Assignment 2: Contextual Advertising Movie System

Student Name	Student Number	Course	Assignment	Date
Ted Kachulis	100970278	COMP4601	A2: Recommender	April 2nd '19

System Setup

1. Copy *'archive'* folder from the assignment zip archive provided and paste it onto your user desktop.

2. Import the two Java (eclipse/maven) dynamic web projects *'COMP4601-A2'* and *'COMP4601-RS'* from assignment.zip into eclipse. Open them in eclipse.

   COMP4601-A2:
   Archive tools for data preparation, to be run as java application(s).

   COMP4601-RS:
   Web services and collection interfaces. To be run on Apache 9.0 server.

System Execution

1. Start a MongoDB client connection in terminal prior to launching the application.

2. Build & run '*ArchiveDataImport.java'* in *'COMP4601-A2'* as *'java application'*. Observe console, observe Robo3T, wait a minute or two.

3. Build & run '*ArchiveScoreExtractor.java'* in *'COMP4601-A2'* as *'java application'*. Observe console, observe Robo3T, can take 20-30 minutes if fresh.

4. Build & run '*ArchiveGenreExtractor.java'* in *'COMP4601-A2'* as *'java application'*. Observe console, observe Robo3T, can take 30 minutes if fresh.

5. Build & run '*ContextualAdvertisingSystem.java'* in *'COMP4601-RS'* as *'on server'* with *Apache 9.0* server configured. Observe console, ensure server start/load.

6. Test web endpoints via *'system demonstration'* section. Main endpoint can be accessed at: http://localhost:8080/COMP4601-RS/rest/rs/ .

System Demonstration

After system setup and execution, the browser is now ready to be used for demonstration. Follow these endpoint instructions below to see examples of output scenarios.

1. **.../***
   http://localhost:8080/COMP4601-RS/rest/rs/

2. **.../readme**
   http://localhost:8080/COMP4601-RS/rest/rs/readme

3. **.../context**
   http://localhost:8080/COMP4601-RS/rest/rs/context

4. **.../community**
   http://localhost:8080/COMP4601-RS/rest/rs/community

5. **.../advertising/{category}**

   Action & Classics Category URL:
   http://localhost:8080/COMP4601-RS/rest/rs/advertising/actionclassics

   Fear Fanatics Category URL:
   http://localhost:8080/COMP4601-RS/rest/rs/advertising/fearfanatics

   Laugh Lovers Category URL:
   http://localhost:8080/COMP4601-RS/rest/rs/advertising/laughlovers

6. **.../fetch/{user}/{page}**

   Action & Classics User, Looking at Horror Movie Page
   localhost:8080/COMP4601-RS/rest/rs/fetch/A100JCBNALJFAW/B006H90TLI

   Fear Fanatics User, Looking at Action Movie Page
   localhost:8080/COMP4601-RS/rest/rs/fetch/A26O0T192IBKY1/B00004CJ20

   Laugh Lovers User, Looking at Comedy Movie Page
   localhost:8080/COMP4601-RS/rest/rs/fetch/A18CMGIQZ1OAA2/6301978277

System Documentation

Feature	Briefing
ArchiveDataImport.java	When run in eclipse as a java application, this class sets up MongoDB collections for ‘users’ / ‘movies’ / ‘reviews’, then loads them in from the local archive’s html files. Data is parsed and very basic extracted data is loaded into the database, ready to be further processed by the following classes *ArchiveScoreExtractor.java* and *ArchiveGenreExtractor.java*.
/readme	Displays an html result. Though not required in submission endpoint requirements, I find it acts as a nice endpoint the testing user can refer to while navigating the website paths. Readme will display information about the available path endpoints, as well as some basic C.A.S. movie system information.
ArchiveScoreExtractor.java	When run in eclipse as a java application, this class loads the MongoDB collections for ‘users’ / ‘movies’ / ‘reviews’, takes the data, and uses it for further processing. The script also parses a word bank of positive and negative words - then uses a forM of term frequency comparison to rank reviews with an extracted score. Both the user given scoring as well as the program extracted scoring are stored for each interview. The average score for each movie is later extracted and add to the movie object data in the collections as well.

ArchiveGenreExtractor.java	When run in eclipse as a java application, this class loads the MongoDB collections for ‘users’ / ‘movies’ / ‘reviews’, then further process the data. For each genre (Action, Comedy, Horror), a small set of initial training data is provided. This training data consists of a HashMap, containing a MovieId and a predetermined genre (human interpretation). These paired sets of training data are then processed to create a term frequency model representing the genre. This model is then compared to other reviews, and a genre is extracted based on term similarity. Each review is assessed and the attribute is added to the object in the database. Movie genres are assessed and added by finding the averaged genre result of its reviews. All data is then stored in database, and can be viewed on Robo3T. After running the score extractor and genre extractor, there is no need to run them again unless the data import class is run - it clears the archive and reloads it.
/context	When this endpoint is accessed, UserProfiles are loaded from the DB and kept in a collection instance for the browser endpoints using LoadUserService. Data is further used to categorize and display users.
/community	This endpoint displays a table of users labeled by community. UserProfiles in our users collection already have several attributes, which are further processed to determine community/categorization of users. The users disposition towards a certain genre is used to assess what category of users with a similar opinion they belong to.

/advertising/{category}	Using {category} as a parameter, this endpoint responds with html based on the desired category to view. Advertisements are stored on a remote repository and are all accessible through endpoints using html string tags (img src = ***). Will display 3 ads per category accessed. Category options: *fearfanatics / laughlovers / actionclassics*. Advertisements shown are actual movies from database with associated MovieID's. When ads are shown in browser on a user/page, they are augmented with a link to view the recommended movie page.
/fetch/{user}/{page}	This endpoint displays an augmented movie page paired with a recommended banner ad based on the user who is accessing it. How the *user* parameter helps augment the system: ● Engine checks users disposition towards certain genres. ● Recommended movie ad is displayed based on this user, and the recommended movie Is linked for the user under the ad. How the *page* parameter helps augment the system: ● Engine loads movie collection from the database and holds them in a collection instance for use with the browser. The collection is built using LoadMovieService. ● Movie data is displayed with augmented content (improved score, a genre) and a link to the movie reviews.

Suggest Algorithm

I can implement the SUGGEST algorithm using the collections of data I already have, stored in the database (optionally using the collections created by the engine during endpoint access). The key information used from these collections would be the reviews. Each labelled with a user id, a score, and genre, it would be possible to use these reviews to extract a prediction representing a users disposition towards a given movie.

For a given *user profile* **UP** from the *collection of users* **UC**, we can determine a method to SUGGEST movies (pages) to a *user profile* **UP.1** based on scoring from other *user profiles* **UP.2 … UP.n** which are *similar* to **UP.1**. Similarity/user friends can be found from the *socialgraph* provided and mentioned in the assignment specifications. It is assumed that the arraylist of similar users used in the below algorithm would be extracted from the *socialgraph* <*V, E*> beforehand.

This aforementioned *similarity* would be based off of the reviews given by a user. The "similar" users to **UP.1** would be defined as users which share a collection of similar reviews, using score and sentiment from reviews as metrics for comparison scoring.

Once these collections are understood, and a similarity collection is provided, the steps of algorithm below must be followed.

1. For each *user profile* **UP.n** in *collection of users* with **UP.1 similarity**:
    a. Define new empty list **suggestedList**
    b. Retrieve current **UP** reviews
        i. Get **UP's** highest scored films (top X, SUGGEST qty needed)
        ii. Check if MovieID of these films exist in **UP.1** watched list
        iii. If not already watched, store this/these MovieID(s)
    c. Add the stored MovieID's from each similar **UP** into **suggestedList**
2. Return list of suggestions after checking all similar **UP's.**

The algorithm provides the ability to return a list of *movieIDs* that are highly scored by other users with similar characteristics to the provided *userID* input. The returned list also ensures the user has not seen (reviewed) this film before. A similar set of users could be found based of genre preference, similar review sentiment for a shared set of movies, etc.

Implemented to the *contextual advertising system*, we could better suggest the best suited movies for a given user. At the moment, a predetermined set of advertisements or ‘suggestions’ are available for a given user type. If I implemented this, and based recommended movies off of similar users favorite films, the contextual advertising system would be much more realistic - however it would take some tweaking to ensure the same movie was not recommended every time for each user, assuming their social network has not changed since a previous SUGGEST request.