Movie Web Scraper

Abstract

Report about design and development of a Web scraper, a Python bot, thought for retrieving data to populate and update movie info collections on Movienator MongoDB document-oriented database .

1 Scope of Scraper Bot

The aim of this bot, developed in Python (3.7) language, is to solve the problem of lack of data, indeed Movienator app needs a lot of movie information and details to work sensibly, so the scraper works providing from several sources (MyMovies, IMDb ..) all types of data required, according to the ETL ("Extract-Transform-Load") paradigm.

We can identify two main uses of the scraper, as follow:

1. Find new movies and related details; a movie is "new" if not already stored as a MongoDB collection.
2. Update movies with most recent data, this operation is carried out periodically.

Below are listed all the attributes collected by the scraper for each movie:

- movie title;
- movie genre;
- URL to movie poster;
- movie description;
- public release date;
- aggregate rate of external source;
- extra attributes (storyline, tagline, production country, budget ..)

2 Software design according to ETL model

2.1 Data extraction

The very first step is data extraction, made possible by sequential HTTP requests (performed by Requests module), followed by the utilization of Beautiful Soup, a Python library for pulling data out of HTML and XML files. Data, among different http responses, are retrieved by a JSON-LD tag, contained within html body of every single movie page.

```
def get_ld_json(self, url: str) -> dict:#this method return parsed info as a dict starting from scraped page
    parser = "html.parser"
    req = requests.get(url)
    print(req.encoding)
    if self.source == "mymovies":
        text = str(req.content, 'UTF-8', errors='replace')
    else:
        text = req.text
    soup = BeautifulSoup(text, parser)

    if self.source == "imdb":
        ld_json = soup.find("script", {"type":"application/ld+json"})
        if ld_json is not None:
            json_dict = json.loads(normalize_json_string("".join(ld_json.contents)))
            json_dict.update(self.find_additional_info_from_imdb(soup))
            return json_dict
        else:
            return None
    elif self.source == "mymovies":
        json_scripts = soup.findAll("script", {"type":"application/ld+json"})
        for script in json_scripts:
            obj = json.loads(normalize_json_string(script.getText()))
            if ("name" in obj.keys() and "genre" in obj.keys() ):
                return obj#return json.loads("".join( soup.findAll("script", {"type":"application/ld+json"})[
            else:
                continue
        return None
```

Figure 1: Data extraction, performed by Requests and BeautifulSoup methods

Web indexing is made agile through two separate techniques:

1. Across IMDb pages, every movie page is uniquely identified by imdb_id, exploiting a standard URL format, "*https://www.imdb.com/title/ttxxxxxx/*", where "*xxxxxx*" is the integer id to locate the source.

2. Exploiting MyMovies Web API, it's possible to find movies URL starting from their titles; "*https://www.mymovies.it/ricerca/ricerca.php?limit=true&q=yyyyyyy*", this is the query format, where the real argument (passed by GET method) is actually just one, "*q*" and "*yyyyyyy*" rapresents movie title.

2.2 Data transformation

All the input data, retrieved from Web, are parsed into Python dictionaries, exploiting json library methods applied to deserialized data content, which is obtained by the *"soup"* in the previous phase.

```
def LoadMovie(self,movieId):#movieId can be an integer ->IMDB, a string like 'https://www.mymovies.it/film/yyy

    if movieId.startswith("https://www.mymovies.it/"):
        self.source = "mymovies"
        """add tomato"""
    else:
        self.source = "imdb"
        movieId = web_sources[self.source] + str(movieId)
    #getting json results
    req_movie = self.get_ld_json(movieId)
    if req_movie == None:
        return

    #miss values manager
    for key in MovieScraper.Attributes + MovieScraper.EXTRA_ATTRIBUTES:
        if key not in req_movie.keys():
            req_movie[key]= None

    nt = {"source":str(self.source), "movie":{} } #new dict element

    #listing attributes separately
    for key in MovieScraper.Attributes + MovieScraper.EXTRA_ATTRIBUTES:
        #gestire il name nel formato opportuno per ricavare una lista di nomi compatibile rottent e mymovies
        exec("self."+key+".append(req_movie[key])" ) in locals()
        nt["movie"][key] = req_movie[key]

    new_movie_doc = nt
    self.moviesInfo.append(new_movie_doc)
    return new_movie_doc
```

Figure 2: Data parsing and standardization

In the figure above (Fig.2) we can look at the code that performs a first processing of data, retrieved by the dictionary resulting from the deserialization of the json content. Missing features are converted in properties with *None* as a value, the objetive is to deal with standard format dictionaries, solving the problem of heterogeneous data sources.

2.3 Data loading: updating the database

New scraped data, both for features update and addition, once parsed, are stored on MongoDB server by means of PyMongo API for Mongo DBMS. BSON is the standard format used to store documents and make remote procedure calls in MongoDB, luckily python dictionares are directly mapped to BSON type as object and vice versa, so serialization (writings to Mongo) and deserialization (readings from Mongo) processes are easily managed.

```
def UpdateMongoMovies(self,coll_name,nrows=20):#make use of indxes file

    mysource_p = ms.web_sources.get("mymovies",None)
    if mysource_p == None :
        raise Exception("Sorry, requested source not available")

    coll_iterator = self.getMoviesByLastScraped(nrows)

    for movie in coll_iterator:
        # mark movie as scraped to prevent other scrapers to scrape the same
        # movie
        self.db['movies'].find_one_and_update({
            '_id': movie['_id']
        }, {
            '$set': {
                'last_scraped': datetime.now()
            }
        })

        print("\n--getting movie--\n")
        #extract movie by source
        #data from mymovies
        scrape = ms.MovieScraper()
        mm_movie_info = scrape.getMovieFromMyMovie(movie)
        #data from imdb
        """aggiungere dati di imdb per film con stesso id"""
        im_movie_info = scrape.LoadMovie(movie["_id"])

        pprint(im_movie_info)
        pprint(mm_movie_info)

        #updatedinfo
        upd_dic ={'ratings':[]}
```

Figure 3: Data extraction (performed by Requests and BeautifulSoup libs)

In order to perform optimized update operations, a *last_scraped* attribute is added (and modified) every time an update operation is executed from the script: this attribute is just an index to keep trace of the movies that were not updated longer. The code above (Fig.3) shows how to process consecutive movies, getting them by *self.getMoviesByLastScraped* method and proceeding on integrating scraped data.

```python
#imdb data manager
if (im_movie_info ==None):
    print("\nnessun contenuto imdb da aggiornare\n")
else:
    #append new rate
    im_movie_aggr_info = im_movie_info["movie"]["aggregateRating"]
    if (im_movie_aggr_info !=None):
        if (im_movie_aggr_info["ratingValue"] != ''):
            ratepoints =float(im_movie_aggr_info["ratingValue"])
            newrate = {
                "source":"IMDb",
                "avgrating": ratepoints,
                "count": im_movie_aggr_info["ratingCount"],
                "weight": 0.5,
                "last_update": datetime.now()
            }
            upd_dic["ratings"].append(newrate)

    #update diffs on mongodb
    im_movie_info = im_movie_info["movie"]

    # IMDb overrides mymovies result
    if "description" in im_movie_info and im_movie_info['description']:
        upd_dic["description"] = im_movie_info["description"]

    if "image" in im_movie_info and im_movie_info['image']:
        upd_dic["poster"] = im_movie_info["image"]

    for attribute in ms.MovieScraper.EXTRA_ATTRIBUTES:
        if im_movie_info[attribute]:
            upd_dic[attribute] = im_movie_info[attribute]
```

Figure 4: IMDb data integration

```python
#mymovie data manager
if (mm_movie_info == None):
    print("\nnessun contenuto mymovie da aggiornare\n")
else:
    print(mm_movie_info)
    if ("ratings" in movie.keys()):
        #append new rate
        mm_movie_aggr_info = mm_movie_info["movie"]["aggregateRating"]
        if (mm_movie_aggr_info != None):
            if (mm_movie_aggr_info["ratingValue"] != ''):
                ratepoints = float(mm_movie_aggr_info["ratingValue"])
                newrate = {
                    "source": "MyMovies",
                    "avgrating": ratepoints,
                    "count": mm_movie_aggr_info["ratingCount"],
                    "weight": 1,
                    "last_update": datetime.now()
                }
                upd_dic["ratings"].append(newrate)

    if "description" in mm_movie_info:
        upd_dic["description"] = mm_movie_info["description"]
        upd_dic["description_ita"] = mm_movie_info["description"]

    if "image" in mm_movie_info:
        upd_dic["poster"] = mm_movie_info["image"][0]["url"]
```

Figure 5: MyMovies data integration

Some types of information, like *total_rating* feature, need to be computed incrementally, merging each time new scraped data coming from both MyMovies and IMDb.
In order to interact with Mongo database we exploited several methods from PyMongo module, as:

- findOne()
- updateOne()
- find_one_and_update()
- bulk_write()

All these methods take dictionaries as argument, with the exception of *bulk_write*, that is used for increasing write throughput.

```
#load updated movie ratings
operations = []
for rating in upd_dic['ratings']:
    operations += [
        UpdateOne(
            {'_id': movie['_id']},
            {'$pull': {
                "ratings": {
                    "source": rating['source']
                }
            }}
        ),
        UpdateOne(
            {'_id': movie['_id']},
            {'$push': {
                "ratings": rating
            }}
        ),
    ]

if operations:
    self.db['movies'].bulk_write(operations)

del upd_dic['ratings']

upd_dic['last_scraped'] = datetime.now()
# update other info
self.db["movies"].find_one_and_update({
        '_id': movie['_id']
    }, {
        '$set': upd_dic
})
```

Figure 6: bulk update operations