Homework 3

Kaggle Name: Noven_G

In [ ]:
```
import gzip
from collections import defaultdict
import numpy as np
from random import sample, randint
from sklearn.metrics import accuracy_score
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import os.path
import pprint
import random
```

In [2]:
```
def readGz(f):
  for l in gzip.open(f):
    yield eval(l)

data = list(readGz('train.json.gz'))
```

In [3]:
```
train = data[0:100000]

val = data[100000:]
```

1.Although we have built a validation set, it only consists of positive samples. For this task we also need examples of user/item pairs that weren't purchased. Build such a set by randomly sampling users and items until you have 100,000 non-purchased user/item pairs. This random sample combined with your 100,000 validation reviews now corresponds to the complete validation set for the purchase prediction task. Evaluate the performance (accuracy) of the baseline model on the validation set you have built

In [4]:
```
unique_reviewer = list(set([d['reviewerID'] for d in data]))
unique_business = list(set([d['itemID'] for d in data]))
```

In [5]:
```
visited_pairs = []
for datum in data:
    visited_pairs.append((datum['reviewerID'], datum['itemID']))
visited_pairs = set(visited_pairs)
len(visited_pairs)
```

Out[5]:
```
200000
```

In [6]:
```
non_visted_pairs = set()
counter = 100000
while counter:
    random_item_id = random.sample(unique_business, 1)[0]
    random_userID = random.sample(unique_reviewer, 1)[0]
    if ((random_userID, random_item_id) not in visited_pairs
        and (random_userID, random_item_id) not in non_visted_pairs):
            non_visted_pairs.add((random_userID, random_item_id))
            counter -= 1
```

In [7]:
```
sampled_dataset = [[d[0], d[1]] for d in non_visted_pairs]
y_sampled = [0]*len(sampled_dataset)
```

In [8]:
```
x_train = [[d['reviewerID'], d['itemID']] for d in train]
y_train = [1]*len(x_train)
```

In [9]:
```
val_data = [[d['reviewerID'], d['itemID']] for d in val]
y_validation = [1]*len(val_data)
```

In [10]:
```
val_data = val_data + sampled_dataset

x_val = val_data
y_val = y_validation + y_sampled
```

In [11]:
```
businessCount = defaultdict(int)
totalPurchases = 0
for l in x_train:
    user,business = l[0],l[1]
    businessCount[business] += 1
    totalPurchases += 1

mostPopular = [(businessCount[x], x) for x in businessCount]
mostPopular.sort()
mostPopular.reverse()

return1 = set()
count = 0
for ic, i in mostPopular:
    count += ic
    return1.add(i)

    if count > totalPurchases/2:
        break
```

In [12]:
```
y_val_predictions = []
for x in x_val:
    if x[1] in return1:
        y_val_predictions.append(1)
    else:
        y_val_predictions.append(0)
```

In [13]:
```
print("Performance (accuracy) of the baseline model on the validation
set is: ", str(accuracy_score(y_val_predictions, y_val)))
```

```
Performance (accuracy) of the baseline model on the validation set i
s:  0.62942
```

In [14]:
```
predictions = open("predictions_Purchase.txt", 'w')
for l in open("pairs_Purchase.txt"):
  if l.startswith("reviewerID"):
    #header
    predictions.write(l)
    continue
  u,i = l.strip().split('-')
  if i in return1:
    predictions.write(u + '-' + i + ",1\n")
  else:
    predictions.write(u + '-' + i + ",0\n")

predictions.close()
```

2.The existing `purchase prediction' baseline just returns True if the item in question` is popular,' using a threshold of the 50th percentile of popularity (totalPurchases/2). Assuming that the `non-purchased' test examples are a random sample of user-purchase pairs, is this particular threshold value the best? If not, see if you can nd a better one (and report its performance), or if so, explain why it is the best

The optimum threshold to get the best score in the validation set is 49% which is very close to 50% as in the baseline code. This is because
half the reviewer/item paris is negative samples that has never happened and half is positive as in the trainig data. The predictor will perform fairly better on the easy data than the hard ones, therefore, the best cut occurs around 50%.

In [15]:
```
businessCount = defaultdict(int)
totalPurchases = 0
for l in x_train:
    user,business = l[0],l[1]
    businessCount[business] += 1
    totalPurchases += 1

mostPopular = [(businessCount[x], x) for x in businessCount]
mostPopular.sort()
mostPopular.reverse()

def partition_at(threshold):
    mostPopular_business = set()
    count = 0
    for ic, i in mostPopular:
        count += ic
        mostPopular_business.add(i)
        if count > totalPurchases*((100.0 - threshold)/100.0):
            break
    return mostPopular_business
```

In [16]:
```
thresholds = np.arange(0, 100, 1)
threshold_accuracies = []

for t in thresholds:
    check_set = partition_at(t)
    y_validation_predictions = []
    for x in x_val:
        if x[1] in check_set:
            y_validation_predictions.append(1)
        else:
            y_validation_predictions.append(0)
    accuracy = accuracy_score(y_validation_predictions, y_val)
    threshold_accuracies.append(accuracy)
```

In [17]:
```
for i in range(len(threshold_accuracies)):
    print("Percentile: ", str(thresholds[i]), " Accuracy: ", str(threshold_accuracies[i]))

index_max_thresh_accu = threshold_accuracies.index(max(threshold_accuracies))
print("\nA better value of accuracy is ", str(max(threshold_accuracies)))
print("It occurs at ", str(thresholds[index_max_thresh_accu]), str(" percentile."))
```

```
Percentile:  0  Accuracy:  0.50229
Percentile:  1  Accuracy:  0.509265
Percentile:  2  Accuracy:  0.51715
Percentile:  3  Accuracy:  0.522355
Percentile:  4  Accuracy:  0.527005
Percentile:  5  Accuracy:  0.531705
```

```
Percentile:  6  Accuracy:  0.536635
Percentile:  7  Accuracy:  0.541355
Percentile:  8  Accuracy:  0.54643
Percentile:  9  Accuracy:  0.55062
Percentile:  10  Accuracy:  0.554465
Percentile:  11  Accuracy:  0.55843
Percentile:  12  Accuracy:  0.562065
Percentile:  13  Accuracy:  0.566105
Percentile:  14  Accuracy:  0.569565
Percentile:  15  Accuracy:  0.573125
Percentile:  16  Accuracy:  0.576165
Percentile:  17  Accuracy:  0.579455
Percentile:  18  Accuracy:  0.58318
Percentile:  19  Accuracy:  0.58688
Percentile:  20  Accuracy:  0.590245
Percentile:  21  Accuracy:  0.593415
Percentile:  22  Accuracy:  0.59584
Percentile:  23  Accuracy:  0.598
Percentile:  24  Accuracy:  0.600295
Percentile:  25  Accuracy:  0.602395
Percentile:  26  Accuracy:  0.604655
Percentile:  27  Accuracy:  0.606575
Percentile:  28  Accuracy:  0.608865
Percentile:  29  Accuracy:  0.6113
Percentile:  30  Accuracy:  0.613505
Percentile:  31  Accuracy:  0.61542
Percentile:  32  Accuracy:  0.617965
Percentile:  33  Accuracy:  0.619455
Percentile:  34  Accuracy:  0.62025
Percentile:  35  Accuracy:  0.621295
Percentile:  36  Accuracy:  0.62258
Percentile:  37  Accuracy:  0.623715
Percentile:  38  Accuracy:  0.62509
Percentile:  39  Accuracy:  0.625785
Percentile:  40  Accuracy:  0.627015
Percentile:  41  Accuracy:  0.628205
Percentile:  42  Accuracy:  0.62835
Percentile:  43  Accuracy:  0.62863
Percentile:  44  Accuracy:  0.62865
Percentile:  45  Accuracy:  0.628935
Percentile:  46  Accuracy:  0.62949
Percentile:  47  Accuracy:  0.62973
Percentile:  48  Accuracy:  0.629715
Percentile:  49  Accuracy:  0.629635
Percentile:  50  Accuracy:  0.62942
Percentile:  51  Accuracy:  0.628755
Percentile:  52  Accuracy:  0.628475
Percentile:  53  Accuracy:  0.62789
Percentile:  54  Accuracy:  0.62696
Percentile:  55  Accuracy:  0.62571
Percentile:  56  Accuracy:  0.624515
Percentile:  57  Accuracy:  0.623505
Percentile:  58  Accuracy:  0.62252
```

```
Percentile:  59  Accuracy:  0.62128
Percentile:  60  Accuracy:  0.61966
Percentile:  61  Accuracy:  0.617625
Percentile:  62  Accuracy:  0.61639
Percentile:  63  Accuracy:  0.614495
Percentile:  64  Accuracy:  0.61271
Percentile:  65  Accuracy:  0.61087
Percentile:  66  Accuracy:  0.609105
Percentile:  67  Accuracy:  0.606975
Percentile:  68  Accuracy:  0.604975
Percentile:  69  Accuracy:  0.60336
Percentile:  70  Accuracy:  0.601015
Percentile:  71  Accuracy:  0.59901
Percentile:  72  Accuracy:  0.59657
Percentile:  73  Accuracy:  0.59413
Percentile:  74  Accuracy:  0.59122
Percentile:  75  Accuracy:  0.588985
Percentile:  76  Accuracy:  0.58649
Percentile:  77  Accuracy:  0.58409
Percentile:  78  Accuracy:  0.581365
Percentile:  79  Accuracy:  0.578755
Percentile:  80  Accuracy:  0.575335
Percentile:  81  Accuracy:  0.572135
Percentile:  82  Accuracy:  0.568755
Percentile:  83  Accuracy:  0.5656
Percentile:  84  Accuracy:  0.562005
Percentile:  85  Accuracy:  0.55887
Percentile:  86  Accuracy:  0.555745
Percentile:  87  Accuracy:  0.55226
Percentile:  88  Accuracy:  0.54852
Percentile:  89  Accuracy:  0.544775
Percentile:  90  Accuracy:  0.541195
Percentile:  91  Accuracy:  0.53738
Percentile:  92  Accuracy:  0.5333
Percentile:  93  Accuracy:  0.52977
Percentile:  94  Accuracy:  0.52616
Percentile:  95  Accuracy:  0.522235
Percentile:  96  Accuracy:  0.51766
Percentile:  97  Accuracy:  0.513405
Percentile:  98  Accuracy:  0.509185
Percentile:  99  Accuracy:  0.50491

A better value of accuracy is  0.62973
It occurs at  47  percentile.
```

3.Users may tend to repeatedly purchase items of the same type. Build a baseline that returns `True' if a user has purchased an item of the same category before (at least one category in common), or zero otherwise

In [18]:
```python
## def flatten(items, seqtypes=(list, tuple)):
    ## for i, x in enumerate(items):
      ##  while i < len(items) and isinstance(items[i], seqtypes):
          ##  items[i:i+1] = items[i]
  ##  return items
## cat = [d['categories'] for d in data]

## fcat = flatten(cat[:])
## my_cat = list(set(fcat))


## reviewerIds = dict(zip(unique_reviewer, range(len(unique_reviewer))
))
## businessIds = dict(zip(unique_business, range(len(unique_business))
))
```

In [ ]:
```python

```

In [19]:
```python
userCategoriesVisited = defaultdict(set)
for d in train:
    uId = d['reviewerID']
    # reviewerIds[unique_reviewer]
    userCategoriesVisited[uId].add(d['categoryID'])


businessCategories = defaultdict(set)
for d in train:
    bId = d['itemID']
    businessCategories[bId].add(d['categoryID'])
```

In [20]:
```python
Popular_business = partition_at(40)
```

In [21]:
```python
def predict(uId, bId):
    if bId in Popular_business:
        cId = businessCategories[bId]
        for c in cId:
            if c in userCategoriesVisited[uId]:
                  return 1


    return 0
```

In [22]:
```python
y_train_predictions = []
for x in x_train:
    uid = x[0]
    bid = x[1]
    y_train_predictions.append(predict(uid, bid))
```

In [23]:
```
accu = accuracy_score(y_train_predictions,y_train)
print("Accuracy is ", str(accu))
```

```
Accuracy is  0.60003
```

In [24]:
```
y_validation_predictions = []
for x in x_val:
    uid = x[0]
    bid = x[1]
    y_validation_predictions.append(predict(uid, bid))
```

In [25]:
```
acc = accuracy_score(y_validation_predictions, y_val)
print("Accuracy is ", str(acc))
```

```
Accuracy is  0.629645
```

In [26]:
```
predictions = open("predictions_Purchase.txt", 'w')
for l in open("pairs_Purchase.txt"):
    if l.startswith("reviewerID"):
    #header
        predictions.write(l)
        continue
    u,i = l.strip().split('-')
    predictions.write(u + "-" + i + "," + str(int(predict(u, i))) + "\
n")
predictions.close()
```

Part_B

Q5. What is the performance of a trivial predictor on the validation set, and what is the value of alpha?

The performance of a trival predictor on the validation set is 1.222; Alpha is 4.23

In [27]:
```
def feat(d):
    return [1]

training_data = data[:100000]
x_train = [feat(d) for d in training_data]
y_train = [d['rating'] for d in training_data]

theta,residuals,rank,s = np.linalg.lstsq(x_train, y_train, rcond=None)
print("The value of alpha is ", str(theta[0]))
```

```
The value of alpha is  4.231999999999982
```

In [28]:
```
validation_data = data[100000:]
validation_dataset = [feat(d) for d in validation_data]
y_validation = [d['rating'] for d in validation_data]
y_validation_predictions = validation_dataset*theta

sum = 0.0
for i in range(len(y_validation)):
    sum += (y_validation_predictions[i] - y_validation[i])**2
MSE_validation = sum/len(y_validation)
```

In [29]:
```
print("MSE on validation set: ", str(MSE_validation[0]))
```

```
MSE on validation set:  1.222481119999119
```

Q6 Report the MSE on the validation set.

MSE for validation set without negative sample is: 1.28

In [30]:
```
## unique_reviewer = list(set([d['reviewerID'] for d in data]))
## unique_business = list(set([d['itemID'] for d in data]))
## reviewerIds = dict(zip(unique_reviewer, range(len(unique_reviewer))
))
## businessIds = dict(zip(unique_business, range(len(unique_business))
))
## num_reviewer = len(unique_reviewer)
## num_business = len(unique_business)
```

In [31]:
```
unique_reviewer = list(set([d['reviewerID'] for d in data]))
unique_business = list(set([d['itemID'] for d in data]))
reviewerIds = dict(zip(unique_reviewer, range(len(unique_reviewer))))
businessIds = dict(zip(unique_business, range(len(unique_business))))
```

In [32]:
```
ratings_by_users = defaultdict(dict)
ratings_for_businesses = defaultdict(dict)

for d in training_data:
    index_u = reviewerIds[d['reviewerID']]
    index_b = businessIds[d['itemID']]
    ratings_by_users[index_u][index_b] = d['rating']
    ratings_for_businesses[index_b][index_u] = d['rating']
```

In [33]:
```
alpha = theta[0]
betaU = [1.0] * len(reviewerIds)
betaI = [1.0] * len(businessIds)
```

In [34]:
```
def feature_for_visitedpairs(datum):
    feat = [1]
    feat.append(reviewerIds[datum['reviewerID']])
    feat.append(businessIds[datum['itemID']])
    return feat
```

In [35]:
```
def feature_for_nonvisitedpairs(datum):
    feat = [1]
    feat.append(reviewerIds[datum.split(",")[1]])
    feat.append(businessIds[datum.split(",")[0]])
    return feat
```

In [36]:
```
def update_alpha_beta(alpha, bU, bI, Ru, Rb, train_data, lam):
    a = 0.0
    for uid in list(range(len(bU))):
        for bid, rating in Ru[uid].items():
            a += (rating - (bU[uid] + bI[bid]))/len(train_data)
    alpha = a

    bU_new = [0.0]*len(bU)
    for uid in list(range(len(bU))):
        for bid, rating in Ru[uid].items():
                bU_new[uid] += (rating - (alpha + bI[bid]))/(lam + len
(Ru[uid]))
    bU = bU_new

    bI_new = [0.0]*len(bI)
    for bid in list(range(len(bI))):
        for uid, rating in Rb[bid].items():
            bI_new[bid] += (rating - (alpha + bU[uid]))/(lam + len(Rb[
bid]))
    bI = bI_new
    return a, bU_new, bI_new
```

In [37]:
```
def find_cost(alpha, bU, bI, Ru, lam):
    bUsq = np.sum(np.square(bU))
    bIsq = np.sum(np.square(bI))

    term1 = 0.0
    for uid in list(range(len(bU))):
        for bid, r in Ru[uid].items():
            term1 += (alpha + bU[uid] + bI[bid] - r)**2
    return term1 + lam*(bUsq + bIsq)
```

In [38]:
```
def RMSE_after_prediction(x, y, a, bU, bI):
    y_pred = []
    for i in range(len(x)):
        r = a*x[i][0] + bU[x[i][1]] + bI[x[i][2]]
        y_pred.append(r)
        term = 0.0
    for i in range(len(y)):
        term += (y_pred[i] - y[i])**2
    RMSE = np.sqrt(term/len(y))
    return RMSE
```

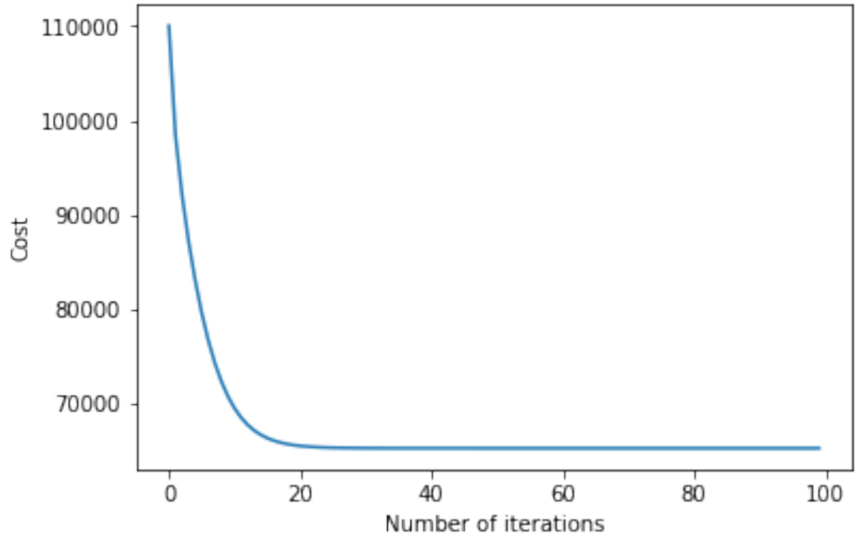
In [39]:
```
validation_data = data[100000:]
validation_dataset = [feature_for_visitedpairs(d) for d in validation_
data]
y_validation = [d['rating'] for d in validation_data]
```

In [40]:
```
## training_data = data[:100000]
## x_train = [feature(d) for d in training_data]
## y_train = [d['rating'] for d in training_data]
```

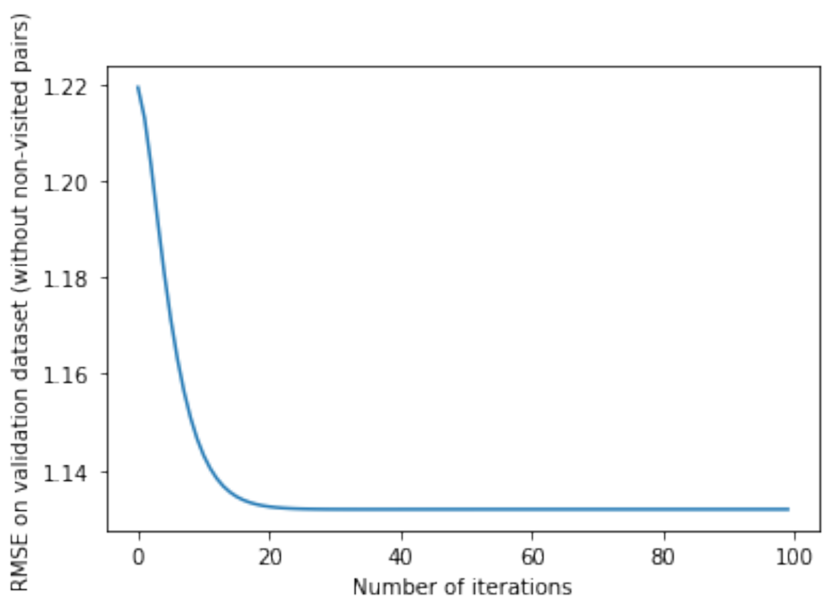
In [41]:
```
costs = []
RMSEs_validation = []
for i in range(100):
    alpha, betaU, betaI = update_alpha_beta(alpha, betaU, betaI, ratin
gs_by_users, ratings_for_businesses, training_data, 1)
    costs.append(find_cost(alpha, betaU, betaI, ratings_by_users, 1))
    RMSE = RMSE_after_prediction(validation_dataset,
                                y_validation, alpha,
                                betaU, betaI)
    RMSEs_validation.append(RMSE)
```

In [42]:
```
# len(betaU)
```

In [43]:
```
plt.plot(list(range(100)), costs)
plt.xlabel("Number of iterations")
plt.ylabel("Cost")
plt.show()
```



In [44]:
```
plt.plot(list(range(100)), RMSEs_validation)
plt.xlabel("Number of iterations")
plt.ylabel("RMSE on validation dataset (without non-visited pairs)")
plt.show()
```



In [45]:
```
#print(costs[-50:])

for i in range(-100, 0):
    print(costs[i])
```

```
110055.68276790378
98453.27679568082
92138.35839313632
```

87212.88059707338
83098.68129620174
79627.30890941042
76719.81605930525
74315.6322593436
72354.9582875464
70776.86537060371
69521.67499796652
68533.68477063197
67763.04696483236
67166.65308756748
66708.2298516524
66357.919178599
66091.5771488285
65889.9622401072
65737.92259864071
65623.64508224317
65537.99608579386
65473.96322186162
65426.19466320092
65390.626587894236
65364.18656167886
65344.56032157947
65330.01026849555
65319.235384489766
65311.26387286031
65305.37135774775
65301.01886592849
65297.80600414744
65295.43573555063
65293.68796334832
65292.39977198857
65291.450681786955
65290.751666694516
65290.23698885067
65289.858136521725
65289.579329431304
65289.374189998285
65289.22328048262
65289.11228237338
65289.030651534405
65288.970625464135
65288.926490898855
65288.894043737084
65288.870190990405
65288.85265750387
65288.839769962135
65288.83029784066
65288.823336343
65288.818220243396
65288.81446049655
65288.81169760978
65288.809667334834

```
65288.80817545154
65288.80707921308
65288.80627371211
65288.80568185209
65288.80524697638
65288.804927449106
65288.804692681195
65288.80452018702
65288.80439345245
65288.80430033867
65288.804231924245
65288.804181661486
65288.80414473287
65288.80411759982
65288.80409766627
65288.80408302188
65288.804072261104
65288.80406435569
65288.80405854891
65288.80405428191
65288.804051147425
65288.80404884391
65288.80404715131
65288.80404590851
65288.804044995675
65288.80404432443
65288.80404383108
65288.80404346886
65288.804043203054
65288.80404300817
65288.80404286372
65288.80404275877
65288.804042680444
65288.80404262347
65288.80404258308
65288.804042552314
65288.80404252877
65288.804042512005
65288.804042499796
65288.80404249125
65288.80404248462
65288.8040424801
65288.804042474396
65288.80404247324
```

In [46]:
```
min(costs)
```

Out[46]:
```
65288.80404247324
```

the value tends to converge at 65288.804.

In [47]:
```
validation_data = data[100000:]
validation_dataset = [feature_for_visitedpairs(d) for d in validation_
data]
y_validation = [d['rating'] for d in validation_data]
```

In [48]:
```
y_validation_predictions = []
for i in range(len(validation_dataset)):
    rating = alpha*validation_dataset[i][0]
    rating += betaU[validation_dataset[i][1]]
    rating += betaI[validation_dataset[i][2]]
    y_validation_predictions.append(rating)

sum = 0.0
for i in range(len(y_validation)):
    sum += (y_validation_predictions[i] - y_validation[i])**2
MSE_val_set = sum/len(y_validation)
```

In [49]:
```
print("MSE for validation set without negative sample is: ", str(MSE_v
al_set))
```

```
MSE for validation set without negative sample is:  1.28111878381150
01
```

Q7. Report the user and item IDs that have the largest and smallest values of β.

In [50]:
```
x=reviewerIds.items()
xi=businessIds.items()
```

In [51]:
```
print("User with the largest beta:", str([v[0] for i, v in enumerate(x
) if v[1] == betaU.index((max(betaU)))]))
```

```
User with the largest beta: ['U495776285']
```

In [52]:
```
print("Item with the largest beta:", str([v[0] for i, v in enumerate(x
i) if v[1] == betaI.index((max(betaI)))]))
```

```
Item with the largest beta: ['I809804570']
```

In [53]:
```
print("User with the smallest beta:", str([v[0] for i, v in enumerate(
x) if v[1] == betaU.index((min(betaU)))]))
```

```
User with the smallest beta: ['U204516481']
```

In [54]:
```
print("Item with the smallest beta:", str([v[0] for i, v in enumerate(
xi) if v[1] == betaI.index((min(betaI)))]))
```

```
Item with the smallest beta: ['I511389419']
```

Q8. Find a better value of λ using your validation set. Report the value you chose, its MSE, and upload your solution to Kaggle by running it on the test data.

In [55]:
```
def MSE_for_prediction(x, y, a, bU, bI):
    y_pred = []
    for i in range(len(x)):
        r = a*x[i][0] + bU[x[i][1]] + bI[x[i][2]]
        y_pred.append(r)
    Sum = 0.0
    for i in range(len(y)):
        Sum += (y_pred[i] - y[i])**2
    MSE = Sum/len(y)
    return MSE
```

In [56]:
```
lambdas = [0.1, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 20.
0, 50.0, 100.0, 1000.0]
```

In [57]:
```
validation_data = data[100000:]
validation_dataset = [feature_for_visitedpairs(d) for d in validation_
data]
y_validation = [d['rating'] for d in validation_data]
```

In [58]:
```
MSEs_validation=[]
for lam in lambdas:
    alpha = theta[0]
    betaU = [1.0] * len(reviewerIds)
    betaI = [1.0] * len(businessIds)
    for i in range(100):
        alpha, betaU, betaI = update_alpha_beta(alpha, betaU, betaI, r
atings_by_users, ratings_for_businesses, training_data, lam)
    MSE = MSE_for_prediction(validation_dataset, y_validation, alpha,
betaU, betaI)
    print("Lambda: ", str(lam), "MSE: ", str(MSE))
    MSEs_validation.append(MSE)
```

```
Lambda:  0.1 MSE:  1.753980328249047
Lambda:  1.0 MSE:  1.2811187838115001
Lambda:  2.0 MSE:  1.1894792370143021
Lambda:  3.0 MSE:  1.1583185109618455
Lambda:  4.0 MSE:  1.145389600638842
Lambda:  5.0 MSE:  1.1398956065799253
Lambda:  6.0 MSE:  1.1379239574804334
Lambda:  7.0 MSE:  1.1377680309690588
Lambda:  8.0 MSE:  1.138591933828355
Lambda:  9.0 MSE:  1.1399528829961025
Lambda:  10.0 MSE:  1.1416030979547032
Lambda:  20.0 MSE:  1.1588274305831505
Lambda:  50.0 MSE:  1.1848301598317852
Lambda:  100.0 MSE:  1.199824808665946
Lambda:  1000.0 MSE:  1.2196224622340652
```

In [59]:
```
print("The lambda for the smallest MSE of validation set is ", str(7.0
))
```

```
The lambda for the smallest MSE of validation set is  7.0
```

In [60]:
```
print("The minimum MSE of validation set is ", str(min(MSEs_validation
)))
```

```
The minimum MSE of validation set is  1.1377680309690588
```

In [61]:
```
### Training to optimize model
```

In [62]:
```
training = data[:100000]
train_set = [feature_for_visitedpairs(d) for d in training]
y_training = [d['rating'] for d in training]
```

In [63]:
```
bestlam = 7.0
alpha = theta[0]
betaU = [1.0] * len(reviewerIds)
betaI = [1.0] * len(businessIds)

for i in range(100):
    alpha, betaU, betaI = update_alpha_beta(alpha, betaU, betaI, ratin
gs_by_users, ratings_for_businesses, training_data, bestlam)
```

In [64]:
```
def pred_rating(uid, bid, a, bU, bI, ru, rb):
    rating = 0.0
    if (uid not in reviewerIds and bid in businessIds):
        for u in bU:
            rating += a + u + bI[businessIds[bid]]
        rating = rating/len(bU)
    elif (uid in reviewerIds and bid not in businessIds):
        for i in bI:
            rating += a + bU[reviewerIds[uid]] + i
        rating = rating/len(bI)
    elif (uid not in reviewerIds and bid not in businessIds):
        rating = a
    elif (uid in reviewerIds and bid in businessIds):
        rating += a + bU[reviewerIds[uid]] + bI[businessIds[bid]]
    return rating
```

In [65]:
```
predictions = open("predictions_Rating.txt", 'w')
y_test_ratings_predictions = []
for l in open("pairs_Rating.txt"):
    if l.startswith("reviewerID"):
    #header
        predictions.write(l)
        continue
    uid, bid = l.strip().split('-')
    Rating = pred_rating(uid, bid, alpha, betaU, betaI, ratings_by_use
rs, ratings_for_businesses)
    predictions.write(uid + '-' + bid + ',' + str(Rating) + '\n')
    y_test_ratings_predictions.append(Rating)

predictions.close()
```