Build-A-Movie

Introduction to Data Science mini-project

University of Helsinki

Faculty of Science

Enrico Buratto
Perttu Lähteenlahti
Touko Puro

Academic Year 2021-2022

Abstract

The aim of this project is to build a tool for movie directors and investors to play around with movie data with the goal of picking the best combination of actors, directors, and genres to build the highest-performing imaginary movie they can come up with. The performance is based on an estimated box office score model built from existing movie box office numbers and IMDb ratings. The final goal is to give movie directors a general direction on which movie can be a good investment of time and money and which not using a simple and fast-forward web application.

Contents

1 Introduction

1.1 The problem

With the work reported in this document, we try to provide movie directors and investors a tool to help them in the decision-making process wether to direct and invest in a movie or not.
In order to do that, we first focused on some of the crucial features of a movie, such as actors, directors, budget, genres and so on. Then, we found the right data sources (*i.e.* IMDb and Box Office Mojo) and we gathered all the needed data from them, we did some preprocessing on it and we used it to build two machine learning models.
These models are then used by a simple webapp to infere a possible IMDb rating and the possible worldwide lifetime gross given the features of the movie that need to be checked; this application consists in a simple and straightforward web application in which the user can enter possible key people, genres and budget and then get the results in an effortless fashion.

1.2 Document overview

In this document we will analyze our work step by step. In section §2 we talk about the data, *i.e.* from which sources we got it, how we managed it and how we processed it for further usage; in section §3 we then talk of data analysis, *i.e.* on which data we focused on and how we used it for our project's purposes.
Section §4 describes the final product we were able to produce; in section §5, finally, we analyze some possible work that could be done on our project in order to improve it and make it even more useful and, maybe, for even different categories of users.

2 Data

2.1 Data sources

The data for our project comes from two different sources: IMDb and Box Office Mojo. We gathered data from both of these websites in different ways, and then we combined them in order to build a consistent and coherent dataset to use statically in further steps.

2.1.1 IMDb

IMDb is a famous online database of information about movies and tv series. In order to gather data from IMDb we could have followed several ways, which we analyzed in order to find the most suitable for us:

- **IMDb APIs**: IMDb offers APIs to get both specific and generic data from IMDb. Even if this could have been a suitable method, these APIs are not for free; thus, we decided not to use them;
- **IMDbPY**: IMDbPY is a python library that offers almost the same functionalities of the official IMDb APIs but for free; it uses the same static datasets discussed in the next point;
- **IMDb datasets**: IMDb offers also static datasets for free. These datasets consist in turn of 6 different sub-datasets with one common field (`tconst`), and a seventh one that is out of the scope of our project.

We finally opted for the last method (*i.e.* IMDb datasets) because we would have used IMDbPY in the same way, that is downloading the datasets in a static fashion.
From IMDb we gathered data regarding:

- General information about the titles, including the genres;
- Information about the people involved (actors, directors, etc...);
- Information about ratings.

The single fields we then used are described in the data selection and cleaning section (§2.3).

2.1.2 Box Office Mojo

Box Office Mojo is a website that tracks box office information about movies. With Box Office Mojo the situation was different than with IMDb: this website does not offer, in fact, any official API, and all the unofficial APIs are now deprecated due to continuous changes to the website.
Since the data from Box Office Mojo was (and is) of upmost importance for our models, we decided to continue on this way and write a web scraper from scratch. This scraper works as follows:

- For every title from IMDb, the script searches it on the website using a GET request after processing the string. This pre-processing consists in escaping special characters, truncating it to 40 characters and removing numbers; the purpose of this procedure is to improve the probabilities to find the movie;
- If there's one or more results, the script checks the similarity between the first result's title and the search query with the `ratio()` function of the class `SequenceMatcher` from `difflib` python library:
  - If the similarity is more than 0.6 (60%) it is pretty sure that the searched title is the right one, so the script proceeds to the next step;
  - If the similarity is less than 0.6 the title is discarded and no more used for the model training; this is done in order to not insert wrong data, that could invalidate the models, in the dataset;
- After making sure of the correctness of the result, the scraper then proceeds to gather the data from the first result's page.

From Box Office Mojo we gathered data regarding the budget and the different categories of earnings (domestic, international and worldwide); this, as per the IMDb data, will be better analyzed in the data selection section (§2.3).
In Appendix A the reader can view an example of the search on the website and of the data we gather from the result page.
Note that this procedure consists in scraping the information from the first result's page; this means that it is assured that it worked during this phase of the project, but it can't be trusted after some time since the website is continuously under development and even a small change in the front-end (*e.g.* a different html tag) could make the script no longer functional.

2.2 Data management

During this part of the project we used a PostgreSQL database in order to work together on a unique dataset; we took advantage of Heroku's free plan in order to store and manipulate the data for free.
We started using the database between the IMDb data gathering and the execution of the Box Office Mojo scraper: first, we downloaded the data from IMDb; secondly, we uploaded the different datasets as tables into Heroku; then we used the common database to scrape the data from Box Office Mojo using the titles of the movies, which have been extracted via a SQL query.
Finally, we created a last table with all the data we needed to proceed with the model training; a more extensive description of the crucial parts of these steps is reported in the next section.
In Appendix B reader may find the database schema with a description of every table for reference.

2.3 Data selection and cleaning

As just said, after downloading the datasets from IMDb, we pushed them into tables on Heroku. We then started joining and filtering them using SQL queries both via the console and via simple python code; since all the tables of our interest had the same `tconst` field, it has been pretty easy to manipulate the data.
Since the webscraper takes about 2 seconds to get the needed data for each movie, and there where more than 1˙000˙000 entries, first of all we filtered the movies to reduce the number of title to search. In order to do that, we took into account:

- **Runtime minutes**: we set a lower bound for runtime minutes to 80, in order to exclude short films;

- **Number of votes**: we set a lower bound for number of votes on IMDb of 10˙000;
- **Type of title**: we applied the query only on titles with category equals to ”movie” (*i.e.* cinema movies) and ”tvMovie”.

This filter is summarized by the following query (explanation of the names used in this query can be found in Appendix B):

```
SELECT tb.tconst, original_title
FROM title_basics AS tb JOIN title_ratings AS tr
ON tb.tconst = tr.tconst
WHERE runtime_minutes > 80.0 AND num_votes > 10000
AND (title_type='movie' OR title_type='tvMovie')
```

We then started the webscraper using `original_title` as input string for the GET requests.

Regarding the management of missing data, we thought of which method was more suitable for us: we could have done some inference on the missing fields, *e.g.* using the average or the median value of the column, or we could have just discarded the rows for which some field was missing. We decided to adopt this last technique, since our only interest was to have a consistent dataset to train the models correctly and, moreover, we had a huge amount of data.
Once finished the execution of the script we cleaned the data. Since it was gathered directly from the html tags of the website, everything was formatted as a string: we then did some processing for every row in every field in order to remove unwanted characters and cast the values to big int format.
Then we performed a final join operation in order to get a unique and complete table; the final table schema is reported in Table 1 and it is useful for the further sections.

Column name	**Data type**	**Description**
tconst	bigint	Unique identifier of the movie
genres	text	Array containing the genres of the movie
primary_name	text	Name of person involved in movie
category	text	Category of primary_name role (actor, director, ...)
original_title	text	Title of the movie
worldwide_lifetime_gross	bigint	Total earnings of the movie
budget	bigint	Budget of the movie
average_rating	double precision	IMDb rating of the movie

Table 1: Schema of the final table

As a last step of this proceeding, we downloaded the table as a csv file.
Note that during this whole process there was no need to perform any data cleaning on IMDb data but only on Box Office Mojo data (except for the empty row/fields removal), since the former was already cleaned and ready to use.

3 Data analysis

As previously stated, we used the data we gathered in order to train two machine learning models: one to predict the IMDb rating and the other to predict the earnings of the input movie. Here follows a summary of the data analysis step.

3.1 Features

The final features we used are the one listed in Table 1 (except for `tconst`, that is left on the table only for possible future reuse, and `original_title`).
In order to make categorical variables (`genres` and `primary_name` along with `category`) usable by the machine learning algorithm, we firstly did one-hot-encoding on them in a separate way; we then concatenated the encoded genres and people columns to the dataset file, along with `worldwide_lifetime_gross`, `budget` and `average_rating`.

It is important to underline that `primary_name` and `category` have been considered as just "people"; we did this assumption based on two facts:

- Actors, directors, writers are usually different people because these jobs are separate jobs;
- If an actor also wrote something, or a director also acted (*e.g.* Quentin Tarantino), probably it is inside the same movie.

We then decided to reduce the dimensionality of data considering all the professionals on a same level.

3.2 Targets

As already stated, the targets of the models are two:

- IMDb rating;
- Worldwide Lifetime Gross.

3.3 Models

We analyzed two different options for the models:

- **Linear regression**: an option was to use multivariate linear regression. The pros of this kind of model are that is simple and is computionally efficient, hence the training part would be fast. However, this approach brings with it some problems like the assumption of linearity, *i.e.* the target and the feature are linearly correlated, and most important the independence of the features, *i.e.* the model assumes that the dependent variables are not correlated with each other;
- **Random Forest**: the other option was to use a random forest classifier. The pros of this model are that it works well with non-linear data and that it has low risk of overfitting; however, the training process for this model would have been surely lower than linear regression.

We finally opted for random forest because we didn't need the training to be fast: in our workflow, indeed, we only had to have a binary file with the model for further usage, but the training part might as well be slow.
We then used `RandomForestRegressor` from `scikit-learn` python library.

4 Final product

4.1 Back-end component

After building the models, we discussed on how to offer the prediction in the most effective way. We discussed and tried two options for the back-end:

- **Lambda functions**: as discussed in the next section, for the deployment we used Vercel; this platform allows to write simple APIs modeled as lambda functions: it is enough to upload a simple python file into the `API/` folder, and the deployment service automatically create the endpoint. Alternatively, AWS (Amazon Web Services) Lambda was taken into account;
- **Private server**: another option was to use a dedicated server with a python HTTP server to expose two endpoints in order to apply the prediction on user input.

At first we tried with the first option, *i.e.* with the Vercel's automatic API builder: we wrote the code and we uploaded it, and the deployment was succesful. However, we soon met a problem: the computation time was too long and the requests timed out.
We then tried with AWS Lambda, but the size of the image was too big because of the high amount of libraries needed.
We finally opted for the private server, so we set up a VPS (Virtual Private Server) using the DigitalOcean corresponding service. On this server we set up an HTTP python server that offers two endpoints: `/predict-rating` and `/predict-wlg`. A very short description of these APIs can be found in Table 2.

Endpoint	Input	Model	Output
`/predict-rating`	Array of people Array of genres Budget	rating_forest	IMDb rating
`/predict-wlg`	Array of people Array of genres Budget	worldwide_lifetime_gross	Predicted earnings

Table 2: Description of APIs.

More detailed API specification can however be found at Appendix C.

4.2 Front-end component

The Front-end is built using the following technologies:

- **React**: React is a JavaScript library for building user interfaces through component based approach.
- **Next.js**: is a framework for React. In this project it is used to handle routing and static site generation.
- **TypeScript**: TypeScript is a strongly typed programming language that builds on JavaScript.
- **Tailwind**: Tailwind is essentially a collection of CSS utility classes that make it easier to build user interface styles.

The front-end is deliberately simple, consisting of only short introduction to the Build-A-Movie website, three different select inputs allowing you to select multiple values, and the result section. The front-end is also designed responsively and with the goal of being both usable and accessible on all devices.

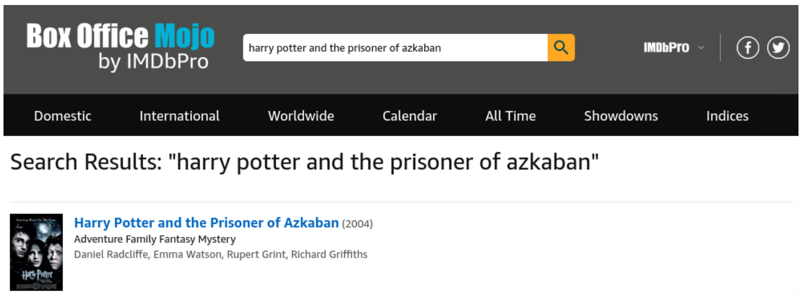
One of the hardest parts about the front-end was getting the multi-selects to work correctly. First problem was finding a good multi-select component which would allow for easy asynchronous data. Second was actually getting the data. The first implementations used Next.js cloud functions to find the right options from a large CSV file. However this turned out to be very slow so the next implementation used a combination of Postgres database hosted on Heroku, and then served through a GraphQL API in Hasura. This solution worked well, and enabled for example filtering directly in the multi-select component. However, as the amount of data was too large for Heroku's free plan, the databases were automatically deleted by Heroku after seven days. The current solution also used Heroku to host the databases, but currently they incur a minuscule cost, so this solution is not a long term one.

5 Future work

We wish to conclude this report talking about some possible improvements that could be done to our work. Remaining on the same target audience of our project (*i.e.* movie professionals), a good enhancement that could be done is inferring the score and the earnings not only on the described features, but also on the title and/or on the synopsis. For instance, natural language processing can be performed on some user input text in order to analyze similarities also on the plot of the movie; some useful tools to do that could be Natural Language Toolkit (NLTK) and TextBlob. Wanting then to enlarge the target audience, movie enthusiasts can be included making the website more like a game, adding features like a *net of interconnected movies* to play around with similar movies, or some gamification technique such as global leaderboards and challenges.

Appendices

A Box Office Mojo scraper example

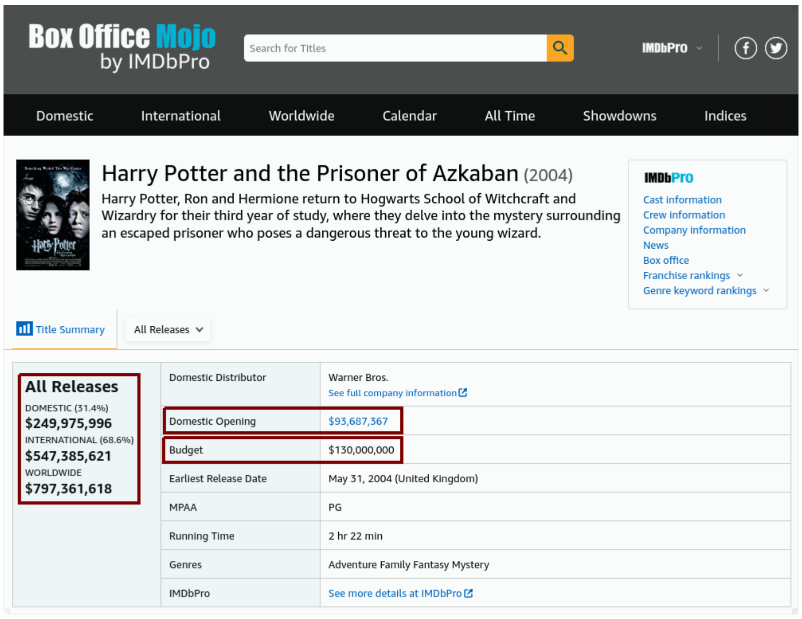


harry potter and the prisoner of azkaban

Search Results: "harry potter and the prisoner of azkaban"

Harry Potter and the Prisoner of Azkaban (2004)
Adventure Family Fantasy Mystery
Daniel Radcliffe, Emma Watson, Rupert Grint, Richard Griffiths

Figure 1: First step: search of a movie title on Box Office Mojo.



Harry Potter and the Prisoner of Azkaban (2004)

Harry Potter, Ron and Hermione return to Hogwarts School of Witchcraft and Wizardry for their third year of study, where they delve into the mystery surrounding an escaped prisoner who poses a dangerous threat to the young wizard.

All Releases
DOMESTIC (31.4%) $249,975,996
INTERNATIONAL (68.6%) $547,385,621
WORLDWIDE $797,361,618

Domestic Distributor	Warner Bros. See full company information
Domestic Opening	$93,687,367
Budget	$130,000,000
Earliest Release Date	May 31, 2004 (United Kingdom)
MPAA	PG
Running Time	2 hr 22 min
Genres	Adventure Family Fantasy Mystery
IMDbPro	See more details at IMDbPro

Figure 2: Second step: scraping of the data of interest.

B Database schema

Table name	Description
title_basics	Basic information about movies
title_principals	Information about people working on the movies
title_akas	Different titles for movies (location-based)
title_ratings	Information about IMDb ratings
title_crew	Information about directors and writers of the movies
movie_boxoffice	Information about box office (gathered with the webscraper)

Table 3: Information schema of the tables in the database.

Column name	Data type	Description
tconst	integer	Unique identifier of the movie
primaryTitle	text	Principal title of the movie
endYear	integer	Year when the movie was finished
original_title	text	Original title of the movie
startYear	integer	Year when the movie started being directed
runtime_minutes	integer	Runtime of the movie in minutes
title_type	character varying	Type of the title
genres	text	Array containing the genres of the movie
isAdult	boolean	Boolean value that states if the movie is for adult audience

Table 4: Schema for title_basics table.

Column name	Data type	Description
tconst	integer	As above
nconst	integer	Unique identifier of the name/person
job	character varying	Job of the person
characters	character varying	Name of the character played if applicable
category	character varying	Category of job that person was in
ordering	integer	Number to uniquely identify rows for a given tconst

Table 5: Schema for title_principals table.

Column name	Data type	Description
isOriginalTitle	boolean	0: not original title; 1: original title
attributes	character varying	Additional terms to describe this alternative title
tconst	integer	Unique identifier of the title
language	character varying	Language of the title
title	text	Localized title
types	character varying	Enumerated set of attributes for this alternative title
region	character varying	Region for this version of the title
ordering	integer	Number to uniquely identify rows for a given tconst

Table 6: Schema for title_akas table.

Column name	Data type	Description
tconst	integer	Unique identifier of the title
average_rating	double precision	Weighted average of all the individual user ratings
num_votes	integer	Number of votes

Table 7: Schema for title_ratings table.

Column name	Data type	Description
tconst	integer	Unique identifier of the title
directors	text	Array of directors of the movie
writers	text	Array of writers of the movie

Table 8: Schema for title_crew table.

Column name	**Data type**	**Description**
tconst	bigint	Unique identifier of the title
original_title	text	Original title of the movie
worldwide_lifetime_gross	bigint	Total earnings of the movie
budget	bigint	Budget of the movie

Table 9: Schema for movie_boxoffice table.

C API specification

C.1 /predict-rating

Request

Name	**Method**	**Description**
/predict-rating	POST	Get IMDb rating prediction

Table 10: `/predict-rating` requests.

Parameters

Name	**Data type**	**Required/optional**	**Description**
genres	array (string)	required	List of genres
people	array (string)	required	List of people (directors, actors, etc...)
budget	big int	required	Budget

Table 11: `/predict-rating` POST parameters.

Response

Name	**Data type**	**Description**
result	json (string)	Result (predicted rating) in json format

Table 12: `/predict-rating` POST parameters.

C.2 /predict-wlg

Request

Name	**Method**	**Description**
/predict-wlg	POST	Get worldwide lifetime gross

Table 13: `/predict-wlg` requests.

Parameters

Name	**Data type**	**Required/optional**	**Description**
genres	array (string)	required	List of genres
people	array (string)	required	List of people (directors, actors, etc...)
budget	big int	required	Budget

Table 14: `/predict-wlg` POST parameters.

Response

Name	**Data type**	**Description**
result	json (string)	Result (predicted earnings) in json format

Table 15: `/predict-wlg` POST parameters.