeto de la clase `SAR_Crawler` y en función del modo elegido invoca al método `wikipedia_crawling_from_url` o `wikipedia_crawling_from_url_list`.

```
crawler = SAR_Wiki_Crawler()
if args.initial_url is not None:
    crawler.wikipedia_crawling_from_url(
        args.initial_url, args.document_limit, args.out_base_filename,
        args.batch_size, args.max_depth_level
    )
else:
    crawler.wikipedia_crawling_from_url_list(
        args.urls_filename, args.document_limit, args.out_base_filename,
        args.batch_size
    )
```

El resultado del proceso de crawling se almacena en uno o varios ficheros de texto donde cada línea se corresponde con un artículo analizado en formato json. El argumento `--document-limit` indica el número total de artículos a recuperar, `--batch-size` indica el número máximo de artículos en cada fichero de salida y `--out-base-filename` indica en nombre base para los ficheros de salida. De esta forma, si `--document-limit` es *130*, `--batch-size` *50* y `--out-base-filename` *salida.json*, como resultado del crawling se crean tres ficheros: *salida 1 3.json* con 50 art ículos, *salida 2 3.json* con 50 artículos y *salida 3 3.json* con los restantes 30.
En la plantilla proporcionada en PoliformaT están definidos todos los métodos de la clase `SAR_Crawler` con excepción de `parse_wikipedia_textual_content` y `start_crawling` que se deben completar: - -

- `parse_wikipedia_textual_content`, realiza el análisis del contenido textual de una entrada de la Wikipedia y devuelve un diccionario con la estructura deseada.

```
def parse_wikipedia_textual_content(self, text: str, url: str) -> Optional[Dict[str, Union[str,List]]]:
    """Devuelve una estructura tipo artículo a partir del text en crudo

    Args:
        text (str): Texto en crudo del artículo de la Wikipedia
        url (str): url del artículo, para añadirlo como un campo

    Returns:

        Optional[Dict[str, Union[str,List[Dict[str,Union[str,List[str,str]]]]]]]:

        devuelve un diccionario con las claves 'url', 'title', 'summary', 'sections':
            Los valores asociados a 'url', 'title' y 'summary' son cadenas,
            el valor asociado a 'sections' es una lista de posibles secciones.
                Cada sección es un diccionario con 'name', 'text' y 'subsections',
                    los valores asociados a 'name' y 'text' son cadenas y,
                    el valor asociado a 'subsections' es una lista de posibles subsecciones
                    en forma de diccionario con 'name' y 'text'.

        en caso de no encontrar título o resúmen del artículo, devolverá None

    """
    def clean_text(txt):
        return '\n'.join(l for l in txt.split('\n') if len(l) > 0).strip()

    document = None
```

```
    # COMPLETAR

    return document
```

La estructura de cada artículo es la siguiente. Un artículo es un diccionario python con cuatro claves `url`, `title`, `summary`, y `sections`. Los valores asociados a `url`, `title` y `summary` son cadenas mientras que el valor asociado a `sections` es una lista de posibles secciones. Cada sección por su parte es un diccionarios con `name`, `text` y `subsections`. Los valores asociados a `name` y `text` son cadenas y el valor asociado a `subsections` es una lista de posibles subsecciones en forma de diccionario con `name` y `text` como claves y valores de tipo cadena.

Las Figuras 2 y 3 muestran respectivamente la entrada y la salida del método `parse_wikipedia_textual_content` para la página: https://es.wikipedia.org/wiki/Videojuego.

```
    ##Videojuego## nombre
    Un videojuego, o juego de vídeo es un juego electrónico en el que una o más personas ...

    Al dispositivo de entrada, usado para manipular un videojuego se le conoce como controlador ...

    Generalmente los videojuegos hacen uso de otras maneras, aparte de la imagen, de proveer ...

    ==Historia== sección (sin subsección devuelve una lista vacía en las subsecciones)
    Los orígenes del videojuego se remontan a la década de 1950, cuando poco después de la ...

    ==Generalidades== sección
    Típicamente, los videojuegos recrean entornos y situaciones virtuales en los que el ....

    Dependiendo del videojuego, una partida puede disputarla una sola persona contra la ...

    Existen videojuegos de muchos tipos. Algunos de los géneros más representativos son los ...

    --Tecnología-- subsección
    Un videojuego se ejecuta gracias a un programa de software (el videojuego en sí) que es ...
...

    --Plataformas--
    Los distintos tipos de dispositivo en los que se ejecutan los videojuegos se conocen como ...


    --Géneros--
    Los videojuegos se pueden clasificar en géneros atendiendo a factores ...


    --Multijugador--
    En muchos juegos se puede encontrar la opción de multijugador, es decir, que varias personas ...

    ==Industria del videojuego==

...
```

Figure 2: Salida de `get_wikipedia_entry_content` para https://es.wikipedia.org/wiki/Videojuego.

```
{
"url": "https://es.wikipedia.org/wiki/Videojuego",
"title": "Videojuego",
"summary": "Un videojuego, o juego de v\u00eddeo es un juego electr\u00f3nico e...",
"sections":
  [
    {"name": "Historia",
     "text": "Los or\u00edgenes del videojuego se remontan a la d\u00e9cada de 1950...",
     "subsections": []},
    {"name": "Generalidades",
      "text": "T\u00edpicamente, los videojuegos recrean entornos y situaciones ...",
      "subsections":
      [
        {
         "name": "Tecnolog\u00eda",
         "text": "Un videojuego se ejecuta gracias a un programa de software (el v..."
         },
        {
         "name": "Plataformas",
         "text": "Los distintos tipos de dispositivo en los que se ejecutan los vid..."
        },
        ...
      ]
    ...
  ]
}
```


Figure 3: Análisis de `parse_wikipedia_textual_content` para la salida textual de https://es.wikipedia.org/wiki/Videojuego.

Para realizar el análisis (*parsing*) del texto os recomendamos el uso de las expresiones regulares definidas en el constructor de la clase.

```
# Expresiones regulares útiles para el parseo del documento
self.title_sum_re = re.compile(r"##(?P<title>.+)##\n(?P<summary>((?!==.+==).+|\n)+)(?P<rest>(.+|\n)*)")
self.sections_re = re.compile(r"==.+==\n")
self.section_re = re.compile(r"==(?P<name>.+)==\n(?P<text>((?!--.+--).+|\n)*)(?P<rest>(.+|\n)*)")
self.subsections_re = re.compile(r"--.+--\n")
self.subsection_re = re.compile(r"--(?P<name>.+)--\n(?P<text>(.+|\n)*)")
```

1º hace un match con el titulo, resumen y el resto (secciones) si algún documento no tiene ni titulo ni resumen no se considera como documento, y lo devolvemos como "none" y no cuenta como documentos recuperados
2º Donde están los dobles guiones, devuelve le indicador del primer == que aparece, para saber los índices de los iguales y por ende donde terminan las secciones
3ºMatch con el nombre, texto y resto (subsecciones)
4º igual que el dos solo que ahora los dobles guiones (--) para saber donde termina cada subsección
5º nombre de la subsección y el texto que contiene

métodos útiles para las expresiones regulares:
se_re.match(text).groupdit() ->
se_re.finditer(text) ->

- `start_crawling`, realiza el crawling utilizando una lista de páginas iniciales y valores de configuración para el control del proceso de captura.

```python
def start_crawling(self,
                initial_urls: List[str], document_limit: int,
                base_filename: str, batch_size: Optional[int], max_depth_level: int,
                ):


    """Comienza la captura de entradas de la Wikipedia a partir de una lista de urls válidas,
        termina cuando no hay urls en la cola o llega al máximo de documentos a capturar.

    Args:
        initial_urls: Direcciones a artículos de la Wikipedia
        document_limit (int): Máximo número de documentos a capturar
        base_filename (str): Nombre base del fichero de guardado.
        batch_size (Optional[int]): Cada cuantos documentos se guardan en
            fichero. Si se asigna None, se guardará al finalizar la captura.
        max_depth_level (int): Profundidad máxima de captura.
    """

    # URLs válidas, ya visitadas (se hayan procesado, o no, correctamente)
    visited = set()
    # URLs en cola
    to_process = set(initial_urls)
    # Direcciones a visitar
    queue = [(0, "", url) for url in to_process]
    hq.heapify(queue)
    # Buffer de documentos capturados
    documents: List[dict] = []
    # Contador del número de documentos capturados
    total_documents_captured = 0
    # Contador del número de ficheros escritos
    files_count = 0

    # En caso de que no utilicemos bach_size, asignamos None a total_files
    # así el guardado no modificará el nombre del fichero base
    if batch_size is None:
        total_files = None
    else:
        # Suponemos que vamos a poder alcanzar el límite para la nomenclatura
        # de guardado
        total_files = math.ceil(document_limit / batch_size)

    # COMPLETAR
```

meanheap -- cola de prioridades.
Las prioridades son profundidad, url padre, url propia

El funcionamiento básico debe ser:

1. Seleccionar una página no procesada de la cola de prioridad.
2. Descarga el contenido textual de la página y los enlaces que aparecen en ella.
3. Añadir, si procede, los enlaces a la cola de páginas pendientes de procesar.
4. Analizar el contenido textual para generar el diccionario con el contenido estructurado del artículo.

Este proceso debe repetirse hasta que se cumplan las condiciones de parada. Además, se deben ir guardando los bloques de artículos en memoria secundaria cuando sea necesario.

Para añadir un enlace a la cola se debe cumplir, como mínimo, que:

- Es una página correcta de la Wikipedia en castellano. Para comprobar que una url es válida puedes utilizar el método `is_valid_url`.
- No ha sido procesado previamente.

Se recomienda utilizar el método `get_wikipedia_entry_content` para la descarga del contenido de un artículo de Wikipedia y `save_documents` para guardar un bloque de artículos (una lista de diccionarios) en memoria secundaria. Puesto que será imposible visitar todas las páginas de la Wikipedia, *tampoco es el objetivo de este proyecto*, es necesario priorizar unas páginas sobre otras. Se sugiere utilizar una cola de prioridad para almacenar las urls pendientes de ser procesadas, para este fin es útil utilizar la librería `heapq` que ya está importada en la plantilla proporcionada.

```
import heapq as hq
```

Para más información sobre el uso de la cola de prioridad puedes consultar su documentación en https://docs.python.org/3/library/heapq.html.

1.2 Indexador (`SAR_Indexer.py`)

La Figura 4 muestra el mensaje de ayuda del programa `SAR_Indexer.py`. Se ha incluido la biblioteca `argparse`, para facilitar el análisis de los argumentos de entrada del programa. Este análisis ya está incluido en el código que hay disponible en PoliformaT y, por tanto, no se debe modificar ni subir el fichero `SAR_Indexer.py`.

```
$ python SAR_Indexer.py --help
usage: SAR_Indexer.py [-h] [-S] [-P] [-M] [-O] dir index

Index a directory with Wikipedia articles in json format.

positional arguments:
  dir                   directory with the Wikipedia articles.
  index                 name of the index.

optional arguments:
  -h, --help            show this help message and exit
  -S, --stem            compute stem index.
  -P, --permuterm       compute permuterm index.
  -M, --multifield      compute index for all the fields.
  -O, --positional      compute positional index.
```

Figure 4: Ayuda del `SAR_Indexer.py` (incluye las funcionalidades ampliadas).

Las funcionalidades básicas con las que debe contar el indexador son las siguientes:

- Requiere obligatoriamente dos argumentos de entrada:
  - **dir**, el directorio donde está la colección de artículos en formato `json`, y
  - **index**, el nombre del fichero donde se guardará la información del índice creado.
- El programa crea un objeto de la clase `SAR_Project` e invoca a su método `index_dir`.

  El método `index_dir`, junto con el resto de métodos a los que éste llame, deberá:
  - Procesar los documentos `.json` del directorio de manera recursiva y extraer los artículos.
  - Tokenizar cada artículo:
    - Eliminando símbolos no alfanuméricos (comillas, interrogantes, ...) y
    - extrayendo los términos (consideraremos separadores de términos los espacios, los saltos de línea y los tabuladores).

    No se deben distinguir mayúsculas y minúsculas en la indexación.
  - Asignar a cada documento procesado un identificador único (docid) que será un entero secuencial. En este contexto un documento es un fichero en memoria secundaria.
  - Asignar a cada artículo un identificador único. Se debe saber a qué documento (fichero) pertenece cada artículo y qué posición relativa ocupa dentro de él.
  - Crear un índice invertido accesible por término. Cada entrada contendrá una lista con los artículos en los que aparece ese término.
- Finalmente, el programa principal guarda en disco toda la información del índice con el método `save_info`. También muestra información sobre el proceso de indexado.

```
indexer = SAR_Indexer()
t0 = time.time()
indexer.index_dir(args.dir, **vars(args))
t1 = time.time()
indexer.save_info(args.index)
t2 = time.time()
indexer.show_stats()
print("Time indexing: %2.2fs." % (t1 - t0))
print("Time saving: %2.2fs." % (t2 - t1))
print()
```

A continuación se muestra un extracto del método `index_dir` ya incluido en la biblioteca.

```
def index_dir(self, root, **args):
    ...
    for d, subdirs, files in os.walk(root):
        for filename in files:
            if filename.endswith('.json'):
                fullname = os.path.join(d, filename)
                self.index_file(fullname)
    ########################################
    ## COMPLETAR PARA FUNCIONALIDADES EXTRA ##
    ########################################
```

Como se puede observar, el método realiza el recorrido recursivo por el directorio utilizando la función `walk` de la biblioteca `os`. Para conseguir la funcionalidad básica de indexado sólo hace falta completar el método `index_file`.

```
def index_file(self, filename):
    ...
    for i, line in enumerate(open(filename)):
        j = self.parse_article(line)

        #################
        ### COMPLETAR ###
        #################
```

El método `parse_article` recibe una cadena (objeto json guardado por el crawler) y devuelve un diccionario procesado en el que todos los valores son cadenas. Las claves de este diccionario son:

- `url`, dirección del artículo. Este campo identifica al artículo, no se pueden indexar dos artículos con la misma url.
- `title`, título del artículo.
- `summary`, resumen del artículo. Parte inicial del artículo previa a las secciones.
- `all`, el contenido de todo el artículo. Incluye título, resumen y todas las secciones y subsecciones.
- `section-name`, nombre de todas las secciones y subsecciones del artículo.

En la versión obligatoria se debe indexar todo el contenido del artículo, txt = `j['all']`.

1.3 Recuperador de artículos (SAR_Searcher.py)

La Figura 5 muestra el mensaje de ayuda del programa `SAR_Searcher.py`. Este fichero no se debe modificar ni subir a PoliformaT.

```
$ python SAR_Searcher.py --help

usage: SAR_Searcher.py [-h] [-S] [-N | -C] [-A]
                       [-Q query | -L qlist | -T test]
                       index

Search the index.

positional arguments:
  index                 name of the index.

optional arguments:
  -h, --help            show this help message and exit
  -S, --stem            use stem index by default.
  -N, --snippet         show a snippet of the retrieved documents.
  -C, --count           show only the number of documents retrieved.
  -A, --all             show all the results. If not used, only the first 10
                        results are showed. Does not apply with -C and -T
                        options.
  -Q query, --query query
                        query.
  -L qlist, --list qlist
                        file with queries.
  -T test, --test test  file with queries and results, for testing.
```

Figure 5: Ayuda del `SAR_Searcher` (incluye funcionalidades ampliadas).

Las funcionalidades básicas del recuperador de artículos (*searcher*) son las siguientes:

- Sólo tiene un argumento obligatorio, `index`, que es el nombre del fichero que contiene el índice guardado previamente con el programa `SAR_Indexer.py`.
- Tiene cuatro modos de funcionamiento distintos determinados por utilizar los argumentos `-Q`, `-L`, `-T` o no indicar ninguno de ellos:
  - `-Q` permite pasar una consulta directamente en la llamada al programa. Por ejemplo, `python SAR_Searcher.py indice.bin -Q 'python AND precisión'` , resuelve la consulta `'python AND precisión'` utilizando para ello el índice guardado en el fichero `indice.bin`, muestra el resultado y finaliza.
  - `-L` recibe una lista de consultas mediante un fichero, muestra el resultado consulta por consulta y finaliza. Por ejemplo, `python SAR_Searcher.py indice.bin -L query_list.txt`, resuelve las consultas contenidas en el fichero `query_list.txt` utilizando el índice guardado en el fichero `indice.bin`.

  - `-T` se utiliza para evaluar el recuperador utilizando una lista de consultas resueltas. Esto permite determinar si el programa funciona correctamente. Cada línea del fichero de consultas debe tener una consulta y el número de artículos que se deben recuperar para esa consulta; los dos valores (consulta y cardinalidad de la respuesta) deben estar separados por un tabulador. Por ejemplo, `python SAR_Searcher.py indice.bin -T test_list.txt` comprueba si todas las consultas contenidas en `test_list.txt` obtienen el resultado esperado sobre el índice guardado en el fichero `indice.bin`. Si todas las consultas devuelven el número de artículos esperado, se mostrará el mensaje `'Parece que todo ha ido bien, buen trabajo!'`, en caso contrario, se mostrará un mensaje de error resaltando la consulta con el resultado incorrecto.
  - En otro caso, el programa entrará en un bucle de petición de consulta y devolución de artículos relevantes.

- Se debe permitir utilizar las conectivas AND, OR y NOT en las consultas. El orden de evaluación y prioridad de las conectivas será de izquierda a derecha. Por ejemplo, la consulta `'term1 AND NOT term2 OR term3'` deberá devolver los artículos que contienen `term1` pero no `term2` más los artículos que contienen `term3`.

  IMPORTANTE: para realizar la intersección (AND) y unión (OR) de *postings list* se deben implementar los algoritmos de *merge* vistos en teoría, en los métodos:

  - `and_posting(self, p1, p2)` y
  - `or_posting(self, p1, p2)`.

- Cuando se elige la opción `-Q` o el modo interactivo, la presentación de los resultados se realizará en función de los argumentos opcionales con los que se ejecute el programa `python SAR_Searcher.py`:

  - `-C`, muestra la consulta y el número de artículos devuelto por el buscador, como en el modo `-L`.
  - `-N`, muestra la consulta y, para cada artículo recuperado por el buscador:
    * En una línea, un número de orden para numerar los artículos recuperados, el identificador del artículo y la url.
    * En una segunda línea, el título del artículo.
    * Posteriormente, un trozo del artículo donde aparezcan las palabras de la consulta encontradas (`snippet`). En consulta muy complicadas no es necesario que aparezcan las palabras buscadas en el snippet.
  - Si no se indica ninguno de los argumentos anteriores se mostrará, en una línea por artículo, la misma información que en la opción `-N` pero sin incluir el *snippet*.

- Si se incluye el argumento `-A`, el recuperador mostrará todos los artículos recuperados, en caso contrario (la opción por defecto) sólo se mostrarán los 10 primeros.

2 Funcionalidades ampliadas (hasta 1 punto)

Para obtener la máxima puntuación, además de las funcionalidades básicas, se deberán implementar perfectamente al menos 3 de las siguientes funcionalidades extra.

Atención: Las funcionalidades extra sólo se puntuarán si las funcionalidades básicas funcionan correctamente.

2.1 Paréntesis

Con la funcionalidad básica, la aplicación debe permitir el uso de conectivas `'AND'` y `'OR'`, además del `'NOT'`, para hacer consultas complejas. En ese caso, la evaluación predeterminada será de izquierda a derecha.

Con esta ampliación se permitirá el uso de paréntesis, `'('` y `')'`, para modificar la prelación/precedencia de los operadores lógicos.

Ejemplo de funcionamiento: la consulta `'term1 AND NOT (term2 OR term3)'` deberá devolver los artículos que contienen `'term1'` pero no `'term2'` ni `'term3'`.

Para lograr esta funcionalidad será necesario ampliar el método `solve_query` para realizar algún tipo de análisis sintáctico (*parsing*) más elaborado de la consulta. Para facilitar el análisis se puede asumir que todas las consultas estarán bien construidas (es decir, no es necesario detectar errores sintácticos).

2.2 Stemming

Permitir la realización de *stemming* de documentos y consultas. Se ha añadido un argumento (`-S`, `--stem`) para activar esta funcionalidad tanto al crear los índices como al hacer las consultas.

Ya se ha incluido un *stemmer* en castellano; para obtener el *stem* de un *token* es suficiente con hacer `self.stemmer.stem(token)`. La clase `SAR_Project` tiene un atributo `self.sindex` para guardar el índice de *stems*.

Además de otros métodos, para conseguir esta funcionalidad se sugiere completar y utilizar los métodos:

- `make_stemming`, para crear el índice de *stems*.
- `get_stemming`, para obtener la postings list asociada al *stem* de un término.

2.3 Multifield

En el funcionamiento básico de `SAR_Indexer.py` sólo se indexa el contenido del campo `'all'` del diccionario del artículo.

Con la ampliación *multifield*, se añadirán índices adicionales para la url del artículo, el título, el resumen u los nombres de secciones. Una vez implementada la funcionalidad, los términos de las consultas podrán incluir uno de los prefijos `'url:'`, `'title:'`, `'summary:'`,

'section-name:' y 'all:' para indicar que ese término se debe buscar en el índice correspondiente. Si no se indica ningún campo, el término se buscará en el índice del artículo completo ('all').

Se debe permitir mezclar búsquedas sobre diferentes índices en la misma consulta.

Ejemplo de funcionamiento: la consulta 'title:información AND precisión' deberá recuperar los artículos donde aparezca 'información' en el título y 'precisión' en cualquier parte.

Como recomendación de implementación se sugiere utilizar el índice principal, self.index, para guardar los índices de todos los campos. Esto se puede conseguir haciendo que sea un diccionario de diccionarios de forma que tenga un primer nivel para determinar el campo. Por ejemplo, self.index['title'] se corresponda con el diccionario para almacenar los términos del campo 'title'.

Debe tokenizarse el contenido de todos los campos a excepción del campo 'url', que contiene el valor de la dirección del artículo y, por este motivo, no debe tokenizarse. En la plantilla del fichero SAR_lib.py se ha añadido una lista con el nombre de todos los campos junto con un booleano que determina si el campo debe ser tokenizado o no:

```
# lista de campos, el booleano indica si se debe tokenizar el campo
# NECESARIO PARA LA AMPLIACION MULTIFIELD
fields = [
    ("all", True), ("title", True), ("summary", True),
    ("section-name", True), ('url', False)
]
```

2.4 Búsquedas posicionales

Esta ampliación permite la búsqueda de varios términos consecutivos utilizando las dobles comillas. Esto hace necesario el uso de *postings list* posicionales.

Ejemplo de funcionamiento: buscar "fin de semana" con la funcionalidad implementada deberá devolver sólo los artículos en los que los tres términos aparecen de forma consecutiva, mientras que buscar fin de semana encontraría todos los artículos en los que aparecen los tres términos sin importar la posición.

Para conseguir este funcionamiento será necesario almacenar en el índice invertido, además de en qué artículos aparece cada termino, en qué posiciones aparecen.

Para el recuperador, será necesario completar el método get_positionals para obtener la *postings list* de una secuencia consecutiva de términos.

2.5 Permuterm

Esta funcionalidad permitirá la búsqueda utilizando '*' y '?' como comodines (*wildcard queries*). Sólo se permite un comodín por palabra.

Ejemplo de funcionamiento: buscar s*dney encontraría todas las páginas que contengan algún término que comience por 's' y termine en 'dney'.

Para lograr esta funcionalidad será necesario implementar índices *permuterm*. Sin embargo, no os compliquéis mucho con la estructura/eficiencia del índice: una lista ordenada por *permuterm* y una búsqueda dicotómica puede dar buenos resultados.

En esta ampliación se recomienda utilizar el atributo `self.ptindex` para guardar el índice *permuterm* y completar el método `get_permuterm` para obtener la *postings list* de un término utilizando el índice *permuterm*.

3 Funcionamiento conjunto de las funcionalidades extra

Se espera que todas las funcionalidades extra implementadas funcionen de forma conjunta en la misma consulta, teniendo en cuenta que:

- El uso de *stemming* no se aplica a las búsquedas posicionales. Con la opción de *stemming* activada, la consulta `día AND "semana"` realizará *stemming* en `día` pero no en `semana`.
- En las búsquedas posicionales no se podrán utilizar comodines. La consulta `'"fin de sem*a"'` no está permitida.
- El *stemming* no se aplica a las búsquedas con comodines. En la consulta `sem*a` no se aplica *stemming* aun con la función activada.

4 FAQ

A continuación se da respuesta a algunas preguntas frecuentes. Si surgen más dudas se irán añadiendo en revisiones sucesivas de este documento.

¿Qué se entrega?

El responsable del grupo deberá subir a la tarea de PoliformaT únicamente tres ficheros:

- `SAR_Crawler_lib.py`, con las funcionalidades del crawler implementadas.
- `SAR_lib.py`, con las funcionalidades del indexador y búscador implementadas.
- `informe.pdf`, con el informe descriptivo del trabajo realizado. El informe debe incluir como mínimo:
  - Enumeración de los miembros del grupo y descripción de la contribución al proyecto de cada miembro.
  - Enumeración de las funcionalidades extra implementadas y descripción de su implementación.
  - Descripción y justificación de las decisiones de implementación realizadas.
  - Descripción del método de coordinación utilizado por los miembros del grupo en la realización del proyecto.
  - Podéis añadir toda la información que consideréis importante, incluidas opiniones subjetivas de la marcha del proyecto y las dificultades afrontadas.

Se permiten los reenvíos mientras la tarea esté activa, se evaluará la última entrega.

¿Cómo será la evaluación del proyecto?

Para evaluar el proyecto se tendrán en cuenta múltiples factores:

1. El funcionamiento correcto del crawler. Se realizará una descarga acotada y se evaluará el contenido descargado. Para hacer esta evaluación se utilizará el programa `SAR_Crawler.py` original y la última versión de la biblioteca `SAR_Crawler_lib.py` subida por el responsable del grupo a la tarea de PoliformaT.

2. El funcionamiento correcto del indexador y del búscador, tanto de las funcionalidades básicas como de las funcionalidades extras implementadas. Se utilizará la opción de test, argumento `-T` en el buscador, para comprobar que los resultados son los esperados. Para hacer esta evaluación se utilizarán los programas `SAR_Indexer.py` y `SAR_Searcher.py` originales y la última versión de la biblioteca `SAR_lib.py` subida por el responsable del grupo a la tarea de PoliformaT.

3. El código enviado. Se hará una evaluación de la eficiencia y corrección del código desarrollado por el grupo, penalizándose las implementaciones poco eficiente o excesivamente farragosas. Se recomienda añadir comentarios donde se considere oportuno (sin llegar al extremo de comentar las cosas demasiado obvias).

4. La memoria del proyecto. La memoria del proyecto debe ser lo suficientemente descriptiva para entender como se ha realizado el proyecto y ser correcta tanto en contenido como en formato.

5. También se realizará una prueba objetiva individual y por grupo. **Todos** los miembros del grupo **deben ser conocedores del funcionamiento completo de la aplicación**, no sólo de la parte que han implementado.

¿Hay que procesar los documentos con alguna biblioteca?

Los documentos que se utilizan en el **indexador** y el **buscador** son los generados por el **crawler**. Son ficheros de texto donde cada línea es un objeto en formato *json*. Además de los ficheros que se puedan descargar con el crawler, el PoliformaT hay disponibles ficheros para poder hacer pruebas controladas.
Parcialmente, la grabación y la carga de los ficheros *json* ya está incluida en las plantillas de las bibliotecas `SAR_Crawler_lib.py` y `SAR_lib.py`.

¿Las ampliaciones han de ser acumulativas?

Para obtener la máxima puntuación **sí**, las ampliaciones deben ser acumulativas, con las únicas excepciones comentadas en el apartado correspondiente.
Entre los ficheros de ayuda proporcionados hay ejemplos de combinaciones de funcionalidades (sobre todo para poder comprobar si se obtiene el resultado esperado).

¿Cómo se generan los snippets?

La descripción de *snippet* da libertad para generarlos de varias formas, siempre que esas formas tengan sentido. Algunas opciones (existen otras) serían:

- **Trabajar a nivel de palabras** (lista de palabras). Obtener el índice de la primera y última ocurrencia de cada término (puede haber términos que no aparezcan en el cuerpo del artículo). Quedarse con las posiciones mínima y máxima de los índices anteriores. Extraer un fragmento de texto entre las posiciones anteriores añadiendo un pequeño contexto a izquierda y derecha. Esta opción puede dar snippets muy largos si hay términos al inicio y al fin.
- **Sacar un snippet de cada término** (su primera ocurrencia, para simplificar) poniendo dicho término con un contexto antes y después. Opcionalmente se pueden unir segmentos que se solapen. Esta opción es *ligeramente* más compleja que la anterior.

No hace falta respetar ni las mayúsculas ni los saltos de línea del documento original. Se puede trabajar con los textos normalizados.
En cualquier caso, **en la memoria del proyecto se debe describir el método utilizado para calcular los snippets**.

¿Se pueden usar operadores de conjuntos (tipo `set` en python) en lugar de los algoritmos de unión e intersección de postings lists vistos en teoría?

NO, para unir e intersectar *postings lists* se deben utilizar los algoritmos vistos en teoría, el AND y el OR de dos *postings list*. También se debe implementar el NOT de una *postings list* sin utilizar los métodos de conjuntos. No es por una cuestión de eficiencia, sino para practicar lo visto en la teoría de la asignatura.

¿Se puede tener un diccionario inverso para la versión de stemming?

Para la ampliación de *stemming* se sugieren estas 2 opciones:

- Tener un diccionario donde cada clave sea un *stem* y el valor asociado a esa clave sea la lista de términos con ese *stem*. Este diccionario se puede generar al final del indexado con el método `make_stemming`. Por ejemplo, la clave `'quij'` podría tener asociada como valor la lista `['quijada', 'quijote', 'quijotesco']`.

  La forma de usar este diccionario auxiliar es:

  - obtener el *stem* del término de la *query*.
  - la *postings list* del *stem* será la unión de las postings list de todos los términos con ese *stem*.

- Generar otro diccionario inverso donde los términos son *stems*. Esto ocupa más memoria que la opción anterior.

Si os habéis planteado alguna otra opción eficiente, podéis implementarla.
En cualquier caso, **en en la memoria del proyecto se debe describir el método de stemming útilizado**.

¿Qué pasa si el número de artículos no coincide con la referencia?

En la mayoría de casos esto es debido a uno de estos dos factores:

- Una implementación errónea de los métodos de indexación o de recuperación de los artículos.
- Una normalización y *tokenización* de los artículos distinta de la utilizada para el cálculo de la referencia.

¿Cómo combinar consultas posicionales y stopwords?

En este proyecto no se eliminan *stopwords*.

Además de las bibliotecas `SAR_Crawler.py` y `SAR_lib.py`, ¿se pueden añadir más ficheros que contengan funciones o clases adicionales?

NO. Eso sí, podéis añadir a las clases `SAR_Crawler` y `SAR_Project` todos los atributos y/o métodos que consideréis conveniente. Incluso se pueden añadir más clases o funciones pero siempre dentro de las dos bibliotecas, sin modificar los programas de crawling, indexación y recuperación de artículos ni añadir más ficheros.

Si implementamos funcionalidades extra, ¿tienen que funcionar sólo si se pasan ciertos argumentos o tienen que funcionar siempre?

Para que se active el *stemming* se le debe indicar con el parámetro **-S** en el searcher. Las demás funcionalidades extra deben ser *implicitas*.

Apéndices

A Ejemplos de indexación

```
$ python SAR_Indexer.py refs/100 100_simple.bin
=======================================
Number of indexed files: 3
---------------------------------------
Number of indexed articles: 296
---------------------------------------
TOKENS:
        # of tokens in 'all': 62555
---------------------------------------
Positional queries are NOT allowed.
=======================================
Time indexing: 0.70s.
Time saving: 0.09s.
```

```
$ python SAR_Indexer.py refs/1000 1000_all.bin -S -P -M -O
=======================================
Number of indexed files: 3
---------------------------------------
Number of indexed articles: 2779
---------------------------------------
TOKENS:
        # of tokens in 'all': 238908
        # of tokens in 'title': 3015
        # of tokens in 'summary': 36732
        # of tokens in 'section-name': 9830
        # of tokens in 'url': 2779
---------------------------------------
PERMUTERMS:
        # of permuterms in 'all': 2116670
        # of permuterms in 'title': 23557
        # of permuterms in 'summary': 324559
        # of permuterms in 'section-name': 87828
        # of permuterms in 'url': 132848
---------------------------------------
STEMS:
        # of stems in 'all': 166236
        # of stems in 'title': 2769
        # of stems in 'summary': 23442
        # of stems in 'section-name': 7092
        # of stems in 'url': 2749
---------------------------------------
Positional queries are allowed.
=======================================
Time indexing: 33.13s.
Time saving: 7.07s.
```

B Ejemplos de recuperación

```
$ python SAR_Searcher.py 100_simple.bin -Q 'precisión AND NOT exhaustividad'
========================================
# 01 (  8) Google:        https://es.wikipedia.org/wiki/Google
# 02 ( 10) Indexación Semántica Latente:
↪  https://es.wikipedia.org/wiki/Indexaci%C3%B3n_Sem%C3%A1ntica_Latente
# 03 ( 12) Inteligencia
↪  artificial:        https://es.wikipedia.org/wiki/Inteligencia_artificial
# 04 ( 16) Keith van Rijsbergen: https://es.wikipedia.org/wiki/Keith_van_Rijsbergen
# 05 ( 20) Modelo Booleano Extendido:
↪  https://es.wikipedia.org/wiki/Modelo_Booleano_Extendido
# 06 ( 25) Recuperación (memoria):
↪  https://es.wikipedia.org/wiki/Recuperaci%C3%B3n_(memoria)
# 07 ( 27) Recuperación difusa: https://es.wikipedia.org/wiki/Recuperaci%C3%B3n_difusa
# 08 ( 32) Software: https://es.wikipedia.org/wiki/Software
# 09 ( 34) Vannevar Bush: https://es.wikipedia.org/wiki/Vannevar_Bush
# 10 ( 35) Aprendizaje organizacional:
↪  https://es.wikipedia.org/wiki/Aprendizaje_organizacional
========================================
Number of results: 25
```

```
$ python SAR_Searcher.py 100_simple.bin -Q 'exhaustividad AND precisión' -N
========================================
# 1 (  0)
↪  https://es.wikipedia.org/wiki/B%C3%BAsqueda_y_recuperaci%C3%B3n_de_informaci%C3%B3n
Búsqueda y recuperación de información
        correctitud 3 1 precisión 3 2 exhaustividad 3 3 proposición ... el promedio de
↪  las puntuaciones medias de precisión para cada ...

# 2 (  7) https://es.wikipedia.org/wiki/Gerard_Salton
Gerard Salton
        los valores de precisión y exhaustividad en sistemas documentales de ... de
↪  mejorar los valores de precisión y exhaustividad en sistemas ...

# 3 ( 26) https://es.wikipedia.org/wiki/Recuperaci%C3%B3n_de_informaci%C3%B3n
Búsqueda y recuperación de información
        correctitud 3 1 precisión 3 2 exhaustividad 3 3 proposición ... el promedio de
↪  las puntuaciones medias de precisión para cada ...
========================================
Number of results: 3
```

```
$ python SAR_Searcher.py 100_simple.bin -L queries.txt
precisión        28
exhaustividad        4
precisión AND exhaustividad        3
precisión OR exhaustividad        29
precisión AND NOT exhaustividad        25
NOT precisión AND exhaustividad        1
```

```
$ python SAR_Searcher.py 100_simple.bin -T test_queries.txt
```

```
precisión        28
exhaustividad        4
>>>>precisión AND exhaustividad        3 != 2<<<<
precisión OR exhaustividad        29
precisión AND NOT exhaustividad        25
NOT precisión AND exhaustividad        1
NOT precisión AND exhaustividad

Parece que hay alguna consulta mal :-(
```