# CS550: MovieLens Recommendation System.

Jaini Patel
jp1891
Rutgers University

Harsh Patel
hkp49
Rutgers University

Abhinav Madahar
am2229
Rutgers University

Chaitanya Vallabhaneni
cv346
Rutgers University

***Abstract*—Recommendation systems are an important part of many online platforms, especially for online streaming services. With the regular increase in the number of movies, user often cannot decide which movie to watch. The purpose of this project is to build a recommendation system that helps users decide which movie to watch by recommending them movies based on their tastes and preferences. We used different filtering techniques, we compared their methods, and we compared their outputs to calculate the errors. We implemented cosine similarity using different sets of filters to perform content-based filtering and item-based collaborative filtering methods ($k$NN, SVD) and chose the one which was more focused. We used content-based filtering to make the application because it produced more accurate results.**

**Keywords:** Content-based Filtering, Item-Based Collaborative Filtering, Cosine Similarity, kNN algorithm, SVD, Hybrid method.

## I. Introduction

Nowadays, recommendation systems are used in a wide variety of applications, and they provide personalized online product or service recommendation for users. They can recommend movies, songs, and more. The recommendation engine tries to predict the rating or preference of the particular product for a given user and helps in improving the user experience while shopping. Companies use recommendation systems to encourage their users to use more of their products; for example, Netflix recommends movies to its users so that they will spend more time on Netflix. The data generated from the recommendation engines can be analyzed to take strategic decisions by the marketing team of a company. For this project, we built a recommendation model that suggests the best movies for a user based on a user-supplied movie. We considered two techniques for recommendation systems: content-based filtering and collaborative filtering.

## II. Data Collection and Preprocessing

The first step in building the recommendation system was to collect the data and preprocess it. We used the Full MovieLens data set, which has data from 45,000 movies and 270,000 users, spanning 26 million ratings. It has been used extensively before as a reference data set.

## III. Algorithms

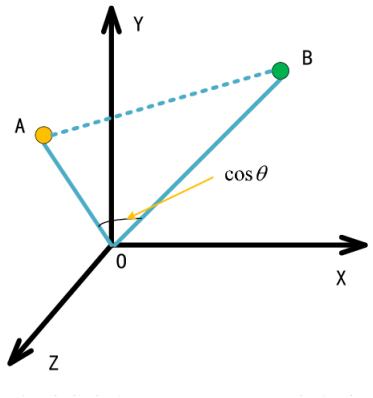
To implement content-based and item-based approach, we used these algorithms for finding similarity.

### A. Content-Based Filtering Method

**Cosine Similarity Approach:** Cosine similarity is a measure of similarity between two vectors. We find it by computing the cosine of the angle $\theta$ between the two vectors. It can be applied to items available on a data set to compute similarity to one another via keywords or other metrics. Similarity between two vectors is calculated by taking the dot product of the two vectors and dividing it by the magnitude value. Formally, we write that

$$\cos\theta = \frac{A \cdot B}{||A||\,||B||}.$$

The cosine similarity of two vectors increases as the angle between them decreases.




We applied two content-based recommendation systems using cosine similarity:

1) **Movie Overviews and Taglines - uses** *movies_metadata* **file.** This system works on the overview and tagline attributes from the movies metadata file and considers those as description of the movie. This description is then used passed to TF-IDF encoding scheme to find the weights of the words in each of the description. These encoded movie vectors

```
tf = TfidfVectorizer(analyzer='word',ngram_range=(1, 2),min_df=0, stop_words='english')
tfidf_matrix = tf.fit_transform(smd['description'])
```

are then passed to the *linear_kernel* function to find the cosine similarities as this function works faster for large matrix. The obtained cosine similarity matrix is later used to recommend the movie to the user using the following function:

```
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)
# cosine_sim.shape
```

```
def get_recommendations(title):
    idx = indices[title]
    if idx.size > 1:
        idx = idx[0]
    sim_scores = list(enumerate(cosine_sim[idx]))
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)
    sim_scores = sim_scores[1:31]
    movie_indices = [i[0] for i in sim_scores]
    return titles.iloc[movie_indices]
```

**Output: (Using Movies)**

```
get_recommendations('The Dark Knight').head(10)
```

```
18252                              The Dark Knight Rises
150                                      Batman Forever
1328                                     Batman Returns
21193    Batman Unmasked: The Psychology of the Dark Kn...
15511                          Batman: Under the Red Hood
20231              Batman: The Dark Knight Returns, Part 2
41973                          The Lego Batman Movie
585                                              Batman
25266                                Batman vs Dracula
18035                                 Batman: Year One
Name: title, dtype: object
```

2) **Movie Cast, Crew, Keywords and Genre.** This uses the credits file and the keywords file. It uses cast, crew, keywords and genre of the movies and makes a sample metadata for each movies with this information. This sample metadata is then used passed to Count Vectorizer to create a count matrix. The count matrix is passed to the cosine_similarity function to generate the cosine similarity matrix, which is further used in the same recommendation function which we used for the description-based recommender.

```
smd['soup'] = smd['keywords'] + smd['cast'] + smd['director'] + smd['genres']
smd['soup'] = smd['soup'].apply(lambda x: ' '.join(x))
```

```
count = CountVectorizer(analyzer='word',ngram_range=(1, 2),min_df=0, stop_words='english')
count_matrix = count.fit_transform(smd['soup'])
```

```
cosine_sim = cosine_similarity(count_matrix, count_matrix)
```

**Output: (Using Keywords)**

```
get_recommendations('The Dark Knight').head(10)
```

```
18442    The Dark Knight Rises
10210            Batman Begins
11463             The Prestige
26110                Doodlebug
26111                Doodlebug
2486                 Following
45843                  Dunkirk
5302                 Insomnia
15651                Inception
4126                   Memento
Name: title, dtype: object
```

**Further Work:** Further, we have used the generated cosine similarity matrix (using keywords) in improved_recommendation function, which sorts the most similar movies based on the weighted ratings of the movies. It uses vote_count, vote_averages, title and year from the movies_metadata file and recommends the movies on the basis of the similar weighted ratings.

```
def improved_recommendations(title):
    idx = indices[title]
    if idx.size > 1:
        idx = idx[0]
    sim_scores = list(enumerate(cosine_sim[idx]))
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)
    sim_scores = sim_scores[1:26]
    movie_indices = [i[0] for i in sim_scores]

    movies = smd.iloc[movie_indices][['title', 'vote_count', 'vote_average', 'year']]
    vote_counts = movies[movies['vote_count'].notnull()]['vote_count'].astype('int')
    vote_averages = movies[movies['vote_average'].notnull()]['vote_average'].astype('int')
    C = vote_averages.mean()
    m = vote_counts.quantile(0.60)
    qualified = movies[(movies['vote_count'] >= m) & (movies['vote_count'].notnull()) & (movies['vote_average'].notnull())]
    qualified['vote_count'] = qualified['vote_count'].astype('int')
    qualified['vote_average'] = qualified['vote_average'].astype('int')
    qualified['wr'] = qualified.apply(weighted_rating, axis=1)
    qualified = qualified.sort_values('wr', ascending=False).head(10)
    return qualified
```

**Output: (Using Keywords and Ratings)**

```
improved_recommendations('The Dark Knight')
```

| | title | vote_count | vote_average | year | wr |
|---|---|---|---|---|---|
| **15651** | Inception | 14075 | 8 | 2010 | 7.917588 |
| **23076** | Interstellar | 11187 | 8 | 2014 | 7.897107 |
| **11463** | The Prestige | 4510 | 8 | 2006 | 7.758148 |
| **4126** | Memento | 4168 | 8 | 2000 | 7.740175 |
| **18442** | The Dark Knight Rises | 9263 | 7 | 2012 | 6.921448 |
| **10210** | Batman Begins | 7511 | 7 | 2005 | 6.904127 |
| **45843** | Dunkirk | 2712 | 7 | 2017 | 6.757878 |
| **1349** | Batman Returns | 1706 | 6 | 1992 | 5.846862 |
| **31282** | Batman v Superman: Dawn of Justice | 7189 | 5 | 2016 | 5.013943 |
| **1511** | Batman & Robin | 1447 | 4 | 1997 | 4.287233 |

### *B. Item-Based Collaborative Filtering Method:*

- **KNN Algorithm:** In this recommendation system, K-Nearest Neighbors algorithm is used and the items will be clustered based on rating of item given by users and item will be recommends based on similar items.
  KNN is a model that classifies data points based on the points that are most similar to it. KNN works as it uses the deeply rooted mathematical theories it. When implementing KNN, the first step is to transform data points into feature vectors, or their mathematical value. The algorithm then works by finding the distance between the mathematical values of these points. The most common way to find this distance is the Euclidean distance:

$$d(\mathbf{p},\mathbf{q}) = d(\mathbf{q},\mathbf{p}) = \sqrt{(q_1-p_1)^2+(q_2-p_2)^2+\cdots+(q_n-p_n)^2}$$
$$= \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}.$$

  KNN runs this formula to compute the distance between each data point and the test data. It then finds the probability of these points being similar to the test data and classifies it based on which points share the highest probabilities.

- **SVD Approach:** The factorization of a matrix into three matrices is known as singular vector decomposition. Here we consider a matrix A as a transformation that acts on a vector x by multiplication to produce a new vector Ax. The SVD of an $m*n$ matrix A with real values is a factorization of A as $U\sum V^T$, where U

is an $m*m$ orthogonal matrix, V is a $n*n$ orthogonal matrix, and $\sum$ is a diagonal matrix with non-negative real entries on the diagonal.

**Key terms:**

1) span - basically gives us the entire range of possible values that you can get by multiplying each vector in a set of vectors by any real coefficient and then adding these values up (each of these combinations is called a linear combination).
2) Orthogonal - means perpendicular but in multi-dimensional space. In the movie-lens data we have key features such as $User\_Id$, $Movie\_Id$ and Ratings. Our task is to sort the movie recommendations based on these three traits.

These features particularly create noise in the data. SVD here identifies a relevant sub-space of all the dimensions created by the noise in the data. The SVD finds an optimal line in multi-dimensional space that allows it to classify the data in the simplest way. Here, n = 3 total number of dimensions, the SVD algorithm will best fit in n-1 dimensions. Then in n-2, n-3. As the value of n goes down, SVD moves from weaker approximations to stronger approximations of the data.
The singular values go down with each dimension, which tells us that each dimension is adding less and less value. Our goal is to stop adding dimensions to our approximation when the singular values become relatively negligible.
In the implementation, we have used sk-learn library SVD method to, implement our movie recommendations based on the $User\_Id$, $Movie\_Id$ and Ratings.

**Error:**

```
Evaluating RMSE, MAE of algorithm SVD on 3 split(s).

                  Fold 1  Fold 2  Fold 3  Mean    Std
RMSE (testset)    0.8057  0.8053  0.8061  0.8057  0.0003
MAE (testset)     0.6104  0.6102  0.6106  0.6104  0.0002
Fit time          746.15  753.71  776.67  758.84  12.98
Test time         106.20  105.93  95.89   102.68  4.80

{'test_rmse': array([0.80571985, 0.80532833, 0.80614815]),
 'test_mae': array([0.6103733 , 0.61015395, 0.61057662]),
 'fit_time': (746.1520004272461, 753.7127070426941, 776.6682245731354),
 'test_time': (106.19903898239136, 105.93161129951477, 95.89438819885254)}
```

**Prediction:** This will be clearer in the hybrid model.

```
Prediction(uid=10, iid=3020, r_ui=4, est=4.125189161173455, details={'was_impossible': False})
```

Now, building a universal hybrid model recommend movies based on a particular user. Here, we have computed a cosine similarity matrix for the movie metadata namely, keywords, cast, directors, crew etc. and we have created a recommendation function which takes $user\_id$ and movie name as input and recommends similar movies to the particular user.

Here is a snippet of the function:

```
[80] ▷ ▸≡ M↓
    def hybrid_movie_recommendations(userId, title):
        idx = indices[title]
        tmdbId = id_map.loc[title]['id']
        movie_id = id_map.loc[title]['movieId']
        similarity_scores = list(enumerate(cosine_sim[int(idx)]))
        similarity_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)
        similarity_scores = similarity_scores[1:26]
        movie_indices = [i[0] for i in similarity_scores]
        movies = lmd.iloc[movie_indices][['title', 'vote_count', 'vote_average', 'year', 'id']]
        movies['est'] = movies['id'].apply(lambda x: svd.predict(userId, indices_map.loc[x]['movieId']).est)
        movies = movies.sort_values('est', ascending=False)
        return movies.head(30)
```

**Output:** Here is the output: (This recommendations are for $user\_id$ 3000 and movie The Godfather)

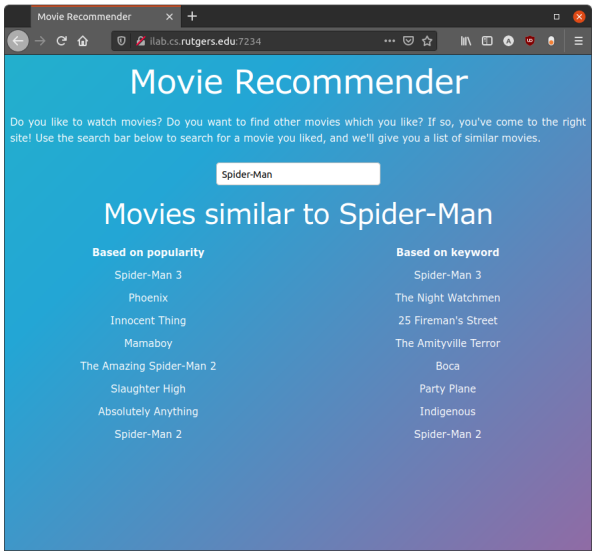
```
[81] ▷ ▸≡ M↓
hybrid_movie_recommendations(3000, 'The Godfather')        # this are the best possible recommendations based on a particular user
```

| | title | vote_count | vote_average | year | id | est |
|---|---|---|---|---|---|---|
| 1199 | The Godfather: Part II | 3418.0 | 8.3 | 1974 | 240 | 4.246934 |
| 1186 | Apocalypse Now | 2112.0 | 8.0 | 1979 | 28 | 4.102027 |
| 3636 | The Conversation | 377.0 | 7.5 | 1974 | 592 | 3.847925 |
| 13474 | Chinese Coffee | 16.0 | 6.3 | 2000 | 49511 | 3.674987 |
| 5309 | The Gambler | 21.0 | 7.0 | 1974 | 44880 | 3.593851 |
| 8911 | Rumble Fish | 141.0 | 6.8 | 1983 | 232 | 3.347481 |
| 1934 | The Godfather: Part III | 1589.0 | 7.1 | 1990 | 242 | 3.323440 |
| 1614 | The Rainmaker | 239.0 | 6.7 | 1997 | 11975 | 3.302125 |
| 4470 | Tucker: The Man and His Dream | 71.0 | 6.5 | 1988 | 28176 | 3.254798 |
| 1312 | Dracula | 1087.0 | 7.1 | 1992 | 6114 | 3.245694 |
| 14219 | Tetro | 49.0 | 6.7 | 2009 | 11928 | 3.241373 |
| 2025 | The Outsiders | 293.0 | 6.9 | 1983 | 227 | 3.180590 |
| 23540 | Hide in Plain Sight | 3.0 | 5.8 | 1980 | 48949 | 3.164419 |

The hybrid model utilizes the entire dataset and gives its recommendations.

## IV. Development

The website gives the user recommendations for movies based on a single movie. The image below is a screenshot of the website in use. Users can write the name of a movie, and the website will show different lists of recommendations. Each list is based on a slightly different source; one is for keywords and the other is for popularity.




### A. *Frontend*

The front-end for the website is built in HTML, CSS, and JavaScript. We chose not to use any frameworks like React.js or Vue.js because the website is so simple that we can write

the code more cleanly with vanilla JS. In total, the front-end comprises only 96 lines of code.

When the user writes a movie in the search box, the front-end sends an AJAX request to the back-end. The back-end exposes an endpoint, /recommendation, which accepts arguments for the query movie and the technique to use (e.g. keywords or popularity). After the back-end returns the recommendations list, the front-end displays the recommendations in a list. This way, we do not need to reload the entire page, which feels much more comfortable for the users.

We also added a gradient animation as a background for the website. Gradient backgrounds are in fashion now, and they will probably stay popular for a few years.

### *B. **Backend***

The back-end is written using Python with Flask. We tried writing a server in Django instead of Flask, but we found that the Django server delivered the same capabilities as the Flask server while being much more complicated and harder to understand. Our final server is contained in a single file of 101 lines of code.

It exposes a static directory to serve the front-end and a single endpoint, /recommendation. The recommendation endpoint gives recommendations using a cosine similarity matrix. Our cosine similarity matrix is over 20 GB large, so we do not store the entire matrix in memory. Instead, when the user asks for recommendations for a movie, we only read the line of the matrix file corresponding to that movie. This way, we avoid the massive overhead of loading the entire matrix into memory. We also make the code scale better in memory; we use $O(n)$ memory where n is the number of movies whereas loading the entire matrix would use $O(n^2)$ memory.

One big challenge is reading the line from disk to memory. Although reading a line is much faster than reading the entire file, it still takes a few seconds. In the future, we want to use a different technique to read a single line from the file, or we would replace the file storage with a database.

## V. Conclusion

In this project, we applied both content-based filtering and item-based collaborative filtering to build the recommendation system using the MovieLens Dataset. For the content-based filtering, we used cosine similarity method and found the similar movies on two different metrics; i.e, one based on similar movie description and other based on similar keywords, cast, crew, genre and ratings of the movie. For the item-based collaborative filtering approach, we used KNN algorithm and SVD, from which we got approximately $80\%$ accuracy. Further we have also made a hybrid recommendation system which uses cosine similarity matrix based on keywords and SVD to make recommendations. After comparing the results for each movie recommender, we considered cosine similarity approach gave us better results than other methods and so we have then used them for our webpage. For the further work, we plan to consider the working of other advanced recommendation approaches to improve our results.

## VI. References


1) https://www.kaggle.com/gspmoreira/recommender-systems-in-python-101
2) https://www.kaggle.com/vipulgandhi/pca-beginner-s-guide-to-dimensionality-reduction/comments
3) https://towardsdatascience.com/recommender-system-application-development-part-1-of-4-cosine-similarity-f6dbcd768e83
4) https://towardsdatascience.com/recommender-system-application-development-part-1-of-4-cosine-similarity-f6dbcd768e83
5) https://www.youtube.com/watch?v=ueKXSupHz6Q
6) https://www.kaggle.com/muhammadayman/recommendation-system-using-cosine-similarity?scriptVersionId=58384845