Movie Recommendation System

GNR-652-PROJECT

Avyakta Wrat 180070010
Devank Rajvanshi 18D100009
Meeta Malviya 180070034

June 2020

Contents

1 Introduction

We have attempted to develop a movie recommendation system from scratch using the most basic approach (involving only simple mathematical concepts like correlation matrices and SVD) to relatively complex deep learning models.

The need of the hour today for businesses is to shift to a customer-oriented base from product-oriented. Digitization has aided in this process. Knowing the likes and dislikes of the customers become important in such scenarios. Recommendation Systems play a major role in today's accelerating business needs. With a better recommendation, a business is more likely to lure its customers into buying a product of their liking, which will accrue to a businesses' profits and its sales. Our goal was to study these systems and develop our own search engines with a high performance.

2 Types of Recommendation System

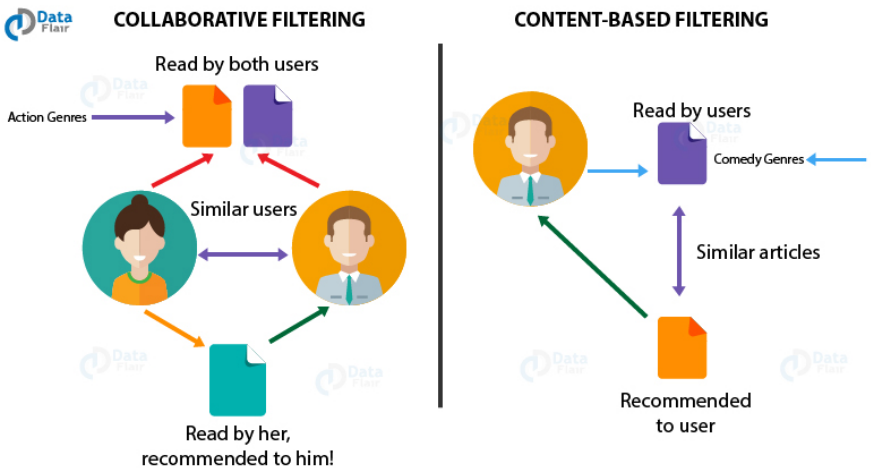


Figure 1: Types-of-Recommendation-Systems

2.1 Content-based filtering

The content-based recommendation systems focus on the attributes of the items and give users recommendations based on the similarity between the movie features like their genres, movie name, artists, directors etc.

2.2 Collaborative filtering

Collaborative filtering produces recommendations based on the knowledge of users' attitude to items. It is based on the similarity based in preferences, tastes and choices. These parameters are measured using user ratings.

There are 2 main types of memory-based collaborative filtering algorithms:

1. **User-User Collaborative Filtering:** Finding look alike users based on similarity and recommend movies which first user's look-alike has chosen in past. Computation of every user-pair is time consuming.

2. **Item-Item Collaborative Filtering:** Here, instead of finding a user's look-alike, we try finding a movie's look-alike. For a new user, this algorithm is less time consuming than user-user collaborative as we don't need all the similarity scores between users. And with a fixed number of movies, movie-movie look alike matrix is fixed over time.

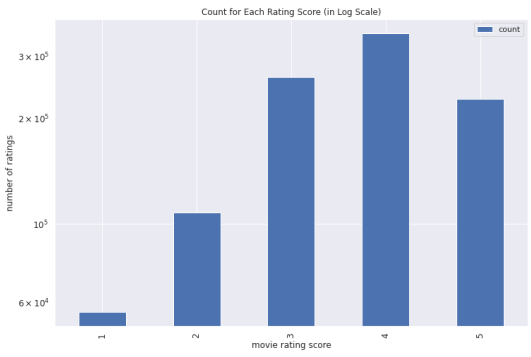
3 About Dataset

Data set of 1M ratings is used from **Movie lens.**
It contains around 1M ratings given by around 6k users on around 4k movies.
The data-set used contains :-

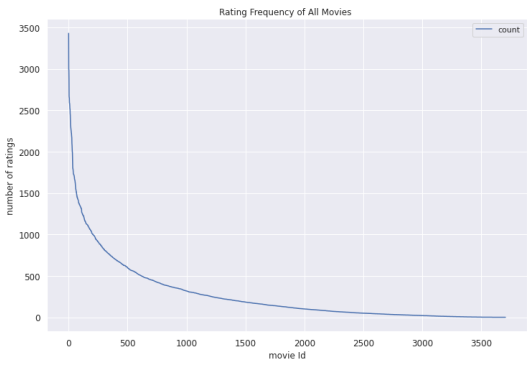
- Movies dataframe - containes 3883 movies with moviesID, title and genres.
- Users dataframe - containes 6040 users with userID, gender, zipcode, age.
- Ratings dataframe - containes 1000209 ratings with their userID, movieID and ratings.

4 Exploratory Data Analysis

1. We observe that users give 3 or 4 more than any other ratings.
2. There are few movies only which are highly rated. The data is **highly skewed.**
3. There are about 600 out of 4k movies that are rated more than 500 times.
4. There are 5 genre which have more than 400 movies and about 11 genres with less than 200 movies. Which shows that few genres are more preferred than others.

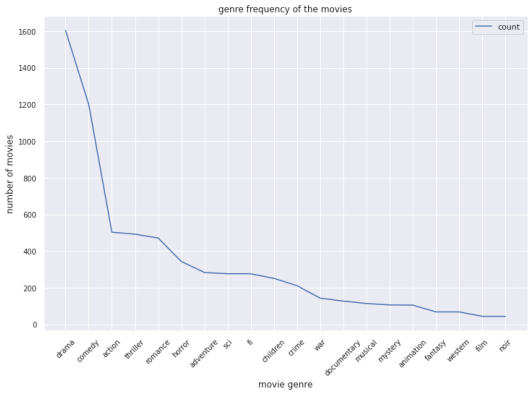


(a) number of rating



(b) rating frequency of movies

(a) rating frequency of movies in log



(b) rating freq of genres

5 Data Pre-processing

- A user-movie matrix containing the ratings given by user to each movie was created from the ratings.dat dataset. Users as rows and movies as columns. The empty fields were filled with 0.

```
#df_ratings_drop_users is obtained from ratings matrix after
    removing unpopular movies and non-active users.
movie_user_mat = df_ratings_drop_users.pivot(index='movieId',
    columns='userId', values='rating').fillna(0)
```

- **Outliers were Removed from movies dataframe.** Those movies which had less than 50 ratings were removed and subsequently those ratings which were given to the unpopular movies were removed.

```
popularity_thres = 50
popular_movies = list(set(df_movies_cnt.query('count >=
    @popularity_thres').index))
df_ratings_drop_movies = ratings[ratings.movieId.isin(
    popular_movies)]
```

- **Outliers were removed from ratings data frame.** Ratings given by non-active users, ie those users who had given less than 50 ratings were removed for better performance of the model.

```
ratings_thres = 50
active_users = list(set(df_users_cnt.query('count >=
    @ratings_thres').index))
df_ratings_drop_users = df_ratings_drop_movies[
    df_ratings_drop_movies.userId.isin(active_users)]
```

- The genres of the movies were transformed from text to a vector feature space using TF-IDF vectorizer. This stores the frequency of words used in the genre column.

```
 from sklearn.feature_extraction.text import TfidfVectorizer
tf = TfidfVectorizer(analyzer='word',ngram_range=(1, 2),min_df
    =0, stop_words='english')
tfidf_matrix = tf.fit_transform(movies['genres'])
```

6 Models

Each of the following sections and EDA are compiled in separate Jupyter Notebooks.

6.1 Content Based & Collaborative Filtering using Similarity Matrices

Choice of Similarity Measure

1. **Cosine Similarity:** This was used in content-based recommendation
    - Similarity is the cosine of the angle between the 2 vectors of the item vectors of A and B
    - Hence, lesser the angle, greater the similarity score
2. **Pearson Similarity**
    - Similarity is the pearson correlation coefficient

Mathematically, there are subtle differences between the above two, however with the user-user collaborative filtering, the latter showed better performance in terms of RMSE loss.

The reason for the above can be explained as follows:

1. If two users have rated 2 movies in common, pearson ignores the other movies and calculates similarity score. However, cosine accounts for the uncommon movies in its dimension by giving that vector component a value of 0 where it's unrated.

Content-Based Recommendation

A content based model was devised that suggested movies of same genres or similar title.

1. The genre of the movies (text) were processed using TF-IDF Vectorizer into feature vectors that can be used as input to estimator.
2. Calculating the Dot Product of these feature vectors will directly give us the Cosine Similarity Score. This is done using "linear_kernel" function from scikit library.
3. Based on the cosine similarity score, we defined a function "genre_recommendations" to recommend us the movies.

```
genre_recommendations('Toy Story (1995)')
```

```
1050              Aladdin and the King of Thieves (1996)
2072                           American Tail, An (1986)
2073          American Tail: Fievel Goes West, An (1991)
2285                           Rugrats Movie, The (1998)
2286                                Bug's Life, A (1998)
3045                                  Toy Story 2 (1999)
3542                               Saludos Amigos (1943)
3682                                  Chicken Run (2000)
3685       Adventures of Rocky and Bullwinkle, The (2000)
236                                Goofy Movie, A (1995)
12                                           Balto (1995)
241                              Gumby: The Movie (1995)
310                            Swan Princess, The (1994)
592                                    Pinocchio (1940)
612                               Aristocats, The (1970)
700                              Oliver & Company (1988)
876     Land Before Time III: The Time of the Great Gi...
1010          Winnie the Pooh and the Blustery Day (1968)
1012                        Sword in the Stone, The (1963)
1020                         Fox and the Hound, The (1981)
Name: title, dtype: object
```

Figure 4: content based output

It is useful for users having a unique taste, however it has major drawbacks **Drawbacks:**

1. It does not recommend movies outside a user's content profile

Collaborative Filtering

Results of Collaborative Filtering:

1. The ratings.dat file was first divided into train and test dataset and the corresponding user-movie matrix was built. Correlation matrices were found on train data and validated on test data.

2. Please note, this algorithm does not have any optimization process. The names "Train" and "Test" have been just used to distinguish seen and unseen data.

3. Training Error

   (a) RMSE loss on User Based CF: 2.89

   (b) RMSE loss on Item Based CF: 3.16

4. Testing Error

   (a) RMSE loss on User Based CF: 2.90

   (b) RMSE loss on Item Based CF: 3.16

5. The Errors are quite high, a mean difference of 2 to 3 ratings shows a very poor model. Hence, we looked into further advanced models to get better performance.

6.2 Singular Value Decomposition

This is **Collaborative Filtering** approach.
SVD was used to reduce dimensions of the features from the user-movie matrix. Sparsity of the dataset was found to be 95.5 %. A five fold cross-validation was used to evaluate the model built from "surprise" library.
The steps of SVD are mentioned below.

1. User Rating Mean is calculated and subtracted from user-movie matrix to get Ratings_demeaned.
2. Ratings_demeaned was singular value decomposed to find U, sigma and V. Value of k (the reduced dimension space) was tuned on the train dataset.
3. The SVD decreases the dimension of the utility matrix R by extracting its latent factors. It maps each user and each item into a k-dimensional latent space.
4. Matrix was reconstructed from decomposed matrices: $R_{mxn} = U_{mxk}\Sigma_{kxk}V^T_{kxn}$
5. The reconstructed matrix was used to suggest movies. It was sorted in order of ratings for the user, and the movies not already rated by the user was given as output.

Results of SVD:

1. SVD was implemented in two ways:
   (a) Manually finding SVD and reconstructing the matrix (with the help of scipy library)
   (b) Using the surprise library to predict recommendations

```
# Use the SVD algorithm.
svd = SVD()

# Compute the RMSE of the SVD algorithm.
cross_validate(svd, data, measures=['RMSE', 'MAE'], cv=5,
    verbose=True)
```

2. Recommendation system built from Surprise library gave better results
   (a) Test RMSE loss :0.8738 (Average of five-fold test)
3. A great reduction in RMSE was noted compared to previous models

6.3 KNN Clustering

This is a **item-item based collaborative filtering** approach.

What is KNN clustering?

The k-nearest neighbors (KNN) algorithm is a simple, easy-to-implement **supervised machine learning algorithm**. KNN captures the idea of similarity.
In KNN classification, the **output is a class membership**. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors.

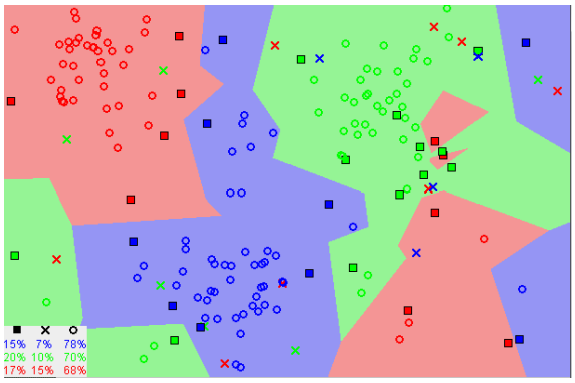


Figure 5: KNN

How model works-

In the model, we input a movie index its ratings and user info (which is a row from movie-row matrix). It then finds a 1D distance (via ratings) of the input movie to other rated movies by different users. The maximum distance movie i.e. most similar movie (due to cosine function) is given as output.

Steps to KNN based approach

1. The data is read in data frame as ratings, users and movies. These df's are processed as discribed in EDA section

2. The processed data is used to create a matrix(namely $movie\_user\_mat$) between moviesId and userId as rows and columns respectively. The values of the cell of matrix($movie\_user\_mat[i, j]$) is the rating given by j^th user on i^th movie. This matrix is transformed into scipy sparse matrix for easy computation.

3. A mapper(namely $movie\_to\_idx$) is a dictionary which is created, that maps movie to it's index according to movies dataframe.

4. The matrix is fed into NearestNeighbors model of sklearn. 'Cosine' similarity metric is used with brute algorithm.

```
# define model
model_knn = NearestNeighbors(metric='cosine', algorithm='brute'
    , n_neighbors=20, n_jobs=-1)
# fit
model_knn.fit(movie_user_mat_sparse)
```

5. *Fuzzy_matching* function takes in favorite movie as input and gives out index of most similar movie listed in mapper. The similarity is calculated via fuzz ratio.

```
for title, idx in mapper.items():
    ratio = fuzz.ratio(title.lower(), fav_movie.lower())
    if ratio >= 60:
        match_tuple.append((title, idx, ratio))
# sort
match_tuple = sorted(match_tuple, key=lambda x: x[2])[::-1]

```

6. In the function *make_recommendation* the data(i.e. the *movie_user_mat*) is fit in knn model, it then finds n nearest neighbours of data[idx], where idx given out by *Fuzzy_matching* function for favorite movie.
   The distance is sorted in top n neighbours with maximum distance(i.e. minimum angle as cosine similarity is used) is printed with movies name mapped via reverse mapping of movieID to movie.

```
 # fit
model_knn.fit(data)
idx = fuzzy_matching(mapper, fav_movie, verbose=True)

distances, indices = model_knn.kneighbors(data[idx],
   n_neighbors=n_recommendations+1)
# get list of raw idx of recommendations
raw_recommends = \
    sorted(list(zip(indices.squeeze().tolist(), distances.
   squeeze().tolist())), key=lambda x: x[1])[:0:-1]
```

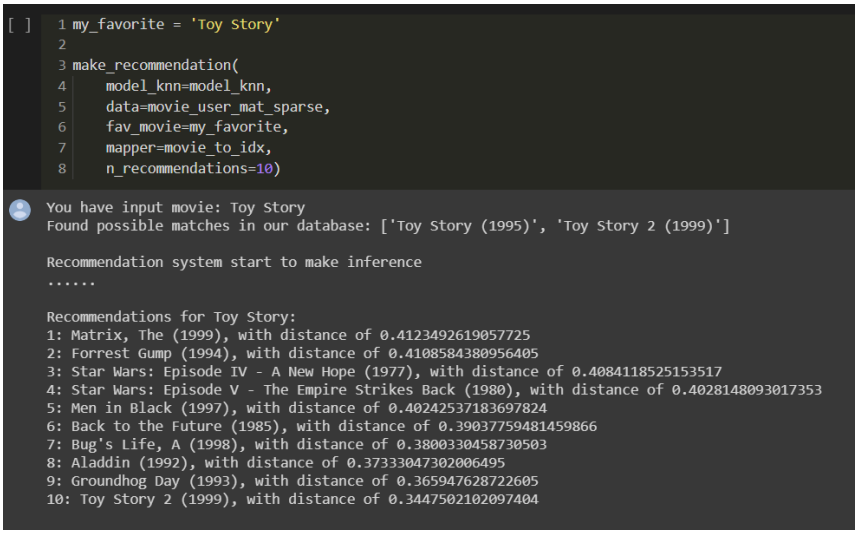


Figure 6: KNN

6.4 AutoEncoders

This is **user-user based Collaborative Filtering** approach.
The Auto Encoders were coded in pyTorch for our project.

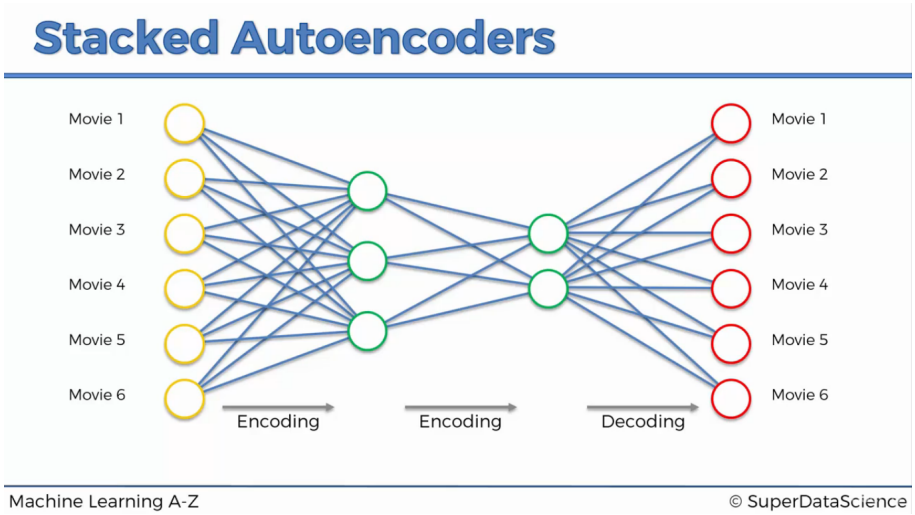


Figure 7: Stacked Autoencoders

What are Autoencoders?

An Autoencoder is an artificial neural network used to learn a representation (encoding) for a set of input data, usually to a achieve dimensionality reduction. The output layer has the same number of neurons as the input layer for the purpose of reconstructing it's own inputs. This makes an Autoencoder a form of unsupervised learning, which means no labelled data are necessary - only a set of input data instead of input-output pairs.

Training of an autoencoder

The input to the network was the user-movie matrix with a subset of rating used for training and the rest used in testing.

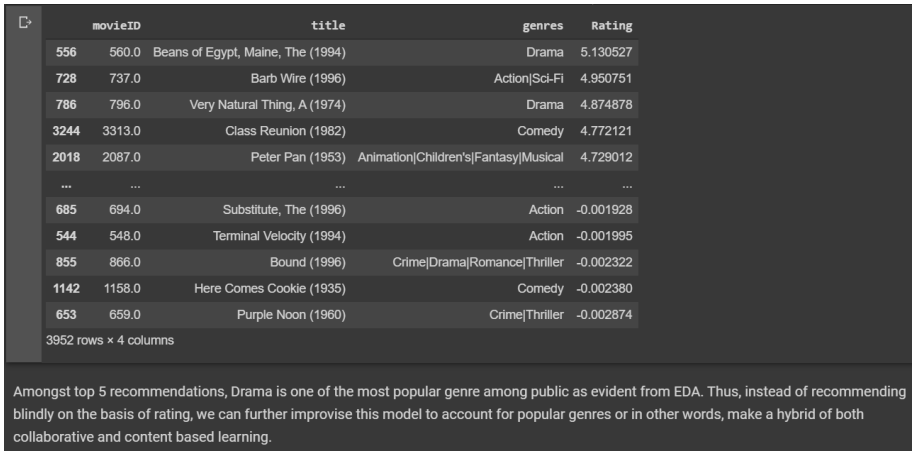
An exact re-creation of the input is not possible, we try to achieve a close approximation. Input of 0 implies the movie was not rated. Hence this is not counted in calculating the loss function. For this, the output from the autoencoders for the unrated movies are assigned 0. Mean squared error still accounts for the unrated movies in its denominator. This is corrected using a mean_corrector. Here is a summary of the optimization process.

1. In every epoch, all rows of user-movie matrix was passed.
2. Output from the autoencoder (NN model) is computed.
3. For unrated movies at the start, the output is assigned exact 0, followed by loss calculation.
4. A correction Factor is used in the loss function to ignore the unrated movies in the loss function.
5. Reducing the MSE is the optimization Goal
6. The loss is backpropogated and the weights updated.
7. A total of 200 epochs was run.

Results

This recommendation system is really good when there are few distinguishing parameters to movies. But this system lacks recommendation for new users as model doesn't know there taste ;)

1. Train RMSE loss : 0.8886
2. Test RMSE loss : 0.9231



	movieID	title	genres	Rating
556	560.0	Beans of Egypt, Maine, The (1994)	Drama	5.130527
728	737.0	Barb Wire (1996)	Action|Sci-Fi	4.950751
786	796.0	Very Natural Thing, A (1974)	Drama	4.874878
3244	3313.0	Class Reunion (1982)	Comedy	4.772121
2018	2087.0	Peter Pan (1953)	Animation|Children's|Fantasy|Musical	4.729012
...	...	...	...	...
685	694.0	Substitute, The (1996)	Action	-0.001928
544	548.0	Terminal Velocity (1994)	Action	-0.001995
855	866.0	Bound (1996)	Crime|Drama|Romance|Thriller	-0.002322
1142	1158.0	Here Comes Cookie (1935)	Comedy	-0.002380
653	659.0	Purple Noon (1960)	Crime|Thriller	-0.002874

3952 rows × 4 columns

Amongst top 5 recommendations, Drama is one of the most popular genre among public as evident from EDA. Thus, instead of recommending blindly on the basis of rating, we can further improvise this model to account for popular genres or in other words, make a hybrid of both collaborative and content based learning.

Figure 8: AutoEncoder Recommendation

6.5 Embeddings

This is a **Item-item based collaborative filtering approach.** In this approach, we try to map the movie ids to a latent space to form embeddings and then build a multi-layer perceptron classifier to learn predicting the movie ratings. **The model learns the embeddings via back-propagation.** The intuition behind using embeddings is that each component of the embedding can be thought of as a higher level representation of movie , for eg: different genres and their combinations. We build a model for each user. **The model will learn to predict the ratings of a user.** We think this approach performs better because the model will learn the features from the movies instead of manually deciding the features. Our model predicts well in the limit of ± 1.
Whenever we want to recommend a new movie to an existing user, we can pass that movie id to the user's model to estimate the rating. If that rating is high, we can recommend that movie to the user.

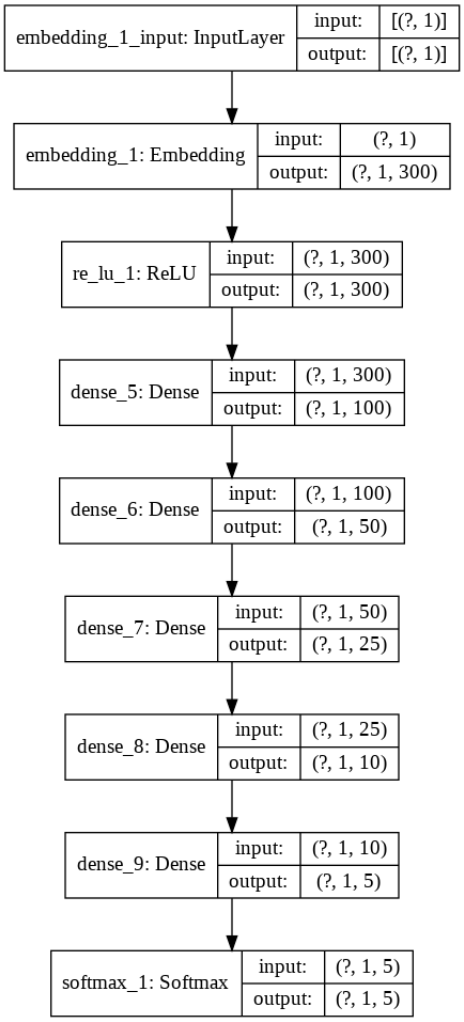


Figure 9: Embedding Model

The model was trained as follows:

1. A user is selected and the all the ratings given by him are extracted.
2. Each of these movie Ids is passed in to the model to get 5 output probabilities for each of the rating value of 1 to 5.
3. Cross Entropy loss was used as an optimization criterion and Adam optimizer with a learning rate of 0.0001 and weight decay of 1e-6 is used.
4. Backpropagation was used to compute the gradients and update the parameters.
5. The model was trained for 200 epochs.
6. PyTorch was used to train the model.

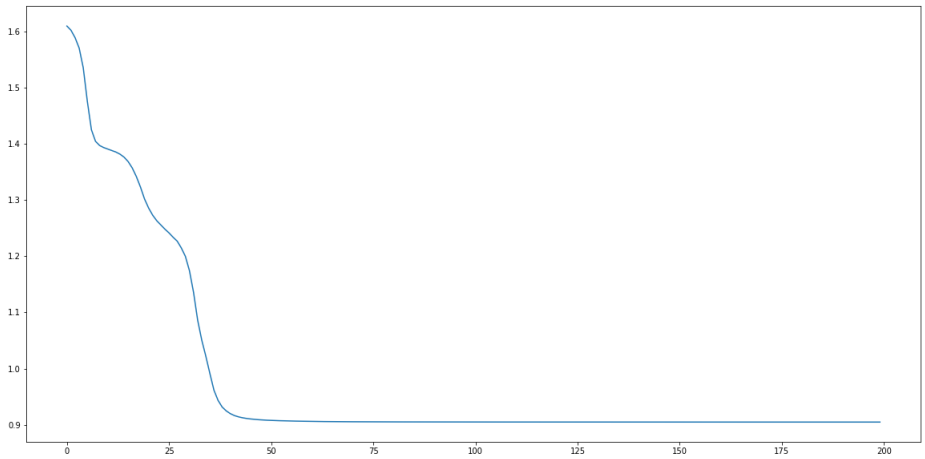


Figure 10: Loss vs iterations

Here is an working example prediction for a user:

```
test_inp = ratings['movieId'][ratings['userId'] != user_id]
test_inp = torch.tensor(test_inp.values, dtype=torch.long).cuda()
out = net(test_inp)
# print(out.shape)

idxs = []
for i in range(out.shape[0]):
    if torch.argmax(out[i]) >= 4:
        idxs.append(i)

# print(len(idxs))
movie_dict = {idx:title for idx, title in zip(movies['movieID'],
    movies['title'])}
# print(len(movie_dict))
```

```
recommended = []
for idx in idxs:
    try:
        recommended.append(movie_dict[idx])
    except:
        pass
from pprint import pprint
pprint(recommended[:10])
```

```
['Ace Ventura: When Nature Calls (1995)',
 'Big Green, The (1995)',
 'Two Bits (1995)',
 'Congo (1995)',
 'First Knight (1995)',
 'Johnny Mnemonic (1995)',
 'Nine Months (1995)',
 'Hideaway (1995)',
 'Ladybird Ladybird (1994)',
 'Shallow Grave (1994)']
```

Figure 11: Sample Output for user[idx=1]

7 Conclusion

Implementing various models, gave us an insight to the working of recommendation system. The common thing in all was the "search for similarity": to be close enough but not too much to get completely biased and spoil the test data-sets. Starting from finding similarity matrices to doing complex deep learning models involving auto-encoders and embedding, we had a long journey and a plethora of experiences.

Each model had a different interpretation for (Collaborative) 'filtering' and hence any recommendation made, largely depended upon how good data-set is and which recommendation model we use.

Another observation was real-life data is very skewed or biased and hence most popular item gets recommended to users even if they aren't of their taste.

This recommendation systems have become a major factor in trying/ buying new things. We always look for ratings and reviews given by other users before trying (take for instance shopping at amazon). And hence developers/ producers give focus on feedback as it forms a basis for attracting new customers who would have otherwise been very skeptical to try out a new online product.

8 Quick Links

Link to GitHub: *github.com/avyaktawrat/Evaluat-inator*
Link to Presentation: *Click Here*