Notas

A continuación se describen los desafíos más importantes que se fueron presentando mientras se desarrollaba el trabajo.

Modelo homogéneo

Antes de comenzar a escribir el código para extraer la información, decidimos pensar el modelo de información que deseábamos extraer. Una vez hecho eso, optamos por utilizar las dataclasses de Python para el guardado de los datos ya que permite una mejor organización y seguridad de los campos obtenidos.

Como método de almacenamiento elegimos un archivo de texto plano en formato JSON ya que los datos que se deseaban extraer eran pocos como para utilizar una base de datos, aunque a medida que fue avanzando el trabajo, notamos la desventaja de utilizar JSON para actualizar los datos si fuese necesario. El resultado fue el siguiente:

```
@dataclass_json
@dataclass
class Show:
    cinema: str
    room: str
    language: str
    time: datetime = iso_datetime


@dataclass_json
@dataclass
class Movie:
    title: str
    genres: List[str]
    languages: List[str]
    origins: List[str]
    duration: Optional[int]
    directors: List[str]
    rated: Optional[str]
    actors: List[str]
    synopsis: str
    trailer: Optional[str]
    shows: List[Show]
    released: bool
```

Definir un modelo de antemano resultó muy ventajoso, ya que permitió que ambos scrapers trabajen con esta estructura en común.

Scrapers

Cinema La Plata

Se realizó el scrapeo de la sección de películas en cartelera (http://www.cinemalaplata.com/Cartelera.aspx) y películas que son futuros estrenos (http://www.cinemalaplata.com/Cartelera.aspx?seccion=FUTURO).

Luego de analizar el código de las páginas nos dimos cuenta de que la estructura del HTML en ambos casos era el mismo. Por este motivo se optó por definir una única clase *MovieParser* que se encargaría de procesar el listado de peliculas obtenidas. Esto resulto ventajoso ya que se pudo reutilizar la misma clase para parsear los dos listados.

Tiempo de procesamiento

El tiempo de procesamiento resulta muy costoso si se realizan las requests sincrónicamente, por lo tanto, utilizamos threads para realizarlas asincrónicamente. Esto bajó el tiempo aproximadamente de 40 segundos a 6 segundos.

Cinépolis

Al igual que Cinema La Plata, se realizó el scrapeo de la cartelera (https://www.cinepolis.com.ar/) y de los futuros estrenos (https://www.cinepolis.com.ar/proximos-estrenos).

Carga dinámica

El primer problema que surgió fue que el sitio cargaba algunas partes en forma dinámica mediante Javascript y otras partes en forma estática. La carga dinámica es común en la actualidad ya que los frameworks más conocidos de Javascript lo utilizan para un mejor manejo del estado de la información, el problema es que el procesamiento lo hace el navegador, por lo tanto, al solicitar el sitio mediante el método GET desde Python la información no se ve reflejada.

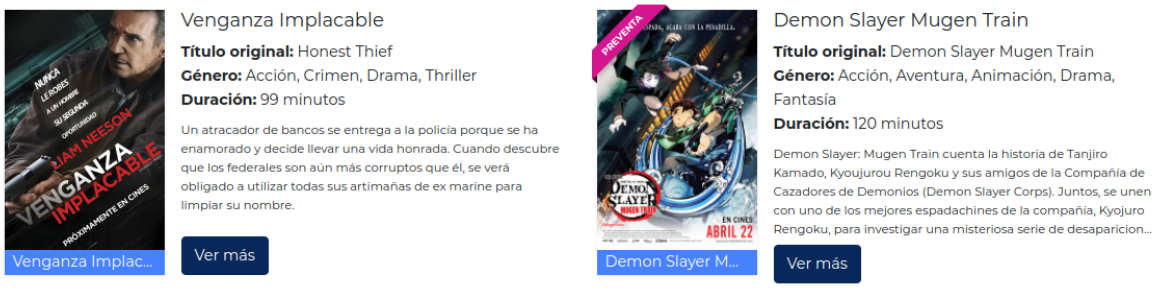
Se investigó si la página utilizaba algún método para reflejar esa información de forma estática, pero no hubo resultados. Por lo tanto, se decidió utilizar la librería de Python Selenium, la cual permite utilizar un navegador para el scrapeo de las páginas y selectores de elementos como BeautifulSoup.

El problema de Selenium, es que es muy demandante de recursos y resulta muy engorroso de utilizar en comparación a otras librerías como BeautifulSoup. ¿Por qué engorroso?

1. Porque se necesita instalar un webdriver para utilizarlo y dificulta la compartición del código.
2. Se dificulta paralelizar, ya que solo se puede interactuar con el sitio actual, por lo tanto, para agilizar el proceso se deberían abrir muchos navegadores y esto resulta muy costoso en cuanto a recursos.
3. Depende de la visualización del navegador que se esté utilizando, es decir, la ejecución puede diferir si se utiliza un webdriver de Firefox y otro de Chrome. También depende del tamaño de la ventana donde lo estemos ejecutando, ya que el sitio puede dibujar diferentes HTML para mantener el diseño responsivo.

Selectores complejos

Más allá de las desventajas, Selenium nos permitió scrapear el sitio exitosamente y pudimos utilizar otros preprocesadores para seleccionar elementos como XPATH, el cual resulta muy ventajoso para selecciones complejas, como por ejemplo: En la sección de futuros estrenos aparecen películas en preventa y no deben ser procesadas, ya que son películas que están en la cartelera.



Como se puede ver, se tenian que filtrar todas las películas que no tienen el logotipo de "Preventa", por lo tanto, se requerían muchas líneas de código utilizando selectores como CSS o por tags de HTML. Mientras que con XPATH se resolvió con una sola expresión: `"//a[contains(@class, 'movie-thumb')] not(div[contains(@class, 'movie-thumb-ribbon')])]"`

Datos técnicos no estructurados

En el detalle de la película, los datos técnicos no poseían una estructura clara, observemos:

```
▼<div class="col mt-3 mt-md-0">
   <h4>Datos técnicos</h4>
   <hr>
  ▼<p> == $0
     <strong>Título Original</strong>
     ": Honest Thief"
     <br>
     <strong>Origen</strong>
     ": Estados Unidos"
     <br>
     <strong>Género</strong>
     ": Thriller, Acción, Crimen, Drama
         "
     <br>
     <strong>Director</strong>
     ": Mark Williams"
     <br>
     <strong>Actores</strong>
     ": Liam Neeson, Kate Walsh, Jai Cou
     Adam Teper, Jose Guns Alves"
     <br>
     <strong>Calificación</strong>
     ": N/A"
     <br>
     <strong>Duración</strong>
     ": 99 min."
     <br>
     <strong>Distribuidora</strong>
     ": BF Paris"
     <br>
   </p>
```

Se optó por utilizar un diccionario, en donde la key sea el texto del tag *strong* y el contenido sea el nodo de texto que le sigue. Pero acá aparece un nuevo problema:

Nodos de texto

Librerías como BeautifulSoup y Selenium no soportan seleccionar nodos de texto, ya que los preprocesadores que utilizan por debajo no lo permiten. Por lo tanto consideramos parsear todo con Python pidiendole el contenido del html al tag *p* que se ve en la imagen, pero en lugar de eso, se optó por usar los selectores que proporciona el framework Scrapy que reconocen nodos de texto.

Por lo tanto, el código quedó de la siguiente forma:

```
def get_technical_fields(self):
    html = self._get_tech_fields_html()
    selector = scrapy.selectors.Selector(text=html)

    fields_content = selector.xpath("//p/text()").getall()
    fields_name = selector.xpath("//p/strong/text()").getall()

    return dict(zip(fields_name, fields_content))
```

- Nota: El zip de Python permite agrupar los elementos posición a posición (https://docs.python.org/3.3/library/functions.html#zip)

Duplicación de la información

Títulos

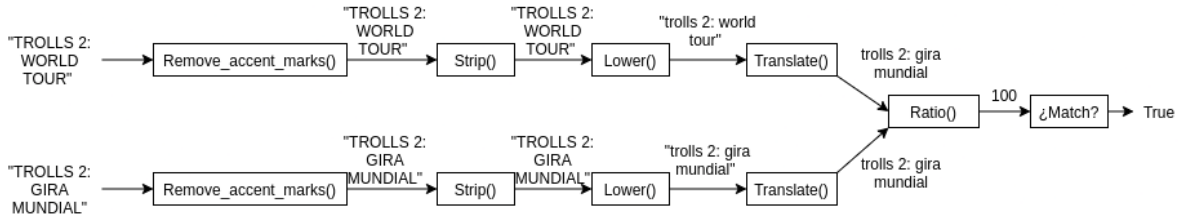
Los títulos de las películas están escritos de forma diferente en cada sitio, lo que provoca que haya duplicación si juntamos la información recolectada de ambos scrapers. Para solucionar esto consideramos las siguientes opciones:

1. Scrapear el buscador de IMDB para obtener el imdb_id de la película para identificarlas.
2. Normalizar el título mediante pipelines, tanto de forma sintáctica como de forma semántica.

La opción 1 se descartó ya que está fuera del alcance del trabajo. Por lo tanto, se optó con la opción 2, el pipeline implementado fue el siguiente:

1. **Remove_accent_marks**: Eliminar los acentos
2. **Strip**: Eliminar caracteres sobrantes en los bordes de la palabra
3. **Lower**: Convertir a minúsculas
4. **Translate**: Traducir a español
5. **Ratio**: Calcular el radio parcial sintácticamente utilizando fuzzywuzzy
6. **Match**: Se compara el radio contra una cota prefijada, para tener una mayor precisión elegimos 90 como cota

A continuación se muestra un ejemplo con la película trolls 2, la cuál aparece escrita de forma diferente en ambos sitios:



Si bien no es exacto el procedimiento, quizás se podría mejorar con el uso de una Inteligencia Artificial que reconozca si ambos textos son iguales. En cuanto a la información extraída hasta el día de la fecha no hubo errores de duplicación.

Resto de campos

A continuación se describe en forma breve las estrategias tomadas para mezclar el resto de los campos de la película, cabe recalcar que cada campo se normalizó hasta la estapa 3 del pipeline descrito anteriormente, ya que determinamos que no eran campos tan complejos como el identificador.

- **Lenguajes**, **géneros**, **orígenes**, **directores**, **actores**, **shows**: Se aplicó unión de conjuntos (para eliminar repetidos).
- **Duración**: Se optó por la máxima duración, ya que la diferencia entre las duraciones de las películas existentes es mínima.
- **Trailer**: Se optó por tomar el primero que se procese.
- **Sinopsis**: Se optó por la sinópsis con mayor número de caracteres.
- **Calificación**: En este caso, al presentar multiples variaciones en ambas páginas se decidió armar una tabla para normalizar las diferentes nomenclaturas de una misma calificación. En caso de que las páginas difieran en la calificación de una pelicula se lo tratará como indeterminado.
- *Nota* : Especialmente con el campo orígenes tuvimos un inconveniente para el cual no encontramos una forma simple de resolver. Notamos en los resultados del merge que se podían dar casos en los que las dos páginas les den un nombre distinto al mismo país (ej: EEUU y Estados Unidos).

Estrategia de mergeo

Para la implementación de la clase *MovieRepository* decidimos aplicar el patrón Strategy a la hora de realizar el mergeo de la información. Esto nos brinda la posibilidad de optar por diferentes mecanismos de merge incluyendo todos los beneficios que el patrón Strategy nos brinda. Para esta entrega desarrollamos una única estrategia que intenta mezclar la información de ambas páginas, tratando de maximizar y completar la información obtenida. Otras posibles implementaciónes podrían haber sido darle preferencia a algúna de las dos páginas en especifico o la busqueda de información en páginas de terceros como IMBD.