Project code (1 line)

Link: https://github.com/jingjie-wan/SI507

Data sources (1/2 - 1 page)

**1 IMDb Top 250 Movies**

The first data source is a webpage of the top-rated 250 movies in Internet Movie Database(IMDb).

Source: http://www.imdb.com/chart/top

Format: HTML

Access & Cache:

I accessed the data by scraping. I use **requests** library to make a HTTP request to the above URL. Then I use **BeautifulSoup** to parse and extract movie information from the response content.

I used cache so HTTP request to the website only have to be made once. To be more specific, I saved the text of the web response in a html file. If the file already exists locally, the program directly read the file.

Summary of Data:

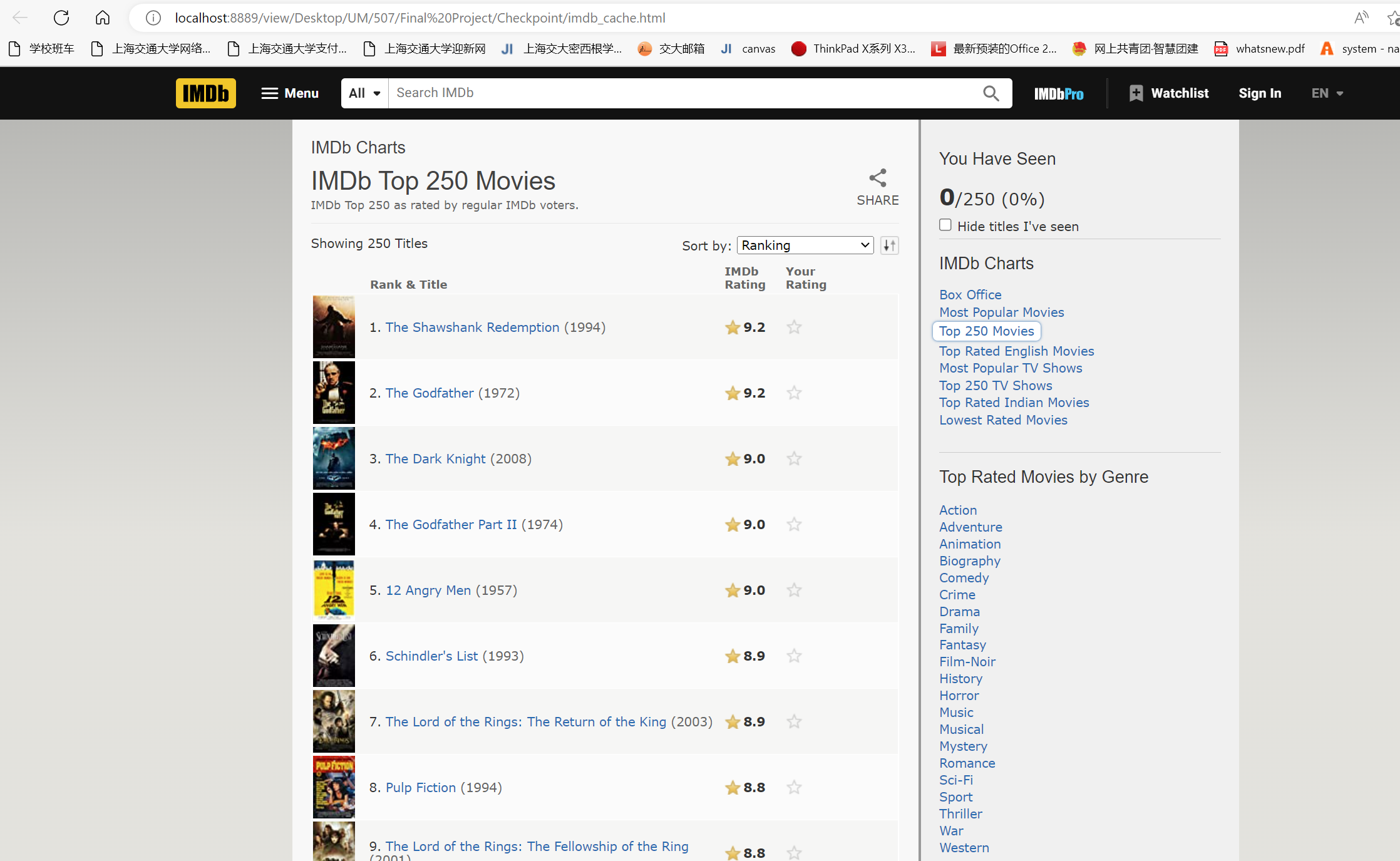
* #Records available: 250
* #Records retrieved: 250
* Description:

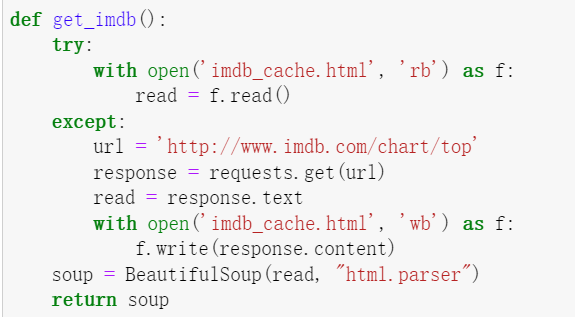
Every record contains basic information of one of the top 250 movies. Here are the important fields of each record:

* The rank (of rating) of the movie (*place*)
* The IMDB number of the movie, which is unique and Is used widely to identify movies in many databases (*IMDB\_number*)
* The title of the movie (*title*)
* The rating of the movie on IMDB website (*rating*)
* The year that the movie released (*year*)
* The director of the movie (*direct*)
* The main actors and actress (in list) (*stars*)

Evidence of Caching:

The snapshots of the cache file(imdb\_cache.html) and my code that implement caching are shown below:





**2 Open Movie Database**

Since the first data source only provides limited information, I used the second database which contains detailed information about movies to complement the movie data. The two datasets can be merged by the IMDB number.

Source: https://www.omdbapi.com/

Format: JSON

Access & Cache:

I accessed the data by Web API which requires API key (http://www.omdbapi.com/?apikey=[yourkey]&). I used **requests** library again while I can only use IMDB number from the first database to make request to one movie's data at a time. I converted these json data to dictionary (which keys are the IMDB numbers of the movies).

To cache the data, I turned the dictionary containing data of each movies to json format and saved it as json file. If the program finds the file locally, it would directly read the file instead of making all the requests again.

Summary of Data:

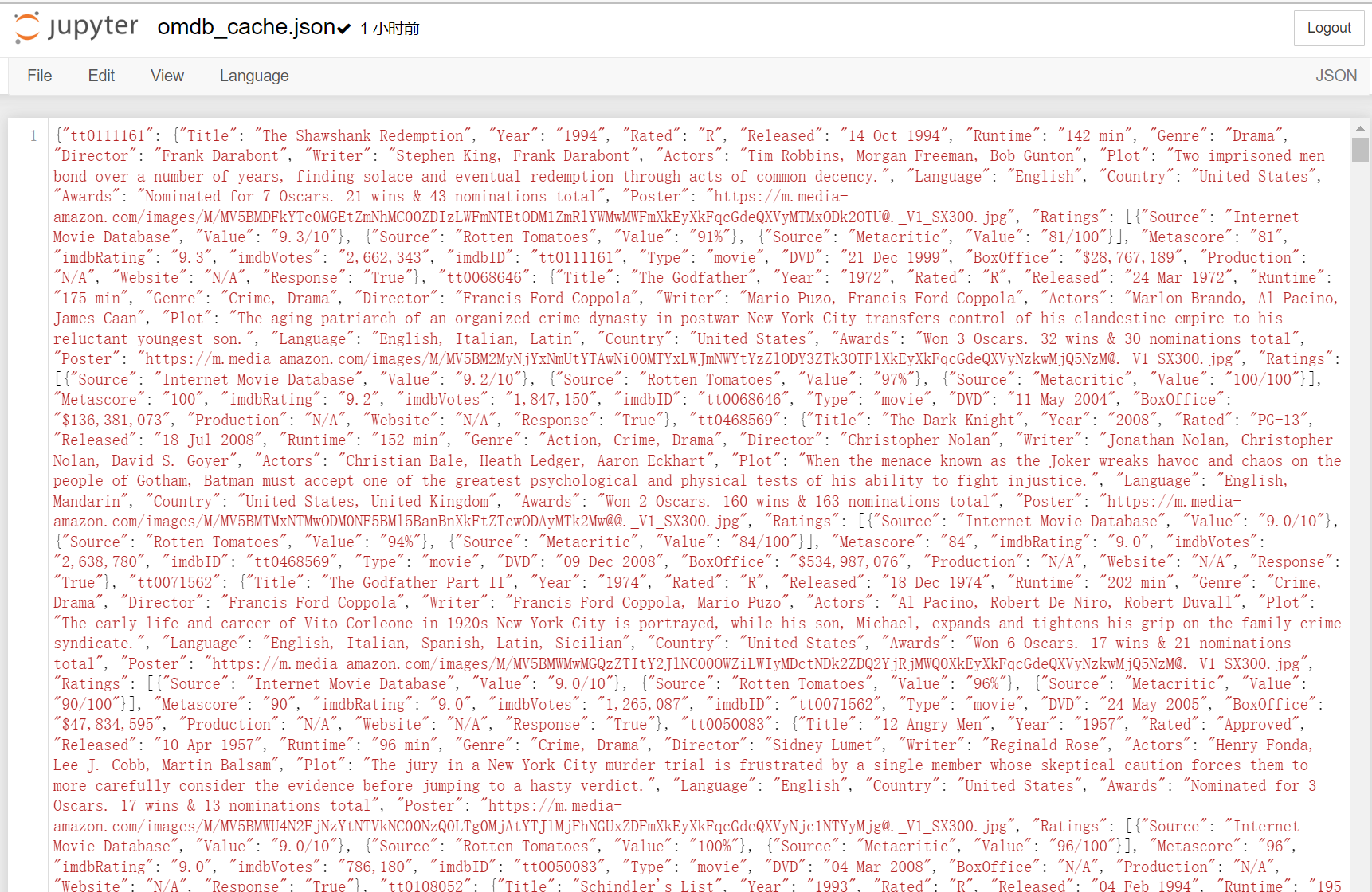
* #Records available: About 1 million
* #Records retrieved: 250
* Description:

Every record contains detailed information a movie, including runtime, awards, rated, genre and so on. Here are the important fields of each record:

* The IMDB number of the movie (enable it to be merged with the first database to get complete information) (*IMDB\_number*)
* The runtime of the movie (*runtime*)
* The genres of the movie (in list) (*genre*)
* Languages and countries of the movie (in list) (*language*, *country*)
* whether the film has been nominated and won oscar award (*nominated\_oscar*, *won\_oscar*)
* Box office (in dollar) (*box\_office*)

Evidence of Caching:

The snapshots of the cache file(omdb\_cache.json) and my code that implement caching are shown below:





Data Structure (1/2 - 1 page)

I used tree as the data structure in this project. The tree is stored as tuple, whose first element is a question, second element is the left subtree containing questions and movies if the answer to the question is yes, and third element is the right subtree containing questions and movies if the answer to the question is no. When the questions are exhausted or there is only one movie in a subtree, the subtree would be a leaf in the form of (movies, None, None), where 'movies' is a string containing the information of all movies in the leaf.

Data is organized into tree in Tree.json by the following steps by Construct\_tree.py:

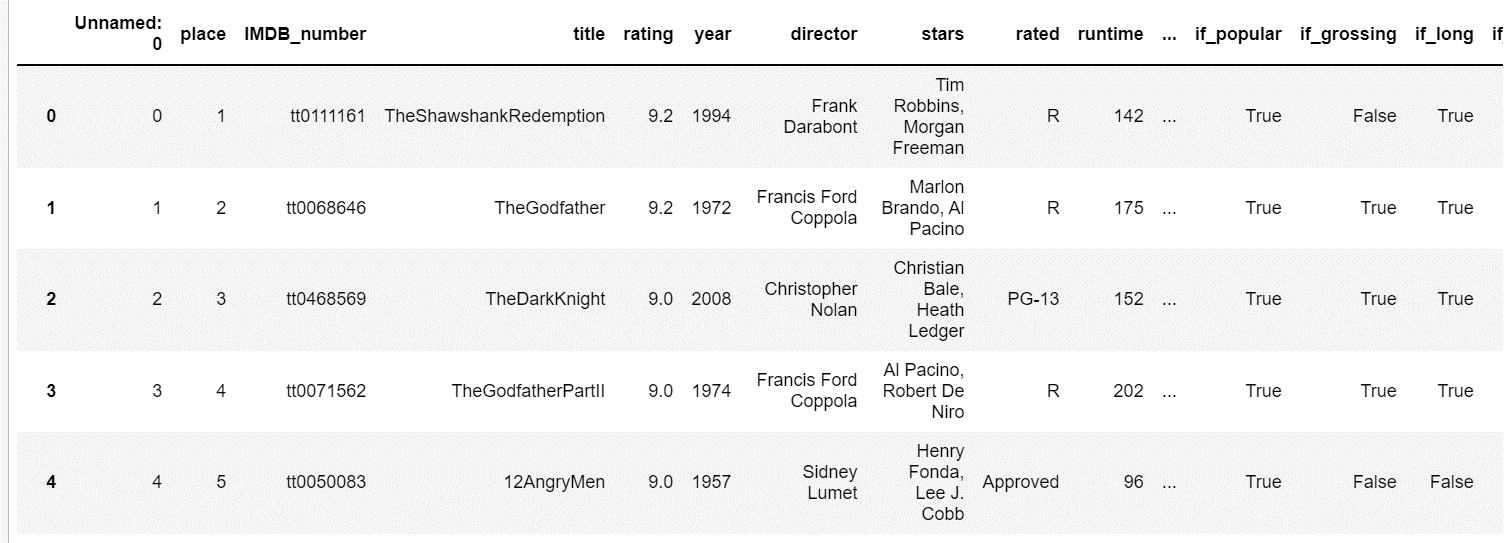
Pre-process the dataset collected\_data.csv which containing information of top 250 movies into a dataset df, which columns are boolean values of questions about movies according to the original data.

Initialize a tuple with (df, None, None), then replace the first element df with the first question to ask the user, the second element with the dataset df1 of corresponding movies when answer to the first question is 'yes' (True in values), and the third element with df2 with False in values. Then replace the second and third elements with tuples of the second question and the corresponding datasets recursively. The recursion stops when questions are exhausted or there is 0 or 1 movie in a subtree.

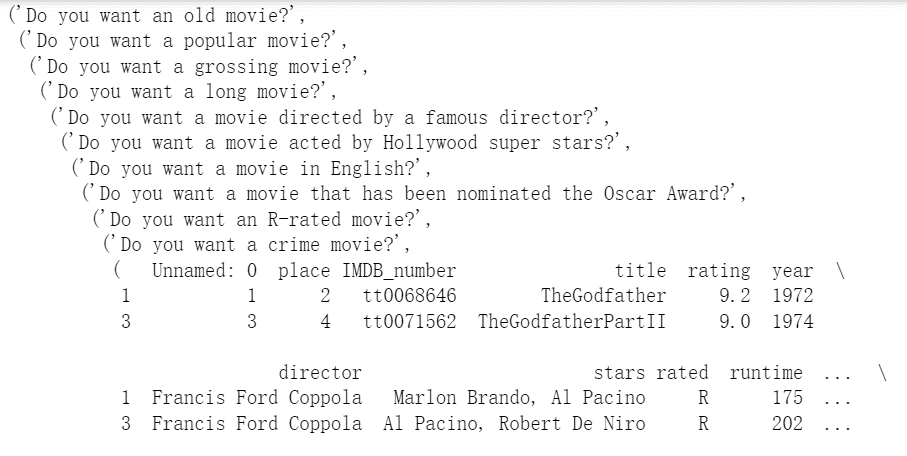
To make it easier to read, when saving the tree into file, I turn the tuple into strings and the structure of the tree is prompted by lines of 'Internal node' and 'Leaf'. I also turn the datasets into strings containing information of movies.

Screenshots:

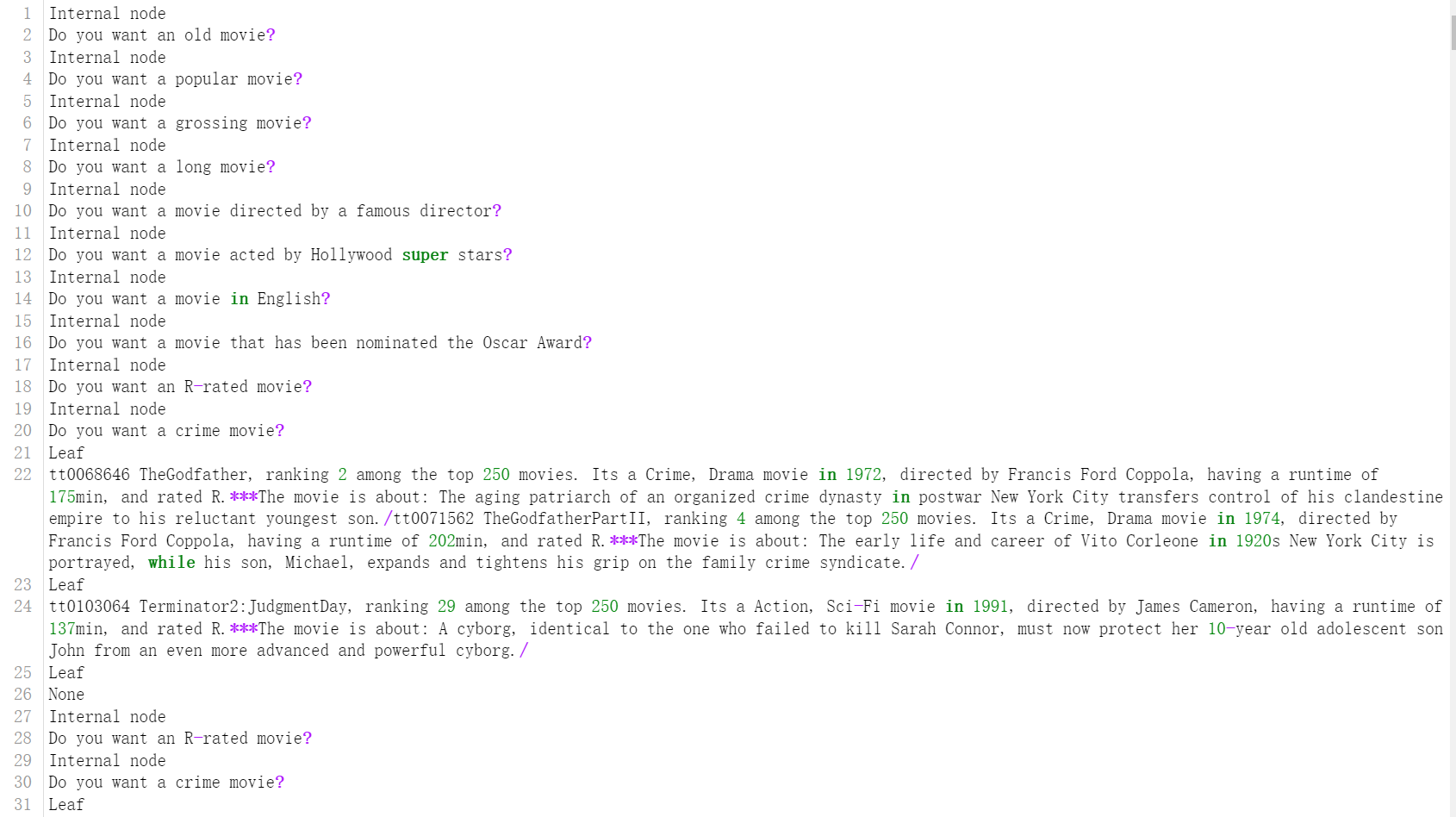
* Data (the first several columns are information of movies, followed by columns of boolean used to construct the tree)



* Data structure:
  + Tree shown in tuple (part of )



* + Tree shown in string (in json file) (part of )



Interaction and Presentation Plans (1/2 page)

After answering a series of questions about the requirements for movies, the user will be given four options for displaying and selecting the recommended movies.

* Firstly, two options are given: (1) see the recommended movies in simple mode (just the titles); (2) in detailed mode (including their titles, places among 250 movies, genres, released year, directors, runtime, etc.)
* Next, another two options are given: (1) see the plot of a specific movie; (2) launching a browser which jumps to the IMDB website of a specific move.

and JSON file with your graphs or trees

I mainly used command line prompts for interaction and presentation. Launching a browser that jumps to a specific movie website is also used as a presentation method.