Springboard

Capstone Project 3

Movies Recommendation Engine

Using TensorFlow Recommenders (TFRS)- Hybrid Recommender

Final Report

November 2021

Aktham Momani – Data Science: Advanced Machine Learning

Mentor: Kenneth Gil-Pasquel

Table of Contents

1. Introduction

Over the last few decades, with the advent of YouTube, Amazon, Netflix and many other Web services, recommendation platforms are becoming much more part of our lives from e commerce, suggesting the customers articles that could be of interest.

In a very general way, Recommendation systems are algorithms program to present related things to users with items being movies to watch, books to read, products to buy or anything else, depending on the industry. Recommendation systems are very important in some industries because they can produce a large amount of money if they're effective, or if they are a way of standing out dramatically from competitors. The aim of the recommendation framework is to produce relevant suggestions for the collection of users of objects or products that may be of interest to them. Suggestions for Amazon books or Netflix shows are all really world examples of how industry leading systems work. The architecture of such recommendation engines depends on the domain and basic characteristics of the available data.

There are mainly Three types of recommendation engines:

- Collaborative filtering.
- Content based Filtering.
- Hybrid (Combination of Collaborative and Content based Filtering).

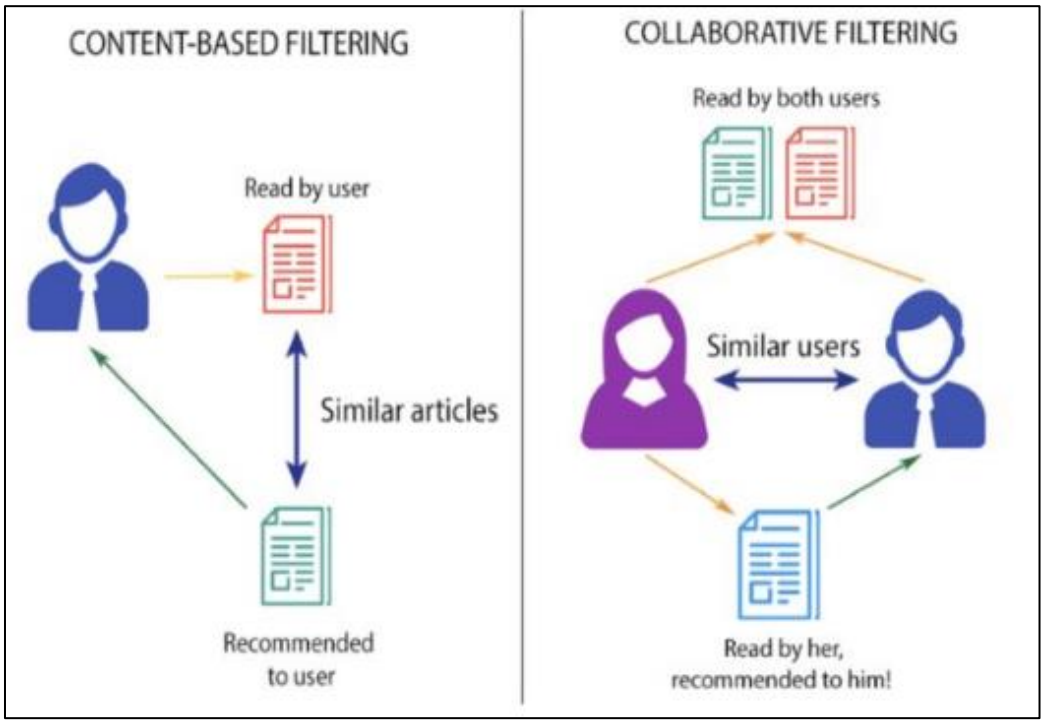


Figure 1: Recommendation Engines Types

2. Objective

This Project is all about how to successfully formulate a recommendation engine, the difference between implicit and explicit feedback and how to build a movie recommendation system with TensorFlow and TFRS.

Google/YouTube is all about connecting people to the movies/videos they love. To help customers find those movies, they developed world-class movie recommendation system called TensorFlow Recommender (TFRS). Its job is to predict whether someone will enjoy a movie based on how much they liked or disliked other movies. Google/YouTube uses those predictions to make personal videos recommendations based on each user's unique tastes.

Why choosing TensorFlow Recommenders (TFRS)?

- TensorFlow Recommenders (TFRS) is a library for building recommender system models.
- It helps with the full workflow of building a recommender system: data preparation, model formulation, training, evaluation, and deployment.
- It's built on Keras and aims to have a gentle learning curve while still giving you the flexibility to build complex models.

TFRS makes it possible to:

* Build and evaluate flexible recommendation retrieval models.

* Freely incorporate item, user, and context information into recommendation models.

* Train multi-task models that jointly optimize multiple recommendation objectives.

TFRS is open source and available on **Github**.

To learn more, see the tutorial on how to build a movie recommender system, or check the API docs for the API

3. Data

Because of the richness of the metadata in Tensorflow Movie Lens dataset, we have decided to choose 1 million Movie lens from TensorFlow to be our main dataset for this project. Also, we used both datasets from **Movielens website: movies metadata & credits.**

Movie Lens contains a set of movie ratings from the **Movielens website**, a movie recommendation service. This dataset was collected and maintained by **GroupLens** , a research group at the University of Minnesota. There are 5 versions included: "25m", "latest-small", "100k", "1m", "20m". In all datasets, the movies data and ratings data are joined on "movie_Id". The 25m dataset, latest-small dataset, and 20m dataset contain only movie data and rating data. The 1m dataset and 100k dataset contain demographic data in addition to movie and rating data.

movie_lens/1m can be treated in two ways:

- It can be interpreted as expressing which movies the users watched (and rated), and which they did not. This is a form of *implicit feedback*, where users' watches tell us which things they prefer to see and which they'd rather not see (This means that every movie a user watched is a positive example, and every movie they have not seen is an implicit negative example).
- It can also be seen as expressing how much the users liked the movies they did watch. This is a form of *explicit feedback*: given that a user watched a movie, we can tell roughly how much they liked by looking at the rating they have given.

a) movie_lens/1m-ratings

- Config description: This dataset contains 1,000,085 anonymous ratings of approximately 3,619 movies made by 6,040 MovieLens users who joined MovieLens. Ratings are in whole-star increments. This dataset contains demographic data of users in addition to data on movies and ratings.
- This dataset is the largest dataset that includes demographic data from movie_lens.
- "user_gender": gender of the user who made the rating; a true value corresponds to male
- "bucketized_user_age": bucketized age values of the user who made the rating, the values and the corresponding ranges are:
    - 1: "Under 18"
    - 18: "18-24"
    - 25: "25-34"
    - 35: "35-44"
    - 45: "45-49"
    - 50: "50-55"
    - 56: "56+"

- "movie_genres": The Genres of the movies are classified into 21 different classes as below:
    - 0: Action
    - 1: Adventure
    - 2: Animation
    - 3: Children
    - 4: Comedy
    - 5: Crime
    - 6: Documentary
    - 7: Drama
    - 8: Fantasy
    - 9: Film-Noir
    - 10: Horror
    - 11: IMAX
    - 12: Musical
    - 13: Mystery
    - 14: Romance
    - 15: Sci-Fi
    - 16: Thriller
    - 17: Unknown
    - *18: War
    - * 19: Western
    - * 20: no genres listed
- "user_occupation_label": the occupation of the user who made the rating represented by an integer-encoded label; labels are preprocessed to be consistent across different versions
- "user_occupation_text": the occupation of the user who made the rating in the original string; different versions can have different set of raw text labels
- "user_zip_code": the zip code of the user who made the rating.
- "release_date": This is the movie release date, in unix epoch (UTC - units of seconds) (int64).
- "director": This is the director of the movie.
- "start": This is the main star of the movie.

b) movie_lens/1m-movies:

- Config description: This dataset contains data of approximately 3,619 movies rated in the 1m dataset.

4. Data Wrangling

4.1 Introduction

The Data wrangling step focuses on collecting our data, organizing it, and making sure it's well defined. For our project we have collected below datasets to have a good foundation so we can build a Deep Learning model with the best performance possible:

a) movie_lens/1m-ratings

b) movie_lens/1m-movies:

c) movies_metadata.csv

d) credits.csv

4.2 Objectives

We have 4 Datasets to support this project as shown above, so we'll focus in below:

- Convert Tensorflow datasets (1m-ratings and 1m-movies) from Tensorflow (***ensorflow_datasets.core.as_dataframe.StyledDataFrame) to pandas DataFrame (*pandas.core.frame.DataFrame) *for easy data wrangling*** (Some Pandas method doesn't work well with *StyledDataFrame from TensorFlow)*
- Fix any wrong values in user_zip_code (Any zipcode >5 characters).
- Fixing **"movie_genres":** let's make sure that genres are the format of a list for easy access. Merging and concatenation will be needed.
- Fixing **"user_occupation_label":** one category label is missing "10" causing 'K-12 student' & 'college/grad student' to be labeled as "17" so here, we'll assign "10" to 'K-12 student'.
- Add 5 more features to the original Dataset: **'cast', 'director', 'cast_size', 'crew_size', 'imdb_id', 'release_date' and movie_lens_movie_id** --> Will get these features using 2 datasets from **Movielens website:**
    - **movies_metadata.csv**
    - **credits.csv**
- Fix existing wrong movie title (or in some cases misspelled).
- Remove all special characters or letter accents from Movie titles, cast and director.
- Add movie id which is matching the original movie id in the movie lens original dataset (for some reason the movie id from Tensorflow dataset is not matching).
- Fix duplicates movie_title with same movie_id.
- After fixing above items, we converted Pandas DataFrame to Tensorflow dataset:
    - From 'cast' features, let's drop all secondary casting and keep only the star of the movie and let's call the feature "star".
    - Let's make sure to keep only the important columns.

- Change the data types of the important features to fit with Tensorflow-Recommender TFRS Library.
- Keep in mind **tfds** currently does not support float64 so we'll be using int64 or float32 depends on the data.
- Wrap the **pandas DataFrame** into **tf.data.Dataset** object using **tf.data.Dataset.from_tensor_slices** (To check other options - **here**)

	bucketized_user_age	movie_genres	movie_id	movie_title	timestamp	user_gender	user_id	user_occupation_label	user_occupation_text	user_rating	user_zip_code
0	25.0	[3, 4]	b'586'	b'Home Alone (1990)'	975897100	True	b'595'	6	b'executive/managerial'	4.0	b'10019'
1	35.0	[0, 1, 4, 14]	b'1197'	b'Princess Bride, The (1987)'	972790580	False	b'2804'	11	b'other/not specified'	5.0	b'46234'
2	25.0	[4, 14]	b'2502'	b'Office Space (1999)'	974757114	True	b'1457'	0	b'academic/educator'	4.0	b'95472'
3	18.0	[1, 3, 8]	b'2'	b'Jumanji (1995)'	965806208	True	b'3887'	17	b'college/grad student'	3.0	b'80513'
4	35.0	[10, 16]	b'1717'	b'Scream 2 (1997)'	976474533	True	b'329'	6	b'executive/managerial'	2.0	b'02115'

Table 1 Original Dataset Head (TensorFlow)

	bucketized_user_age	movie_genres	movie_id	movie_title	timestamp	user_gender	user_id	user_occupation_label	user_occupation_text	user_rating	user_zip_code	director	release_date	star
0	50	[7]	1251	Eight and half	974089380	False	2497	14	sales/marketing	3	37922	Federico Fellini	-217123200	Marcello Mastroianni
1	18	[7]	1251	Eight and half	986722200	True	671	17	college/grad student	5	61761	Federico Fellini	-217123200	Marcello Mastroianni
2	45	[7]	1251	Eight and half	960071880	False	5590	12	programmer	2	94117	Federico Fellini	-217123200	Marcello Mastroianni
3	25	[7]	1251	Eight and half	1011993120	True	1851	20	unemployed	5	59602	Federico Fellini	-217123200	Marcello Mastroianni
4	35	[7]	1251	Eight and half	963100320	False	5526	1	artist	5	27514	Federico Fellini	-217123200	Marcello Mastroianni

Table 2 Personal Final Dataset

5. Exploratory Data Analysis

5.1 Introduction

Now that we've obtained, cleaned, and wrangled our dataset into a form that's ready for analysis, it's time to perform exploratory data analysis (EDA).

Figure 2 WordCloud representing worst Movies per rating - All Time

5.2 Objectives

- To get familiar with the features in our final DataFrame.
- Generally, understand the core characteristics of our cleaned DataFrame.
- Explore the data relationships of all the features and understand how the features compare to the response variable.
- Let's be creative and think about interesting figures and all the plots that can be created to help deepen our understanding of the data.

5.3 Initial Statistics Summary

	bucketized_user_age	timestamp	user_occupation_label	user_rating	user_zip_code
count	1000209.0	1000209.0	1000209.0	1000209.0	1000209.0
mean	29.7	972243695.4	11.1	3.6	54230.9
std	11.8	12152558.9	6.6	1.1	32090.6
min	1.0	956703932.0	0.0	1.0	231.0
25%	25.0	965302637.0	6.0	3.0	23185.0
50%	25.0	973018006.0	12.0	4.0	55129.0
75%	35.0	975220939.0	17.0	4.0	90004.0
max	56.0	1046454590.0	21.0	5.0	99945.0

Table 3 Statistic Summary

5.4 What is the preferable month of the year to rate/watch movies?

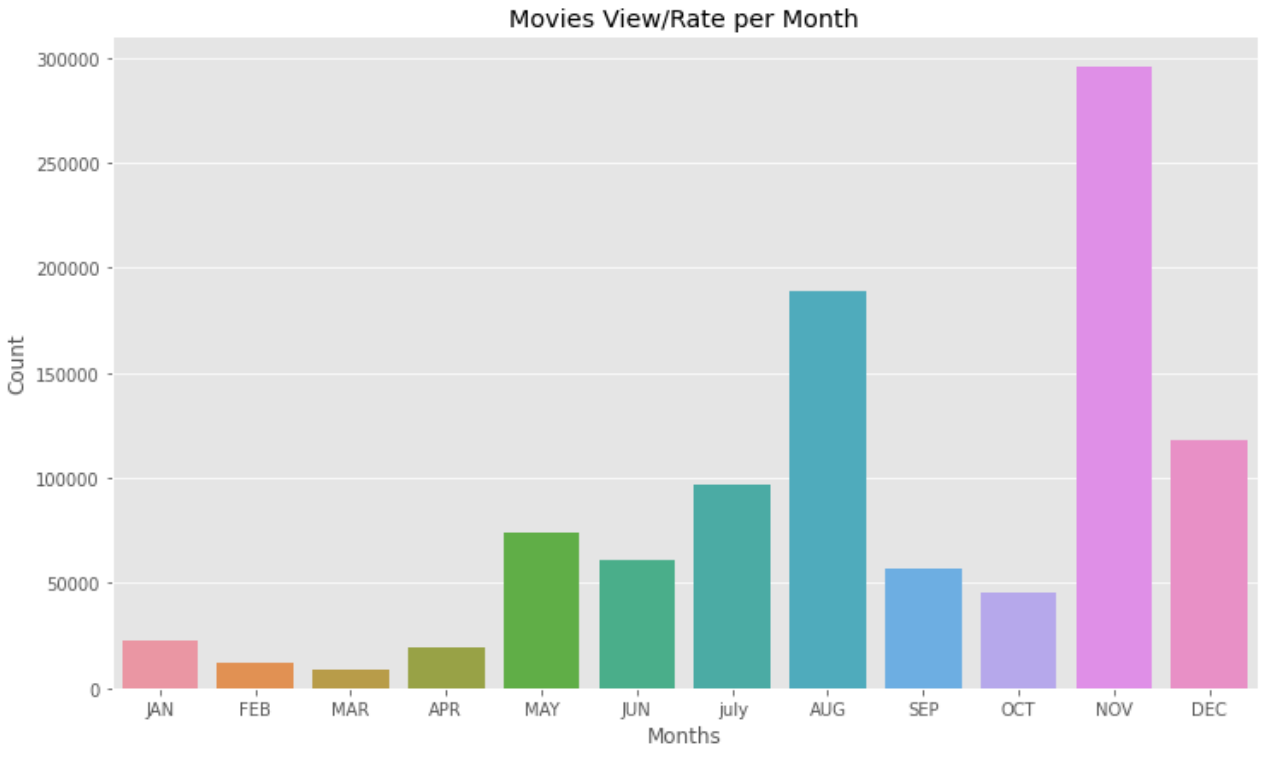


Figure 3

Summer & Holidays Months are the highest, which make sense!!!

5.5 What is the preferable day of the week to rate/watch movies?

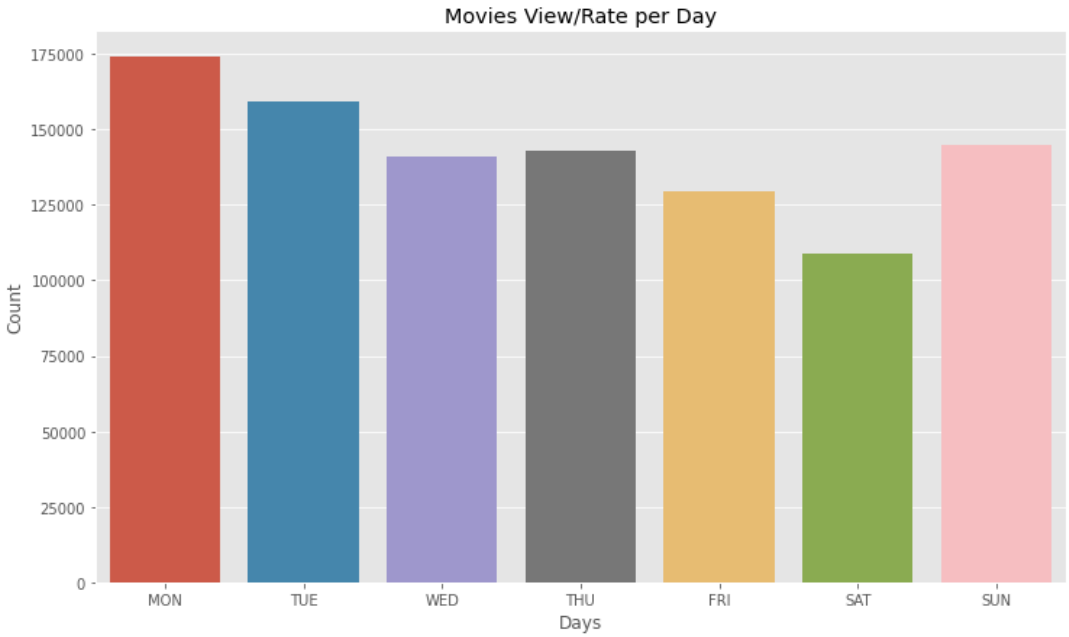


Figure 4

As shown above, looks like people enjoys watching/rating movies during weekdays and probably going out for a theater during the weekend (low rating/watching).

5.6 Who watches/rates more movies Men/Women?

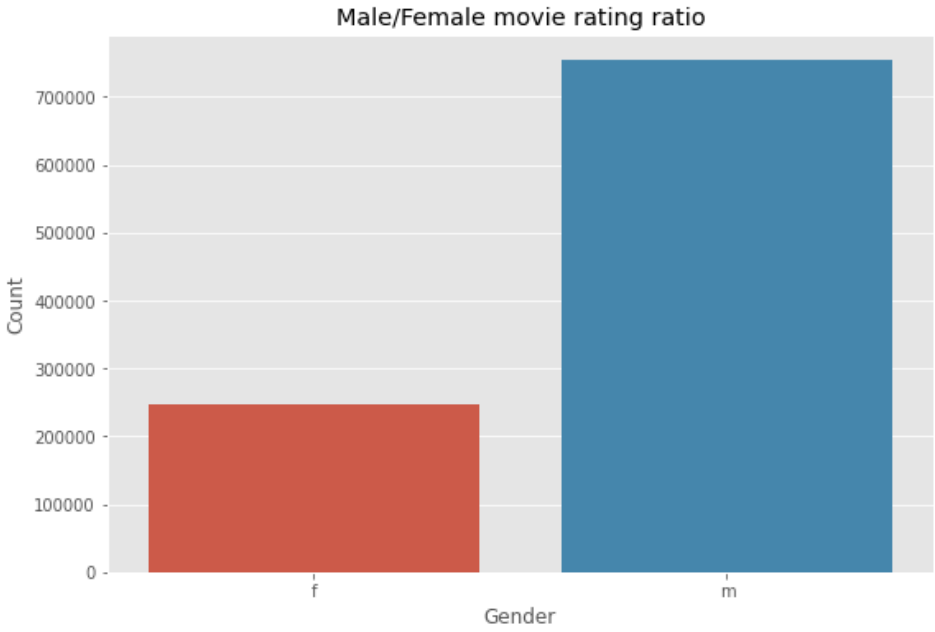


Figure 5

Males look like are more interesting in rating movies than females!!

5.7 What age group watches more movies?

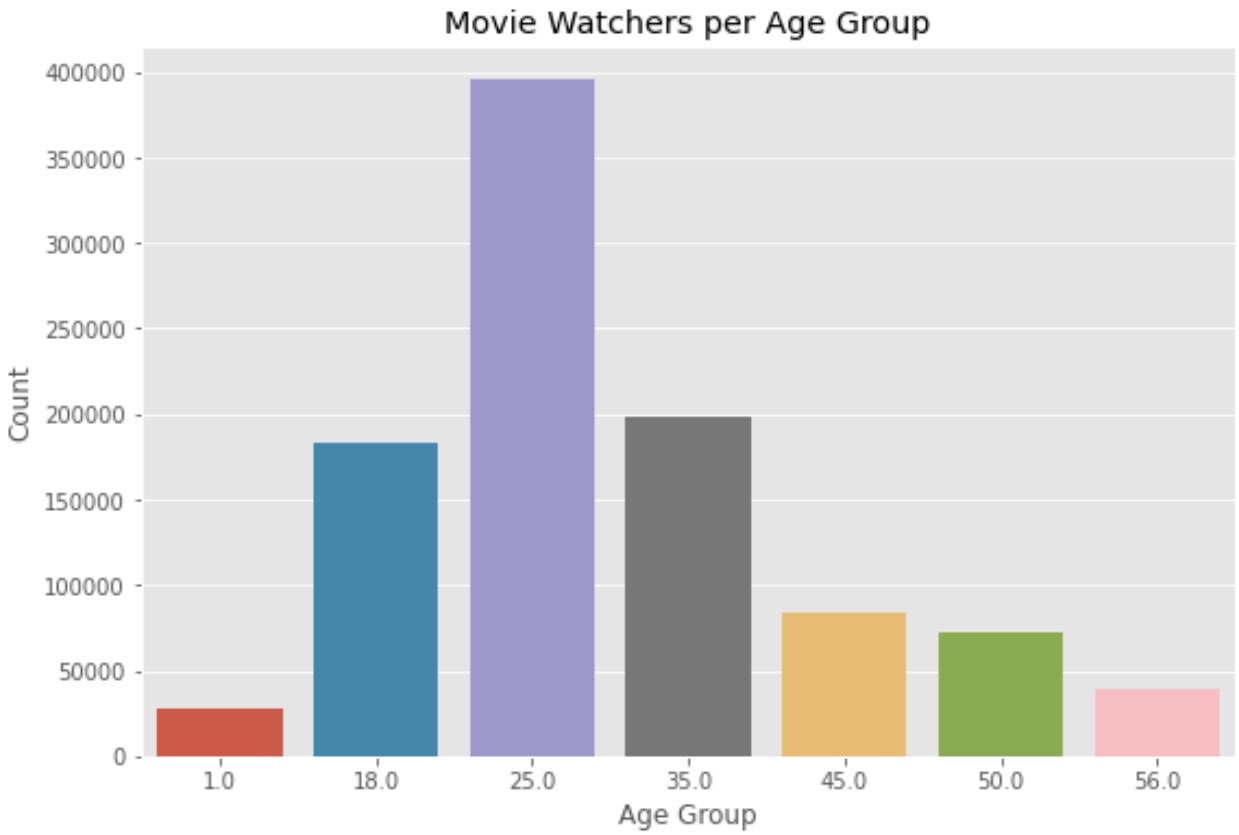


Figure 6

As shown above, users aged between 18 to 34 are most who watch or rate Movies !!!:

- 25: 25-34 are most age group that rate movies
- 35: 35-44 2nd runner age group that rate movies
- 18: 18-24 3rd runner age group that rate movies

5.8 What kind of occupant watches/rates more movies?

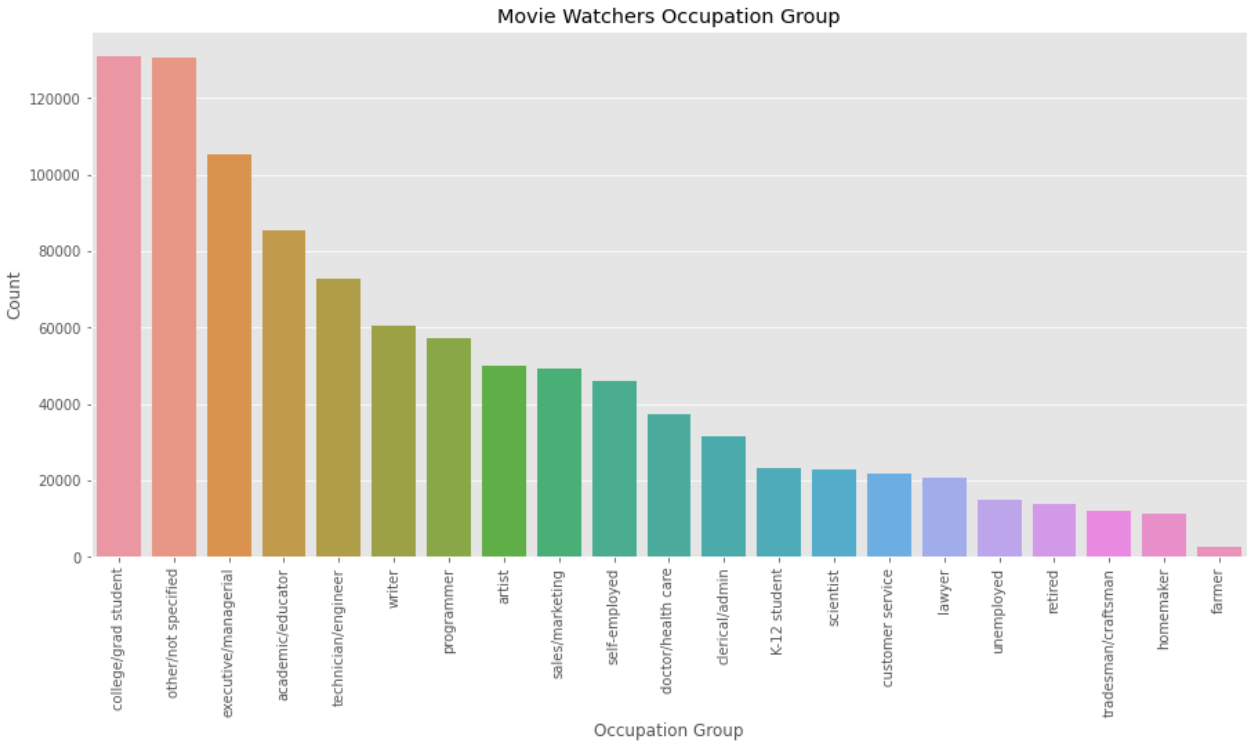


Figure 7

Ok, let's drill down and explore more details ....

	user_occupation_text	bucketized_user_age	0
0	college/grad student	18.0	88024
3	college/grad student	25.0	38971
66	college/grad student	35.0	3124
112	college/grad student	50.0	442
122	college/grad student	45.0	250
123	college/grad student	1.0	221

Table 4 College/grad student per Age

Alright, now we have more insights ... looks like college/grad students aged between 18-34 are the most who watches or rates movies.

5.9 How much rating people give mostly? Distributed between genders?

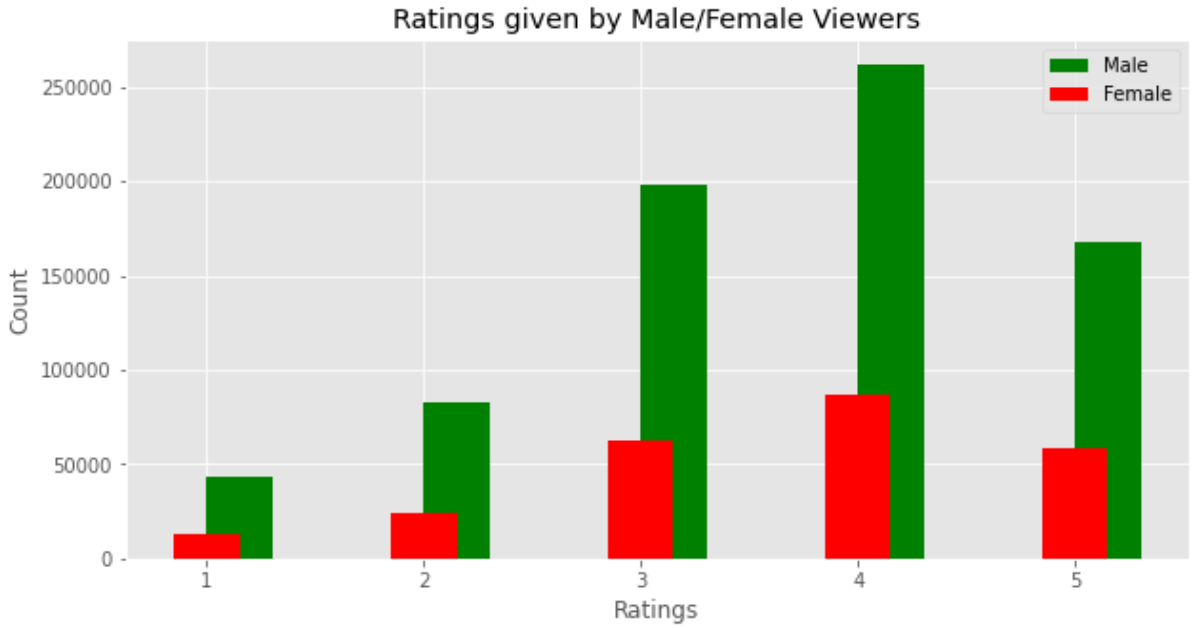


Figure 8

Ok, That's interesting both males and females have shown the same trend in ratings and both have geven 4 as the highest ratings!!!

5.10 What are the most rated movies? In terms of:

5.10.1 All Time:

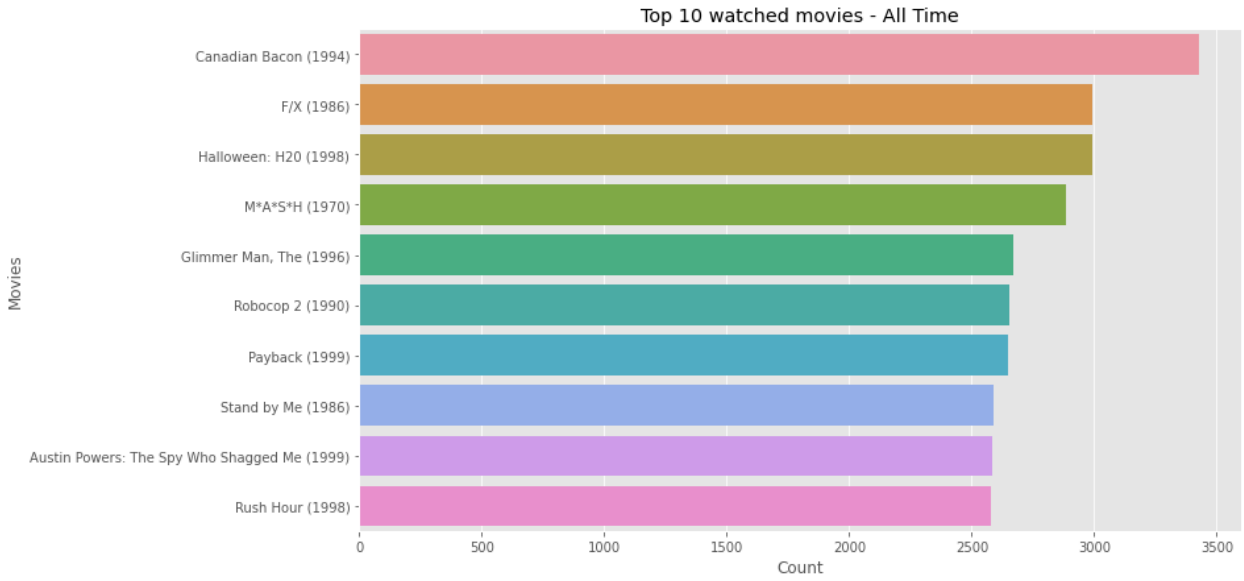
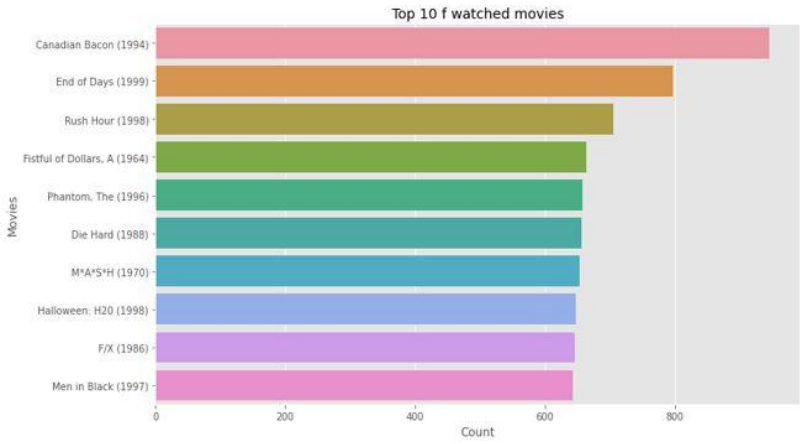


Figure 9

5.10.2 Gender Group:



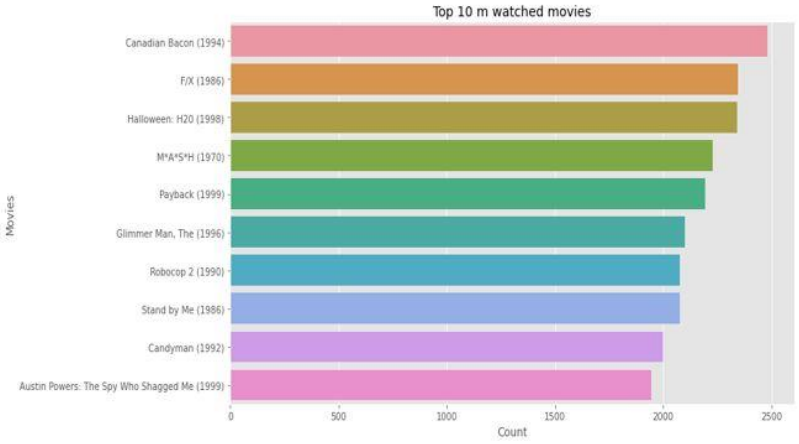


Figure 10

5.10.3 Age Group:

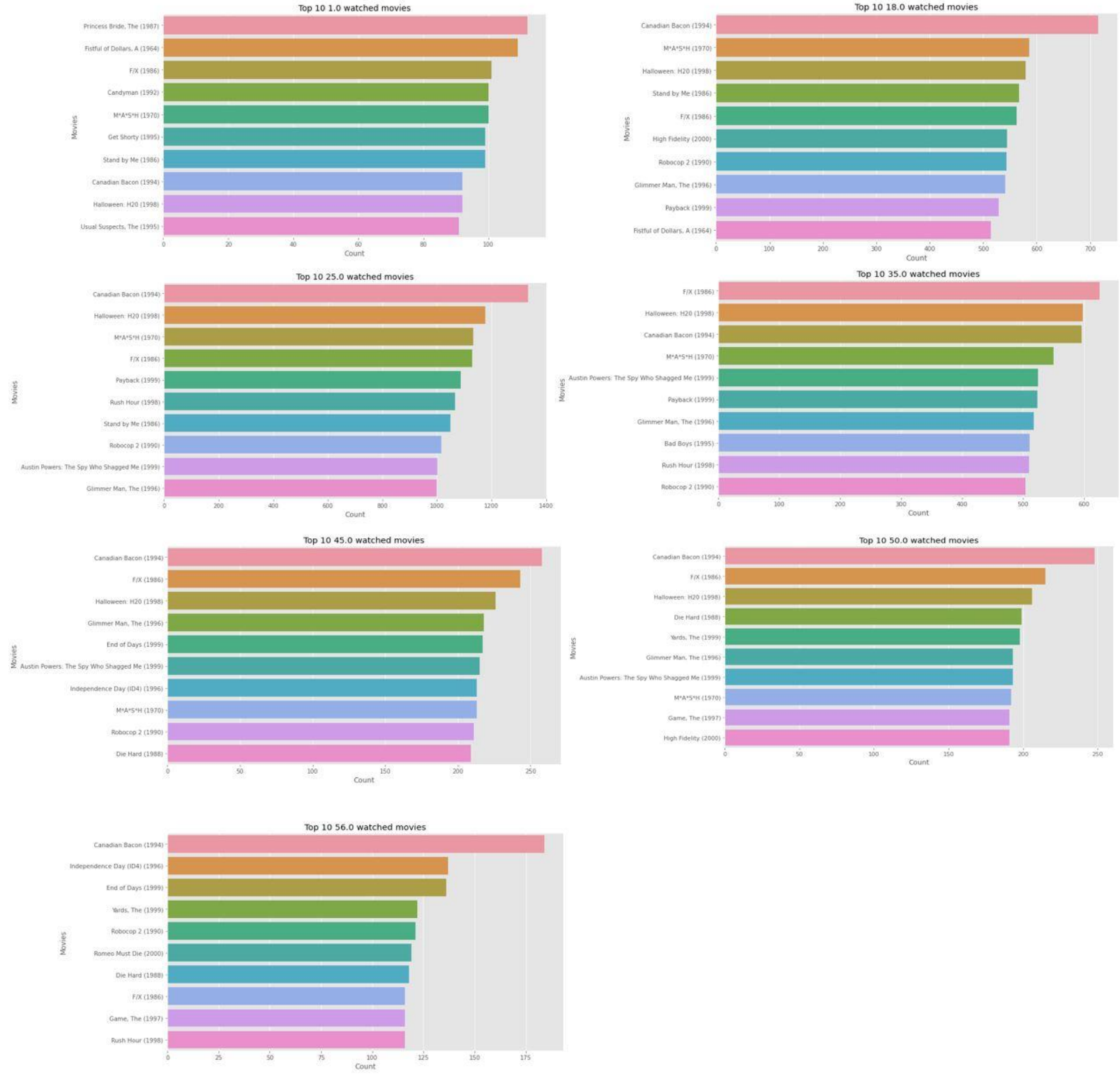


Figure 11

5.11 What are the most loved Movies? In terms of:

5.11.1 All Time:

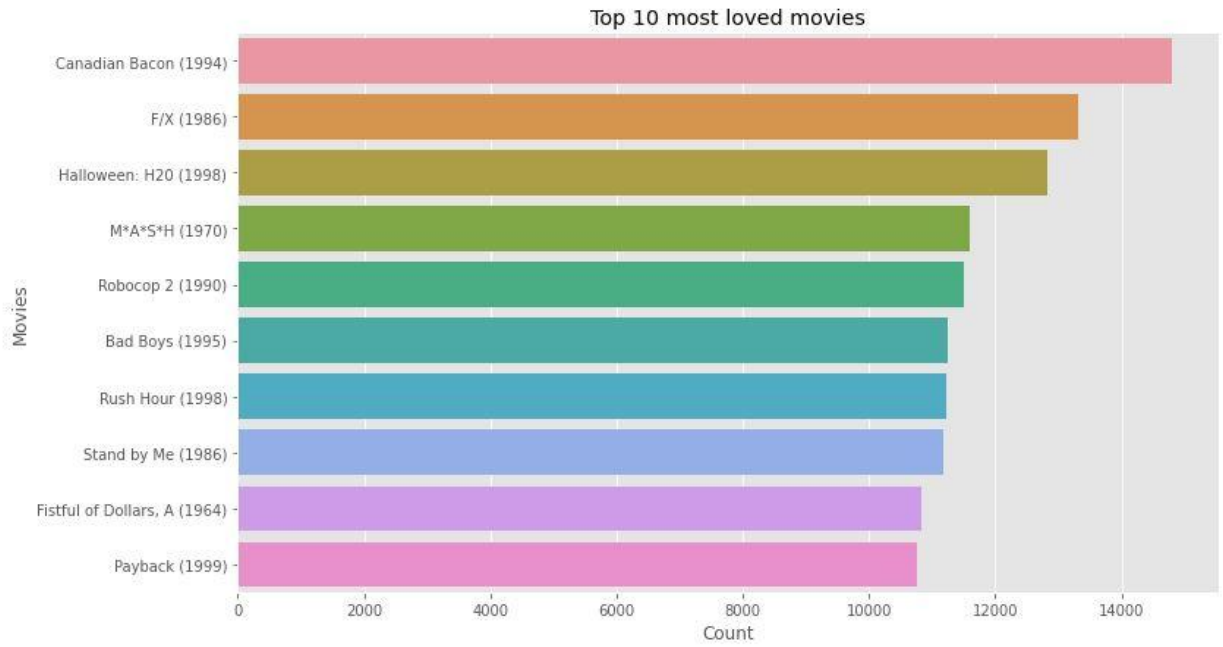


Figure 12

5.11.2 Gender Group:

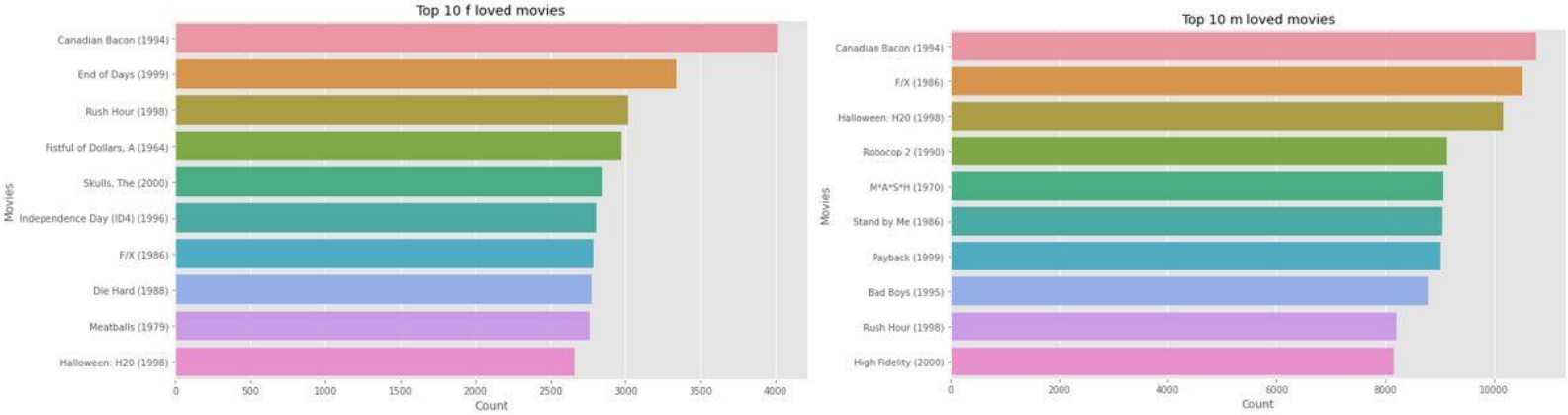


Figure 13

5.11.3 Age Group:

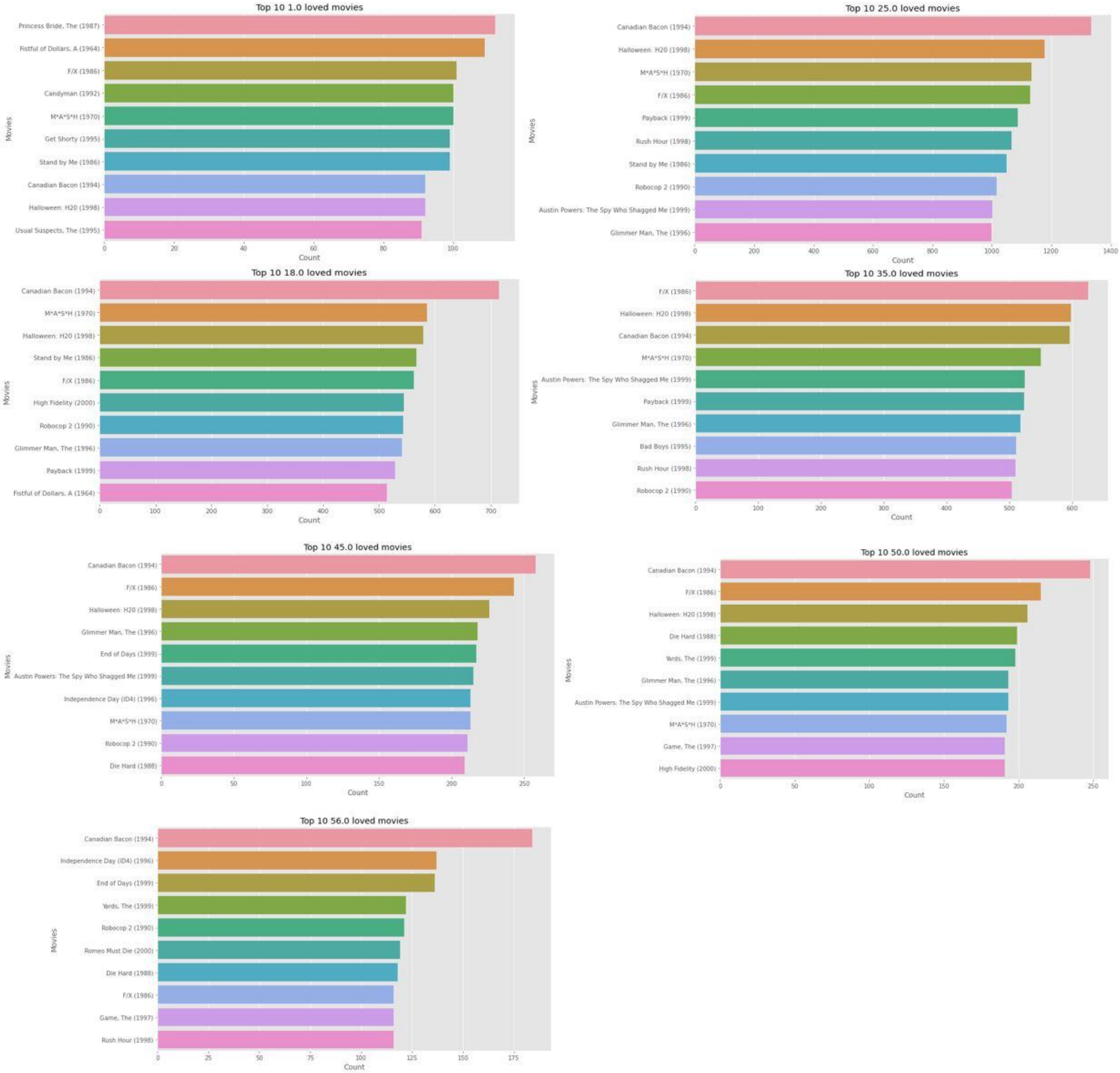


Figure 14

5.12 Which year the users were interested the most to rate/watch movies?

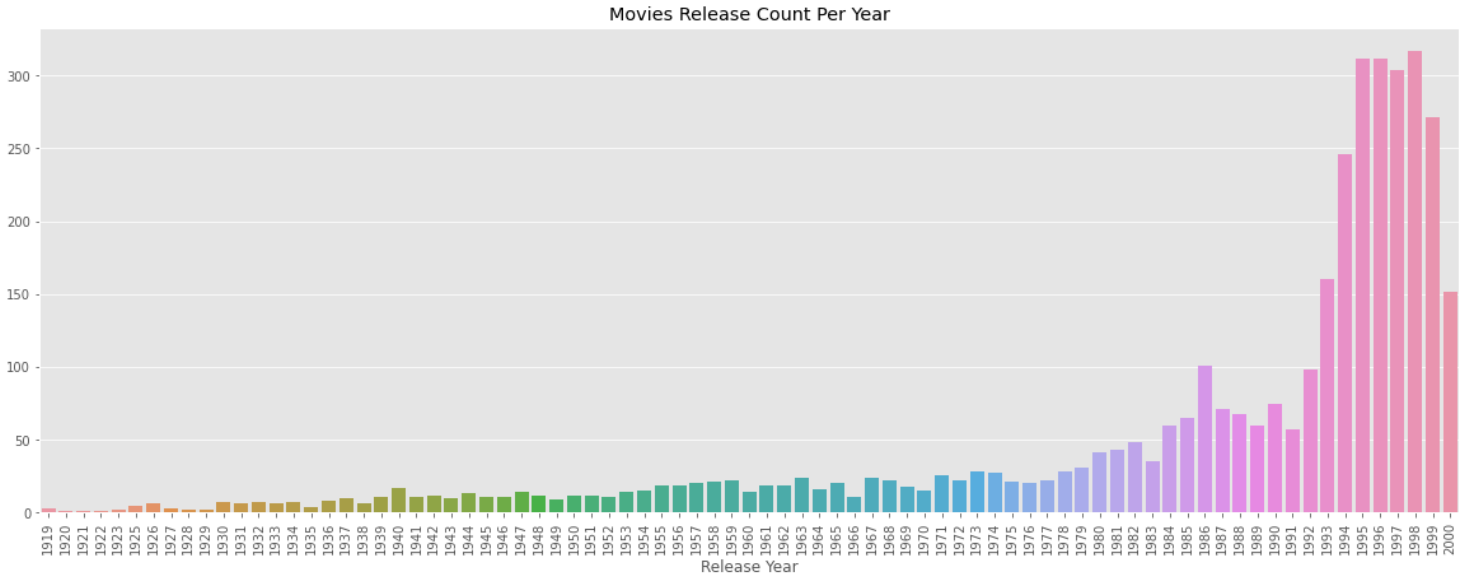


Figure 15

Alright, looks like our users were mainly interested in rating/watching the 90s Movies.

5.13 What are the worst movies per rating?

To Answer this question, we'll be using **wordcloud package**. Documentations can be found **here.**

Figure 16

5.14 Is there any relations between the user rating and location?

To answer this questions, let's do the following:

- We'll be using **folium Library,** due to its impressive capability of interactive visualization. More details about folium and how to install check here.
- Let's create Two Dataframes --> **One**: Avg user rating per Zip Code and **Two:** Takes user occupants per Zipcodes.
- For folium to work without error, zipcode needs to be in a string format.
- Try to limit the number of zipcodes to avoid any memory errors, in our case we'll be taking only CA Zipcodes which are available in the Movie lens 1M Dataset.
- Also, we'll be using external databases:
    - To get the coordinate of the zipcodes --> This is needed for to highlight the zipcodes as labels in the map.
    - To get a geoJSON file which has coordinates of zipcode --> This is needed to map the zip codes boundaries.

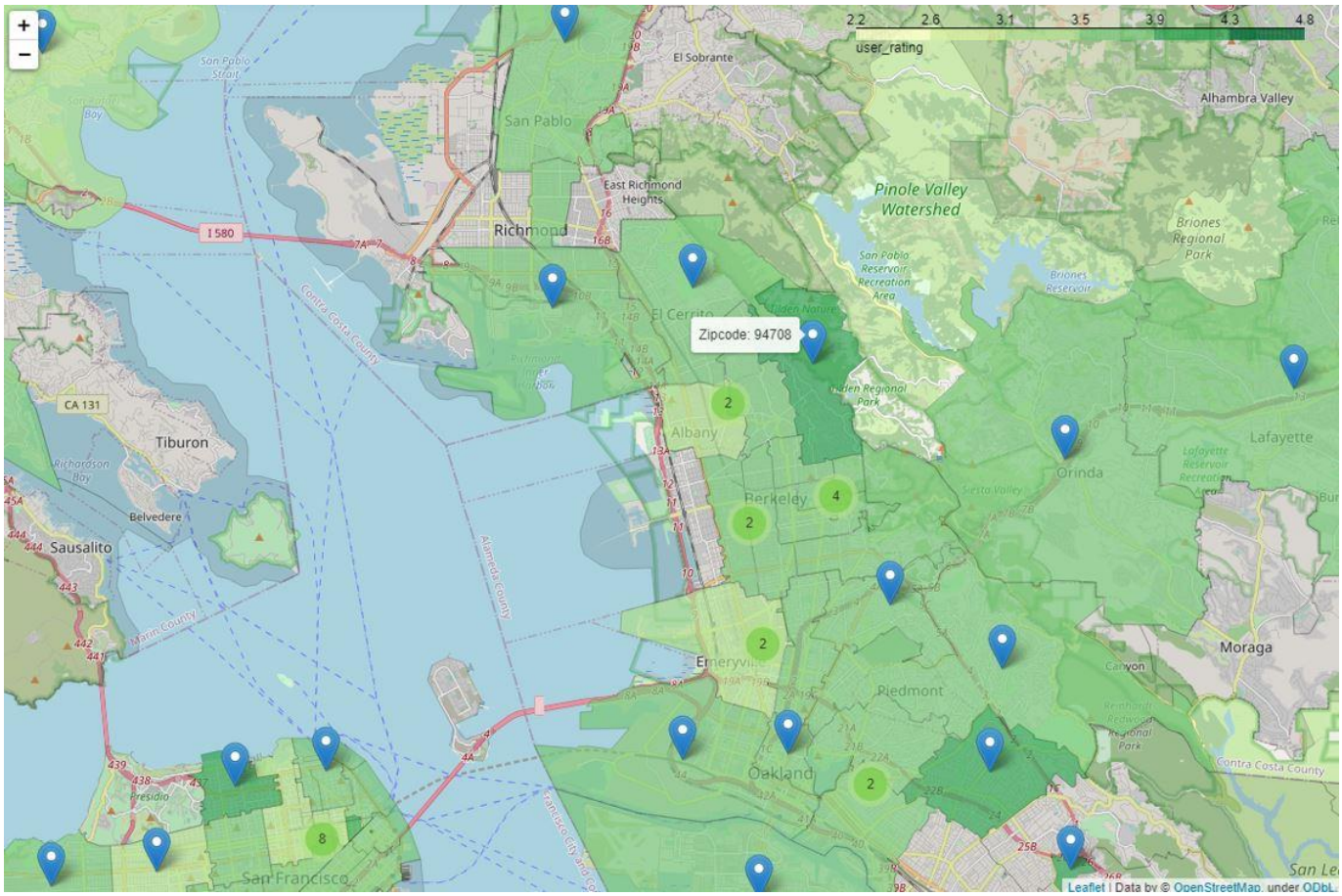


Figure 17 Movies user rating mapped to zipcode using folium

Alright, looks like the most high rates zipcodes are located in Universities area ... so let's drill down to see and confirm that if the students are the group who's causing such high rates. For this let's create a seond dataframe that takes user occupants per Zipcodes:

		Occupants_count
user_zip_code	user_occupation_text	
231	other/not specified	78
606	farmer	195
681	K-12 student	69
693	technician/engineer	139
918	clerical/admin	146
...	...	...
99703	homemaker	57
99709	technician/engineer	89
99801	other/not specified	41
99826	tradesman/craftsman	32
99945	executive/managerial	20

Table 5 Occupants count per zipcode and user occupation

	user_zip_code	user_occupation_text	Occupants_count
4863	94708	college/grad student	104
4864	94708	writer	104

Table 6 UC Berkeley Zipcode Movie Lens data

As we expected, looks like, student are the group driving the rating, so Yes location has an influence on the rating!!

5.15 What's the most popular Genre in our dataset?

To answer we need to transform existing movie_genre feature from list of strings to list of list because Pandas read the lists (genres) as a string which is a problem because we cannot even loop the lists to count the genres, so in this case we used **literal_eval library from ast (Abstract Syntax Tree)** to help Python to process trees of the Python abstract syntax grammar, as shown below:

```
for i, l in enumerate(final["movie_genres"]):
    print("list",i,"is",type(l))
```

```
list 0 is <class 'str'>
list 1 is <class 'str'>
list 2 is <class 'str'>
list 3 is <class 'str'>
list 4 is <class 'str'>
list 5 is <class 'str'>
list 6 is <class 'str'>
list 7 is <class 'str'>
list 8 is <class 'str'>
list 9 is <class 'str'>
list 10 is <class 'str'>
list 11 is <class 'str'>
list 12 is <class 'str'>
list 13 is <class 'str'>
```

```
for i, l in enumerate(final["movie_genres"]):
    print("list",i,"is",type(l))
```

```
list 0 is <class 'list'>
list 1 is <class 'list'>
list 2 is <class 'list'>
list 3 is <class 'list'>
list 4 is <class 'list'>
list 5 is <class 'list'>
list 6 is <class 'list'>
list 7 is <class 'list'>
list 8 is <class 'list'>
list 9 is <class 'list'>
list 10 is <class 'list'>
list 11 is <class 'list'>
list 12 is <class 'list'>
list 13 is <class 'list'>
```

Figure 18 literal_eval from AST

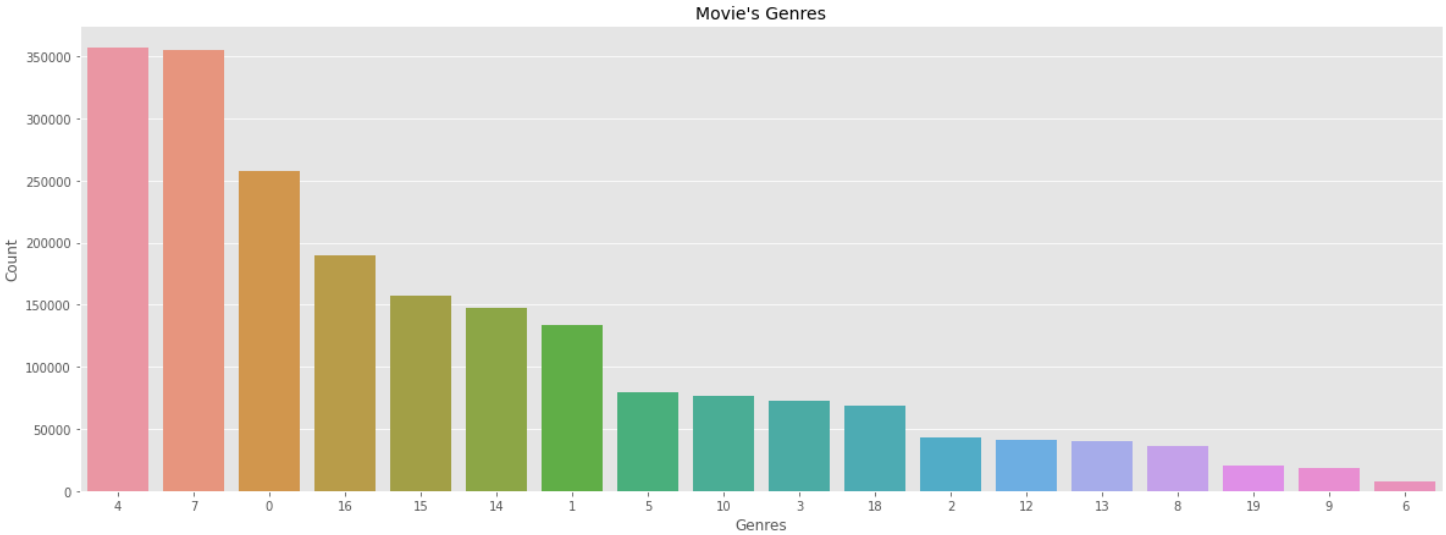


Figure 19

From above we can see that the most popular Genres are Comedy and Drama followed by Action:

- "movie_genres": The Genres of the movies are classified into 21 different classes as below:
  - 4: Comedy, 7: Drama, 0: Action ..etc.

5.16 The top 15 busiest (Famous!) Directors?

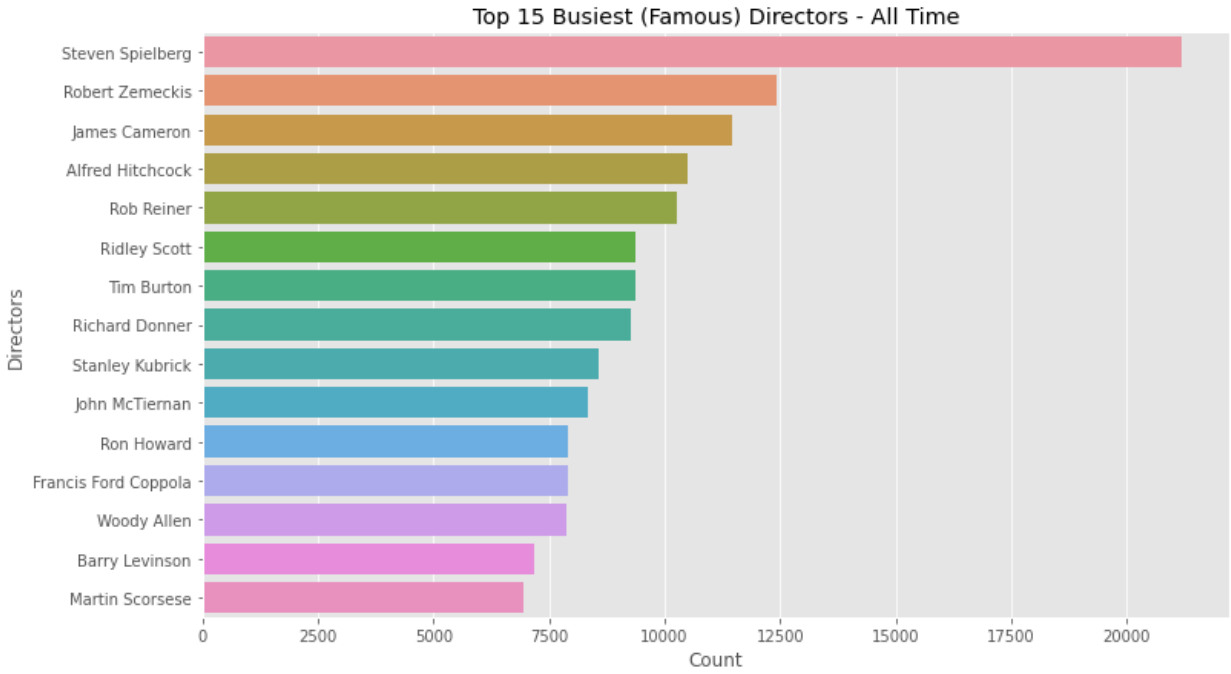


Figure 20

5.17 The top 15 busiest (Famous!) Actors?

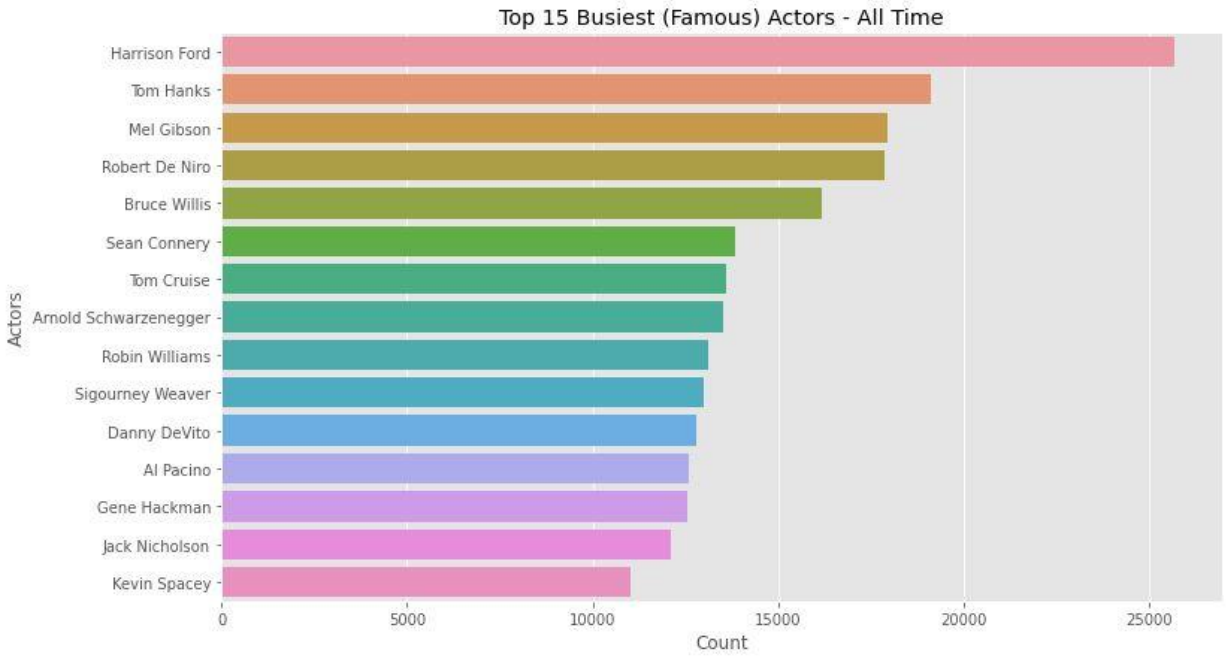


Figure 21

5.18 The top 15 largest cast Movies – All Time

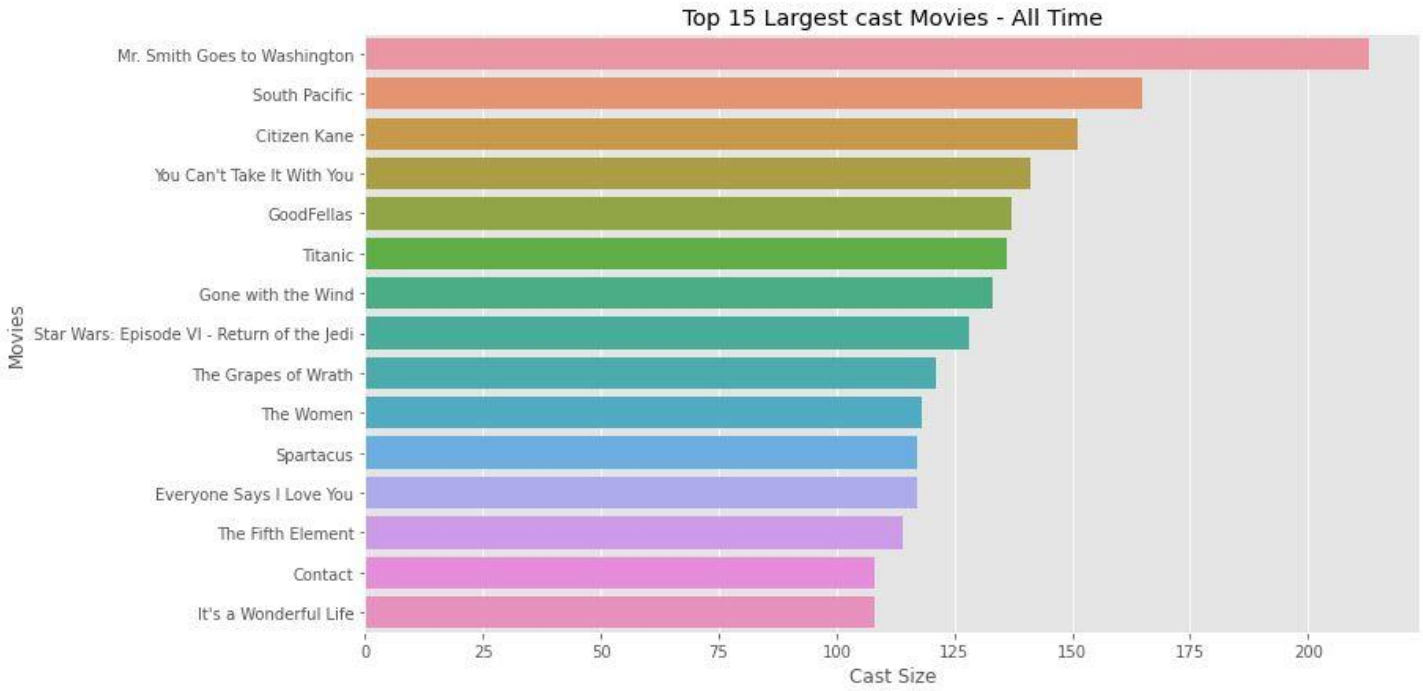


Figure 22

5.19 The top 15 largest crew Movies – All Time

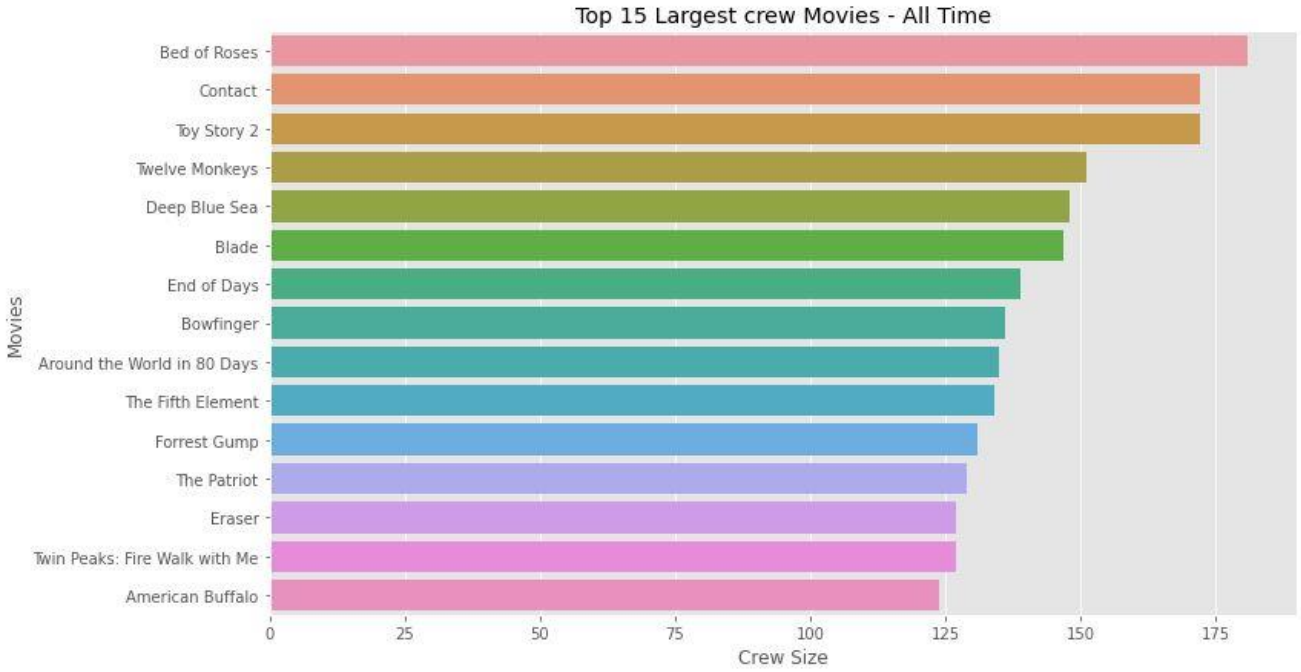


Figure 23

6. Machine Learning Modeling: TensorFlow Recommenders (TFRS)

6.1 Introduction

Here comes the really fun step: modeling! For this step, we'll be:

- Feature importance using Deep and Cross Network (DCN-v2)
- Training multiple TensorFlow Recommenders.
- Apply hyperparameters tuning where applicable to ensure every algorithm will result in best prediction possible.
- Finally, evaluate these Models.

TensorFlow Recommenders (TFRS):

- Deep and Cross Network (DCN-v2) For feature Importance
- The Two-Tower Model (Retrieval Model)
- The Ranking Model
- Multi-Task Model (Joint Model: Retrieval + Ranking)

6.2 Features Importance Using Deep & Cross Network (DCN-V2)

Deep and cross network, short for DCN, came out of Google Research, and is designed to learn explicit and bounded-degree cross features effectively:

- Large and sparse feature space is extremely hard to train.
- Oftentimes, we needed to do a lot of manual feature engineering, including designing cross features, which is very challenging and less effective.
- Whilst possible to use additional neural networks under such circumstances, it's not the most efficient approach.

Deep and cross network (DCN) is specifically designed to tackle all above challenges.

6.2.1 Feature Cross

Let's say we're building a recommender system to sell a blender to customers. Then our customers' past purchase history, such as purchased bananas and purchased cooking books, or geographic features are single features. If one has purchased both bananas and cooking books, then this customer will be more likely to click on the recommended blender. The combination of purchased bananas and the purchased cooking books is referred to as feature cross, which provides additional interaction information beyond the individual features.

6.2.2 Cross Network

In real world recommendation systems, we often have large and sparse feature space. So, identifying effective feature processes in this setting would often require manual feature engineering or exhaustive search, which is highly inefficient. To tackle this issue, **Google Research team has proposed Deep and Cross Network, DCN.**

It starts with an input layer, typically an embedding layer, followed by a cross network containing multiple cross layers that models explicitly feature interactions, and then combines with a deep network that models implicit feature interactions. The deep network is just a traditional multilayer construction. But the core of DCN is really the cross network. It explicitly applies feature crossing at each layer. And the highest polynomial degree increases with layer depth. The figure here shows the deep and cross layer in the mathematical form.

6.2.3 Deep & Cross Network

There are a couple of ways to combine the cross network and the deep network:

- Stack the deep network on top of the cross network.
- Place deep & cross networks in parallel.

6.2.4 Model Structure

We first train a DCN model with a stacked structure, that is, the inputs are fed to a cross network followed by a deep network.

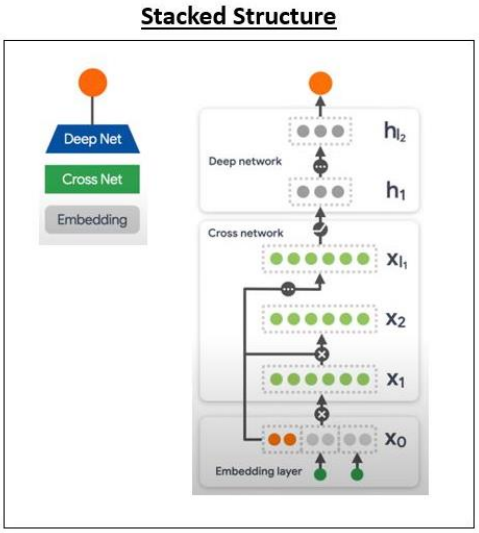


Figure 24

6.2.5 DCN-v2: Features Importance

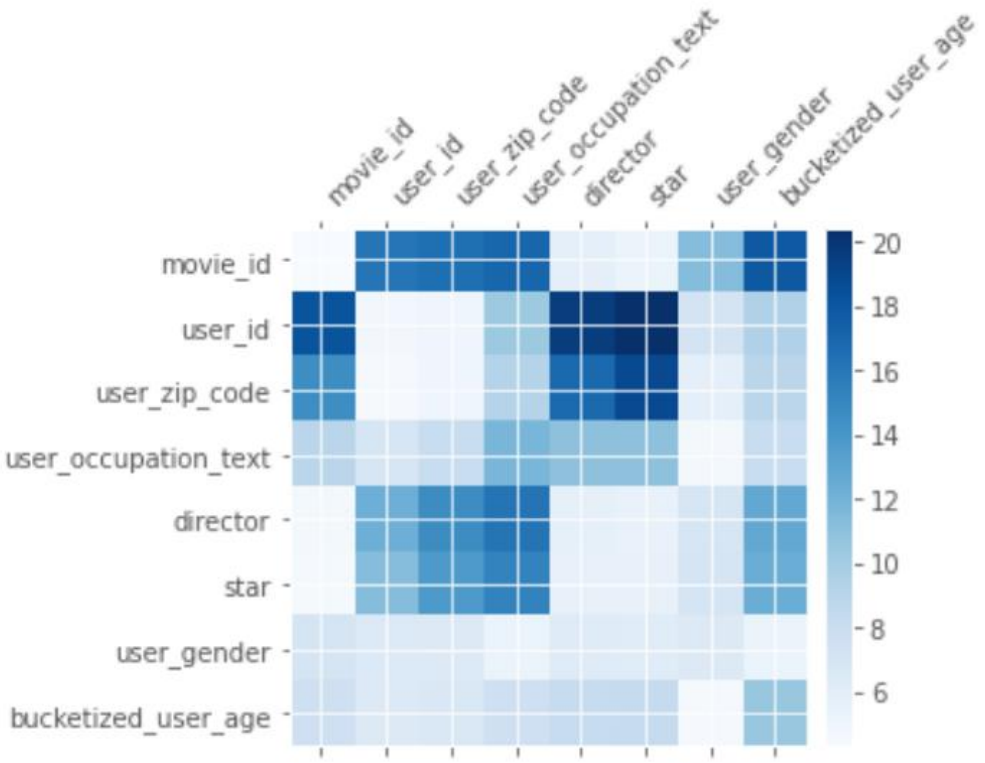


Figure 25 Movie Personal Dataset - Features Importance

One of the nice things about DCN is that we can visualize the weights from the cross network and see if it has successfully learned the important feature process.
As shown above, the stronger the interaction between two features is. In this case, the feature cross of user ID and movie ID, director, star are of great importance.

6.3 First Stage: Retrieval (The Two Towers Model)

Real-world recommender systems are often composed of two stages:

- The retrieval stage (Selects recommendation candidates): is responsible for selecting an initial set of hundreds of candidates from all possible candidates. The main objective of this model is to efficiently weed out all candidates that the user is not interested in. Because the retrieval model may be dealing with millions of candidates, it has to be computationally efficient.

- The ranking stage (Selects the best candidates and rank them): takes the outputs of the retrieval model and fine-tunes them to select the best possible handful of recommendations. Its task is to narrow down the set of items the user may be interested in to a shortlist of likely candidates.

Retrieval models are often composed of two sub-models:

The retrieval model embeds user ID's and movie ID's of rated movies into embedding layers of the same dimension:

- A query model computing the query representation (normally a fixed-dimensionality embedding vector) using query features.
- A candidate model computing the candidate representation (an equally-sized vector) using the candidate features.

As shown below, the two models are multiplied to create a query-candidate affinity scores for each rating during training. If the affinity score for the rating is higher than other for other candidates, then we can consider the model is a good one!

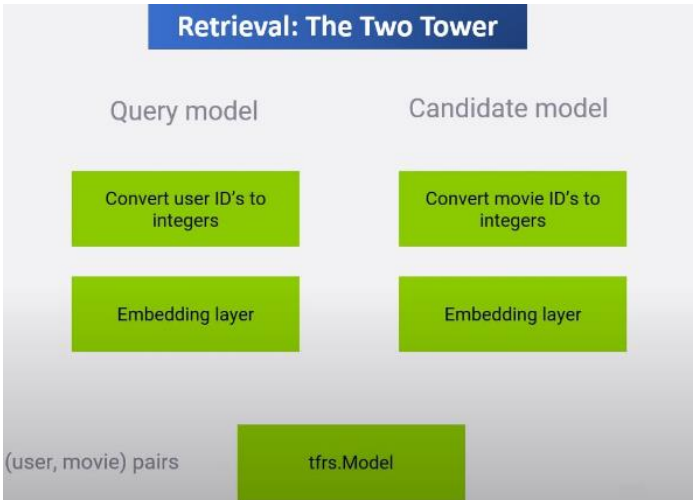


Figure 26

Embedding layer Magic:

As shown above, we might think of the ***embedding layers*** as just a way of encoding right a way of forcing the categorical data into some sort of a standard format that can be easily fed into a neural network and usually that's how it's used but embedding layers are more than that! The way they're working under the hood is every unique id is being mapped to a vector of n dimensions maybe it's 32 dimensions or 64 dimensions, but it's going to be like a vector of 32 floating point values and we can think of this as a position in a 32-dimensional space that represents the similarity between one user id and another or between one movie id and another so by using ***embedding layers*** in this way we're kind of getting around that whole problem of ***data sparsity*** and ***sparse vectors*** and at the same time, we're getting a measure of similarity out of the deal as well so it's a very simple and straightforward way of getting a recommendation candidates and then we could just brute force sort them all and get our top k recommendations.

The outputs of the two models are then multiplied together to give a query-candidate affinity score, with higher scores expressing a better match between the candidate and the query.

In this Model, we built and trained such a two-tower model using the Movielens dataset (1m Dataset):

- Get our data and split it into a training and test set.
- Implement a retrieval model.
- Fit and evaluate it.

Tuning Summary:

As we can see below, we managed to improve accuracy and reduce loss by:

- Increase embedding_dimension from 32-> 64
- keeping learning_rate 0.1

As a result, loss was reduced from 903.56 (Baseline) to 846.05 and top_10_accuracy was improved from 3.2% to 4.3%.

Tuning	top_1_accuracy	top_5_accuracy	top_10_accuracy	top_50_accuracy	top_100_accuracy	loss
Baseline Model: embedding_dimension 32 -> 64, learning_rate 0.1, epochs=5	0.002739	0.016660	0.032353	0.125389	0.207780	903.56
Model_1: embedding_dimension 32 -> 64, learning_rate 0.1 , epochs=5	0.002999	0.020739	0.039571	0.143078	0.231983	874.51
Model_2: embedding_dimension 64, learning_rate 0.01, epochs=5	0.003095	0.021155	0.040475	0.145901	0.235966	868.06
Model_3: embedding_dimension 64, learning_rate 0.001, epochs=5	0.002995	0.021155	0.040491	0.145969	0.236346	867.46
Model_4: embedding_dimension 64, learning_rate 0.1, epochs=32	0.003347	0.022155	0.043127	0.159742	0.258966	846.05

Table 7 Retrieval Model Tuning Summary

6.4 Second Stage: Ranking

The ranking stage takes the outputs of the retrieval model and fine-tunes them to select the best possible handful of recommendations. Its task is to narrow down the set of items the user may be interested in to a shortlist of likely candidates.

Tuning Summary

As shown below, we managed to reduce RMSE and loss from 93% to 86.89% and 81% to 64% respectively using below:

- Increased embedding_dimension from 32 ==> 64.
- Increased epochs from 3 ==> 32
- Increase Dense Layers from 2 ==> 4
- Adding Dropout + Adding Max Norm

Ranking Tuning	RMSE	loss
Baseline Model: 2 Dense Layers + embedding_dimension = 32 + epochs = 3	0.931866	0.810238
Model 1: 2 Dense Layers + embedding_dimension = 64 + epochs = 32	0.898368	0.853799
Model 2: Deeper: 3 Dense Layers + embedding_dimension = 64 + epochs = 32	0.873792	0.805714
Model 3: Deeper: 4 Dense Layers + embedding_dimension = 64 + epochs = 32	0.868916	0.65914
Model 4: Deeper: 4 Dense Layers + embedding_dimension = 64 + epochs = 32 + Adding Dropout + Adding Max Norm	0.866493	0.643070

Table 8 Ranking Model Tuning Summary

6.5 Multi-Task Model (Joint Model): The Two Tower + The Ranking Models

In the **TFRS Modeling - Two-Tower** we built a retrieval system using movie watches as positive interaction signals.

In many applications, however, there are multiple rich sources of feedback to draw upon. For example, an e-commerce site may record user visits to product pages (abundant, but relatively low signal), image clicks, adding to cart, and, finally, purchases. It may even record post-purchase signals such as reviews and returns.

Integrating all these different forms of feedback is critical to building systems that users love to use, and that do not optimize for any one metric at the expense of overall performance.

In addition, building a joint model for multiple tasks may produce better results than building a number of task-specific models. This is especially true where some data is abundant (for example, clicks), and some data is sparse (purchases, returns, manual reviews). In those scenarios, a joint model may be able to use representations learned from the abundant task to improve its predictions on the sparse task via a phenomenon known as transfer learning. For example, this paper shows that a model predicting explicit user ratings from sparse user surveys can be substantially improved by adding an auxiliary task that uses abundant click log data.

In this Model, we built a multi-objective recommender for Movielens, using both implicit (movie watches) and explicit signals (ratings).

Due to the important and the high possibility of this model:

- We'll focusing in using the important features as shown from DCN-v2 (Figure 25): "user_occupation_text", "user_gender", "director", "star", "bucketized_user_age",
- Normalize all Numerical Features: timestamps, user_age, user_gender.
- Reconfigured all Movie Lens Classes (TensorFlow Recommenders) to accommodate the new embedding design due to new features, having deeper neural networks and adding regularization to help overfitting:
    - class UserModel
    - class QueryModel
    - class MovieModel
    - class MovieModel
    - class CandidateModel
    - class MovielensModel

Tuning Summary

As shown below, Joint Model failed to beat baseline Joint model:

Models	Retrieval top-100 accuracy	Ranking RMSE
Model 1: Rating-specialized model	0.031	0.940
Model 2: Retrieval-specialized model	0.158	3.868
Model 3: Joint model	0.158	0.974
Model 4: Joint model (Tuned)	0.129	0.992

Table 9 Multi-Task Model Tuning Summary

Next Step:

As I mentioned, there's huge potential in this model before we give up, so we'll complete:

- More Hyperparameters fine tuning.
- Add the missing sequential features like movie_genres and age (Model failed), so there will be more research in this part.

6.6 TensorFlow Recommender (TFRS) overall Models Performance

As we can see below baseline managed to provide decent retrieval prediction with low RMSE Ranking.

The Two-Tower:

Tuning	top_1_accuracy	top_5_accuracy	top_10_accuracy	top_50_accuracy	top_100_accuracy	loss
Baseline Model: embedding_dimension 32 -> 64, learning_rate 0.1, epochs=5	0.002739	0.016660	0.032353	0.125389	0.207780	903.56
Model_1: embedding_dimension 32 -> 64, learning_rate 0.1 , epochs=5	0.002999	0.020739	0.039571	0.143078	0.231983	874.51
Model_2: embedding_dimension 64, learning_rate 0.01, epochs=5	0.003095	0.021155	0.040475	0.145901	0.235966	868.06
Model_3: embedding_dimension 64, learning_rate 0.001, epochs=5	0.002995	0.021155	0.040491	0.145969	0.236346	867.46
Model_4: embedding_dimension 64, learning_rate 0.1, epochs=32	0.003347	0.022155	0.043127	0.159742	0.258966	846.05

The Ranking

Ranking Tuning	RMSE	loss
Baseline Model: 2 Dense Layers + embedding_dimension = 32 + epochs = 3	0.931866	0.810238
Model 1: 2 Dense Layers + embedding_dimension = 64 + epochs = 32	0.898368	0.853799
Model 2: Deeper: 3 Dense Layers + embedding_dimension = 64 + epochs = 32	0.873792	0.805714
Model 3: Deeper: 4 Dense Layers + embedding_dimension = 64 + epochs = 32	0.868916	0.65914
Model 4: Deeper: 4 Dense Layers + embedding_dimension = 64 + epochs = 32 + Adding Dropout + Adding Max Norm	0.866493	0.643070

The Multi-Task Model:

Models	Retrieval top-100 accuracy	Ranking RMSE
Model 1: Rating-specialized model	0.031	0.940
Model 2: Retrieval-specialized model	0.158	3.868
Model 3: Joint model	0.158	0.974
Model 4: Joint model (Tuned)	0.129	0.992

7. Future Work

- More research is needed to optimize existing Tensorflow Recommenders Classes to accommodate the extra features created in this project
- More fine tune is needed in order a better retrieval prediction and low RMSE Ranking from the Joint Model (Multi-Task)
- Explore and check Listwise ranking and see the benefits if any.
- Building Maxilla App