Implementation and Testing

In this paper we will be going over implementation and testing of the GOAT Recommender. There are three main components in our implementation: the database infrastructure, GUI, and data loading & processing.

Implementation of the database includes a mysql database that is on a personal server that interacts with the application through a level of PHP. The PHP calls a connection each time the user sends a request. A request can be called while logging in, signing up, or inputting data. On the front end in the Java code HTTPURLConnection is being used to access the online script that has the PHP needed for that certain request. Any data that is gathered from the database and needs to be sent back to the application is saved in a JSONArray. The Java code inside the HTTPURLConnection is responsible for getting that data out and saving that data accordingly. An example of this is in the LoginController.java line 131.

Testing procedures for the database include manual tasks. After the HTTPURLConnection is established we simply start the application and see how it handles the request. Inside the Java code we catch IOExceptions and JSONExceptions to make sure that nothing is going wrong.

The PHP code all works the same. First a connection is established with the database and then if the database is receiving information then it uses SQL to insert the information. If it is getting information then it uses SQL select to get the information. All checks are done in Java when it comes to validating the information. The PHP gets tested manually through a run when the script is finished. In the beginning the script put in dummy values with the insert command and it was successful.

When it comes to implementation of the GOAT GUI we are using Netbeans 12. We have been coding in Java with Maven with the FXML JavaFX Maven Archetype. Each window is an FXML file and we are implementing SceneBuilder as a tool to develop them visually. SceneBuilder has offered a multitude of design options. Instead of typing out every line of code, SceneBuilder has been a useful tool to not only edit FXML files but to also link functions from java files in Netbeans to specific actions or events that the FXML files may elicit.

The goat\_gui project has a multitude of java class files. Many of them are classes filled with getters and setters, others are controllers for the FXML files that handle any and every action the user is given and acts appropriately. These programs hold functions for loading separate FXML files, displaying pertinent information depending on the window, and sending data from one window to the next.

Testing the GOAT GUI is easily done as all one has to do is act as if they are a user. Netbeans runs the project, and the debugging starts. If there are issues with traversing from window to window, we look to SceneBuilder and make sure that the FXML is connected to the correct controller and then if the functions are connected to the correct buttons, labels and so on. If there remains a problem, we go to the controller of the specific FXML and run through the code.

With the data processing functions, we’ve so far been able to implement a set of PHP scripts that process through our source data, filter items that don’t meet a number of programmed criteria, and output the results in our specified data schema. This turned out to be a larger task than we had originally anticipated. The movie filtering code used source data from IMDB, which was provided in a tab-separated value format. The data contained a field which told us if it was a movie or another form of media, so it was fairly simple to filter out TV shows and short films. Additionally, IMDB provides a flag for adult content, so it was fairly simple to filter out this content as well.

The video games dataset was also fairly simple to process. The entries were given as JSON objects, one entry per line in the text file, so it was simple to decode each entry. The specification for these objects were provided, so it was very easy to copy the data from the JSON object into our data schema. Like with movies, we filtered out any entry that wasn’t a video game using the provided “type” datafield in the source data, and filtered out any game that listed “adult” as being one of it’s genres.

The books dataset is significantly more complicated. The dataset is formatted in the MARC 21 bibliographic format produced by the Library of Congress. MARC 21 has to be flexible enough to include all the various materials the Library of Congress records in it’s database, which includes books and other published written materials as well as photographs, maps and globes, audio recordings, government records, and more. Thankfully this standard is extremely well documented (if somewhat obtuse due to its complexity), so we were able to implement a parsing function based off this documentation. The MARC 21 format is XML so we utilized the simple XML parser included in PHP to parse database entries. The MARC 21 format includes tags in a control field which signal what type of media an item us, so we were able to utilize that to filter out any non-printed text material in the database. The format also includes tags signaling what kind of written work an entry is, so we were able to filter out all reference publications from the dataset. After this we filtered out anything without an ISBN, which we believe has filtered the dataset down to just the “entertainment books” we wanted to feature in our application. Due to the way the MARC 21 format is structured, it was easiest to implement the main parsing function as a foreach loop, which looked through each datafield in the bibliographic entries, looked for the data fields with the appropriate tag number as specified in the documentation for the data we were looking for, and copy that data directly into our return object (in some cases, we performed additional reformatting to avoid potential format problems with our database).

To test these parsing functions, we wrote a number of functions that we could use in an interactive PHP instance which would output their results directly to the console. This way we could see the results the parsing functions would give as we were creating them and immediately resolve any issues. Once we were sure the parsing function itself was running effectively, we added some feedback into the scanning functions that read through the whole file so that we could see how far the function had read into the file and see where in the datasets issues were occurring..

Due to memory limitations, each of these parsing functions operate on the file stream. The parsing functions read a line from our source data and immediately write it to the output files. All these filtering functions run locally and output their results to a CSV with a | delimiter. We are currently in the process of implementing PHP code to copy these datafiles into our database. This code pulls all the data on each entry from the CSV files and writes it to the database using SQL. When writing the genre associations for each entry, the code will utilize a total genre occurrence file written by the parsing script to create decimal numbers needed for the recommendation algorithm. These decimal numbers represent the TFIDF score, calculated from the number of times the genre occurs in the entry (which in every case will be 1) multiplied by the base-10 log of the inverse ratio of term frequency in the entire dataset to number of entries in the dataset. While this system won’t be perfect because there’s no indication of how strongly a particular genre or keyword is associated with a document, only that it is associated at all, we believe it will be a good prototype algorithm that will allow us to complete development of this prototype application.