AIT – MSC IN SOFTWARE ENGNEERING

A Mobile Recommender System for Movies

Accessing an R service from an Android Client

Stephen Jacob

4/11/2016

Supervisor: Dr. Abir Awad

Abstract

The purpose of this project is to develop a recommender Android application for movies. The server will be developed in R and deployed to OpenCPU using GitHub and Webhooks (a continuous integration mechanism). It exposes a REST interface for recommendations. A predict method expects a list of movies along with the clients' recommendations. It then makes a prediction about other movies that the client would like based on a collaborative filtering algorithm. The Android client component allows the user to find and rate movies that the user has seen. This information is then sent to the server which returns a list of recommendations to the client.

Contents

1. Introduction

1.1 Main Objective

The main objective of this project is to build a mobile Android application that accesses a machine learning service. The machine learning service should provide a public interface to its clients. The Android client should use this public interface to access the data mining or machine learning service.

The rationale for this is that a mobile client can make use of machine learning prediction services without having to execute the machine learning algorithms on the client side. Also the data and the models can be stored on the server side and don't have to be downloaded to each client.

1.2 Recommendation Application

For this project I chose to implement a recommendation service for movies. The idea is for the user to rate movies using the Android application. The application then sends the rated movies to the service. The server then finds a number of movies to recommend to the client based on the movies rated by the client.

Recommendations are found using a Collaborative Filtering algorithm. The main types of Collaborative Filtering algorithms are User-Based Collaborative Filtering and Item Based Collaborative Filtering. This project uses a User-Based Collaborative Filtering algorithm. The algorithm finds similar users based on rating data, that is for example finds users who tend to like the same films. It then can recommend movies that are rated highly by similar users.

To implement the server side, I chose the programming language R as this is widely used in data science applications. In addition, there are libraries for lots of machine learning algorithms and in particular there is a library called "recommenderlab" that implements the User-Based Collaborative Filtering algorithm that I choose to use.

Having considered a number of options for the deployment platform, I choose OpenCPU to host my application server. OpenCPU provides support for a HTTP based API to hosted data services that are written in R. Using OpenCPU one can host one's own servers but in this project I have used an OpenCPU public service.

For the recommendation algorithm to work training data is required. For this project the training data consists of movies and their ratings by a set of users. The dataset used was obtained from the

MovieLens project. This is made freely available for download and was used as training data for the recommendation system. The data consists of 8552 movies rated by 706 users with 100052 ratings.

The client component is implemented as an Android application. Data about the 8552 movies is stored in a SQLiteDatabase. The client can present the user with lists of movies and allows the user to rate movies. To obtain recommendations the client sends the movie ratings to the server and receives back the recommended movies which are then presented to the user.

1.3 Remainder of Thesis

The research carried out is outlined in Section 2. Section 2.1 outlines the research in the application domain, i.e recommender systems. Section 2.2 outlines the technologies required to implement the system and examines alternatives. For the server side implementation I examined the R programming language and various R libraries, in particular jsonlite and recommenderlab. This section also examines the R SparseMatrix class as this is used extensively in the recommenderlab library. In order to deploy the service to OpenCPU it is necessary to implement it as an R package. This section also examines R packages and how they are created.

I then examined various deployment options including yHat, Shiny, Azure and finally OpenCPU. Again in order to deploy to OpenCPU it is necessary to push the project to open-source source code repository GitHub. It is also necessary to define webhooks which deploy the project to OpenCPU automatically on an update to GitHub. GitHub and webhooks are examined in this section also.

Finally in the technology section, I examined cURL and PostMan. CURL is a command line utility for issuing HTTP requests and Postman is a Chrome plugin to do the same.

Section 3 gives an overview of the prototype implementation. It outlines requirements for the application, the system architecture and system design both server side and client side. The implementation subsection outlines how the application works and provides some screenshots of the application running. The testing subsection outlines testing carried out during the project.

Finally, section 4 is the conclusion.

2. Research

2.1 Application Domain

2.1.1 Recommender Systems

A recommender system [10] is an information filtering application that predicts the 'rating' or 'preference' a user would give a particular entity. Recommendation algorithms are common present day topics for researchers in the fields of machine learning and data mining. These applications have become immensely popular in recent years, especially regarding items such as movies, books, music and social media in general. The algorithm serves to evaluate patterns of behaviour to know what a user would prefer from collection of activities he's never come across. In doing so, recommender systems have come to change the way people find out information, how they find and try new things and how they meet other people.

2.1.1.1 Recommender Systems in E-Commerce

Recommender systems are also widely utilized for web commerce applications [11]. Through the use of recommendation algorithms, commercial companies have gradually changed their approach from simply mass producing standardized marketing products to specializing in a variety of customized products that meet the multiple needs of the greater general population. There are three ways in which recommender applications enhance e-commerce sales.

- Web sites: browsers who look at a commerce website often find what they are looking for on a site with the help of a recommender system
- Cross-sell: a recommender system helps to improve commercialism by suggesting similar products for the customer to buy
- Customer loyalty: the more times the user visits the site, the more likely the application proves to create a relationship between the customer and the site investing in learning about the users wants, and then presenting customer interfaces that represent the customer's wants,

2.1.1.1.1 Amazon

Amazon.com, an online e-commerce site of music, books, movies, is one of many e-commerce sites that use a recommendation algorithm. Each available item for purchase is presented by an information page, giving data of the item of product's purchase information.

In the case of Amazon, there are two separate generated recommendation lists. The first is presented to users displaying products similar to that purchased by the user. The other list presents the tag lines, authors, makers, social media, etc of the product that matches those of similar products frequently purchased by customers.

The Customer Comments section in Amazon also allows a customer to write feedback on the particular products so to present their own recommendations.

2.1.1.1.2. Ebay

Ebay is another e-commerce retailer site that utilizes a recommender system. This is carried out on ebay via the FeedBack Profile feature.

The FeedBack Profile feature of ebay.com is available for use by both registered buyers and sellers. Those who purchase from ebay.com can view the recommendation profiles of sellers. Both parties can contribute to the feedback of customers with whom they have done business. It consists of rating with three values (satisfied/neutral/dissatisfied). This feedback-based application provides a recommendation system.

2.1.1.1.3. IMDB

The Internet Movie Database is a television-based database that uses a personalized recommender system that helps users find movies or TV shows they'd like.

The recommender system on IMDB is applied as follows:

- they take all the movies and TV shows a user has rated or added to his or her watch-list
- compare their data to ratings made by other users
- find the movies/shows that people who have similar interests to the user like or have rated
- for each recommendation, you can then see a list of movies/shows on which it was based

2.1.2. Collaborative filtering

A collaborative filtering algorithm is an approach to the design of a recommender system. The basis of collaborative filtering is an algorithm that filters information involving collaboration amongst multiple different point-of-views, collaborators and sources.

Applications that use a collaborative filtering algorithm will typically have immensely large data sets. Overwhelming amounts of data make an information filtering method necessary so a user can extract the information needed. The motivation of collaborative filtering originates from the notion that an individual gets the best recommendations from others who have interests similar to his.

Collaborative filtering algorithms will typically come in two different forms: user-based collaborative filtering and item-based collaborative filtering.

2.1.2.1. User-Based Collaborative Filtering

User based collaborative filtering is based on the notion that an algorithm recommends a product to a user based on the preferences of other similar users, that is, users who have similar interests in products. UBCF is carried out as follows:

- a user declares his preferences, i.e. rates a product
- look for users who have similar interests, rate similar products
- use the ratings from said similarly minded users to make predictions for the active user

2.1.2.2. Item-Based Collaborative Filtering

Item based collaborative filtering makes its predictions, by contrast, in an item-centric manner. The predictions calculated are based on the similarity between different items using people's ratings of those items. The process is carried out as follows:

- the algorithm finds the likeliness between two items
- it then recommends the most similar items to a user's already-prepared list of rated items

2.1.3. Data Sets

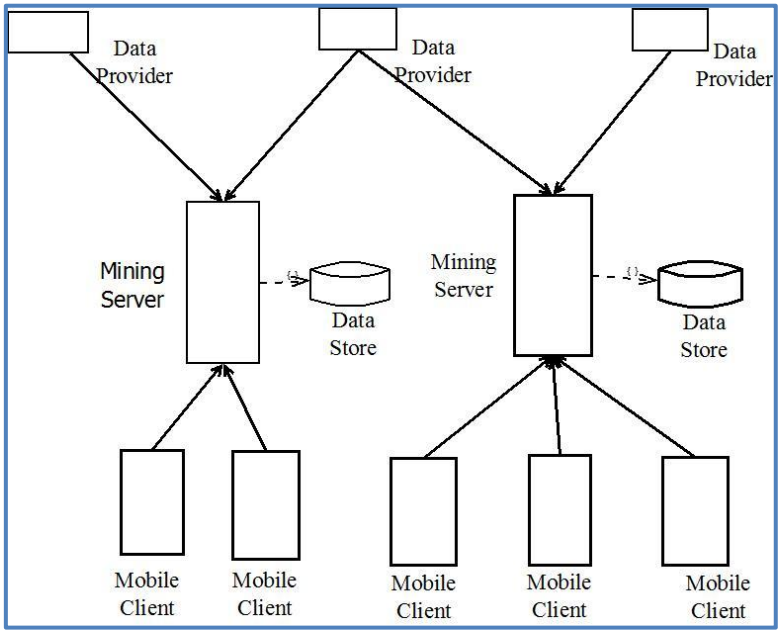
Regarding data, the MovieLens website makes public a large dataset of historical data regarding user's ratings of movies. This is in normalized relational data form. This seems suitable for the project. But the data has to be manipulated in R in order to be suitable for use with recommenderlab.

2.1.4. Mobile Data-Mining Services

Mobile data-mining is referred to as the process of using handheld devices to run machine learning applications. The data for these apps can be remotely accessed from the site where the users work. A client will need to perform an analysis of their machine learning algorithm of the data obtained from the repositories located far away from the site where the users work.

A benefit of running client programs on a mobile device is that users can run their algorithm and obtain the data mining results from the application regardless of their current physical location. The data obtained for the app can be accessed remotely. Therefore users can execute data-mining tasks and display their results to people from wherever they are.

A system used in a mobile data-mining application will be modelled after the client-server architecture. This architecture requires three different components: data providers, mobile clients and mining servers. This architecture is displayed in the image below, taken from [12] :



2.2 Technology

2.2.1 Implementation Languages

An implementation language is a software programming language that's used for executing computer programs. The two programming languages that seem most useable for this project are R and Python.

Python is a widely used general-purpose dynamic programming language. It supports multiple programming paradigms, has a large comprehensive library as well as a dynamic type system. Likewise, R is widely used, but includes other libraries, paradigms and tools than Python. R can be used to execute programs that are object-oriented, applications that contain data in the form of attributes. Therefore, the programming language R appears most suited for the application's purposes.

2.2.1.1 R Programming

R is a software programming language, but also functions as a development environment used for statistical computing. It is commonly used in the process of developing data mining software. The source code of the R software environment is written mainly in the programming languages C, Fortran and R itself.

The language R is highly popular as it uses many different paradigms in coding. It has support for the multi-paradigm programming language which provides support to implement multiple programming paradigms in a programming language. Other paradigms that can be used in R include array, object-oriented, imperative, functional, procedural and reflective. R and its libraries also implement a wide variety of tools including statistical and graphical techniques. While R has support for graphical front ends it also has a command line interface.

R has multiple data structures besides arrays, including vectors, matrices, data frames and lists. These data structures are utilized in procedural programming and object-oriented programming including generic functions.

2.2.2 jsonlite Package

This package, jsonlite[8], converts R objects into JSON objects or the other way around. JSON arrays are mapped to lists in R. There are further examples in the implementation section.

2.2.3 R SparseMatrix

A sparse matrix is used to store the rating matrix for recommendations. A sparse matrix is a matrix where most elements are zero. If a sparse matrix is stored in a 2D array then there will be a lot of blank spaces and memory will be wasted.

In R, the class sparseMatrix is used to store sparse matrices. The constructor is:

```
sparseMatrix(i = ep, j = ep, p, x, dims, dimnames, symmetric = FALSE, index1 = TRUE,
                                 giveCsparse = TRUE, check = TRUE, use.last.ij = FALSE)
```

i - a vector specifying the row indices of the non-zero elements

j - a vector specifying the column indices of the non-zero elements

x - a vector of the values

dim - the dimensions of the matrix

For example, suppose the following are the no-zero elements:

(1, 35) 4

(3, 100) 2

(5, 234) 3

(1, 766) 1

(2, 1000) 5

Then i= (1, 3, 5, 1, 2), j = (35, 100, 234, 766, 1000) and x = (4, 2, 3, 1, 5).

sparseMatrix <- sparseMatrix(i, j, x = x, dims=c(600,8552))

This creates a 600 X 8552 sparseMatrix object with 5 values set.

2.2.4. R recommenderlab Package

I have started looking at collaborative filtering; both user and content based collaborative filtering. I have also been looking at the R library recommenderlab which provides basic algorithms (and also the means of developing and using one's own algorithm in the framework if necessary.)

The R package recommenderlab is a library that provides an environment to develop a recommendation algorithm.

2.2.4.1. predict

The primary method I will use from recommenderlab is predict. The preferred format of the method that I will use for my algorithm is displayed below:

```
predict(model, inputData, type)
```

```
model = the recommender model object
```

```
inputData = data for active users, default type is "ratingMatrix"
```

```
type = type of recommendation the user wants
```

2.2.5. R reshape Package

The R library reshape allows a programmer to flexibly restructure and aggregate data.

To convert a csv file into a matrix, run the following lines of code:

```
library(reshape2)
matrix <- acast(trainingData, userId ~ movieId)
class(matrix)
```

2.2.5.1. RealRatingMatrix

The main classes for our purposes are realRatingMatrix and recommender.

The realRatingMatrix contains ratings, typically 1 to 5. The ratings are stored in a dgCMatrix which a sparseMatrix containing numeric values. The rows of the matrix correspond to userIds and the columns (in our case) correspond to the movieIds. The values correspond to the ratings. For example in the first non-zero element above the user with userId 1 rates a movieId 35 with a 4.

To convert an ordinary matrix to a realRatingMatrix, do the following:

```
realRatingMatrix <- as(matrix, "realRatingMatrix")
```

2.2.6. Creating R Packages

R packages are used for publishing R code, for example CRAN (Comprehensive R Archive Network). The packages can then be used by using install.packages() and library() in an R script.

R packages are important for this project because we need to deploy to OpenCPU and the only thing that can be deployed to openCPU is an R package.

Using RStudio, there is an option to create an R package. This creates the appropriate folder structure for the project. An R package needs:

- An R folder that contains all the R scripts
- A man folder that contains the R documentation (Rd) files.
- A DESCRIPTION file
- A NAMESPACE file

A tool called roxygen is very useful because it generates documentation files and the NAMESPACE file.

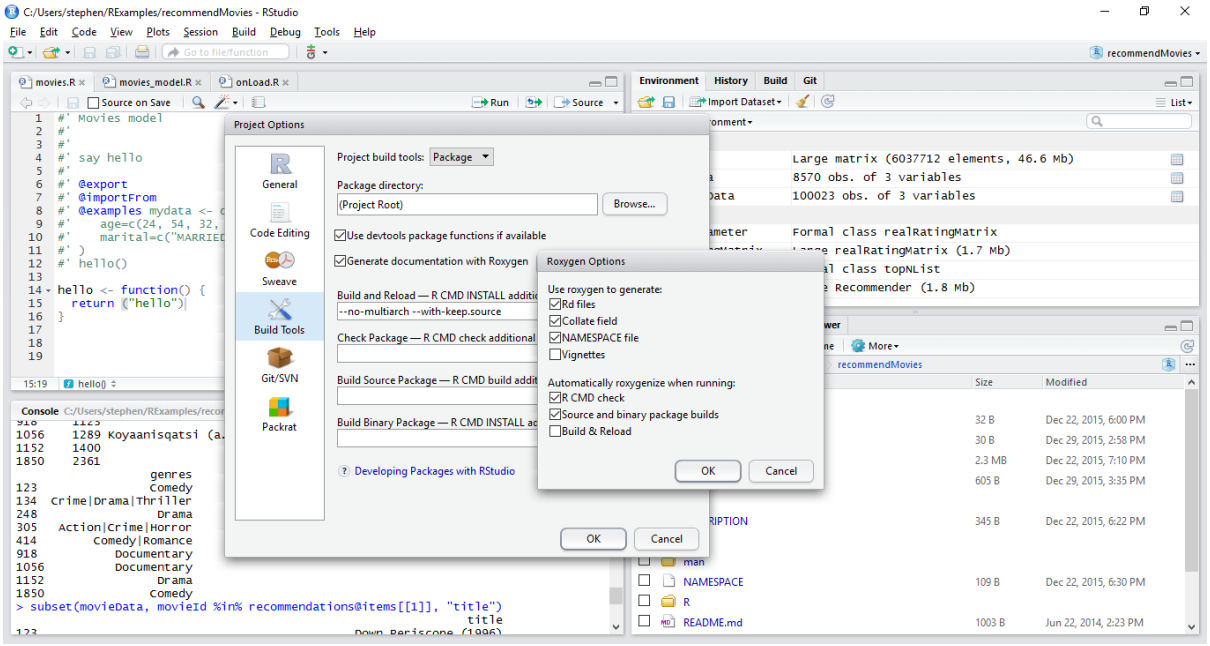
2.2.7. roxygen2

roxygen2 is a tool use for generating R documentation for an R package. It is similar to javadoc in Java. Documentation is generated from special comments starting with # '.

Example comments to go here:

You can configure RStudio to run roxygen2 whenever a package is built.

Tools – Project Options – Build Tools



For every R file in the package an .Rd file is generated in the man folder. Also the NAMESPACE file for the package is generated.

2.3. R Packages - Coursera Developing Data Products

2.3.1. DESCRIPTION file

```
Package: recommendMovies
Type: Package
Title: Recommend movies
Version: 0.1
Date: 2015-12-22
Author: Stephen Jacob
Maintainer: Stephen Jacob <stephen.a.jacob@gmail.com>
Description: OpenCPU app to recommend movies
License: Apache License 2.0
LazyData: false
Depends:
    R (>= 2.10)
Imports:
    recommenderlab
RoxygenNote: 5.0.0
```

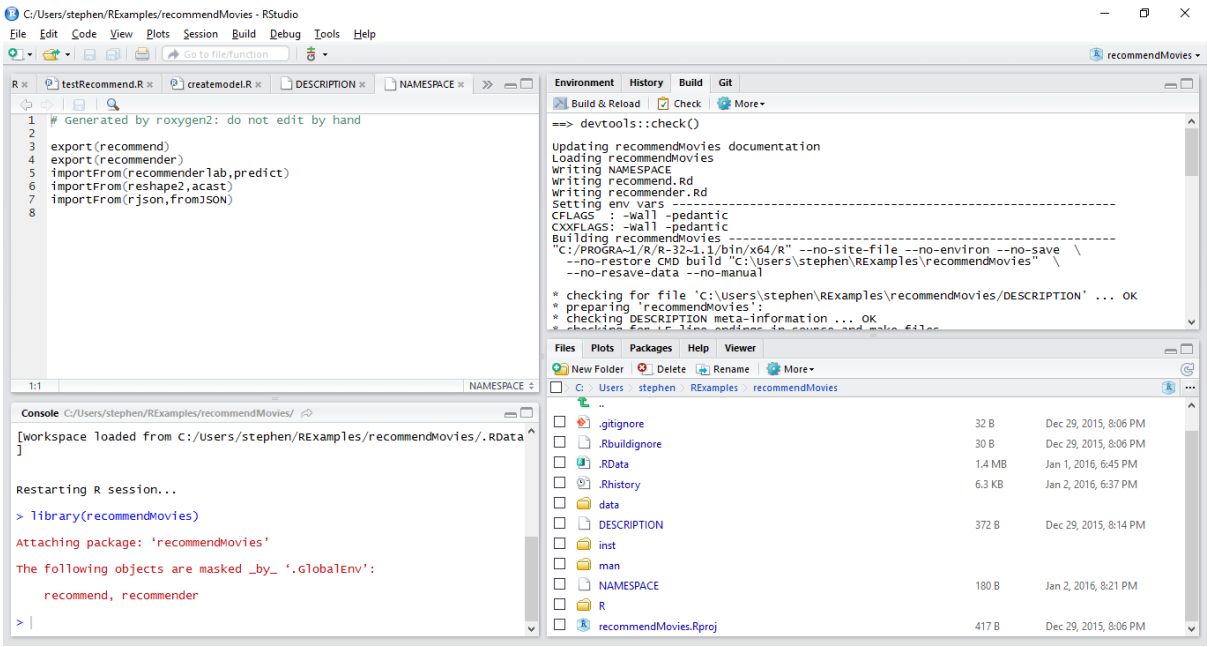
2.3.2 NAMESPACE file

This is a file generated by a utility called roxygen2.

```
export(getRecommendation)
export(recommender)
importFrom(Matrix,sparseMatrix)
importFrom(recommenderlab,predict)
importFrom(reshape2,acast)
```

2.3.3. Checking R packages

RStudio provides a check option under the build panel which checks the R package to see if it has been constructed correctly (See top right window below).



2.3.4. Including a Model Object in an R Package

The term model is used here in the sense of a data mining model.

In general when an R package is used to make a prediction, it is necessary to take the training data and build a model object. In this project the model object is of type Recommender which has been created from the training data which is a rating matrix. The algorithm used is a user-based collaborative filtering algorithm.

The model object can saved in a RDA file (R Data Access file). To specify that the object is loaded when the R package is loaded, so it's necessary to write an .onLoad () function and place it in a file called onLoad.R in the R folder for the package.

In createmodel.R, the `recommender` model will be created and will be of type user-based collaborative filtering, the loaded data used will be stored in a realRatingMatrix

```
recommender = Recommender(
  realRatingMatrix[1:nrow(realRatingMatrix)],method = "UBCF", param = list(
    normalize = "Z-score",method = "Cosine",nn = 5, minRating = 1
  )
)
# save the model
save(recommender, file="data/recommender.rda")
```

In onLoad.R,

```
.onLoad <- function(lib, pkg){
  #automatically loads the dataset when package is loaded
  #do not use this in combination with lazydata=true
  utils::data(recommender, package = pkg, envir = parent.env(environment()))
}
```

2.4 Deployment Options

2.4.1. Yhat

I have had a quick look at a number of deployment options. Initially I planned to use the hosting service *yhat*. This used to provide a free hosting platform for R applications but the company now seems to be more commercialised and I don't know if this approach is still possible. I will still ask them to see if it would still be possible for me to have free hosting for a student project.

2.4.2. Shiny

Another option is to build a *ShinyApp* which is a web application that RStudio will host for free.

2.4.3. Azure

Another is the implementation of Microsoft Azure, a cloud computing service to deploy my application.

2.4.4. OpenCPU

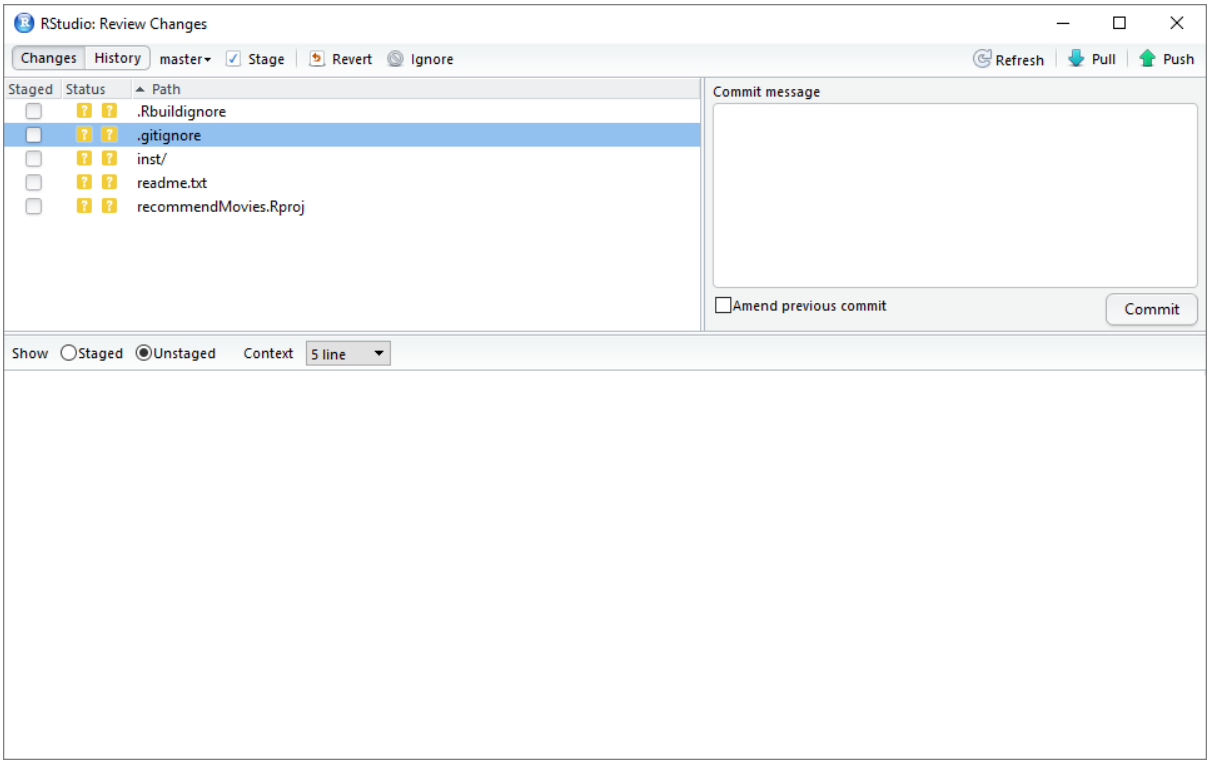
Using the software system OpenCPU is yet another option for deploying the app. The API is used to embed and develop as well as deploy scientific and research computing. This option appears the most suitable for deployment. This is covered in more detail below.

2.5. GitHub

GitHub is an open-source Web repository hosting service. It allows users to create their own repositories to store, develop and share their own software applications, browse and contribute to many different other projects. GitHub is highly recommended as it facilitates version control management.

- In RStudio, there is built in support for Git. To push to a remote GitHub repository
- Push to a local Git repository from within RStudio
- In GitHub, create the repository
- Command line (in the .git folder)
- git remote add origin https://github.com/stephenajacob/recommendMovies.git
- git push -u origin maste

Now RStudio provides an option to push to the remote repository.



2.5.1. Webhooks

A webhook is a tool that builds or sets up integrations that allow one to subscribe to an event within the repository. When this said event is triggered, the result will send a HTTP POST payload to the webhook's configured URL.

Settings – Webhooks and services – Add webhook

Payload URL: `https://public.opencpu.org/ocpu/webhook`

Content type: `application/x-www-form-urlencoded`

Which event would you like to trigger this webhook: `The push button event.`

Now whenever a push of a button of the application is made, the webhook will be triggered and the application will be deployed to the OpenCPU server.

2.14 OpenCPU

This application will be implemented with OpenCPU acting as a host server. As mentioned above the application's server side must be implemented as an R package. It is possible to use public servers or host your own. The OpenCPU is supported on operating systems Ubuntu and Fedora, and requires the Apache web server.

This project will use the public servers. The previous section has already described how it is possible to deploy from GitHub using a webhook. Once an R function has been deployed OpenCPU supports HTTP/REST access to this function.

For example, from the OpenCPU documentation:

```
curl http://public.opencpu.org/ocpu/library/stats/R/rnorm/json \
-H "Content-Type: application/json" -d '{"n":3, "mean": 10, "sd":10}'
```

The above request invokes the following R function call:

```
library(jsonlite)
args <- fromJSON('{"n":3, "mean": 10, "sd":10}')
output <- do.call(stats::rnorm, args)
toJSON(output)
```

For specifically this project, the following command is invoked:

```
curl
https://public.opencpu.org/ocpu/github/stephenajacob/recommendMovies/R/getRecommend
ation/json -H "Content-Type: application/json" -d "{\"movies\": [34, 87, 236, 312,
397, 650], \"ratings\": [5, 2, 3, 4, 1, 2]}"
```

The backslashes are necessary inside the JSON string because this was called on Windows. Information on the command "curl" is outlined below.

This results in the following R function call:

```
args <- fromJSON(jsonString)
str(args)
output <- do.call(getRecommendation, args)
toJSON(output)
```

The getRecommendation() function is

```
getRecommendation <- function(movies, ratings) {
       i <- rep.int(1, length(movies))

       sparseMatrix <- sparseMatrix(i, movies, x = ratings, dims=c(1,8552))

       inputParameter <- new("realRatingMatrix", data = sparseMatrix)

       #str(inputParameter)

       recommendations <-
              predict(recommender, inputParameter, type = "topNList")

  return(recommendations@items)}
```

cURL

cURL is a software library and a command-line tool used for transferring data from or to a server, It is capable of transferring between servers of several various protocols including HTTPS.

Android

Android is an operating system that's primarily used for designing hand-held mobile devices such as smart phones or tablets.

Prototype Implementation

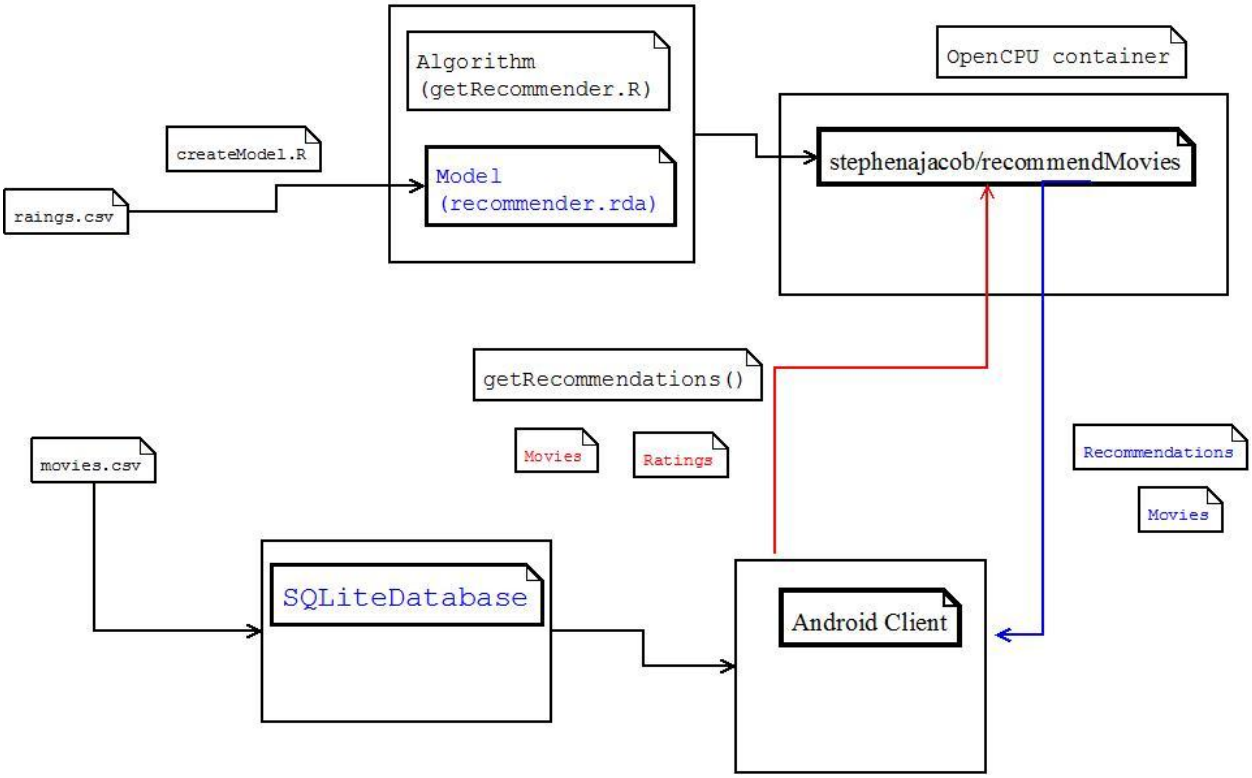
Requirements

The application will allow a user to:

- rate a movie stored in the provided database
- select one's favourite movie to display movie data
- search for movies by both titles and genres
- clear movie ratings of a previous user
- return movie recommendations to another user based on their input data

When returning the list of recommended movies, the algorithm that the application will use to return the data will be based off User-Based collaborative filtering (UBCF) [9].

System Architecture



At the beginning of the application we have two files *ratings.*csv and *movies.*csv obtained from the website MovieLens [3]. The format of these is as follows:

movies.csv

movieId	title	genres
1	Toy	Adventure|Animation|Children|Comedy|Fantasy
2	Jumanji (1995)	Adventure|Children|Fantasy
3	Grumpier Old Men (1995)	Comedy|Romance
4	Waiting to Exhale (1995)	Comedy|Drama|Romance

rating.csv

userId	movieId	rating	timestamp
1	6	2	980730861
1	22	3	980731380
1	32	2	980731926
1	50	5	980732037
1	110	4	980730408

Note that the movieId corresponds to a foreign key from ratings to movies and that ratings can be between 1 and 5 inclusive.

Before the application starts, it is necessary to create a model that will be deployed to the server and will be used by the recommendation algorithm. The *createModel.R* script creates a recommender object that uses the movie ratings as training data. This model is then saved as recommender.rda in the R project directory.

In the R project, there is also a file named *getRecommender.R* which denotes the actual algorithm that will return the list of recommended movies. Both the algorithm and the model are deployed to the OpenCPU container.

In the Android client, the data from *movies.csv* is loaded into a table called MOVIES in an SQLiteDatabase. This ensures that the same movieIds from *movies.csv* will be used on the client side and on the server side.

The format of the MOVIES table is as follows:

```
CREATE TABLE MOVIES (ID INTEGER PRIMARY KEY,
```

```
MOVIE_ID                                                INTEGER,
        MOVIE_TITLE TEXT,

  MOVIE_GENRES TEXT,

  MOVIE_RATING INTEGER)
```

The MOVIE_RATING column is used to store the rating for a particular movie.

It is possible for the Android application to search for a particular movie, search for movies by genre, etc. The movies are displayed with a RatingBar which allows the user to rate a movie between 1 and 5.

When the user wants to obtain movie recommendations the client application does the following:

it obtains a list of all movies from the database that have been rated by the user

it creates a JSON string which contains those movieIds and their respective ratings, for example, {"movies": [34, 87, 236, 312, 397, 650], "ratings": [5, 2, 3, 4, 1, 2]}

it makes a HTTP POST request to the server application passing the above JSON string

gets back a JSON string of recommended movies for example, [[628, 655, 238, 977, 49, 702, 661, 79, 968, 630]]

obtains the titles corresponding to these movieIds and presents the recommendations on a list to the user.

Design – Server Side

Firstly, what needs to be done is to build a recommender model object (an R object) from the training data. This is done as follows in the file createModel.R:

- Load the libraries recommenderlab and reshape2
- Read in the training data which is a collection of 100023 ratings of movies (movieId) by users (userId)
- Convert table data into a matrix (rows are users, columns are movies)
- Convert matrix into a realRatingMatrix (this is an object in the recommenderlab package)
- Create a Recommender object from the realRatingMatrix
- Save the model to a file called “recommender.rda”.

The following is the R code to implement this.

```
library(recommenderlab)
library(reshape2)

trainingData <- read.csv("inst/ratings.csv",header = TRUE)

# Remove 'timestamp' column. We do not need it
trainingData <- trainingData[,-c(4)]


matrix <- acast(trainingData, userId ~ movieId)
class(matrix)

# Convert matrix into realRatingMatrix data structure
#   realRatingMatrix is a recommenderlab sparse-matrix like data-structure

realRatingMatrix <- as(matrix, "realRatingMatrix")
realRatingMatrix

recommender = Recommender(
  realRatingMatrix[1:nrow(realRatingMatrix)],method = "UBCF", param = list(
    normalize = "Z-score",method = "Cosine",nn = 5, minRating = 1
  )
)

#Save the model
save(recommender, file="data/recommender.rda")
```

The realRatingMatrix is a 706 x 8552 matrix with 100023 ratings. There are 706 users and 8552 movies.

Note that the size of the original matrix is 23.5 Mb. The realRatingMatrix is an efficient storage of this data and turns out to be 1.7 Mb. The final Recommender object is of size 1.8 Mb.

The calculation of the Recommender object is done using a User Based Collaborative Filtering algorithm. This is specified in the method parameter. The data is first normalized so that all user ratings have the same mean and standard deviation.

The next task for the R server application is to develop an algorithm to output a list of recommended movie components. This file is deployed to the server. This file, named getRecommendation.R, develops the algorithm as follows:

- inputs the movies and respective ratings
- creates a sparseMatrix containing the values of ratings with the dimensions of 1 to the maximum number of movies
- declare the sparsematrix as a realRatingMatrix
- takes the existing recommender model to create recommendations
- the resulting recommendations list should be of type “TopNList”
- returns the recommended movies

The following is the R code needed for the above:

```
getRecommendation <- function(movies, ratings) {

  i <- rep.int(1, length(movies))

  sparseMatrix <- sparseMatrix(i, movies, x = ratings, dims=c(1,8552))

  inputParameter <- new("realRatingMatrix", data = sparseMatrix)

  #str(inputParameter)

  recommendations <-
    predict(recommender, inputParameter, type = "topNList")

  return(recommendations@items)
}
```

Note:

The variable *i* is used to specify rows indices for the sparseMatrix, just as the variable movies specifies the number of columns.

The type specified as “topNList” which results in a top-N list with recommendations.

The last R file the R project will call is the *testRecommend2.R* that will make the request to the database for the recommended movie list. This is what the file does:

- Load the libraries recommenderlab, reshape2 and jsonlite
- Load the recommender.rda into the project environment
- Call the getRecommendation.R file
- post the data as JSON to the server
- return the output code to JSON format

The code to do this is outlined below:

```
library(recommenderlab)
library(reshape2)
library(jsonlite)

source("R/getRecommendation.R")
load(file = "data/recommender.rda")


jsonString <- '{"movies": [34, 87, 236, 312, 397, 650], "ratings": [5, 2, 3, 4, 1,
2]}'

# inside opencpu
args <- fromJSON(jsonString)
str(args)
output <- do.call(getRecommendation, args)
toJSON(output)
```

Design – Client Side

On the client side, the first thing to do is load a database of movies into the Android application. The database will be a SQLiteDatabase which exposes methods to execute an SQL database.

In the client component, this is done in the following order:

- Build a Java class named DatabaseHandler to execute queries in an SQL database
- Start an Android Activity
- Read in the training data which is a collection of movies from the android project directory

The following is the code written in the Android application to read in the movie data into an instance of the DatabaseHandler as lain out below.

```
DatabaseHandler db = new DatabaseHandler(this);
try {
    InputStream input = getAssets().open("movies.csv");
    BufferedReader reader = new BufferedReader(new InputStreamReader(input));
    String inputLine = reader.readLine();

    while ((inputLine = reader.readLine()) != null) {
        System.out.println("Input: " + inputLine);
        String[] array = inputLine.split(",");
        int movieId = Integer.parseInt(array[0]);
        String title = array[1];
        String genres = array[2];
        Movie newMovie = new Movie(movieId, title, genres, 0);
        db.addMovie(newMovie);
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

The DatabaseHandler class will be used to define methods to execute SQL commands on the database used in the application. The class will have methods to:

- Add a movie to the database
    - `addMovie(Movie movie)`
- Returns a movie from the database
    - `getMovie(String movieTitle)`
- Return a list of movies
    - `getAllMovies()`
- Return all movies that are rated
    - `getRatedMovies()`
- Search for a movie based on input movie_id, titles or genres
    - `getMoviesByGenre(String genre)`
    - `getMoviesTitle(String movieText)`
    - `getMoviesById(List<Integer> movieIds)`
- Set a rating for a single movie
    - `setMovieRating(Movie movie)`
- Clear the rated movies from the app
    - `clearRatings()`

The other major task to implement for the Android client side is to declare a Java class

The resulting recommended list of movies is displayed in the implementation below. This requires a number of conditions to work:

- An Android Activity, RecommendMovie retrieves all movies the user has rated
- Convert the rated movies into a string of JSON data
- Declare a class MyHttpConnection, that makes Http url connections, POST requests, and sends and gets JSON data to and from the server
- The server then returns the list recommended movies from the loaded dataase
- A method onPostExecute is declared in the Android recommending activity that takes a list of Integers and adds all the recommended data from the response to the list of Integers.

```
private String getRatingsAsJSONString(){
    List<Movie> movies = db.getRatedMovies();

    List<Movie> newMovies = new ArrayList<Movie>();
    for(int i = 0; i < movies.size()-1; i++){
        newMovies.add(movies.get(i));
    }

    String moviesJSONData = "{\"movies\": [";
    for(Movie mID: newMovies){
        moviesJSONData += mID.getMovieId() + ", ";
    }
    moviesJSONData += movies.get(movies.size()-1).getMovieId();
    moviesJSONData += "], \"ratings\": [";
    for(Movie mRating: newMovies) {
        moviesJSONData += mRating.getRating() + ", ";
    }
    moviesJSONData += movies.get(movies.size()-1).getRating();
    moviesJSONData += "]}";

    System.out.println("JSON Request: " + moviesJSONData);
    return moviesJSONData;
}
```

```
protected void onCreate(Bundle savedInstanceState) {

.

.

.

String url = "https://public.opencpu.org/ocpu/github/stephenajacob/" +
        "recommendMovies/R/getRecommendation/json" ;
String jsonString = getRatingsAsJSONString();
PostTask task = new PostTask(this);
task.execute(url, jsonString);

}
```

```
public PostTask(AsyncResponse delegate){
    this.delegate = delegate;
}

@Override
protected String doInBackground(String... params) {
        String data = "";
    try {
        MyHttpConnection myHttpConnection = new MyHttpConnection();
        data = myHttpConnection.postToUrl(params[0], params[1]);
    } catch (Exception e) {
        Log.d("Background Task", e.toString());
    }
    System.out.println("data: " + data);
    return data;
}
```

```
@Override
protected void onPostExecute(String result) {
    super.onPostExecute(result);
    System.out.println("Result onPostExecute" + result);
    int start = result.lastIndexOf("[") + 1 ;
    int end = result.indexOf("]") - 1 ;
    String s = result.substring(start, end);
    System.out.println(s);

    String[] sa = s.split(",");
    List<Integer> movieIds = new ArrayList<Integer>(sa.length);

    for(int i = 0; i < sa.length ; i++){
        movieIds.add(Integer.parseInt(sa[i].trim()));
        System.out.println(movieIds.get(i));
    }
    List<Movie> recommendedMovies = db.getMoviesById(movieIds);
    delegate.processFinish(recommendedMovies);
}
```

The following code outlines the class, MyHttpConnection that connects to the web server urls, allows the content type to be of type JSON, sends POST requests and disconnects from the URL.

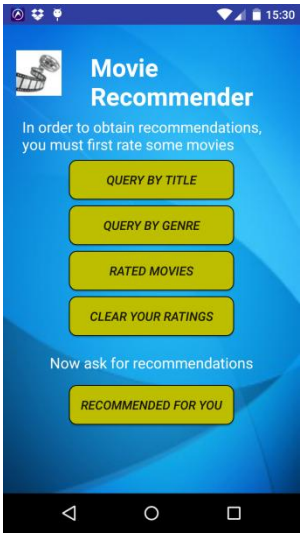
```
public class MyHttpConnection {

public String postToUrl(String jsonURL, String jsonPostData) throws IOException {
    String data = "";
    InputStream iStream = null;
    HttpsURLConnection urlConnection = null;
    try {
        System.out.println("INSIDE TRY");
        URL url = new URL(jsonURL);
        urlConnection = (HttpsURLConnection) url.openConnection();
```

```
        urlConnection.setRequestProperty("Content-Type", "application/json");
        urlConnection.setRequestProperty("Accept", "application/json");
        urlConnection.setDoOutput(true);
        urlConnection.connect();
        System.out.println("AFTER CONNECT");

        // Send post request
        DataOutputStream wr = new
DataOutputStream(urlConnection.getOutputStream());
        wr.writeBytes(jsonPostData);
        wr.flush();
        wr.close();

        System.out.println("AFTER CLOSE");
        iStream = urlConnection.getInputStream();
        BufferedReader br = new BufferedReader(new InputStreamReader(
                iStream));
        StringBuffer sb = new StringBuffer();
        String line = "";
        while ((line = br.readLine()) != null) {
            sb.append(line);
        }
        data = sb.toString();
        System.out.println("DATA: " + data);
                br.close();
    } catch (Exception e) {
        System.out.println("Exception" + e.toString());
    } finally {
        iStream.close();
        urlConnection.disconnect();
    }
    return data;
}

}
```

Implementation

My application for recommending movies will be deployed to a portable Android device, such as a mobile phone.

Once a user opens the app, the following user interface (UI) is displayed:

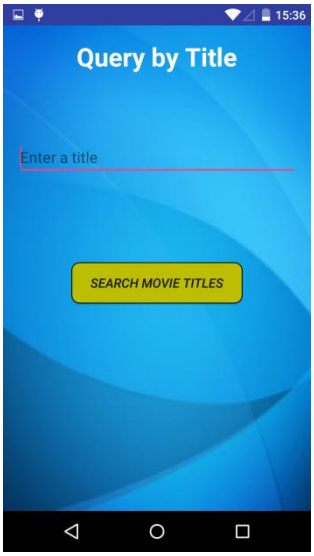


On the UI above there are a number of buttons that allow the user to perform a number of actions. The first two buttons allow for the following two functions:

- to search for movies by the their titles
- or by their genres

To search for a movie based on their title, the user clicks on the button “Query By Title”.

The UI to enter the movie title is displayed below:



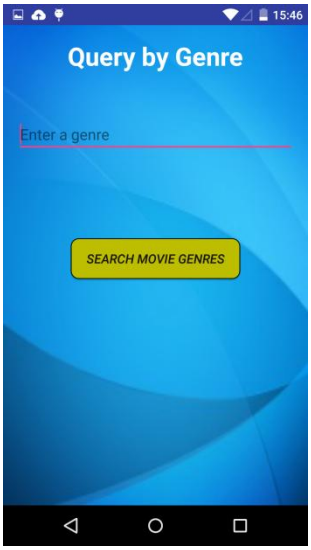
The user will then enter their input text in the box labelled ‘Enter a title’, e.g. crime. The user then presses the button labelled ‘Search Movie Titles’.

The resulting UI is displayed is a list of all discovered movies whose titles contain the text ‘Crime’.

The scrollable list above displays the components with a title for the movie as well a RatingBar widget that will allow the user to rate the respective movie between one to five stars. They can also edit the rating afterwards. This rating for a particular movie will then be saved to the database. Beneath the list is an option to return to the main page.

Note: every resulting list in the app will have the very same format as the one above.

To enter to search for movies by a single particular genre, back on the opening UI, the button labeled ”Query By Genre” is clicked and the UI below is displayed.



If a user wanted to search for movies that contain action as one of its genres, they would type in ‘action’ in the textbox to search for all movies with the genre.

The result is a list of action movies which is displayed on the following UI:

Every list allows the user to rate movies. Each user can see their ratings by clicking the button labelled “Rated Movies” on the main UI. The following UI is the result, and the one afterward is the result of the user not having rated any movies.

The user can also use the app to return them a list of movies recommended to the user based on movies rated by the user.

For example because the earlier examples of the querying the database for movies looked for movies orientated with ‘crime’ and ’action’, this will return a list of movies that have action as one of their genres.

The UI for this recommended list is displayed below:

Another option on the main page for the user is to clear the database of all his rated movies.

Testing

Test file testRecommend2 as an R program

To test this file, it only must be run in the R project environment locally. It will declare a simple list of movie Ids in the form of a JSON String and will return a similar list of recommended ids.

This file calls the `getRecommendation.R` file which outputs the list of recommendations and is deployed to the server.

In turn this file uses the recommender model created from the `createModel.R` file as the model used in the predict method.

The code and the resulting recommendations are displayed from an R project console below.

```
> library(recommenderlab)
> library(reshape2)
> library(jsonlite)

> source("R/getRecommendation.R")
> load(file = "data/recommender.rda")
>
>
> jsonString <- '{"movies": [34, 87, 236, 312, 397, 650], "ratings": [5, 2, 3, 4,
1, 2]}'
>
> # inside opencpu
> args <- fromJSON(jsonString)
> str(args)
List of 2
 $ movies : int [1:6] 34 87 236 312 397 650
 $ ratings: int [1:6] 5 2 3 4 1 2
> output <- do.call(getRecommendation, args)
> toJSON(output)
```

Resulting recommendations:

```
[[293,532,11,330,417,539,414,102,323,439]]
```

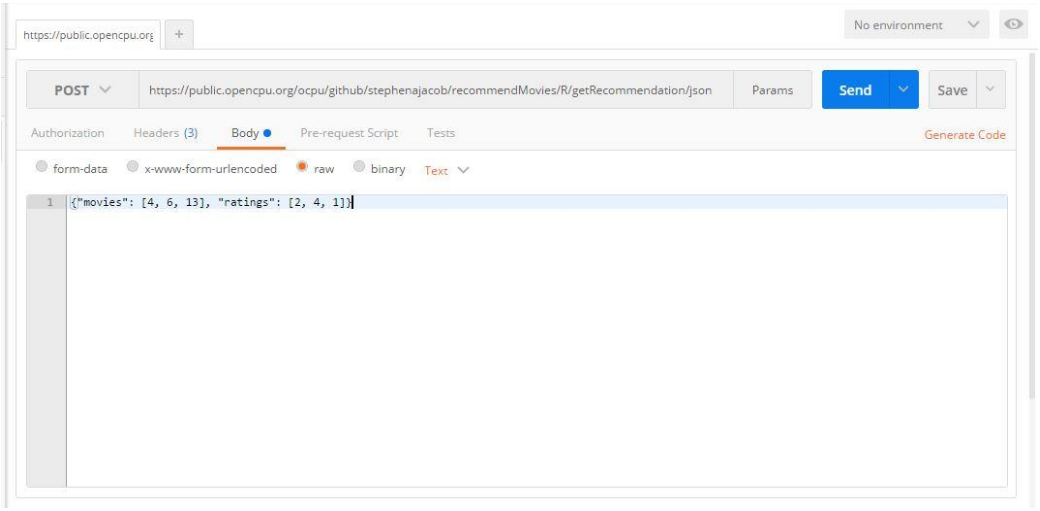
Test file testRecommend2 as an R Package

For this test case, the R.file will be converted into an R Package, the database will be loaded and the R application will be deployed to OpenCPU.

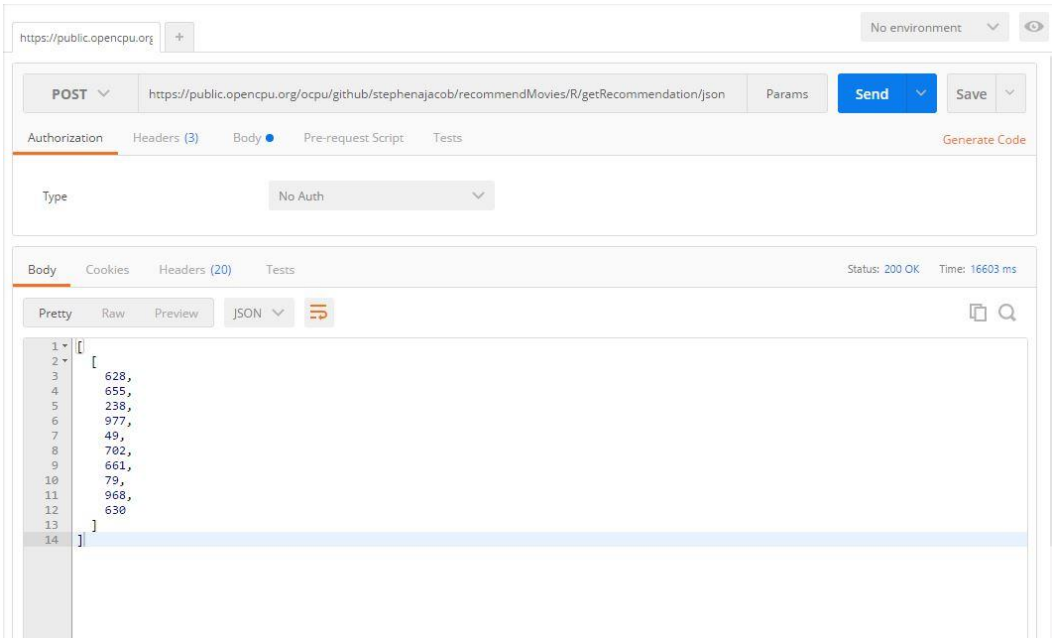
I will now run the same R file, getRecommendation within the following HTTP request of type POST from the application PostMan:

https://public.opencpu.org/ocpu/github/stephenajacob/recommendMovies/R/getRecommendation/json

The end result will be a list of recommended movie ids in JSON format. A screenshot from PostMan with the input JSON data of movie ids and ratings is seen below:



The following screenshot from PostMan below displays the recommendations from the JSON input from above:



Testing the Android Client

Model Movie class

This model class represented the actual movie component. Throughout the entire run of the application, all the following attributes were coded and tested:

- Data fields for the class
- Constructor for class
- Accessors and mutators

SQLiteDatabase

The implementation of the Android application uses a SQLiteDatabase to store, query and save data concerning movies.

REST Application

The Android client application was configured to send HTTP POST request to the R server to return to the user a list of recommended movies. This was possible via the MyHttpConnection class I declared.

This was tested whilst running the Android app by printing out the Http request url and then the returned list of recommendations as a JSON string of data.

The entire Android app was tested thoroughly and completely and as shown in the implementation all outputs, UIs and functions, were examined for correctness.

Conclusion

Objectives

The primary objective of this project was to develop an application to recommend to the user movies that appeal to their liking. The application would use a collaborative filtering algorithm to predict movies the user would be interested in. A secondary objective would be for the application to be deployed to a mobile device.

The architecture of the application itself would be composed of a server and a client component. The client allows the user to rate movies. It then makes calls to the server, sending those rated movies to the server in. The server component will then respond by sending movies recommended to the user back to the client component.

Project Overview

To implement the server component, it would be written in the programming language R. Another possible implementation language was Python. It was decided R would be the language to develop the server component. R had support for libraries, and software paradigms more equipped to develop a recommender system including the R library recommenderlab and the class RSpareMatrix to contain the data. R also supports paradigms for object-oriented programming. The server also had to store files of a collection of movies and a set of ratings of those movies by a set of users. Once the server component was completed, the R service was tested first as a local application.

Then it was necessary to deploy the server to a software service platform. The options I looked at included: yHat, Shiny, Azure and OpenCPU. I decided that the software platform I would use would be OpenCPU as the service would support REST calls from the client and easy updates to the application. I had to create a GitHub repository for my application in order to properly deploy it. Every update would be pushed to the GitHub respository, and this would trigger webhooks which would deploy those updates to the OpenCPU platform.

Now that the R service was capable of being deployed to the OpenCPU public service, it was necessary to test the service using the Google Chrome application Postman. This would happen in the following order. The client would rate a number of movies. Afterwards, the client makes HTTP POST requests to the server, sending these movies and their respective ratings as JSON data and the server would respond by sending back a list of movies recommended to the user also in JSON data. [Note that if a user only rated one or two movies for the request, it is possible that an empty set of

recommended movies would be returned.] It was shown that the overall functionality of the service was implemented correctly.

To implement the Android client side, it was necessary to store the movie data as previously used on the server side. The data would be stored in an SQLiteDatabase. On the Android client side, to make it possible to access and query the movie data, a Java class named DatabaseHandler was developed. The DatabaseHandler class was primarily used to query data and allow users to rate movies. They can query the movie data by searching for movies based on their titles or by specific genres. Resulting movies are displayed in lists with the components labelled with the movie titles, and to allow the active user to rate the movies, a widget called a RatingBar is displayed beneath the title. This RatingBar allows a user to actively rate the movies between one and five inclusively. An active user could also change their ratings later on as well as clear the application of its ratings. When a user rates a movie, an ActionListener object is responsible for updating the rating data. Finally the client makes a HTTP Post request to the R server, sending the movie data, that is ids and ratings to the server, and the server sends back movies recommended to the client.

Evaluation

I will now evaluate the use, design and functionality for this application. Regarding meeting the initial requirements of this project, they were all met to a large extent. For the server component, R was the chosen programming language used to implement it. The R programming language is highly recommended for statistical and machine learning programs. Statistical coding was needed for the recommendation algorithm required for the application so R was the ideal language for the server. In addition RStudio is a very user-friendly IDE for developing in R.

Regarding the client, Android was the ideal mobile platform, because I had worked with Android before. GitHub was also chosen to be a data repository host service as it supports version control and employs the use of webhooks which facilitate updates being posted to the application contained in the software hosting service OpenCPU. Because the client communicates with the server via HTTP REST requests of the type POST, OpenCPU was the ideal choice for a hosting server as it has support for REST communication. The SQLiteDataBase format was a very good choice for the database being used as it making SQL requests in the application relatively easy and Android could implement libraries that could access an SQLiteDatabase. In my opinion, the application works very effectively, handles a worldwide issue regarding recommending algorithms and has a detailed architecture.

A major decision I had to make was to choose a suitable software hosting service. I researched four hosting services until I chose OpenCPU. The deployment process for OpenCPU was particularly user-

friendly. When the R application was pushed to GitHub, webhooks defined in GitHub would automatically update the application on OpenCPU.

Further Work

If I was to expand the project further, I would offer further options in the user interface of the client. I would update the DatabaseHandler class to query for movies by their producer or their primary actor. This information for movies would need to be downloaded from a public service such as the Internet Movie Database (IMDB). If a user was more interested in books or music I'd present the option to select and search for movies, books or CDs on the opening screen of the application.

Also the current application returns the top ten recommendations. I would parameterize the application to allow users to change the number of recommended movies. I would also extend the service so that a user could request a personal rating for a particular movie.

I believe this was a very topical application to develop for my MSc project. I provided an application that is modelled after an activity that is common in the present day: recommending specific merchandise to customers. The coding and software tools used in the project have boosted my confidence in being a Java developer.

References:

[1] recommenderlab: A Framework for Developing and Testing Recommendation Algorithms - https://cran.r-project.org/web/packages/recommenderlab/vignettes/recommenderlab.pdf, [accessed 15 Dec 2015]

[2] RStudio Support: Writing Package Documentation - https://support.rstudio.com/hc/en-us/articles/200532317-Writing-Package-Documentation, [accessed 07 Sept 2015]

[3] MovieLens Datasets - http://grouplens.org/datasets/movielens/ , [accessed 15 Aug 2015]

[4] yHat -https://www.yhathq.com/ [accessed 20 Oct 2015]

[5] ShinyApp - http://shiny.rstudio.com [accessed 14 Oct 2015]

[6] Introduction to Recommender Systems by University of Minnesota - https://www.coursera.org/learn/recommender-systems/home, [accessed 29 Oct 2015]

[7] OpenCPU – An API for Embedded Scientific Computing, [accessed 03 Nov 2015]

[8] jsonlite – R Documentation - http://www.rdocumentation.org/packages/jsonlite, [accessed 22 Jan 2016]

[9] User-based Collaborative Filtering - https://cran.r-project.org/web/packages/recommenderlab/vignettes/recommenderlab.pdf, [accessed 29 Jan 2016]

[10] Introduction to Recommender Systems - https://www.coursera.org/learn/recommender-systems, [accessed 07 Feb 2016]

[11] Recommender Systems in E-Commerce - http://files.grouplens.org/papers/ec-99.pdf, [accessed 25 Feb 2016]

[12] Next Generation of Data Mining, Edited by Hillol Kargupta, Jiawei Han, Philip S. Yu, Rajeev Motwani, and Vipin Kumar, Chapter 11, Service-Oriented Architectures for Distributed and Mobile Knowledge Discovery, Domenico Talia and Paolo Trunfio