Assignment 3 – Presentation – Asher Elazary

In [33]:

```python
from sklearn.model_selection import train_test_split

n_users = df.user_id.unique().shape[0]
n_items = df.item_id.unique().shape[0]
print(str(n_users) + ' users')
print(str(n_items) + ' items')

train_df, test_df = train_test_split(df, test_size=0.2, random_state = 10)
train_df, test_df

# Training Dataset
train_ds = np.zeros((n_users, n_items))
for row in train_df.itertuples():
    train_ds[row[1]-1, row[2]-1] = row[3]

# Testing Dataset
test_ds = np.zeros((n_users, n_items))
for row in test_df.itertuples():
    test_ds[row[1]-1, row[2]-1] = row[3]
```

```
943 users
1682 items
```

Slope One

$$\mathrm{dev}_{j,i} = \sum_{u \in S_{j,i}(\chi)} \frac{u_j - u_i}{card(S_{j,i}(\chi))}.$$

In [15]:

```
@njit
def make_pairwise_calcs(x):
    #zero arrays to build matrix for number of items
    dev_arr = np.zeros((n_items, n_items))
    freq_arr = np.zeros((n_items, n_items))

    #for each item pair, calculate the average deviation and store in matrix
    for j in range(n_items):
        #get u rows containing j items
        j_in_u = x[:, j] != 0
        if(np.any(j_in_u)):
            for i in range(j + 1, n_items):
                #get u rows containing i items
                i_in_u = x[:, i] != 0
                #get the boolean intersection mask for j and i items
                intersections = np.logical_and(j_in_u, i_in_u)
                #get the sets
                u_x_ji = x[intersections]
                #get the number of sets
                card_ji = u_x_ji.shape[0]
                #if co-rated items exist...
                if(card_ji > 0):
                    #calculate the avg deviation between the pairs
                    dev = np.sum((u_x_ji[:, j] - u_x_ji[:, i]) / card_ji)
                    #utilise matrix similarity
                    dev_arr[j, i] = dev
                    dev_arr[i, j] = -dev
                    freq_arr[j, i] = card_ji
                    freq_arr[i, j] = card_ji

    return dev_arr, freq_arr
```

$$P(u)_j = \frac{1}{card(R_j)} \sum_{i \in R_j} (\mathrm{dev}_{j,i} + u_i)$$

In [18]:

```
@njit
def predict_s1(u, j , x , dev_arr, freq_arr):
    #all rated item indices for user
    u_indices = np.where(x[u])[0]
    #item indices not including j
    i_neq_j = u_indices != j
    u_indices = u_indices[i_neq_j]
    #reset variables
    pre_ji = 0
    #for each item in the user's ratings
    for i in u_indices:
        #lookup the precomputed co-rating frequency
        card_ji = freq_arr[j, i]
        #if the co-rated item pair exists
        if(card_ji > 0):
            #lookup the average deviation per item pair
            dev_ji = dev_arr[j, i]
            #user prediction for one item pair average deviation + the user rating i
            pre_ji += ((dev_ji + x[u, i]))

    #item prediction is average of all predictions (in reciprocal form)
    return pre_ji * (1/u_indices.size)

def make_predictions_s1(X, y):
    pre_arr = np.copy(X)
    y_pre_indices = np.argwhere(y)

    dev_arr, freq_arr = make_pairwise_calcs(X)

    for this_pre in y_pre_indices:
        u = this_pre[0]
        j = this_pre[1]
        pre_arr[u][j] = predict_s1(u, j, X, dev_arr, freq_arr)

    return pre_arr
```

In [18]:

```
@njit
def predict_s1(u, j , x , dev_arr, freq_arr):
    #all rated item indices for user
    u_indices = np.where(x[u])[0]
    #item indices not including j
    i_neq_j = u_indices != j
    u_indices = u_indices[i_neq_j]
    #reset variables
    pre_ji = 0
    #for each item in the user's ratings
    for i in u_indices:
        #lookup the precomputed co-rating frequency
        card_ji = freq_arr[j, i]
        #if the co-rated item pair exists
        if(card_ji > 0):
            #lookup the average deviation per item pair
            dev_ji = dev_arr[j, i]
            #user prediction for one item pair average deviation + the user rating i
            pre_ji += ((dev_ji + x[u, i]))

    #item prediction is average of all predictions (in reciprocal form)
    return pre_ji * (1/u_indices.size)

def make_predictions_s1(X, y):
    pre_arr = np.copy(X)
    y_pre_indices = np.argwhere(y)

    dev_arr, freq_arr = make_pairwise_calcs(X)

    for this_pre in y_pre_indices:
        u = this_pre[0]
        j = this_pre[1]
        pre_arr[u][j] = predict_s1(u, j, X, dev_arr, freq_arr)

    return pre_arr
```

In [19]:

```
%%time
pre_arr_s1 = make_predictions_s1(train_ds, test_ds)
print(evaluate(test_ds, pre_arr_s1))
```

```
(0.763855716760231 3, 0.9803994548163598)
CPU times: user 22.3 s, sys: 1 s, total: 23.3 s
```

```
Wall time: 26 s
```

Weighted Slope One

$$P^{wS1}(u)_j = \frac{\sum_{i \in S(u)-\{j\}} (\mathrm{dev}_{j,i} + u_i) c_{j,i}}{\sum_{i \in S(u)-\{j\}} c_{j,i}}$$

In [22]:

```
#---WS1---#
@njit
def predict_ws1(u, j , x , dev_arr, freq_arr):

    u_indices = np.where(x[u])[0]
    i_neq_j = u_indices != j
    u_indices = u_indices[i_neq_j]

    #reset variables
    weighted_pre_ji = 0
    weighted_users = 0

    for i in u_indices:
        card_ji = freq_arr[j, i]
        if(card_ji > 0):
            dev_ji = dev_arr[j, i]
            #this time, we weight the prediction based on the amount of users that have rated item-pair
            weighted_pre_ji += ((dev_ji + x[u, i]) * card_ji)
            weighted_users += card_ji

    if(weighted_users > 0):
        #returned prediction for item is average of all predictions weighted by the users per item pair prediction
        return weighted_pre_ji / weighted_users
    else:
        return 0

def make_predictions_ws1(X, y):
    pre_arr = np.copy(X)
    y_pre_indices = np.argwhere(y)

    dev_arr, freq_arr = make_pairwise_calcs(X)

    for this_pre in y_pre_indices:
        u = this_pre[0]
        j = this_pre[1]
        pre_arr[u][j] = predict_ws1(u, j, X, dev_arr, freq_arr)

    return pre_arr
```

In [23]:

```
%%time
pre_arr_ws1 = make_predictions_ws1(train_ds, test_ds)
print(evaluate(test_ds, pre_arr_ws1))
```

```
(0.744669520796919, 0.9533358060475977)
CPU times: user 21.9 s, sys: 953 ms, total: 22.8 s
Wall time: 25.3 s
```

Centred Cosine Similarity

$$\text{cosine similarity} = S_C(A, B) := \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum\limits_{i=1}^{n} A_i B_i}{\sqrt{\sum\limits_{i=1}^{n} A_i^2} \sqrt{\sum\limits_{i=1}^{n} B_i^2}},$$

Image source: Wikipedia

In [27]:

```
@njit
def make_centred_cosine(x):

    def centred_cosine_similarity(a,b):
        #calculate the mean of each set (ignore 0 values)
        a_mean = np.sum(a) / np.count_nonzero(a)
        b_mean = np.sum(b) / np.count_nonzero(b)
        #subtract the mean of each set from itself (ignore 0 values)
        #normalise the magnitude of the vectors
        a = np.where(a > 0, a - a_mean, a)
        a = np.where(b > 0, a - b_mean, b)

        #calculate the dot product between sets
        this_dot = np.sum((a*b))
        #calculate the magnitude for each set
        mag_a = np.sqrt(np.sum(np.square(a)))
        mag_b = np.sqrt(np.sum(np.square(b)))
        this_mag = mag_a * mag_b
        #get the cosine angle, representing similarity between sets
        return (this_dot / this_mag)

    #calculate the symmetrical matrix of cosine similarities
    u_sim = np.ones((n_users, n_users))
    for u0 in range(n_users):
        for u1 in range(u0 + 1, n_users):
            this_sim = centred_cosine_similarity(x[u0], x[u1])
            #write to symetrical coordinates
            u_sim[u0, u1] =  this_sim
            u_sim[u1, u0] =  this_sim

    return u_sim
```

Modified Weighted Slope One

$$dev_{j,i} = \lambda \sum_{u \in S_{j,i}(\chi)} \frac{u_j - u_i}{card(S_{j,i}(\chi))} + (1-\lambda) \frac{\sum_{u \in S_{j,i}(\chi)} ((u_j - u_i) \cdot exp(sim(u, u')))}{\sum_{u \in S_{j,i}(\chi)} (exp(sim(u, u')) \cdot card(S_{j,i}(\chi)))},$$

In [24]:

```
@njit
def predict_ws1_modified(u_selected, j, x, u_sim, LAMBDA):

    #all rated item indices for user
    u_selected_indices = np.where(x[u_selected])[0]
    #reinitalise accumulators
    weighted_dev = 0
    weighted_users = 0

    #for each item in the user array
    for i in u_selected_indices:
    #if i == j, nex iteration
        if(i == j):
            continue

        #get u rows containing j items
        j_in_u = x[:, j] != 0
        #get u rows containing i items
        i_in_u = x[:, i] != 0
        #get the boolean intersection mask for j and i items
        intersections = np.logical_and(j_in_u, i_in_u)
        #get the indices
        u_set_indices = np.nonzero(intersections)[0]
        #get the number of users for co-rated sets
        card_ji = u_set_indices.size

        #if co-rating exists for item pair
        if(card_ji > 0):

            #deviation between item pairs (as vector)
            dev = x[u_set_indices, j] - x[u_set_indices, i]
            #calculate the average deviation between item pairs
            avg_dev = np.sum(dev / card_ji)

            #remove selected user from users of co-rated items
            u_set_indices = u_set_indices[u_set_indices != u_selected]
            #get all cosine similarities between selected users and users of co-rated items
            exp_cosine_sim = u_sim[u_set_indices, u_selected]
            #calculate the average deviation between item pairs, weighted by the user similarity between co-rated items
            exp_cosine_dev = np.sum(exp_cosine_sim * dev)
            exp_cosine_users = np.sum(card_ji * exp_cosine_sim)
            avg_user_sim = exp_cosine_dev / exp_cosine_users
            #interpolate between the two averages
            dev_ji = (LAMBDA * avg_dev) + ((1-LAMBDA)*(avg_user_sim))
            #user prediction for one item pair is the interpolation between the two deviation functions,
            #averaged for every i j pair
            weighted_dev += ((dev_ji + x[u_selected, i]) * card_ji)
            weighted_users += card_ji

    if(weighted_users > 0):
        return weighted_dev / weighted_users
    else:
        return 0
```

In [25]:

```
def make_predictions_ws1_modified(X, y):
    #make a copy of the train dataset
    pre_arr = np.copy(X)
    #identify rated indices of the test dataset
    y_pre_indices = np.argwhere(y)

    #calculate the user similarity matrix. Apply exp2 function to emphasise stronger similarities and minimise weaker simi
    u_sim = np.exp2(make_centred_cosine(X))

    #predict the ratings for these indices in the train dataset
    for this_pre in y_pre_indices:
        this_u = this_pre[0]
        this_j = this_pre[1]
        pre_arr[this_u][this_j] = predict_ws1_modified(this_u, this_j, X, u_sim, 0)

    return pre_arr
```

In [28]:

```
%%time
pre_arr_ws1_modified = make_predictions_ws1_modified(train_ds, test_ds)
print(evaluate(test_ds, pre_arr_ws1_modified))
```

```
(0.8106610636194549, 1.028542049922151)
CPU times: user 1min 12s, sys: 1.54 s, total: 1min 13s
Wall time: 1min 26s
```

References

‘Cosine similarity’ 2023, Wikipedia.

Lemire, D & Maclachlan, A 2008, ‘Slope One Predictors for Online Rating-Based Collaborative Filtering’,.

Ren, Y 2023, ‘Practical Data Science with Python - COSC 2670/2738 - Assignment 3’,.