Movie Recommender Online System

Advanced UI/UX based front-end app design

Jun Yin <jy2528@columbia.edu>, Jingmei Zhao <jingmei.zhao@columbia.edu>

Hadoop/Spark based backend event processing engine & recommender and audience profile

Mick Lin <cl3469@columbia.edu>, Dingyu Yao <dy2307@columbia.edu>, Jingtao Zhu <jz2664@columbia.edu>, Yitong Wang <yw2786@columbia.edu>, Sihan Wu <sw3013@columbia.edu>

Content meta-data creation from image, voice, text analytics

Yitong Xu <yx2376@columbia.edu>, Ching-Hui Hsu <ch3230@columbia.edu>, Yaning Yu <yy2723@columbia.edu>

Demo video link: https://www.youtube.com/watch?v=cs7U9zyJlho

Github repository: https://github.com/micklinISgood/Movie-Scene-Recommend, https://github.com/YaningForDS/Movie-Recommendation

Abstract—**Use the MovieLens dataset to build a movie recommender using collaborative filtering with Spark's Alternating Least Squares implementation. First, we got and parse movies and ratings data into Spark RDDs. Then we build and use the recommender and persist it for later use in our on-line recommender system.**

Keywords-movie recommendation; Collaborative Filtering; Alternating Least Squares; Spark.

I. Introduction

There is no denying that internet has dominated human lives everyday, and users/consumers has to face the problem of too much choice. To help the users/consumers cope with this information explosion, large quantities of recommendation systems are deployed. For example, a number of such online recommendation systems implemented and used are the recommendation system for books at Amazon.com , for movies at MovieLens.org, CDs at CDNow.com (from Amazon.com), etc. A glimpse of the profit of some websites is shown in table below:

Netflix	2/3rd of the movies watched are recommended
Google News	recommendations generate 38% more click-troughs
Amazon	35% sales from recommendations
Choicestream	28% of the people would buy more music if they found what they liked

Table 1 (Companies benefit through recommendation system)

The Recommender Systems generate recommendations which are partially accepted by the user, this in return provides an implicit or explicit feedback immediately or at a next stage. The actions of the users and their feedbacks can be stored in the recommender database and may be used for generating new recommendations in the next user-system interactions. The economic potential of theses recommender systems have led some of the biggest e-commerce websites (like Amazon.com, snapdeal.com) and the online movie rental company Netflix to make these systems a salient part of their websites. The personalized web recommendation systems are recently applied to provide different types of customized information to their respective users. In this paper, we want to build a movie online recommender system based on collaborative filtering with Spark's Alternating Least Squares.

II. Recommendation Dataset

MovieLens 20M dataset contains 100004 ratings and 1296 tag applications across 9125 movies. These data were created by 671 users between January 09, 1995 and October 16, 2016. This dataset was generated on October 17, 2016. Users were selected at random for inclusion.

All selected users had rated at least 20 movies. Movie information is contained in the file movies.csv. Each line of this file has the following format:movieId,title,genres. All ratings are contained in the file ratings.csv. Each line of this file has the format:userId,movieId,rating,timestamp.

Data model:


```
{
  "event": "watch_interval",
  "watch_interval": "20.008688:21.172696",
  "uid": "cl3469@columbia.edu",
  "remote_addr": "160.39.12.174",
  "mid": "7",
  "epoch": 1481127735559
}

{
  "event": "click_video",
  "epoch": 1481127679816,
  "remote_addr": "160.39.12.174",
  "uid": "cl3469@columbia.edu",
  "mid": "7"
}

{
  "event": "rec_list",
  "rec_list": ["7", "128648", "99822", "84374", "113775"],
  "epoch": 1482187923192,
  "uid": "lol",
  "remote_addr": "160.39.142.183"
}
```

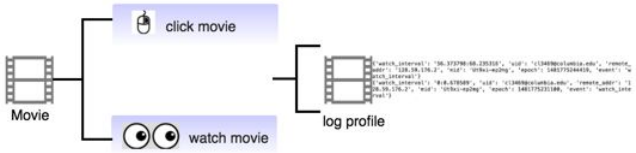



Figure 1 (Data Model)

III. Collaborative Filtering

Collaborative Filtering (CF) is a method of making automatic predictions about the interests of a user by learning its preferences (or taste) based on information of his engagements with a set of available items, along with other users' engagements with the same set of items.

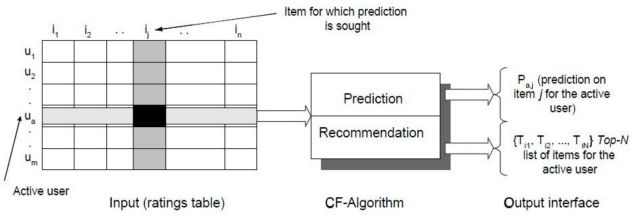


Figure 2 (Overview of collaborative filtering process)

We have users u for items i matrix as in the following:

$$Q_{ui} = \begin{cases} r & \text{if user u rate item i} \\ 0 & \text{if user u did not rate item i} \end{cases}$$

where r is what rating values can be.
Depend on only the movies that have ratings from the users and do not make any assumption around the movies that are not rated in the recommendation. Let's call this weight matrix as such:

$$w_{ui} = \begin{cases} 0 & \text{if } q_{ui} = 0 \\ 1 & \text{else} \end{cases}$$

Then, cost functions that we are trying to minimize is in the following:

$$J(x_u) = (q_u - x_u Y) W_u (q_u - x_u Y)^T + \lambda x_u x_u^T$$

$$J(y_i) = (q_i - X y_i) W_i (q_i - X y_i)^T + \lambda y_i y_i^T$$

Solutions for factor vectors are given as follows:

$$x_u = (Y W_u Y^T + \lambda I)^{-1} Y W_u q_u$$

$$y_i = (X^T W i X + \lambda I)^{-1} X^T W_i q_i$$

In the regularization, we may want to incorporate both factor matrices in the update rules as well if we want to be more restrictive.

IV. Pyspark movie recommendation

The design of our movie recommendation system is based on users' behavior by counting times of click and accumulating watch intervals. Combining two factors to give a value of rating to each movie that the user visited. This map of rating, uid{mid{rating}}, is then feed to the movie recommendation computation in spark.
At the beginning, we assign all ratings to 0 to each video for a user. (We host 36 movie trailers in AWS S3 bucket, so the size of map is usr*36)

```
rate_map = defaultdict(lambda: defaultdict( lambda:0))
```

The rating of a movie can be influenced by two factors, user's watch time and click time. The watch time is evaluated from the ratio of the movie been watched. Given a movie m1 with 2mins' length, 120 seconds (S), if the user i watch 30 seconds (Q), 30/120 (Q/S) = 1/4. Here, we can accumulate a unit of ¼ towards the rating of this movie for user i.
Moreover, we capture the user's action of clicking. Every time the user watch a movie (click on the movie), we append a unit to the movie (mid) in his rating list. That is, if the user i watch Q seconds, the rating for this movie by this user is now 1+Q/S.
Finally, a normalized rating map is used to feed spark movie recommendation engine.

The main computation and evaluation is done by spark in our movie recommendation engine. Data feed in engine includes ratings.csv, movies.csv, and the real-time rating map from user's action. We use rating.csv as the base reference for tanning model, it contains 100004 ratings information, and formatted as: userId, movieId, rating, timestamp. The movies.csv dataset includes 9125 movies, and formatted as: movieId, title, genres, where the recommendation engine recommend movies from.
First, we parse the movies.csv and ratings.csv to two RDDs. For each line in the ratings dataset, we create a tuple of (userId, movieId, Rating), the timestamp is dropped because irrelevant. For each line in the movies dataset, we create a tuple of (moiveId, title, genres). We use Python split() to parse each lines.

```
ratings_raw_RDD = self.sc.textFile('ratings.csv')
ratings_raw_data_header = ratings_raw_RDD.take(1)[0]
self.ratings_RDD = ratings_raw_RDD\
        .filter(lambda line: line!=ratings_raw_data_header)\
        .map(lambda line: line.split(","))\
        .map(lambda tokens:(int(tokens[0]),int(tokens[1]),\
        float(tokens[2]))).cache()

movies_raw_RDD = self.sc.textFile('movies.csv')
movies_raw_data_header = movies_raw_RDD.take(1)[0]
self.movies_RDD = movies_raw_RDD\
        .filter(lambda line: line!=movies_raw_data_header)\
        .map(lambda line: line.split(","))\
        .map(lambda tokens: (int(tokens[0]),tokens[1],\
        tokens[2])).cache()
self.movies_titles_RDD = self.movies_RDD\
        .map(lambda x: (int(x[0]),x[1]))
self.movies_genres = self.movies_RDD\
        .map(lambda x: (int(x[0]),x[2]))
```

Then, we get to train our initial model using ALS. The parameters we used are: rank = 8, seed = 5L, iterations = 10, regularization_parameter = 0.1

```
def __train_model(self):
        self.model = ALS.train(self.ratings_RDD, self.rank,\
                seed=Self.seed,\
                iterations=self.iterations,\
                lambda_=self.regularization_parameter)
```

Besides, we also need a pre-calculated movies rating counts. We count the number of ratings per movie to give recommendations of movies with a certain minimum number of ratings, updates the movies ratings counts from the current data self.ratings_RDD.

```
def __count_and_average_ratings(self):
        movie_ID_with_ratings_RDD = self.ratings_RDD\
                .map(lambda x: (x[1], x[2])).groupByKey()
        movie_ID_with_avg_ratings_RDD \
                =movie_ID_with_ratings_RDD\
                .map(get_counts_and_averages)
        self.movies_rating_counts_RDD\
                = movie_ID_with_avg_ratings_RDD\
                .map(lambda x: (x[0], x[1][0]))
```

To make recommendation based on user's behavior, instead of using this model every time, we need to train the model again but including the new user's preferences to compare them with all other users in the complete dataset. In another word, the recommendation model needs to be trained every time we have updated new user ratings. This of course makes the process expensive as every single update from user can trigger the model training process. Spark's high scalability is the solution, and that's the biggest reason we implement the recommendation engine in Spark.

Every time the engine receive ratings from the new user (myRatings), we put them in a new RDD. Assign 0 as the new user id because it's not used in MovieLens dataset.

```
new_user_ID = 0
myRatingsRDD = self.sc.parallelize(myRatings)
```

We use Spark's union() transformation to add the new ratings information to ratings RDD, and compute the average ratings again.

```
self.ratings_RDD = self.ratings_RDD.union(myRatingsRDD)
self.__count_and_average_ratings()
```

Then, training the ALS model again with the new rating and using the same parameters mentioned above.

```
self.__train_model()
```

Take movie IDs and keep those not on the ID list for the computations. Using new_user_unrated_movies_RDD with the spark function predictall() to predict new ratings for the movies list.

```
new_user_ratings_ids = map(lambda x: x[1], myRatings)
new_user_unrated_movies_RDD = (self.movies_RDD\
        .filter(lambda x: x[0] not in new_user_ratings_ids)\
        .map(lambda x: (new_user_ID, x[0])))
new_user_recommendations_RDD = self.model\
        .predictAll(new_user_unrated_movies_RDD)
```

Finally, the engine rearranges recommendation result, map the result to have (rating, title, genres, ratings count). The top_movie gives a 5-movie recommended list for the user.

```
self.ratings_RDD =\
         self.ratings_RDD.subtract(myRatingsRDD)
new_user_recommendations_rating_RDD =\
         new_user_recommendations_RDD\
        .map(lambda x: (x.product, x.rating))
new_user_recommendations_rating_title_and_count_RDD=
        new_user_recommendations_rating_RDD\
        .join(self.movies_titles_RDD)\
        .join(self.movies_genres)\
        .join(self.movies_rating_counts_RDD)
        .map(lambda r: (r[1][0][0][1], r[1][0][0][0], \
        r[1][0][1], r[1][1]))
top_movies =\
        new_user_recommendations_rating_title_and_count_RDD
        \
        .filter(lambda r: r[2]>=25)\
        .takeOrdered(5, key=lambda x: -x[1])
```

Above illustration gives a recommendation list in response to every new user preference captured. It's expensive to train a model every time. Therefore, we use small dataset to train the model for a quicker response.

V. Algorithm Evaluation

Alternating Least Squares rotates between fixing one of the unknowns ui or vj . When one is fixed the other can be computed by solving the least-squares problem. This approach is useful because it turns the previous non-convex problem into a quadratic that can be solved optimally. A general description of the algorithm for ALS algorithm for collaborative filtering is as follows:

Step 1 Initialize matrix V by assigning the average rating for that movie as the first row, and small random numbers for the remaining entries.

Step 2 Fix V , solve U by minimizing the RMSE function.

Step 3 Fix U, solve V by minimizing the RMSE function similarly.

Step 4 Repeat Steps 2 and 3 until convergence.

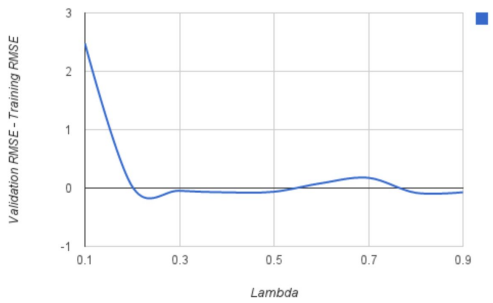


Figure 3

VI. Online Recommendation at movie and scene level

Yitong Xu, Yaning Yu, Saho Qi

Low-rank approximation SVD is appropriate for the so-called In-Matrix prediction (shown in Figure 4(a)), however, in terms of Out-of-Matrix approximation (Figure 4(b)), where movies have never been seen before, latent factor model is hard to generalize to those previously unseen and new-released movies. Therefore, we need a system that can recommend relevant and new-released movies as well as scenes to our users.

To achieve this goal, our designed system is built as follows. First, we build a model to characterize each movie into 410 themes. This is achieved by using keywords from movie overview as features and then fed into Watson's Natural Language Classifier. The data is from Allmovie.com. The keywords are extracted by probabilistic topic modeling LDA and Alchemy. By doing this, when a new movie released, we can predict its themes and do recommendation. The meta-data training process is shown in Figure 5. Therefore, we won't have the "cold start" problems for new or unseen movies. Similarly, we need to characterize each user based on his/her behaviors, for example, we utilize the log of each click and the logging information. Instead of factorizing the rating matrix and directly do recommendation, we apply the SVD to characterize the user's underlying preference. Intuitively, SVD wants to search some latent factors that can characterize both users and movies, such as, the movies' genres. Users may have special preference to certain groups of genres and movies are also grouped by different genres. Let K denotes the number of latent factors. Our group build ALS model on Spark using the MovieLens datasets, trying grid search and cross-validation for choosing the best K. It turns out that K = 40 is the best. This result roughly matches the total number of genres, which is 44, in the data. We also set K = 40 as the number of topics in our LDA model.

In terms of scene-level recommendation, the algorithm work as follows. First, we will log the real-time click behavior of a given user, pass the scene-level time interval to the backend. Then we will cut the each interval in the "movie" into several images, and parse all images by ResNet50 and Waston Visual Recognition, group all extracted keywords, feed the keywords to Waston Natural language Classifier (NLC). The NLC will tag each scene with some themes as defined above. Finally, given user's' preference on themes, we would be able to offer our user not only the a list of movies, but also a list of scene.

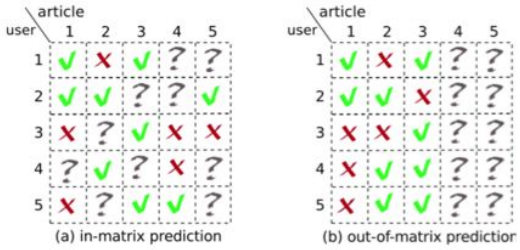


Figure 4 Illustration of the two tasks for scientific article recommendation systems.

VII. Cloud Technology

The technology we use during the implement process including: EC2, SQS, SNS, EMR, Apache Spark, S3, ElasticSearch, Flask.

Users generate data by clicking movies or watch movie intervals, which are parsed into log file and stored into SQS via websocket engine and then cached into ElasticSearch. EMR cluster will get data from ES and send it to our Recommendation engine using spark to make recommendations and the result will be sent to the front end via websocket engine.

The whole process is shown as below:

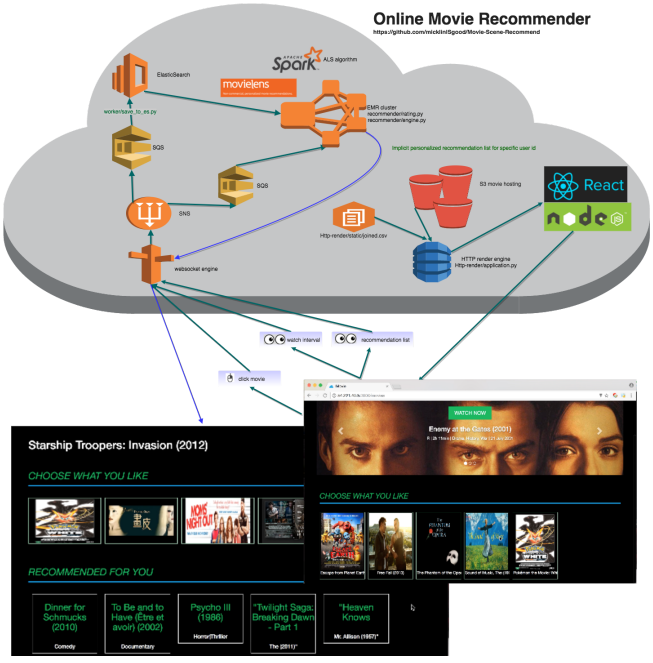


Figure 5 System flow process

VIII. Code structure

Hadoop/Spark based backend event processing engine

& Recommender and audience profile

HTTP render-engine

- User watch_interval event catch: we use html5 <video> tag to play the videos which we hosted on S3. video tag api provides onplay , onpause, ontimeupdate and onseeked callback to listen on. However, when onseeked is triggered, the video's current time will ping to the seeked position. We desired to get the final time position before the video was seeked. Here, we keep an array to push all the current time of the video while ontimeupdate is triggered. Therefore, when onseeked is triggered, we can look up the array to get the final time position before seeked. Below is the code sample. And we pass the interval with key "watch_interval" to backend, where 3rd time from the end of array is the time before seeked is triggered. Full code.

```
data["watch_interval"]=start+":"+last_m[la
st_m.length-3];
```

- In-memory Random recommendation list by Flask: We load our 36 movie meta data which is already joined with MovieLen's id to memory and generate a random list from it. Here is our schema {"category","snapshot","name","vid","length","S3 playable url", "mid"}. For each category, we have 4 movies. The random recommendation list's input is mid, also the id in MovieLen, and it's output is a list without itself. To keep the list relevant, we set the first element of the list is one from the same category and the rest of 4 elements are picked from the rest of 34 videos randomly. Code Section.

Websocket-engine

- Multi-session broadcast: We utilized the Flask socketIO to broadcast the asynchronous response from the recommender based on a user id which may have several sessions online. The main issue for this part initially is that we only consider that a user can only have a session at a time. In this scenario, only one session, one tab, will receive the response from the recommender if there are two tabs with the same user id browsing our application. Here is the best practice to solve this issue, where room=uid, line 7, means that emit the message to a room named uid which contains all the sessions of uid. SidToUid is a dictionary to track which session belongs which uid. Full code.

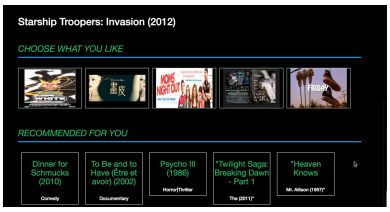
```
1.  def sendRecommendation(uid, rec_list):
2.    if uid in SidToUid.values():
3.        data = {}
4.        data["action"] = "rec"
5.        data["rec_list"] = rec_list
6.        try:
7.            emit('message',\
    json.dumps(data),room=uid)
8.        except Exception as e:
9.            cleanUserByUid(uid)
10.           print e
```

- Event sourcing via SNS+SQS: we take advantage of AWS queue service to pass out the user event data to SQS from a SNS. One SQS worker will persist all the user log for historical query if needed. The other SQS worker directly processed the event data and invoke the spark ALS recommender to give the recommendation response. Code Section.

Recommender through spark ALS

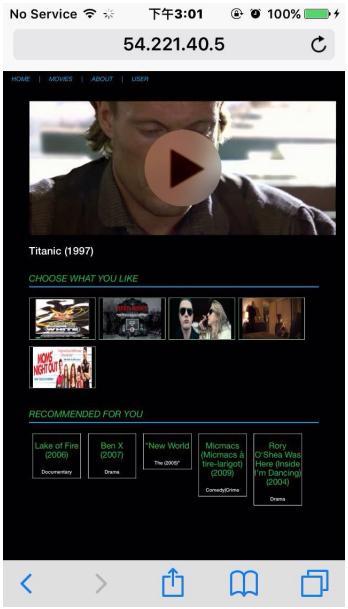
- In-memory rating accumulation & multi-thread recommendation per event: we defined a global dictionary to store the converted ratings collectively. Then, for each valid event, we will send the accumulated ratings lists of list per uid to the spark instance. Full code.
- Spark ALS recommendation: For responsiveness sake, we use movieLen small dataset to recompute the ALS model per event. Since ALS needs to give out the recommendation of unseen movies, we need to feed new rating table of a user to it after every event. Then, recompute the model to get the recommendation list. There are 9125 movies in the MovieLen small dataset. Therefore, we present the string updation in the front end because we only got 36 of them hosted. Code Section.

IX. Result



- Solution for scene Recommendation

Currently, we movie trailers to be our video candidates which can not show the property of a scene. We investigate that MovieClip did the scene clips which might be more suitable for our analysis. Therefore, we can treat a clip as short movie and apply the same rating converting mechanism and spark als recommendation mentioned above. To combine the meta-data team's work, they can analyze each clip to return a relevant recommendation list. In the end, we will have two recommendation lists, one from als and the other from meta-data team. We can simply join these two lists with an order:



1. Intersected clips will go first class, meaning content relevant and most taste-likely user preferred.
2. Rest of recommendations can go random order

Moreover, we may need to add a "watch again"section since our rating mechanism is accumulative.

Advanced Cross-platform UI Design And Implementation

Jun Yin, Dingyu Yao

In addition to building the backend, we also have a UI team who build a cross-platform web application front-end interface to ensure good user experience.

Among all the available front-end frameworks, we chose React, since it has high performance, maintainable code structure, and an active technology community. Also, we used redux to handle out front-end data flow.

Specifically, we build six pages: a homepage, a login page, a signup page, a movie displaying page, a movie viewing page with real-time recommendations, and a user profile page.

When users first open the application, they will see the a page with the banner "See what you love", a paragraph "Your personalized Movie Land", and a sign in button.

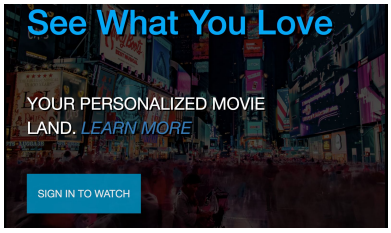


Figure 8. Homepage

Then the user will be guide to the login page asking for her/his username and password. If a users does not have an account, s/he can go to the signup page by clicking on the words below the Login button.

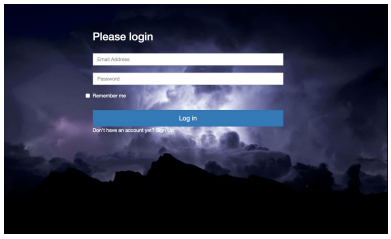


Figure 9. Login Page

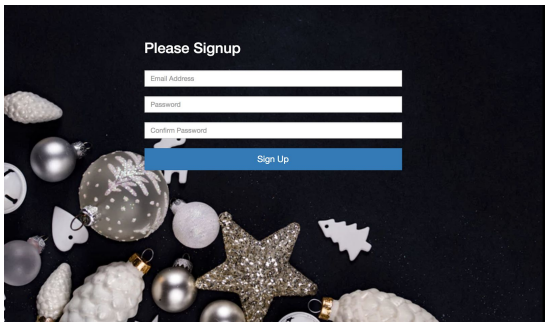


Figure 10. Signup Page

After users are logged in, they will see the movie display page. The data source of the accordion and the bottom can be different by calling different API ends.

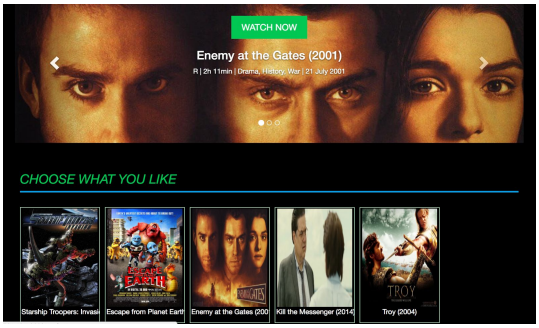


Figure 11. Movie Display Page

If the user is interested in any movie, s/he can click on it to get more information. The figure below shows an example of a movie's information.

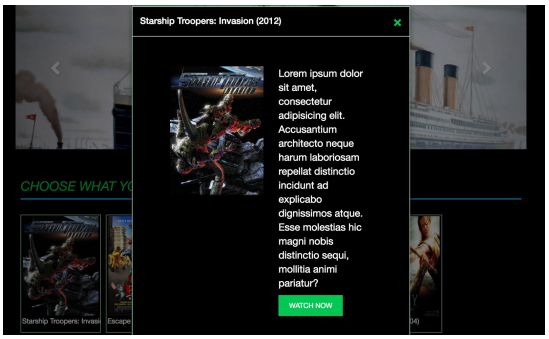


Figure 12. Movie info Dialog

If the user is interested in the movie, s/he will click on the green "Watch Now" button to enter the movie watching page.

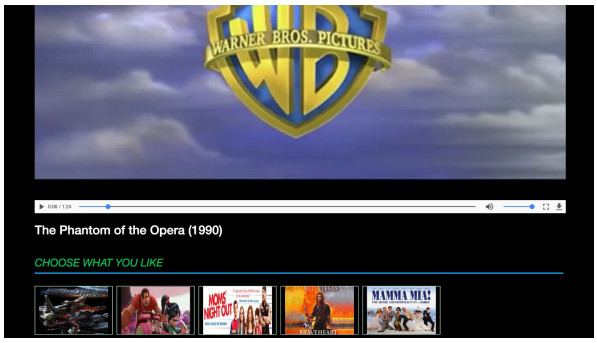


Figure 13. Movie watching page

The top of the page is an HTML movie player, and the button is a list of movies for users to choose. Each click that the user makes will be send to the backend, and based on user actions, a list of recommended movies will be rendered below.

Figure 14. recommended List

In addition to these pages, we also build a user profile page, but for the time being, we don't have real user data yet. In the future, id we have user data, we can render the page with real data.

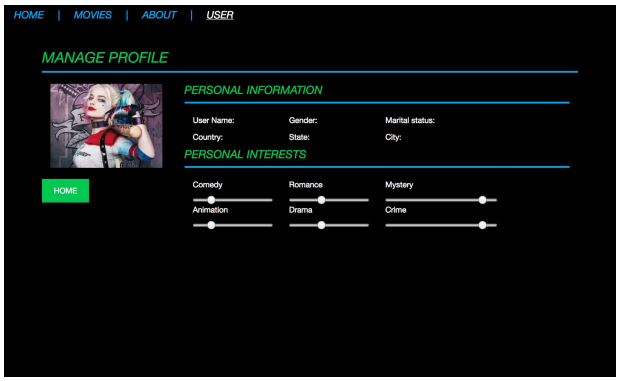


Figure 15. User Profile

It is noteworthy that we used the react-bootstrap and wrote customized css to make our web application responsive and cross-platform. Now you can have a smooth user experience no matter you are using a phone, a tablet, or a computer browser.

In the future, if there are new functionalities, we will add new components and keep enriching user experience.

X. Conclusion

In this project, we implement collaborative filtering for movie recommendation. Besides that, we create metadata for both user and movie/scene and design a more robust recommender engine to solve the 'cold start' problem. It allows a user to select his choices from a list of movies and then recommend him movies/scenes based on our recommender algorithm. By the nature of our system, it is not an easy task to evaluate the performance since there is no right or wrong recommendation; it is just a matter of opinions. Based on informal evaluations that we carried out over a small set of users we got a positive response from them. We would like to have a larger data set that will enable more meaningful results using our system.