In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
from scipy import stats
```

In [2]:
```
# Read data
names = ['userid', 'itemid', 'rating', 'timestamp']
raw_data = pd.read_csv('./ml-100k/u.data', sep='\t', names=names)

# save data in a numpy array where each user ratings have their own rows
userids = sorted(list(raw_data['userid'].unique()))
itemids = sorted(list(raw_data['itemid'].unique()))

# first save in list of lists, use None values if user has not rated item
data = [[None] * len(itemids) for x in range(len(userids))]

# find ratings made by each user
for i in range(len(userids)):
    # dict of ratings for user i+1 (key = itemid, value = rating)
    user_ratings = dict(zip(raw_data.loc[raw_data['userid'] == (i+1)].itemid, raw_data.
    for j in range(len(itemids)):
        # check if user has rated item with id j+1
        if j+1 in user_ratings:
            data[i][j] = user_ratings[j+1]

data = np.array(data)
```

Part A

User-based collaborative filtering approach from Assignment 1

In [3]:
```
# a, b = userids, data = whole data set
def similarity(a,b, data):
    data_a = data[a-1] # remember that indexing starts from 0, but userids from 1
    data_b = data[b-1]

    # dicts with itemids and ratings
    dict_a = {i: r for i, r in enumerate(data_a, start=1) if r is not None}
    dict_b = {i: r for i, r in enumerate(data_b, start=1) if r is not None}

    # intersections of common itemids
    P = list(set(dict_a).intersection(set(dict_b)))

    if len(P) < 2:
        return 0

    # keep only common itemids
    dict_a = {id: dict_a[id] for id in P}
    dict_b = {id: dict_b[id] for id in P}

    # Create constants
```

```
    const_a = list(dict_a.values())
    const_b = list(dict_b.values())

    sim, p = stats.pearsonr(const_a, const_b)

    # Check for NaN
    if sim != sim:
        return 0
    return sim
```

In [4]:

```
# Similarity matrix
N = 0
sim_matrix = [[1] * len(userids) for x in range(len(userids))]
for i in range(len(userids)):
    for j in range(i+1, len(userids)):
        sim_matrix[i][j] = sim_matrix[j][i] = similarity(i+1, j+1, data)

sim_matrix = np.array(sim_matrix)
```

```
B:\Anaconda\envs\recommender\lib\site-packages\scipy\stats\stats.py:4023: PearsonRConsta
ntInputWarning: An input array is constant; the correlation coefficient is not defined.
  warnings.warn(PearsonRConstantInputWarning())
```

In [5]:

```
# a = userid, p = itemid, data = whole data set,
# sim = similarity matrix t = similarity threshold
def predict(a, p, data, sim_matrix, t):
    sim = sim_matrix[a-1]
    # mean of ratings given by user a
    mean_a = np.mean([r for r in data[a-1] if r is not None])

    # transform similarities to dict (key = userid, value = similarity) and filter out
    sim = {i: s for i, s in enumerate(sim, start=1) if s >= t}

    n = 0
    d = 0

    for b in sim:
        # chekc if user b has not rated the item
        if data[b-1][p-1] == None:
            continue

        mean_b = np.mean([r for r in data[b-1] if r is not None])
        n += sim[b] * (data[b-1][p-1] - mean_b)
        d += sim[b]

    if n == 0:
        return mean_a

    return mean_a + n/d
```

Average aggregation method

In [6]:

```
# g = groud of users (list of usedids), i = itemid, data = whole dataset
def average_aggregation(g, i, data):
    # ratings for item i, given by users in the group
    ratings = []
```

```
    # obtaing ratings, either from data or predict it
    for user in g:
        rating = data[user-1][i-1]
        if rating == None:
            rating = predict(user, i, data, sim_matrix, 10)
        ratings.append(rating)

    return np.average(ratings)
```

Least misery aggregation method

In [7]:

```
def least_misery_aggregation(g, i, data):
    # ratings for item i, given by users in the group
    ratings = []

    # obtaing ratings, either from data or predict it
    for user in g:
        rating = data[user-1][i-1]
        if rating == None:
            rating = predict(user, i, data, sim_matrix, 10)
        ratings.append(rating)

    return np.min(ratings)
```

Top 20 recommendations for a group of 3 users

In [10]:

```
g = [1, 11, 111]

# dicts for both aggregation ratings (key=itemid, value=group rating)
avg_ratings = {}
lm_ratings = {}

for i in itemids:
    avg_ratings[i] = average_aggregation(g, i, data)
    lm_ratings[i] = least_misery_aggregation(g, i, data)

# sort both dicts so that highly rated items for the group are first
avg_ratings = dict(sorted(avg_ratings.items(), key=lambda x: x[1], reverse=True))
lm_ratings = dict(sorted(lm_ratings.items(), key=lambda x: x[1], reverse=True))
```

Recommendations with average method

In [11]:

```
recommendations = dict(list(avg_ratings.items())[:20])
df = pd.DataFrame(list(zip(list(recommendations.keys()), list(recommendations.values()))
print(df)
```

```
   itemid    rating
0     258  4.666667
1       9  4.513889
2      15  4.513889
3     173  4.513889
4     196  4.513889
5     268  4.513889
```

```
6       269  4.488029
7       286  4.203431
8        28  4.180556
9        86  4.180556
10      100  4.180556
11      111  4.180556
12      191  4.180556
13      208  4.180556
14      242  4.154696
15      277  4.050654
16      318  4.050654
17      332  4.050654
18      357  4.050654
19      423  4.050654
```

Recommendations with least misery method

In [12]:

```
recommendations = dict(list(lm_ratings.items())[:20])
df = pd.DataFrame(list(zip(list(recommendations.keys()), list(recommendations.values())
print(df)
```

```
    itemid    rating
0      258  4.000000
1      286  3.610294
2      301  3.610294
3        9  3.541667
4       15  3.541667
5       22  3.541667
6       28  3.541667
7       47  3.541667
8       51  3.541667
9       56  3.541667
10      79  3.541667
11      86  3.541667
12     100  3.541667
13     107  3.541667
14     111  3.541667
15     135  3.541667
16     173  3.541667
17     185  3.541667
18     191  3.541667
19     194  3.541667
```

Part B

We propose that the disagreements between users are taken into account with disagreement variance (Sihem Amer-Yahia, Senjuti Basu Roy, Ashish Chawlat, Gautam Das, and Cong Yu. 2009. Group recommendation: semantics and efficiency. Proc. VLDB Endow. 2, 1 (August 2009), 754–765. DOI:https://doi-org.libproxy.tuni.fi/10.14778/1687627.1687713).

Disagreement variance is defined as $dis(g,i) = \frac{1}{|g|}\sum \limits _{u\in g} (r^{*}(u, i) - mean)^2 $, where $mean$ is the mean of ratings the users in group g have given to item i.

Using the calculated variance, group recommendations are computed with consensus function defined as $con(g,i) = w_1 \times r^{*}(g,i) + w_2 \times (1-dis(g,i))$, where $w_1 + w_2 = 1$. These weights define how important we want the group disagreement to be in the recommendation.

In [13]:

```
def disagreement_variance(g, i, data):
    # ratings for item i, given by users in the group
    ratings = []

    # obtaing ratings, either from data or predict it
    for user in g:
        rating = data[user-1][i-1]
        if rating == None:
            rating = predict(user, i, data, sim_matrix, 10)
        ratings.append(rating)

    ratings_mean = np.mean(ratings)

    # calculate and return the disagreement variance acording to the formula presented
    dis = (1/len(ratings) * np.sum([(r - ratings_mean) ** 2 for r  in ratings]))
    return dis
```

In [14]:

```
def consensus(g, i, data):
    w1 = 0.9
    w2 = 1-w1
    return w1 * average_aggregation(g, i, data) + w2 * (1-disagreement_variance(g, i, d
```

Show top 20 recommendations, where disagreements have been taken into account.

In [15]:

```
g = [1, 11, 111]
# dict for group ratings for all items (key=itemid, value=group rating for item)
ratings = {}

for i in itemids:
    ratings[i] = consensus(g, i, data)

# sort dict so that highly rated items for the group are first
ratings = dict(sorted(ratings.items(), key=lambda x: x[1], reverse=True))

recommendations = dict(list(ratings.items())[:20])
df = pd.DataFrame(list(zip(list(recommendations.keys()), list(recommendations.values())
print(df)
```

```
    itemid    rating
0      258  4.277778
1        9  4.115239
2       15  4.115239
3      173  4.115239
4      196  4.115239
5      268  4.115239
6      269  4.086804
7      286  3.848831
8       28  3.825424
9       86  3.825424
10     100  3.825424
11     111  3.825424
12     191  3.825424
13     208  3.825424
14     242  3.798713
15     277  3.700447
```

```
16     318  3.700447
17     332  3.700447
18     357  3.700447
19     423  3.700447
```

In [ ]: