Movie Recommendation

Wei Yan

wyglauk@gmail.com

Contents

Introduction

This project demands developing an algorithm for recommending the related movies based on the given one, in order to do that, I have developed an search engine prototype with an inverted-index as the core, and using the Tfidf and cosine similarity to scoring the documents. The data this engine makes use of are the tags dataset *tags.csv* and movie information dataset *movies.csv*, to be more specific, the inverted-index is constructed based on the tags data, and recommendation is made by firstly mapping the movie to its tags and then using these tags to conduct the searching.

Dependencies

In order to run the program, the following dependencies need to be satisfied:

1. Python 3.7
2. Pandas 0.23.4
3. Numpy 1.15.4
4. Flask 1.0.2

The project is developed in a windows 10 environment, but it was also tested on a CentOS 7 platform.

Usage

The steps of running the program at local machine is as following:

1. **Set the *servingPort* term in */global_settings.py* file**, to ensure the port is not occupied by other routines. The term is shown in the following figure, which is currently set as 11111. The reason of setting the port is, the API of the program is in fact developed by using Flask and works like a web service.

```
servingPort = 11111    # the porting that inverted index is serving at
```

Fig 1. Serving port in global_settings.py

2. **Decompress the */persistance/persistence.zip* to local**, the result should look like the following figure after the decompression. Specifically, the *postings* directory contains 33170 sub directories, each directory corresponds to a tag in the lexicon. The *docInfo* file contains the information of movies, the *lastPUnitId* records the ID of last added posting unit, and the *lexicon* contains the tags and IDs of corresponding posting units. As described in the introduction, these files are generated based on the raw dataset *tags.csv* and *movies.csv*, which could be found in the directory */dataset*.

postings	02/12/2018 17:16	File folder	
docInfo	02/12/2018 17:16	File	2,844 KB
lastPUnitId	02/12/2018 17:15	File	1 KB
lexicon	02/12/2018 17:15	File	1,864 KB

Fig 2. Decompressed persisted files.

3. **Go to the root path of project and run the */Api.py* script in console with command *[python Api.py]*.** When the sentence "Running on …" is printed out in the console, the persisted inverted-index is loaded and program starts serving. An example can be seen from the following figure.

```
(C:\ProgramData\Anaconda3) C:\desktop\workspace\moive_recommendation>python Api.py
2018-12-03 14:31:01,323 [INFO] - Index.py:165 load last posting unit id
2018-12-03 14:31:01,326 [INFO] - Index.py:137 load lexicon
2018-12-03 14:31:01,820 [INFO] - Index.py:147 load posting units
2018-12-03 14:34:27,666 [INFO] - Index.py:171 load doc info
 * Running on http://127.0.0.1:11111/ (Press CTRL+C to quit)
```

Fig 3. Start running the program.

4. There are two approaches to make use of the engine to get recommendations:
    a. **Open another command line window, go to the root path of project and run the */search.py* script with command *[python search.py <movie id>]*,** which will return a json of list of recommended movie IDs:

```
(C:\ProgramData\Anaconda3) C:\desktop\workspace\moive_recommendation>python search.py 541
b'[541, 26985, 172, 95875, 1274, 5445, 7163, 76, 2571, 741, 2916, 198, 5046, 27904, 6934,
27660, 98019, 260, 6365, 4370]'
```

Fig 4. Search via console.

    b. **Open an internet explorer (e.g. Chrome) to visit the URL *http://127.0.0.1:11111/display_search?movieId=<movieId>*,** which will return a page contains the recommended movies and related movie information, use movie 541 as an example:

127.0.0.1:11111/display_search?movieId=541

<table>
<tr><th>docId</th><th>rankingScore</th><th>title</th><th>genre</th><th>tagNum</th></tr>
<tr><td>541</td><td>1480.363804</td><td>Blade_Runner_(1982)</td><td>Action|Sci-Fi|Thriller</td><td>965</td></tr>
<tr><td>26985</td><td>760.855639</td><td>Nirvana_(1997)</td><td>Action|Sci-Fi</td><td>1</td></tr>
<tr><td>172</td><td>723.218694</td><td>Johnny_Mnemonic_(1995)</td><td>Action|Sci-Fi|Thriller</td><td>105</td></tr>
<tr><td>95875</td><td>703.471206</td><td>Total_Recall_(2012)</td><td>Action|Sci-Fi|Thriller</td><td>184</td></tr>
<tr><td>1274</td><td>698.135007</td><td>Akira_(1988)</td><td>Action|Adventure|Animation|Sci-Fi</td><td>273</td></tr>
<tr><td>5445</td><td>673.495140</td><td>Minority_Report_(2002)</td><td>Action|Crime|Mystery|Sci-Fi|Thriller</td><td>459</td></tr>
<tr><td>7163</td><td>671.815677</td><td>Paycheck_(2003)</td><td>Action|Sci-Fi|Thriller</td><td>73</td></tr>
<tr><td>76</td><td>657.614123</td><td>Screamers_(1995)</td><td>Action|Sci-Fi|Thriller</td><td>38</td></tr>
<tr><td>2571</td><td>654.363520</td><td>Matrix,_The_(1999)</td><td>Action|Sci-Fi|Thriller</td><td>1425</td></tr>
<tr><td>741</td><td>650.777418</td><td>Ghost_in_the_Shell_(Kôkaku_kidôtai)_(1995)</td><td>Animation|Sci-Fi</td><td>203</td></tr>
<tr><td>2916</td><td>650.371800</td><td>Total_Recall_(1990)</td><td>Action|Adventure|Sci-Fi|Thriller</td><td>430</td></tr>
<tr><td>198</td><td>612.815257</td><td>Strange_Days_(1995)</td><td>Action|Crime|Drama|Mystery|Sci-Fi|Thriller</td><td>137</td></tr>
<tr><td>5046</td><td>590.707551</td><td>Impostor_(2002)</td><td>Action|Drama|Sci-Fi|Thriller</td><td>51</td></tr>
<tr><td>27904</td><td>565.938198</td><td>Scanner_Darkly,_A_(2006)</td><td>Animation|Drama|Mystery|Sci-Fi|Thriller</td><td>358</td></tr>
<tr><td>6934</td><td>563.621742</td><td>Matrix_Revolutions,_The_(2003)</td><td>Action|Adventure|Sci-Fi|Thriller|IMAX</td><td>274</td></tr>
<tr><td>27660</td><td>534.118386</td><td>Animatrix,_The_(2003)</td><td>Action|Animation|Drama|Sci-Fi</td><td>118</td></tr>
<tr><td>98019</td><td>522.036033</td><td>Vexille_(Bekushiru:_2077_Nihon_sakoku)_(2007)</td><td>Action|Animation|Sci-Fi</td><td>8</td></tr>
<tr><td>260</td><td>519.290036</td><td>Star_Wars:_Episode_IV_-_A_New_Hope_(1977)</td><td>Action|Adventure|Sci-Fi</td><td>771</td></tr>
<tr><td>6365</td><td>516.763604</td><td>Matrix_Reloaded,_The_(2003)</td><td>Action|Adventure|Sci-Fi|Thriller|IMAX</td><td>324</td></tr>
<tr><td>4370</td><td>514.611791</td><td>A.I._Artificial_Intelligence_(2001)</td><td>Adventure|Drama|Sci-Fi</td><td>287</td></tr>
</table>

Fig 5. Search via internet explorer.

Architecture

The architecture of this project refers to and modifies from my previous work toyEngine (Which is written with Java and could be found at https://github.com/wyangla/toyEngine), the main differences between these two projects are: **a)** how the intermediate information is stored and accessed, use the computed Tfidf value of each posting unit as an example, in this project they are temporarily maintained by the posting unit itself, however, in project toyEngine these kind of information are maintained by independent HashMaps controlled by the entity called *information_manager*; **b)** this project does not consider the real time units adding and removing, so that there is no complicated locking and units deactivation mechanism implemented.

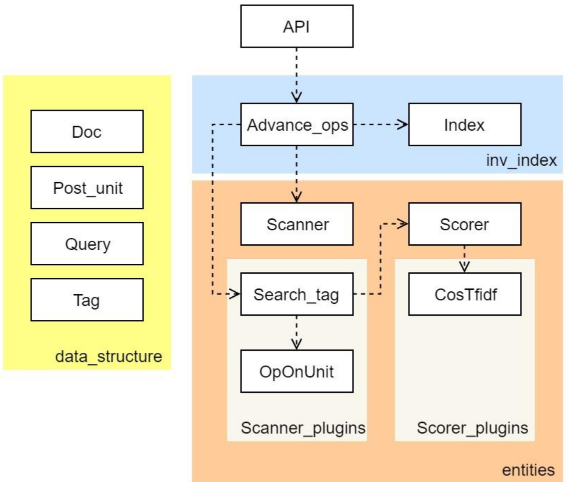

Fig 5. Overall architecture of the project.

Figure 5 shows the overall architecture of the project. It can be seen that there are mainly three packages contain most of the modules. On the left of the figure, there is the package *data_structure* which contains four basic data structures which are widely used in most of the modules of the project. Specifically, *Doc* is the data structure contains information of the movie; *Post_unit* object is the element forming the posting list, which points to each other so as to providing the fast scanning functionality; *Query* contains the information of the searched target movie, carrying the information like related tags and corresponding frequency value, etc.; *Tag* object denotes the tag which carries the information like IDs of the related posting units, etc. As described in previous paragraph, the intermediate information like the contributed Tfidf score of each posting unit is temporarily carried by the *Post_unit* object itself, which could be used for calculating the length of vector of some movies, this kind of information is not persisted (i.e. will be lost when the program exited) due to they are not really the permanent information that would be fixed for a long time, however it can easily be persisted by do few modifications to the *flatten / deflatten* methods in the basic data structures.

Fig 6. Data flow logic of the project.

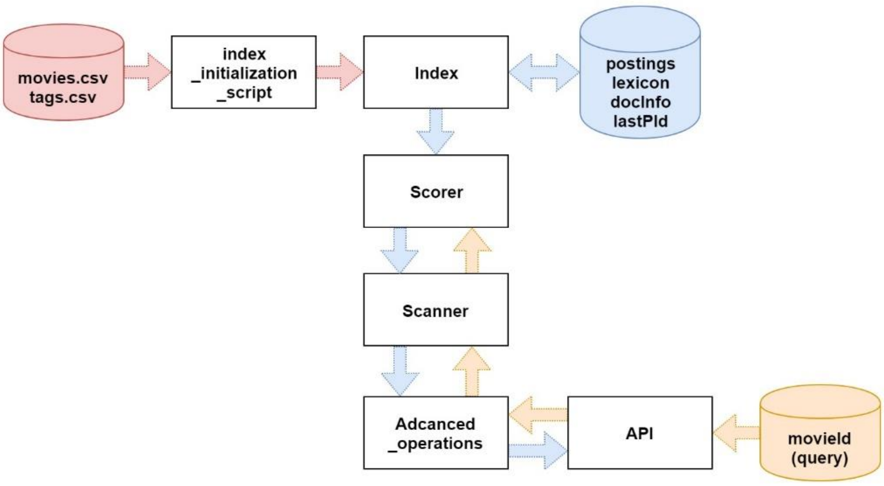

Figure 6 shows the data flow logic of the project. To be more specific, **a)** firstly the raw datasets are processed by the script */scripts/index_initialization_script* to generate the *post_unit* objects and *doc* objects, then **b)** these objects are feed into the *index*, to be more specific, into 3 dictionaries *postings*, *lexicon* and *docInfo* of *Index* object. The *Index* provides functionalities of adding new data to construct the inverted index (which consists of the previous three dictionaries), persisting the constructed inverted index and loading the persisted data back to reconstruct the inverted index. So that, **c)** after the construction of inverted index, the *Index* persists the related dictionaries to local file system. When the user runs the */Api.py* script to start the service, **d)** the persisted data are loaded back into memory to reconstruct the inverted index. Then, **e)** when the user passes one movie ID to API to get the recommendations, **f)** the query is passed to the *Advanced_operations,* **g - h)** which in turn makes use of the *Scanner* and *Scorer* to calculate scores on each posting unit, and the calculated scores are collected and passed back to the *Advanced_operations*, in which **i)** the movie information are integrated with the ranking scores to generate the message for displaying, finally this message is passed to the API and returned to the user.

Correctness

In order to ensure the correctness of the program, during the developing process, testing modules are developed and used, which could be found in directory */tests*. However, as the developing process going these tests are not ensured to be always working as they are not maintained continually. But in order to ensure the cosine similarity among Tfidf vector representations is correct, I have made some manually calculation at the end of script */scripts/index_initialization_script.ipynb*, the results is shown as below:

```
total movie number 19501

movieId 32943
tags
    mike_leigh
        df: 15  tf: 2
    movielens_top_pick
        df: 49  tf: 1
    criterion
        df: 905 tf: 1

Cosine similarity of Tfidf vector of
    32943 to 32943 equals 22.852446507866762
    32943 to 106048 equals 12.540948417736711
```

```
movieId 106048
tags
    mike_leigh
        df: 15  tf: 1
    realistic
        df: 124 tf: 1
    tv_movie
        df: 7   tf: 1

Cosine similarity of Tfidf vector of
    106048 to 106048 equals 17.06505518795686
    106048 to 32943 equals 9.36494815924087
```

Fig 7. Manually calculation results.

The manually calculation results are identical to the results of the program so as have proved the correctness of the algorithm. What needs to be mentioned is **the reason of cosine similarity score is not with in interval [0, 1] here is because it is not normalized with the length of query, instead, it is only normalized with the document length, this makes no difference to the recommendation result as the query length is the same for all the documents**.

Potential Evaluation Strategy

The most accurate evaluation strategy is **a)** firstly manually create a set of queries and best recommending results, **b)** use the algorithm to generate the ranking results of each query and compute metrics (e.g. MRR, NDCG) on each ranking result, then average the same metrics on all queries as the overall performance of algorithm. However this strategy demands huge human effort.

An alternative evaluation strategy is using the recommendation of MovieLen as the perfect ranking, then based on that to evaluate the performance of the proposed algorithm. This approach demands developing crawler and collecting the data.

The development of evaluation framework and data collection are not involved in this project due to the limit of time.

Total Time Consumption

Coding: 3 days, report writing: 1day.