연세대학교
YONSEI UNIVERSITY

개인화 추천시스템

기말 프로젝트 보고서

DA 2022311907 이종우
DA 2022311916 조규원
DA 2022311930 김영채
DA 2022311934 장서호

[목차]

1. Introduction

1) Amazon 데이터를 사용하여 RMSE와 F1 score를 기준으로 정확한 rating을 예측하는 알고리즘의 최적의 조합을 찾는 것을 목표로 진행하였습니다.

[1] 사용 데이터
Rating 데이터의 user_id, item_id, rating을 기본 변수로 사용하였고, timestamp변수를 이용하여 year, month, day, season(4계절:winter[0], spring[1], summer[2], fall[3]) 4개의 파생변수를 생성하였습니다.

	item_id	user_id	rating	year	month	day	season
0	A0955928C2RRWOWZN7UC	B00191WVF6	4.0	2017	2	17	0
1	A0955928C2RRWOWZN7UC	B005WY3TMA	4.0	2015	6	14	2
2	A0955928C2RRWOWZN7UC	B0090XWU8S	4.0	2017	4	22	1
3	A0955928C2RRWOWZN7UC	B00FXYTLIK	4.0	2015	7	10	2
4	A0955928C2RRWOWZN7UC	B00HMZG3YS	4.0	2015	7	10	2
...	...	...	...	...	...	...	...
99737	AZZYW4YOE1B6E	B009AYLDSU	5.0	2013	12	10	0
99738	AZZYW4YOE1B6E	B00E055H5O	4.0	2015	5	25	1
99739	AZZYW4YOE1B6E	B00E8HGWIK	5.0	2013	12	1	0
99740	AZZYW4YOE1B6E	B00M58CMTM	5.0	2014	10	8	3
99741	AZZYW4YOE1B6E	B00VWVZ0V0	4.0	2016	12	27	0

99742 rows × 7 columns

Review 데이터를 join하여 rating 데이터와 함께 사용하려고 시도했으나, join 후 Train/Test를 나눴을 때 test set에서 user가 item을 사기도 전에 text(review)가 있게 됩니다. 이렇게 되면 rating 예측에 text가 치팅이 되어버리게 됩니다. 그래서 review 데이터는 사용하지 못하고 rating 데이터만 사용하였습니다.

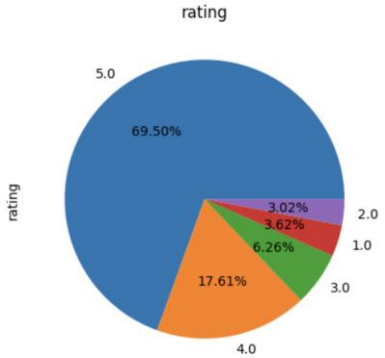
2) Metapath2vec을 이용하여 user에 따라 item 임베딩 후 시계열 데이터로 변환시켜 LSTM을 사용하여 rating을 예측하는 알고리즘을 개발하였습니다.

user별로 구분하기 위해서 user별 rating을 one-hot encoding으로 붙여서 user embedding으로 사용하였습니다. LSTM에서 서로 다른 user끼리의 embedding은 멀어지고 동일한 user의 embedding은 가까워지도록 조정하여 User embedding을 구한 후 이를 cf_knn에 적용하여 rating값을 예측한 후 rmse를 계산하였습니다.

2. EDA & IDEA

1) rating의 분포

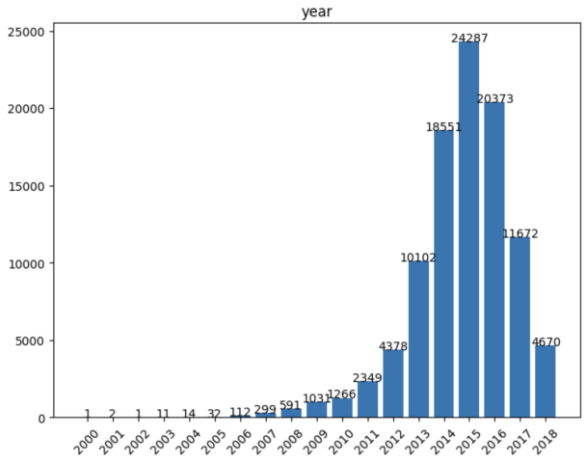
```
count    99742.000000
mean         4.463305
std          0.994834
min          1.000000
25%          4.000000
50%          5.000000
75%          5.000000
max          5.000000
Name: rating, dtype: float64
```



평균이 4.46인 것을 이용하여 성능을 올리기 위하여 모델에 사용되는 mean_rating의 값을 4.0으로 지정하였습니다.

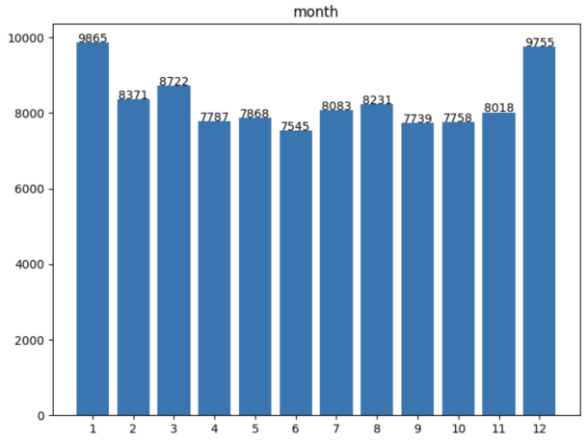
2) 파생변수 분포

[1] year의 분포



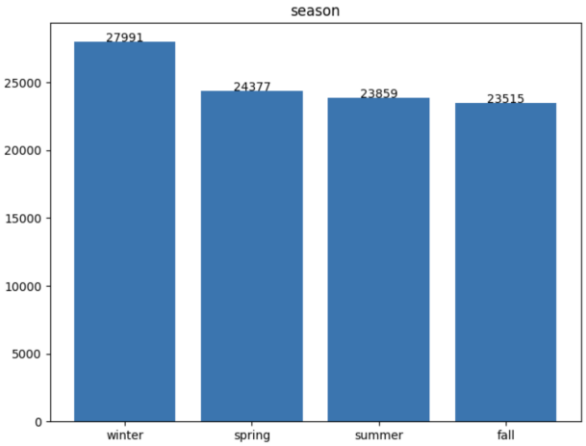
다음으론 파생변수로 만든 ['year']의 분포입니다. 데이터가 2000년~2018년도로, 2015년도로 갈수록 더 많은 데이터가 분포해 있다가 다시 줄어드는 분포인 것을 알 수 있습니다.

[2] month의 분포



위의 그래프는 파생변수 중 ['month']입니다. 1월과 12월에 가장 많은 분포를 보이고 있습니다. 연말 또는 연초에 물건을 많이 구입한 것을 알고 있습니다. 1월과 12월이 가장 많기 때문에 추후 파생변수 ['season']에서도 겨울이 가장 많이 차지하지 않을까 예상합니다.

[3] season의 분포



저희는 12/1/2월을 winter로, 3/4/5월을 spring, 그리고 6/7/8월을 summer, 9/10/11월을 가을로 지정했습니다.

3. Embedding with Metapath2vec & LSTM

1) 이론

[Metapath2vec]

이전까지 나온 많은 graph embedding 모델들은 대부분 Homogeneous Graph를 다루고 있다는 한계점이 있습니다. 그러나 현실의 그래프는 Heterogeneous Graph로 여러 종류의 노드와 엣지를 가지고 있습니다. Heterogeneous Graph 상의 임의의 node n_0에서 연결된 edge를 따라 $m-1$번 이동해서 다른 node n_m에 도착했다고 가정하면 다음와 같은 path가 생성됩니다.

$$n_0 \xrightarrow{e_1} n_1 \xrightarrow{e_2} \cdots \xrightarrow{e_{m-1}} n_{m-1} \xrightarrow{e_m} n_m$$

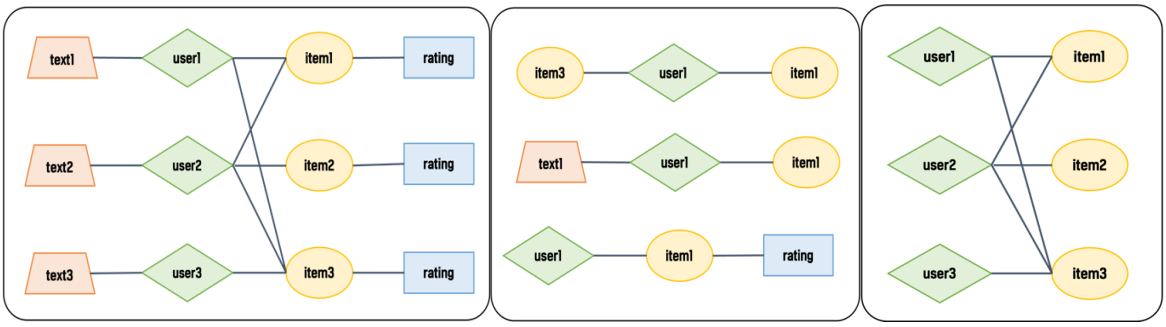
여기서, K번째 node의 type을 o_k, edge의 type을 l_k로 하면 위의 path는 아래와 같이 표현됩니다.

$$o_0 \xrightarrow{l_1} o_1 \xrightarrow{l_2} \cdots \xrightarrow{l_{m-1}} o_{m-1} \xrightarrow{l_m} o_m$$

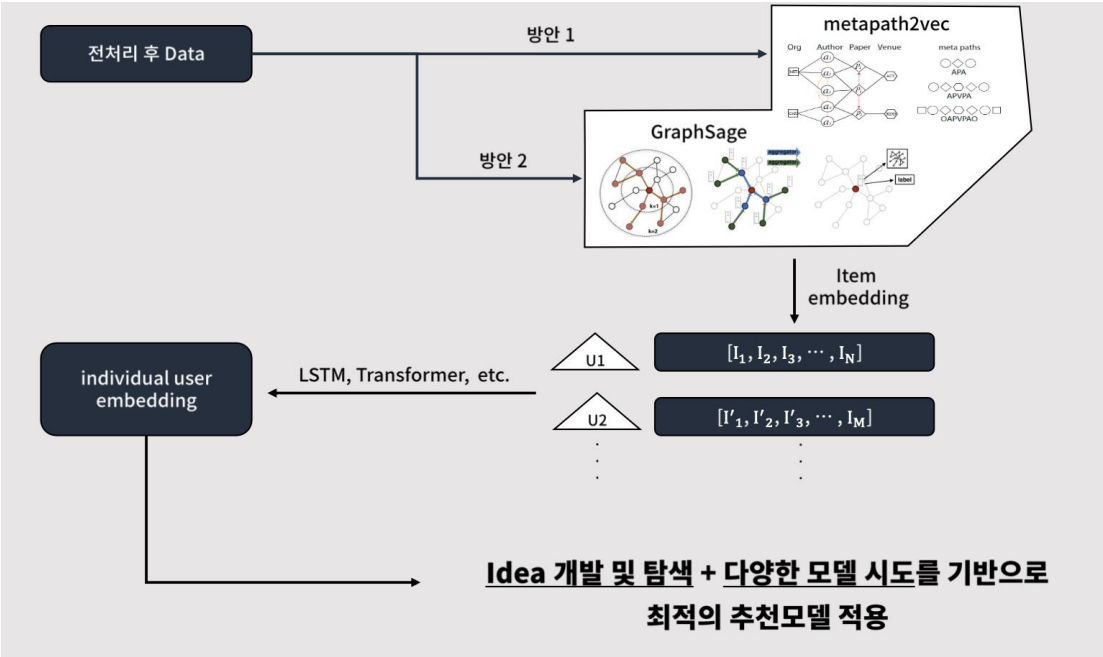
위와 같이 표현된 node와 edge의 type을 만족하는 path들을 metapath라고 합니다. o_k, l_k에는 실제 이어진 같은 type의 node나 edge라면 무엇이든 올 수 있습니다.

이러한 metapath 방법을 베이스로 metapath2vec, HIN2vec, HERec등의 Heterogeneous Graph 구조를 저차원의 벡터로 임베딩하는 여러 방법이 제안되었습니다.
그 중 Metapath2vec은 Node 종류에 따른 고유한 정보를 통해 다른 관계를 찾는 목적을 갖고 random walk 방법론에 기반하여 학습할 임의의 path들을 생성하는 대신, node의 종류에 따른 관계의 고유한 정보를 반영하는 적절한 path만 학습합니다.

Ratings data와 Reviews data로 나타낸 Heterogeneous Graph/Metapaths/Metapath-based Graph



[Heterogeneous Graph] [Metapaths] [Metapath-based Graph]



2) 방법

1. Metapath2vec을 이용해 item들을 embedding

```
metapath = [('item','purchased_by','user'),
            ('user','purchase','item')]
```

2. 각 item embedding 뒤에 rating score를 one-hot encoding으로 붙여 user를 특정할 수 있게 만들고 Item embedding을 시간 순으로 정렬해 유저의 구매 내역 생성

```
res_dict[group_name].append(cur_embedding+rate_embedding)
```

3. LSTM에 넣기 전에 몇 개 이상의 sequence를 가진 user만 살릴 건지 정하기

```
for idx in range(len(set(res_dict.keys())  & set(user_list))):
    _id = user_list[idx]
    if len(res_dict[_id]) > 3:  # 몇개 sequence 이상만 남길거냐 (중요)
```

```
def __init__(self, dim_in, dim_h, dim_out, lstm_layer=4, seq_length=4):
```

4. LSTM에 넣을 때 몇 개의 sequence를 기준으로 학습시킬 것인지 정하기

5. 동일 user의 embedding인지 아닌지 구분할 수 있게 학습

```
if len(pos_idx_list) != 0:
    pos_h = h[pos_idx_list]
    pos_loss = -1 * torch.log(pos_h + 1e-7).mean()

if len(neg_idx_list) != 0:
    neg_h = h[neg_idx_list]
    neg_loss = -1 * torch.log(1-neg_h + 1e-7).mean()
```

   1) 동일 user이면 pos_loss
   2) 다른 user이면 neg_loss

```
loss = pos_loss + neg_loss
```

6. 학습시킨 LSTM을 이용해 만들어진 user embedding을 이용, cf_bias_knn으로 user 평점 구하기
7. 아이디어로 embedding된 거리에 반비례하게 기록을 반영하도록 모델 생성

```
dist_f = 1 / (euc_dist_matrix1 + 1e-7)

residual = np.dot(item_ratings, dist_f) / dist_f.sum()

mean_rating = user_mean + residual
```

3) 결과

[metapath2vec with lstm]

	코드 실행 결과	특이사항	RMSE
1회차	0.9768502642667779	K = 90 Rand_int = 12, 15, 18	0.9768
2회차	0.9737212475427675		0.9737
3회차	0.9646687645069192		0.9646
평균			0.9717

	코드 실행 결과	특이사항	RMSE
1회차	0.9716478722497782	K = 100 Rand_int = 12, 15, 18	0.9716
2회차	0.9626835226564142		0.9627
3회차	0.962059496142738		0.9621
평균			0.9654

4. Modeling

- 사용 모델

CF/CF_KNN/IBCF/IBCF_KNN/UBCF/UBCF(파생)/UBCF_KNN
FM/FM(파생)/MF/DL/AE/SVD/SVD(파생)/MF+IBCF/MF+DL/CF+MF+DL

먼저, 하나의 모델만 비교했을 때, 다음과 같습니다.

[CF]

	코드 실행 결과	특이사항	RMSE
1회차	1.0780823520720648	mean_rating = 4.0	1.0780
2회차	1.0831559488614027		1.0831
3회차	1.0847632125452185		1.0847
평균			1.0819

Mean_ratings을 4.0으로 맞추었습니다. 위에 EDA에서 보시면, ratings의 평균이 4.46인 것을 이용하여 4.0으로 맞추어 진행하였습니다.

[CF_KNN]

	코드 실행 결과	특이사항	RMSE
1회차	1.071630041506556	K:30, mean_rating=4.0	1.0716
2회차	1.0833631415285878		1.0833
3회차	1.0751248149411898		1.0751
평균			1.0766

하이퍼 파라미터 튜닝 결과, K가 30일 때 가장 높은 성능을 보여 k를 30으로 설정하였습니다.

다음은 IBCF와 UBCF의 성능입니다.

[UBCF]

	코드 실행 결과	특이사항	RMSE
1회차	score(ubcf_bias)	mean_rating = 3.0	1.0590
2회차	score(ubcf_bias)		1.0563
3회차	score(ubcf_bias)		1.0580
평균			1.0578

[UBCF]

	코드 실행 결과	특이사항	RMSE
1회차	score(ubcf_bias)	mean_rating = 4.0	1.0590
2회차	score(ubcf_bias)		1.0563
3회차	score(ubcf_bias)		1.0580
평균			1.0578

[UBCF]

	코드 실행 결과	특이사항	RMSE
1회차	score(ubcf_bias)	mean_rating = 5.0	1.0430
2회차	score(ubcf_bias)		1.0440
3회차	score(ubcf_bias)		1.0510
평균			1.0460

UBCF를 가지고 mean_ratings을 몇으로 지정하냐에 따른 RMSE값을 비교해보았습니다. 먼저, 3.0의 경우, 제일 낮은 RMSE값을 보였습니다. 4.0으로 지정하여 돌려본 결과, 3.0보다 성능이 좋았으며, 가장 ratings의 빈도가 많은 5.0으로 지정하였을 때가 가장 성능이 높게 나왔습니다.

UBCF에서 KNN을 합친 UBCF_KNN모델을 시도해보았습니다.

[UBCF_KNN]

	코드 실행 결과	특이사항	RMSE
1회차	`score(ubcf_bias_knn, 30)`	mean_rating = 3.0 neighbor = 30	1.0472
2회차	`score(ubcf_bias_knn, 30)`		1.0547
3회차	`score(ubcf_bias_knn, 30)`		1.0504
평균			1.0508

가장 기본값(3.0)으로 지정하여 실행해 보았습니다. neighbor은 30일 때 최적이므로 30으로 동일하게 지정합니다.

[UBCF_KNN]

	코드 실행 결과	특이사항	RMSE
1회차	`score(ubcf_bias_knn, 30)`	prediction = 4.0 neighbor = 30	1.0589
2회차	`score(ubcf_bias_knn, 30)`		1.0506
3회차	`score(ubcf_bias_knn, 30)`		1.0576
평균			1.0557

다음으로 위에서 mean_rating을 UBCF_KNN에 대해서도 4.0으로 시도해보았습니다.

다음으론 IBCF입니다.

[IBCF]

	코드 실행 결과	특이사항	RMSE
1회차	`1.1175454968664 31`	mean_rating=4.0	1.1175
2회차	`1.1284085595416622`		1.1284
3회차	`1.1306105898571668`		1.1306
평균			1.1255

[IBCF_KNN]

	코드 실행 결과	특이사항	RMSE
1회차	`1.1332034419486006`	K=30, mean_rating=4.0	1.1332
2회차	`1.1234545274245704`		1.1234
3회차	`1.128551756702786`		1.1285
평균			1.1283

SVD의 경우, 최종 rmse값을 구하기 위해서는 많은 CPU 메모리와 시간이 필요합니다. 그래서 SVD는 한 번 실행하였습니다. 이때 파라미터 튜닝을 통해 최적의 beta값과 최적의 lamda값을 도출하였습니다.

[SVD]

	코드 실행 결과	특이사항	RMSE
1회차	102 0.9348955044913322	beta=0.05, lamda=0.0012	0.9348

추후 SVD에 파생변수를 추가하여 진행하였지만 RMSE 값에 거의 변화는 없었습니다.

[SVD+파생변수]

	코드 실행 결과	특이사항	RMSE
1회차	Iteration: 10 ; Train RMSE = 0.942350 ; Test RMSE = 0.980951 Iteration: 20 ; Train RMSE = 0.912556 ; Test RMSE = 0.964649 Iteration: 30 ; Train RMSE = 0.890901 ; Test RMSE = 0.954096 Iteration: 40 ; Train RMSE = 0.874519 ; Test RMSE = 0.947015 Iteration: 50 ; Train RMSE = 0.861828 ; Test RMSE = 0.942253 Iteration: 60 ; Train RMSE = 0.851829 ; Test RMSE = 0.939087 Iteration: 70 ; Train RMSE = 0.843838 ; Test RMSE = 0.937046 Iteration: 80 ; Train RMSE = 0.837370 ; Test RMSE = 0.935822 Iteration: 90 ; Train RMSE = 0.832071 ; Test RMSE = 0.935187 Iteration: 100 ; Train RMSE = 0.827675 ; Test RMSE = 0.934969		0.9349

다음은 MF에 대해서 시도해 보았습니다. (처음으로 RMSE값이 1보다 작아졌습니다.!)

[MF]

	코드 실행 결과	특이사항	RMSE
1회차	Iteration: 10 ; Train RMSE = 0.865913 ; Test RMSE = 0.931891 Iteration: 20 ; Train RMSE = 0.830278 ; Test RMSE = 0.925167 19 0.9251668267100773		0.9251
2회차	Iteration: 10 ; Train RMSE = 0.868233 ; Test RMSE = 0.927199 Iteration: 20 ; Train RMSE = 0.832764 ; Test RMSE = 0.920066 19 0.9200658484908479		0.9200
3회차	Iteration: 10 ; Train RMSE = 0.866456 ; Test RMSE = 0.931832 Iteration: 20 ; Train RMSE = 0.830650 ; Test RMSE = 0.924539 19 0.9245391327840744		0.9245
평균			0.9232

다음으로 FM을 시도해보았습니다.

[FM]

	코드 실행 결과	특이사항	RMSE
1회차	Iteration: 10 ; Train RMSE = 0.921605 ; Test RMSE = 0.955357 Iteration: 20 ; Train RMSE = 0.881820 ; Test RMSE = 0.938234 Iteration: 30 ; Train RMSE = 0.857637 ; Test RMSE = 0.930554 Iteration: 40 ; Train RMSE = 0.842035 ; Test RMSE = 0.927552 Iteration: 50 ; Train RMSE = 0.831558 ; Test RMSE = 0.927036		0.9270

2회차	Iteration: 10 ; Train RMSE = 0.926058 ; Test RMSE = 0.944277 Iteration: 20 ; Train RMSE = 0.886486 ; Test RMSE = 0.926901 Iteration: 30 ; Train RMSE = 0.862481 ; Test RMSE = 0.918910 Iteration: 40 ; Train RMSE = 0.846979 ; Test RMSE = 0.915607 Iteration: 50 ; Train RMSE = 0.836545 ; Test RMSE = 0.914831		0.9148
3회차	Iteration: 10 ; Train RMSE = 0.921693 ; Test RMSE = 0.954398 Iteration: 20 ; Train RMSE = 0.881749 ; Test RMSE = 0.937684 Iteration: 30 ; Train RMSE = 0.857487 ; Test RMSE = 0.930175 Iteration: 40 ; Train RMSE = 0.841825 ; Test RMSE = 0.927276 Iteration: 50 ; Train RMSE = 0.831303 ; Test RMSE = 0.926836		0.9268
평균			0.9229

그 다음 FM에 파생변수(year/month/days/season)를 넣었습니다.

[FM + context]

	코드 실행 결과	특이사항	RMSE
1회차	Iteration: 10 ; Train RMSE = 0.982534 ; Test RMSE = 0.983480 Iteration: 20 ; Train RMSE = 0.971543 ; Test RMSE = 0.976783 Iteration: 30 ; Train RMSE = 0.960168 ; Test RMSE = 0.970535 Iteration: 40 ; Train RMSE = 0.946086 ; Test RMSE = 0.963521 Iteration: 50 ; Train RMSE = 0.926068 ; Test RMSE = 0.954406 Iteration: 60 ; Train RMSE = 0.898960 ; Test RMSE = 0.943624 Iteration: 70 ; Train RMSE = 0.867233 ; Test RMSE = 0.933749 Iteration: 80 ; Train RMSE = 0.834711 ; Test RMSE = 0.927628 Iteration: 90 ; Train RMSE = 0.803472 ; Test RMSE = 0.926503	**파생변수**	0.9263
2회차	Iteration: 10 ; Train RMSE = 0.979162 ; Test RMSE = 0.991690 Iteration: 20 ; Train RMSE = 0.968416 ; Test RMSE = 0.984903 Iteration: 30 ; Train RMSE = 0.957532 ; Test RMSE = 0.978677 Iteration: 40 ; Train RMSE = 0.944521 ; Test RMSE = 0.971894 Iteration: 50 ; Train RMSE = 0.926356 ; Test RMSE = 0.963247 Iteration: 60 ; Train RMSE = 0.901382 ; Test RMSE = 0.952653 Iteration: 70 ; Train RMSE = 0.871155 ; Test RMSE = 0.942075 Iteration: 80 ; Train RMSE = 0.839116 ; Test RMSE = 0.934351 Iteration: 90 ; Train RMSE = 0.807717 ; Test RMSE = 0.931199		0.9311
3회차	Iteration: 10 ; Train RMSE = 0.984138 ; Test RMSE = 0.979574 Iteration: 20 ; Train RMSE = 0.973246 ; Test RMSE = 0.972880 Iteration: 30 ; Train RMSE = 0.962170 ; Test RMSE = 0.966749 Iteration: 40 ; Train RMSE = 0.948886 ; Test RMSE = 0.960136 Iteration: 50 ; Train RMSE = 0.930313 ; Test RMSE = 0.951713 Iteration: 60 ; Train RMSE = 0.904745 ; Test RMSE = 0.941372 Iteration: 70 ; Train RMSE = 0.873685 ; Test RMSE = 0.931019 Iteration: 80 ; Train RMSE = 0.840540 ; Test RMSE = 0.923653 Iteration: 90 ; Train RMSE = 0.807785 ; Test RMSE = 0.921177		0.9211
평균			0.9261

다음으로는 AutoEncoder와 DL 입니다.

[AutoEncoder]

	코드 실행 결과	특이사항	RMSE
1회차	0.9576460439536624		0.9576
2회차	0.9731435794203123		0.9731
3회차	0.9645056291909657		0.9645
평균			0.9650

[DL]

	코드 실행 결과	특이사항	RMSE
1회차	1.0319313261266707		1.0319
2회차	0.9861546531546906		0.9861
3회차	1.0089845964362796		1.0090
평균			1.009

단독으로 모델을 돌릴 때에는 FM이 가장 낮은 RMSE값을 보여주며, MF가 두번째로 RMSE 값이 낮습니다.

다음으로는 hybrid 모델입니다.

기본 CF(IBCF) 모델과 MF을 하이브리드 했습니다.

[MF + IBCF]

	코드 실행 결과	특이사항	F1
1회차	Hybrid: (0.010778985507246376, 0.032751568635186475, 0.013801614369882989)		0.0138
2회차	Hybrid: (0.010631950094928127, 0.032277653411441544, 0.013641918207746828)		0.0136
3회차	Hybrid: (0.010385032537960954, 0.031862346672789886, 0.013377042125164296)		0.0133
평균			0.0135

[MF + DL]

	코드 실행 결과	특이사항	RMSE
1회차	Weights - 0.91 : 0.09 ; RMSE = 0.9201608 Weights - 0.92 : 0.08 ; RMSE = 0.9197535 Weights - 0.93 : 0.07 ; RMSE = 0.9194078 Weights - 0.94 : 0.06 ; RMSE = 0.9191238 Weights - 0.95 : 0.05 ; RMSE = 0.9189016 Weights - 0.96 : 0.04 ; RMSE = 0.9187412 Weights - 0.97 : 0.03 ; RMSE = 0.9186427 **Weights - 0.98 : 0.02 ; RMSE = 0.9186060** Weights - 0.99 : 0.01 ; RMSE = 0.9186312	MF : 0.98 DL : 0.02 + 파생변수	0.9186
2회차	Weights - 0.91 : 0.09 ; RMSE = 0.9137624 Weights - 0.92 : 0.08 ; RMSE = 0.9133021 Weights - 0.93 : 0.07 ; RMSE = 0.9129101 Weights - 0.94 : 0.06 ; RMSE = 0.9125864 Weights - 0.95 : 0.05 ; RMSE = 0.9123310 Weights - 0.96 : 0.04 ; RMSE = 0.9121441 Weights - 0.97 : 0.03 ; RMSE = 0.9120256 **Weights - 0.98 : 0.02 ; RMSE = 0.9119756** Weights - 0.99 : 0.01 ; RMSE = 0.9119941	MF : 0.98 DL : 0.02 + 파생변수	0.9120
3회차	Weights - 0.91 : 0.09 ; RMSE = 0.9316310 Weights - 0.92 : 0.08 ; RMSE = 0.9312823 Weights - 0.93 : 0.07 ; RMSE = 0.9310000 Weights - 0.94 : 0.06 ; RMSE = 0.9307839 Weights - 0.95 : 0.05 ; RMSE = 0.9306342 Weights - 0.96 : 0.04 ; RMSE = 0.9305509 **Weights - 0.97 : 0.03 ; RMSE = 0.9305340** Weights - 0.98 : 0.02 ; RMSE = 0.9305834 Weights - 0.99 : 0.01 ; RMSE = 0.9306993	MF : 0.97 DL : 0.03 + 파생변수	0.9305
평균			0.9204

CF는 IBCF보다 UBCF모델이 더 낮은 RMSE값을 보였기 때문에 ubcf를 채택하여 MF와 DL까지 Hybrid하여 돌려보았습니다. 최적의 모델 가중치 비율을 찾기 위해 하이퍼 파라미터를 조절 하였습니다. 다음 결과가 저희가 찾은 최적의 모델 가중치 조합입니다.

[CF+MF+DL]

	코드 실행 결과	특이사항	RMSE
1회차	최고의 황금비율 MF : '0.91' ubcf_bias_knn : '0.07' DL : '0.02' RMSE = 0.9252566	MF : 0.91 CF : 0.07 DL : 0.02	0.9252
2회차	최고의 황금비율 MF : '0.91' ubcf_bias_knn : '0.07' DL : '0.02' RMSE = 0.9205714	MF : 0.91 CF : 0.07 DL : 0.02	0.9206
3회차	최고의 황금비율 MF : '0.91' ubcf_bias_knn : '0.08' DL : '0.01' RMSE = 0.9159859	MF : 0.91 CF : 0.08 DL : 0.01	0.9160
평균			0.9206

5. 최종 선택 모델

metapath2vec을 이용하여 embedding후 LSTM에 적용한 알고리즘의 경우 RMSE값이 0.9654가 나왔습니다. 그리고 RMSE값을 낮추기 위해, 기존 모델들을 다양한 방식으로 hybrid를 시도해 보았습니다. 단독보다는 2개의 hybrid model이, 2개의 hybrid model보다는 3개의 hybrid model이 더 좋은 성능을 나타냈습니다.

최종적으로 저희가 선택한 모델은 가장 낮은 RMSE값을 보인 3개의 하이브리드 모델, **CF+MF+DL 모델**입니다. 이 모델은 Hybrid2(CF+MF)와 Hybrid3(MF+DL)을 혼합한 Hybrid(CF+MF+DL) 모델입니다.

```
NEW_MF(ratings_temp, K=200, alpha=0.0014, beta=0.075, iterations=350, tolerance=0.0000001, verbose=True)
```

MF에서 tolerance를 1e-7로 감소시켰고,

```
# ubcf_bias_knn 최적의 k 값 찾기
a = 1.5

# RMSE값이 1.5보다 작은 것만 비교
for K in range(15, 45, 3):
    result1 = recommender1(recomm_list, K)
    best_score = RMSE2(ratings_test['rating'], result1)
    if a > best_score:
        a = best_score
        best_knn = K
    else:
        pass
```

ubcf_bias_knn에서 k 값을 15~45사이에서 3 step 씩 변화시키며 최적의 k값을 찾아 적용시키도록 코드를 작성하였습니다.

```
## 3중 for문을 활용하여 최적의 weight의 비율을 구함
# MF
box_w = [0.75, 0.76, 0.77, 0.78, 0.79, 0.80, 0.81, 0.82, 0.83, 0.84, 0.85, 0.86, 0.87, 0.88, 0.89, 0.90, 0.91, 0.92, 0.93, 0.94, 0.95, 0.96, 0.97, 0.98, 0.99, 1.0]
# CF
box_x = [0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.10, 0.11, 0.12, 0.13, 0.14, 0.15]
# DL
box_y = [0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.10, 0.11, 0.12]

#그냥 큰 수치 임의로 지정
best_RMSE = 10
for x in box_w: # x는 MF
    for y in box_x: # y는 CF
        for z in box_y: # z는 DL
            if x+y+z == 1:
                try:
                    weight = [x,y,z] #최적의 weight의 비율
                    predictions = []
                    for i, number in enumerate(result0):
                        predictions.append(result0[i] * weight[0] + result1[i] * weight[1]+ result2[i] * weight[2])
                    # RMSE구하기
                    rmse2 = RMSE2(ratings_test['rating'], predictions)
                    if best_RMSE > rmse2:                    # New best record
                        best_RMSE = rmse2
                        best_ratio = " 최고의 황금비율 \n MF : '%.2f' \n ubcf_bias_knn : '%.2f' \n DL : '%.2f' \n RMSE = %.7f" % (weight[0], weight[1], weight[2], RMSE2(ratings_test['rating'], predictions))
                except:
                    pass
```

이 후 각 모델에 대한 가중치를 0.00부터 1.00까지 조절해가며 가장 높은 성능을 보이는 모델 별 가중치 비율을 찾았습니다. 일반적으로 MF의 경우 0.8~0.9, CF와 DL은 0.10 미만의 범위일때 가장 높은 성능을 나타냈고, 해당 탐색시간을 줄여 모델을 경량화시키기 위해 MF의 범위를 0.75~1.00, CF의 범위를 0.00~0.15, DL의 범위를 0.00~0.12로 한정시켰습니다.

```
최고의 황금비율
MF : '0.89'
ubcf_bias_knn : '0.09'
DL : '0.02'
RMSE = 0.9187310
```

(MF+CF+DL 모형에서 마지막에 나온 값)

이러한 조건 아래에서 확인한 3회 평균 RMSE 값은 평균 0.9198로 가장 좋은 성능을 보였습니다.

6. Lessons learned: 이 프로젝트로 배운점

첫번째로 아쉬운 점은 Reviews 데이터를 사용하지 못한 것입니다. Reviews 데이터를 자칫 잘못 사용하게 되면 Cheating의 위험이 있다고 판단하여, Reviews 데이터를 사용하지 않았습니다. 하지만 프로젝트가 끝난 지금, Reviews 데이터를 아예 사용하지 않는 것이 아니라, 전처리와 다른 시도를 통해 모델에 적용해 봤으면 하는 아쉬움이 남습니다. 만약 Reviews 데이터를 사용한다면, 감성분석을 이용하여 FM에 적용해 보고 싶습니다.

두번째로 아쉬운 점은 새로운 추천시스템 모델을 적용해 보지 못한 것입니다. Embedding을 새로운 시도로 만들었음에도 불구하고, Cf_Knn 모델에 적용한 것이 아쉬웠습니다. 또한, 수업시간에서 다루지 않은 모델을 시도하지 않은 것에 대한 아쉬움이 큽니다. 파생변수를 추가하여도 RMSE 값이 쉽게 낮아지지 않았기 때문에, 처음 목표를 “모델의 성능을 올리기 위해 새로운 조합”을 찾는 것이 아니라 “RMSE이 값을 낮추기 위해 새로운 알고리즘 찾기 및 만들기”로 정했다면 어땠을까 라는 아쉬움이 큽니다.

마지막으로 user embedding을 시도했을 때, LSTM seqeunce의 최적값을 찾지 못한 것이 아쉽습니다. 임베딩에서 제외되는 user를 최소화 하기 위해 ‘4’로 지정하였습니다. 이때, train set 기준으로 3706명의 user들의 분포를 보았을 때, 최소값이 30이였습니다. 때문에 그보다 하나 높은 1로 했지만, 몇몇을 임베딩에서 제외시키더라도, 더

높은 값으로 LSTM seqeunce의 수를 지정해서 비교했으면 이라는 아쉬움이 있습니다.

7. Reference

HMSG : Heterogeneous Graph Neural network based on Metapath Subgraph Learning

metapath2vec: Scalable Representation Learning for Heterogeneous Networks