Recommendation systems. Collaborative filtering.

Recommendation systems are engines predicting ratings which users may give to certain items. These systems are widely used to predict ratings of movies, books, news and many other things. There are several ways to produce recommendations and one of them is collaborative filtering. This method is based on collecting information about users' behaviour or preferences and prediction is based on similarities between users. Or in simpler terms: if users have similar preferences on some issues, they will likely have similar preferences on other issues.

Usually collaborating filtering is divided into two categories: user-based and item-based. In user-based we look for users who are similar to the target user and use their ratings to calculate the prediction for the target user. In item-based we create a matrix with relationships between the items, find the preferences of the active user based on the matrix and find items, which he could like.

These two ways are often called memory-based approach as they load all data into the memory. Advantages of memory-based approach are:

- Simplicity of implementation;
- Good explainability of the results;
- Easiness of adding new users;

Disadvantages:

- Could be slow, as loads all data into memory;
- Data usually is sparse, so there could be no similar items/users, which will make predictions impossible;

Another approach is model-based. This means building model based on the dataset to find underlying patterns in the data.

Advantages:

- Works better with sparse data;
- Less prone to overfit;

Disadvantges:

- Information could be lost due to dimensionality reduction;
- Most models have problems with explainability;

In this notebook I show how these methods can be implemented.

I use dataset with restaurants ratings. Citation, as requested:

Blanca Vargas-Govea, Juan Gabriel GonzÃ¡lez-Serna, Rafael Ponce-MedellÃn. Effects of relevant contextual features in the performance of a restaurant recommender system. In RecSysâ€™11: Workshop on Context Aware Recommender Systems (CARS-2011), Chicago, IL, USA, October 23, 2011.

1. Memory-based.

   1.1. User-based. Pearson correlation, neighborhood-based.

   1.2. Item-based. Slope-one recommendation.

   1.3. Cosine similarity.

2. Model-based. ALS.

In [1]:
```
import numpy as np
import pandas as pd
```

In [2]:
```
data = pd.read_csv('UCI/RCdata/rating_final.csv')
```

In [3]:
```
data.head(10)
```

Out[3]:

	userID	placeID	rating	food_rating	service_rating
0	U1077	135085	2	2	2
1	U1077	135038	2	2	1
2	U1077	132825	2	2	2
3	U1077	135060	1	2	2
4	U1068	135104	1	1	2
5	U1068	132740	0	0	0
6	U1068	132663	1	1	1
7	U1068	132732	0	0	0
8	U1068	132630	1	1	1
9	U1067	132584	2	2	2

Users give restaurants ratings based of food, service and overall quality. Possible ratings are 0, 1, 2. To distinguish zero ratings from lack of ratings I replace zero ratings with very small values. I'll use only overall rating in the analysis.

In [4]:
```
data['rating'] = data['rating'].apply(lambda x: 0.000001 if x == 0 else x)
```

In [5]:
```
#Sparse matrix.
ratings = data.pivot_table(index='userID', columns='placeID', values='rating
```

Memory-based.

User-based. Pearson correlation, neighborhood-based.

Algorithm uses Pearson correlation:

$$simil(x,y) = \frac{\sum\limits_{i \in I_{xy}} (r_{x,i} - \bar{r}_x)(r_{y,i} - \bar{r}_y)}{\sqrt{\sum\limits_{i \in I_{xy}} (r_{x,i} - \bar{r}_x)^2 \sum\limits_{i \in I_{xy}} (r_{y,i} - \bar{r}_y)^2}}$$

, where: r - ratings; x, y - users; Ixy is the set of items rated by both user x and user y.

At first similarities between users are calculates using Pearson correlation, then users, who are most similar to the target user are identified. Recommendations are generated based on their ratings.

In [6]:
```
def pearson(user1, user2, df):
    '''
    Calculates similarity between two users. Takes user's ids and dataframe
    '''

    df_short = df[df[user1].notnull() & df[user2].notnull()]

    if len(df_short) == 0:
        return 0

    else:
        rat1 = [row[user1] for i, row in df_short.iterrows()]
        rat2 = [row[user2] for i, row in df_short.iterrows()]

        numerator = sum([(rat1[i] - sum(rat1)/len(rat1)) * (rat2[i] - sum(ra
        denominator1 = sum([(rat1[i] - sum(rat1)/len(rat1)) ** 2 for i in ra
        denominator2 = sum([(rat2[i] - sum(rat2)/len(rat2)) ** 2 for i in ra

        if denominator1 * denominator2 == 0:
            return 0
        else:
            return numerator / ((denominator1 * denominator2) ** 0.5)
```

In [7]:
```
#Dataframe is transposed, for easier processing.
pearson('U1103', 'U1028', ratings.transpose())
```

Out[7]: 0.2970444347126886

In [8]:
```
def get_neighbours(user_id, df):
    '''
    Creates a sorted list of users, who are most similar to specified user.
    all other users and sort by similarity.
    '''
    distances = [(user, pearson(user_id, user, df)) for user in df.columns i

    distances.sort(key=lambda x: x[1], reverse=True)

    distances = [i for i in distances if i[1] > 0]
    return distances
```

In [9]:
```
get_neighbours('U1103', ratings.transpose())
```

Out[9]: [('U1068', 0.79056917787109249), ('U1028', 0.2970444347126886)]

In [10]:
```
def recommend(user, df, n_users=2, n_recommendations=2):
    '''
    Generate recommendations for the user. Take userID and Dataframe as inpu
    each place they rated. Return sorted list of places and their scores.
    '''

    recommendations = {}
    nearest = get_neighbours(user, df)

    n_users = n_users if n_users <= len(nearest) else len(nearest)

    user_ratings = df[df[user].notnull()][user]

    place_ratings = []

    for i in range(n_users):
        neighbour_ratings = df[df[nearest[i][0]].notnull()][nearest[i][0]]
        for place in neighbour_ratings.index:
            if place not in user_ratings.index:
                place_ratings.append([place,neighbour_ratings[place],nearest

    recommendations = get_ratings(place_ratings)
    return recommendations[:n_recommendations]

def get_ratings(place_ratings):

    '''
    Creates Dataframe from list of lists. Calculates weighted rarings for ea
    '''

    ratings_df = pd.DataFrame(place_ratings, columns=['placeID', 'rating', '

    ratings_df['total_weight'] = ratings_df['weight'].groupby(ratings_df['pl
    recommendations = []

    for i in ratings_df.placeID.unique():
        place_ratings = 0
```

```
        df_short = ratings_df.loc[ratings_df.placeID == i]
        for j, row in df_short.iterrows():
            place_ratings += row[1] * row[2] / row[3]
        recommendations.append((i, place_ratings))

    recommendations = [i for i in recommendations if i[1] >= 1]

    recommendations.sort(key=lambda x: x[1], reverse=True)
    return recommendations
```

In [11]:
```
recommend('U1068', ratings.transpose(),5,5)
```

Out[11]:
```
[(132564, 2.0), (132613, 1.3934554478666792), (132717, 1.0000005000000001)]
```

Item-based. Slope-one recommendation.

The idea behind slope one algorithm is simple: calculate average difference in ratings for each pair of items and use this difference as prediction. For example if users generally rate item A higher than item B by 1 point, then to predict rating for item A we take a rating of rating B by targer user and add 1 point. Usually there are more items that two, so weighted average is used. The algorithm's main advantages are simplicity and speed.

In [12]:
```
def get_dev_fr(data):

    '''
    Calculates average difference between each pair of places and frequency
    for cases, where a user rated both places.
    '''

    data_dev = pd.DataFrame(index=data.columns,columns=data.columns)
    data_fr = pd.DataFrame(index=data.columns,columns=data.columns)
    for i in data_dev.columns:
        for j in data_dev.columns:
            df_loc = data[data[i].notnull() & data[j].notnull()]
            if len(df_loc) != 0:

                data_dev.loc[i,j] = (sum(df_loc[i]) - sum(df_loc[j]))/len(df

                data_fr.loc[i,j] = len(df_loc) if i != j else 0
    return data_dev, data_fr
```

In [13]:
```
data_dev, data_fr = get_dev_fr(ratings)
```

In [14]:
```
def slopeone(user, data):

    '''
    Generate recommended ratings for each place which user didn't rate addir
    '''
    #Places, which user didn't rate. The condition finds nan values.
    recommendation = [i for i in data.columns if data.loc[user,i] != data.lc
    recommendation_dictionary = {}
```

```
    for j in recommendation:
        score = 0
        denominator = 0
        for i in data.columns.drop(recommendation):

            if data_dev.loc[j,i] == data_dev.loc[j,i] and data_fr.loc[j,i] =
                score += (data.loc[user,i] + data_dev.loc[j,i]) * data_fr.lc
                denominator += data_fr.loc[j,i]
        if denominator == 0:
            recommendation_dictionary[j] = 0
        else:
            score = score/denominator
            recommendation_dictionary[j] = score

    recommendation_dictionary = {k:round(v,2) for k, v in recommendation_dic
    return sorted(recommendation_dictionary.items(), key=lambda x: x[1], rev
```

In [15]:
```
slopeone('U1103', ratings)
```

Out[15]:
```
[(132564, 2.0), (132668, 1.5), (132608, 1.24), (132665, 1.0), (132715, 1.
0)]
```

Cosine similarity.

Another metric for similarity is cosine similarity.

$$\mathrm{simil}(x, y) = \cos(\vec{x}, \vec{y}) = \frac{\vec{x} \cdot \vec{y}}{||\vec{x}|| \times ||\vec{y}||} = \frac{\sum\limits_{i \in I_{xy}} r_{x,i} r_{y,i}}{\sqrt{\sum\limits_{i \in I_x} r_{x,i}^2} \sqrt{\sum\limits_{i \in I_y} r_{y,i}^2}}$$

Predictions are generated in a similar way to the previous methods - as a weighted rating of other users/items. Also it is a good idea to remove user's bias. Users tend to give low or high ratings for all movies. So I'll take in consideration average ratings of users.

$$\hat{r}_{xi} = \bar{r}_x + \frac{\sum\limits_y sim(x, y)(r_{y,i} - \bar{r}_y)}{\sum\limits_y |sim(x, y)|}$$

In [16]:
```
ratings_filled = data.pivot_table(index='userID', columns='placeID', values=
ratings_filled = ratings_filled.astype(float).values
```

In [17]:
```
def similarity(ratings, matrix_type='user', epsilon=1e-9):
    if matrix_type == 'user':
        sim = ratings.dot(ratings.T) + epsilon
    elif matrix_type == 'place':
        sim = ratings.T.dot(ratings) + epsilon
```

```
    norms = np.array([np.sqrt(np.diagonal(sim))])
    return (sim / norms / norms.T)
```

In [18]:
```
user_similarity = similarity(ratings_filled, matrix_type='user')
item_similarity = similarity(ratings_filled, matrix_type='place')
```

In [19]:
```
def predict(ratings, similarity, matrix_type='user'):

    '''
    Predict places based on similarity.
    '''

    if matrix_type == 'user':
        #Bias as sum of non-zero values divided by the number of non-zer0 va
        user_bias = np.true_divide(ratings.sum(axis=1),(ratings!=0).sum(axis
        ratings = (ratings - user_bias[:, np.newaxis]).copy()
        pred = similarity.dot(ratings) / np.array([np.abs(similarity).sum(ax
        pred += user_bias[:, np.newaxis]

    elif matrix_type == 'place':
        item_bias = np.true_divide(ratings.sum(axis=0),(ratings!=0).sum(axis
        ratings = (ratings - item_bias[np.newaxis, :]).copy()
        pred = ratings.dot(similarity) / np.array([np.abs(similarity).sum(ax
        pred += item_bias[np.newaxis, :]

    return pred

def recommend_cosine(rating, matrix, user):

    '''
    If user has rated a place, replace predicted rating with 0. Return top-5
    '''

    predictions = [[0 if rating[j][i] > 0 else matrix[j][i] for i in range(l
    recommendations = pd.DataFrame(index=ratings.index,columns=ratings.colum
    return recommendations[user].sort_values(ascending=False)[:5]
```

In [20]:
```
user_pred = predict(ratings_filled, user_similarity, matrix_type='user')
item_pred = predict(ratings_filled, item_similarity, matrix_type='place')
```

In [21]:
```
recommend_cosine(ratings_filled, item_pred, 'U1103')
```

Out[21]:
```
placeID
132660    0.7804
132955    0.6337
135034    0.6262
134986    0.6229
132922    0.4647
Name: U1103, dtype: float64
```

In [22]:
```
recommend_cosine(ratings_filled, user_pred, 'U1103')
```

Out[22]:
```
placeID
132660    0.3974
132740    0.3504
132608    0.3306
132594    0.2953
132609    0.1946
Name: U1103, dtype: float64
```

Model-based. ALS.

Model-based collaborative filtering strives to find latent features in the data. Matrix factorization is a commonly used method. It implies finding two matrices so that their multiplication will yield the matrix with ratings: ones that we already have and predicted ones. One matrix is for users (P), the other one is for items (Q). Ratings matrix (R) is their multiplication. One dimension of matrices P and Q is number of users/items respectively, the other is the number of latent features.

There are several algorithms with which matrix factorization could be done. Alternating least squares is one of them.

$$\underset{Q*,P*}{min} \sum_{(u,i)\epsilon K} (r_{ui} - Q_i^T P_u)^2 + \lambda(\|Q_i\|^2 + \|P_u\|^2)$$

The algorithms optimizes the difference between the original ratings and the ratings which are produced by the multiplication of aforementioned matrices. Second part of the formula is regularization. The idea of ALS is to fix one of matrices (P or Q), optimize for the other matrix, then at the next step fix the second matrix and optimize for the first one.

$$p_i = A_i^{-1} V_i \; with \; A_i = Q_{I_i} Q_{I_i}^T + \lambda n_{p_i} E \; and \; V_i = Q_{I_i} R^T(i, I_i)$$

$$q_j = A_j^{-1} V_j \; with \; A_j = P_{I_j} P_{I_j}^T + \lambda n_{q_j} E \; and \; V_j = P_{I_j} R^T(I_j, j)$$

P. S. This part is heavilly based on this article.

At first I need to split the data into train and test to check how the error changes with each step. Train and test should have the same dimensions as original data, so here is a function for it. I take users, who have rated at least one movie and select their three ratings - this is test. Train is all other values.

In [23]:
```
def train_test_split(ratings):
    test = np.zeros(ratings.shape)
    train = ratings.copy()
    non_zero = [i for i in range(ratings.shape[0]) if sum(ratings[i]) > 0]
    for user in non_zero:
        test_ratings = np.random.choice(ratings[user, :].nonzero()[0],
                                        size=3,
```

```
                                              replace=False)
        train[user, test_ratings] = 0.
        test[user, test_ratings] = ratings[user, test_ratings]

    return train, test
```

In [24]:
```
R, T = train_test_split(ratings_filled)
```

Now I need index matrix, where value 1 means that a certain user has rated a particular item.

In [25]:
```
I = R.copy()
I[I > 0] = 1
I[I == 0] = 0

I2 = T.copy()
I2[I2 > 0] = 1
I2[I2 == 0] = 0
```

In [26]:
```
def rmse(I,R,Q,P):
    return np.sqrt(np.sum((I * (R - np.dot(P.T,Q)))**2)/len(R[R > 0]))
```

In [27]:
```
def als(R=R, T=T, lmbda=0.1, k=40, n_epochs=30, I=I, I2=I2):

    '''
    Function for ALS. Takes matrices and parameters as inputs.
    Lmbda - learning rate;
    k - dimensionality of latent feature space,
    n_epochs - number of epochs for training.
    '''
    #Number of users and items.
    m, n = R.shape
    P = 1.5 * np.random.rand(k,m) # Latent user feature matrix.
    Q = 1.5 * np.random.rand(k,n) # Latent places feature matrix.
    Q[0,:] = R[R != 0].mean(axis=0) # Avg. rating for each movie for initial
    E = np.eye(k) # (k x k)-dimensional idendity matrix.

    train_errors = []
    test_errors = []


    for epoch in range(n_epochs):
        # Fix Q and estimate P
        for i, Ii in enumerate(I):
            nui = np.count_nonzero(Ii)
            if (nui == 0): nui = 1

            a = np.dot(np.diag(Ii), Q.T)
            Ai = np.dot(Q, a) + lmbda * nui * E
            v = np.dot(np.diag(Ii), R[i].T)
            Vi = np.dot(Q, v)
            P[:,i] = np.linalg.solve(Ai,Vi)

        # Fix P and estimate Q
```

```
        for j, Ij in enumerate(I.T):
            nmj = np.count_nonzero(Ij)
            if (nmj == 0): nmj = 1

            a = np.dot(np.diag(Ij), P.T)
            Aj = np.dot(P, a) + lmbda * nmj * E
            v = np.dot(np.diag(Ij), R[:,j])
            Vj = np.dot(P, v)
            Q[:,j] = np.linalg.solve(Aj,Vj)

        train_rmse = rmse(I,R,Q,P)
        test_rmse = rmse(I2,T,Q,P)
        train_errors.append(train_rmse)
        test_errors.append(test_rmse)

        print(f'[Epoch {epoch+1}/{n_epochs}] train error: {train_rmse:6.6},

        if len(train_errors) > 1 and test_errors[-1:] > test_errors[-2:-1]:
            break
    print('Test error stopped improving, algorithm stopped')

    R = pd.DataFrame(R)
    R.columns = ratings.columns
    R.index = ratings.index

    R_pred = pd.DataFrame(np.dot(P.T,Q))
    R_pred.columns = ratings.columns
    R_pred.index = ratings.index

    return pd.DataFrame(R), R_pred
```

In [28]:

```
R, R_pred = als()
```

```
[Epoch 1/30] train error: 0.720442, test error: 1.13765
[Epoch 2/30] train error: 0.309338, test error: 0.961706
[Epoch 3/30] train error: 0.249321, test error: 0.948285
[Epoch 4/30] train error: 0.224138, test error: 0.944137
[Epoch 5/30] train error: 0.211072, test error: 0.94301
[Epoch 6/30] train error: 0.203513, test error: 0.943203
Test error stopped improving, algorithm stopped
```

So this is it. Now let's compare original and predicted ratings of a user.

In [29]:

```
user_ratings = R.transpose()['U1123'][R.transpose()['U1123'].sort_values(asc
predictions = pd.DataFrame(user_ratings)
predictions.columns = ['Actual']
predictions['Predicted'] = R_pred.loc['U1123',user_ratings.index]
predictions
```

Out[29]:

	Actual	Predicted
placeID		
132584	1.0	0.995225
132594	1.0	0.923987
132733	1.0	0.994598
132740	1.0	0.874841
135104	2.0	1.554813

The difference in ratings truly isn't very big. And now let's see the recommendations.

In [30]:
```
R_pred.loc['U1123',set(R_pred.transpose().index)-set(user_ratings.index)].sc
```

Out[30]:
```
placeID
135034    1.497545
132958    1.487171
132723    1.453056
135075    1.435531
132755    1.422608
Name: U1123, dtype: float64
```

Conclusions

Collaborative filtering has a lot of methods, which have certain advantages and disadvantages. So methods should be chosen depending on the cituation. Also collaborative filtering can be combined with other approaches to building recommendation systems - like content-based approaches.