Assignment 1

Fall 2018 CSE 258 Recommender System and Data Mining

Qi Ma

PID: A53263366

11/13/2018

Kaggle Username:

1. Purchase Prediction: Macchiato
2. Rating Prediction: Macchiato

Task 1: Purchase Prediction

Model 1: kth nearest neighbors:

In the pattern recognition, the k-nearest neighbors algorithm (k-NN) is a non-parametric method used for classification and regression. The input consists of the k closest training examples in the feature space. The output depends on whether k-NN is used for classification or regression.

Evaluate similarity based on the item categories the reviewer purchased.

By intuition that the reviewers would like to purchase an item which was purchased by other similar reviewers before, we assume that the k nearest neighbors of the user ever purchased this item, then the reviewer would like to buy it most likely.

And for items and reviewers that never show before, it is highly likely that they are new, we would predict them as 0.

Code snippet:

1. Use the total dataset to build kNN model. Make a dictionary to store the items that each reviewer ever purchased.

In [3]:
```
reviewers_purchased_count = {}
items_purchased_count = {}
reviewer_index = []
item_index = []
reviewers_items = defaultdict(set)
#a=0
for datum in data:
    #a+=1
    #print a
    if datum['reviewerID'] not in reviewers_purchased_count:
        reviewers_purchased_count[datum['reviewerID']] = 1
    else:
        reviewers_purchased_count[datum['reviewerID']] += 1
    if datum['itemID'] not in items_rate_count:
        items_purchased_count[datum['itemID']] = 1
    else:
        items_purchased_count[datum['itemID']] += 1
    if datum['reviewerID'] not in user_index:
        reviewer_index.append(datum['reviewerID'])
    if datum['itemID'] not in item_index:
        item_index.append(datum['itemID'])
```

2. For each reviewer, traverse the whole dataset to calculate the Jaccard similarity with other reviewers and then make a ranking based on Jaccard similarity then choose the first kth reviewers whose similarity is high according to this reviewer.

In [6]:
```
def kNN(k,features):
    total_reviewers = len(features)
    k_neighbor = [[0] * k for i in range(len(features))]
    for reviewer in range(total_reviewers):
        similarity = [0]*total_reviewers
        for other_reviewer in range(total_reviewers):
            distance = len(features[reviewer]&features[other_reviewer])
            similarity[other_reviewer] = distance
        for neighbor in range(k):
            max_similarity_neighbor = similarity.index(max(similarity))
            k_neighbor[reviewer].append(max_similarity_neighbor)
            similarity[max_similarity_neighbor] = 0
    return k_neighbor
```

Parameter tuning:

At the very first, I set the k = 5, such that for each reviewer, calculate the 5 nearest reviewers for future prediction.

```
k_neighbors = kNN(5, reviewers_items)
```

```
k_neighbors
```

```
[[0, 0, 0, 0, 0, 0, 34277, 550, 789, 868],
 [0, 0, 0, 0, 0, 1, 9361, 13698, 221, 292],
 [0, 0, 0, 0, 0, 2, 4971, 26458, 89, 170],
 [0, 0, 0, 0, 0, 3, 10908, 13139, 43, 146],
 [0, 0, 0, 0, 0, 4, 24, 188, 221, 384],
 [0, 0, 0, 0, 0, 5, 2703, 9806, 25, 252],
 [0, 0, 0, 0, 0, 6, 793, 2820, 3760, 4968],
 [0, 0, 0, 0, 0, 7, 169, 454, 1811, 2156],
 [0, 0, 0, 0, 0, 8, 209, 257, 837, 939],
 [0, 0, 0, 0, 0, 9, 9507, 26284, 188, 909],
 [0, 0, 0, 0, 0, 10, 2471, 2853, 4332, 4542],
 [0, 0, 0, 0, 0, 11, 317, 328, 516, 725],
 [0, 0, 0, 0, 0, 12, 161, 400, 1196, 1259],
 [0, 0, 0, 0, 0, 13, 711, 1725, 2189, 2358],
 [0, 0, 0, 0, 0, 14, 704, 916, 1698, 2438],
 [0, 0, 0, 0, 0, 15, 15780, 16726, 47, 104],
 [0, 0, 0, 0, 0, 16, 4307, 1411, 1679, 1870],
 [0, 0, 0, 0, 0, 17, 4345, 9251, 16480, 52],
 [0, 0, 0, 0, 0, 18, 143, 601, 840, 1316],
 [0, 0, 0, 0, 0, 19, 2604, 3045, 3165, 4016],
```

But the result is limited only to 5 nearest reviewers, they might just purchase limited amounts of items, which results for most zeros in prediction.

So we could increase the value of k, the result is improved. However, it takes a lot of time to train the model, as the increasing of k, the computation becomes more complexed.

3. For the k neighbors of each reviewer, calculate the weight of each item they have purchased.

In [10]:
```
X={}
Y={}
user_count = {}
for user in range(len(user_index)):
    X[user] = {}
    Y[user] = {}
    weight=len(users_items[user])/10.642260416112382
    prefer_items = {}
    for neighbor in k_neighbors[user]:
        for item in users_items[neighbor]:
            if item not in prefer_items.keys():
                prefer_items[item] = 1.0/len(users_items[neighbor])
            else:
                prefer_items[item] += 1.0/len(users_items[neighbor])
    for i in prefer_items:
        X[user][i] = prefer_items[i]/weight
        Y[user][i] = 1 if i in users_items[user] else 0
```

4. For the test set, read the reviewer-item pairs. Check whether the reviewer or item is in the previous data set or not. Then make a judgement in specified situation.

```
for threshould in [1.0,0.99,0.98,0.97,0.96,0.95,0.94,0.93,0.92,0.91,0.9,0.89,0.8,0.7,0.6,0.5]:
    predictions = open("predictions_Purchase.csv", 'w')
    for l in open("pairs_Purchase.txt",'r'):
        if l.startswith("userID"):
            #header
            predictions.write(l)
            continue
        username,itemname = l.strip().split('-')
        if username in user_index and itemname in item_index:
            u_index = user_index.index(username)
            i_index = item_index.index(itemname)
            if i_index in X[u_index].keys():
                if Y[u_index][i_index] == 1:
                    prediction = 1
                else:
                    sorted(X[u_index].items(), key=lambda item:item[1] , reverse=True)
                    if list(X[u_index]).index(i_index) < len(X[u_index])*threshould:
                        prediction = 1
                    else:
                        prediction = 0
            else:
                prediction = 0
        else:
            prediction = 0

        if prediction == 1:
            predictions.write(username + '-' + itemname + ",1\n")
        else:
            predictions.write(username + '-' + itemname + ",0\n")
    predictions.close()
```

Analysis: If we want to take more neighbors of a reviewer into consideration, larger k is required. However, it takes so much time to calculate kNN, especially when k is very large. So I consider other method based on similarity comparison.

Model 2: Similarity comparison

Reviewers would like to visit an item which was already purchased by people who is similar to themselves.

The most important part of the prediction is to build the distance model

1. Euclidean distance model

(reviewer_item[reviewer1] ∪ reviewer_item[reviewer1])

Code snippets:

```
reviewer_item = defaultdict(list)
item_reviewer = defaultdict(list)
for l in data:
    reviewer,item = l['reviewerID'],l['itemID']
    reviewer_item[reviewer].append(item)
    item_reviewer[item].append(reviewer)
```

```
predictions = open("predictions_p.csv", 'w')
for l in open("pairs_Purchase.txt"):
    if l.startswith("reviewerID"):
        #header
        predictions.write(l)
        continue
    reviewer1,item = l.strip().split('-')
    max = (reviewer2, -1)
    if reviewer1 in reviewer_item and item in item_reviewer:
        for reviewer2 in item_reviewer[item]:
            intersection = list(set(reviewer_item[reviewer1]).intersection(set(reviewer_item[reviewer2])))
            union = list(set(reviewer_item[reviewer1]).union(set(reviewer_item[reviewer2])))
            sim = float(len(intersection) / len(union))
            max = max if max[1] > sim else (reviewer2, sim)
        print(max)
        if max[1] > 0.003:
            predictions.write(reviewer1 + '-' + item + ',1\n')
        else:
            predictions.write(reviewer1 + '-' + item + ',0\n')
    else:
        predictions.write(reviewer1 + '-' + item + ',0\n')
predictions.close()
```

The result on Kaggle is: 0.60307

3. Pearson correlation: use rating to calculate similarity

$$Sim\ (u, v) = \frac{\sum_{i \in I_u \cap I_v}(R_{u,v} - \bar{R}_u)(R_{v,i} - \bar{R}_v)}{\sqrt{\sum_{i \in I_u \cap I_v}(R_{u,i} - \bar{R}_u)^2 \ \sum_{i \in I_u \cap I_v}(R_{v,i} - \bar{R}_v)^2}}$$

```
In [ ]: def pearson(uid1, uid2, reviewer1, reviewer2, sim):
            if len(sim) == 0:
                return 0
            avg_reviewer1 = float(sum(reviewer1[item] for item in reviewer1)) / len(reviewer1)
            avg_reviewer2 = float(sum(reviewer2[item] for item in reviewer2)) / len(reviewer2)
            nominator = sum((reviewer_item[uid1][item] - avg_reviewer1)*(reviewer_item[uid2][item] -
                                                                         avg_revewier2) for item in sim)
            denominator  = sum(((reviewer_item[uid1][item] - avg_reviewer1)**2)*((reviewer_item[uid2][item] -
                                                                                  avg_reviewer2)**2) for item in sim)**0.5
            if denominator == 0:
                return 1
            return float(nominator) / denominator
```

And I compare several times to tune a better boundary to determine if similarity beyond how much, the model will get better prediction performance, which is highly relied on the accuracy.

```
def makePrediction(user, item):
    if user not in user_list:
    # If we don't know about the user, then see how popular it is?
        randomGuessCount.append((user,item))
        return 1
    if item not in item_list:
    # If we don't know about the item, then see how active user is?
        randomGuessCount.append((user,item))
        return 1
    otherItemsBoughtByUser = [entry['itemID'] for entry in I_u[user]]
  # Finally, If this item is similar enough to one of the other items this
  # user bought, predict 1
    for otherItem in otherItemsBoughtByUser:
        similarity = PearsonItemSimilarity(i, otherItem)
    if math.fabs(similarity) > 0.6:
        return 1
    if item in return1 or user in return2:
        return 1
    return 0
```

Finally, I found that when we set 0.6 as the determine boundary, the Pearson Similarity model will get better performance among all.

The score evaluated by Kaggle is 0.63600, we can see that there is a little improvement compared with kNN model.

Task 2: Rating Prediction

Model 1: The second simple Latent-Factor model

Inspired by Homework 3, I firstly use the second-simplest model the professor has discussed in class.

Assumption: Since the task for prediction is rating, the model applied for this task is Latent Factor Model. We take reviewer and item rating bias into consideration.

$$f(u,i) = \alpha + \beta_u + \beta_i$$

To get α, β_u and β_i, we need to:

- Build two dictionaries for referencing reviewer and item (reviewer_item and item_reviewer) pairs.
- Tune lambda and repeat step 2-6 until converge
- Iteratively update optimized α, β_u and β_i using regression, every time fix another two variables when computing another.

$$\alpha = \frac{\sum_{u,i \in train}(R_{u,i} - (\beta_u + \beta_i))}{N_{train}}$$

$$\beta_u = \frac{\sum_{i \in I_u} R_{u,i} - (\alpha + \beta_i)}{\lambda + |I_u|}$$

$$\beta_i = \frac{\sum_{u \in U_i} R_{u,i} - (\alpha + \beta_u)}{\lambda + |U_i|}$$

- Build β_i and β_u dictionary to store
- In prediction, if a new reviewer or new item appears, we will ignore the bias and the average rating accordingly

- Calculate the accuracy

Code snippets:

```
def train(lamda, Average, reviewer_item, item_reviewer, pair_rating):
    alpha = 0
    beta_reviewer = defaultdict(int)
    beta_item = defaultdict(int)

    i=0
    while i < 500:
        i += 1

        for reviewer in reviewer_item.keys():
            beta_reviewer[reviewer]=sum((pair_rating[reviewer + x][0]-Average -beta_item[x]) /
                                        for x in reviewer_item[reviewer])/(lamda+len(reviewer_item[reviewer]))
        for item in item_reviewer.keys():
            beta_item[item]=sum((pair_rating[x + item][0]-Average-beta_reviewer[x]) f  /
                                or x in item_reviewer[item])/(lamda+len(item_reviewer[item]))

    for reviewer in reviewer_item.keys():
        for item in reviewer_item[reviewer]:
            alpha += ((pair_rating[reviewer+item][0]-beta_item[item]-beta_reviewer[reviewer]))/100000
    print ("alpha", alpha)

    MSE=0
    for l in valid_data:
        reviewer,item = l['reviewerID'],l['itemID']
        rate_predict=beta_reviewer[reviewer]+beta_item[item]+alpha
        MSE = MSE + (rate_predict - l['rating']) ** 2
    MSE=MSE/100000
    print("lamda is: ", lamda)
    print("MSE is: ", MSE)
    return alpha, beta_reviewer, beta_item
```

Tune the value of lambda:

```
lamda_test=[1, 4, 5, 6, 7, 8, 10, 100]
for lamda in lamda_test:
    train(lamda, Average, reviewer_item, item_reviewer,pair_rating)
```

```
alpha 4.231388674091933
lamda is:  1
MSE is:  1.28113923201379
alpha 4.230918478095691
lamda is:  4
MSE is:  1.1454069139152079
alpha 4.230876700210596
lamda is:  5
MSE is:  1.1399110617720556
alpha 4.230854964744218
lamda is:  6
MSE is:  1.1379377877593821
alpha 4.23084569302904
lamda is:  7
MSE is:  1.1377804626335801
alpha 4.23084440310799
lamda is:  8
MSE is:  1.1386031650822064
alpha 4.230855668883374
lamda is:  10
MSE is:  1.1416124042480875
alpha 4.231466168529679
lamda is:  100
MSE is:  1.1998254049208708
```

When lambda is 6.7, we get the approximately peak location. The score on Kaggle is 1.15132.

```
alpha, beta_reviewer, beta_item = train(6.7, Average, reviewer_item, item_reviewer, pair_rating)
predictions = open("predictions_Rating.csv", 'w')
for l in open("pairs_Rating.txt"):
    if l.startswith("reviewerID"):
        predictions.write(l)
        continue
    reviewer, item = l.strip().split('-')
    rating_pred = alpha + beta_reviewer[reviewer] + beta_item[item]
    predictions.write(reviewer + '-' + item + "," + str(rating_pred) + '\n')
predictions.close()
```

```
alpha 4.2308474742046585
lamda is:  6.7
MSE is:  1.1376969305466105
```

Model 2: more complex model (take γ_u, γ_i into consideration)

Then we consider more complicated model, adding the gamma.

$$Prediction = \alpha + \beta_u + \beta_i + \gamma_u \gamma_i$$

α is the global average value of rating for all training data.

β_u is the reviewer's rating bias above or below the average rating value.

β_i is the item's rating bias between items received rating and average rating value.

$$\arg\min_{\alpha,\beta,\gamma} \underbrace{\sum_{u,i}(\alpha+\beta_u+\beta_i+\gamma_u\cdot\gamma_i-R_{u,i})^2}_{\text{error}}+\lambda \underbrace{\left[\sum_u \beta_u^2 + \sum_i \beta_i^2 + \sum_i \|\gamma_i\|_2^2 + \sum_u \|\gamma_u\|_2^2\right]}_{\text{regularizer}}$$

γ_u and γ_i are randomly initialized with dimension K.

We can solve the α, β_u, and β_i by closed form just like what we have done in Homework 3.

```
reviewers_items = {}
items_reviewers = {}
for datum in train_set:
    u, i = datum['reviewerID'], datum['itemID']
    if not reviewers_items.get(u):
        reviewers_items[u] = [(i, datum['rating'])]
    else:
        reviewers_items[u].append((i, datum['rating']))
    if not items_reviewers.get(i):
        items_reviewers[i] = [(u, datum['rating'])]
    else:
        items_reviewers[i].append((u, datum['rating']))
sum_rating = 0
count_rating = 0
for u,i in reviewers_items.items():
    for i_info in i:
        sum_rating += i_info[1]
        count_rating += 1
alpha = float(sum_rating)/count_rating
```

```
beta_u = {}
for u in reviewers_items.keys():
    count_bias = 0
    count_rate = 0
    for i in reviewers_items[u]:
        count_rate += 1
        count_bias += float(i[1]-alpha)
    beta_u[u]=float(count_bias)/count_rate
```

```
beta_i = {}
for i in items_reviewers.keys():
    count_bias = 0
    count_rate = 0
    for u in items_reviewers[i]:
        count_rate += 1
        count_bias += float(u[1]-alpha)
    beta_i[i]=float(count_bias)/count_rate
```

```
for u, i in reviewers_items.items():
    for info in i:
        sum_rating += info[1]
        count_rating += 1
alpha = float(sum_rating)/count_rating
```

```
gamma_u = {}
gamma_i = {}
K = 10
num_reviewerss = K
num_items = K
for u in reviewers_items.keys():
    gamma_u[u] = numpy.random.random((1,K))
for i in items_reviewers.keys():
    gamma_i[i] = numpy.random.random((1,K))
```

```
for i in reviewers_items.keys():
    for j in range(K):
        gamma_u[i][0][j] = gamma_u[i][0][j] - 0.5
        gamma_u[i][0][j] = gamma_u[i][0][j] * 0.00001
for i in items_reviewers.keys():
    for j in range(K):
        gamma_i[i][0][j] = gamma_i[i][0][j] - 0.5
        gamma_i[i][0][j] = gamma_i[i][0][j] * 0.00001
```

Method to updating parameters:

$$argmin_{\alpha,\beta,\gamma} \sum_{u,i} (\alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i - R_{u,i})^2 + \lambda[\sum_u \beta_u^2 + \sum_i \beta_i^2 + \sum_i \|y_i\|_2^2 + \sum_u \|y_u\|_2^2]$$

1. Find α, β and determine K.
2. Initialize value of y_u and y_i randomly between [-0.0000005,0.000005] (I tried several times, when we set the interval in this range, even though we narrow the interval range, there might be little influence on the final MSE result.)
3. Fixed y_i, use closed form solution to update α and β. Use alternating

least squares to update y_u.

4. Fixed y_u, use closed form solution to update α and β. Use alternating least squares to update y_i.

5. Repeat step 3 and 4 until convergence

```
def update_alpha_beta(Alpha,Betareviewer,Betaitem,gammaU,gammaI,lam):
    sum_for_alpha = 0
    for u in reviewers_items.keys():
        for i in reviewers_items[u]:
            sum_for_alpha += i[1] - Betareviewer[u] - Betaitem[i[0]] - gammaU[u].dot(gammaI[i[0]].transpose())[0][0]
    Alpha = float(sum_for_alpha) / len(train_set)
    # Update beta_reviewer
    for u in reviewers_items.keys():
        sum_for_betareviewer = 0
        count_item = 0
        for i in reviewers_items[u]:
            count_item += 1
            sum_for_betareviewer += i[1] - Alpha - Betaitem[i[0]] - gammaU[u].dot(gammaI[i[0]].transpose())[0][0]
        Betareviewer[u] = float(sum_for_betareviewer) / (lam + count_item)
    # Update beta_item
    for i in items_reviewers.keys():
        sum_for_betaitem = 0
        count_reviewer = 0
        for u in items_reviewers[i]:
            count_reviewer += 1
            sum_for_betaitem += u[1] - Alpha - Betareviewer[u[0]] - gammaU[u[0]].dot(gammaI[i].transpose())[0][0]
        Betaitem[i] = float(sum_for_betaitem) / (lam + count_reviewer)
    return Alpha,Betareviewer,Betaitem
```

```
def update_gamma_u(Alpha,Betareviewer,Betaitem,gammaU,gammaI,K,lam,rate,index):
    total = len(train_set)
    # Update gammaU
    for u in reviewers_items.keys():
        sum_for_gammaU = 0
        update_flag = bool(random.getrandbits(1))
        if (update_flag):
            for i in reviewers_items[u]:
                sum_for_gammaU += i[1] - Alpha - Betareviewer[u] - Betaitem[i[0]] - gammaU[u].dot
                (gammaI[i[0]].transpose())[0][0]
                for count_gamma in [index]:
                    diff = float(-2 * sum_for_gammaU * gammaI[i[0]][0][count_gamma]) / total
                    gammaU[u][0][count_gamma] = gammaU[u][0][count_gamma] - rate * diff
    return gammaU
```

```
def update_gamma_i(Alpha,Betareviewer,Betaitem,gammaU,gammaI,K,lam,rate,index):
    total = len(train_set)
    # Update gammaI
    for i in items_reviewers.keys():
        sum_for_gammaI = 0
        update_flag = bool(random.getrandbits(1))
        if (update_flag):
            for u in items_reviewers[i]:
                sum_for_gammaI += u[1] - Alpha - Betareviewer[u[0]] - Betaitem[i] - gammaU[u[0]].dot
                (gammaI[i].transpose())[0][0]
                for count_gamma in [index]:
                    diff = float(-2 * sum_for_gammaI * gammaU[u[0]][0][count_gamma]) / total
                    gammaI[i][0][count_gamma] = gammaI[i][0][count_gamma] - rate * diff
    return gammaI
```

Then we define if the updated gamma_i and gamma_u is no more than 0.000001 larger than the original ones, they are converged respectively.

Finally, we get the MSE evaluated by Kaggle as 1.14412.