Collaborative Filtering for Recommendation System using Yahoo! Music User Ratings

JUN 13, 2021

20181353 김예진
20161200 이영호

CONTACT

Ulsan National Institute of Science and Technology
Address 50 UNIST-gil, Ulju-gun, Ulsan, 44919, Korea
Tel. +82 52 217 0114 **Web.** www.unist.ac.kr

Industrial Engineering
5th Engineering Building Room 302-4
Web. http://sdm.unist.ac.kr

CONTENTS

Introduction - Objective

Problem Definition

1. Predict **Rating** of user's unrated music
2. Reducing computation time

Metric

- **Root Mean Square Error(RMSE) - Accuracy**

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$$

- **Relative Cross Entropy(RCE)[1] – Fairness**

$$RCE = \frac{CE_{naive} - CE_{pred}}{CE_{naive}} * 100$$

- **Running Time - Efficiency**

$$Running\ Time = End\ Time - Start\ Time$$

Introduction - Data Set

YAHOO! LABS

R2 - Yahoo! Music User Ratings of Songs with Artist, Album, and Genre Meta Information, v. 1.0 (1.4 Gbyte & 1.1 Gbyte)

This dataset represents a snapshot of the Yahoo! Music community's preferences for various songs. The dataset contains over 717 million ratings of 136 thousand songs given by 1.8 million users of Yahoo! Music services. The data was collected between 2002 and 2006. Each song in the dataset is accompanied by artist, album, and genre attributes. The users, songs, artists, and albums are represented by randomly assigned numeric id's so that no identifying information is revealed. The mapping from genre id's to genre, as well as the genre hierarchy, is given. There are 2 sets in this dataset. Part one is 1.4 Gbytes and part 2 is 1.1 Gbytes.

Here are all the papers published on this Webscope Dataset:

- Rating music: Accounting for rating preferences
- Asking Questions and Developing Trust
- "All roads lead to Rome": optimistic recovery for distributed iterative data processing
- Distributed matrix factorization with mapreduce using a series of broadcast-joins
- Scalable similarity-based neighborhood methods with MapReduce

Dataset

- **3 txt files,** have **76,344,627 rows(1GB) rating data file, 136,736 rows song attributes file, and genre file**
- **200,000 user, 136,736 song**

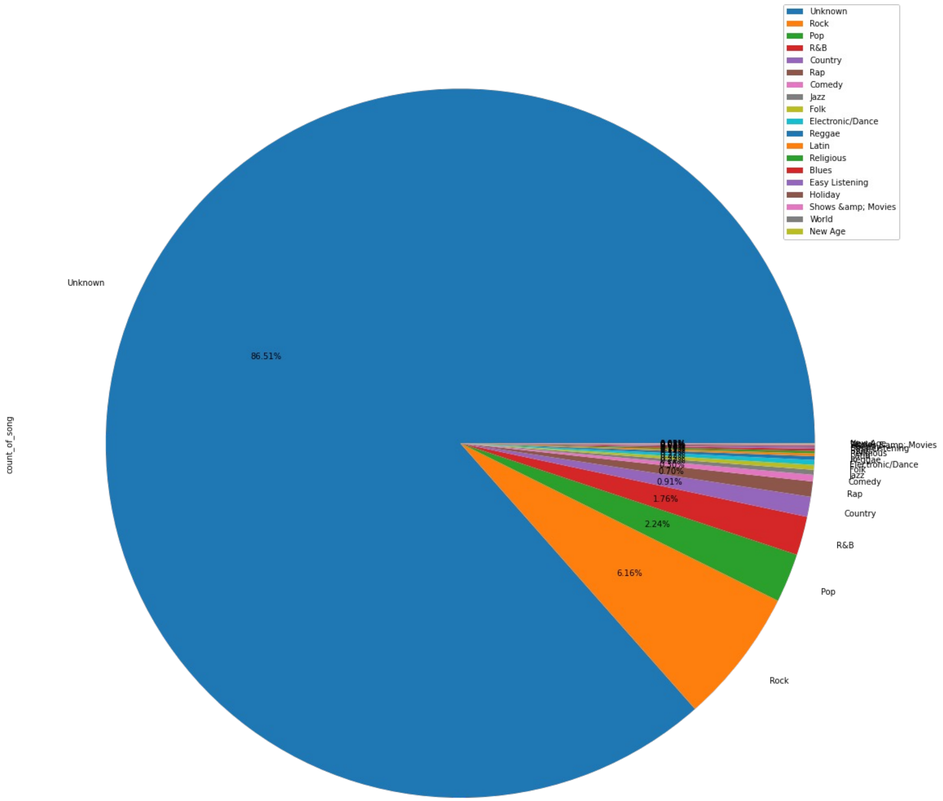
File Name	Columns
user-song-ratings	“user id”, “song id”, “rating”
song-attributes	“song id”, “album id”, “artist id”, “genre id”
genre-hierarchy	“genre id”, “parent genre id”, “level”, “genre name”

Introduction: Restrictions

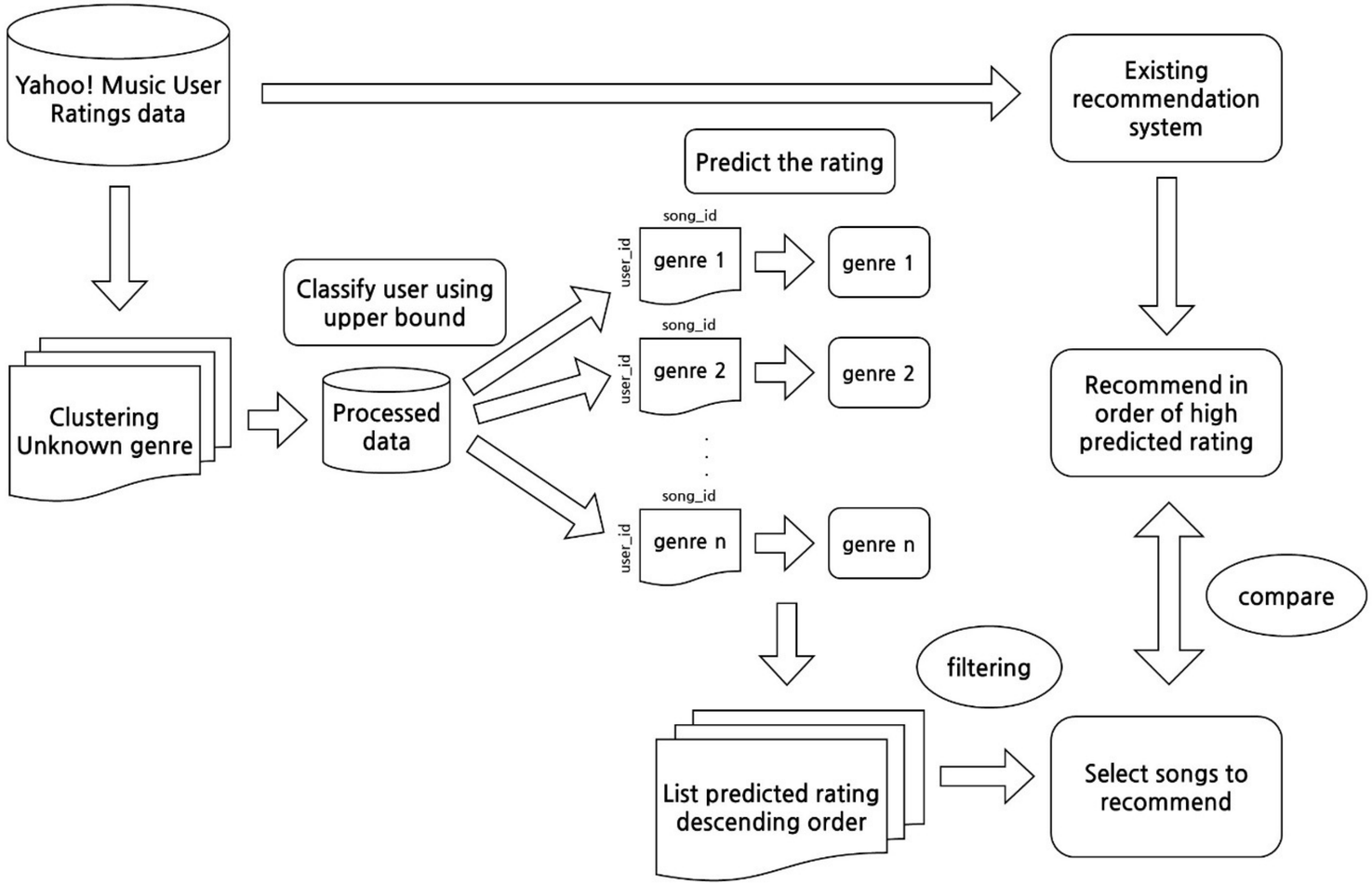
Imbalance Genres

- Genre of song have severe imbalance
- Most of the songs belong to the unknown genre(85%)

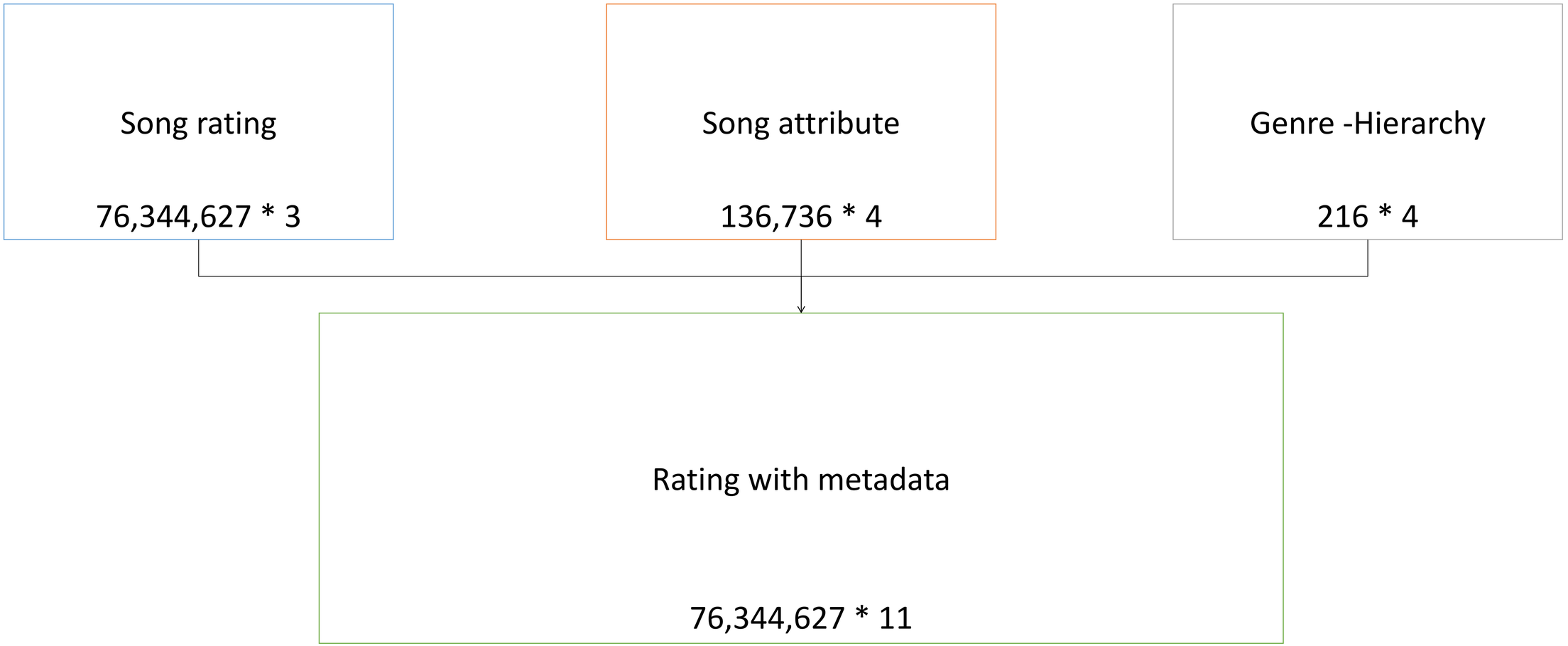
	large_category	count_of_song
13	Unknown	118294.0
7	Rock	8422.0
14	Pop	3059.0
9	R&B	2409.0
6	Country	1248.0
15	Rap	951.0
3	Comedy	410.0
8	Jazz	307.0
1	Folk	306.0
17	Electronic/Dance	302.0
0	Reggae	234.0
5	Latin	170.0
16	Religious	152.0
10	Blues	131.0
11	Easy Listening	114.0
4	Holiday	94.0
12	Shows & Movies	75.0
18	World	37.0
2	New Age	21.0



Model Architecture



Pre-Processing : Merge Data



user_id	song_id	rating	album_id	artist_id	genre_id	parent_genre_id	level	genre_name	large_category	large_category_genre_id

Pre-Processing : Unknown Music

```
unknown_df = pd.DataFrame(df[df['large_category_genre_id'] == 0])
```

executed in 2.85s, finished 16:12:01 2021-06-10

```
print('Total unique users in the dataset', unknown_df['user_id'].nunique())
print('Total unique songs in the dataset', unknown_df['song_id'].nunique())
print('Total unique artists in the dataset', unknown_df['artist_id'].nunique())
print('Total unique albums in the dataset', unknown_df['album_id'].nunique())
```

executed in 3.18s, finished 16:12:04 2021-06-10

```
Total unique users in the dataset 199993
Total unique songs in the dataset 118294
Total unique artists in the dataset 9282
Total unique albums in the dataset 18318
```

85% of the songs belong to the unknown genre
99.9965% of users rated the unknown song

Unknown genre **should be divided into small genre**

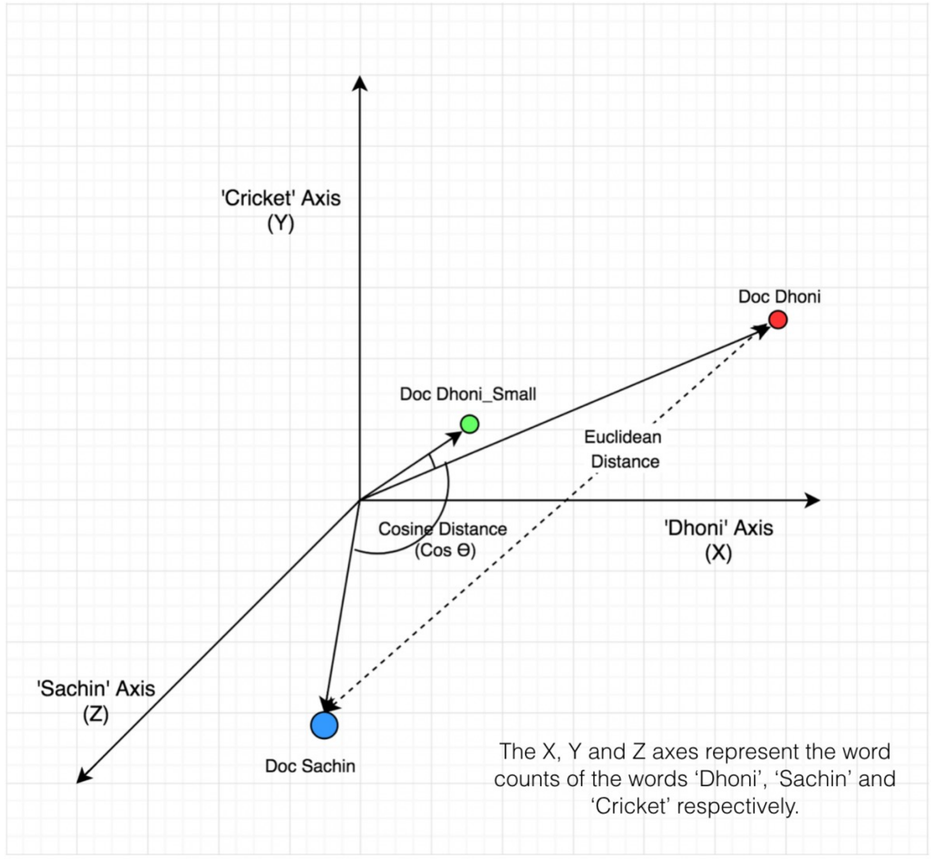
-> **Clustering**

Clustering Methods

1. Cosine Similarity of Album [2]
2. UMAP [3]
3. HDBSCAN [4]

Pre-Processing : Cosine Similarity

Projection of Documents in 3D Space



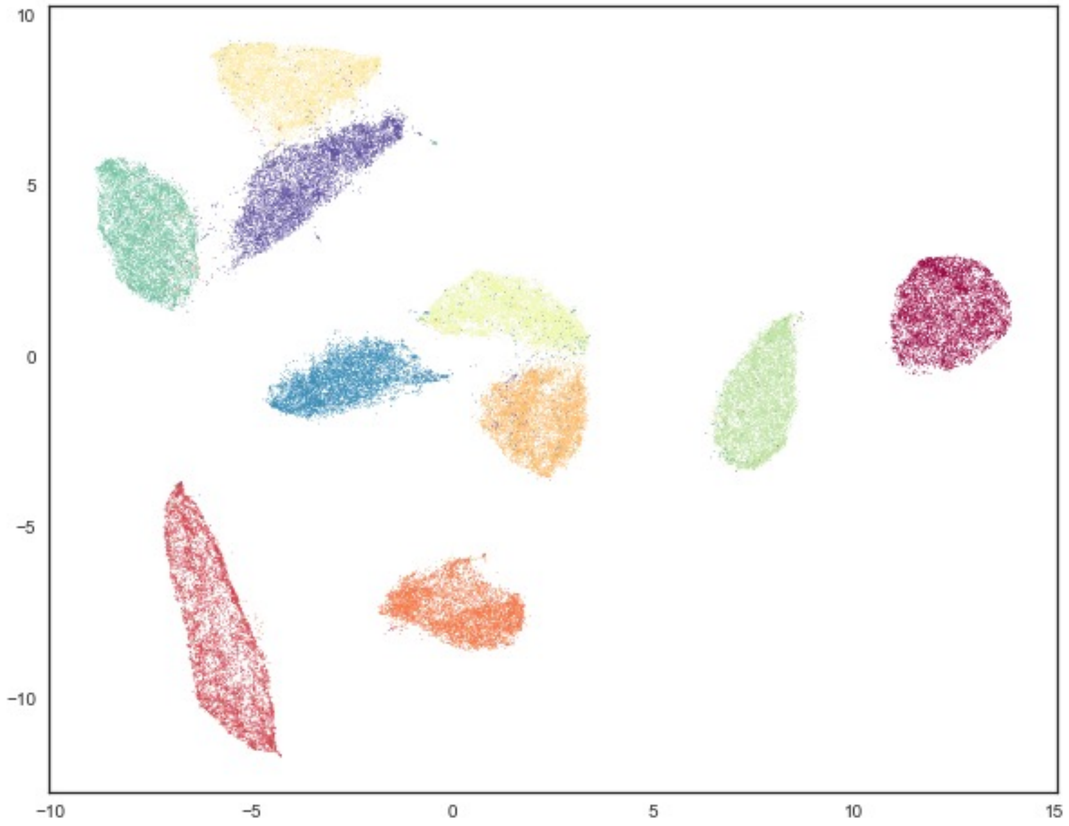
$$Score = rating\ mean + 0.5 * \sqrt{rating\ count}$$

	910	2517	5303	8324	10686	10758	14290	15761	16694	16721
910	1.000000	0.354922	0.142443	0.073738	0.261590	0.230487	0.411836	0.217362	0.384418	0.439335
2517	0.354922	1.000000	0.054260	0.023750	0.338906	0.293373	0.356595	0.205200	0.303396	0.336590
5303	0.142443	0.054260	1.000000	0.067580	0.041882	0.029121	0.074460	0.061805	0.070344	0.084377
8324	0.073738	0.023750	0.067580	1.000000	0.015969	0.013471	0.033933	0.033137	0.036152	0.056281
10686	0.261590	0.338906	0.041882	0.015969	1.000000	0.302106	0.339150	0.184586	0.246270	0.276887
...	...	...	...	...	...	...	...	...	...	...
18077	0.025422	0.014754	0.024641	0.000929	0.014415	0.010402	0.026461	0.032351	0.025865	0.023753
1099	0.009000	0.003928	0.005377	0.000885	0.005499	0.011274	0.008134	0.013499	0.005035	0.007127
5481	0.035835	0.012379	0.020928	0.035718	0.016390	0.011068	0.017434	0.023115	0.011353	0.018531
10109	0.026387	0.017797	0.009168	0.006868	0.012363	0.015879	0.019055	0.024506	0.019174	0.017022
4632	0.017184	0.008493	0.006997	0.001856	0.011343	0.005448	0.012030	0.008208	0.009537	0.011897

18318 rows × 18318 columns

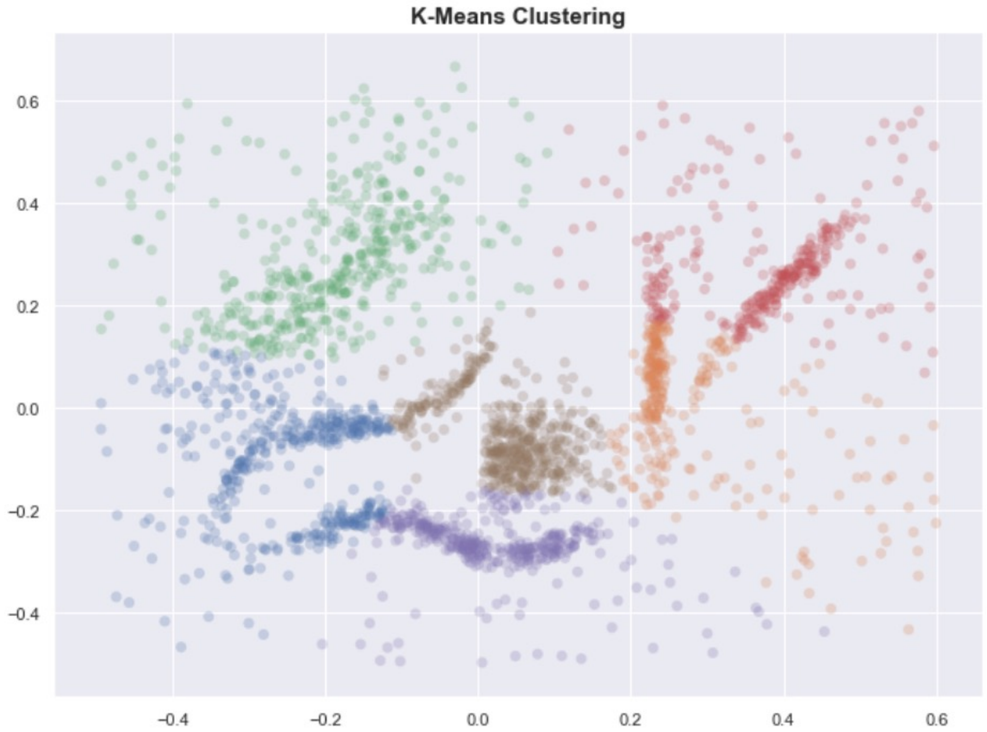
Pre-Processing : UMAP

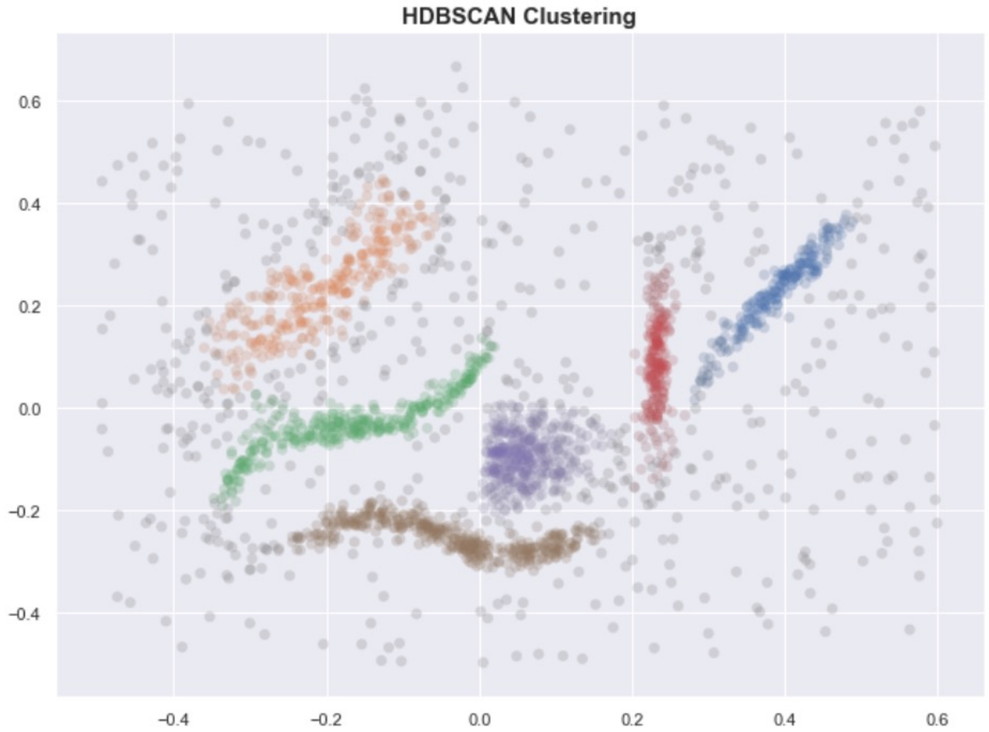
UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction



Pre-Processing : H-DBSCAN

HDBSCAN: Hierarchical Density-based Spatial Clustering of Applications with Noise





Pre-Processing : Rebuild Dataset

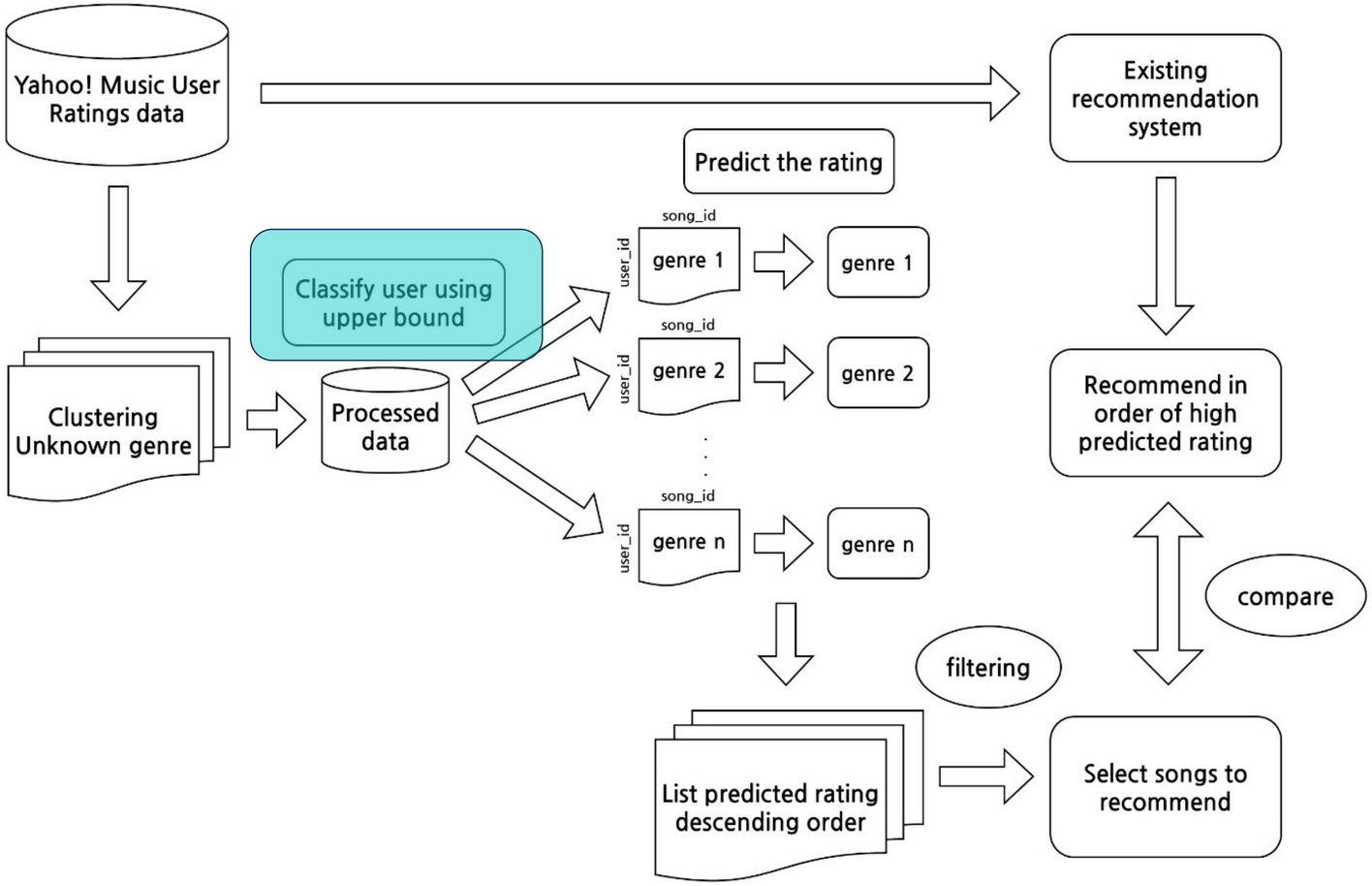
```
print('Total unique users in the dataset', df['user_id'].nunique())
print('Total unique songs in the dataset', df['song_id'].nunique())
print('Total unique category in the dataset', df['large_category'].nunique())
```

executed in 6.85s, finished 16:14:19 2021-06-12

```
Total unique users in the dataset 200000
Total unique songs in the dataset 136736
Total unique category in the dataset 37
```

$$Score = rating\ mean + 0.5 * \sqrt{rating\ count}$$

Experiment


Yahoo! Music User Ratings data
Existing recommendation system
Predict the rating
Classify user using upper bound
song_id
user_id
genre 1
genre 1
song_id
user_id
genre 2
genre 2
Clustering Unknown genre
Processed data
Recommend in order of high predicted rating
song_id
user_id
genre n
genre n
compare
filtering
List predicted rating descending order
Select songs to recommend

Experiment

1,category의 unknown을 'rating', '들은 수'를 이용해 분류한다.

```
df=df[['user_id','song_id','rating','large_category']]
```

2, 각 category 별로 평균점수를 계산한다.

```
category_mean = df.groupby(['user_id','large_category'],as_index=False).mean().drop(['song_id'],axis=1)
```

```
category_mean
```

```
# upperbound 추가
```

```
category_df['upperbound']=category_df['rating']+0.1*(category_df['number']**(1/2))
```

```
category_df.sort_values(by='upperbound', ascending=False)
category_df.sort_values(by='user_id')
```

```
df_user_category= category_df.loc[category_df.groupby(['user_id'])['upperbound'].idxmax()]
df_user_category
```

	user_id	large_category	rating	number	upperbound
3	0	Unknown_20	4.583333	12	4.929743
12	1	Unknown_10	5.000000	1	5.100000
29	2	Unknown_10	3.571429	84	4.487944
33	3	Country	4.000000	1	4.100000
47	4	R&B	5.000000	1	5.100000
...	...	...	...	...	...
125206	9995	Unknown_1	5.000000	1	5.100000
125222	9996	Unknown_10	4.920000	100	5.920000
125223	9997	Pop	5.000000	1	5.100000
125234	9998	Unknown_11	4.000000	1	4.100000
125237	9999	Unknown_0	4.875000	8	5.157843

10000 rows × 5 columns

1, make the dataframe with ‘user_id’, ‘song_id’, ‘rating’, ‘category’

2, calculate the average score of each category by each user

3, we also find out how many times each user has listened to each category

4, merge that two dataframes

5, finally, for each category, apply this data to upperbound expression to give the user the best song category

Experiment

For example,,, user_id = 0

user_id	song_id	rating	large_category
0	166	5	Unknown_0
0	2245	4	Unknown_22
0	3637	4	Unknown_20
0	5580	4	Unknown_16
0	5859	4	Unknown_0
0	7121	3	Unknown_0
0	10405	4	Unknown_0
0	16794	5	Unknown_0
0	21252	4	Unknown_20
0	27331	5	Unknown_20
0	32438	5	Unknown_20
0	32744	4	Unknown_22
0	34995	5	Unknown_20
0	35182	4	Unknown_20
0	52539	5	Unknown_20
0	55240	4	Unknown_0
0	68083	4	Unknown_0
0	69907	3	Unknown_0
0	73195	5	Unknown_22
0	77441	4	Unknown_0
0	78482	5	Unknown_20
0	86578	4	Unknown_20
0	90468	4	Unknown_0
0	96835	4	Unknown_20
0	98553	4	Unknown_0

73195	5	Unknown_22
77441	4	Unknown_0
78482	5	Unknown_20
86578	4	Unknown_20
90468	4	Unknown_0
96835	4	Unknown_20
98553	4	Unknown_0
119511	4	Unknown_22
119715	3	Unknown_0
123754	4	Unknown_22
128638	5	Unknown_20
128719	5	Unknown_0
133758	4	Unknown_0
136250	5	Unknown_20
17821	5	Rock
60088	4	Rock

$$\text{Score} = \text{rating mean} + 0.5 * \sqrt{\text{rating count}}$$

user_id	large_category	rating
0	Rock	4.500000
0	Unknown_0	4.000000
0	Unknown_16	4.000000
0	Unknown_20	4.583333
0	Unknown_22	4.200000

+

number
2
14
1
12
5

upperbound
4.641421
4.374166
4.100000
4.929743
4.423607

So, user_id = 0 → category = 'Unknown_20'

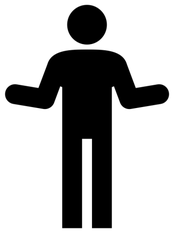
Experiment

Why we use this upperbound expression?

…. To give weight to rating count !!

$$\text{Score} = \text{rating mean} + 0.5 * \sqrt{\text{rating count}}$$

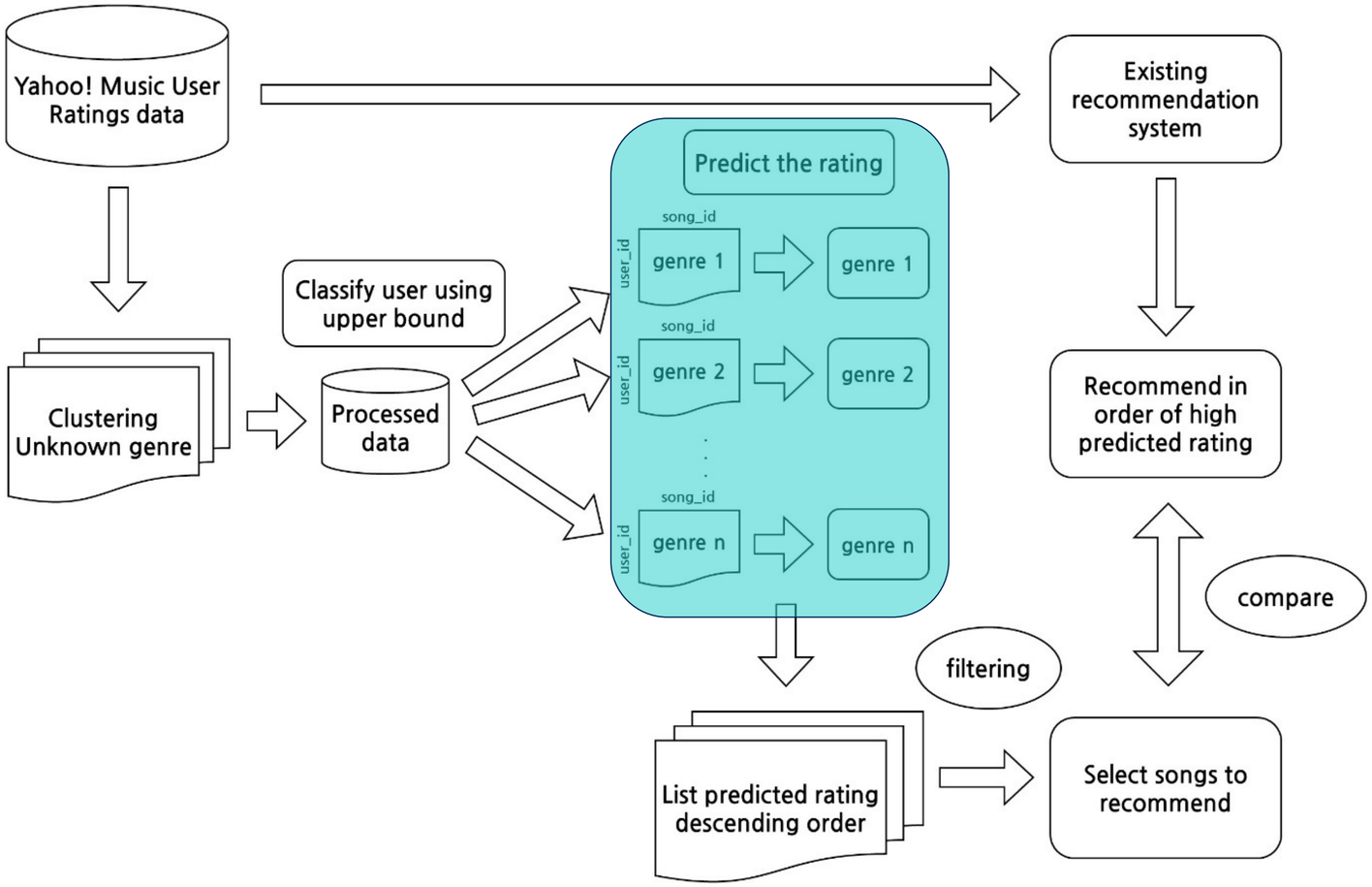
genre	Rating mean	Rating count
A	4	2
B	3.9	100



If only rating mean are considered
- The genre A will be picked
- But he only listen 2 times
  → not enough information

If rating count are also considered
- Genre A = 4 + 0.5*$\sqrt{2} = 4.7071$
- Genre B = 3.9 + 0.5*$\sqrt{100}$ = 8.9
- The genre B will be picked
- Sufficient infomation

Experiment


Yahoo! Music User Ratings data
Existing recommendation system
Predict the rating
song_id
user_id
genre 1
genre 1
Classify user using upper bound
song_id
genre 2
genre 2
Clustering Unknown genre
Processed data
Recommend in order of high predicted rating
song_id
genre n
genre n
compare
filtering
List predicted rating descending order
Select songs to recommend

Experiment

5, category에 해당하는 노래와 user간의 svd를 구한다.

```
best_category_df=df_user_category[['user_id','large_category']]
```

```
category_df_svd = pd.merge(best_category_df,df,on=['user_id','large_category'])
```

```
category_df_svd_df = category_df_svd[['user_id','song_id','rating','large_category']]
category_df_svd_df
```

```
category_svd_total = {}

for i in category_df_svd_df['large_category'].unique():
    genre_df = category_df_svd_df[category_df_svd_df['large_category']==i]

    df_matrix= genre_df.pivot(index='user_id',
                              columns='song_id',
                              values='rating').fillna(0)

    df_array = df_matrix.to_numpy()

    factorizer = MatrixFactorization(df_array, k=3, learning_rate=0.01, reg_param=0.01, epochs=300, verbose=True)
    factorizer.fit()

    category_svd = factorizer.get_complete_matrix()
    df_category_svd=pd.DataFrame(category_svd,
                             index=df_matrix.index,
                             columns=df_matrix.columns)

    category_svd_total[i]=df_category_svd
```

Experiment

Previous...

SGD(Stochastic Gradient Descent)

1, randomly select U, V^T matrices and calculate A

$$\underset{U}{\begin{bmatrix}1 & 2\\ 0 & 0\\ 3 & 0\end{bmatrix}} \quad \underset{V^T}{\begin{bmatrix}1 & 0\\ 1 & 2\end{bmatrix}} = \underset{\text{Estimated A}}{\begin{bmatrix}3 & 4\\ 0 & 0\\ 3 & 0\end{bmatrix}}$$

2, compare with real A matrices

$$\underset{\text{Estimated A}}{\begin{bmatrix}3 & 4\\ 0 & 0\\ 3 & 0\end{bmatrix}} \quad \underset{\text{real A}}{\begin{bmatrix}3 & ?\\ 0 & 5\\ ? & 0\end{bmatrix}} \quad \underset{U}{\begin{bmatrix}1 & 2\\ 0 & 0\\ 3 & 0\end{bmatrix}} \quad \underset{V^T}{\begin{bmatrix}1 & 0\\ 1 & 2\end{bmatrix}}$$

3, minimize the error and update $U\Sigma V^T$ matrices with L2 regularization

$$U_{new} = U - \eta(error * V + L_2 * U)$$
$$V_{new} = V - \eta(error * U + L_2 * V)$$

[0 -0.05]=[0 0]-0.005((5-0)*[0 2]+0.05*[0 0])
[0 1.9995]=[0 2]-0.005((5-0)*[0 0]+0.05*[0 2])

$$\underset{U}{\begin{bmatrix}1 & 2\\ 0 & -0.05\\ 3 & 0\end{bmatrix}} \quad \underset{V^T}{\begin{bmatrix}1 & 0\\ 1 & 1.9995\end{bmatrix}}$$

4, repeat this until we have smallest error!!!

UNIST FIRST IN CHANGE

```python
import numpy as np


class MatrixFactorization():
    def __init__(self, R, k, learning_rate, reg_param, epochs, verbose=False):
        """
        :param R: rating matrix
        :param k: latent parameter
        :param learning_rate: alpha on weight update
        :param reg_param: beta on weight update
        :param epochs: training epochs
        :param verbose: print status
        """

        self._R = R
        self._num_users, self._num_items = R.shape
        self._k = k
        self._learning_rate = learning_rate
        self._reg_param = reg_param
        self._epochs = epochs
        self._verbose = verbose


def gradient_descent(self, i, j, rating):
    """
    graident descent function

    :param i: user index of matrix
    :param j: item index of matrix
    :param rating: rating of (i,j)
    """

    # get error
    prediction = self.get_prediction(i, j)
    error = rating - prediction

    # update biases
    self._b_P[i] += self._learning_rate * (error - self._reg_param * self._b_P[i])
    self._b_Q[j] += self._learning_rate * (error - self._reg_param * self._b_Q[j])

    # update latent feature
    dp, dq = self.gradient(error, i, j)
    self._P[i, :] += self._learning_rate * dp
    self._Q[j, :] += self._learning_rate * dq
```

Experiment

For Unknown_20 category…

Dataframe[user_id, song_id, rating, category]

```
genre_df = category_df_svd_df[category_df_svd_df['large_category']=='Unknown_20']
genre_df
```

	user_id	song_id	rating	large_category
0	0	3637	4	Unknown_20
1	0	21252	4	Unknown_20
2	0	27331	5	Unknown_20
3	0	32438	5	Unknown_20
4	0	34995	5	Unknown_20
...	...	...	...	...
1147413	9970	75914	5	Unknown_20
1147414	9970	115031	5	Unknown_20
1147415	9970	122401	5	Unknown_20
1147416	9970	131065	5	Unknown_20
1148027	9986	75030	5	Unknown_20

9300 rows × 4 columns

Index = user_id
Columns = song_id
Values = rating

```
df_matrix= genre_df.pivot(index='user_id',
                          columns='song_id',
                          values='rating').fillna(0)
df_matrix
```

song_id	209	256	319	320	347	363	380	390	397	401	...	136250	136254	136336	136340	136370	136379	136518	136597	136699	136716
user_id																					
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	5.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
13	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
22	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
30	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
140	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
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
9933	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
9942	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	1.0	0.0
9963	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
9970	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
9986	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

395 rows × 2390 columns

Change dataframe into array

```
df_array = df_matrix.to_numpy()
df_array
```

```
array([[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.]])
```

Experiment

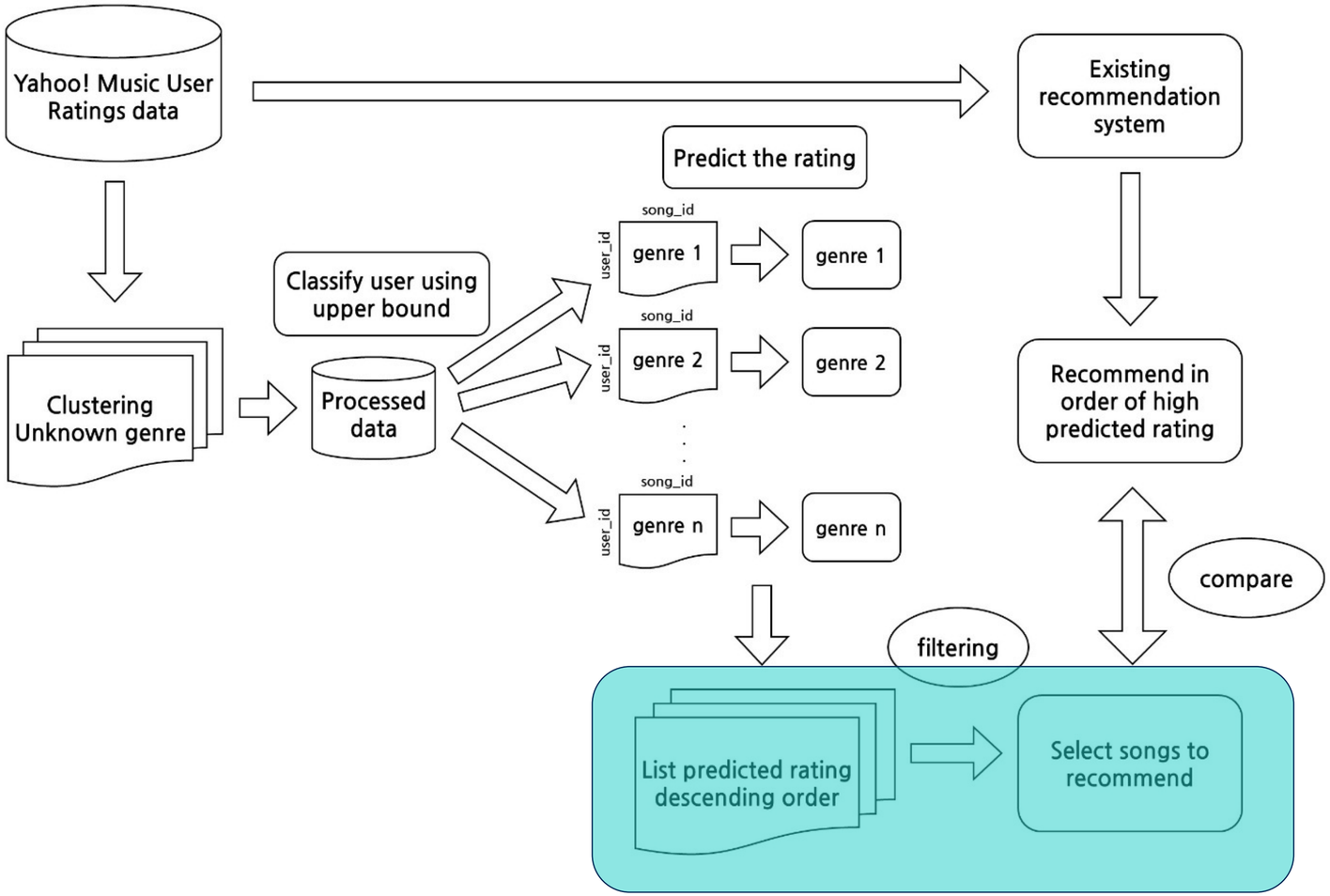
Finally we got predicted rating dataframe!

```
category_svd_total['Unknown_20']
```

song_id	209	256	319	320	347	363	380	390	397	401	...	136250	136254	136336	136340	136
user_id																
0	6.195019	6.449271	6.596405	7.890145	5.407030	7.220720	4.548959	8.809305	4.344454	3.405630	...	5.103801	8.718177	7.270592	5.669906	7.880
13	5.154584	5.088111	3.758022	5.999788	2.354009	5.013039	5.295045	5.767685	2.068105	1.539373	...	3.909732	3.506524	3.871671	2.081741	4.408
22	5.197904	4.900934	4.969257	4.773148	5.914698	3.759785	3.399978	2.979079	5.385024	3.693321	...	3.297526	2.913856	1.899497	4.439137	1.341
30	5.573394	5.500587	6.164991	5.774582	6.439957	5.549207	3.336723	5.331795	5.648836	4.254125	...	4.577280	5.777107	4.431024	6.167434	4.117
140	5.092162	4.961825	5.618705	4.950137	5.578059	5.551188	3.180080	4.820353	5.041632	4.033540	...	5.014455	4.782433	4.138933	6.103379	3.714
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
9933	5.870685	5.885545	4.690485	7.162332	3.835132	5.056001	5.404602	6.477275	3.157524	2.029342	...	3.344766	5.067381	4.333078	2.437361	4.974
9942	5.968544	5.664897	5.013396	5.838140	5.817167	3.859552	4.753227	3.601860	5.319756	3.557722	...	3.168194	2.949095	2.035176	3.534913	1.755
9963	5.708438	5.252317	4.653389	4.515174	4.102160	5.680179	5.633447	3.742030	4.475108	4.206331	...	6.428088	1.384173	2.976348	4.986486	2.386
9970	4.764461	4.791453	5.119446	5.339995	4.193139	5.880458	3.304524	5.999639	3.589113	2.933927	...	4.844363	5.491326	5.008085	5.183923	5.064
9986	6.553778	6.260283	5.194313	6.529342	5.338383	4.838235	6.010838	4.685240	5.039598	3.708089	...	4.227148	3.219032	2.991208	3.584629	2.914

395 rows × 2390 columns

Experiment


Yahoo! Music User Ratings data
Existing recommendation system
Predict the rating
song_id
user_id
genre 1
genre 1
Classify user using upper bound
song_id
user_id
genre 2
genre 2
Clustering Unknown genre
Processed data
Recommend in order of high predicted rating
song_id
user_id
genre n
genre n
compare
filtering
List predicted rating descending order
Select songs to recommend

Experiment

6. 예측된 점수가 높은 순서대로 top100 노래를 선택한다.

```
top_df = pd.DataFrame()
nlargest = 100


for i in category_svd_total:
    order = np.argsort(-category_svd_total[i].values, axis=1)[:, :nlargest]
    if category_svd_total[i].shape[1]<100:
        result = pd.DataFrame(category_svd_total[i].columns[order],
                        columns=['top{}'.format(i) for i in range(1, category_svd_total[i].shape[1]+1 )],
                        index=category_svd_total[i].index)
    else:
        result = pd.DataFrame(category_svd_total[i].columns[order],
                              columns=['top{}'.format(i) for i in range(1, nlargest+1)],
                              index=category_svd_total[i].index)

    top_df = pd.concat([top_df,result])
```

```
top_df
```

user_id	top1	top2	top3	top4	top5	top6	top7	top8	top9	top10	...	top91	top92	top93	top94	top95
0	134980	82690	101013.0	77735.0	27032.0	8375.0	16451.0	118601.0	32256.0	118313.0	...	390.0	96434.0	783.0	59533.0	68906.0
13	134980	20864	8375.0	82690.0	36378.0	102329.0	77789.0	57524.0	22821.0	69575.0	...	27032.0	3889.0	136005.0	107343.0	53647.0
22	57800	114455	61371.0	9589.0	108247.0	39275.0	27009.0	76485.0	131226.0	37009.0	...	71462.0	8144.0	119608.0	131065.0	66918.0
30	57800	92172	61371.0	44958.0	115405.0	85514.0	103558.0	7723.0	51457.0	32855.0	...	28873.0	34154.0	4368.0	69411.0	47763.0
140	57800	103558	44958.0	7723.0	51457.0	92172.0	39088.0	78356.0	35697.0	61371.0	...	91046.0	17539.0	76117.0	38043.0	114018.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
7203	94790	98775	80694.0	8560.0	73917.0	87677.0	61441.0	85065.0	9576.0	9226.0	...	NaN	NaN	NaN	NaN	NaN
7363	70111	94790	17285.0	126901.0	875.0	84379.0	52366.0	48197.0	85065.0	132268.0	...	NaN	NaN	NaN	NaN	NaN
8272	87677	84379	131899.0	98775.0	85065.0	54312.0	132798.0	80694.0	16473.0	17285.0	...	NaN	NaN	NaN	NaN	NaN
8744	9226	94790	59363.0	8560.0	9576.0	98775.0	61441.0	73917.0	10134.0	16473.0	...	NaN	NaN	NaN	NaN	NaN
9099	84379	131899	98775.0	16473.0	54312.0	9576.0	94790.0	8560.0	132268.0	74109.0	...	NaN	NaN	NaN	NaN	NaN

10000 rows × 100 columns

Experiment

	RMSE	RCE	Computing Time
KNN			
SVD			
SGD			
Our Model			

Conclusion

1. RMSE :

2. RCE :

3. Computing Time:

Reference

[1] Belli, Luca, et al. "Privacy-Aware Recommender Systems Challenge on Twitter's Home Timeline." arXiv e-prints (2020): arXiv-2004.

[2] Asanov, Daniar. "Algorithms and methods in recommender systems." Berlin Institute of Technology, Berlin, Germany (2011).

[3] McInnes, Leland, John Healy, and James Melville. "Umap: Uniform manifold approximation and projection for dimension reduction." arXiv preprint arXiv:1802.03426 (2018).

[4] McInnes, Leland, John Healy, and Steve Astels. "hdbscan: Hierarchical density based clustering." Journal of Open Source Software 2.11 (2017): 205.

THANK YOU