2020년 2학기 데이터관리와 분석

프로젝트#2: DB mining & Automated Recommendation System

12조
2017-13918 황재훈
2017-10854 최의현
2017-18868 조형찬
2017-15751 장준혁
2017-10089 윤성진

1 . Part1

R1-1

```
cursor.execute('''DROP TABLE IF EXISTS app_prize_list;''')
#cursor.execute('''ALTER Table app DROP app_prize;''') #재실행시에는 이 코드를 같이 돌릴 필요가 있음
cursor.execute('''ALTER TABLE app ADD (app_prize TINYINT(1) default 0);''')
cursor.execute('''
    CREATE TABLE if not exists app_prize_list(
    id varchar(255) not null,
    primary key(id));''')

f_apl = open('C:/Users/User/Downloads/DMA_project2/DMA_project2/app_prize_list.txt', 'r', encoding='utf-8')
d_apl = f_apl.readlines()
for i in range(0, len(d_apl)):
    li_apl = d_apl[i].replace('\n', '')
    li_apl = li_apl.split(', ')
    cursor.execute('''
                    INSERT INTO app_prize_list VALUES (%s)''',
                   (li_apl))
cnx.commit()
f_apl.close()

cursor.execute('''
    update app
    set app_prize=1
    where app.id in
    (select app_prize_list.id
    from app_prize_list);
    ''')
cursor.execute('''DROP TABLE IF EXISTS app_prize_list;''')
cnx.commit()
print('1-1 clear')
```

app table에 빈 app_prize column을 생성하고 app_prize_list table을 새로 생성하였다. 이때,

app_prize_list table의 primary key를 id로 설정하였다. 그 후, app_prize_list.txt 파일의 데이터를 파이썬으로 불러와서 한 줄씩 읽은 후, 새로 만든 app_prize_list table에 넣어주었다. 이후, app_prize_list table의 id가 app table의 id에 존재할 때, app table의 app_prize column에 1을 넣도록 코딩하였다. (duplicate error를 방지하기 위해 1-1 코드를 반복 실행하는 경우, #cursor.execute('''ALTER table app DROP app_prize;''')를 실행시켜준다.))

R1-2

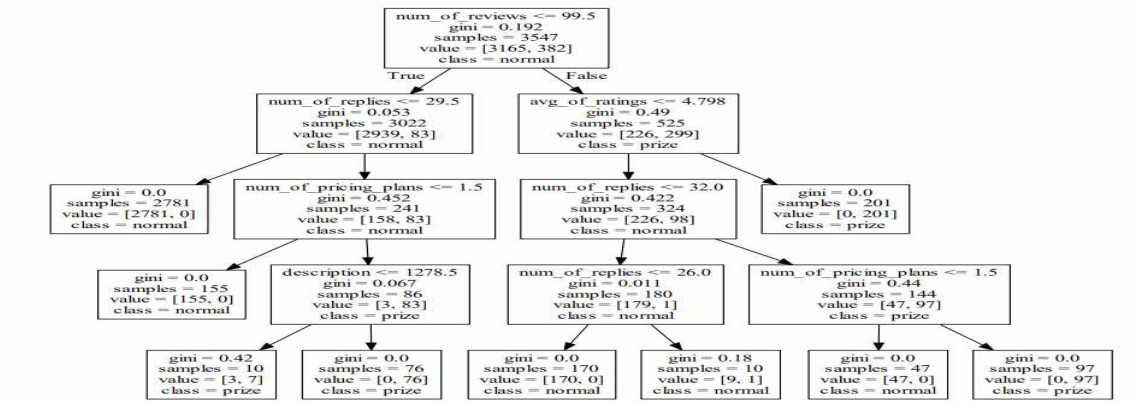
```
# TODO: Requirement 1-2. WRITE MYSQL QUERY AND EXECUTE. SAVE to .csv file
col_names = ['id', 'app_prize', 'description', 'have_pricing_hint', 'num_of_categories', 'num_of_pricing_plans',
             'num_of_reviews', 'avg_of_ratings', 'num_of_replies']
fopen = open('DMA_project2_team%02d_part1.csv' % team, 'w', encoding='utf-8')
for col in col_names:
    fopen.write(col)
    if col == 'num_of_replies':
        fopen.write('\n')
    else:
        fopen.write(',')
nested_query = '''
    select app.id as id,app.app_prize as app_prize, app.description as description, if(app.pricing_hint is null, 0,
    1) as have_pricing_hint,
    (select count(*) from app_category where app_category.app_id = app.id group by app_id) as num_of_categories,
    (select count(*) from pricing_plan where pricing_plan.app_id = app.id group by app_id) as num_of_pricing_plans,
    if((select count(*) from review where app_id = app.id group by app_id) is null, 0, (select count(*) from review
    where app_id = app.id group by app_id)) as num_of_reviews,
    if((select avg(rating) from review where app_id = app.id group by app_id) is null, 0,(select avg(rating) from
     review where app_id = app.id group by app_id) ) as avg_of_ratings,
    if((select count(*) from reply where app.developer_id=reply.developer_id  group by developer_id) is null, 0,
    (select count(*) from reply where app.developer_id=reply.developer_id  group by developer_id)) as num_of_replies
    from app'''
cursor.execute(nested_query)
rows = cursor.fetchall()
for row in rows:
    for i in range(0, 9):
        if i == 8:
            fopen.write(str(row[i]))
            fopen.write('\n')
        else:
            fopen.write(str(row[i]))
            fopen.write(',')
fopen.close()
print('1-2 clear')
```

DMA_project2_team12_part1.csv 파일을 파이썬으로 열고, 조건을 만족하는, column name을 먼저 입력하였다. 그 후, 각 column의 조건을 만족시키는 data를 넣기 위하여 하나의 Nested query를 실행하였다. 하나의 Nested query는 ALIAS 구문을 이용한, SELECT AS를 이용하여 각각의 조건에 맞추어 csv 파일에 입력한 column 제목과 일치하도록 별칭 하였다. 이때, count와 group by를 사용하여 num_of_categories, num_of_pricing_plans, num_of_review, avg_or_ratings, num_of_replies을 입력해 주었고 그 결과를 csv 파일에 저장하였다.

R1-3

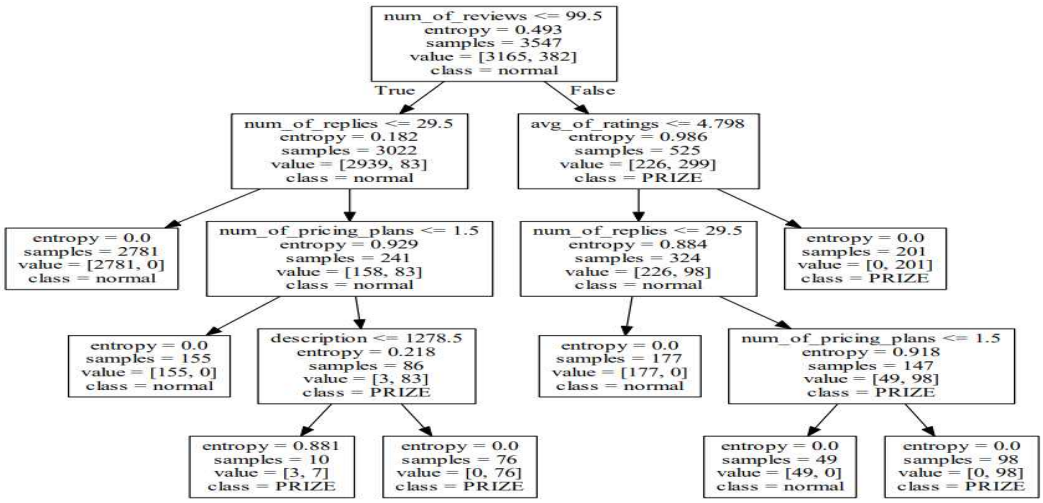
```
# TODO: Requirement 1-3. MAKE AND SAVE DECISION TREE
# gini file name: DMA_project2_team##_part1_gini.pdf
# entropy file name: DMA_project2_team##_part1_entropy.pdf
# preprocessing
app_info_dt = pd.read_csv('./DMA_project2_team%02d_part1.csv' %team)
app_prize_dt = app_info_dt['app_prize']
cols_for_train = ['description', 'have_pricing_hint', 'num_of_categories', 'num_of_pricing_plans', 'num_of_reviews',
                  'avg_of_ratings', 'num_of_replies']
app_info_dt = app_info_dt[cols_for_train]
print('preprocessing is done')
# gini DT
DT_gini = tree.DecisionTreeClassifier(criterion='gini', max_depth=5, min_samples_leaf=10, random_state=0)
DT_gini.fit(X=app_info_dt, y=app_prize_dt)
graph = tree.export_graphviz(DT_gini, out_file=None, feature_names=['description', 'have_pricing_hint',
                                                                    'num_of_categories', 'num_of_pricing_plans',
                                                                    'num_of_reviews', 'avg_of_ratings',
                                                                    'num_of_replies'], class_names=['normal', 'PRIZE'])
graph = graphviz.Source(graph)
graph.render('DMA_project2_team12_part1_gini', view=True)
print('create DT with gini')

# entropy DT
DT_entropy = tree.DecisionTreeClassifier(criterion='entropy', max_depth=5, min_samples_leaf=10, random_state=0)
DT_entropy.fit(X=app_info_dt, y=app_prize_dt)
graph = tree.export_graphviz(DT_entropy, out_file=None, feature_names=['description', 'have_pricing_hint',
                                                                       'num_of_categories', 'num_of_pricing_plans',
                                                                       'num_of_reviews', 'avg_of_ratings',
                                                                       'num_of_replies'], class_names=['normal', 'PRIZE'])
graph = graphviz.Source(graph)
graph.render('DMA_project2_team12_part1_entropy', view=True)
print('create DT with entropy')

print('1-3 clear')
```

1-2에서 반환받은 결과를 이용하여 의사결정 나무를 생성하기 위해 pandas로 csv파일을 불러 왔다. 이후 gini와 entropy 기준을 이용하여 decision tree를 만들기 위하여 주어진 PRIZE 수여기준에 맞는 조건들을 입력해 주었다. 이때 훈련 및 테스트 데이터 세트를 동일하게 만들기 위하여 random_state=0으로 만들어주었고 X 는 app_prize를 제외한 1-2에서 반환한 데이터, Y → app_prize에 해당하는 데이터를 대입하고, graphviz를 이용하여 decision tree를 그려주었다.



우선, 위의 gini 분석을 보자. max_depth가 5이나, depth가 4까지만, 분석된 것을 볼 수 있다. 모두 gini가 0.0 값을 가지거나, min_samples_leaf를 10으로 설정하였기 때문에, 4번째

에서 종료된 것을 볼 수 있다. 의사 결정 나무가 pure한 것을 볼 수 있었으며, 이로 인해 의사결정 나무가 적절한 것을 볼 수 있었다. 또한 num_of_review를 기준으로 99.5로 나누었을 때, 상을 받은 것과 그렇지 않은 게 상당히 잘 구분되는 것으로 보아 num_of_review가 상을 받는데 관여하는 중요한 요소인 것을 알 수 있었다.



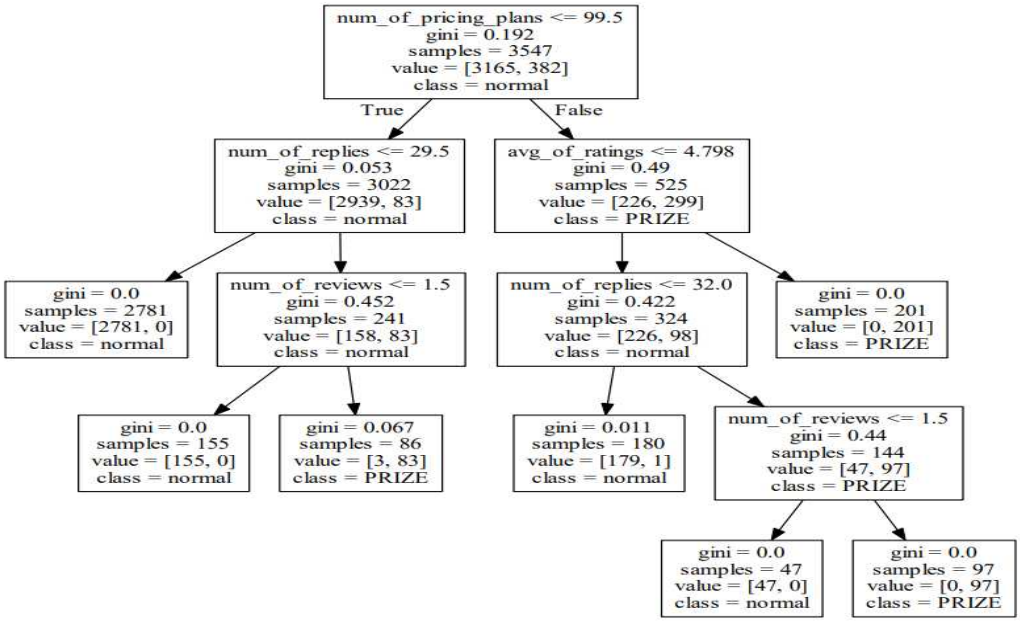
위와 마찬가지로 max_depth=5였으나, min_samples_leaf를 10으로 설정해놓았기 때문에, depth가 4일 때, entropy값이 0.0이거나, sample 개수가 10이었기 때문에, depth가 4일 때 분석이 종료된 현상을 관찰할 수 있었다. entropy값이 0.0이 많이 나온 것으로 보아 적절한 의사 결정나무를 만든 것을 알 수 있었다. 또한, gini로 나눴을 때랑 마찬가지로, num_of_reviews 기준으로 나누었을 때, 상을 받은 것과 그렇지 않은 게 잘 구분되는 것을 볼 수 있었고, 중요한 요소임을 확인할 수 있었다.

R1-4

```
app_info_dt=pd.read_csv('C:/Users/User/Desktop/DMA_project2_team12_part1.csv')
app_prize_dt=app_info_dt['app_prize']
cols_for_train=['description','num_of_categories','num_of_reviews','num_of_pricing_plans','avg_of_ratings','num_of_replies']
app_info_dt=app_info_dt[cols_for_train]
DT_revised=tree.DecisionTreeClassifier(criterion='gini', max_depth=4, min_samples_leaf=5, random_state=1, min_impurity_decrease=0.002)
DT_revised.fit(X=app_info_dt, y=app_prize_dt)
graph=tree.export_graphviz(DT_revised, out_file=None, feature_names=['description','num_of_categories','num_of_pricing_plans',
                                                                     'num_of_reviews','avg_of_ratings','num_of_replies'],class_names=['normal','PRIZE'])
```

R1-4 코딩은 위와 같이 하였. 고려한 조건은 아래에 기술하였다.

생성한 의사결정 나무는 아래와 같다.



1-3에서 생성한 의사결정 나무와 다른 PRIZE 수여 기준에 대한 의사결정나무를 만들기 위해 아래와 같은 조건들을 이용해 생성하였다.

사용된 input features:
'description','num_of_categories','num_of_reviews','num_of_pricing_plans','avg_of_ratings','num_of_replies'
# Node Impurity criterion: gini
# sklearn.tree.DecisionTreeClassifier에 입력한 속성 :
max_depth=4, min_samples_leaf=5, random_state=0, min_impurity_decrease=0.002

위와 같은 조건을 설정한 이유는 다음과 같다.

have_pricing_hint는 기존의 DT에서 중요한 기준이 되지 못하는 것 같아 input feature에서 삭제하였다.
기존의 gini모델이 좀 더 개선의 여지가 보여서 gini 모델을 채택하였다.
max_depth가 4를 넘어가면 overfitting되는 경향이 있는 것 같아 4로 설정하였고 min_samples_leaf가 적을 수록 상세하게 분류가 가능하므로 5로 낮추었고 overfitting을 방지하기 위해 min_impurity_decrease를 0.002로 미세하게 조정하였다.

변경한 gini DT 기존의 gini DT를 비교해 보았을 때, impurity가 더 낮고 tree도 간단해져 판단하기가 더 쉬워짐을 확인할 수 있었다. 또한 기존의 entropy DT의 마지막 layer 중 왼쪽에 있는 결과 값은 오히려 impurity가 높아지는 결과를 도출했었는데, 개선된 Tree에선 이 부분을 굳이 더 partitioning을 진행하지 않으며 낮은 impurity를 유지하는 것을 확인하였다.

2. Part2

R2-1

```
cursor.execute('''create or replace view category_score as
    with A as (select A.id, count(*) as nd from category A join category_developer B on A.id = B.category_id group by A.id),
    B as (select A.id, count(*) as nu from category A join category_user B on A.id = B.category_id group by A.id),
    C as (select A.id, count(*) as na from category A join app_category B on A.id = B.category_id group by A.id),
    D as (select D.id, D.title , A.nd, B.nu, C.na from ((category D left join A on D.id = A.id) left join B on D.id
    = B.id) left join C on D.id = C.id)
    select id as category_id, title as category_title, if(nd is null, 0, nd) as num_developer, if(nu is null, 0, nu)
    as num_user, na as num_app, if(nd is null, 0, nd)+if(nu is null, 0, nu)+na as score from D order by score desc limit 30
        ''')
fopen = open('DMA_project2_team%02d_part2_category.csv' % team, 'w', encoding='utf-8', newline='')
col_category=['category_id','category_title','num_developer','num_user','num_app','score']
for col in col_category:
    fopen.write(col)
    if col == 'score':
        fopen.write('\n')
    else:
        fopen.write(',')
writer = csv.writer(fopen)
cursor.execute('select * from category_score')
a = cursor.fetchall()
for i in a:
    writer.writerow(i)
fopen.close()
print('2-1 clear')
```

주어진 조건을 만족하는 category_score라는 view를 하나의 SQL문장으로 만들기 위해, with 구문을 이용하여 여러 가지 임시테이블을 만들었다. 예를 들어, Num_developer라는 컬럼을 추가하기 위해, 카테고리 테이블과 카테고리 디벨로퍼 테이블을 조인하고, 아이디 별로 그룹화하여, 카테고리 아이디와 해당 카테고리를 좋아하는 디벨로퍼 수를 칼럼으로 갖는 임시테이블을 만들었다. 이와 같은 방법으로 임시테이블을 세 개 만들고 난 후, 카테고리 아이디, 카테고리 타이틀을 컬럼으로 갖고 세 개의 임시테이블에 아이디를 조회한 값을 넣을 수 있게 Left join하였다. 이후 score 상위 30개만을 내림차순으로 자른후 view에 저장하였다. 이후 저장된 view는 csv모듈을 활용하여 csv 파일에 기록하였다.

R2-2

```
# TODO: Requirement 2-2. CREATE 2 VIEWS AND SAVE partial one to .csv file
cursor.execute(
    '''create or replace view user_item_rating as with real_app_category as  (select app_id, app_category.category_id
    from app_category, category_score where app_category.category_id = category_score.category_id), user_category as
    ((select review.user_id, real_app_category.category_id from review ,real_app_category where review.app_id =
    real_app_category.app_id) union all (select app.developer_id, real_app_category.category_id from app,
    real_app_category where app.id = real_app_category.app_id)), user_category_score as (select user_id, category_id,
    count(*) as score from user_category group by user_id, category_id), user_category_like as
    ((select category_user.user_id, category_user.category_id from category_user, category_score where category_user
    .category_id = category_score.category_id) union all (select category_developer.developer_id, category_developer.
    category_id from category_developer, category_score where category_developer.category_id = category_score.category_id))
    , user_and_developer as (select id from user union all select id from developer), user_category_likeit as
    (select user_id, category_id, count(*) as likeit from user_category_like group by user_id, category_id)
    select A.id, E.title as category, if(C.likeit is not null,5,0) + if(D.score is not null,if(D.score>5,5,D.score),0)
     as rating from (((user_and_developer A join category_score B) left join user_category_likeit C on A.id =
     C.user_id and B.category_id = C.category_id) left join user_category_score D on A.id = D.user_id and B.category_id =
     D.category_id), category E where (C.likeit is not null or D.score is not null) and B.category_id = E.id''')
cursor.execute(
    '''create or replace view partial_user_item_rating as with id_count as (select id, count(*) as counta
    from user_item_rating group by id) select A.id, A.category, A.rating from user_item_rating A join id_count B where
     A.id = B.id and B.counta >= 12''')

fopen = open('DMA_project2_team%02d_part2_UIR.csv' % team, 'w', encoding='utf-8', newline='')
col_UIR=['user','category','rating']
for col in col_UIR:
    fopen.write(col)
    if col == 'rating':
        fopen.write('\n')
    else:
        fopen.write(',')
writer2 = csv.writer(fopen)
cursor.execute('''SELECT * FROM partial_user_item_rating''')
b = cursor.fetchall()
for i in b:
    writer2.writerow(i)
fopen.close()
print('2-2 clear')
```

우선, user_item_rating이라는 view를 하나의 SQL문장으로 만들기 위해서 2-1과 같이 여러 임시테이블을 만들었다. 예를 들어, 생성한 user_category_likeit 임시테이블은 유저, 카테고리, 해당 유저가 카테고리를 like하는지 여부가 0 or 1의 값을 갖는 세 개의 칼럼으로 구성되어 있다. 필요한 임시테이블을 모두 만든 후, 유저와 카테고리를 조건 없이 join한 후 유저아이디와 카테고리 아이디의 조인 조건을 가지고 임시테이블을 차례로 조인하여 view를 생성하였다. 이어서 partial view를 만들기 위하여 위의 view를 아이디로 그룹화하여 개수를 담는 임시테이블을 만들고 이를 '해당 임시테이블의 카운트 수가 12를 넘어야 한다'를 join 조건으로 활용하여 view를 완성했다. 완성한 뷰는 csv모듈을 사용하여 csv 파일에 기록하였다.

R2-3

```
# TODO: Requirement 2-3. MAKE HORIZONTAL VIEW
# file name: DMA_project2_team##_part2_horizontal.pkl
df = pd.read_csv('./DMA_project2_team12_part2_UIR.csv', names=['id', 'category', 'rating'], header=0)
df1 = df.pivot_table(index='id', columns='category', values='rating')
for i in range(2, 11):
    df1 = df1.replace(i, 1)
df1 = df1.fillna(0)
df1.to_pickle('DMA_project2_team12_part2_horizontal.pkl')
```

horizontal table을 만들기 위해 pandas 라이브러리를 활용하였다. 2-2에서 생성한 파일을 읽고 이를 horizontal table로 만드는 전처리 과정은 다음과 같다. 우선, 해당 데이터 프레임을 pivot_table 기능을 사용하여 horizontal화 한 후 1 이외의 값을 갖는 2~10까지의 값을 for문을 통한 replace 함수를 이용하여 모두 1로 바꿔주고, fillna(0)을 이용하여 nan 값을 0으로 치환하였다. 이처럼 완성한 horizontal table을 pkl 파일에 저장하였다..

R2-4

```
# TODO: Requirement 2-4. ASSOCIATION ANALYSIS
# filename: DMA_project2_team##_part2_association.pkl (pandas dataframe)
frequent_itemsets = apriori(df1, min_support=0.05, use_colnames=True)
rules = association_rules(frequent_itemsets, metric='lift', min_threshold=2)
rules.to_pickle('DMA_project2_team12_part2_association.pkl')
cursor.close()

print('2-3 and 2-4 clear')
```

주어진 조건을 만족하는 frequent itemset을 구성하기 위해 min_support를 0.05로 두고 apriori 알고리즘을 적용하였다. 이 frequent itemset을 바탕으로 rule을 구성하는데 조건대로 lift의 최소 값을 2로 설정하고 룰을 생성했다. 이처럼 생성한 룰을 pkl 파일로 저장하였다.

	anteceden	consequents	antecedent support	consequent support	support	confidence	lift
13066	frozenset({	frozenset({'Find	0.05589207	0.314977974	0.055892	1	3.174825
170104	frozenset({	frozenset({'Mar	0.073513216	0.308645374	0.073513	1	3.239964
169936	frozenset({	frozenset({'Find	0.063601322	0.314977974	0.063601	1	3.174825
169944	frozenset({	frozenset({'Stor	0.063601322	0.314427313	0.063601	1	3.180385
169948	frozenset({	frozenset({'Mar	0.063601322	0.309196035	0.063601	1	3.234194
169976	frozenset({	frozenset({'Mar	0.063601322	0.308645374	0.063601	1	3.239964
170064	frozenset({	frozenset({'Find	0.073513216	0.314977974	0.073513	1	3.174825
8366	frozenset({	frozenset({'Find	0.080947137	0.314977974	0.080947	1	3.174825
170072	frozenset({	frozenset({'Stor	0.073513216	0.314427313	0.073513	1	3.180385
8370	frozenset({	frozenset({'Stor	0.080947137	0.314427313	0.080947	1	3.180385

	antecedents	consequer	antecedent support	consequent support	support	confidence	lift
1368	frozenset({'Pr	frozenset({	0.107654185	0.089207048	0.052588	0.488491049	5.475924
1367	frozenset({'Fir	frozenset({	0.089207048	0.107654185	0.052588	0.589506173	5.475924
11660	frozenset({'Fir	frozenset({	0.088656388	0.107654185	0.052037	0.586956522	5.452241
11663	frozenset({'Pr	frozenset({	0.107654185	0.088656388	0.052037	0.483375959	5.452241
11656	frozenset({'St	frozenset({	0.107103524	0.089207048	0.052037	0.485861183	5.446444
11667	frozenset({'Fir	frozenset({	0.089207048	0.107103524	0.052037	0.583333333	5.446444
1369	frozenset({'Fir	frozenset({	0.122797357	0.080121145	0.052588	0.428251121	5.345045
1366	frozenset({'In	frozenset({	0.080121145	0.122797357	0.052588	0.656357388	5.345045
11664	frozenset({'Fir	frozenset({	0.122797357	0.079570485	0.052037	0.423766816	5.325678
11659	frozenset({'St	frozenset({	0.079570485	0.122797357	0.052037	0.653979239	5.325678

confidence기준과 lift기준으로 상위 10개의 결과물을 넣었다. lift는 confidence의 값에서 consequent support값을 나누어 준 것이다. 여기서 lift의 중요성을 알 수 있다. 그러므로 단순히 confidence를 고려하는 것보다는 더욱더 유의미한 상관관계를 얻을 수 있게 된다. lift가 높으면 양의 상관관계가 높다고 생각할 수 있는 지표이다. 예를 들어, lift 값이 가장 큰 1368번을 예를 들어보면, ('Productivity', 'Dropshipping')와 ('Finding and adding products', 'Inverntory management')의 양의 상관관계가 크기 때문에 lift값이 크게 나온다.

3. Part3

R3-1

```python
# TODO: Requirement 3-1. WRITE get_top_n
def get_top_n(algo, testset, id_list, n=10, user_based=True):
    results = defaultdict(list)
    if user_based:
        # TODO: testset의 데이터 중에 user id가 id_list 안에 있는 데이터만 따로 testset_id로 저장
        # Hint: testset은 (user_id, item_id, default_rating)의 tuple을 요소로 갖는 list
        testset_id = [t for t in testset if (t[0] in id_list)]
        predictions = algo.test(testset_id)
        for uid, iid, true_r, est, _ in predictions:
            # TODO: results는 user_id를 key로,  [(item_id, estimated_rating)의 tuple이 모인 list]를 value로 갖는 dictionary
            results[uid].append((iid, est))
            pass
    else:
        # TODO: testset의 데이터 중 item id가 id_list 안에 있는 데이터만 따로 testset_id라는 list로 저장
        # Hint: testset은 (user_id, item_id, default_rating)의 tuple을 요소로 갖는 list
        testset_id = [t for t in testset if (t[1] in id_list)]
        predictions = algo.test(testset_id)
        for uid, iid, true_r, est, _ in predictions:
            # TODO: results는 item_id를 key로, [(user_id, estimated_rating)의 tuple이 모인 list]를 value로 갖는 dictionary
            results[iid].append((uid, est))
            pass
    for id_, ratings in results.items():
        # TODO: rating 순서대로 정렬하고 top-n개만 유지
        temp_rating=sorted(ratings, key=lambda x:x[1], reverse=True)
        results[id_]=temp_rating[:n]
        pass

    return results
```

for문을 이용하여, testset 데이터 중 user id가 id_list 안에 있는 데이터에 대해서 testset_id를 저장했다. 이후, append 함수를 사용하여, user_id를 key로, item_id, estimated_rating을 value로 모아 dictionary를 만들었다. 그리고 testset_id를 testset의 데이터 중, item id가 id_list에 존재하는 경우 list로 저장하였다. 앞에서와 마찬가지로, item_id를 key로, user_id와 estimated_rating을 value로 갖는 dictionary를 만들었고, 맨 밑의 for문에서 sort와 reverse = True를 사용하여, rating을 내림차순으로 정렬하고 상위 n개만 유지하도록 하였다.

R3-2

3-2-1

```python
# TODO: set algorithm for 3-2-1
sim_options = {'name': 'cosine', 'user_based': True}
user_knnbasic = surprise.KNNBasic(sim_options=sim_options)
user_knnbasic.fit(trainset)

results = get_top_n(user_knnbasic, testset, uid_list, n=10, user_based=True)

with open('3-2-1.txt', 'w') as f:
    for uid, ratings in sorted(results.items(), key=lambda x: x[0]):
        f.write('User ID %s top-10 results\n' % uid)
        for iid, score in ratings:
            f.write('Item ID %s\tscore %s\n' % (iid, str(score)))
        f.write('\n')
print('3-2-1.txt completed')
```

3-2-1에서는 cosine 유사도를 사용하기 위하여 sim_options에서 알맞은 parameter를 넣어준 후 surprise 라이브러리의 KNNBasic 알고리즘을 사용하여 user_knnbasic을 algo로 설정한 후 fit 해주었고 그 결과를 3-2-1.txt로 출력하였다.

3-2-2

```python
# TODO: set algorithm for 3-2-2
sim_options1 = {'name': 'pearson', 'user_based': True}
user_knnwithmeans = surprise.KNNWithMeans(sim_options=sim_options1)
user_knnwithmeans.fit(trainset)

results = get_top_n(user_knnwithmeans, testset, uid_list, n=10, user_based=True)

with open('3-2-2.txt', 'w') as f:
    for uid, ratings in sorted(results.items(), key=lambda x: x[0]):
        f.write('User ID %s top-10 results\n' % uid)
        for iid, score in ratings:
            f.write('Item ID %s\tscore %s\n' % (iid, str(score)))
        f.write('\n')
print('3-2-2.txt completed')
```

3-2-2에서는 pearson 유사도를 사용하기 위하여 sim_options1에 알맞은 parameter를 넣어준 후 surprise 라이브러리의 KNNWithMeans 알고리즘을 사용하여 user_knnwithmeans를 algo로 설정한 후 fit을 해주었고 그 결과를 3-2-2.txt로 출력하였다.

3-2-3

```python
# TODO: 3-2-3. Best Model
candidate_list_ub = []
ub_sim_options_list = [{'name': 'cosine', 'user_based': True}, {'name': 'msd', 'user_based': True},
                       {'name': 'pearson', 'user_based': True}]
for i in ub_sim_options_list:
    algo_list = [surprise.KNNBasic(sim_options=i), surprise.KNNWithMeans(sim_options=i),
                 surprise.KNNWithZScore(sim_options=i), surprise.KNNBaseline(sim_options=i)]
    for algo in algo_list:
        start = time.time()
        cv = surprise.model_selection.cross_validate(algo, data, measures=['RMSE', 'MAE'], cv=5, verbose=False)
        cv_time = str(datetime.timedelta(seconds=int(time.time() - start)))
        mean_rmse = '{:.3f}'.format(np.mean(cv['test_rmse']))
        mean_mae = '{:.3f}'.format(np.mean(cv['test_mae']))
        if algo == algo_list[0]:
            which_algo = 'KNNBasic'
        elif algo == algo_list[1]:
            which_algo = 'KNNWithMeans'
        elif algo == algo_list[2]:
            which_algo = 'KNNWithZScore'
        else:
            which_algo = 'KNNBaseline'
        candidate = [which_algo, i, mean_rmse, mean_mae, cv_time]
        print(candidate)
        candidate_list_ub.append(candidate)

print('minimize mean_rmse =', min(candidate_list_ub, key=lambda x: x[2]))
print('minimize mean_mae = ', min(candidate_list_ub, key=lambda x: x[3]))
print('minimize cv_time = ', min(candidate_list_ub, key=lambda x: x[4]))
```

3-2-3 에서는 위에서 사용했던 알고리즘 뿐만 아니라, KNNWithZScore, KNNBaseline 알고리즘과 cosine, msd, pearson의 유사도 함수를 이용하여 가장 성능이 좋은 것을 찾아 보았다. 주어진 조건인 cross validation(k=5, random_state=0)을 기준으로, rmse, mae, time 세 가지 경우에서 min 값을 이용하여 가장 좋은 모델을 찾았 보았다. (verbose=False로 설정하고 세가지 조건에 대해서 출력해주었다.)

3-2-3의 전체 결과는 아래와 같다.

```
['KNNBasic', {'name': 'cosine', 'user_based': True}, '0.907', '0.606', '0:01:47']
['KNNWithMeans', {'name': 'cosine', 'user_based': True}, '0.864', '0.628', '0:01:51']
['KNNWithZScore', {'name': 'cosine', 'user_based': True}, '0.866', '0.621', '0:01:55']
['KNNBaseline', {'name': 'cosine', 'user_based': True}, '0.848', '0.584', '0:01:54']
['KNNBasic', {'name': 'msd', 'user_based': True}, '0.846', '0.571', '0:01:10']
['KNNWithMeans', {'name': 'msd', 'user_based': True}, '0.834', '0.613', '0:01:15']
['KNNWithZScore', {'name': 'msd', 'user_based': True}, '0.852', '0.616', '0:01:16']
['KNNBaseline', {'name': 'msd', 'user_based': True}, '0.818', '0.563', '0:01:18']
['KNNBasic', {'name': 'pearson', 'user_based': True}, '0.912', '0.638', '0:02:08']
['KNNWithMeans', {'name': 'pearson', 'user_based': True}, '0.858', '0.610', '0:02:17']
['KNNWithZScore', {'name': 'pearson', 'user_based': True}, '0.850', '0.592', '0:02:13']
['KNNBaseline', {'name': 'pearson', 'user_based': True}, '0.867', '0.603', '0:02:14']
```

```
['KNNBasic', {'name': 'cosine', 'user_based': True}, '0.907', '0.606', '0:01:47']
['KNNWithMeans', {'name': 'cosine', 'user_based': True}, '0.864', '0.628', '0:01:51']
['KNNWithZScore', {'name': 'cosine', 'user_based': True}, '0.866', '0.621', '0:01:55']
['KNNBaseline', {'name': 'cosine', 'user_based': True}, '0.848', '0.584', '0:01:54']
['KNNBasic', {'name': 'msd', 'user_based': True}, '0.846', '0.571', '0:01:10']
['KNNWithMeans', {'name': 'msd', 'user_based': True}, '0.834', '0.613', '0:01:15']
['KNNWithZScore', {'name': 'msd', 'user_based': True}, '0.852', '0.616', '0:01:16']
['KNNBaseline', {'name': 'msd', 'user_based': True}, '0.818', '0.563', '0:01:18']
['KNNBasic', {'name': 'pearson', 'user_based': True}, '0.912', '0.638', '0:02:08']
['KNNWithMeans', {'name': 'pearson', 'user_based': True}, '0.858', '0.610', '0:02:17']
['KNNWithZScore', {'name': 'pearson', 'user_based': True}, '0.850', '0.592', '0:02:13']
['KNNBaseline', {'name': 'pearson', 'user_based': True}, '0.867', '0.603', '0:02:14']
```

이 중, min 값들을 모은 최종 결과는 다음과 같다.

```
minimize mean_rmse = ['KNNBaseline', {'name': 'msd', 'user_based': True}, '0.818', '0.563', '0:01:18']
minimize mean_mae = ['KNNBaseline', {'name': 'msd', 'user_based': True}, '0.818', '0.563', '0:01:18']
minimize cv_time = ['KNNBasic', {'name': 'msd', 'user_based': True}, '0.846', '0.571', '0:01:10']
```

따라서, 가장 좋은 성능을 보이는 모델 best_model_ub은 (RMSE 기준)
best_model_ub = surprise.KNNBaseline(sim_options = {'name: 'msd', 'user_based': True})
이다.

3-2-3에서 다양한 알고리즘 비교를 위해 고려한 사항은 다음과 같다.

RMSE / MAE / time 세가지 경우에 대해 살펴보았다.
k: (max)number of nearest neighbors
k가 너무 작으면 noise points에 민감해지고 k가 너무 크면 다른 class에 속한 points들을 포함할 가능성이 커지는 점을 고려했다. (ch.8 slide 50)
실제로 k를 증가시켜 (k=150 등) cross validation을 해보면, 같은 algorithm과 name 하에서 k만 변화시켰을 때 RMSE와 MAE에 큰 영향을 주지 못함을 확인할 수 있었다.
따라서, k=40 (default value) 로 특별히 변화시키지 않고 algorithm의 종류와 유사도(name)의 종류만 변화시켜 성능을 비교 및 평가하였다.
사용한 algorithm의 종류: KNNBasic, KNNWithMeans, KNNWithZScore, KNNBaseline
사용한 유사도(name)의 종류: cosine, msd, pearson
따라서, 총 12개 model에 대해 성능을 비교 및 평가하였다.

R3-3

3-3-1

```
# TODO - set algorithm for 3-3-1
sim_options2 = {'name': 'cosine', 'user_based': False}
item_knnbasic = surprise.KNNBasic(sim_options=sim_options2)
item_knnbasic.fit(trainset)

results = get_top_n(item_knnbasic, testset, iid_list, n=10, user_based=False)

with open('3-3-1.txt', 'w') as f:
    for iid, ratings in sorted(results.items(), key=lambda x: x[0]):
        f.write('Item ID %s top-10 results\n' % iid)
        for uid, score in ratings:
            f.write('User ID %s\tscore %s\n' % (uid, str(score)))
        f.write('\n')
print('3-3-1.txt completed')
```

3-3-1 에서는 iid_list을 만들어 category list를 만들었다. sim_options2에서 cosine 유사도 분석을 하기 위하여 그에 따른 알맞은 parameter를 넣어주었다. surprise의 KNNBasic 알고리즘을 사용하여, item_knnbasic을 algo로 설정하여 사용하여 fit 해주었고 그 결과를 3-3-1.txt로 출력하였다.

3-3-2

```
sim_options3 = {'name': 'pearson', 'user_based': False}
item_knnwithmeans = surprise.KNNWithMeans(sim_options=sim_options3)
item_knnwithmeans.fit(trainset)

results = get_top_n(item_knnwithmeans, testset, iid_list, n=10, user_based=False)

with open('3-3-2.txt', 'w') as f:
    for iid, ratings in sorted(results.items(), key=lambda x: x[0]):
        f.write('Item ID %s top-10 results\n' % iid)
        for uid, score in ratings:
            f.write('User ID %s\tscore %s\n' % (uid, str(score)))
        f.write('\n')
print('3-3-2.txt completed')
```

3-3-2에서도 마찬가지로 sim_options3에서, pearson 유사도 분석을 하기 위하여 그에 따라 알맞은 parameter를 넣어주었다. surprise의 KNNWithMeans 알고리즘을 사용하여, item_knnwithmeans를 algo로 설정하여 사용하여 fit 해주었고 그 결과를 3-3-2.txt로 출력하였다.

3-3-3

```python
# TODO: 3-3-3. Best Model
candidate_list_ib = []
ib_sim_options_list = [{'name': 'cosine', 'user_based': False}, {'name': 'msd', 'user_based': False},
                       {'name': 'pearson', 'user_based': False}]
for i in ib_sim_options_list:
    algo_list = [surprise.KNNBasic(sim_options=i), surprise.KNNWithMeans(sim_options=i),
                 surprise.KNNWithZScore(sim_options=i), surprise.KNNBaseline(sim_options=i)]
    for algo in algo_list:
        start = time.time()
        cv = surprise.model_selection.cross_validate(algo, data, measures=['RMSE', 'MAE'], cv=5, verbose=False)
        cv_time = str(datetime.timedelta(seconds=int(time.time() - start)))
        mean_rmse = '{:.3f}'.format(np.mean(cv['test_rmse']))
        mean_mae = '{:.3f}'.format(np.mean(cv['test_mae']))
        if algo == algo_list[0]:
            which_algo = 'KNNBasic'
        elif algo == algo_list[1]:
            which_algo = 'KNNWithMeans'
        elif algo == algo_list[2]:
            which_algo = 'KNNWithZScore'
        else:
            which_algo = 'KNNBaseline'
        candidate = [which_algo, i, mean_rmse, mean_mae, cv_time]
        print(candidate)
        candidate_list_ib.append(candidate)

print('minimize mean_rmse =', min(candidate_list_ib, key=lambda x: x[2]))
print('minimize mean_mae = ', min(candidate_list_ib, key=lambda x: x[3]))
print('minimize cv_time = ', min(candidate_list_ib, key=lambda x: x[4]))
```

3-3-3에서도 3-2-3과 마찬가지로 KNNWithMeans, KNNWithZScore, KNNBaseline, KNNBasic 알고리즘을 pearson과 cosine와 msd 유사도를 이용하여 RMSE, MAE, time 경우에서 가장 성능이 좋은 것을 찾아 보았다.

3-3-3의 전체 결과는 아래와 같다.

```
['KNNBasic', {'name': 'cosine', 'user_based': False}, '1.183', '0.877', '0:00:01']
['KNNWithMeans', {'name': 'cosine', 'user_based': False}, '0.818', '0.581', '0:00:02']
['KNNWithZScore', {'name': 'cosine', 'user_based': False}, '0.791', '0.551', '0:00:02']
['KNNBaseline', {'name': 'cosine', 'user_based': False}, '0.817', '0.579', '0:00:02']
['KNNBasic', {'name': 'msd', 'user_based': False}, '1.039', '0.716', '0:00:01']
['KNNWithMeans', {'name': 'msd', 'user_based': False}, '0.795', '0.558', '0:00:01']
['KNNWithZScore', {'name': 'msd', 'user_based': False}, '0.782', '0.542', '0:00:02']
['KNNBaseline', {'name': 'msd', 'user_based': False}, '0.792', '0.556', '0:00:03']
['KNNBasic', {'name': 'pearson', 'user_based': False}, '1.118', '0.842', '0:00:01']
['KNNWithMeans', {'name': 'pearson', 'user_based': False}, '0.798', '0.564', '0:00:02']
['KNNWithZScore', {'name': 'pearson', 'user_based': False}, '0.764', '0.533', '0:00:02']
['KNNBaseline', {'name': 'pearson', 'user_based': False}, '0.796', '0.562', '0:00:03']
```

```
['KNNBasic', {'name': 'cosine', 'user_based': False}, '1.183', '0.877', '0:00:01']
['KNNWithMeans', {'name': 'cosine', 'user_based': False}, '0.818', '0.581', '0:00:02']
['KNNWithZScore', {'name': 'cosine', 'user_based': False}, '0.791', '0.551', '0:00:02']
['KNNBaseline', {'name': 'cosine', 'user_based': False}, '0.817', '0.579', '0:00:02']
['KNNBasic', {'name': 'msd', 'user_based': False}, '1.039', '0.716', '0:00:01']
['KNNWithMeans', {'name': 'msd', 'user_based': False}, '0.795', '0.558', '0:00:01']
['KNNWithZScore', {'name': 'msd', 'user_based': False}, '0.782', '0.542', '0:00:02']
['KNNBaseline', {'name': 'msd', 'user_based': False}, '0.792', '0.556', '0:00:03']
['KNNBasic', {'name': 'pearson', 'user_based': False}, '1.118', '0.842', '0:00:01']
['KNNWithMeans', {'name': 'pearson', 'user_based': False}, '0.798', '0.564', '0:00:02']
['KNNWithZScore', {'name': 'pearson', 'user_based': False}, '0.764', '0.533', '0:00:02']
['KNNBaseline', {'name': 'pearson', 'user_based': False}, '0.796', '0.562', '0:00:03']
```

이 중, min 값들을 모은 최종 결과는 다음과 같다.

```
minimize mean_rmse = ['KNNWithZScore', {'name': 'pearson', 'user_based': False}, '0.764', '0.533', '0:00:02']
minimize mean_mae = ['KNNWithZScore', {'name': 'pearson', 'user_based': False}, '0.764', '0.533', '0:00:02']
minimize cv_time = ['KNNBasic', {'name': 'cosine', 'user_based': False}, '1.183', '0.877', '0:00:01']
```

따라서, 가장 좋은 성능을 보이는 모델 best_model_Ib는 (RMSE 기준)
best_model_Ib = surprise.KNNWithZScore(sim_options = {'name: 'pearson', 'user_based': False}) 이다.

3-3-3에서 다양한 알고리즘 비교를 위해 고려한 사항은 다음과 같다.

3-2-3과 같은 방식으로 성능 비교 및 평가하였다.
user_based 부분만 False로 변경하여 수행하였다.
3-2-3에서 similarity matrix들이 이미 계산되어 있으므로 이어서 3-3-3을 수행했을 때 cv_time은 매우 작게 나오게 되는 점을 유의하여야 한다.

R3-4

3-4-1 부터 3-4-4의 코드는 아래와 같다.

3-4-1

```
# TODO: set algorithm for 3-4-1
svd_50 = surprise.SVD(n_factors=100, n_epochs=50, biased=False)
svd_50.fit(trainset)
results = get_top_n(svd_50, testset, uid_list, n=10, user_based=True)

with open('3-4-1.txt', 'w') as f:
    for uid, ratings in sorted(results.items(), key=lambda x: x[0]):
        f.write('User ID %s top-10 results\n' % uid)
        for iid, score in ratings:
            f.write('Item ID %s\tscore %s\n' % (iid, str(score)))
        f.write('\n')
print('3-4-1.txt completed')
```

3-4-2

```
# TODO: set algorithm for 3-4-2
svd_100 = surprise.SVD(n_factors=200, n_epochs=100, biased=True)
svd_100.fit(trainset)
results = get_top_n(svd_100, testset, uid_list, n=10, user_based=True)

with open('3-4-2.txt', 'w') as f:
    for uid, ratings in sorted(results.items(), key=lambda x: x[0]):
        f.write('User ID %s top-10 results\n' % uid)
        for iid, score in ratings:
            f.write('Item ID %s\tscore %s\n' % (iid, str(score)))
        f.write('\n')
print('3-4-2.txt completed')
```

3-4-3

```
# TODO: set algorithm for 3-4-3
svdpp_50 = surprise.SVDpp(n_factors=100, n_epochs=50)
svdpp_50.fit(trainset)
results = get_top_n(svdpp_50, testset, uid_list, n=10, user_based=True)

with open('3-4-3.txt', 'w') as f:
    for uid, ratings in sorted(results.items(), key=lambda x: x[0]):
        f.write('User ID %s top-10 results\n' % uid)
        for iid, score in ratings:
            f.write('Item ID %s\tscore %s\n' % (iid, str(score)))
        f.write('\n')
print('3-4-3.txt completed')
```

3-4-4

```
# TODO: set algorithm for 3-4-4
svdpp_100 = surprise.SVDpp(n_factors=100, n_epochs=100)
svdpp_100.fit(trainset)
results = get_top_n(svdpp_100, testset, uid_list, n=10, user_based=True)

with open('3-4-4.txt', 'w') as f:
    for uid, ratings in sorted(results.items(), key=lambda x: x[0]):
        f.write('User ID %s top-10 results\n' % uid)
        for iid, score in ratings:
            f.write('Item ID %s\tscore %s\n' % (iid, str(score)))
        f.write('\n')
print('3-4-4.txt completed')
```

3-4-1 부터 3-4-4에서는 3명의 user에 대한 추천 결과를 텍스트 파일로 출력한다. 3-4-1부

터 3-4-4까지는 알고리즘만 달라서, 그 부분을 중점적으로 설명을 하도록 하겠다.
우선 3-4-1에서는, SVD(n_factors=100, n_epoch=50, biased=False) 알고리즘을 이용하였다. surprise 라이브러리의 SVD를 사용하여 해당하는 parameter들을 조건에 맞게 넣어주었다. 3-4-2에서는 3-4-1에서 해당하는 parameter 값들을 SVD(n_factors=200, n_epoch=100, biased=True)로 변경시켜 주었다.
3-4-3와 3-4-4에서는 SVD++알고리즘을 사용하였다. 위와 마찬가지로, surprise 라이브러리 SVD++를 사용하여. 3-4-3, 3-4-4 둘다 n_factors를 100으로, n_epochs는 각각 50,100으로 설정하였다. 이후 각 결과를 .txt 파일로 출력하였다.

3-4-5

마찬가지로 cross validation(k=5, randdom_state=0)을 기준으로 가장 좋은 성능을 보이는 모델을 찾기 위하여, 아래와 같이 실행하였다.

```
# TODO: 3-4-5. Best Model
candidate_list_mf = []
for factors in range(50, 250, 50):
    for epochs in range(10, 40, 10):
        SVD_biased = surprise.SVD(n_factors=factors, n_epochs=epochs, random_state=0, biased=True)
        SVD_unbiased = surprise.SVD(n_factors=factors, n_epochs=epochs, random_state=0, biased=False)
        SVDpp = surprise.SVDpp(n_factors=factors, n_epochs=epochs, random_state=0)
        NMF_biased = surprise.NMF(n_factors=factors, n_epochs=epochs, random_state=0, biased=True)
        NMF_unbiased = surprise.NMF(n_factors=factors, n_epochs=epochs, random_state=0, biased=False)
        algo_list = [SVD_biased, SVD_unbiased, SVDpp, NMF_biased, NMF_unbiased]
        for algo in algo_list:
            start = time.time()
            cv = surprise.model_selection.cross_validate(algo, data, measures=['RMSE', 'MAE'], cv=5, verbose=False)
            cv_time = str(datetime.timedelta(seconds=int(time.time() - start)))
            mean_rmse = '{:.3f}'.format(np.mean(cv['test_rmse']))
            mean_mae = '{:.3f}'.format(np.mean(cv['test_mae']))
            if algo == algo_list[0]:
                which_algo = 'SVD_biased'
            elif algo == algo_list[1]:
                which_algo = 'SVD_unbiased'
            elif algo == algo_list[2]:
                which_algo = 'SVDpp'
            elif algo == algo_list[3]:
                which_algo = 'NMF_biased'
            else:
                which_algo = 'NMF_unbiased'
            candidate = ['n_factors = %s' % factors, 'n_epochs = %s' % epochs, 'algorithm = %s' % which_algo,
                         mean_rmse, mean_mae, cv_time]
            print(candidate)
            candidate_list_mf.append(candidate)

print('minimize mean_rmse = ', min(candidate_list_mf, key=lambda x: x[3]))
print('minimize mean_mae = ', min(candidate_list_mf, key=lambda x: x[4]))
print('minimize cv_time = ', min(candidate_list_mf, key=lambda x: x[5]))
```

3-4-5의 전체 결과는 다음과 같다.

```
['n_factors = 50', 'n_epochs = 10', 'algorithm = SVD_biased', '0.804', '0.562', '0:00:04']
['n_factors = 50', 'n_epochs = 10', 'algorithm = SVD_unbiased', '0.780', '0.515', '0:00:04']
['n_factors = 50', 'n_epochs = 10', 'algorithm = SVDpp', '0.756', '0.513', '0:00:38']
['n_factors = 50', 'n_epochs = 10', 'algorithm = NMF_biased', '2.516', '1.801', '0:00:05']
['n_factors = 50', 'n_epochs = 10', 'algorithm = NMF_unbiased', '2.047', '1.698', '0:00:05']
['n_factors = 50', 'n_epochs = 20', 'algorithm = SVD_biased', '0.779', '0.539', '0:00:08']
['n_factors = 50', 'n_epochs = 20', 'algorithm = SVD_unbiased', '0.779', '0.514', '0:00:08']
['n_factors = 50', 'n_epochs = 20', 'algorithm = SVDpp', '0.752', '0:01:13']
['n_factors = 50', 'n_epochs = 20', 'algorithm = NMF_biased', '0.997', '0.647', '0:00:11']
['n_factors = 50', 'n_epochs = 20', 'algorithm = NMF_unbiased', '0.963', '0.737', '0:00:10']
['n_factors = 50', 'n_epochs = 30', 'algorithm = SVD_biased', '0.773', '0.535', '0:00:12']
['n_factors = 50', 'n_epochs = 30', 'algorithm = SVD_unbiased', '0.771', '0.510', '0:00:11']
['n_factors = 50', 'n_epochs = 30', 'algorithm = SVDpp', '0.752', '0.513', '0:01:49']
['n_factors = 50', 'n_epochs = 30', 'algorithm = NMF_biased', '1.501', '0.852', '0:00:16']
['n_factors = 50', 'n_epochs = 30', 'algorithm = NMF_unbiased', '0.766', '0.553', '0:00:15']
```

['n_factors = 50', 'n_epochs = 10', 'algorithm = SVD_biased', '0.804', '0.562', '0:00:04']
['n_factors = 50', 'n_epochs = 10', 'algorithm = SVD_unbiased', '0.780', '0.515', '0:00:04']
['n_factors = 50', 'n_epochs = 10', 'algorithm = SVDpp', '0.756', '0.513', '0:00:38']
['n_factors = 50', 'n_epochs = 10', 'algorithm = NMF_biased', '2.516', '1.801', '0:00:05']
['n_factors = 50', 'n_epochs = 10', 'algorithm = NMF_unbiased', '2.047', '1.698', '0:00:05']
['n_factors = 50', 'n_epochs = 20', 'algorithm = SVD_biased', '0.779', '0.539', '0:00:08']
['n_factors = 50', 'n_epochs = 20', 'algorithm = SVD_unbiased', '0.779', '0.514', '0:00:08']
['n_factors = 50', 'n_epochs = 20', 'algorithm = SVDpp', '0.752', '0.512', '0:01:13']
['n_factors = 50', 'n_epochs = 20', 'algorithm = NMF_biased', '0.997', '0.647', '0:00:11']
['n_factors = 50', 'n_epochs = 20', 'algorithm = NMF_unbiased', '0.963', '0.737', '0:00:10']
['n_factors = 50', 'n_epochs = 30', 'algorithm = SVD_biased', '0.773', '0.535', '0:00:12']
['n_factors = 50', 'n_epochs = 30', 'algorithm = SVD_unbiased', '0.771', '0.510', '0:00:11']
['n_factors = 50', 'n_epochs = 30', 'algorithm = SVDpp', '0.752', '0.513', '0:01:49']
['n_factors = 50', 'n_epochs = 30', 'algorithm = NMF_biased', '1.501', '0.852', '0:00:16']
['n_factors = 50', 'n_epochs = 30', 'algorithm = NMF_unbiased', '0.766', '0.553', '0:00:15']

['n_factors = 100', 'n_epochs = 10', 'algorithm = SVD_biased', '0.800', '0.558', '0:00:06']
['n_factors = 100', 'n_epochs = 10', 'algorithm = SVD_unbiased', '0.782', '0.514', '0:00:06']
['n_factors = 100', 'n_epochs = 10', 'algorithm = SVDpp', '0.749', '0.511', '0:00:58']
['n_factors = 100', 'n_epochs = 10', 'algorithm = NMF_biased', '1.624', '1.106', '0:00:09']
['n_factors = 100', 'n_epochs = 10', 'algorithm = NMF_unbiased', '2.365', '2.002', '0:00:09']
['n_factors = 100', 'n_epochs = 20', 'algorithm = SVD_biased', '0.781', '0.542', '0:00:12']
['n_factors = 100', 'n_epochs = 20', 'algorithm = SVD_unbiased', '0.778', '0.510', '0:00:12']

['n_factors = 100', 'n_epochs = 10', 'algorithm = SVD_biased', '0.800', '0.558', '0:00:06']
['n_factors = 100', 'n_epochs = 10', 'algorithm = SVD_unbiased', '0.782', '0.514', '0:00:06']
['n_factors = 100', 'n_epochs = 10', 'algorithm = SVDpp', '0.749', '0.511', '0:00:58']
['n_factors = 100', 'n_epochs = 10', 'algorithm = NMF_biased', '1.624', '1.106', '0:00:09']
['n_factors = 100', 'n_epochs = 10', 'algorithm = NMF_unbiased', '2.365', '2.002', '0:00:09']
['n_factors = 100', 'n_epochs = 20', 'algorithm = SVD_biased', '0.781', '0.542', '0:00:12']
['n_factors = 100', 'n_epochs = 20', 'algorithm = SVD_unbiased', '0.778', '0.510', '0:00:12']

['n_factors = 100', 'n_epochs = 20', 'algorithm = SVDpp', '0.748', '0.514', '0:01:59']
['n_factors = 100', 'n_epochs = 20', 'algorithm = NMF_biased', '1.045', '0.691', '0:00:17']
['n_factors = 100', 'n_epochs = 20', 'algorithm = NMF_unbiased', '1.020', '0.793', '0:00:17']
['n_factors = 100', 'n_epochs = 30', 'algorithm = SVD_biased', '0.779', '0.537', '0:00:18']
['n_factors = 100', 'n_epochs = 30', 'algorithm = SVD_unbiased', '0.777', '0.510', '0:00:18']
['n_factors = 100', 'n_epochs = 30', 'algorithm = SVDpp', '0.754', '0.521', '0:02:55']
['n_factors = 100', 'n_epochs = 30', 'algorithm = NMF_biased', '1.076', '0.690', '0:00:25']
['n_factors = 100', 'n_epochs = 30', 'algorithm = NMF_unbiased', '0.770', '0.562', '0:00:25']
['n_factors = 150', 'n_epochs = 10', 'algorithm = SVD_biased', '0.797', '0.556', '0:00:08']
['n_factors = 150', 'n_epochs = 10', 'algorithm = SVD_unbiased', '0.784', '0.514', '0:00:08']
['n_factors = 150', 'n_epochs = 10', 'algorithm = SVDpp', '0.747', '0.514', '0:01:26']
['n_factors = 150', 'n_epochs = 10', 'algorithm = NMF_biased', '1.218', '0.746', '0:00:12']
['n_factors = 150', 'n_epochs = 10', 'algorithm = NMF_unbiased', '2.521', '2.144', '0:00:12']
['n_factors = 150', 'n_epochs = 20', 'algorithm = SVD_biased', '0.781', '0.541', '0:00:16']
['n_factors = 150', 'n_epochs = 20', 'algorithm = SVD_unbiased', '0.784', '0.513', '0:00:16']

['n_factors = 150', 'n_epochs = 20', 'algorithm = SVDpp', '0.747', '0.514', '0:02:51']
['n_factors = 150', 'n_epochs = 20', 'algorithm = NMF_biased', '1.563', '0.882', '0:00:25']
['n_factors = 150', 'n_epochs = 20', 'algorithm = NMF_unbiased', '1.067', '0.832', '0:00:23']
['n_factors = 150', 'n_epochs = 30', 'algorithm = SVD_biased', '0.779', '0.542', '0:00:24']
['n_factors = 150', 'n_epochs = 30', 'algorithm = SVD_unbiased', '0.782', '0.511', '0:00:24']
['n_factors = 150', 'n_epochs = 30', 'algorithm = SVDpp', '0.754', '0.520', '0:04:11']
['n_factors = 150', 'n_epochs = 30', 'algorithm = NMF_biased', '1.134', '0.702', '0:00:34']
['n_factors = 150', 'n_epochs = 30', 'algorithm = NMF_unbiased', '0.774', '0.568', '0:00:34']

```
['n_factors = 200', 'n_epochs = 10', 'algorithm = SVD_biased', '0.798', '0.557', '0:00:10']
['n_factors = 200', 'n_epochs = 10', 'algorithm = SVD_unbiased', '0.785', '0.514', '0:00:10']
['n_factors = 200', 'n_epochs = 10', 'algorithm = SVDpp', '0.758', '0.517', '0:02:05']
['n_factors = 200', 'n_epochs = 10', 'algorithm = NMF_biased', '1.653', '0.935', '0:00:15']
['n_factors = 200', 'n_epochs = 10', 'algorithm = NMF_unbiased', '2.586', '2.205', '0:00:15']
['n_factors = 200', 'n_epochs = 20', 'algorithm = SVD_biased', '0.783', '0.543', '0:00:20']
['n_factors = 200', 'n_epochs = 20', 'algorithm = SVD_unbiased', '0.783', '0.511', '0:00:20']
['n_factors = 200', 'n_epochs = 20', 'algorithm = SVDpp', '0.757', '0.521', '0:03:49']
['n_factors = 200', 'n_epochs = 20', 'algorithm = NMF_biased', '1.720', '1.085', '0:00:30']
['n_factors = 200', 'n_epochs = 20', 'algorithm = NMF_unbiased', '1.100', '0.860', '0:00:30']
['n_factors = 200', 'n_epochs = 30', 'algorithm = SVD_biased', '0.782', '0.547', '0:00:30']
['n_factors = 200', 'n_epochs = 30', 'algorithm = SVD_unbiased', '0.782', '0.510', '0:00:30']
['n_factors = 200', 'n_epochs = 30', 'algorithm = SVDpp', '0.754', '0.526', '0:05:37']
['n_factors = 200', 'n_epochs = 30', 'algorithm = NMF_biased', '1.218', '0.744', '0:00:45']
['n_factors = 200', 'n_epochs = 30', 'algorithm = NMF_unbiased', '0.779', '0.574', '0:00:45']
```

이 중, min 값인 최종 결과는 다음과 같다.

```
minimize mean_rmse =  ['n_factors = 150', 'n_epochs = 10', 'algorithm = SVDpp', '0.747', '0.514', '0:01:26']
minimize mean_mae =  ['n_factors = 50', 'n_epochs = 30', 'algorithm = SVD_unbiased', '0.771', '0.510', '0:00:11']
minimize cv_time =  ['n_factors = 50', 'n_epochs = 10', 'algorithm = SVD_biased', '0.804', '0.562', '0:00:04']
```

따라서, 가장 좋은 성능을 보이는 모델 best_model_mf는 (RMSE 기준)
best_model_mf = surprise.SVDpp(n_factors=150, n_epochs=10, random_state=0) 이다.

3-4-5에서 고려한 사항은 다음과 같다.

n_factors: latent factor 수 (default=100)
default 값을 중심으로 해서 50, 100, 150, 200의 값을 가지도록 설정하였다.
n_epochs: SGD 수행 횟수 (default=20)
default 값을 중심으로 해서 10, 20, 30의 값을 가지도록 설정하였다.
SVD 및 NMF의 경우에는 biased or unbiased 고려하였다.