UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

Data Scraping, Data Visualisation and Data Modelling

Final Year Project

Aisling Smyth 18233511

Department of Mathematics and Statistics

BSc in Mathematical Sciences with Computing

Supervised by Professor Norma Bargary

27 March 2022

Table of Contents

Abstract

This project consists of 3 main parts: web-scraping, data visualisation, and finally data modelling using RStudio. I utilised the *rvest* package (Wickham & RStudio, 2021) to scrape the data from an IMDb site of 'The Best Movie Musicals' (IMDb, 2021), and from there created an interactive application, where a user can browse the available musicals and also input their preferences in order to generate a movie recommendation.

Preface

The first part of the project was to decide what website(s) to scrape and try and come up with a concept for what to do with this data. I began by looking at various sites listing open-source data sets (awesomedata, 2021) to try and get inspiration, followed by looking up web-scraping projects (Octoparse, 2021) that other people had carried out to see if I could find some possible options to do something similar.

My big passion in life is musical theatre, and so I really wanted to come up with something to do with this. I thought of ideas ranging from trying to predict ticket prices/sales, to performing sentiment analysis on scripts to see if there is a 'formula' for a successful musical.

Given that there are so many musicals available, I thought that I could create something that would help me to discover new ones I may like. This gave me the idea of trying to create a musical-recommendation application, such that using genres/actors/top-rated musicals, I could find other, similar ones to try.

Unfortunately, it proved very difficult to find much data with respect to live theatre. As a result, I used IMDb (IMDb, 2021), and my search is limited to movie musicals, of which (at the time of writing) there were 10,204 options.

Now that I had my idea, I split the work into 3 parts: scraping the data, analysing it and trying to create a prediction model, and finally, visualising it.

Data Scraping

Background

Before beginning this project, I had never encountered the concept of data scraping. Data scraping "*refers to a technique in which a computer program extracts data from output generated from another program*" (CloudFlare, 2021). The scraped data can then be easily accessed in a local file on your computer (for example in a spreadsheet) and the information can be used for any number of purposes. These range from scraping TripAdvisor to compare various holiday packages, to finding a public consensus on Twitter regarding a TV show, to combining the data with machine learning to try and predict your competitor's pricing.

However, while there are many advantages to data scraping, it can also be misused by certain people, for example to harvest email addresses from websites to sell them on to spammers/scammers (which is illegal in many jurisdictions). As a result, some sites employ measures to try and prevent their site from being scraped (for example, by regularly changing the HTML markup). This means that some sites, such as Facebook and Netflix, are very difficult to scrape.

Others will, for instance, make it very expensive to scrape their data. SoundCharts is a website which sells data from "*real-time Airplay, Playlists, Charts and Social media monitoring*" (Soundcharts, 2021), with the most basic package costing €49 per month, for only 10 artists' information! SoundCharts also restrict data scraping on the data you purchase (Soundcharts, 2021).

The Basics

When building a web page, it will generally be done using a combination of three main programming languages: Hypertext Markup Language (or HTML), Cascading Style Sheets (or CSS), and JavaScript. An analogy I found helpful while trying to understand each language's use is: "*the HTML document [provides] the bones of a webpage, while CSS provides the skin, and JavaScript provides the brains*" (Vodnik, 2020).

HTML (ISO, 2000)

HTML is used to create the body of a webpage, and specifies the type of information that the items on a webpage contains, and how they relate to one another in the page's overall structure; from its headings, to paragraphs, to images.

The code will be comprised of many paired 'tags', surrounded by <> (opening tag) and </> (closing tag) symbols. Whatever text is between the <> - for example body, h1 (h for heading, the number beside saying how big – h1 is bigger than h2 which is bigger than h3, and so on), or p (for paragraph) - determines what type of object the text between the opening and corresponding closing tag will be (Geeks, 2020).

Figure 1: HTML Tags (DataFlair, 2022)

For example, `<p> This is a paragraph! </p>` creates a paragraph containing the text "This is a paragraph!", while `<h1> This is a heading! </h1>` creates a heading containing the text "This is a heading!"

CSS (Meyer, 2006)

CSS improves the (not very visually appealing) website that has been created using HTML via controlling the site's appearance using colours, layouts and fonts.

For example, in order to make our HTML paragraph big, bold and blue, we would add the following CSS code (being careful to say 'color' and not 'colour') :

```
p{font-weight: bold;  color: blue; font-size: 200%;}
```
(Geeks, 2020).

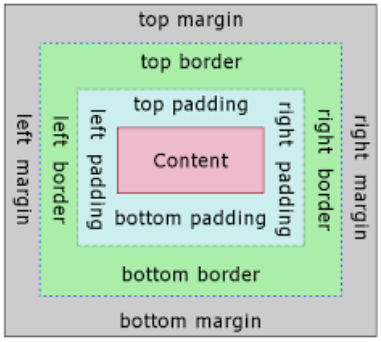


Figure 2: CSS Box Model (Darwish, 2013)

Finding the Data

To scrape the data from the site, we must start by looking at the source code of a site (`ctrl U` is a shortcut to see a page's source if you'd like to try). We then must use a combination of text matching/searching for the tags we want (e.g., a given paragraph that contains the data we want).

Sometimes this will be very easy, as some classes will be named, and then we just need to specify the class's name to find it. Other times, there will be multiple unnamed classes to sort through to find the one(s) we want.

My First Web-Scraping Tutorial – The Basics

Web-scraping can be performed using programming languages such as Python and also using some automated bots. However, I used the RStudio package rvest (coming from the phrase har**vest**) (Wickham & RStudio, 2021) to do my scraping.

I started by looking at this tutorial (Pascual, 2020), where I scraped data from a very sparse site, with only one paragraph, which looks like this:

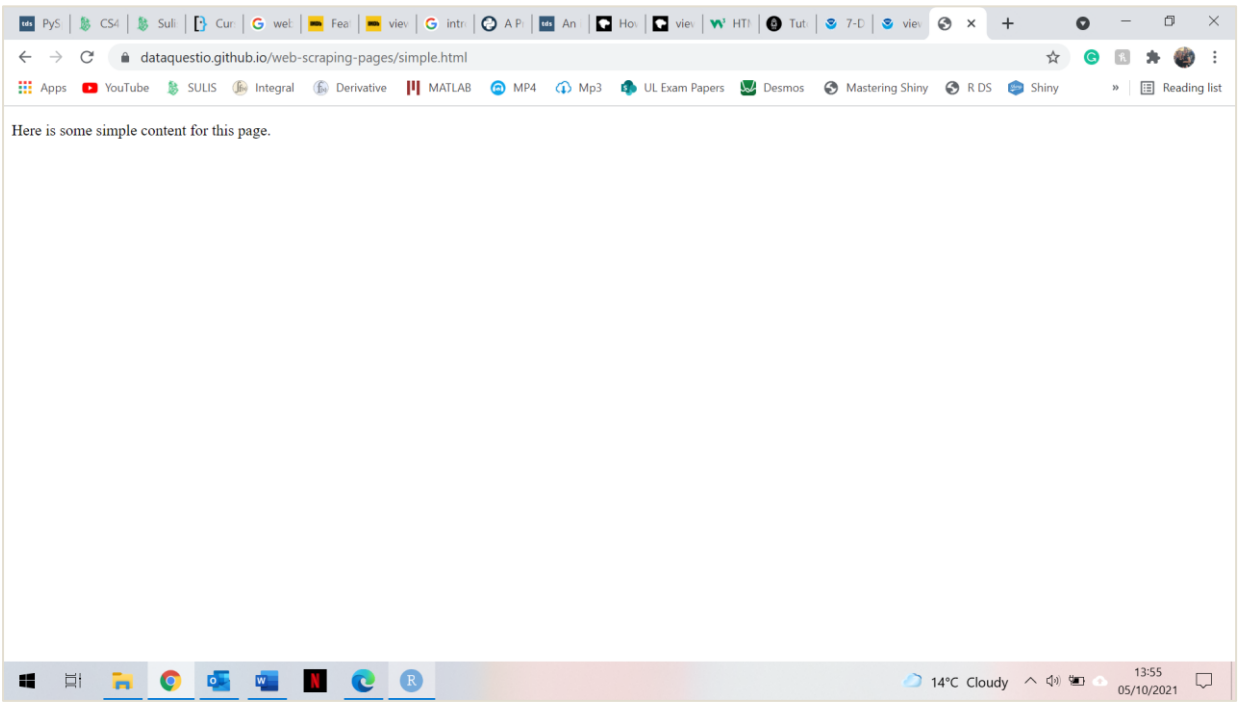


Figure 3: A Simple Webpage (Pascual, 2020)

We start by loading in the rvest library (Wickham & RStudio, 2021) in RStudio, and from there the `read_html()` command can be used to scan in the site we want by placing its link between the brackets (or in my case, giving the link a name – `link` – and then putting the link's name between the brackets), in this instance this site (Pascual, 2020).

```
library(rvest) # read in library
link = "https://dataquestio.github.io/web-scraping-pages/simple.html" # site we want
page = read_html(link) # read in site
# Alternatively could just do
# page = read_html("https://dataquestio.github.io/web-scraping-pages/simple.html") directly
```

Figure 4: Using read_html()

However, as we can see, the output is the whole page in HTML format, and so isn't very useful.

```
> page
{html_document}
<html>
[1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">\n<title>A simple  ...
[2] <body>\n        <p>Here is some simple content for this page.</p>\n    </body>
```

Figure 5: Using read_html()

Therefore, we need to use the `html_nodes()` command to search for a specific tag. To help pinpoint the class we want, there are some tools available, a common one being the inspector tool on the web browser. However, as some of these tools request permission to access all data on all sites you visit, I decided it was safer to find the information 'by hand', meaning manually reading through the source code to find the sections I wanted.

In this example, there is only one paragraph (denoted in tags by 'p'), so I'll just search for that node (assigning it the name '`paragraph_node`').

```
paragraph_node = page %>% html_nodes("p")
# apply html_nodes() to our page, searching for a paragraph, assign it the name 'paragraph_node'
```

Figure 6: Using html_nodes()

Again, the output is in HTML format, but this time it's just the paragraph (as opposed to the whole page like before).

```
> paragraph_node
{xml_nodeset (1)}
[1] <p>Here is some simple content for this page.</p>
```

Figure 7: Identifying a Specific Node

From there, we can parse this node using `html_text()` to view just the actual (plain) text found, and assign it the name '`paragraph_text`':

```
paragraph_text = paragraph_node %>% html_text()
# apply html_text() function to our node, and assign it the name 'paragraph_text'
```

Figure 8: Parsing A Node Using html_text()

And here is the output:

```
> paragraph_text
[1] "Here is some simple content for this page."
```

Figure 9: paragraph_text's Output

'Actual' webpages would have many more tags and paragraphs, but this really simplistic first-try helped me understand the basics of web scraping.

A More Complex Webpage - Finding Tags

Having scraped a 'bare bones' site, I then followed another, more advanced tutorial (Pascual, 2020), to scrape the American National Weather Service website (Service, 2021), showing the weather forecast for San Francisco. This website is much more complex and looks like this:

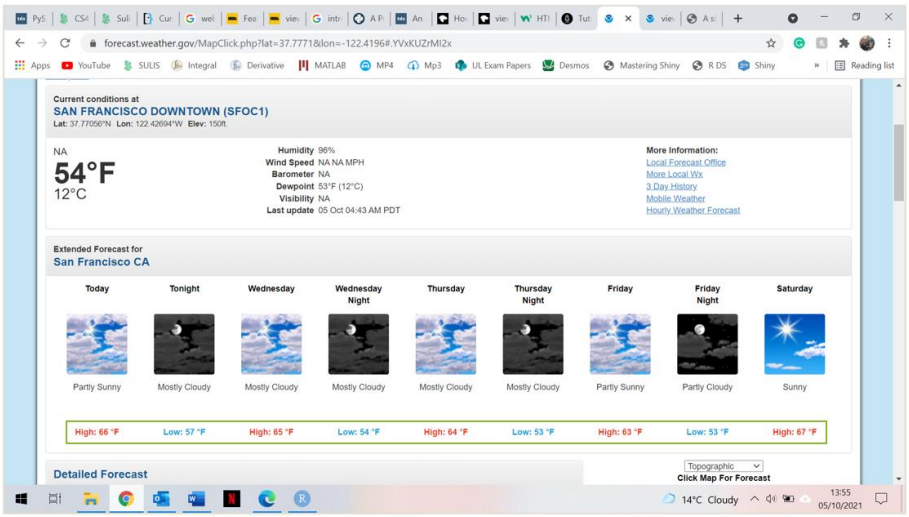


Figure 10: National Weather Service Forecast (Service, 2021)

The site has many paragraphs and so I had to find the tags I wanted. I decided to look at the 7-day forecast (in the green box in Figure 10). To do this, I viewed to page's source (using `ctrl U`) and used `ctrl F` to navigate to where the '7-Day Forecast' is mentioned:

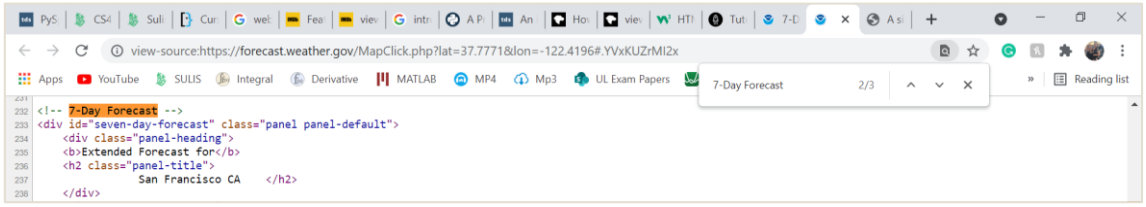


Figure 11: Finding the 7-Day Forecast (Service, 2021)

I was only interested in the temperatures (both high and low), which as we can see are between the 'temp' tags here, found by scrolling across:

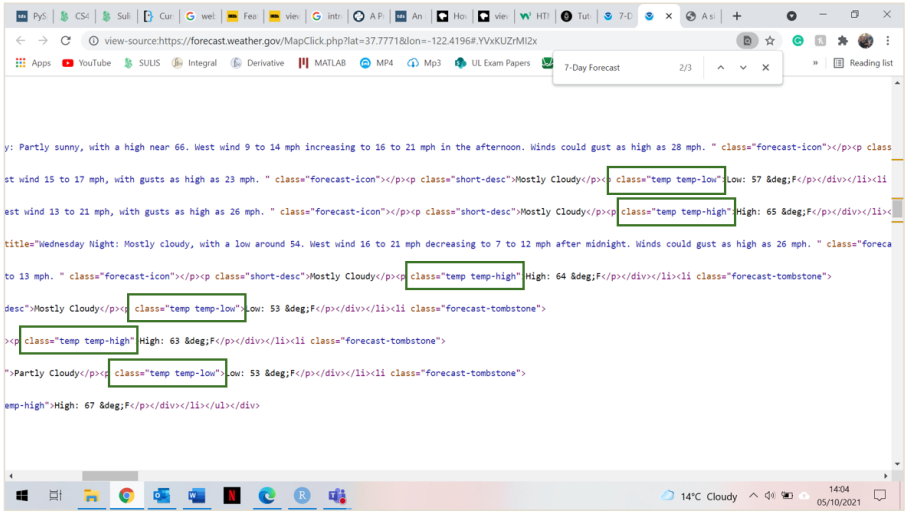


Figure 12: Finding the 'temp' Tag (Service, 2021)

I now know I'm looking for the 'temp' node, and so I can use the steps in the previous tutorial to read in the page and find the data I want:

```
page = read_html("https://forecast.weather.gov/MapClick.php?lat=37.7771&lon=-122.4196#.XlOj6BNKhTY")
node = page %>% html_nodes(".temp") # find temp node
forecasts = node %>% html_text() # convert node to text
```

Figure 13: Identifying the 'temp' Nodes

This gives the following output:

```
> forecasts # print out forecasts
[1] "High: 66 °F" "Low: 57 °F"  "High: 65 °F" "Low: 54 °F"  "High: 64 °F" "Low: 53 °F"  "High: 63 °F"
[8] "Low: 53 °F"  "High: 67 °F"
```

Figure 14: Temp Nodes Output

The above output is what we wanted, and is the same as the information shown in the green box in Figure 10. For this data to be useable, we need to do further manipulation, however as this was merely to see how to scrape, this is sufficient.

First Attempt at Scraping IMDb

I had not anticipated trying to scrape my own webpage yet, however upon looking for another tutorial on web scraping, I discovered this tutorial (Kaushik, 2017) to scrape an IMDb page, so I changed the URL to be the site I wanted to use (IMDb, 2021). While the approach I finally used was quite different to the one in this tutorial, it still taught me some interesting lessons.

Beginning in the usual manner, we first read in our data:

```
#Loading the rvest package
library('rvest')

#Specifying the url for desired website to be scraped
url <- 'https://www.imdb.com/search/title/?title_type=feature&genres=musical&view=advanced'

#Reading the HTML code from the website
webpage <- read_html(url)
```

Figure 15: Reading in the IMDb Data

From there, we scrape:

- Rank: The rank of the film from 1 to 100 on the list of 100 most popular musicals.
- Title: The title of the film.
- Description: The description of the film.
- Runtime: The duration of the film.
- Genre: The genre of the film.
- Rating: The IMDb rating of the film.
- Metascore: The metascore on the IMDb website for the film.
- Votes: Votes cast in favour of the film.
- Gross_Earning_in_Mil: The gross earnings of the film in millions of US dollars.
- Director: The main director of the film. Note, in case of multiple directors, I will take only the first.
- Actor: The main actor in the film. Note, in case of multiple actors, I will take only the first.

Elementwise, starting with rank and using similar methodology for each item, we find the node we want, convert it to text and then perform any manipulation necessary (e.g., converting to a number, removing commas, and so on):

```
# Rank Information
#Using CSS selectors to scrape the rankings section
rank_data_html <- html_nodes(webpage,'.text-primary')
#Converting the ranking data to text
rank_data <- html_text(rank_data_html)
#Let's have a look at the rankings
head(rank_data)
length(rank_data)
#Data-Preprocessing: Converting rankings to numerical
rank_data<-as.numeric(rank_data)
#Let's have another look at the rankings
head(rank_data)
```

Figure 16: Finding 'Rank' Information (Kaushik, 2017)

However, for some fields (such as here with metsascore), we have less than 100 results. This is an issue, given we are trying to scrape 100 movies' information. This could also occur, for example, with regards to gross earnings, as if a movie has not yet been released, this information does not exist.

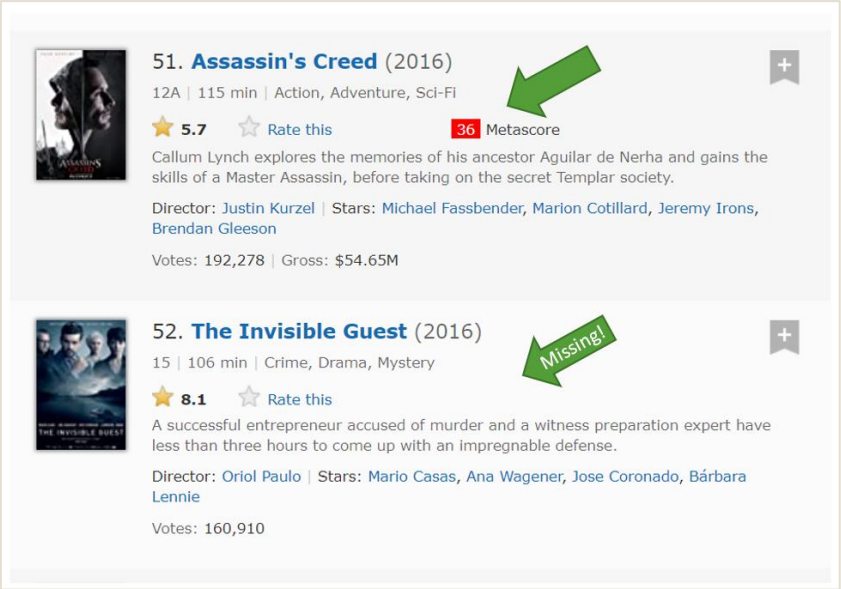


Figure 17: Missing Metascores (IMDb, 2021)

The tutorial deals with such a problem by manually entering *NA* for the movies missing this field. Unfortunately, we cannot just add 4 *NA*s to the list of metascores, as it will then map *NA* to the final 4 movies, as opposed to the ones which are missing this field. This results in a need to manually inspect the page for missing metascores, and then replacing the scores by specifying which movies are missing (an impractical approach for large numbers of movies).

```
#Lets check the length of metascore data
length(metascore_data) # currently 96
# Visually fix missing ones at specified indexes
for (i in c(52, 58, 76, 91)){
  a<-metascore_data[1:(i-1)]
  b<-metascore_data[i:length(metascore_data)]
  metascore_data<-append(a,list("NA"))
  metascore_data<-append(metascore_data,b)
}
#Let's have another look at length of the metascore data
length(metascore_data) # now 100
```

Figure 18: Fixing Missing Metascores (Kaushik, 2017)

The same approach has to be used across several scraped variables, and results in the following table (here only showing first 3 rows and 3 columns):

Rank	Title	Description
1	Dear Evan Hansen	Film adaptation of the Tony and Grammy Award-winning m...
2	My Little Pony: A New Generation	After the time of the Mane 6, Sunny--a young Earth Pony--a...
3	Encanto	A modern movie musical with a bold take on the classic fair...

Figure 19: First IMDb-Scraping Result

While this gave quite satisfactory results (i.e., a table with the information about each movie and *NA*s for missing data), it was not going to be practical for many movies (in my case, I scraped over 10,000 movies and thus I couldn't go through them manually).

I concluded that this issue arose from scraping 'by column' (i.e., all ranks, then all titles, then all descriptions and so on), versus 'by row' (i.e., by movie), as it didn't differentiate between which movies were missing data, just got the list of all available [ranks/titles/descriptions etc.]. From here, I now knew I needed to scrape by movie, and so I started my first attempt at scraping.

My First Attempt – Scraping by Movie

Using what I had learned previously, I had to figure out how to scrape 'by movie' instead of 'by heading'. From the previous example (Kaushik, 2017), I knew most of the necessary tags already, and added a few extra pieces of information, such as age rating.

Unfortunately, to be able to know precisely which movies were missing data (or equivalently at what indices to put my *NA*s), I had to create a function where you could feed in a given index (the movie's position on the page; the first being 1, and so on), and the information for that movie would be scraped. This resulted in the need for a loop, to iterate through all of the given movies on a page.

Having previously obtained all the necessary tags, it was straight-forward to cater to each movie. Before entering each piece of information into the dataframe, I checked to make sure it existed (and if not replaced it with *NA*), and so I quickly had a dataframe with the information I needed.

However, while looking at my dataframe of movies, I noticed that quite a few movie descriptions were assigned the value *NA*:

Rank	Title	Description
1	Dear Evan Hansen	Film adaptation of the Tony and Grammy Award-winning m...
2	My Little Pony: A New Generation	After the time of the Mane 6, Sunny--a young Earth Pony--a...
3	Encanto	NA

Figure 20: NA Issue

But when looking at the webpage for that movie, I could see that a description was in fact available:

Figure 21: Missing Description (IMDb, 2021)

As a result, it was clear that I needed to re-scrape the page, and so I began debugging my code.

My Second Attempt - Debugging

While looking at rows with missing descriptions, I noticed that the same movies had issues in the directors' and actors' columns also as shown below:

Directors	Actors
Stephen Chbosky	Ben Platt, Julianne Moore, Kaitlyn Dever, Amy Adams
Robert Cullen, José Luis Ucha, Mark Fattibene	Vanessa Hudgens, Kimiko Glenn, James Marsden, Sofia Cars...
character(0), Jared Bush, Byron Howard, Charise Castro Smith	character(0), Diane Guerrero, Stephanie Beatriz, John Leguiz...
Kay Cannon	Camila Cabello, Billy Porter, Nicholas Galitzine, Idina Menzel
Jonathan Butterell, Dan Gillespie Sells, Tom MacRae	Max Harwood, Lauren Patel, Sarah Lancashire, Shobna Gulati
Randal Kleiser	John Travolta, Olivia Newton-John, Stockard Channing, Jeff ...
Henry Selick	Danny Elfman, Chris Sarandon, Catherine O'Hara, William Hi...
character(0), Steven Spielberg	character(0), Ansel Elgort, Rachel Zegler, Ariana DeBose, Dav...
Michael Gracey	Hugh Jackman, Michelle Williams, Zac Efron, Zendaya

Figure 22: character(0) Glitch

On further investigation, I realised that sometimes the website's source code would differ on where the description was kept for different movies. Effectively, sometimes it was saved as its own entity, and sometimes it was stored with actors and directors, hence why the two tended to be linked.

I therefore created a check in my code such that if the description 'didn't exist' - where we expected it to - and therefore its length was 0, I would look in the other relevant place for it.

```
desc <- movie %>% html_nodes('.ratings-bar+ .text-muted') %>%
  html_text() %>% gsub("\n","",.)
movie_desc <- ifelse(length(desc)!=0, desc, toString((movie %>% html_nodes(".text-muted+ p"))[1] %>%
                                                       html_text() %>% gsub("\n","",.)))
```

Figure 23: checking description length

I then altered my directors/actors code to reflect this change accordingly:

```
directors <- toString(movie %>% html_nodes(".text-muted+ p") %>%
                            html_text() %>% gsub("\n","",.) %>%
                            str_extract_all(., "(?<=:).+(?=  \\|)") %>%
                            str_trim("right"))
movie_directors <- ifelse(length(desc)!=0, directors, toString((movie %>% html_nodes(".text-muted+ p"))[2]
                                                                html_text() %>% gsub("\n","",.) %>%
                                                                str_extract_all(., "(?<=:).+(?=  \\|)") %
                                                                str_trim("right")))

actors <- toString(movie %>% html_nodes(".text-muted+ p") %>%
                           html_text() %>% gsub("\n","",.) %>%
                           str_extract_all(., "(?<=Stars:).+") %>%
                           str_trim("right"))
movie_actors <- ifelse(length(desc)!=0, actors, toString((movie %>% html_nodes(".text-muted+ p"))[2] %>%
                                                          html_text() %>% gsub("\n","",.) %>%
                                                          str_extract_all(., "(?<=Stars:).+") %>%
                                                          str_trim("right")))
```

Figure 24: altering director/actor code

Once I fixed this bug, I was happy with how my pages were scraping. However, I was only doing one page at a time and needed to implement a way of scraping multiple pages at once.

Scraping Multiple Pages

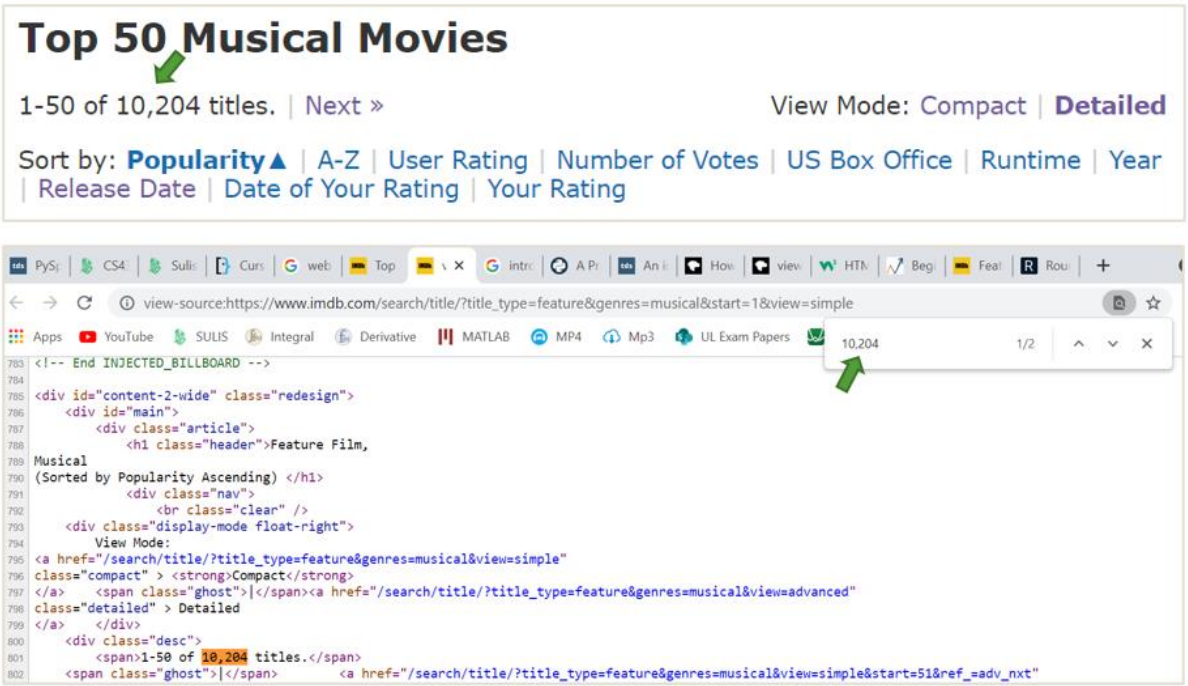
It quickly became clear that there was a pattern in how the links were generated, namely of the form:

'*https://www.imdb.com/search/title/?title_type=feature&genres=musical&start=ID&ref_=adv_nxt*'

Where ID is given by: *1 + (PAGE_NUMBER - 1)*50* (i.e., for page 1 is 1, page 2 is 51, page 3 is 101 etc.).

Figure 25: Page Link Format (IMDb, 2021)

As the number of movies/pages was bound to change over time (it changed from 10,201 to 10,204 in the time it took to type this and then go back to screenshot), I found where the number of titles was saved in the source code by visually checking how many titles were listed, and then using `ctrl F` to quickly find it. From there, I found the total number of pages by dividing the total number of movies by 50 – as there's 50 on each page – and then rounding up.




```
number_of_movies <- (page %>% html_nodes(".desc") %>% html_text())[1] %>%
  str_extract_all(., "(?<= of).+(?= titles)") %>%
  str_trim("both") %>% parse_number()
number_of_pages <- ceiling(number_of_movies/50)
```

Figure 26: Extracting the Number of Titles (IMDb, 2021)

From there, I looped through my previous code by the number of pages (i.e., scraped each page individually), changing the URL to match the previously found format:

```
for(j in 1:number_of_pages)
  {
    link = paste("https://www.imdb.com/search/title/?title_type=feature&genres=musical&start=",
    1 + (j-1)*50, "&ref_=adv_nxt", sep = "")
```

Figure 27: looping through the number of pages

While it was not ideal to have to use a loop, 10,000+ movies were scraped in (on average) about 15 minutes. For my purposes, this timeline was acceptable.

I now had my dataset containing information on all movies of interest, and it was time to start preparing the data for analysis.

Data Preparation

Genres, Directors and Actors

When preparing the data, the genres, directors, and actors were saved as a list. Given that, as part of a recommendation engine, it would be helpful to search by genre (e.g., all family musicals), this posed an issue, because if I were to check if the genre was family, this would return false.

Looking at 'Dear Evan Hansen' as an example, its genres are 'drama' and 'musical':

```
> movies$Genre[1]
[[1]]
[1] "Drama"   "Musical"
```

Figure 28: Dear Evan Hansen's Genres

However, if I merely checked if the movies genre 'equals' drama, this would return false:

```
> movies$Genre[1] == "Drama"
[1] FALSE
```

Figure 29: Genre Does Not Equal Drama

However, if I checked elementwise (i.e., by un-listing), it would return as expected, that the first genre was drama, and the second was not.

```
> movies$Genre[1] %>% unlist() == "Drama"
[1]  TRUE FALSE
```

Figure 30: Unlisting to Check Genre

And so, the issue was because not *all* of the genres in the list were drama, and so I had to check if *any* of the listed genres were drama, not if *all* were.

```
> any(movies$Genre[1] %>% unlist() == "Drama")
[1] TRUE
```

Figure 31: Checking If Any Genre is Drama

I used a similar approach for both directors and actors.

Issues Scraping

I scraped the page containing the 'Top Movie Musicals' (IMDb, 2021), but after looking at a few movies, it became apparent that this page uses a condensed version of the data from each individual movie's IMDb page, and so the data I was seeing on the 'summary' page was a condensed version of the 'actual' (full) version. This caused a few issues, which I will discuss below.

Missing Genres

I first noticed this issue when I realised that not every movie had 'Musical' listed as a genre, such as 'The Nightmare Before Christmas':

Figure 32: Nightmare Before Christmas Genres (IMDb, 2021)

As this was a list of movie musicals, I wondered how this movie could be considered a musical if it's not listed as one (I knew it was a musical, I just didn't know how IMDb knew). This led to me looking at the 'The Nightmare Before Christmas' individual IMDb page. Looking at its listed genres below we can see four genres:

Figure 33: Nightmare Before Christmas Full Genres (IMDb, 2021)

Age Ratings

While scraping, it was evident that the value returned as the age rating would not always match the one I could see on the site. After some investigation, this appeared to be because I was scraping the Irish age-rating, while the page was showing me the UK rating, and so the two occasionally differed.

This explained why I was seeing a different age rating on the website to what I was recording when scraping. The page itself, and its source code, listed the UK rating, however IMDb was picking up that I was in Ireland, and so I was scraping the Irish age rating. An example of this is evident in 'The Rocky Horror Picture Show'. The page I scraped (and its source code) shows the age rating as being 'AA':

Figure 34: Rocky Horror Age Rating (IMDb, 2021)

However, looking at the IMDb page for 'The Rocky Horror Picture Show', the Irish age rating is 18, and so the data I scraped showed 18 as being the age:

Figure 35: Full Certification List (IMDb, 2021)

Images

Continuing using 'The Rocky Horror Picture Show' as an example, I scraped the image listed on the page. However, this is a very poor-quality image (especially apparent in the 'Search Movies' tab), while the specific IMDb page has the same image in much better quality. Here is a comparison:

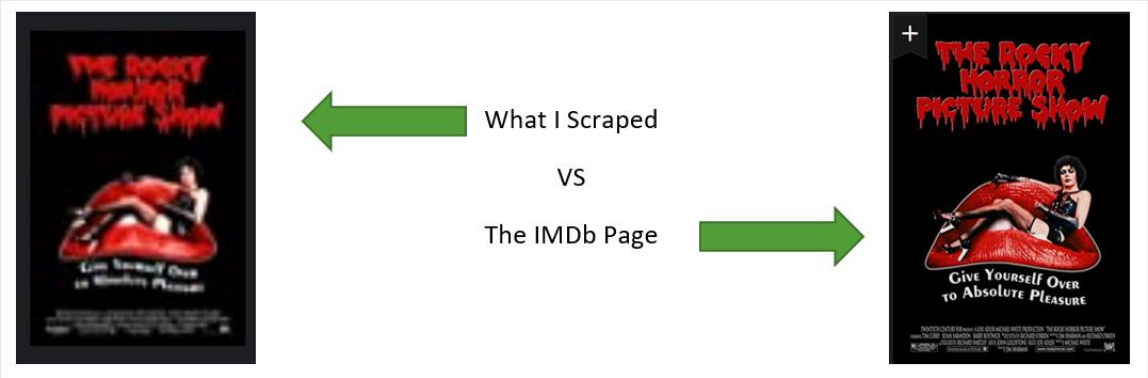


Figure 36: What I Scraped VS the Proper Image (IMDb, 2021)

Solution: Re-Scraping

The only solution to these various issues would be to re-scrape the data, this time pulling from each movie's individual page. Unfortunately, when I tried to begin re-scraping the data, it became apparent that (within the timeframe and scope of this project) this would not be possible. As I discussed at the beginning of this section, some sites make their pages very difficult to scrape. While the 'summary pages' (such as what I scraped initially) are readily available to scrape, the movies' individual pages are much harder to scrape.

To begin with, the page was not actually written in HTML (so I would have had to figure out the language being used and also learn how to scrape it), and furthermore, there appeared to be certain 'security' built in, which would have made the page very hard to scrape (even knowing the language).

As a result, I was ultimately unable to do this within the given timeframe. The scraped age ratings are legitimate age ratings and so aren't really any issue, the scraped genres are correct (just a subset of the full list), and the scraped images are the same (just much, much smaller and hence lower resolution). This means that, while my app would have been more visually appealing with nicer images, and I could have made better predictions with a fuller genre list for each movie, my app is functional and is certainly a good representation of the page I scraped.

This proved a good learning lesson in seeing the 'security measures' a site can use in action, and furthermore showed me the importance of ensuring the page you scrape contains all the information you want or need.

Data Visualisation

Basic Introduction to R Shiny

Shiny Background

To develop an interactive movie recommendation app, I used R Shiny (RStudio, 2012). I had never encountered Shiny before, and so began learning how to use it using a mix of the book '*Mastering Shiny*' (Wickham, 2020) and the tutorial '*Shiny in Seven Lessons*' (RStudio, 2020).

The basic idea of creating a Shiny app is:

1. Load the Shiny library using the `library(shiny)` command.
2. Create a user interface (UI – controls what the app looks like) and a corresponding server (controls what the app does).
3. Combine these to construct an app using the `shinyApp(ui, server)` function.

For example, the code below creates a very basic Shiny app. It does not do anything (as the server is empty), and doesn't look very impressive due to a sparse UI, but it shows the very basic foundation of a Shiny app:

```
# Load in relevant library
library(shiny)

# Create your user interface (UI) - this is what the app looks like
ui <- fluidPage(
  "Hello, world!"
)

# Create your server - this is what the app does
server <- function(input, output, session) {
}

# Construct your app
shinyApp(ui, server)
```

Figure 37: A Basic Shiny App (Wickham, 2020)

Which in turn creates this (very sparse) app:

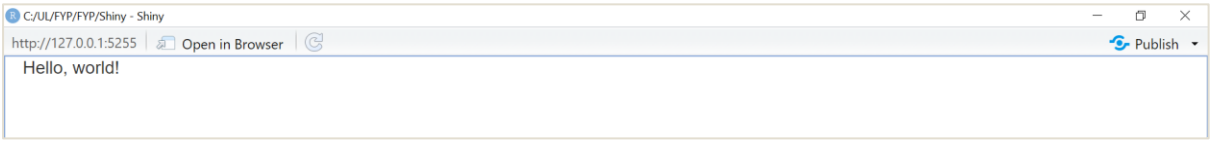


Figure 38: Output of a Basic Shiny App (Wickham, 2020)

Creating a User Interface (UI)

The UI (Churchville, 2021) determines the appearance (e.g., colours, fonts, effects) and layout (e.g., spacing, position) of the Shiny app. An app with a complex UI, but empty server, will look great, but do nothing.

This is a fine balance: making a project that looks good, without losing functionality or being less informative as a result. For example, while graphing, a plain graph may be easy to understand, while a very intricately designed graph would look good, but may be hard to read.

Some of the UI features I used a lot were:

- **Page layout tools:** these determine where everything sits on the app page.
    - The `fluidRow()` function creates rows on your page (i.e., determines where the information will appear vertically on the page).
    - You can then fill these rows using the `column()` function, which creates columns on the page (i.e., specifies where the information will appear horizontally on the page).
- **Stylistic tools:** these determine how things look.
    - I specified a colour scheme using the `theme()` function.
    - I created tabs using the `tabPanel()` function, and inserted images using the `img()` tool.
- **Input tools:** these record the user's inputs for use later in the server.
    - The `selectInput()` command creates a dropdown menu for the dataset you specify.
    - The `checkboxGroupInput()` command is similar, but uses checkboxes instead of a dropdown menu.
- **Output tools:** tells the app where to place an output specified in the server.
    - The `uiOutput()` specifies an output generated based on a user input will be placed at this location.
    - A `tableOutput()` places a table where specified.

Creating a Server

The server depends on the UI you have created. Your UI code will include a lot of (uniquely named) inputs (for example, one of the parameters for the `selectInput()` function is an '`inputId`'). These IDs can then be specified in the server (Ottinger, 2008) to do something, such as creating a table of the data selected in the dropdown menu.

When creating a UI, you have to anticipate your outputs. For example, I had to place tables and dropdown menus etc. on a page that didn't exist yet (i.e., no user inputs) as I was creating the code. This did take a bit of 'back and forth' as I created more functionalities in my server.

Some of the server functions I used were:

- `renderUI()` calls to specify what I wanted to appear in the `uiOutput()`s I created in my UI, for example to show the movie title, description, IMDb link and picture for the movie chosen in the dropdown menu.
- `renderTable()` calls to specify what table I wanted to appear in the `tableOutput()`s I had created in my UI.

Creating My App

Inspiration

While trying to design the look of my app, I came found a movie recommendation engine (Shubhansh2006, 2019), whose style I liked, and started tailoring that code to my needs. I ideally would have created a recommendation system first so that my app could recommend movies, however as my initial aim was to create an app that had all the 'placeholders' I would subsequently need (vs one that entirely works 'from the get-go'), I created placeholders where future functions would go.

'About' Tab

The first tab in my app is an 'About' tab (see Figure 39 below). It gives a brief overview of why the app was created, what it does, and then includes hyperlinks to my R code and to the app itself:

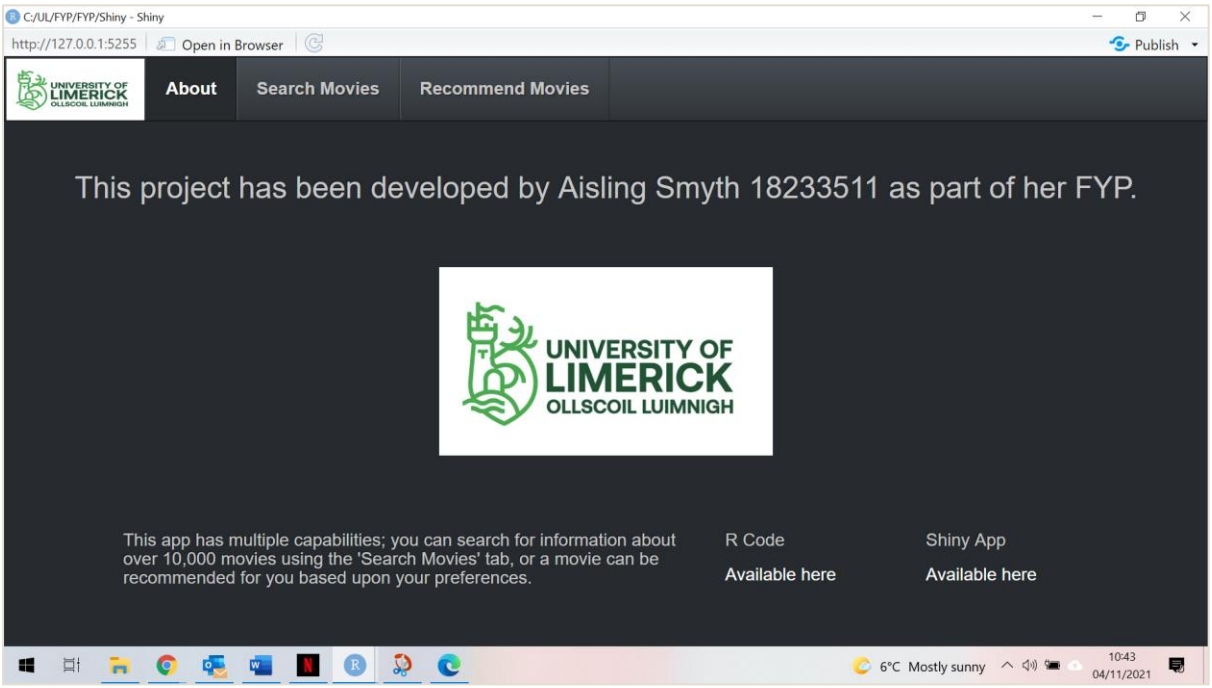


Figure 39: The 'About' Tab

This page was much harder to make than it initially appears. While it doesn't have much functionality in-built (it only contains 2 hyperlinks, and the rest is fixed), it was remarkably difficult to layout. In particular, I used a `fluidRow()` at the bottom to incorporate the paragraph of text, and the two links. Deciding the spacing (widths, heights, alignments etc.), and also how they interacted with each other took much trial and error until I was happy with how it all looked.

'Search Movies' Tab

The initial idea for my second tab was catering to people who want to browse a list of available movie musicals. Upon choosing the movie you want more information about (via the dropdown menu), its title, description, IMDb link and image would be displayed.

I used `wellPanel` to create a slightly darker box, and added in `br()` commands to create gaps for everything to fit better (both shown below in figure 40).

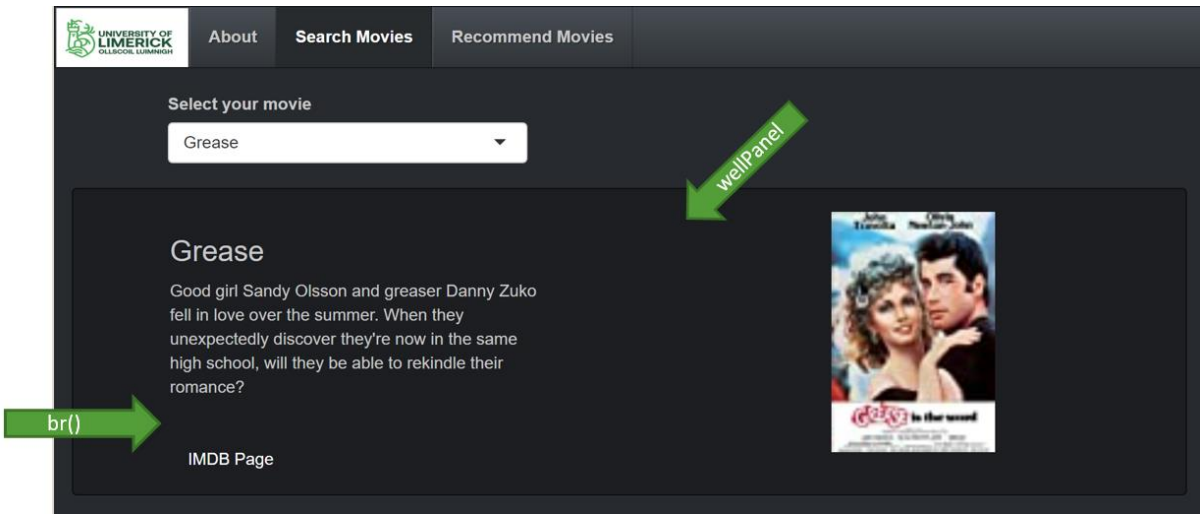


Figure 40: The 'Search Movies' Tab

This page gives a good example of the code required to create this UI:

```
tabPanel(tags$b('Search Movies'), # Create the 'Search Movies' Tab
         fluidRow(column(1), # gap to the left
                  column(8, # dropdown menu is '8 columns' wide
                         selectInput("Movie_chosen", "Select your movie",
                                     movies$Title) # dropdown of titles
                  )),
         wellPanel( # creates darker background
           fluidRow(column(1), # gap to the left
                    column(7, # dropdown menu is '7 columns' wide
                           fluidRow(column(7,uiOutput("Movie_chosen"))), # display title
                           fluidRow(column(7,uiOutput("movie_desc"))), # display description
                           br(), # skip a line
                           br(), # skip a line
                           fluidRow(column(7,uiOutput("Imdb_Link"))) # hyperlink to IMDb
                    ),
                    column(4, htmlOutput('Image')) # show image
           )
         )
),
```

Figure 41: Basic UI Code (Shubhansh2006, 2019)

Note that in my UI, '`Movie_chosen`' was the ID for the dropdown menu (as in a dark green box in figure 41 above), so I use `input$Movie_chosen` in my server (dark green boxes below in figure 42) to specify which input I am using to create my output.

Equally, in my server (below), I use `output$name` (light green boxes in figure 42) to specify the output's name, which I then reference in the UI (light green boxes above in figure 41) to specify which output I want to show there.

The corresponding server is:

```
# Search Movies Tab
output$Movie_chosen <- renderUI({
  fluidRow(h3(input$Movie_chosen)) # Display title chosen in dropdown
})

output$movie_desc <- renderUI({ # Display description of title chosen in dropdown
  fluidRow(movies[movies$Title == input$Movie_chosen, 'Description'])
})

output$Image <- renderUI({ # Display image of title chosen in dropdown
  poster_path <- movies[movies$Title == input$Movie_chosen,'ImageLink']
  img(src = poster_path, height = "200")
})

output$Imdb_Link<-renderUI({ # Create IMDb hyperlink of title chosen in dropdown
  a("IMDB Page",
    href = movies[movies$Title == input$Movie_chosen,'Link'])

})
```

Figure 42: Basic Server Code (Shubhansh2006, 2019)

Final Tab

While I did still think it made sense to have a tab where a user could search for a particular movie, I decided to add a functionality that showed movies similar to the movie the user was looking at, inspired by the traditional '*you may also like…*' feature on sites such as Netflix.

I altered my recommendation function to find the most highly-rated similar movies, and reused the initial UI code to print out the top 5 similar movies. Now the user could learn more about the movie they were looking at, while also being shown similar movies:

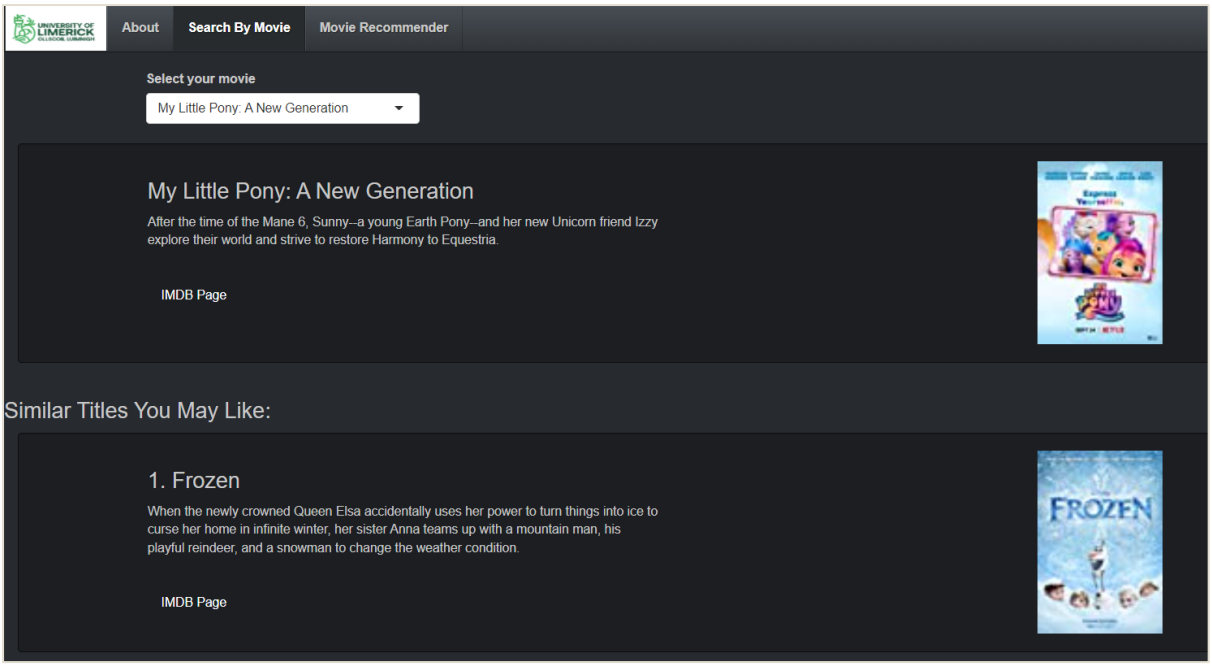


Figure 43: Updated 'Search by Movie' Tab

‘Recommend Movies’ Tab

This tab took the most work; it required a lot of interaction between the UI and server, had a lot of different components, and implements the recommender system itself.

In the original design of the app, there were initially 6 blank dropdown menus as shown in Figure 44:

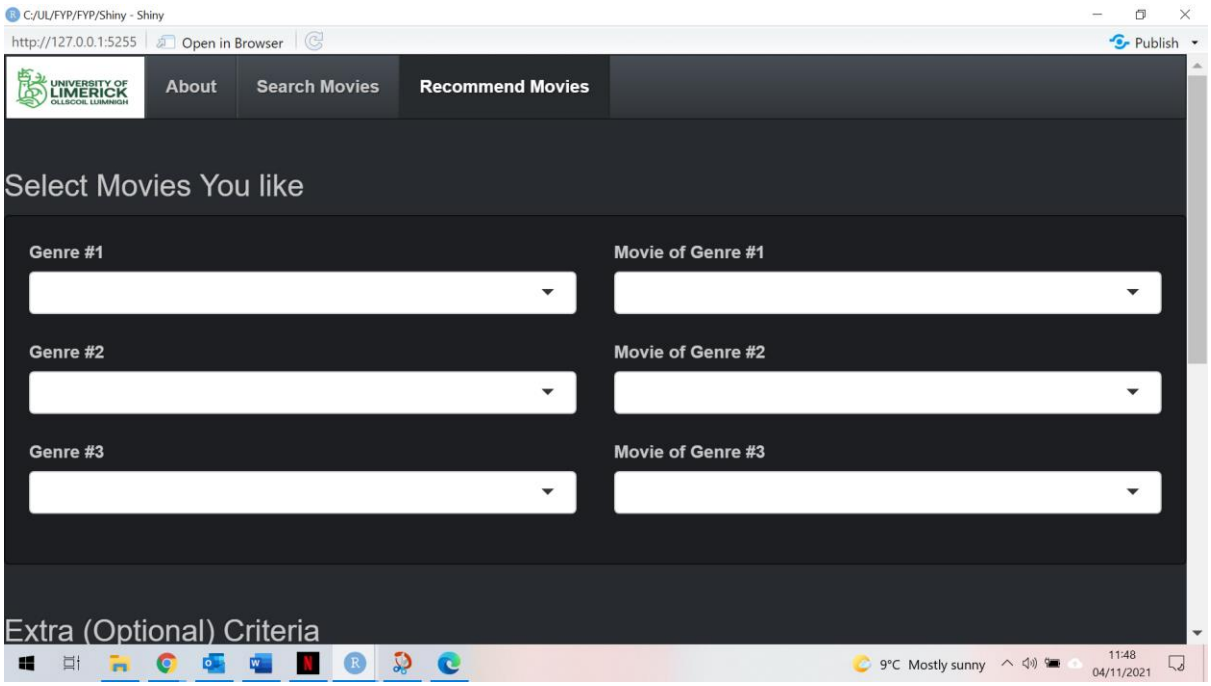


Figure 44: The 'Recommend Movies' Tab

When you chose a genre (left), its corresponding box on the right showed movie titles of that genre as shown below in figure 45:

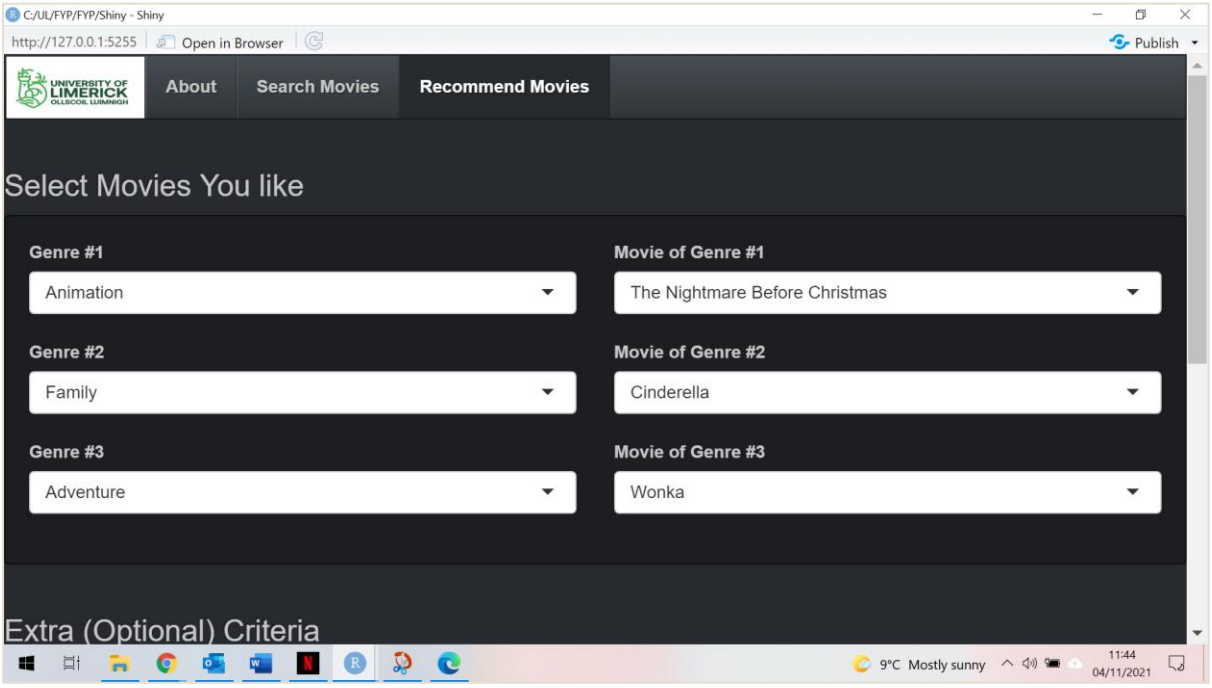


Figure 45: Filled In 'Recommend Movies' Tab

I also added some optional criteria to help personalise recommendations even more. If they are not selected, they default to impose no constraints. You can specify the date range in which the movie was released, the required age range (for example if you enter '13', only movies suitable for 13 or younger will be recommended), and also select any specific genres you'd like:

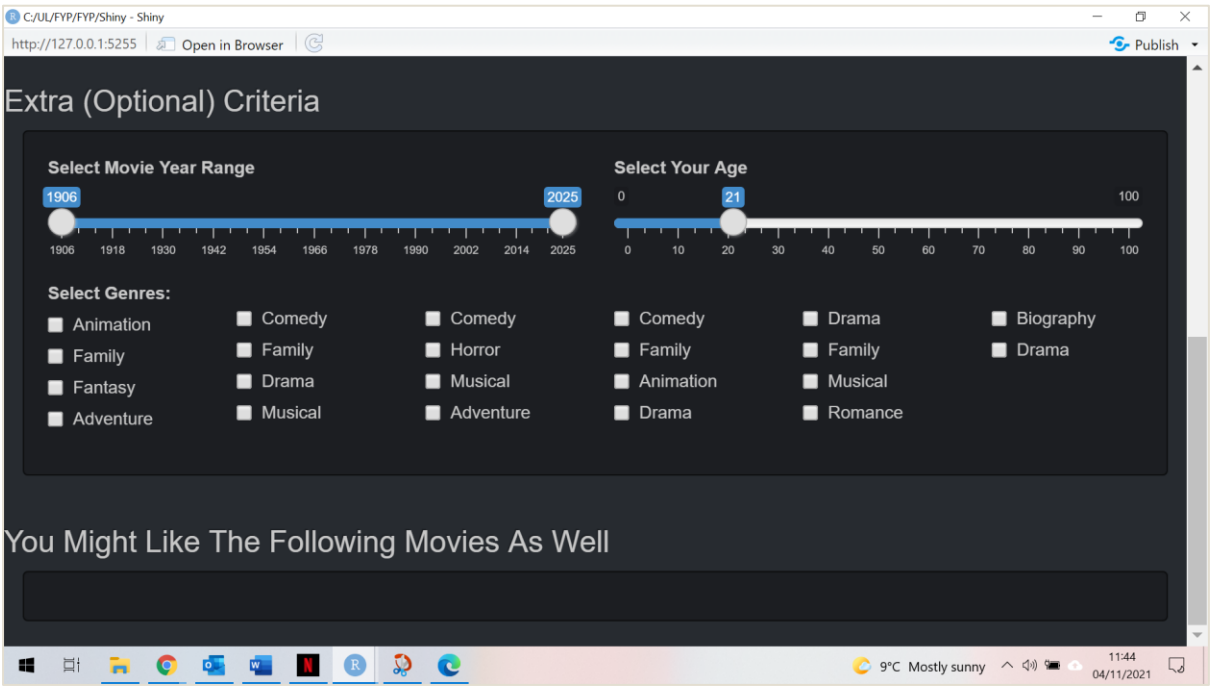


Figure 46: Adding Extra Criteria (Shubhansh2006, 2019)

Final Tab

Once I began creating my recommender system, I discovered that I needed explicit user ratings, and that my proposed design would not be feasible. As discussed later, I randomly generated 5 of the 25 top titles, and asked the user to rate each. Also, in case a user had not seen any of these movies, I made use of an `actionButton()` to allow a user to refresh the page and have new movies recommended:

Figure 47: Getting User Ratings

I also slightly altered my 'extra criteria' section, to make it more useful. I added in the functionality to select specific actors/directors, and rearranged the genres to be in alphabetical order. I initially had this page dynamically updating recommendations as the user changed their ratings/preferences, however this resulted in the app being very slow (as it was constantly updating), and so I also added in an `actionButton()` to tell the app to create recommendations:

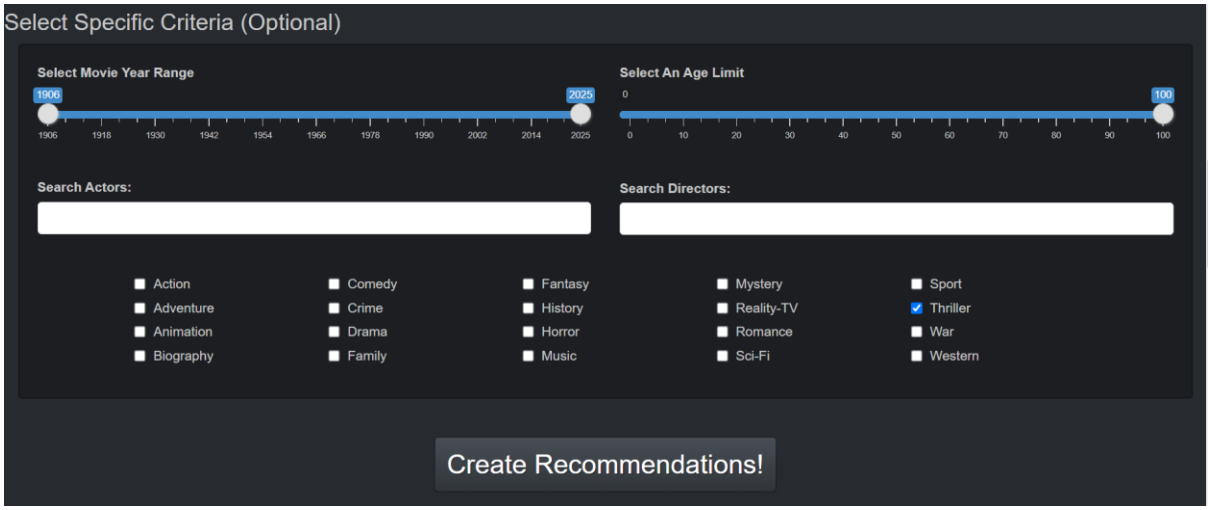


Figure 48: Updated Criteria Section

Finally, once I had recommendations being created, I had to decide how to format it to be visually appealing for the user. I thought of a few options, such as displaying the image for each movie, and upon a user clicking, it would redirect the user to the 'Search Movies' tab, open on that movie. I also considered just outputting titles, or even displaying the table I was using throughout my recommendation function.

I liked the idea of redirecting a user to the 'Search Movies' tab, however this proved difficult to do, and the recommendations tab would reset (losing the list of recommendations), so I did not think it was a workable solution. As a compromise, I re-used the 'Search Movies' code (modifying slightly) to display the movies in the same manner as the 'Search Movies' tab, without re-directing them.

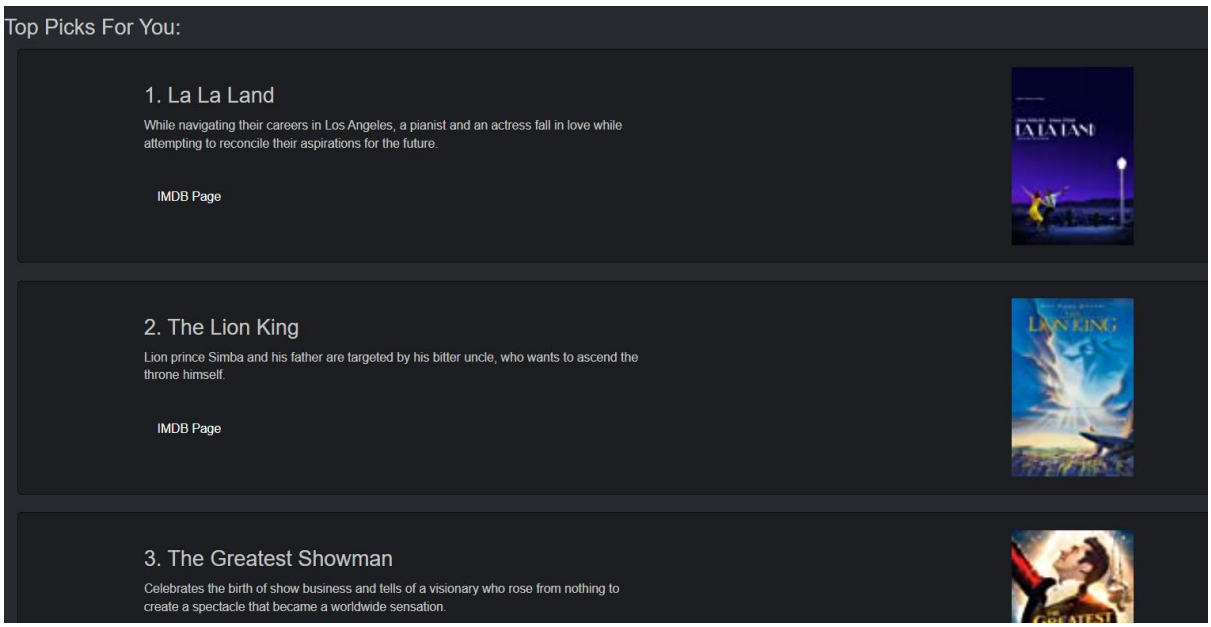


Figure 49: Outputting Recommendations

Recommender Systems

Background

We are all prone to 'choice paralysis'; an "*inability to choose anything at all because there are simply too many options with too many variations to sift through*" (Dee, 2021). Many of us can relate to scrolling endlessly through Netflix, looking for a movie that is just right, until we end up getting frustrated, leading us to give up entirely, choosing an old favourite, or randomly picking an option and hoping for the best. This is summarised by Hick's Law: "*the more choices a person is presented with, the longer the person will take to reach a decision*" (Foundation, 2022).

This introduces the necessity of recommender systems: a data-filtering tool that is "*a subclass of machine learning which generally [deals] with ranking or rating products / users… a recommender system is a system which predicts ratings a user might give to a specific item*" (Vatsal, 2021). As summarised here (Foundation, 2022), "*a recommender system expertly narrows the number of items a particular user could consider so that they can continue calmly browsing and eventually make the right choice, confidently select an item, and feel good about the user experience to the point that they feel like they could — or even want to — do it again*".

This benefits both the user and the company: if I am satisfied with my experience (and with my product) I am much more likely to return than if I am not (in turn benefiting the company), and I am also going to feel more relaxed and comfortable throughout the experience (benefiting me).

While there are many different applications of recommender systems, I am going to be using it to create personalised content. Sites such as Amazon, Netflix and YouTube use algorithms to track what we engage with, and how much we like it. In other words, they want to recommend, to a particular user, products catered to their specific preferences. This can be done:

1. **Explicitly** (Ahmad, 2021): where the user themselves specifies how much they liked a given item. Some examples of this are rating an Amazon product, or (dis)liking a YouTube video.
    a. This can lead to the 'cold-start problem' (Lendave, 2021); if a new – i.e., previously unseen - item is introduced after training has occurred, the model won't be able to include it in personalised recommendations before it has been rated by a large amount of users.
2. **Implicitly** (Ahmad, 2021): where preferences are inferred indirectly. Some examples of this would be watch-time of a video (if you continue watching you probably like it, whereas if you immediately click off you probably don't), how often you visit a product's site or adding an item to a Wishlist.

While these can be very costly and complex to create (Netflix created a challenge to engineer a recommender system superior to its own, with a prize of $1 million! (Rocca & Rocca, 2019)), creating a good recommender system can be invaluable for a company, for example 35% of Amazon's revenue is generated from their recommendation engines (Agrawal, 2021).

How to Create Recommendations

The basic premise of creating recommendations is finding products similar to products the user is deemed to like (via explicit or implicit feedback, as discussed above).

This can be determined using similarity-based metrics (generally via finding a correlation coefficient) or distance-based metrics (points close together when plotted are deemed similar). In general, the resulting similarity will be measured on the interval [0, 1], with a similarity of 1 being the same (i.e., comparing an object with itself) and 0 being radically different (equally, we can calculate a dissimilarity, where 0 means there are no differences – i.e., they are 'the same' – and 1 means they have nothing in common) (Polamuri, 2015).

This can be calculated using any data, which will fall under one of two different data types: numerical or categorical. Numerical data is "*expressed in terms of numbers rather than natural language descriptions*" (Formplus, 2021) (i.e., it is a number on either a discrete or continuous scale), and categorical data "*can be stored into groups or categories with the aid of names or labels*" (Formplus, 2021) (i.e., it is classed by words, such as a colour or type of car).

Calculating Distances

The method in which similarity is calculated is dependent on the types of data we are dealing with; all numerical, all categorical, or a mix of both. In my case, I ended up using all numerical data, and so I will discuss some of the most popular numerical options below.

Euclidean Distance (Liberti & Lavor, 2017)

Derived from Pythagoras' Theorem (Danka, 2021), the Euclidean distance is the length of the path connecting two points, and is probably the most well-known, easiest and most popular method of calculating distances. It has many desirable properties, such as being symmetrical, differentiable, and convex (Harmouch, 2021).

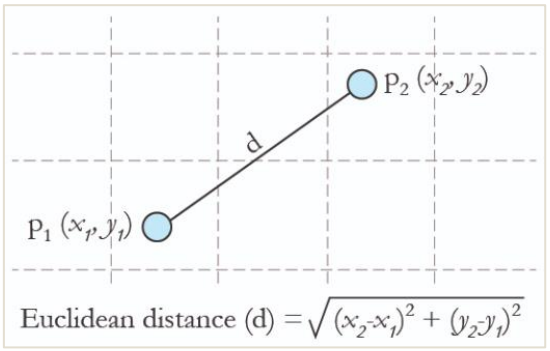


Figure 50: Euclidean Distance (TutorialExample, 2020)

Manhattan distance (Krause, 1987)

The Manhattan distance between 2 points is "*the sum of the absolute differences of their Cartesian coordinates*" (Polamuri, 2015). This is useful when we cannot connect the points 'in a straight line', such as if we were a taxicab weaving through city streets (hence the name and also nickname of Taxicab Distance) (Harmouch, 2021).

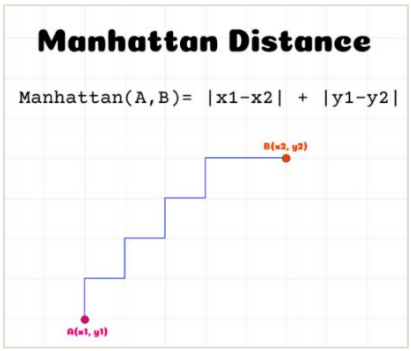


Figure 51: Manhattan Distance (Zornoza, 2020)

Maximum (Chebyshev) Distance (Cantrell, 2000)

The maximum distance can be found by calculating the "*maximum absolute value of the variations between the data samples' coordinates*" (Harmouch, 2021). This function is useful when aiming to find 2 points very far away from each other, for example one being a profit and the other a loss.

$$d(\underline{x}_i, \underline{x}_k) = \max_j |x_{ij} - x_{kj}| \qquad \text{Maximum}$$

Figure 52: Maximum Distance (Bargary, 2022)

Cosine Distance (Connor, 2016)

This is based on the cosine similarity, which is more a calculation of orientation, as opposed to magnitude. Using the cosine similarity, we can find where 2 points are relative to each other, by effectively finding the cosine of the angle between them. The cosine distance is then 1 – cosine similarity.

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$

Figure 53: Cosine Similarity (XILINX, 2022)

Recommendation Methods

While all recommender systems depend on finding similarities, it is not actually necessary to compare the products themselves, and so there are many different methods which describe *what* is being compared.

For example, if 2 users rate multiple products the same way, then it is likely they will agree on other products too (discussed further below). So, when making movie recommendations, I can make recommendations based on the *user* instead of the product (movie) itself.

I identified three main ways of creating comparisons/finding similarities:

1. **Comparing two users** by finding a user with preferences similar to the current user, and hence inferring that the current user may also like other movies this similar user liked. This is called *user-based collaborative filtering*.
2. **Comparing two movies** by finding movies similar to each other, and hence inferring that the user may also like movies similar to the one they previously liked. Movies can be deemed similar if they:
    i. Share attributes that I can determine (e.g., they share the same genres therefore they are probably alike). This is called *content-based filtering*.
    ii. Were rated similarly by the same user. This is called *item-based collaborative filtering*.

These will be discussed in more detail in the next section.

Collaborative Filtering (CF) (Hu, et al., 2008)

The basic concept of collaborative filtering is cited as "*[using] similarities between users and items simultaneously to provide recommendations*" (Course, 2021).

User-Based (Hu, et al., 2008)

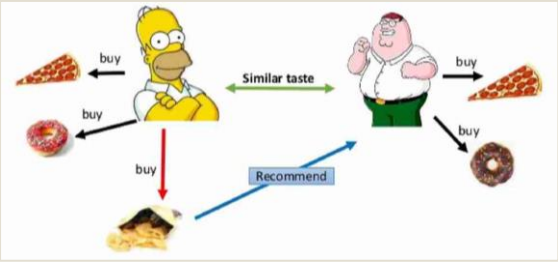


Figure 54: User-Based Collaborative Filtering (Iyer, 2020)

User-based CF occurs when, if two users give multiple items similar ratings, it is inferred that they are similar users (i.e., we assume that, given that the users agreed in the past, they will also continue to tend to agree in the future). This ignores other factors such as age or gender, it is merely based on what the user says they like. In other words, if user A is similar to another user B, then B's preferences can be used to predict what A may like.

Item-Based *(Hu, et al., 2008)*

Item-based CF occurs when, if two items receive similar ratings from the same user, they are said to be similar items. Content-based filtering (CBF) is similar to this in that it matches items, however CBF determines if items are similar using information about the items themselves, whereas item-based CF only depends on users' *opinions* of the item. If a user liked an item C, deemed similar to another item D, then item D would be recommended to the user.

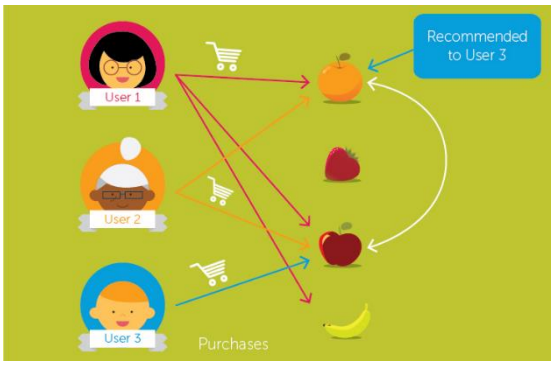


Figure 55: item-based CF (Wenig, 2019)

Content-Based Filtering (CBF) (Balabanovic & Shoham, 1997)

CBF does not rely on other users, merely *this* user's preferences, which are then compared with various aspects of the product. In other words, the products *themselves* are compared to find new products containing (similar) features of a previously liked product (Upwork, 2021).

Figure 56: Content-Based Filtering (Wenig, 2019)

Perhaps the biggest downside of Content-Based Filtering is that the model-creator must hand-pick which features are important (this can be done either by choosing which features are included or by designing a weighting system), leading to possible human oversight (Developers, 2018). Also, given that the model is entirely reliant on this user's data, if there isn't much user-data available, this can lead to limited recommendation powers (Developers, 2018).

Hybrid Recommender Systems (Burke, 2002)

While both collaborative and content-based filtering have their own strengths, a combination of both, in a hybrid recommender system, can harness the best of both together (and equally minimise their individual shortcomings). An example of this would be *"[generating] predictions separately and then combining the prediction*" (Verma, 2021) or equally just "*[adding] the capabilities of collaborative-based methods to a content-based approach*" (Verma, 2021).

Creating My Recommender System

After researching some of the possible methods and functions I could use to create recommendations, I began applying this knowledge to my dataset and began creating my own recommender system.

This involved a few key steps:

1. Choosing my recommendation method
2. Determining comparison criteria
3. Preparing comparison criteria
4. Implementing comparison criteria
5. Getting user ratings
6. Using comparison criteria to find similarities
7. Using similarities to create recommendations
8. Enforcing user constraints
9. Implementing recommendations
10. Outputting recommendations

Choosing My Recommendation Method

When creating my interim report, I proposed using collaborative filtering to create recommendations. Given that I have a lot of information about both the movies, and I also have user ratings, this seemed like a strong possibility for how I would recommend movies.

However, upon trying to implement this, I soon discovered that this wasn't a viable option:

- For user-based CF, I would need a large data set of user ratings (such as the MovieLens Dataset (GroupLens, 2019)), which contains many different users, each of whom have rated many movies. This way I could infer that if my user appears to rate movies similarly to a 'catalogued' user, they may like other movies this same user has liked.
- Similarly, item-based CF was not possible as I didn't know what ratings each individual user gave to movies (i.e., I could not say if movies were similar based off receiving similar user ratings).

This was not an ideal situation as there is *much* literature and examples available online for collaborative-filtering, and I have also used it before. I would have preferred to use CF or a hybrid system, because a content-based filtering system is entirely reliant on me choosing the 'correct' criteria, however I could not find another option given the dataset available to me.

I considered trying to create recommendations using the MovieLens data set (GroupLens, 2019), and from there merging with my scraped data, but this seemed quite complex and arguably unnecessary when I did have another viable option, so I decided to use a content-based approach. As a result, my recommendation function relies *purely* on the information I have about the movies, with user ratings being much less (if at all) important.

Had I been able to use collaborative filtering, creating recommendations would have been very quick and easy, as R has many built-in functions for creating recommendations in this manner. Despite extensive research, there is seemingly no content-based filtering equivalent, and so I coded my own recommender system from scratch, beginning with deciding how to find similarities.

Determining Comparison Criteria

The first step in trying to find similarities between movies is determining what criteria make two movies similar. Of my scraped pieces of information, the 'theme' of the information fell into a few different categories (with respect to its use as comparison criteria):

- **Ranking metric:** theoretically these could be used to determine similarities. However, using 'Rank' as an example, this would mean that movies with 'close' rankings would be deemed similar. I want to recommend the highest ranked of the most relevant movies to my user (i.e., if the user likes movie 9,999, but it and 10,000 are 'close together', I don't want to 'encourage' the algorithm to suggest the movie ranked 10,000/10,000), so I decided it was (potentially) more useful to help *rank* my recommendations, not to create them in the first place. The variables in my data that could be used for this include:
    - *Rank*
    - *Rating*
    - *Metascore*
    - *Votes*
    - *Gross Earnings*
- **Irrelevant:** For some variables (listed below), there was no useful or feasible way to utilise this information.
    - *Title*: a movie's name does not necessarily reflect its content.
    - *Runtime*: this did not strike me as criteria that would affect a user's enjoyment of a movie.
    - *ID*: this is IMDb's own ID for the movie, it does not 'mean anything'.
    - *Link*: this is the link to the movie's IMDb page, it does not 'mean anything'.
    - *Image Link*: this is the link to the movie's poster, it does not 'mean anything'.
- **Already accounted for:** Other variables (listed below) were already used as user constraints; therefore, they were not suitable for calculating similarities.
    - *Year*
    - *Age Rating*
- **Had time permitted:**
    - *Description*: I could have used Natural Language Processing (Chowdhary, 2020) to find, for example, similar 'buzz words' in descriptions, however this was not feasible within the scope of this project and so it was omitted.
    - *Directors*: initially, I tried to utilise this, however it proved too difficult to implement, so I excluded it.
        - I later also concluded that you may not actually like/know the director in question, and so it didn't make sense to include this criterion.
        - Furthermore, I also had already included this under user-constraints. As opposed to recommending movies with 'similar' directors (which may not be helpful), the user can manually specify a director they particularly like.
    - *Actors*: same as directors.
- **Useful:**
    - *Genre*: based upon the information I had, this was the most logical information to use in order to deem 2 movies similar. This did pose issues, as it is categorical data, but this will be discussed below.

Preparing Comparison Criteria

After identifying genres as my comparison criteria, I initially began by trying to find similarities without changing my categorical data to numerical. However, because my genre column was comprised of Strings, the whole String was deemed a genre (i.e., the first genre is "Action,Adventure,Comedy", not "Action", "Adventure", and "Comedy"). One solution was to split the whole dataset up by genre, however, this made the dataframe huge (22,212 x 16 since each movie tends to have 2-3 genres). This approach then couldn't be used as I would run out of disk space.

To continue using Genre as a comparison criterion, I tried to think how to represent genres as numbers. I decided that converting from one genre column (where each row contained multiple genres), to many columns of each genre type (using 1s and 0s to represent true or false, i.e., if a row is of a certain column type, it has 1 in that cell, otherwise 0), made the most sense:

Action	Adventure	Animation	Biography	Comedy	Crime	Drama	Family	Fantasy	History	Horror
0	0	1	0	0	0	0	1	1	0	0
0	1	0	0	1	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0	0	1
0	1	0	0	1	0	0	1	0	0	0
0	0	1	0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0

Figure 57: Binary Genre Columns

Now my genres were in usable format and, as an added bonus, numeric, which would make distance-measuring much easier.

Implementing Comparison Criteria

I initially began using the Euclidean distance to measure how similar my movies were. However, I realised that this (and other metrics previously discussed) wasn't the best metric to use. Any such metrics would compute both rows as having a zero in the same column as similar, whereas in fact only two rows with a *one* in the same column is relevant. In other words, two movies not being an action movie does not imply that they're similar, but two movies that *are* action movies does.

Given that my data is only comprised of 1s and 0s (i.e., it is a binary system where 1 = true and 0 = false), I had to re-evaluate my comparison metric. As a result, I began researching binary similarity metrics, and chose to use the Jaccard Similarity (Tan, et al., 2005).

The Jaccard Similarity is popular for binary data ("*where 0 means that the attribute is absent, and 1 means that it is present*" (Karabiber, 2022)) because it takes into account that only attributes that *do* appear are important. It is calculated by finding the size of the intersection of two sets (i.e., items present in both), and dividing by the size of the union (i.e., total number of items present between the 2 sets) (Karabiber, 2022):

Figure 58: Jaccard Similarity (Patriarchy, 2020)

This could be calculated (at least) 3 ways in R:

1. Using the vegan package and the following function: `vegdist(df, method = "jaccard")` (finnstats, 2021).
2. Using the proxy library and the following command: `proxy::dist(df, method = "Jaccard")` (StackOverflow, 2017).
3. Using the dist function with method set to "binary": `dist(df, method = "binary")` (RDocumentation, 2022).

I implemented each of these in my code, and they all produced the same results. I chose to use the `vegdist` function (RDocumentation, 2017), as it had a built-in parameter to only include the upper diagonal. This was very useful, as a 10,000 x 10,000 matrix uses a lot of storage.

Getting User Ratings

To be able to predict what a user may like, I began by identifying what they have previously liked. As there was no option for implicit ratings, I needed the user themselves to input their (dis)likes.

I considered a few options to determine their preferences, such as having them manually select their favourite movies from a drop-down list (as in my interim report), or asking them to rank 5 pre-determined movies (for example the top 5 most popular). However, a common issue between each idea was that each method would yield the same results every time. Their favourite movies were unlikely to change, and their ratings for the 5 movies were also likely to remain the same.

As a result, I decided to create a dynamic rating-system, where each time a user used the app, they would rank different movies, and as a result get different recommendations. I initially considered randomly generating 5 movies each time, but as so many movies are very old/not well-known, this did not make sense as the user might not have seen any of them.

To summarise, I needed a list of movies that would change upon each use, and that would ask for ratings of movies a user was likely to have seen. To achieve this, I decided to have the user rate randomly generate 5 titles from the 25 most popular (at time of scraping) each time:

```
# Subset to top 25 titles
ratable_movies <- movies[1:25, ]
ratable_titles <- ratable_movies[sample(nrow(ratable_movies)), "Title"]

# Get each title separately
for(n in 1:5){
  assign(paste("rate_movie", toString(n), sep = "_"),
         ratable_titles[n, ] %>% pull()
  )
}

# Output each title for user to rank
output$movie1 <- renderText({rate_movie_1})
output$movie2 <- renderText({rate_movie_2})
output$movie3 <- renderText({rate_movie_3})
output$movie4 <- renderText({rate_movie_4})
output$movie5 <- renderText({rate_movie_5})
```

Figure 59: Outputting the Top 25 Movies

Having user ratings for 5 movies, I could then begin finding movies similar to the ones they liked, and from there create recommendations.

Using Comparison Criteria to Find Similarities

To make my dissimilarity matrix usable (in an acceptable run-time), the columns of my dissimilarity matrix were filtered to only include the 5 rated titles (row names in the green box being the rank of each of the 10,000 movies):

	The Greatest Showman	Rocketman	Annette	Dear Evan Hansen	Vivo
1	1.0000000	1.0000000	1.0000000	1.0000000	0.8000000
2	1.0000000	1.0000000	1.0000000	1.0000000	0.5000000
3	0.0000000	0.5000000	0.5000000	0.6666667	1.0000000
4	1.0000000	1.0000000	1.0000000	1.0000000	0.7500000
5	1.0000000	1.0000000	1.0000000	1.0000000	0.5000000

Figure 60: Dissimilarity Matrix with Column Names

From there, I took each rated movie, and reordered the movies in order of similarity (i.e., a movie most similar – dissimilarity of 0 - to *The Greatest Showman* would appear at the top of its vector). Note that the row names (green box) are now no longer in numeric order, as movie ranked third is most similar to *The Greatest Showman*, followed by the 115th movie, and so on:

	Dissimilarity
3	0
115	0
156	0
277	0
350	0

Figure 61: Reordered Dissimilarity Vector

To make this more readable, I reordered the dataset to the designated order above (note that the ranks in figure 61 below match the row names above in figure 60 – although Rank 3 is missing as that is *The Greatest Showman* itself). I also introduced a similarity column (being '*1 – dissimilarity*', so that 1 now means 2 movies are identical), and an 'original movie rating' column (i.e., the rating the movie it's similar to got):

Rank	Year	Title	Genre	Rating	Votes	Metascore	GrossEarning	Similarity	OriginalMovieRating
115	2021	Music	Drama,Musical	3.1	7118	23	*NA*	1	5
156	1975	Tommy	Drama,Musical	6.6	20208	66	34.25	1	5
277	2004	Swades: We, the People	Drama,Musical	8.2	87260	*NA*	1.22	1	5
350	2018	Everybody's Talking About Jamie	Drama,Musical	7.8	518	*NA*	*NA*	1	5

Figure 62: Dataset Ordered by Similarity

Note that *The Greatest Showman*'s listed genres are drama and musical, and each of the recommended movies have the same genres.

After looping through each of the rated movies, I had a dataframe representing the most similar movies and was ready to utilise this to create recommendations.

Using Similarities to Create Recommendations

This was remarkably challenging to do: I didn't have a function to create recommendations, and so I had to come up with my own ranking criteria to decide the order in which they would be recommended.

Fortunately, while trying to find comparison criteria previously, I had already found multiple columns which could be used to create rankings, namely: Rank, Rating, Metascore, Votes, and Gross Earnings. My next step was to decide what combination or weighting of these seemed appropriate to find a 'best recommendation'.

I decided to create a weighting criterion using my selected ranking columns but make them proportional to how highly weighted the ranking was. This meant that a movie ranked 1 would be unlikely (but not impossible) to influence my top 5 recommendations, but also a movie rated 5/5 wouldn't (necessarily) dominate, as (for example) a movie rated 4/5, that has a 'really popular' similar movie could rank ahead of an 'unpopular' movie deemed the same as a 5/5 rated movie.

Looking at my 'ranking columns' above, I decided on the following method:

- Rank is only proportional to what is the most popular at the instance in time the movie was scraped. It changes daily, and so is not 'concrete' enough to feature in a weighting, nor is it a fair reflection of 'overall most popular' (just most popular *today*), so I discounted it.
- User ratings are certainly useful to determine a general consumer consensus, however without taking into account how many people voted (e.g., 5 'die hard' fans giving it 10/10 and no one else ranking it does not make its 10/10 very fair), it could be very skewed.
    - As a result, I multiplied the user rating by the number of votes, and divided by the average number of votes. This way, only a movie voted on by a 'reliable' number of people would make much impact.
- Metascore represents the critics' rating, and so was an important figure to use. Similarly to above, it was more important how highly it ranked *compared to others*, than how it ranked individually.
    - Again, I divided Metascore by the mean result, so that particularly highly rated scores would have a bigger impact on weighting.
- Continuing with this trend, I divided Gross Earnings by the mean earnings.
- Summing these 3 pieces of information, I had my basis for weightings.

As discussed earlier, however, I wanted this to be proportional to 1) how similar a given movie is to the rated movie and 2) how highly rated the individual movie was. As a result, I decided to multiply my weighting sum by similarity*rating (i.e., a movie 'the same' as a movie rated 5 would get a multiplier of 1*5 = 5, a movie entirely different would get 0*5 = 0, and so on).

This resulted in the following weighting formula:

```
similar_movies$WeightedScore = (similar_movies$Similarity*similar_movies$OriginalMovieRating)*
  ((similar_movies$Rating*similar_movies$Votes/round(mean(movies$Votes, na.rm = TRUE))) +
     similar_movies$Metascore/round(mean(movies$Metascore, na.rm = TRUE)) +
     similar_movies$GrossEarning/round(mean(movies$GrossEarning, na.rm = TRUE)))
```

Figure 63: Weighting Formula

Enforcing User Constraints

Given that a user can select a year range for the movie's release date, an age field (in order to generate age-appropriate movies), and any specific genres/actors/directors they want, I needed to create a function that ensured this criterion would be satisfied.

I began by checking if any specific genres/actors/directors had been selected. If not, that constraint was set to include all genres/actors/directors. Once I had a (not empty) list, I subsetted the data to only include movies listing those selected. I did this step first as it would hugely cut down the size of my returned data, and hence would make subsequent subsetting much more efficient.

```
if(is.null(genres)){
  genres <- levels(genre_list)
}
movies <- movies %>%
  subset(movies$Genre %>% grepl(paste(genres, collapse="|"), .))
```

Figure 64: Checking Selected Genres

Next, I imposed the year-range constraint. By default, it is set to the oldest to newest date (so all movies would be included), changing if the user chooses to. This was easily enforced, ensuring the oldest movie was only as old as the lower bound, and as new as the upper.

```
# Year filter
movies <- movies %>% filter(Year >= year1, Year <= year2)
```

Figure 65: Movie Year Filter

Finally, I created my age filter (again defaulted to the max age of 100 so nothing would be unfairly eliminated). I started by checking the different age categories in my table:

```
> movies %>% group_by(AgeRatingUS) %>% count()
# A tibble: 14 x 2
# Groups:   AgeRatingUS [14]
   AgeRatingUS     n
   <chr>       <int>
 1 (Banned)        3
 2 12              7
 3 12A            27
 4 12PG            1
 5 15             17
 6 15A            10
 7 16              6
 8 18              3
 9 Approved        2
10 G             133
11 PG            121
12 PG-13          34
13 R              28
14 NA           9608
```

Figure 66: Movie Age Rating Filter

From there, I had to convert all the non-numerical ratings to numbers in order to be able to enforce a maximum allowed.

```
movies <- movies %>%
  mutate(AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "PG", 0),
         AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "Approved", 0),
         AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "G", 0),
         AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "R", 18),
         AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "(Banned)", 18),
         AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "PG-13", 13))
```

Figure 67: Converting Ratings to Numbers

Once all my ages were numbers, it was very easy to ensure they were all less than my user's age, the resulting data was now complying with all user constraints, and was ready for use in making movie recommendations.

Issues Subsetting Genres/Actors/Directors

To test the process, I ran each chunk of code multiple times with multiple different versions of each constraint. All combinations tested subsetted my data as expected, so I incorporated the code into my app:

```
poss_movies <- eventReactive(input$updateButton, {movie_subset(movies, input$year[1],
                                                               input$year[2], input$age,
                                                               c(input$genre1,input$genre2,
                                                                 input$genre3,input$genre4,
                                                                 input$genre5),
                                                               input$actors, input$directors)}
)
```

Figure 68: Incorporating Recommendations into the App

Initially it all seemed to work perfectly. Using Hugh Jackman as an example, upon searching his name, his 3 movies would display as expected:

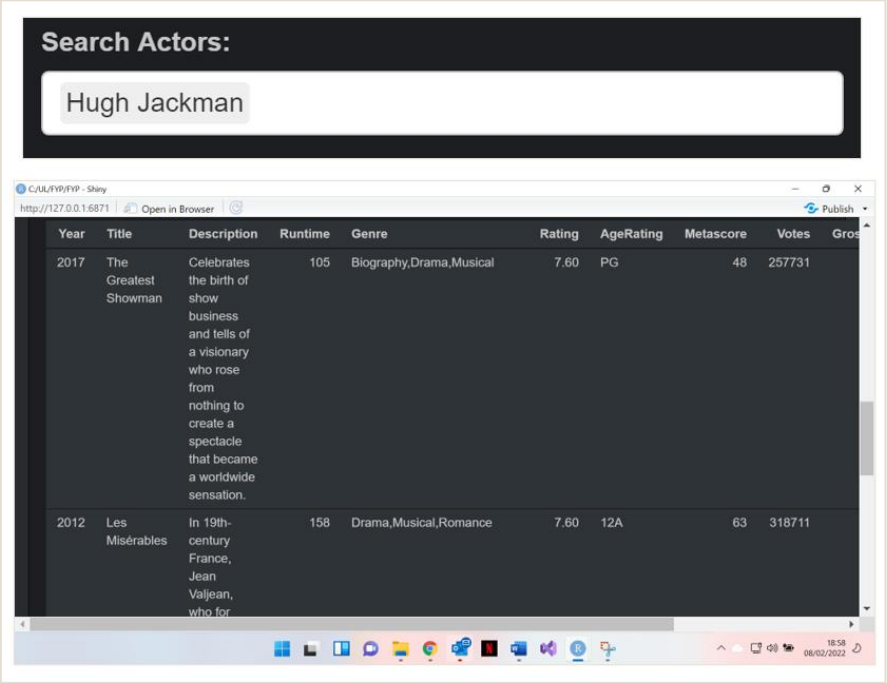


Figure 69: Hugh Jackman's Movies

But for other actors (e.g., Amrish Puri), no movies would show up in the app as shown below:

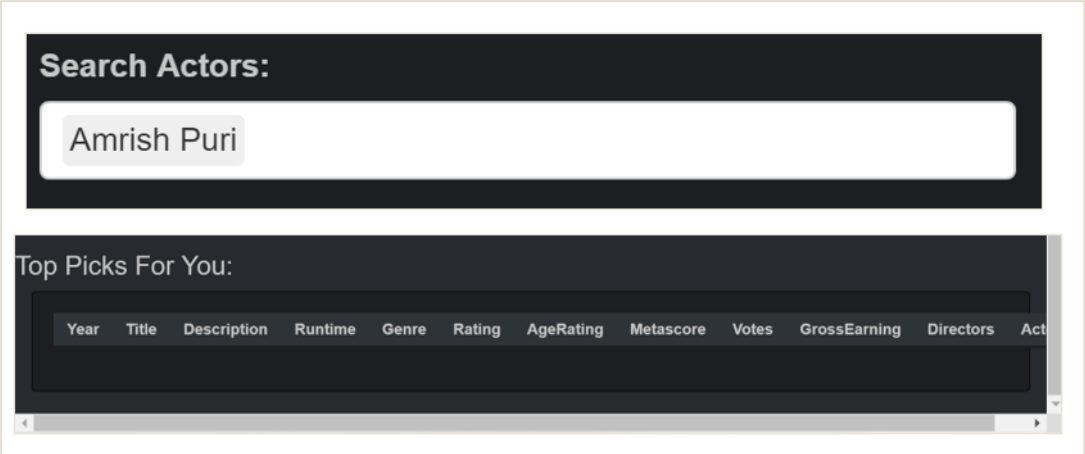


Figure 70: Amrish Puri Error

But his (8) movies would display when manually running the code:

```
> actors = "Amrish Puri"
> movies %>%
+     subset(movies$Actors %>% grepl(paste(actors, collapse="|"), .))
# A tibble: 8 x 16
   Rank  Year Title    Description      Runtime Genre  Rating AgeRatingUS Metascore Votes GrossEarning
  <int> <int> <chr>    <chr>              <int> <chr>   <dbl> <chr>           <int> <int>        <dbl>
1   404  2000 Shaheed~ Udham Singh was~     165 Biogr~    7   NA                 NA   200         NA
2   670  2001 Yaadein~ Raj Singh Puri ~     171 Music~    4.4 NA                 NA  3397          0.96
3   793  1987 Mr. Ind~ A poor but big-~     179 Actio~    7.8 NA                 NA 14857         NA
4   857  1997 Pardes   Kishorilal want~     191 Drama~    7   NA                 NA 14349         NA
5   868  1997 Koyla    A village girl ~     166 Actio~    6.2 NA                 NA  8671         NA
6  1749  2001 Chori C~ A prostitute ag~     165 Comed~    5.4 NA                 NA  4604         NA
7  2341  1990 Kondave~ Unable to toler~     151 Actio~    7   NA                 NA   313         NA
8  7886  1979 Iqraar   Add a Plot           NA Music~    5.3 NA                 NA     8         NA
# ... with 5 more variables: Directors <chr>, Actors <chr>, ID <chr>, Link <chr>, ImageLink <chr>
```

Figure 71: Manually Running Amrish Puri Code

I began to debug the process and noticed that for certain directors and genres, I would also get no results. I had a few hypotheses, such as:

- Movies further down in my list (e.g., movie 5,000+) might be excluded in the search as it would time out. While this did appear to be true 'overall' (i.e., movies lower down did tend to be an issue – but not all), enough worked in the last few rows that it clearly wasn't a 'location' issue.
- Actors/directors not listed first might somehow be excluded.
    - This did not turn out to be correct, but it did lead me to noticing an error in my string-splitting. I had split by "," (same as genre – with no space before or after the comma), but I needed to separate by ", " (with a space to the right of the comma).

On closer inspection, I noticed that Amrish's movies contained a lot of *NA*s, while Hugh's movies had none. Looking at the *NA* columns in question, I was using AgeRating as a constraint, so I revisited that block of code.

To be able to enforce age restrictions, I had to covert the words into numbers. I had accounted for *NA*s by telling the code to exclude them, so I initially thought the *NA*s were a coincidence.

```
movies <- movies %>% filter(!is.na(AgeRatingUS)) %>%
  mutate(AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "PG", 0),
         AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "Approved", 0),
         AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "G", 0),
         AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "R", 18),
         AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "(Banned)", 18),
         AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "PG-13", 13)) %>%
  mutate(AgeRatingUS = parse_number(AgeRatingUS))
```

Figure 72: Checking NAs

But, *NA* is not considered a number, and so when parsing as a number in the final line, I was in fact deleting all of them, and reducing my list of movies from 10,000 to only 392!

movies	10000 obs. of 16 variables
movies_age_filter	392 obs. of 16 variables

Figure 73: NA Dataset Reduction

This immediately explained why so many movies were not showing up for me, and furthermore why 'lower down' movies tended to perform less well; their information is less well filled in and so most of the ages were presumably *NA*s.

As I couldn't guess what level the *NA*s were supposed to be, I set them to the age of "PG" (that way I'd never recommend something too mature, only potentially restrict to 'too young'). I could also now get rid of the 'exclude *NA*s' filter as they had all been replaced.

```
movies$AgeRatingUS[is.na(movies$AgeRatingUS)] <- "PG"
# movies %>% group_by(AgeRatingUS) %>% count() # check available levels
movies <- movies %>%
  mutate(AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "PG", 0),
         AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "Approved", 0),
         AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "G", 0),
         AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "R", 18),
         AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "(Banned)", 18),
         AgeRatingUS = replace(AgeRatingUS, AgeRatingUS == "PG-13", 13)) %>%
  mutate(AgeRatingUS = parse_number(AgeRatingUS))
```

Figure 74: Setting NA to PG

As soon as I changed this, the app immediately ran as expected and I didn't encounter any further subsetting issues.

Implementing User Constraints

Just prior to creating the final recommendations, I called my function to subset the movies to only include titles satisfying the user-inputted criteria, and hence subset my recommendation matrix to only include those titles.

```
poss_movies <- movie_subset(movies, input$year[1],
                            input$year[2], input$age,
                            c(input$genre1,input$genre2,
                              input$genre3,input$genre4,
                              input$genre5),
                            input$actors, input$directors)

recommender_system(movies,
                   poss_movies,
                   user_ratings,
                   genre_list,
                   dissimilarity)
```

```
# Subset to include any user constraints
poss_titles = poss_movies %>% select("Title") %>% pull()
recommendations <- recommendations %>%
  subset(Title %in% poss_titles)
```

Figure 75: Enforcing User Constraints

Implementing Recommendations

Once I had this compiled my weighted ratings for each movie, and enforced my user constraints, I simply removed the previously rated movies, ordered the dataframe to have highest (weighted) rated movies on top, and took the top 5 movies, to create my recommendations:

```
recommendations <- recommendations %>%
  subset(!Title %in% names_order[, 1]) %>% # take out the rated movies
  filter(duplicated(Title) == FALSE) %>% # only one instance of each row
  arrange(desc(WeightedScore)) %>% # highest rated on top
  slice(1:5) %>% # take the top 5
```

Figure 76: Finalising Recommendations

I ensured that the code worked as expected and that the recommendations seemed reasonable, and then implemented the code into my app's server.

In R, if a variable's value depends on user inputs (for me, both my list of possible titles and user ratings depended on information the user themselves selects), it is deemed a *reactive variable* (Datacamp, 2022).

Reactive variables do no behave in the same manner as a regular variable; they can only be used with in a 'reactive context' (this is easy to deal with, any expressions using a reactive merely need to be within a `reactive({ })` wrapper, to tell R that the values inside the curly braces may change), and they are actually more similar to a function than a variable and, as a result cannot be treated as a regular variable.

Prior to this project, I had never used Shiny, and so never encountered reactives before, which required a steep learning curve, as I struggled to find much or any examples or documentation on reactive variables. Some of the biggest issues I ran into were:

1. ***Subsetting data***: in order to subset to a specific row (r) and column (c), the normal code I would be is: `newDataframe = oldDataframe[r, c]` (i.e., specifying the desired row and column within square brackets). However, square brackets are not allowed to be used with a reactive variable, which made subsetting a lot more difficult (and while creating recommendations, a *lot* of subsetting was necessary). I had to use the `select()` command to subset columns, and `subset()` command to subset rows.
    a. This required much more work than merely using square brackets. In particular, subsetting rows was a major challenge, as subset requires logical criteria to decide which rows to pick (i.e., rows satisfying a Boolean condition will be selected), as opposed to just using an index. I couldn't just say '*select row r*', I had to say, '*select the row where the rank is equal to r*' (note that this only works when (a) I have a 'rank' column and (b) the rank column is in descending order).
    b. In effect, instead of just specifying `dataframe[r, n]`, I would have to write: `dataframe %>% subset(Rank = r) %>% select(n)`.
    c. Given that subset requires a logical condition, it has to go first. It happened many times that I would put the select function first (subsetting columns), but then subset would not work as the column I was using for my criteria no longer existed! For example, if I had not selected Rank as one of the columns to keep, it wouldn't be possible to use it to subset, as it was no longer part of the dataframe.
    d. Aside from taking more time and effort, it was actually very difficult to go back through my whole function and replace every subsetting instance (of which there were *many*), resulting in me having to run it line by line (I had to run the app itself, as otherwise my parameters were not reactives, and so would not necessarily show reactive errors), to ensure that each line was 'allowed'.
2. ***Slicing data***: when trying to only take the top 5 rows of a dataframe, I would traditionally write `dataframe %>% slice(1:5)`. However, when dealing with reactives, this function is not allowed. This resulted in a lot of reordering code (i.e., slicing *before* utilising a reactive)/finding workarounds (such as using `subset()` or otherwise) to subset data.
3. ***Utilising reactives***: when dealing with a regular variable, I can just call it by its name (e.g., if my dataframe is called 'df', when I need to use it, I would say df, followed by whatever command I need). However, something I didn't realise for a while, was that (if my dataframe was a reactive) I would have to call `df()` – with brackets after the name. I often forgot the brackets, and then would be very confused why it wouldn't run, but after some practice (and much app-crashing) it was manageable.

Once I had fixed these issues, the app would run, and I was finally creating dynamic recommendations.

Outputting Recommendations

Initially, I was merely checking to see what (if any) output there was, and if it made sense, so I would just print out my recommendation table (with no formatting or style):

Rank	Year	Title	Genre	Rating	Votes	Metascore	GrossEarning	Similarity	OriginalMovieRating	WeightedScore
1	2005	Charlie and the Chocolate Factory	Adventure,Comedy,Family	6.60	449209	72	206.40	1.00	4.00	3211.67
2	2013	Frozen	Animation,Adventure,Comedy	7.40	593891	75	400.70	0.50	4.00	2385.44
3	2019	Aladdin	Adventure,Comedy,Family	6.90	246043	53	355.50	1.00	4.00	1870.48
4	1994	The Lion King	Animation,Adventure,Drama	8.50	987916	88	422.70	0.20	4.00	1813.59
5	2010	Tangled	Animation,Adventure,Comedy	7.70	424806	71	200.80	0.50	4.00	1769.68

Figure 77: Recommendation Table

However, this included much information that would not be of interest to the user (and didn't look very nice), so I updated the look to be more visually appealing. Fortunately, I could recycle my code from the 'Search Movies' tab, so this was very quick and easy:

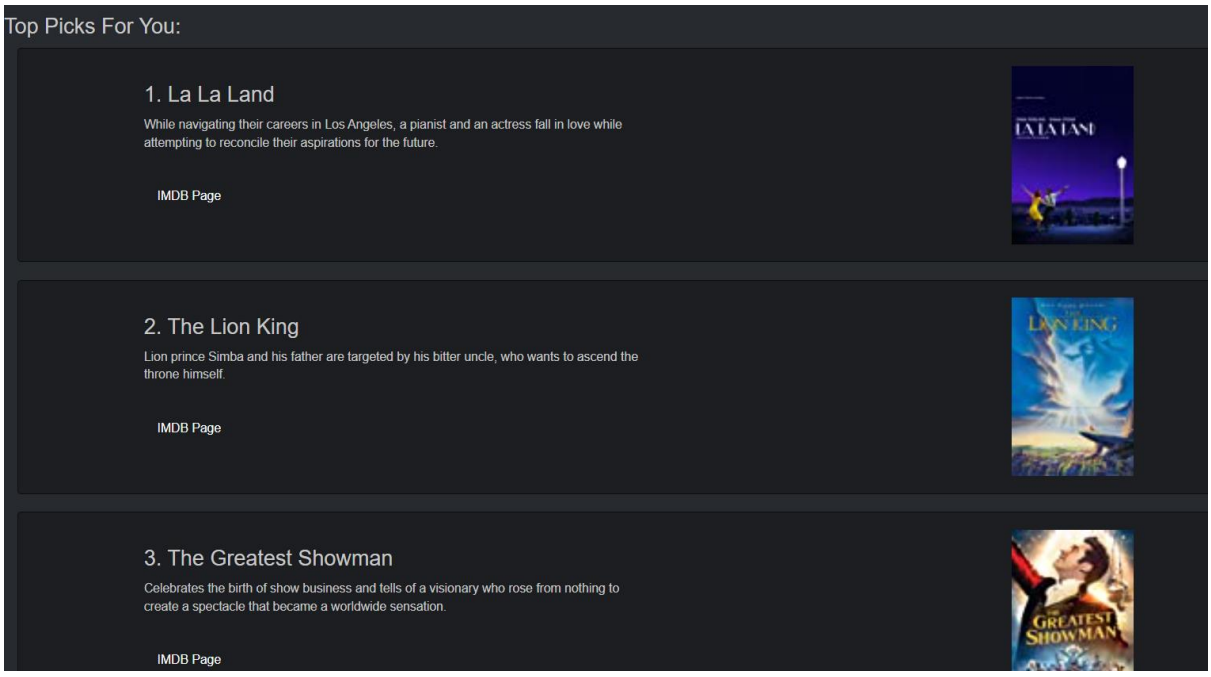


Figure 78: Outputting Recommendations

Conclusion

Overall, I am really pleased with my final app, and I have published both my R code and also the app itself online, with my R code (Smyth, 2022) and my app (Smyth, 2022) available online.

Unfortunately, due to my app needing so much memory (my dissimilarity matrix alone is 10,000 x 10,000), the app is too big for the shiny.io server (RStudio, 2022) (on a basic – free – plan), and so is not 'open to the public' (in fact, it actually shows an error if you try to access it, the error being that it's out of memory), however as the code is available online, a user can download and hence run it themselves, and so a READ ME file has been included to instruct a user how to run the code.

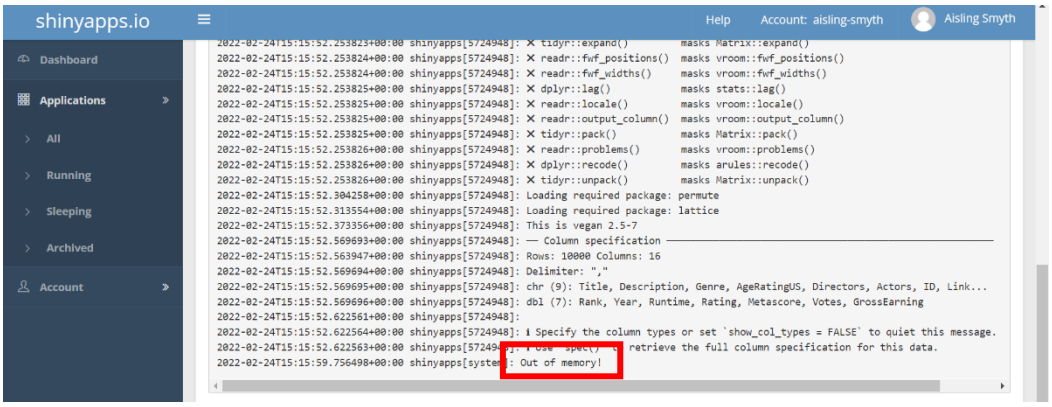


Figure 79: Out of Memory Error

I learned a lot throughout the course of this project: when I started, I had very limited R experience, I had never even heard of Shiny or data-scraping, and I had never created my own recommendation function from scratch. This process was a very steep learning curve, and required a lot of re-running, debugging, and Stack Overflow (StackOverflow, 2022), however I am now very confident in each aspect of my project.

At my interim report, I had scraped my data and created the 'skeleton' of my app. At that point in time, I planned to create my recommender function (which I thought would be very easy, as I thought at the time that I would be able to use built-in functions) and to re-scrape the data (which again, did not sound overly challenging as I only needed the genres and images). It later transpired that re-scraping was not possible within the timeframe I had, and creating recommendations was a very time-consuming and difficult process, given that I had to entirely devise it myself.

If I had more time, or if I were to revisit this project at a later point, there are a few things I would change:

- I would re-scrape the data, as having more genres per movies would make my recommendations better, and clearer images would make it more visually appealing.
- I would incorporate the MovieLens (GroupLens, 2019) dataset to aid in prediction-making. While my recommendations do seem reasonable (from my own knowledge of the musicals, they are quite similar in my opinion), using already existing prediction functions seems more reliable. If this was not possible, I would extend my function to incorporate Natural Language Processing (Chowdhary, 2020) to incorporate descriptions as part of my criteria. I would also potentially include actors/directors in my comparison criteria.
- Starting into semester 2, I had considered incorporating a 'data analysis' tab, where users could see statistics and plots of the data.

In conclusion, this project was very challenging, but it was incredibly stimulating, interesting and a lot of fun. I am really proud of my finished product and of everything I have learned over the months, and as a result, could well see myself continuing to do work of this manner in the future.

Acknowledgements

First and foremost, I must thank my incredible supervisor, Professor Norma Bargary. Throughout my whole time in college, you have been such a guiding light and support, and I am so incredibly lucky to have had you as my mentor and supervisor this year. Your never-ending support, guidance and help kept me going throughout a difficult few months, and you have no idea how much I appreciate it all. You have had such a positive impact on my entire college experience, and I am so, so grateful to you for this.

Next, I would like to thank my second reader, Dr David O'Sullivan. Your kindness, flexibility and willingness to accommodate me when I was too sick to present my interim report really helped me to just concentrate and improve. Without this, I don't know if I would have been able to sit any of my Christmas exams, so a huge thank you for this. As well as this, your questions and advice throughout my interim report were really insightful, thought-provoking and interesting, and helped give me a new perspective for my final semester.

As always, I would not have managed to complete this project without my family's unwavering support. From helping me come up with my initial idea, to listening to my 'way over your head' tangents on scraping and recommender systems, to merely clapping and saying how cool my app was, you were (as always) there for me every step of the way, and I really appreciate it. I am so lucky to have such an incredible family there for me every step of the way, and so grateful for all of you.

Finally, I would like to thank my best friends, Andrew, Ruth, and Arowa, and Kieran for the fun and lightness you brought to my years in UL. Be it getting food together, trying (and failing) to find a Netflix show to watch (how ironic!), or 'just' sitting on the couch and laughing with you after a day's work, being lucky enough to have you in my life these past few years has been such a blessing. Thank you for being such good friends to me, and for all the amazing memories you've given me.

Figure 80: Thank You! (Motorsport, 2015)

Bibliography

Agrawal, S. K., 2021. *Recommendation System -Understanding The Basic Concepts.* [Online]
Available at: https://www.analyticsvidhya.com/blog/2021/07/recommendation-system-understanding-the-basic-concepts/
[Accessed February 2022].

Ahmad, Z., 2021. *Recommender Systems: Explicit Feedback, Implicit Feedback and Hybrid Feedback.* [Online]
Available at: https://medium.com/analytics-vidhya/recommender-systems-explicit-feedback-implicit-feedback-and-hybrid-feedback-ddd1b2cdb3b#:~:text=To%20collect%20explicit%20feedback%20from,provide%20their%20ratings%20for%20items.&text=Even%20though%20this%20allows%20the
[Accessed February 2022].

awesomedata, 2021. *Github - 'Awesome Public Datasets'.* [Online]
Available at: https://github.com/awesomedata/awesome-public-datasets
[Accessed September 2021].

Balabanovic, M. & Shoham, Y., 1997. *Fab: Content-based, Collaborative Recommendation.* s.l.:s.n.

Bargary, P. N., 2022. *MA4128/MS6022 Hierarchical Clustering Handout.* s.l.:s.n.

Burke, R., 2002. *Hybrid Recommender Systems: Survey and Experiments.* s.l.:s.n.

Cantrell, C. D., 2000. *Modern Mathematical Methods for Physicists and Engineers.* s.l.:Cambridge, UK ; New York : Cambridge University Press.

Chowdhary, K. R., 2020. *Fundamentals of Artificial Intelligence.* s.l.:Springer, New Delhi.

Churchville, F., 2021. *Definition: user interface (UI).* [Online]
Available at: https://www.techtarget.com/searchapparchitecture/definition/user-interface-UI#:~:text=The%20user%20interface%20(UI)%20is,an%20application%20or%20a%20website.
[Accessed February 2022].

CloudFlare, 2021. *What is data scraping?.* [Online]
Available at: https://www.cloudflare.com/en-gb/learning/bots/what-is-data-scraping/
[Accessed September 2021].

Connor, R., 2016. *A Tale of Four Metrics.* s.l.:Springer, Cham.

Course, M. L. C., 2021. *Collaborative Filtering and Matrix Factorisation - Basics.* [Online]
Available at: https://developers.google.com/machine-learning/recommendation/collaborative/basics
[Accessed October 2021].

Danka, T., 2021. *The Euclidean Distance Is Just the Pythagorean Theorem.* [Online]
Available at: https://towardsdatascience.com/the-euclidean-distance-is-just-the-pythagorean-theorem-2e672017d875
[Accessed February 2022].

Darwish, N., 2013. *CSS Box Model and Positioning.* [Online]
Available at:

https://www.codeproject.com/Articles/567385/CSSplusBoxplusModelplusandplusPositioning
[Accessed March 2022].

Datacamp, 2022. *Reactivity: simple reactive variable.* [Online]
Available at: https://campus.datacamp.com/courses/case-studies-building-web-applications-with-shiny-in-r/shiny-review?ex=10
[Accessed February 2022].

DataFlair, 2022. *Commonly Used HTML Tags with Examples.* [Online]
Available at: https://data-flair.training/blogs/html-tags-with-examples/
[Accessed February 2022].

Dee, C., 2021. *What is a recommender system (or recommendation engine)?.* [Online]
Available at: https://www.algolia.com/blog/product/what-is-a-recommender-system-recommendation-engine/
[Accessed February 2022].

Developers, G., 2018. *Content-based Filtering Advantages & Disadvantages.* [Online]
Available at: https://developers.google.com/machine-learning/recommendation/content-based/summary
[Accessed February 2022].

finnstats, 2021. *How to Calculate Jaccard Similarity in R.* [Online]
Available at: https://www.r-bloggers.com/2021/11/how-to-calculate-jaccard-similarity-in-r/
[Accessed February 2022].

Formplus, 2021. *Categorical vs Numerical Data: 15 Key Differences & Similarities.* [Online]
Available at: https://www.formpl.us/blog/categorical-numerical-data
[Accessed February 2022].

Foundation, I. D., 2022. *Hick's Law.* [Online]
Available at: https://www.interaction-design.org/literature/topics/hick-s-law
[Accessed February 2022].

Geeks, G. f., 2020. *Difference between HTML and CSS.* [Online]
Available at: https://www.geeksforgeeks.org/difference-between-html-and-css/
[Accessed September 2021].

GroupLens, 2019. *MovieLens.* [Online]
Available at: https://grouplens.org/datasets/movielens/
[Accessed February 2022].

Harmouch, M., 2021. *17 types of similarity and dissimilarity measures used in data science..* [Online]
Available at: https://towardsdatascience.com/17-types-of-similarity-and-dissimilarity-measures-used-in-data-science-3eb914d2681
[Accessed February 2022].

Hu, Y., Koren, Y. & Volinsky, C., 2008. *Collaborative Filtering for Implicit Feedback Datasets.* [Online]
Available at: https://ieeexplore.ieee.org/document/4781121
[Accessed October 2021].

IMDb, 2021. *IMDb Top 50 Musical Movies - Source Code.* [Online]
Available at: view-

source:https://www.imdb.com/search/title/?title_type=feature&genres=musical&view=advanced
[Accessed September 2021].

IMDb, 2021. *The Nightmare Before Christmas.* [Online]
Available at: https://www.imdb.com/title/tt0107688/?ref_=adv_li_tt
[Accessed October 2021].

IMDb, 2021. *The Rocky Horror Picture Show.* [Online]
Available at: https://www.imdb.com/title/tt0073629/?ref_=tt_pg
[Accessed October 2021].

IMDb, 2021. *Top 50 Musical Movies.* [Online]
Available at:
https://www.imdb.com/search/title/?title_type=feature&genres=musical&view=advanced
[Accessed September 2021].

ISO, 2000. *Information technology — Document description and processing languages — HyperText Markup Language (HTML).* [Online]
Available at: https://www.iso.org/standard/27688.html
[Accessed February 2022].

Iyer, N., 2020. *Collaborative Filtering in Pytorch.* [Online]
Available at: https://spiyer99.github.io/Recommendation-System-in-Pytorch/
[Accessed March 2022].

Kalz, M., Drachsler, H., Bruggen, J. v. & Koper, R., 2008. *Wayfinding Services for Open Educational Practices.* [Online]
Available at: https://www.researchgate.net/figure/User-based-collaborative-filtering_fig3_220049693
[Accessed September 2021].

Karabiber, F., 2022. *Jaccard Similarity.* [Online]
Available at: https://www.learndatasci.com/glossary/jaccard-similarity/
[Accessed February 2022].

Kaushik, S., 2017. *Beginner's Guide on Web Scraping in R (using rvest) with hands-on example.* [Online]
Available at: https://www.analyticsvidhya.com/blog/2017/03/beginners-guide-on-web-scraping-in-r-using-rvest-with-hands-on-knowledge/
[Accessed September 2021].

Krause, E. F., 1987. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry.* Revised ed. s.l.:Dover Publications.

Lendave, V., 2021. *Cold-Start Problem in Recommender Systems and its Mitigation Techniques.* [Online]
Available at: https://analyticsindiamag.com/cold-start-problem-in-recommender-systems-and-its-mitigation-techniques/
[Accessed February 2022].

Liberti, L. & Lavor, C., 2017. *Euclidean Distance Geometry: An Introduction.* s.l.:s.n.

Meyer, E. A., 2006. *CSS: The Definitive Guide.* Third ed. s.l.:O'Reilly Media, Inc..

Motorsport, M., 2015. *Thank you.* [Online]
Available at: https://www.magicmotorsport.com/fr/thank-you/
[Accessed March 2022].

Octoparse, 2021. *10 Web Scraping Business Ideas for Everyone.* [Online]
Available at: https://www.octoparse.com/blog/10-web-scraping-business-ideas-for-everyone
[Accessed September 2021].

Ottinger, J., 2008. *What is an App Server?.* [Online]
Available at: https://www.theserverside.com/news/1363671/What-is-an-App-Server
[Accessed February 2022].

Pascual, C., 2020. *Simple Webpage.* [Online]
Available at: https://dataquestio.github.io/web-scraping-pages/simple.html
[Accessed September 2021].

Pascual, C., 2020. *Tutorial: Web Scraping in R with rvest.* [Online]
Available at: https://www.dataquest.io/blog/web-scraping-in-r-rvest/
[Accessed September 2021].

Patriarchy, D. S., 2020. *Calculate Jaccard Similarity in Python.* [Online]
Available at: https://datascienceparichay.com/article/jaccard-similarity-python/
[Accessed February 2022].

Polamuri, S., 2015. *FIVE MOST POPULAR SIMILARITY MEASURES IMPLEMENTATION IN PYTHON.* [Online]
Available at: https://dataaspirant.com/five-most-popular-similarity-measures-implementation-in-python/#:~:text=The%20similarity%20measure%20is%20the,a%20high%20degree%20of%20similarity.
[Accessed February 2022].

RDocumentation, 2017. *vegdist: Dissimilarity Indices for Community Ecologists.* [Online]
Available at: https://www.rdocumentation.org/packages/vegan/versions/2.4-2/topics/vegdist
[Accessed February 2022].

RDocumentation, 2022. *dist.binary: Computation of Distance Matrices for Binary Data.* [Online]
Available at: https://www.rdocumentation.org/packages/ade4/versions/1.7-18/topics/dist.binary
[Accessed February 2022].

Rocca, B. & Rocca, J., 2019. *Introduction to recommender systems.* [Online]
Available at: https://towardsdatascience.com/introduction-to-recommender-systems-6c66cf15ada
[Accessed February 2022].

RStudio, 2012. *Shiny From RStudio.* [Online]
Available at: https://shiny.rstudio.com/
[Accessed October 2021].

RStudio, 2020. *Shiny in Seven Lessons.* [Online]
Available at: https://shiny.rstudio.com/tutorial/written-tutorial/lesson1/
[Accessed October 2021].

RStudio, 2022. *shinyapps.io.* [Online]
Available at: https://www.shinyapps.io/
[Accessed March 2022].

Service, N. W., 2021. *San Francisco Forecast.* [Online]
Available at: https://forecast.weather.gov/MapClick.php?lat=37.7771&lon=-122.4196#.YVxKUZrMI2x
[Accessed September 2021].

Service, N. W., 2021. *San Francisco Forecast - Source Code.* [Online]
Available at: view-source:https://forecast.weather.gov/MapClick.php?lat=37.7771&lon=-122.4196#.YYUbXmDP02y
[Accessed September 2021].

Shubhansh2006, 2019. *R Shiny Movie Recommendation App.* [Online]
Available at: https://github.com/Shubhansh2006/R-Shiny-Movie-Recommendation-App
[Accessed October 2021].

Sievert, C., 2021. *Themes.* [Online]
Available at: https://shiny.rstudio.com/articles/themes.html
[Accessed October 2021].

Smyth, A., 2022. *Final Year Project.* [Online]
Available at: https://aisling-smyth.shinyapps.io/FinalYearProject/
[Accessed March 2022].

Smyth, A., 2022. *FYP.* [Online]
Available at: https://github.com/UL18233511/FYP
[Accessed March 2022].

Soundcharts, 2021. *Plans designed for the music industry.* [Online]
Available at: https://soundcharts.com/pricing
[Accessed September 2021].

StackOverflow, 2017. *calculate jaccard distance between rows in r.* [Online]
Available at: https://stackoverflow.com/questions/41426511/calculate-jaccard-distance-between-rows-in-r
[Accessed February 2022].

StackOverflow, 2022. *Stack Overflow.* [Online]
Available at: https://stackoverflow.com/
[Accessed February 2022].

Tan, P.-N., Steinbach, M. & Kumar, V., 2005. *Introduction to Data Mining.* s.l.:Pearson.

TutorialExample, 2020. *Calculate Euclidean Distance in TensorFlow: A Step Guide – TensorFlow Tutorial.* [Online]
Available at: https://www.tutorialexample.com/calculate-euclidean-distance-in-tensorflow-a-step-guide-tensorflow-tutorial/
[Accessed February 2022].

Upwork, 2021. *Content-based Filtering Advantages & Disadvantages.* [Online]
Available at: https://www.upwork.com/resources/what-is-content-based-filtering
[Accessed February 2022].

Vatsal, 2021. *Recommendation Systems Explained.* [Online]
Available at: https://towardsdatascience.com/recommendation-systems-explained-a42fc60591ed#:~:text=Recommendation%20engines%20are%20a%20subclass,returned%20back%20to%20the%20user.
[Accessed February 2022].

Verma, Y., 2021. *A Guide to Building Hybrid Recommendation Systems for Beginners.* [Online]
Available at: https://analyticsindiamag.com/a-guide-to-building-hybrid-recommendation-systems-for-beginners/
[Accessed February 2022].

Vodnik, S., 2020. *HTML FOR WEB DEVELOPMENT: BUILDING THE BONES OF YOUR WEBSITE.* [Online]
Available at: https://generalassemb.ly/blog/html-web-development-building-bones-website/#:~:text=Hypertext%20Markup%20Language%2C%20or%20HTML,and%20JavaScript%20programming%20its%20functionality
[Accessed September 2021].

Wenig, B., 2019. *Collaborative Filtering.* [Online]
Available at: https://brookewenig.com/ALS.html#/5
[Accessed October 2021].

Wickham, H., 2020. *Mastering Shiny.* [Online]
Available at: https://mastering-shiny.org/index.html
[Accessed October 2021].

Wickham, H. & RStudio, 2021. *Package 'rvest'.* [Online]
Available at: https://cran.r-project.org/web/packages/rvest/rvest.pdf
[Accessed September 2021].

XILINX, A., 2022. *Xilinx Cosine Similarity Alveo Product Overview.* [Online]
Available at: https://xilinx.github.io/graphanalytics/cosinesim/overview.html
[Accessed February 2022].

Zornoza, J., 2020. *Distance Metrics for Machine Learning.* [Online]
Available at: https://aigents.co/data-science-blog/publication/distance-metrics-for-machine-learning
[Accessed February 2022].