

[Get started](#)

To make Medium work, we log user data.

By using Medium, you agree to our

Privacy Policy, including cookie policy.

James Le

16.5K Followers

About

[Follow](#)

You have **2** free member-only stories left this month.

[Sign up for Medium and get an extra one](#)

The 4 Recommendation Engines That Can Predict Your Movie Tastes



James Le Apr 22, 2018 · 18 min read ★

[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

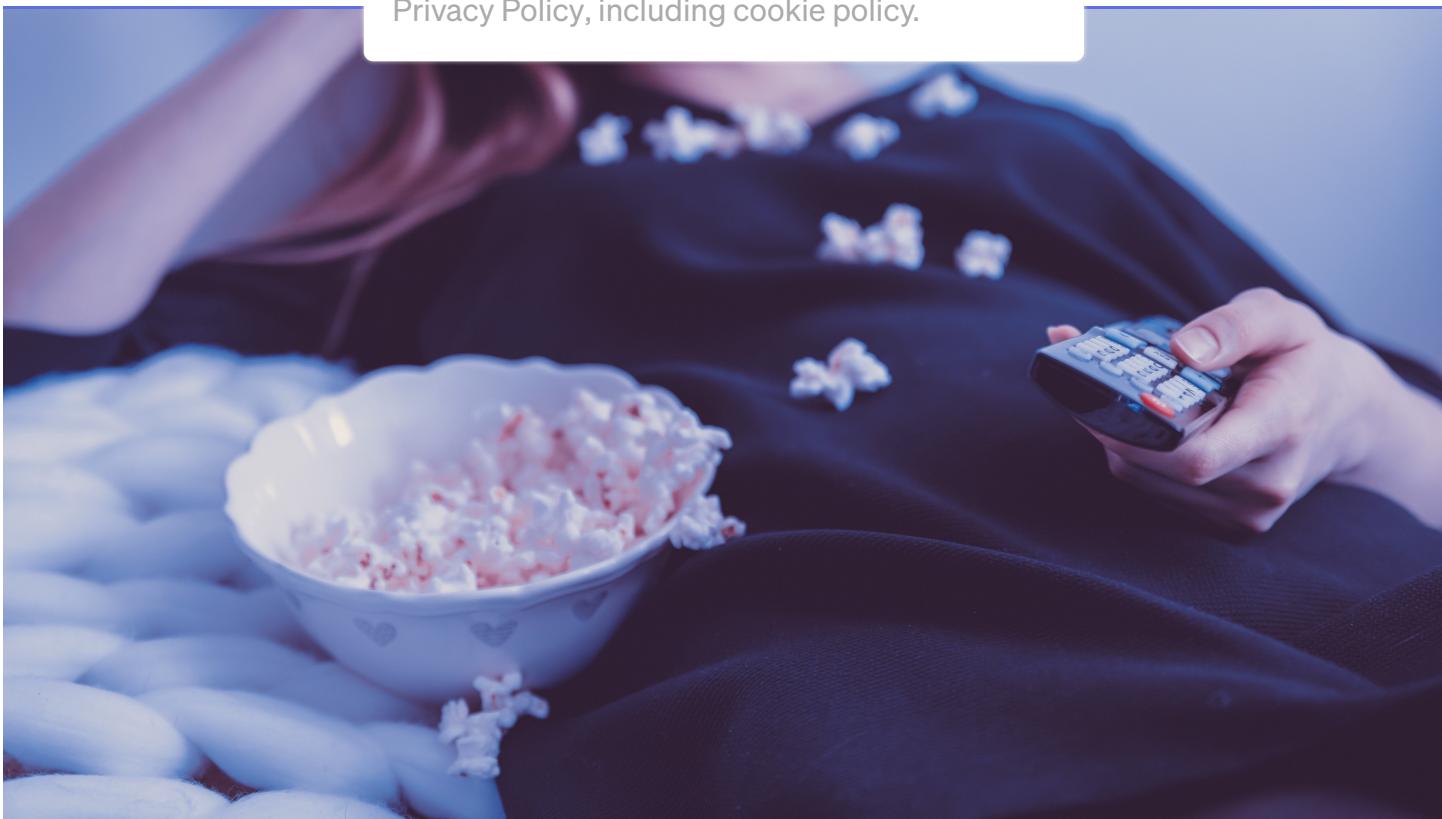


Photo by [JESHOOPTS.COM](#) on [Unsplash](#)

“What movie should I watch this evening?”

Have you ever had to answer this question at least once when you came home from work? As for me — yes, and more than once. From Netflix to Hulu, the need to build robust movie recommendation systems is extremely important given the huge demand for personalized content of modern consumers.

An example of recommendation system is such as this:

- User A watches **Game of Thrones** and **Breaking Bad**.
- User B does search on **Game of Thrones**, then the system suggests **Breaking Bad** from data collected about user A.


[Get started](#)
O

To make Medium work, we log user data.

By using Medium, you agree to our

Privacy Policy, including cookie policy.

Tube (Videos)

Pandora/Spotify

etc.

The screenshot shows the Netflix homepage with a dark background. At the top, there's a navigation bar with 'NETFLIX' and 'Browse' options. Below the navigation, there's a section titled 'Top Picks for Joshua' featuring thumbnails for 'Breaking Bad', 'SING', 'The Fosters', 'New Girl', 'are you here?', and 'BABY DADDY'. Underneath this, there's a 'Trending Now' section with thumbnails for 'shameless', 'Schitt's Creek', 'ORANGE IS THE BLACK', 'OZARK', 'New Girl', 'STRANGER THINGS', and 'THE RA...'. Further down, there's a section titled 'Because you watched Narcos' with thumbnails for 'SURVIVING ESCOBAR ALIAS J.J.', 'COMORBID', 'PABLO ESCOBAR', 'SUBURRA BLOOD ON ROME', 'ALIAS J.J. LA CELEBRIDAD DEL MAL', and 'ANTHONY BOURDAIN PARTS UNKNOWN'. At the bottom, there's a 'New Releases' section with thumbnails for 'BEYOND STRANGER THINGS', 'MOANA', 'THE MIST', 'THE BABYSITTER', 'RIVERDALE', and 'DOCTOR STRANGE'.

In this post, I will show you how to implement the 4 different movie recommendation approaches and evaluate them to see which one has the best performance.

The MovieLens Dataset

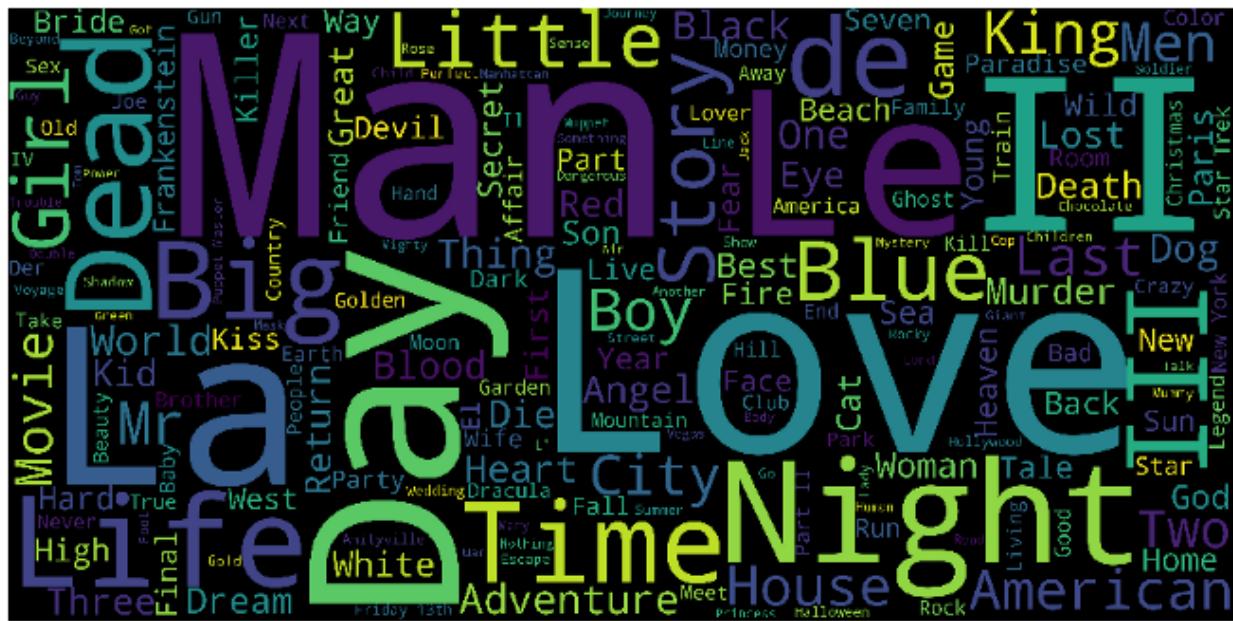
The dataset that I'm working with is [MovieLens](#), one of the most common datasets that is available on the internet for building a Recommender System. The version of the dataset that I'm working with (1M) contains 1,000,209 anonymous ratings of approximately 3,900 movies made by 6,040 MovieLens users who joined MovieLens in 2000.

Get started

- To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

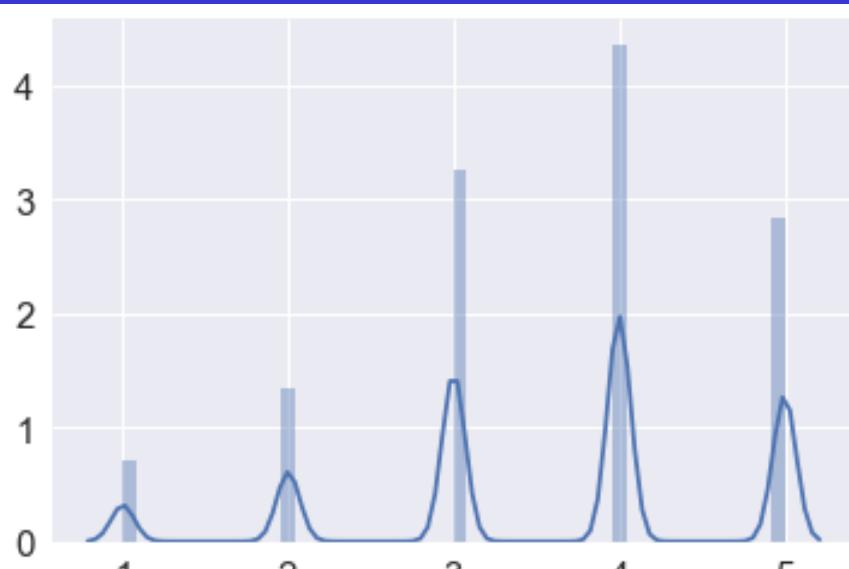


Here's a word-cloud visualization of the movie titles:



Beautiful, isn't it? I can recognize that there are a lot of movie franchises in this dataset, as evidenced by words like *II* and *III*... In addition to that, *Day*, *Love*, *Life*, *Time*, *Night*, *Man*, *Dead*, *American* are among the most commonly occurring words.

Here's a distribution of the user ratings:



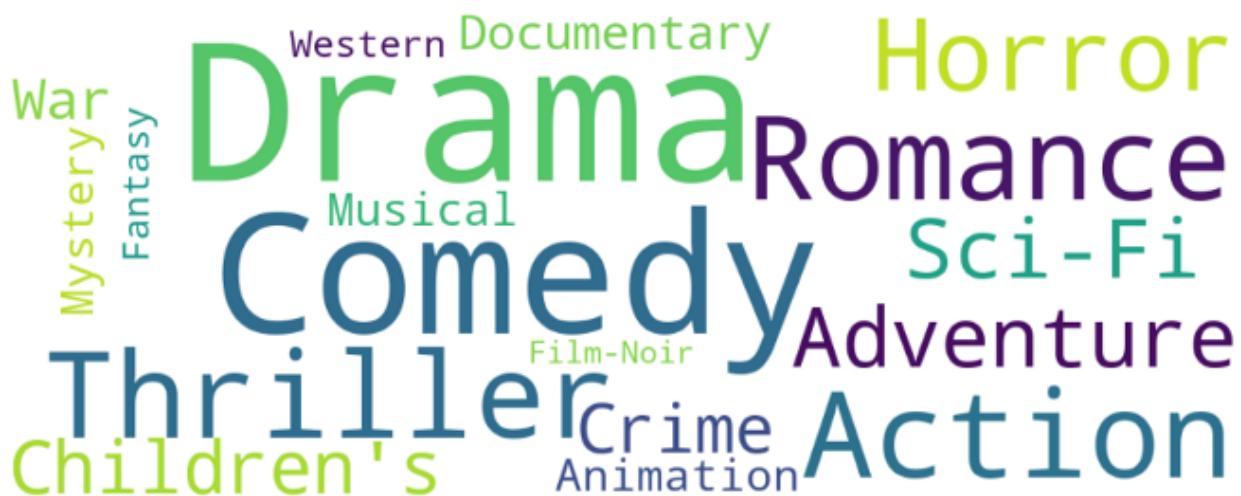
[Get started](#)

- o To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.



It appears that users are quite generous in their ratings. The mean rating is 3.58 on a scale of 5. Half the movies have a rating of 4 and 5. I personally think that a 5-level rating skill wasn't a good indicator as people could have different rating styles (i.e. person A could always use 4 for an average movie, whereas person B only gives 4 out for their favorites). Each user rated at least 20 movies, so I doubt the distribution could be caused just by chance variance in the quality of movies.

Here's another word-cloud of the movie genres:



The top 5 genres are, in that respect order: Drama, Comedy, Action, Thriller, and Romance.

Now let's move on to explore the 4 recommendation systems that can be used. Here they are, in respective order of presentation:

1. Content-Based Filtering

2. Memory-Based Collaborative Filtering

[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

[Deep Learn](#)

1 - Content-Based

The Content-Based Recommender relies on the similarity of the items being recommended. The basic idea is that if you like an item, then you will also like a “similar” item. It generally works well when it’s easy to determine the context/properties of each item.

A content based recommender works with data that the user provides, either explicitly movie ratings for the MovieLens dataset. Based on that data, a user profile is generated, which is then used to make suggestions to the user. As the user provides more inputs or takes actions on the recommendations, the engine becomes more and more accurate.

The Math

The concepts of **Term Frequency (TF)** and **Inverse Document Frequency (IDF)** are used in information retrieval systems and also content based filtering mechanisms (such as a content based recommender). They are used to determine the relative importance of a document / article / news item / movie etc.

TF is simply the frequency of a word in a document. IDF is the inverse of the document frequency among the whole corpus of documents. TF-IDF is used mainly because of two reasons: Suppose we search for “**the results of latest European Soccer games**” on Google. It is certain that “**the**” will occur more frequently than “**soccer games**” but the relative importance of **soccer games** is higher than the search query point of view. In such cases, TF-IDF weighting negates the effect of


[Get started](#)

- o To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

Below is the equation to calculate the TF-IDF score:

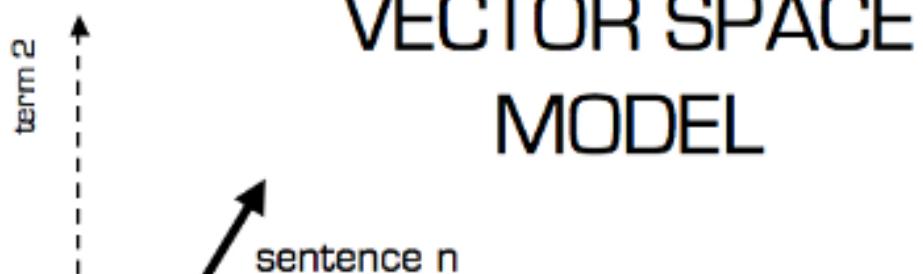
$$\text{tfidf}_{i,j} = \text{tf}_{i,j} \times \log \left(\frac{N}{\text{df}_i} \right)$$

$\text{tf}_{i,j}$ = total number of occurrences of i in j

df_i = total number of documents (speeches) containing i

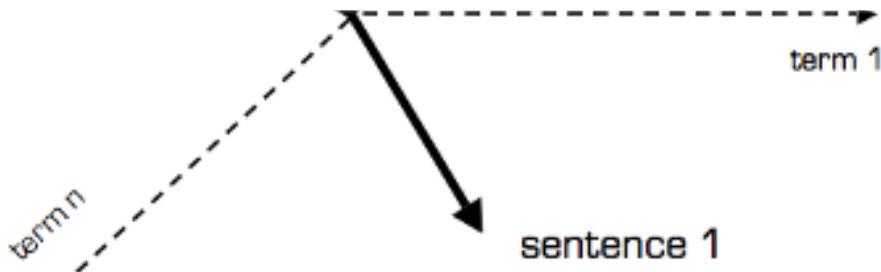
N = total number of documents (speeches)

After calculating TF-IDF scores, how do we determine which items are closer to each other, rather closer to the user profile? This is accomplished using the **Vector Space Model** which computes the proximity based on the angle between the vectors. In this model, each item is stored as a vector of its attributes (which are also vectors) in an **n-dimensional space** and the angles between the vectors are calculated to determine the similarity between the vectors. Next, the user profile vectors are also created based on his actions on previous attributes of items and the similarity between an item and a user is also determined in a similar way.



[Get started](#)

- To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.



Sentence 2 is more likely to be using Term 2 than using Term 1. Vice-versa for Sentence 1. The method of calculating this relative measure is calculated by taking the cosine of the angle between the sentences and the terms. The ultimate reason behind using cosine is that the **value of cosine will increase with decreasing value of the angle** between which signifies more similarity. The vectors are length normalized after which they become vectors of length 1 and then the cosine calculation is simply the sum-product of vectors.

The Code

With all that math in mind, I am going to build a Content-Based Recommendation Engine that computes similarity between movies based on movie genres. It will suggest movies that are most similar to a particular movie based on its genre.

I do not have a quantitative metric to judge the machine's performance so this will have to be done qualitatively. In order to do so, I'll use **TfidfVectorizer** function from **scikit-learn**, which transforms text to feature vectors that can be used as input to estimator.


[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

```
tfidf_matrix = tfidf.fit_transform(movies['genres'])
```

I will be using the [Cosine Similarity](#) to calculate a numeric quantity that denotes the similarity between two movies. Since I have used the TF-IDF Vectorizer, calculating the Dot Product will directly give me the Cosine Similarity Score. Therefore, I will use sklearn's `linear_kernel` instead of `cosine_similarities` since it is much faster.

```
from sklearn.metrics.pairwise import linear_kernel
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)
```

I now have a pairwise cosine similarity matrix for all the movies in the dataset. The next step is to write a function that returns the 20 most similar movies based on the cosine similarity score.

```
# Build a 1-dimensional array with movie titles
titles = movies['title']
indices = pd.Series(movies.index,
index=movies['title'])

# Function that get movie recommendations based on the
# cosine similarity score of movie genres
def genre_recommendations(title):
    idx = indices[title]
    sim_scores = list(enumerate(cosine_sim[idx]))
    sim_scores = sorted(sim_scores, key=lambda x: x[1],
reverse=True)
    sim_scores = sim_scores[1:21]
    movie_indices = [i[0] for i in sim_scores]
    return titles.iloc[movie_indices]
```

[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.



how good the re

```
genre_recommendations('Good Will Hunting  
(1997)').head(20)
```

25	Othello (1995)
26	Now and Then (1995)
29	Shanghai Triad (Yao a yao yao dao waipo qiao) ...
30	Dangerous Minds (1995)
35	Dead Man Walking (1995)
39	Cry, the Beloved Country (1995)
42	Restoration (1995)
52	Lamerica (1994)
54	Georgia (1995)
56	Home for the Holidays (1995)
61	Mr. Holland's Opus (1995)
66	Two Bits (1995)
77	Crossing Guard, The (1995)
79	White Balloon, The (Badkonake Sefid) (1995)
81	Antonia's Line (Antonia) (1995)
82	Once Upon a Time... When We Were Colored (1995)
89	Journey of August King, The (1995)
92	Beautiful Girls (1996)
95	Hate (Haine, La) (1995)
112	Margaret's Museum (1995)

```
genre_recommendations('Toy Story (1995)').head(20)
```

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)

[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.



3685	Ac	e, 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)	

```
genre_recommendations('Saving Private Ryan  
(1998)').head(20)
```

461	Heaven & Earth (1993)
1204	Full Metal Jacket (1987)
1214	Boat, The (Das Boot) (1981)
1222	Glory (1989)
1545	G.I. Jane (1997)
1959	Saving Private Ryan (1998)
2358	Thin Red Line, The (1998)
2993	Longest Day, The (1962)
3559	Flying Tigers (1942)
3574	Fighting Seabees, The (1944)
3585	Guns of Navarone, The (1961)
3684	Patriot, The (2000)
40	Richard III (1995)
152	Beyond Rangoon (1995)

[Get started](#)

O

To make Medium work, we log user data.

By using Medium, you agree to our

Privacy Policy, including cookie policy.

523

641	Courage Under Fire (1996)
967	Nothing Personal (1995)
979	Michael Collins (1996)
1074	Platoon (1986)

As you can see, I have quite a decent list of recommendation for **Good Will Hunting** (Drama), **Toy Story** (Animation, Children's, Comedy), and **Saving Private Ryan** (Action, Thriller, War).

Overall, here are the pros of using content-based recommendation:

- No need for data on other users, thus no cold-start or sparsity problems.
- Can recommend to users with unique tastes.
- Can recommend new & unpopular items.
- Can provide explanations for recommended items by listing content-features that caused an item to be recommended (in this case, movie genres)

However, there are some cons of using this approach:

- Finding the appropriate features is hard.
- Does not recommend items outside a user's content profile.
- Unable to exploit quality judgments of other users.

2 — Collaborative Filtering

[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.



the similarity in users. It

analyses how similar the tastes of one user is to another and makes recommendations on the basis of that.

For instance, if user A likes movies 1, 2, 3 and user B likes movies 2,3,4, then they have similar interests and A should like movie 4 and B should like movie 1. This makes it one of the most commonly used algorithm as it is not dependent on any additional information.

In general, collaborative filtering is the workhorse of recommender engines. The algorithm has a very interesting property of being able to do feature learning on its own, which means that it can start to learn for itself what features to use.

The Math

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

1. User-User Collaborative Filtering: Here we find look alike users based on similarity and recommend movies which first user's look-alike has chosen in past. This algorithm is very effective but takes a lot of time and resources. It requires to compute every user pair information which takes time. Therefore, for big base platforms, this algorithm is hard to implement without a very strong parallelizable system.

2. Item-Item Collaborative Filtering: It is quite similar to previous algorithm, but instead of finding user's look-alike, we try finding movie's look-alike. Once we have movie's look-alike matrix, we can easily recommend alike movies to user who have rated any movie

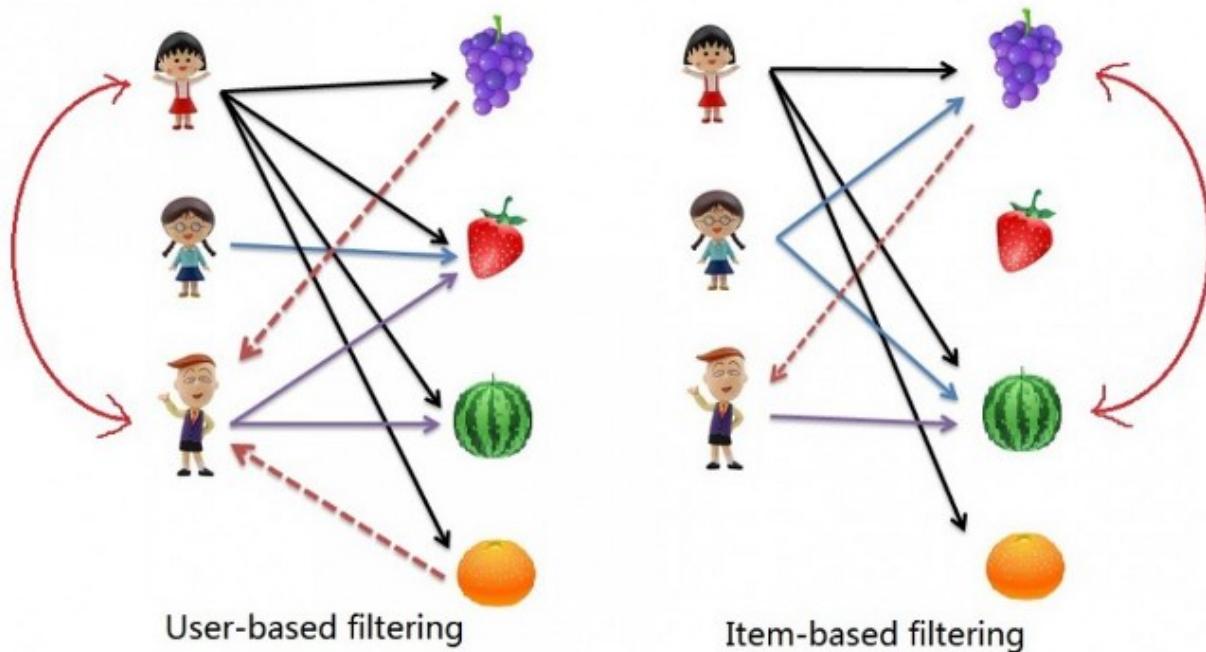

[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

algorithm ta

don't need all similarity scores between users. And with fixed number of movies, movie-movie look alike matrix is fixed over time.

aborate as we



In either scenario, we build a similarity matrix. For user-user collaborative filtering, the **user-similarity matrix** will consist of some distance metrics that measure the similarity between any two pairs of users. Likewise, the **item-similarity matrix** will measure the similarity between any two pairs of items.

There are 3 distance similarity metrics that are usually used in collaborative filtering:

- 1. Jaccard Similarity:** Similarity is based on the number of users which have rated item A and B divided by the number of users who have rated either A or B. It is typically used where we don't

[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

2. Cosine Similarity: (as in the Content-Based system) Similarity is the cosine of the angle between the 2 vectors of the item vectors of A and B. Closer the vectors, smaller will be the angle and larger the cosine.

3. Pearson Similarity: Similarity is the pearson coefficient between the two vectors. For the purpose of diversity, I will use **Pearson Similarity** in this implementation.

The Code

Due to the limited computing power in my laptop, I will build the recommender system using only a subset of the ratings. In particular, I will take a random sample of 20,000 ratings (2%) from the 1M ratings.

I use the **scikit-learn library** to split the dataset into testing and training. **Cross_validation.train_test_split** shuffles and splits the data into two datasets according to the percentage of test examples, which in this case is 0.2.

```
from sklearn import cross_validation as cv
train_data, test_data = cv.train_test_split(small_data,
test_size=0.2)
```

Now I need to create a user-item matrix. Since I have splitted the data into testing and training, I need to create two matrices. The training matrix contains 80% of the ratings and the testing matrix contains 20% of the ratings.

[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

```
['user_id',  
 test_data_matrix = test_data.as_matrix(columns =  
 ['user_id', 'movie_id', 'rating'])]
```

Now I use the `pairwise_distances` function from `sklearn` to calculate the Pearson Correlation Coefficient. This method provides a safe way to take a distance matrix as input, while preserving compatibility with many other algorithms that take a vector array.

```
from sklearn.metrics.pairwise import pairwise_distances  
  
# User Similarity Matrix  
user_correlation = 1 - pairwise_distances(train_data,  
metric='correlation')  
user_correlation[np.isnan(user_correlation)] = 0  
  
# Item Similarity Matrix  
item_correlation = 1 -  
pairwise_distances(train_data_matrix.T,  
metric='correlation')  
item_correlation[np.isnan(item_correlation)] = 0
```

With the similarity matrix in hand, I can now predict the ratings that were not included with the data. Using these predictions, I can then compare them with the test data to attempt to validate the quality of our recommender model.

```
# Function to predict ratings  
def predict(ratings, similarity, type='user'):  
    if type == 'user':  
        mean_user_rating = ratings.mean(axis=1)  
        # Use np.newaxis so that mean_user_rating has  
        same format as ratings
```

[Get started](#)

- o To make Medium work, we log user data.
- o By using Medium, you agree to our Privacy Policy, including cookie policy.

```
similarity = np.array([np.abs(similarity).sum(axis=0) / n_items])
if type == 'item':
    pred = ratings.dot(similarity) / np.array([np.abs(similarity).sum(axis=1)])
return pred
```

The Evaluation

There are many evaluation metrics but one of the most popular metric used to evaluate accuracy of predicted ratings is **Root Mean Squared Error (RMSE)**. I will use the **mean_square_error (MSE)** function from sklearn, where the RMSE is just the square root of MSE. I'll use the scikit-learn's **mean squared error** function as my validation metric. Comparing user- and item-based collaborative filtering, it looks like user-based collaborative filtering gives a better result.

```
from sklearn.metrics import mean_squared_error
from math import sqrt

# Function to calculate RMSE
def rmse(pred, actual):
    # Ignore nonzero terms.
    pred = pred[actual.nonzero()].flatten()
    actual = actual[actual.nonzero()].flatten()
    return sqrt(mean_squared_error(pred, actual))

# Predict ratings on the training data with both
# similarity score
user_prediction = predict(train_data_matrix,
user_correlation, type='user')
item_prediction = predict(train_data_matrix,
item_correlation, type='item')

# RMSE on the train data
print('User-based CF RMSE: ' +
str(rmse(user_prediction, train_data_matrix)))
```

[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

Output

User-based CF RMSE: 0.9999999999999999
Item-based CF RMSE: 114.97271725933925

RMSE of training of model is a metric which measure how much the signal and the noise is explained by the model. I noticed that my RMSE is quite big. I suppose I might have overfitted the training data.

Overall, **Memory-based Collaborative Filtering** is easy to implement and produce reasonable prediction quality. However, there are some drawback of this approach:

- It doesn't address the well-known cold-start problem, that is when new user or new item enters the system.
- It can't deal with sparse data, meaning it's hard to find users that have rated the same items.
- It suffers when new users or items that don't have any ratings enter the system.
- It tends to recommend popular items.

Note: The complete code for content-based and memory-based collaborative filtering can be found in [this Jupyter Notebook](#).

[Get started](#)

- To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

3 — Matrix Factorization

In the previous attempt, I have used memory-based collaborative filtering to make movie recommendations from users' ratings data. I can only try them on a very small data sample (20,000 ratings), and ended up getting pretty high Root Mean Squared Error (bad recommendations). Memory-based collaborative filtering approaches that compute distance relationships between items or users have these two major issues:

1. It doesn't scale particularly well to massive datasets, especially for real-time recommendations based on user behavior similarities — which takes a lot of computations.
2. Ratings matrices may be overfitting to noisy representations of user tastes and preferences. When we use distance based “neighborhood” approaches on raw data, we match to sparse low-level details that we assume represent the user's preference vector instead of the vector itself.

[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.



Thus I'd need to apply **Dimensionality Reduction** technique to derive the tastes and preferences from the raw data, otherwise known as doing low-rank matrix factorization. Why reduce dimensions?

- I can discover hidden correlations / features in the raw data.
- I can remove redundant and noisy features that are not useful.
- I can interpret and visualize the data easier.
- I can also access easier data storage and processing.

The Math

Model-based Collaborative Filtering is based on **matrix factorization (MF)** which has received greater exposure, mainly as an unsupervised learning method for latent variable decomposition and dimensionality reduction. Matrix factorization is widely used for recommender systems where it can deal better with scalability and sparsity than Memory-based CF:

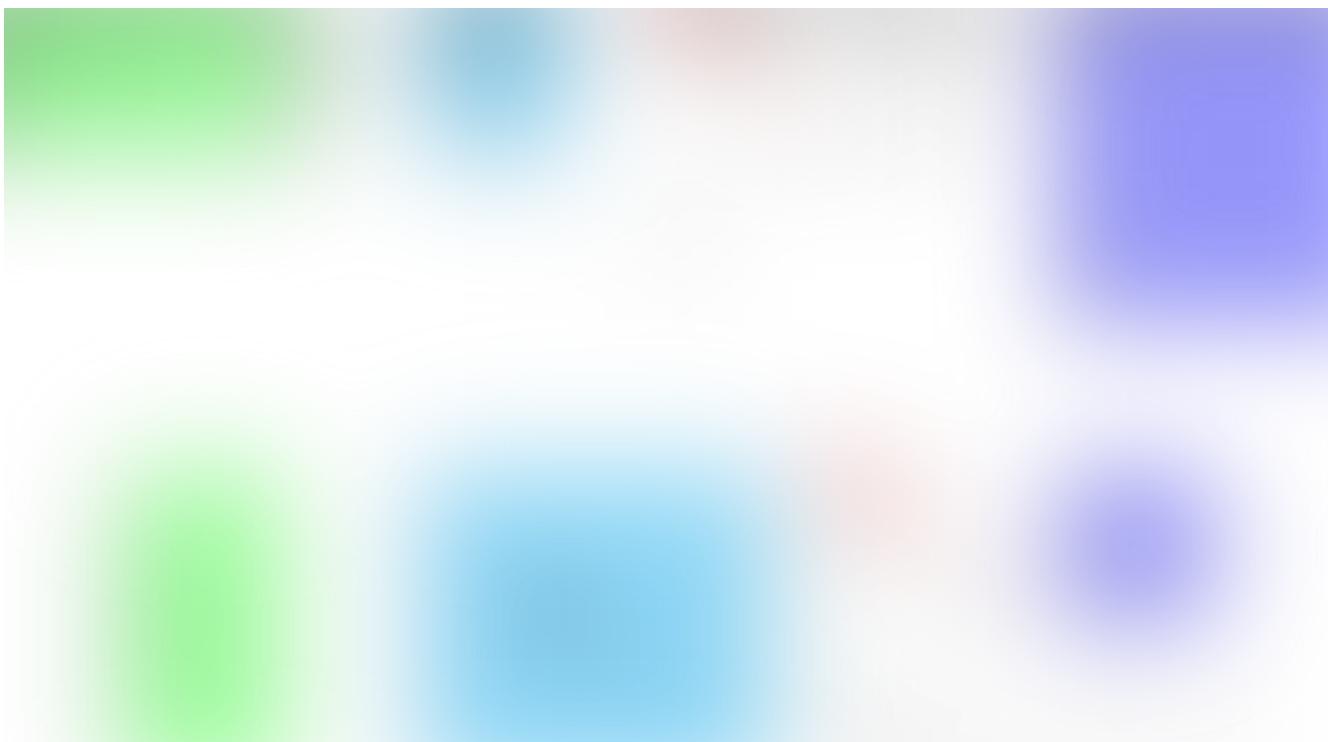
- The goal of MF is to learn the latent preferences of users and the latent attributes of items from known ratings (learn features that

[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

- When you have a very sparse matrix, with a lot of dimensions, by doing matrix factorization, you can restructure the user-item matrix into low-rank structure, and you can represent the matrix by the multiplication of two low-rank matrices, where the rows contain the latent vector.
- You fit this matrix to approximate your original matrix, as closely as possible, by multiplying the low-rank matrices together, which fills in the entries missing in the original matrix.

A well-known matrix factorization method is **Singular value decomposition (SVD)**. At a high level, SVD is an algorithm that decomposes a matrix A into the best lower rank (i.e. smaller/simpler) approximation of the original matrix A. Mathematically, it decomposes A into a two unitary matrices and a diagonal matrix:



[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

values (essential)

, and V^T is the

right singular vectors (movie “features” matrix). U and V^T are column orthonormal, and represent different things: U represents how much users “like” each feature and V^T represents how relevant each feature is to each movie.

To get the lower rank approximation, I take these matrices and keep only the top k features, which can be thought of as the underlying tastes and preferences vectors.

The Code

Scipy and Numpy both have functions to do the singular value decomposition. I’m going to use the Scipy function `svds` because it lets me choose how many latent factors I want to use to approximate the original ratings matrix (instead of having to truncate it after).

```
from scipy.sparse.linalg import svds
U, sigma, Vt = svds(Ratings_demeaned, k = 50)
```

As I’m going to leverage matrix multiplication to get predictions, I’ll convert the `Sum` (now are values) to the diagonal matrix form.

```
sigma = np.diag(sigma)
```

I now have everything I need to make movie ratings predictions for every user. I can do it all at once by following the math and matrix

[Get started](#)

- o To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.



But first, I need to add the user means back to get the actual star ratings prediction.

```
all_user_predicted_ratings = np.dot(np.dot(U, sigma),  
Vt) + user_ratings_mean.reshape(-1, 1)
```

With the predictions matrix for every user, I can build a function to recommend movies for any user. I return the list of movies the user has already rated, for the sake of comparison.

```
preds = pd.DataFrame(all_user_predicted_ratings,  
columns = Ratings.columns)
```

Now I write a function to return the movies with the highest predicted rating that the specified user hasn't already rated.

```
def recommend_movies(predictions, userID, movies,  
original_ratings, num_recommendations):  
  
    # Get and sort the user's predictions  
    user_row_number = userID - 1 # User ID starts at 1,  
    not 0  
    sorted_user_predictions =  
    preds.iloc[user_row_number].sort_values(ascending=False  
    ) # User ID starts at 1  
  
    # Get the user's data and merge in the movie  
    information.  
    user_data =  
    original_ratings[original_ratings.user_id == (userID)]  
    user_full = (user_data.merge(movies, how = 'left',
```

[Get started](#)

- o To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.



```
# Recommend the highest predicted rating movies
# that the user hasn't seen yet.
recommendations =
(movies[~movies['movie_id'].isin(user_full['movie_id'])]
].
merge(pd.DataFrame(sorted_user_predictions).reset_index()
(), how = 'left',
      left_on = 'movie_id',
      right_on = 'movie_id').
rename(columns = {user_row_number:
'Predictions'}).
sort_values('Predictions', ascending = False).
iloc[:num_recommendations, :-1]
)

return user_full, recommendations
```

The Evaluation

Instead of doing evaluation manually like the last time, I will use the [Surprise](#) library that provided various ready-to-use powerful prediction algorithms including (SVD) to evaluate its RMSE (Root Mean Squared Error) on the MovieLens dataset. It is a Python scikit building and analyzing recommender systems.

```
# Import libraries from Surprise package
from surprise import Reader, Dataset, SVD, evaluate

# Load Reader library
reader = Reader()

# Load ratings dataset with Dataset library
data = Dataset.load_from_df(ratings[['user_id',
'movie_id', 'rating']], reader)
```

[Get started](#)

O

To make Medium work, we log user data.

By using Medium, you agree to our

Privacy Policy, including cookie policy.

```
# use the SVD
svd = SVD()
```

```
# Compute the RMSE of the SVD algorithm.
evaluate(svd, data, measures=['RMSE'])
```

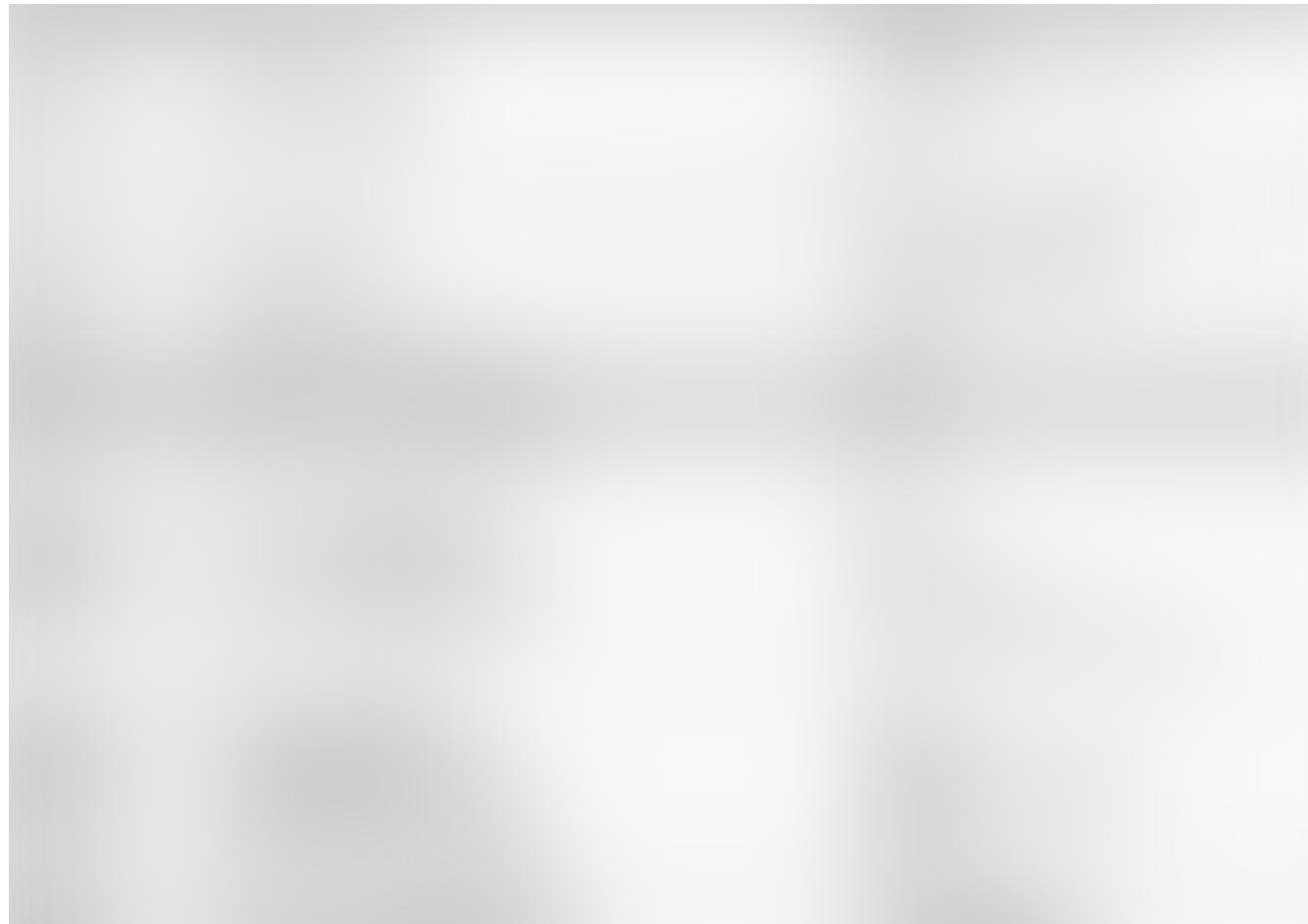
I get a mean Root Mean Square Error of 0.8736 which is pretty good.

The Recommendation

Let's try to recommend 20 movies for user with ID 1310.

```
predictions = recommend_movies(preds, 1310, movies,
ratings, 20)
```

```
predictions
```



[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

These look like pretty good recommendations. It's good to see that, although I didn't actually use the genre of the movie as a feature, the truncated matrix factorization features "picked up" on the underlying tastes and preferences of the user. I've recommended some comedy, drama, and romance movies — all of which were genres of some of this user's top rated movies.

Note: The complete code for SVD Matrix Factorization can be found in [this Jupyter Notebook](#).

4 — Deep Learning

The Math

The idea of using deep learning is similar to that of Model-Based Matrix Factorization. In matrix factorization, we decompose our original sparse matrix into product of 2 low rank orthogonal matrices. For deep learning implementation, we don't need them to be orthogonal, we want our model to learn the values of embedding matrix itself. The user latent features and movie latent features are looked up from the embedding matrices for specific movie-user combination. These are the input values for further linear and non-linear layers. We can pass this input to multiple relu, linear or sigmoid layers and learn the corresponding weights by any optimization algorithm (Adam, SGD, etc.).

The Code

Here are the main components of my neural network:

[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.



- A left embedding layer that creates a Users by Latent Factors matrix.
- A right embedding layer that creates a Movies by Latent Factors matrix.
- When the input to these layers are (i) a user id and (ii) a movie id, they'll return the latent factor vectors for the user and the movie, respectively.
- A merge layer that takes the dot product of these two latent vectors to return the predicted rating.

This code is based on the approach outlined in [Alkahest's blog post Collaborative Filtering in Keras](#).

I then compile the model using Mean Squared Error (MSE) as the loss function and the AdaMax learning algorithm.

```
# Define model
model = CFModel(max_userid, max_movieid, K_FACTORS)
# Compile the model using MSE as the loss function and
# the AdaMax learning algorithm
model.compile(loss='mse', optimizer='adamax')
```

[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

million ratings, \approx

I trained the

model in roughly 6 minutes per epoch (30 epochs ~ 3 hours) inside my Macbook Laptop CPU. I spitted the training and validation data with ratio of 90/10.

```
# Callbacks monitor the validation loss
# Save the model weights each time the validation loss
# has improved
callbacks = [EarlyStopping('val_loss', patience=2),
             ModelCheckpoint('weights.h5',
                             save_best_only=True)]

# Use 30 epochs, 90% training data, 10% validation data
history = model.fit([Users, Movies], Ratings,
                     nb_epoch=30, validation_split=.1, verbose=2,
                     callbacks=callbacks)
```

The next step is to actually predict the ratings a random user will give to a random movie. Below I apply the freshly trained deep learning model for all the users and all the movies, using 100 dimensional embeddings for each of them.

```
# Use the pre-trained model
trained_model = CFModel(max_userid, max_movieid,
                       K_FACTORS)
# Load weights
trained_model.load_weights('weights.h5')
```

Here I define the function to predict user's rating of unrated items.


[Get started](#)

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

```
def predict_user(self, user_id):
    return self._model_predict(user_id, movie_id - 1)
```

The Evaluation

During the training process above, I saved the model weights each time the validation loss has improved. Thus, I can use that value to calculate the best validation Root Mean Square Error.

```
# Show the best validation RMSE
min_val_loss, idx = min((val, idx) for idx, val in
enumerate(history.history['val_loss']))

print 'Minimum RMSE at epoch', '{:d}'.format(idx+1),
'=', '{:.4f}'.format(math.sqrt(min_val_loss))

## Output
Minimum RMSE at epoch 17 = 0.8616
```

The best validation loss is **0.7424** at epoch 17. Taking the square root of that number, I got the RMSE value of **0.8616**, which is better than the RMSE from the SVD Model (**0.8736**).

The Recommendation

Here I make a recommendation list of unrated 20 movies sorted by prediction value for user ID 2000. Let's see it.

```
recommendations =
ratings[ratings['movie_id'].isin(user_ratings['movie_id']) == False][['movie_id']].drop_duplicates()
```

[Get started](#)

- To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

```
recommendations.user_values_by_prediction,  
ascending=False).merge(movies, on='movie_id',  
how='inner',  
suffixes=['_u', '_m']).head(20)
```



This model performed better than all the approaches I attempted before (content-based, user-item similarity collaborative filtering,

[Get started](#)

- o To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

Note: The complete code for Deep Learning Model can be found in [this Jupyter Notebook](#).

Last Takeaway

Recommendation Engine is your companion and advisor to help you make the right choices by providing you tailored options and creating a personalized experience for you. It is beyond any doubt that recommendation engines are getting popular and critical in the new age of things. It is going to be in your best interest to learn to use them for businesses to be more competitive and consumers to be more efficient. I hope that this post has been helpful for you to learn about the 4 different approaches to build your own movie recommendation system. You can view all the source code in my GitHub repo at [this link](#) (<https://github.com/khanhnamle1994/movielens>). Let me know if you have any questions or suggestions on improvement!

[Recommendation System](#)[Movies](#)[Machine Learning](#)[Personalization](#)[Predictions](#)

[Get started](#)

To make Medium work, we log user data.

By using Medium, you agree to our

Privacy Policy, including cookie policy.

Get the Medium app



Download on the
App Store



GET IT ON
Google Play