# PROGRAMMING ASSIGNMENT 2
## CSE 6363 - MACHINE LEARNING

NAVYA R SOGI
1001753085

This assignment aims at building movie recommender system using KNN algorithm that was implemented from scratch in Programming Assignment 1.

## What is a Recommender System?

- Based on previous(past) behaviors, it predicts the likelihood that a user would prefer an item.
- For example, Netflix uses recommendation system. It suggest people new movies according to their past activities that are like watching and voting movies.
- The purpose of recommender systems is recommending new things that are not seen before from people.s

## Approach used to implement KNN algorithm as Recommender System

Item-based filtering approach has been used to implement KNN algorithm as a movie recommender system. This approach computed similarity between items calculated using user's ratings of those items. It is used in systems that have more users than items. Item-based models use rating distributions per item, not per user. Each item(movie) tends to have more ratings than each user, so an item's average rating doesn't change quickly. This leads to more stable distributions in the model, so the model doesn't have to be rebuilt as often. When users consume and rate an item, that item's similar items are picked from the existing system model and added to user's recommendations.

The dataset used for building recommender system is MovieLens Dataset that has 20M movie ratings, 465,000 tag applications applied to 27,000 movies by 138,000 users. Also includes tag genome data with 12 million relevance scores across 1,100 tags.

Since the recommender system is being built on user's ratings to movies, **movies.csv** and **ratings.csv** files are picked.

ratings.csv has:

- userId
- movieId
- rating
- Timestamp

movie.csv has:

- movieId
- genres
- title

The Python code to implement recommender system using KNN is programmed in the following way:

- The first step is to load movies.csv and ratings.csv files. In this movie.csv file, movieid and title columns are used and a new data frame **movies_df** is created. At the same time, userId, movieid and rating columns are used and a new data frame, **rating_df** is created.
- The two csv files are merged into one dataset **df**, based on **movieId** which is common in both the files.
- The **movie_ratingCount** is computed to know how many users have rated for each movie by grouping the fields title and rating from the merged data frame, counting and resetting the index to prevent title from being the index.
- Another data frame is created by merging userId, movieId, rating from ratings.csv with t **movie_ratingCount** data frame.
- A popularity threshold is figured out from observing the distribution of the number of ratings using **describe()** function. 50 is the value chosen and this indicates that whenever a rating count is greater than 50, only those movies are to be considered for recommendation purposes.
- **Matrix Factorization:** A pivot table, **movie_features_df** is created to represent a movie vs user matrix with movieId as indices, userId as columns and ratings as values. This matrix is then converted into data frame, **dataset** and array of vectors, so that it is scalable to be passed to our KNN algorithm implementation.
- In the previous implementation of KNN, distance between two vectors is calculated using Euclidean, Manhattan, Hamming distance measures. Using these, the K nearest neighbors are obtained based on the closest distance between the two vectors. If distance between two vectors is 0, then both vectors are the same. **euclidean_distance(), manhattan_distance(), hamming_distance()** functions implement these distance measures. **get_neighbors()** function locate the most similar neighbors based on the k value passed.
- **popularity_threshold** is used to obtain all the totalRatingCount of different users for a movie, because we want to recommend the movies which are the most popular based on user ratings.
- A train data and test data need to be passed to **get_neighbors()** function. Train data would be the flattened, scaled dataset, and test data would be a randomly generated movie obtained from **dataset**. The function returns calculated distances between the randomly generated test movie and the k-nearest neighbors to it and returns them as movie recommendations to the user based on his/her rating of the test movie.

## Matrix Factorization

The main assumption behind matrix factorization is that there exists a pretty low dimensional latent space of features in which we can represent both users and items and such that the interaction between a user and an item can be obtained by computing the dot product of corresponding dense vectors in that space.
For example, consider that we have a user-movie rating matrix. In order to model the interactions between users and movies, we can assume that:

- there exists some features describing (and telling apart) pretty well movies.
- these features can also be used to describe user preferences (high values for features the user likes, low values otherwise).

In this implementation, a pivot table, movie_features_df is created to represent a movie vs user matrix with movieId as indices, userId as columns and ratings as values. This matrix is then converted into data frame, dataset and array of vectors, so that it is scalable to be passed to our KNN algorithm implementation.

| | movieId | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... | 138484 | 138485 | 138486 | 138487 | 138488 | 138489 | 138490 | 138491 | 138492 | 138 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0.000 | 0.000 | 4.000 | 0.000 | 0.000 | 5.000 | 0.000 | 4.000 | 0.000 | ... | 0.000 | 0.000 | 5.000 | 0.000 | 3.000 | 0.000 | 0.000 | 2.000 | 0.000 | 3. |
| 1 | 2 | 3.500 | 0.000 | 0.000 | 0.000 | 3.000 | 0.000 | 0.000 | 0.000 | 0.000 | ... | 3.000 | 0.000 | 0.000 | 0.000 | 3.000 | 0.000 | 0.000 | 0.000 | 0.000 | 4. |
| 2 | 6 | 0.000 | 0.000 | 0.000 | 3.000 | 0.000 | 0.000 | 0.000 | 3.000 | 0.000 | ... | 5.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0. |
| 3 | 10 | 0.000 | 0.000 | 0.000 | 4.000 | 0.000 | 0.000 | 0.000 | 4.000 | 0.000 | ... | 3.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0. |
| 4 | 17 | 0.000 | 0.000 | 0.000 | 0.000 | 3.000 | 5.000 | 2.000 | 0.000 | 0.000 | ... | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 4.000 | 0.000 | 0.000 | 0. |

5 rows × 135726 columns

## The maximum dataset that the recommender system can use

The MovieLens dataset has 20 million ratings which is significantly a very huge dataset. Hence, we converted a movie-user matrix. Once the matrix has been factorized, we have less information to manipulate in order to make a new recommendation: we can simply multiply a user vector by any item vector in order to estimate the corresponding rating.
In this assignment, popularity_threshold is used to obtain all the totalRatingCount of different users for a movie, because we want to recommend the movies which are the most popular based on user ratings. Increasing this threshold decreases the number of rows in the matrix and vice versa. If the threshold is not used, the program throws an int32 Overflow error that indicates that it cannot use the entire dataset for building the recommender system.
After trying different values of popularity_threshold, it was found that a value of 20000 and above would generate (160, 135726) dimension matrix. Anything below this is not supported.

## The time complexity of the recommender system

Since the test data is compared with every row and every column in the movie vs user matrix or in other words, the train dataset which is the entire flattened movie vs user matrix, which is n*n, the time complexity of the recommender system built using collaborative filtering approach will be **O(n^2) in the best and worst cases where n is the number of rows and columns in the matrices.**

## The performance of the recommender system

To evaluate the performance of the recommender system, similarity between the random test movie selected and the recommended movies using pairwise correlation between two rows of a matrix and cosine similarity between two movie vectors.

**Cosine Similarity:** It measures the cosine of the angle between two vectors projected in a multi-dimensional space. **cosineSimilarity()** function has been used to get the similarity scores between the test movie and the movies obtained as recommendations based on user ratings given for test movie. The results of using cosine similarity measure are tabulated below:

**Movie Recommendations for Train of Life (Train de vie) (1998)**
Train of Life (Train de vie) (1998) was randomly chosen using **random()** function of numpy and passed to **get_neighbors()** function that returns the k-nearest neighbors or the movies with closest similarities as movie recommendations.

| Movie Recommendations | Similarity score |
|---|---|
| Train of Life (Train de vie) (1998) | 1.0 |
| Minority Report (2002) | 0.518954716360731 |
| Hard Word, The (2002) | 0.4183431982511069 |
| Heart of Glass (Herz aus Glas) (1976) | 0.4136765568191381 |
| Chunhyang (2000) | 0.4020336189347188 |
| Clockwise (1986) | 0.3947101648995718 |
| Lan Yu (2001) | 0.3828955193392565 |
| Hot Shoes (1991) | 0.36449068767161363 |
| An Amazing Couple (2002) | 0.3642237301629651 |
| Passion of Joan of Arc, The (Passion de Jeanne d'Arc, La) (1928) | 0.3640437970454177 |

**Pairwise correlation:** The corrwith() function of pandas gives correlations that are computed from all observations that have non missing values for any pair of variables. Suppose, the movie Star Wars: Episode IV - A New Hope (1977) is selected as our random test movie for which we obtain the ratings provided by different users. This is stored in a data frame **starWarsRatings.** This is now compared with the movie recommendations obtained, to obtain the pairwise correlations between the test movie (Star Wars: Episode IV - A New Hope (1977)) and each of the movie recommendations obtained based on the user's rating on Star Wars. The mean ratings for the movie rated by the users is calculated in a separate column **mean**. Ignore movies rated by less than 100 people and store it in a column called **size.** The similarity scores are then obtained as tabulated below:

| title | (rating, size) | (rating, mean) | similarity |
|---|---|---|---|
| Star Wars: Episode IV - A New Hope (1977) | 54502.0 | 4.190672 | 1.000000 |
| Star Wars: Episode V - The Empire Strikes Back (1980) | 45313.0 | 4.188202 | 0.751519 |
| Star Wars: Episode VI - Return of the Jedi (1983) | 46839.0 | 4.004622 | 0.687490 |
| Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark) (1981) | 43295.0 | 4.219009 | 0.472720 |
| Star Wars: Episode I - The Phantom Menace (1999) | 29574.0 | 3.080983 | 0.398518 |
| Lord of the Rings: The Fellowship of the Ring, The (2001) | 37553.0 | 4.137925 | 0.363507 |
| Lord of the Rings: The Two Towers, The (2002) | 33947.0 | 4.107521 | 0.363363 |
| Lord of the Rings: The Return of the King, The (2003) | 31577.0 | 4.142382 | 0.357534 |
| Indiana Jones and the Last Crusade (1989) | 31280.0 | 4.007593 | 0.354749 |
| Superman (1978) | 15089.0 | 3.398005 | 0.330981 |

## Is there a way to scale-up your recommender system to work with very large datasets?

- Following an "offline-online" architectural pattern can maximize data throughput while minimizing latency. Heavy data computation jobs that are not very sensitive to latency are processed offline. These jobs can be triggered periodically, or in response to user events. On the

other hand, jobs that depend on real-time signals are processed online by taking advantage of the previously computed results.

- A simple an effective for scaling-up recommender systems is to design filters that immediately filter out items such as a movie the user just watched. But, not everything can be solved with a simple filter. And, it is important to understand how things like filters can impact algorithms.
- Recommender system research suffers from a disconnect between the size of academic data sets and the scale of industrial production systems. In order to bridge that gap, we propose to generate large-scale user/item interaction data sets by expanding pre-existing public data sets. Our key contribution is a technique that expands user/item incidence matrices matrices to large numbers of rows (users), columns (items), and non-zero values (interactions).

## Files Attached:

- movieRecSystems.ipynb - Python program for building KNN based item-item recommender system
- ratings.csv
- movies.csv

## References:


- https://www.kaggle.com/ecemboluk/recommendation-system-with-cf-using-knn
- https://towardsdatascience.com/prototyping-a-recommender-system-step-by-step-part-1-knn-item-based-collaborative-filtering-637969614ea
- https://towardsdatascience.com/4-ways-to-supercharge-your-recommendation-system-aeac34678ce9
- https://www.kaggle.com/residentmario/notes-on-matrix-factorization-machines