

CS F469: Information Retrieval

Recommender System

Ahsan, Dushyant, Mansi, Sarthak, Shreyasi

Overview

- **Part A:** User Based Collaborative Filtering
- **Part B:** Improvements to UBCF
 - Content Boosting By Significance Weighting
 - Matrix Factorization

Part A: User Based Collaborative Filtering

Splitting Data



```
ratings_train, ratings_test = split_ratings_data(ratings, test_users)
```

Preprocessing Data - Movie IDs

```
def calc_movieId_map(self):  
    # converts movieId from original dataset to a 0 indexed  
    # (contiguous) id  
    index = 0  
    movieId_map = {}  
    for movieId in self.ratings.movieId:  
        if movieId not in movieId_map.keys():  
            movieId_map[movieId] = index  
            index+=1  
    return movieId_map
```

Preprocessing Data - Utility Matrix

```
def get_utility_matrix(self, ratings):
    utility_matrix = np.zeros((self.num_users, self.num_movies))
    # setting values in utility matrix
    for record in self.ratings.iter tuples():
        _, user, movie, rating = record
        utility_matrix[(user - 1), self.movieId_map[movie]] = rating
    return utility_matrix
```

Utility matrix

```
[[4. 4. 4. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

Preprocessing Data - Similarity Matrix

```
# df.corr() calculates the pearson similarities
def get_similarity_matrix(self, utility_matrix):
    utility_df = pd.DataFrame(data=self.utility_matrix.transpose())
    return utility_df.corr()
```

Similarity Matrix

	0	1	2	...	607	608	609
0	1.000000	0.019400	0.053056	...	0.262241	0.085434	0.098719
1	0.019400	1.000000	-0.002594	...	0.032730	0.024373	0.089329
2	0.053056	-0.002594	1.000000	...	0.008096	-0.002963	0.015962
3	0.176920	-0.003804	-0.004556	...	0.116861	0.023930	0.062523
4	0.120866	0.013183	0.001887	...	0.122607	0.258289	0.040372
..
605	0.121981	0.011299	-0.003246	...	0.188354	0.052385	0.093851
606	0.254200	0.005813	0.012885	...	0.258245	0.142533	0.098518
607	0.262241	0.032730	0.008096	...	1.000000	0.109563	0.248944
608	0.085434	0.024373	-0.002963	...	0.109563	1.000000	0.033713
609	0.098719	0.089329	0.015962	...	0.248944	0.033713	1.000000

Preprocessing Data - Average Ratings



```
# returns average ratings on all rated movies of users
def get_average_ratings(self):
    average_ratings = {}
    for user in range(self.utility_matrix.shape[0]):
        util_matrix_user = self.utility_matrix[user]
        util_matrix_user[util_matrix_user == 0] = np.nan
        average_ratings[user] = np.nanmean(util_matrix_user)
    return average_ratings
```


5 fold cross validation - top N similarity

- The neighborhood selection criteria was based on the **Top N recommendation** algorithm that uses a similarity-based vector model to identify the k most similar users to an active user.
- Choice of k as **75** is based on the performance of the predictor on a **5 fold cross validation**, selected from a range of choices between 10-200
- The similarities and ratings of these k most similar users is used for predicting the ratings for unseen movies m for any user u using the **Resnick prediction formula**

$$pred(a, p) = \bar{r}_a + \frac{\sum_{b \in N} sim(a, b) * (r_{b,p} - \bar{r}_b)}{\sum_{b \in N} sim(a, b)}$$

```

1 topN = 75 if content_boosted else 75
2 kf = KFold(n_splits = 5, shuffle = True, random_state = 2) # 5 fold
3 j = 0 # track split number
4 for training, testing in kf.split(ratings_train):
5     j += 1
6     print ("Split # {}".format(j))
7     train = ratings_train.iloc[training]
8     test = ratings_train.iloc[testing]
9     # initialise preprocess class
10    cv_preprocess = preprocess_content_boosted(ratings, train, movies) if content_boosted else
        preprocess_UB_CF(ratings, train)
11    # initialise model and generate neighborhood
12    model = UB_CF(train, cv_preprocess.movieId_map, cv_preprocess.utility_matrix,
13                  cv_preprocess.weighted_similarity_matrix if content_boosted else
14                  cv_preprocess.similarity_matrix, cv_preprocess.average_ratings)
15    model.generate_neighborhood(topN)
16    # get predictions
17    predicted_ratings = np.zeros(test.rating.shape)
18    count = 0
19    for i,record in enumerate(test.itertuples()):
20        _, user, movie, rating = record
21        mv = cv_preprocess.movieId_map[movie]
22
23        try:
24            pred = model.get_resnick(user-1, mv) # get prediction using resnick, 'user-1' here
        because users otherwise users indexed to 1 in ratings.csv
25            predicted_ratings[i] = pred
26        except Exception as e:
27            # print(e)
28            predicted_ratings[i] = 0
29            count += 1 # number of exceptions
30
31    # print("Exception Count is : ", count)
32    print("Mean Absolute Error is : ", mean_absolute_error(test.rating, predicted_ratings))
33    MAE.append(mean_absolute_error(test.rating, predicted_ratings))

```

Results

- The model was evaluated using a 5 fold cross validation by calculating the Mean Absolute Error. The average MAE for the 5 folds turned out to be **0.6692**.
- MAE corresponding to each split: [0.668, 0.666, 0.670, 0.670, 0.672]

	Test User	Predicted Movie	Movies Seen in the Past
0	NaN	Movies, Rating	Movies, Rating >3
1	601.0	('Billy Madison (1995)', 5)	('How to Train Your Dragon (2010)', 5.0)
2	601.0	('Monty Python's Life of Brian (1979)', 5)	('Iron Man (2008)', 5.0)
3	601.0	('Clockwork Orange, A (1971)', 5)	('Saving Private Ryan (1998)', 5.0)
4	601.0	('Blues Brothers, The (1980)', 5)	('Life Is Beautiful (La Vita è bella) (1997)', ...)
5	601.0	('Fantasia (1940)', 5)	('Matrix, The (1999)', 5.0)
6	602.0	('Men in Black (a.k.a. MIB) (1997)', 5)	('Braveheart (1995)', 5.0)
7	602.0	('Wallace & Gromit: The Best of Aardman Animat...	('Clueless (1995)', 5.0)
8	602.0	('Fifth Element, The (1997)', 5)	('In the Line of Fire (1993)', 5.0)
9	602.0	('Welcome to the Dollhouse (1995)', 5)	('In the Name of the Father (1993)', 5.0)
10	602.0	('Prophecy, The (1995)', 5)	('Red Rock West (1992)', 5.0)

Part B: Improvements to the Code and Q and A

Question 1. What are the challenges in CF-RS built in part A?

1. Hard to Include Other Features in Predictions - Completely dependent on Utility Matrix
2. Data Sparsity - Some users have rated as few as 20 movies out of 9700
3. Scaling Problem - Have to generate neighbors at run time causing high inference times
4. Cold Start Problem - Handling new users is not possible

Question 2. What improvements are you proposing?

We use 'content boosting' for improving our model by making use of more information from the dataset and perform associations to improve the semantic significance of the similarity weights used to generate the neighborhoods for the active user.

There are 3 features we indirectly use to create hybrid weights for the similarity calculation :

1. Number of movies the active user and its neighbors have both rated
2. Harmonic mean of the absolute count of movies the users have rated
3. Movie genres

Co-rating weighting factor



`n` = number of movies rated by active user and it's neighbor

`corating_weight` = 1 if `n` >= 50 else `n`/50

Harmonic Means of movies rated



```
n1 = number of movies rated by user 1
n2 = number of movies rated by user 2

m1 = 1 if n1>=50 else n1/50
m2 = 1 if n2>=50 else n2/50

harmonic mean = (2* m1 * m2) / (m1 + m2)
```

Genre Preference Similarity



```
for user i:  
    vg[i] = [g1, g2, g3, .. gN]
```

where N is number of genres,
gk = count of movies with genre k rated by user i

```
genre_similarity_weight[i,j] = cosine_similarity_weight(vg[i], vg[j])
```

Creating the significance weighted similarities

```
for user i and j:  
    hybrid_weights[i,j] = (co_rating_weight[i,j] +  
                           harmonic_mean_weight[i,j] +  
                           genre_similarity_weight[i,j]) / 3  
  
    weighted_similarity_matrix[i,j] = pearson_corr[i,j] * hybrid_weights[i,j]
```

Question 3. How will the proposed improvement address the issues?

1. Co-rating weight :
We assume that we can rely more on the opinion of a neighbor who has a **higher common rated movies** with our active user, than a user who has only two common movies.
2. Harmonic Mean weight:
We assign **negative penalties on the similarities with users who have not rated many movies** as there is insufficient information about their preferences. Harmonic mean has a property of being high only when both values are higher, just as F1 metric is calculated from Precision and Recall
3. Genre preference similarity weight:
People will be recommended movies with a **significance attributed to their genre preferences** even though their neighbors might not have rated the same movies as the ones being recommended.

Question 4. A corner case where this improvement might not work.

1. **Unable to handle new users or cold start problem**

The case when the whole row of utility matrix is unrated, we would get a similarity of 0 with all users to generate neighbors and resnick prediction would also fail to work.

2. Users with very **eccentric tastes**

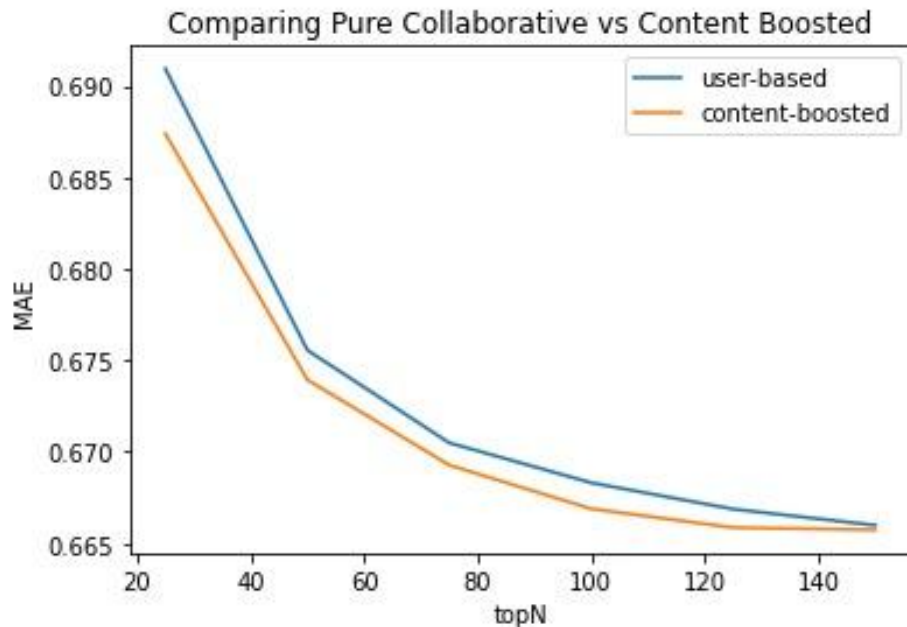
The case when the user may not have many close neighbors and therefore their predicted ratings may not be very accurate.

Especially, when none of the **movies rated by the users have been rated by any other user**, only movie genre information would be useful for the content boosted model.

Question 5. Actual impact of the improvement.

As seen in the plot, the Mean Average Error Values **consistently are lower for the content boosted model** as compared to a pure CF model for all choices of topN (# of neighbors).

Although the margin is thin, we **expect it to improve with the number of users** as there will be more content to capture from the remaining users, thus providing better clusters.



Question 6. What is the significance of multiplying the value of $(r_{v,m} - r_{av})$ by the similarity of user u to user v, in resnick prediction formula?

Similarity of user u and user v is the weight in the weighted sum of ratings. A user having higher similarity with the target user is considered more important than a user having lower similarity. Thus, higher similarity should be rewarded with more contribution in the prediction.

Question 7. Additional possible information from data to improve existing system developed in part A?

The additional information that the dataset has which has not been used in Part A :

1. **Timestamps** on user ratings can be used to track changes in preferences of the user over time, plotting the genre information with time might indicate a change in tastes for a set of users
2. **Genre** Information (included in part B). Additionally we can use pre-trained word embeddings (eg. word2vec) of the genre words to capture which genres are similar instead of a one-hot encoding which we have used.
3. **Associations between users** such as common rated movies (included in Part B)
4. **Tags** can also be converted to vectors using embeddings and further taking similarity might help in getting better neighbors, eg. some tags are actor names.

Extra: Matrix Factorisation

Original User and Movie Rating Matrix

```
Ratings = ratings.pivot(index = 'userId', columns = 'movieId', values = 'rating').fillna(0)
Ratings.head()
```

movieId	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
userId																						
1	4.0	0.0	4.0	0.0	0.0	4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.0	0.0
5	4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.0	0.0

5 rows × 9724 columns

Matrix updated with predicted ratings

```
[63] # Get the matrix of predicted ratings for all users
all_user_predicted_ratings = np.dot(np.dot(U, sigma), Vt) + user_ratings_mean.reshape(-1, 1)
preds = pd.DataFrame(all_user_predicted_ratings, columns = Ratings.columns)
preds.head()
```

movieId	1	2	3	4	5	6	7	8	9	10	
0	2.167328	0.402751	0.840184	-0.076281	-0.551337	2.504091	-0.890114	-0.026443	0.196974	1.593259	-0
1	0.211459	0.006658	0.033455	0.017419	0.183430	-0.062473	0.083037	0.024158	0.049330	-0.152530	0
2	0.003588	0.030518	0.046393	0.008176	-0.006247	0.107328	-0.012416	0.003779	0.007297	-0.059362	-0
3	2.051549	-0.387104	-0.252199	0.087562	0.130465	0.270210	0.477835	0.040313	0.025858	-0.017365	0
4	1.344738	0.778511	0.065749	0.111744	0.273144	0.584426	0.254930	0.128788	-0.085541	1.023455	0

5 rows × 9724 columns

Getting Recommendations

```
[65] already Rated, predictions = recommend_movies(preds, 256, movies, ratings, 20)
```

```
User 256 has already rated 174 movies
```


```
Recommending highest 20 predicted ratings movies not already rated.
```

Sparsity

```
sparsity = round(1.0 - len(ratings) / float(n_users * n_movies), 3)
print('Sparsity level of MovieLens dataset = ' + str(sparsity * 100) + '%')
```

```
Sparsity level of MovieLens dataset = 98.3%
```

Accuracy



```
# Compute the MAE of the SVD algorithm.
accuracy.mae(predictions)
```

```
MAE: 0.6691
0.6690969945133662
```

Contributions

Part A : Sarthak, Ahsan, Mansi

Part B Improvements : Mansi, Ahsan

Part B Matrix Factorization : Dushyant, Shreyasi

Report : Dushyant, Shreyasi

Slides : Sarthak, Ahsan

Questions