Assignment 1 Report

Name: Yan Sun

ID: A53240727

Kaggle username: yansun1996
Display Name: TripleFireYan

Visit Prediction

Method: k-Nearest Neighbor (kNN)

1. Use the whole dataset for building kNN model. Make a dictionary to record the items that each user visit.

```
users_rate_count = {}
items_rate_count = {}
user_index = []
item_index = []
users_items = defaultdict(set)
#a=0
for datum in data:
    #a+=1
    #print a
    if datum['userID'] not in users_rate_count:
        users_rate_count[datum['userID']] = 1
    else:
        users_rate_count[datum['userID']] += 1
    if datum['businessID'] not in items_rate_count:
        items_rate_count[datum['businessID']] = 1
    else:
        items_rate_count[datum['businessID']] += 1
    if datum['userID'] not in user_index:
        user_index.append(datum['userID'])
    if datum['businessID'] not in item_index:
        item_index.append(datum['businessID'])
```

```
for datum in data:
    u_index = user_index.index(datum['userID'])
    i_index = item_index.index(datum['businessID'])
    users_items[u_index].add(i_index)
```

2. As for each user, go through the whole dataset to calculate the Jaccard similarity with other users and then make a ranking based on Jaccard similarity then choose the first kth users whose similarity is high with this user.

###

For user x in data:

For user y in data:

Calculate Jaccard Similarity of x and y

Rank the user based on Jaccard Similarity with user x

Get the first Kth users that have high similarity with user x, then assign them as the k-Nearest Neighbor of user x.

###

```
def kNN(k,u_i):
    total=len(u_i)
    k_neighbor = defaultdict(set)
    for u1 in range(total):
        similarity = [0]*total
        for u2 in range(total):
            similarity[u2] = float(len(u_i[u1] & u_i[u2])) / (len(u_i[u1])+len(u_i[u2]))
        for count in range(k):
            max_similarity_neighbor = similarity.index(max(similarity))
            k_neighbor[u1].add(max_similarity_neighbor)
            similarity[max_similarity_neighbor] = 0
    return k_neighbor
```

3. As for the k neighbors of each user, calculate the weight of each items they have visited based on the following algorithms:

##

For user x in data:

For user y in k-Nearest Neighbor of user x:

Weight = length of items visited by user y / average # of items visited by all users

For item z in items visited by user y:

Score of item z += (1 / length (items visited by user y)) / Weight

Rank all the items visited by all neighbors according to scores

Check whether these items have been visited by user x before

##

```
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 users_k_neighbor[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. As for the test data, read the user-item pair. Check whether the user or item is in the previous data set or not. Then make judgement based on specific situation.

##

For user x and item y pair in test data:

If both x and y in previous dataset:

If x has visited y before:

Prediction = 1

Else:

In order to get the most possible item, make a threshold in the ranking of all X's k-Nearest neighbor's visited items, higher score is better.

if y in this part of items:

prediction = 1

else:

prediction = 0

else:

prediction = 0

##

```
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_Visit_k1620" + str(t) + ".txt", 'w+')
    for l in open("pairs_Visit.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 X[u_index].keys().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()
```

Visit Prediction Result:

I tried to find the best value of K and threshold.

Here is the important part of the results.

K	threshold	Accuracy
100	1	0.79270
180	1	0.85150
900	1	0.86845
1260	1	0.87515
1260	0.99	0.83740
1620	1	0.87510
1620	0.99	0.87235
1620	0.98	0.87470
1800	1	0.87505
1800	0.99	0.87465

Finally, I choose K=1260 and threshold=1 since it has the best accuracy.
Public Accuracy Score: 0.87515 Rank: 355/668
Private Accuracy Score: 0.87540 Rank: 356/668
(Simple Collaborative Filtering: 0.71465)

Analysis: I choose kNN for this problem. It usually takes a lot of time to calculate for kNN, especially when k is large. Maybe this method is not as good as one-class recommendation, which will take less time and can get a better result.

(Rating Prediction starts from next page)

Rating Prediction

Latent Factor Model:

Prediction = alpha + beta_user + beta_item + gamma_user * gamma_item

alpha is the average value of rating for all the training data.
beta_user is the user's rating bias between personal rating tendency and average rating value
beta_item is the item's rating bias between items received rating and average rating value
gamma_user and gamma_item are randomly initialized with dimension K.

```
users_items = {}
items_users = {}
#for datum in data:
for datum in train_set:
    u,i = datum['userID'],datum['businessID']
    if not users_items.has_key(u):
        users_items[u] = [(i,datum['rating'])]
    else:
        users_items[u].append((i,datum['rating']))
    if not items_users.has_key(i):
        items_users[i] = [(u,datum['rating'])]
    else:
        items_users[i].append((u,datum['rating']))
```

```
sum_rating = 0
count_rating = 0
for u,i in users_items.iteritems():
    for i_info in i:
        sum_rating += i_info[1]
        count_rating += 1
alpha = float(sum_rating)/count_rating
```

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

```python
# create gamma
gamma_u = {}
gamma_i = {}
K = 1000
num_users = K
num_items = K
for u in users_items.keys():
    gamma_u[u] = numpy.random.random((1,K))
for i in items_users.keys():
    gamma_i[i] = numpy.random.random((1,K))
```

```python
for i in users_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] * 0.000001
for i in items_users.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] * 0.000001
```

Method for updating parameters:

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

1. Find alpha, beta and determine K.

2. Initialize value of gamma_u and gamma_i randomly between [-0.000005,0.000005]

3. Fixed gamma_i, use closed form solution to update alpha and beta.

Use gradient descent to update gamma_u

4. Fixed gamma_u, use closed form solution to update alpha and beta.

Use gradient descent to update gamma_i

5. Repeat 3 and 4 until convergence

```python
# Update alpha
sum_for_alpha = 0
for u in users_items.keys():
    for i in users_items[u]:
        sum_for_alpha += i[1] - Betauser[u] - Betaitem[i[0]] - gammaU[u].dot(gammaI[i[0]].transpose())[0][0]
Alpha = float(sum_for_alpha) / len(train_set)
# Update beta_user
for u in users_items.keys():
    sum_for_betauser = 0
    count_item = 0
    for i in users_items[u]:
        count_item += 1
        sum_for_betauser += i[1] - Alpha - Betaitem[i[0]] - gammaU[u].dot(gammaI[i[0]].transpose())[0][0]
    Betauser[u] = float(sum_for_betauser) / (lam + count_item)
# Update beta_item
for i in items_users.keys():
    sum_for_betaitem = 0
    count_user = 0
    for u in items_users[i]:
        count_user += 1
        sum_for_betaitem += u[1] - Alpha - Betauser[u[0]] - gammaU[u[0]].dot(gammaI[i].transpose())[0][0]
    Betaitem[i] = float(sum_for_betaitem) / (lam + count_user)
return Alpha,Betauser,Betaitem
```

```python
def update_gamma_u(Alpha,Betauser,Betaitem,gammaU,gammaI,K,lam,rate):
    total = len(train_set)
    # Update gammaU --- Batch gradient descent
    for u in users_items.keys():
        update_flag = bool(random.random()<=0.5)
        if update_flag:
            sum_for_gammaU = 0
            for i in users_items[u]:
                sum_for_gammaU += i[1] - Alpha - Betauser[u] - Betaitem[i[0]] - gammaU[u].dot(gammaI[i[0]].transpose())[0][0]
                for count_gamma in range(K):
                    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,Betauser,Betaitem,gammaU,gammaI,K,lam,rate):
    total = len(train_set)
    # Update gammaI --- Batch gradient descent
    for i in items_users.keys():
        update_flag = bool(random.random()<=0.5)
        if update_flag:
            sum_for_gammaI = 0
            for u in items_users[i]:
                sum_for_gammaI += u[1] - Alpha - Betauser[u[0]] - Betaitem[i] - gammaU[u[0]].dot(gammaI[i].transpose())[0][0]
                for count_gamma in range(K):
                    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
```

Result:

I have tried different value of K and the learning rate of updating gamma. The best situation I have got is λ=4.65 and K=1. In this situation, the score on Kaggle is:

Private Score: 0.75728 Rank: 155/347

Public Score: 0.74522 Rank: 5/347

(Baseline solution: 0.80862)

Analysis:

Most of my time spend on this problem is to find the correct method to initialize gamma, optimize the value of lambda and find the appropriate method of K. However, I cannot get a better result with the increase value of K. Finally, my best situation is K=1, lambda=4.65 but my private score is very low. I think it is due to the fact that the result is a little bit overfitting on the seen part of test set so the RMSE for unseen part get a bad performance compared to seen part.