Similarity and Machine learning-based Recommender system for Jewelry Items

dataset:amazon review dataset of beauty products

In [2]:

```
import gzip
from collections import defaultdict
import random
import numpy
import scipy.optimize
```

In [3]:

```
path="amazon_reviews_us_Jewelry_v1_00.tsv.gz"
```

In [4]:

```
f=gzip.open(path, 'rt', encoding="utf-8")
```

In [5]:

```
header=f.readline()
header=header.strip().split('\t')
```

In [7]:

```
%pprint
```

```
Pretty printing has been turned OFF
```

In [8]:

```
header
```

Out[8]:

```
['marketplace', 'customer_id', 'review_id', 'product_id', 'product_parent', 'product_title',
'product_category', 'star_rating', 'helpful_votes', 'total_votes', 'vine', 'verified_purchase', 'r
eview_headline', 'review_body', 'review_date']
```

our goal for this task is going be to make recommendations of products based on users budget histories. So to do so, we're ignoring anything like ratings or reviews. We're just looking at this collection of reviewer IDs and product IDs. So which item IDs were consumed by which users or user IDs. So out of our data set, we only care about those two fields, reviewer ID and product ID.

Explore the Dataset(s)

In [9]:

```
dataset=[]
for line in f:
    field=line.strip().split('\t')
    d=dict(zip(header, field))
    d['star_rating']=int(d['star_rating'])
    d['helpful_votes']=int(d['helpful_votes'])
    d['total_votes']=int(d['total_votes'])
    dataset.append(d)
```

build these data structures, that is represent for each users, for each user, which items did they consume, and for each items, which users consumed that item. these two sets, users per item and items per user

In [10]:

```
userspe ritem=defaultdict(set)
itemsperuser=defaultdict(set)
```

In [11]:

```
itemNames={}
# build this dictionary of item names, just so we can actually interpret the results we produce a
little bit later. Rather than having item IDs,
#we'll actually be able to say, what was the item being recommended.
```

now we go through our dataset, and we extract the user ID and the item ID for each instance. And then we just add the user to the corresponding user set, and add the item to the corresponding item set.

In [12]:

```
for d in dataset:
    user, item=d['customer_id'], d['product_id']
    userspe ritem[item].add(user)  #which users purchase each item
    itemsperuser[user].add(item)  # which items were purchased by each user.
    itemNames[item]=d['product_title']
```

Jaccard similarity

we want to recommend items similar to a candidate item based on the Jaccard similarity

In [13]:

```
def jaccard(set1, set2):
    numer=len(set1.intersection(set2))
    denom=len(set1.union(set2))
    return numer/denom
```

Build Recommender System

1.simple similarity base recommender system based on the Jaccard similarity

query item, recommend other items that are similar to the query.

In [14]:

```
def mostSimilar(i): #given some input item i
    similarities=[]
    users=userspe ritem[i] #find the set of users who purchased i
    for i2 in userspe ritem: #iterate through all other items other than i to figure out which ones
are most similar.
        if i2 == i:continue
        sim=jaccard(users, userspe ritem[i2]) #Every other item, compute the Jaccard similarity with
i
        similarities.append((sim, i2))
    similarities.sort(reverse=True) #We'll sort all those Jaccard similarities in decreasing order
of similarity
    return similarities[:10] #return the most similar one or several of the most similar ones.
```

In [15]:

```
dataset[2]
```

Out[15]:

```
{'marketplace': 'US', 'customer_id': '27541121', 'review_id': 'R3N5JE5Y4T6W5M', 'product_id':
'B00M2L6KEY', 'product_parent': '697845240', 'product_title': 'Sterling Silver Circle "Friends For
'B00M2L6KFY', 'product_parent': '697845240', 'product_title': 'Sterling Silver Circle "Friends For
ever" Infinity Symbol Pendant Necklace, 18"', 'product_category': 'Jewelry', 'star_rating': 5, 'he
lpful_votes': 0, 'total_votes': 0, 'vine': 'N', 'verified_purchase': 'Y', 'review_headline':
"Exactly as pictured and my daughter's friend loved it! It came packaged in a nice box ...", 'revi
ew_body': "Exactly as pictured and my daughter's friend loved it!  It came packaged in a nice box
and I thought it looked more expensive than it was.  Very happy with this purchase.",
'review_date': '2015-08-31'}
```

In [16]:

```
query=dataset[2]['product_id']
```

In [17]:

```
query
```

Out[17]:

```
'B00M2L6KFY'
```

So this is the ranked list of the top ten most similar items, according to their Jaccard similarity with the query item.

In [18]:

```
mostSimilar(query) #outputs a ranked list of Jaccard similarities followed by item IDs.
```

Out[18]:

```
[(0.008130081300813009, 'B00B1VAY2Q'), (0.0, 'B01HT540FY'), (0.0, 'B014HZVFZW'), (0.0,
'B014HVRNX4'), (0.0, 'B014GDUGKK'), (0.0, 'B014GC4JMM'), (0.0, 'B014GC4JJU'), (0.0, 'B0149CZ3BU'),
(0.0, 'B0147FSITI'), (0.0, 'B0146G25KG')]
```

In [19]:

```
itemNames[query] #items that are considered to be most similar to the query in terms of their
Jaccard similarity
```

Out[19]:

```
'Sterling Silver Circle "Friends Forever" Infinity Symbol Pendant Necklace, 18"'
```

In [20]:

```
[itemNames[x[1]] for x in mostSimilar(query)]
```

Out[20]:

```
['925 Sterling Silver Infinity Symbol Wedding Band Ring', '18K Gold over Sterling Silver .8mm Thin
Italian Box Chain Necklace All Sizes 14" - 40"', 'WigsPedia Roll on Glass Beaded Bracelet - Nepal
Glass Beaded Bangles SET of (100)', 'Square Earrings 14K White Gold Finish Lab Diamonds Pave Set 9
MM Unisex Pierced', 'Sea Glass Drop Earrings', 'Titanium Ring Wedding Band, Engagement Ring with R
eal Wood Inlay, 8mm Comfort Fit Sizes 7 to 13', 'Titanium Ring Wedding Band, Engagement Ring with
Real Wood Inlay, 8mm Comfort Fit Sizes 7 to 13', 'Seattle Seahawks Silver Tone Multi Charm Toggle
Bracelet Sports Football', 'Princess Crown Dangle Bead for Snake Chain Charm Bracelet', 'Modern
Fantasy Unique Designed Axis Embleshed Antiqued Style Braided Adjustable Length Leather Wrap Brace
let']
```

1.1) Efficient similarity based recommender system based on the Jaccard similarity

Unfortunately, its implementation is not very efficient. So what's really slow is that every time we make a recommendation, we have to iterate through all of the items in the dataset. So potentially iterating over millions of items, computing Jaccard similarities for all of them, just to make a single recommendation.

It's going to be sufficient to iterate over those items that were purchased by at least one of the users who purchased item i. That's going to be the complete set of items that could possibly have non-zero Jaccard similarity.

In [21]:

```
def mostSimilarFast(i):
    similarities=[]
    users=usersperitem[i] #find the set of users who purchased i
    candidateitems=set() #build the candidate set of possible items, items j
    for u in users:  #iterate over all of those users,
        candidateitems=candidateitems.union(itemsperuser[u]) #taking the union of all of the items
those users consumed.
    for i2 in candidateitems:
        if i2 == i: continue
        sim=jaccard(users, usersperitem[i2]) #compute the Jaccard similarity with i
        similarities.append((sim, i2))
    similarities.sort(reverse=True) #sort by that Jaccard similarity, and we'll return the most si
milar items
    return similarities[:10]
```

the point is that the union of the sets of all the items purchased by users who purchased i, could not possibly be larger than the full set of items. And in practice, will usually be much, much smaller.

In [22]:

```
mostSimilarFast(query)
```

Out[22]:

```
[(0.008130081300813009, 'B00B1VAY2Q')]
```

this returns exactly the same recommendations, but it does so much more quickly than the previous implementation.

recommender system so that it takes a user as an input and makes recommendations of items that are similar to things that user liked

In [23]:

```
def mostSimilarFastUser(u):
    similarities=[]
    items=itemsperuser[u] #find the set of items purchased by u
    candidateusers=set() #build the candidate set of possible users, users j
    for i in items:  #iterate over all of those items,
        candidateusers=candidateusers.union(usersperitem[i]) #taking the union of all of the users
consumed items.
    for i2 in candidateusers:
        if i2 == u: continue
        sim=jaccard(items, itemsperuser[i2]) #compute the Jaccard similarity with u
        similarities.append((sim, i2))
    similarities.sort(reverse=True) #sort by that Jaccard similarity, and we'll return the most si
milar items
    return similarities[:10]
```

In [24]:

```
query=dataset[2]['customer_id']
```

a user u's rating for an item i, as a weighted combination of all of the ratings of previous items. So we have some vector that says, how did the user rate every single other item that have rated j.

store the individual ratings not just the user IDs and the item IDs we're going to add these two utility data structures, one which stores all of the reviews for a given user and all the reviews for given item, which is the list for each user and a list for each item.

In [25]:

```
reviewsperuser=defaultdict(list) #list of reviews per user per item
reviewsperitem=defaultdict(list)
```

So iterate through our data set, extract the user and item IDs, and append to the corresponding reviews to the user and item lists.

In [26]:

```
for d in dataset:
    user, item=d['customer_id'], d['product_id']
    reviewsperuser[user].append(d)
    reviewsperitem[item].append(d)
```

In [27]:

```
ratingmean=sum([d['star_rating'] for d in dataset])/len(dataset)
```

In [28]:

```
ratingmean #mean rating is used as baseline for comparison
```

Out[28]:

```
4.144090266004357
```

function that predicts the rating given a particular user and a particular item, what rating will that user give to that item?

In [29]:

```
def predictrating(user, item):
    ratings=[] #list of ratings that the user has given the previous items
    similarities=[] # list of Jaccard similarities we're going to use to compute a weighted
average of those ratings
    for d in reviewsperuser[user]: #iterate through all of the items that the user has consumed
        i2=d['product_id']
        if i2 == item: continue
        ratings.append(d['star_rating']) #take star rating that user gave to the previous item,
that's our list of ratings
        similarities.append(jaccard(usersperitem[item], usersperitem[i2])) #take the Jaccard
similarity,that's our list of
                                                                                       #similarities that we'l
e to weight the ratings.
    if(sum(similarities) > 0): #If the item is equal to the query item, we'll skip that one and
we'll continue
        weightedrating=[(x*y) for x,y in zip(ratings, similarities)] #what is the rating times the
Jaccard similarity for each of the
                                                                                  #corresponding elements in
those two lists
        return sum(weightedrating) / sum(similarities)
    else:
        return ratingmean #f the user has not rated any similar items, then this weighted average
won't make any sense
```

In [30]:

```
dataset[1]
```

Out[30]:

```
{'marketplace': 'US', 'customer_id': '11262325', 'review_id': 'R2N0QQ6R4T7YRY', 'product_id':
'B00W5T1H9W', 'product_parent': '26030170', 'product_title': '925 Sterling Silver Finish 6ct Simul
ated Diamond Stud Set', 'product_category': 'Jewelry', 'star_rating': 5, 'helpful_votes': 0,
'total_votes': 0, 'vine': 'N', 'verified_purchase': 'N', 'review_headline': 'Great product.',
'review_body': "Great product.. I got this set for my mother, as she is allergic to bijoux (probab
ly to the nickel in some products, I don't know for sure, I just know that her ears get swallowed,
red and itchy unless is silver or gold).<br />And I'm glad that this is sterling silver and still
at a affordable price. She never got an allergic reaction, and now she wears them every day.", 're
view_date': '2015-08-31'}
```

In [31]:

```
u, i=dataset[1]['customer_id'], dataset[1]['product_id']
```

In [32]:

```
predictrating(u,i)
```

Out[32]:

```
4.144090266004357
```

mean squared error

In [33]:

```
def MSE(predictions, labels):
    differences=[(x-y)**2 for x,y in zip(predictions, labels)]
    return sum(differences)/ len(differences)
```

In [34]:

```
alwaysPredictMean=[ratingmean for d in dataset]
```

In [35]:

```
cfPredictions=[predictrating(d['customer_id'], d['product_id']) for d in dataset]
```

In [36]:

```
labels=[d['star_rating'] for d in dataset]
```

In [37]:

```
MSE(alwaysPredictMean, labels)
```

Out[37]:

```
1.6992712804660657
```

2)Machine learning based recommender system

Recommendations based on models are called latent-factor models.

develop the simple bias-only version of the model which takes just an offset a user bias and an item bias in order to make a prediction. We'll also use a library function for gradient descent, in order to make this a little bit easier

Implementation using gradient descent

using an offset Alpha, a user bias and item bias term, as well as regularizer for those biases,

In [40]:

```
N=len(dataset)
nUsers=len(reviewsperuser)
nItems=len(reviewsperitem)
users=list(reviewsperuser.keys())
items=list(reviewsperitem.keys())
```

Starting it at a good initial value might help us find a good solution later on. But all of these values are going to change and update as we actually run gradient descent.

In [41]:

```
alpha=ratingmean #variables that we're going to try and optimize
```

In [42]:

```
userBiases=defaultdict(float) # each user Beta_u which will store in this vector of user biases, w
as dictionary of user biases
itemBiases=defaultdict(float) # Beta_i, the bias for each item, which will store in this
dictionary on item biases
```

f(u,i)=alpha+beta_u+beta_i there's prediction function which takes if the user and an item knows about all the values of our offset user item bias terms. It just makes a prediction by adding those three together. So the offset term plus user bias for that user, plus the item bias for that item.

In [43]:

```
def prediction(user, item):
    return alpha + userBiases[user] + itemBiases[item]
```

the first complex part of our function. This is some unpack function. This is something we just need for the specific gradient descent library we're going to try and use. So what that library expects is a single vector of all of our parameters. to take our parameters which are Alpha user bias for each user and the item bias for each item and just convert that to a vector. this is a function that takes a vector of all of those parameters and then converts it back to Alpha Beta_u and Beta_i. it takes some vector Theta.

In [44]:

```
#parses theta(flat parameter vector to a actual parameter for the latent fator model)
def unpack(theta):
    global alpha
    global userBiases
    global itemBiases
    alpha=theta[0] #Alpha is going to be the first position in that vector Theta[0].
    userBiases=dict(zip(users, theta[1:nUsers+1])) #user biases are going to be the next set of pos
itions going from all
                                                    #of our users
    itemBiases=dict(zip(items, theta[1+nUsers:])) #The item biases are going to be the next
position starting from position one
                                                    #plus the number of users onwards to the rest o
that vector
```

In [45]:

```
def cost(theta, labels, lamb): #take the current estimate of Theta the labels value Lambda, which
is going to be our
                               #regularization strength
    unpack(theta) # extract the values of Alpha, Beta_u, and Beta_i
    predictions=[prediction(d['customer_id'], d['product_id']) for d in dataset] #use those values
to make predictions
                                                                                  #or the data points
data set or training set
    cost=MSE(predictions, labels)
    print("MSE::"+str(cost)) #compute our mean squared error
    for u in userBiases:
        cost+=lamb*userBiases[u]**2 #Lambda times each user biases squared and Lambda times for
each
                                   #item biases squared, and we'll return that total cost.

    for i in itemBiases:
        cost+=lamb*itemBiases[i]**2
    return cost
```

In [46]:

```
def derivative(theta, labels, lamb):
    unpack(theta) #unpack vector Theta current estimates for Alpha, Beta_u for each user and Beta_
i for each item
    N=len(dataset)
    dalpha=0 #derivative for offset term Alpha
    dUserBiases=defaultdict(float) #derivative for each user
    dItemBiases=defaultdict(float) #derivative for each item
    for d in dataset: #iterate through the entire dataset just once and update each of the
relevant derivatives as we go.
```

```
                      #So if we look at a particular point in the data set, u, i,
        u,i=d['customer_id'],d['product_id']
        pred=prediction(u,i)
        diff=pred-d['star_rating']
        dalpha+= 2/N*diff
                          #Each time we see a user u and an item i, it will update Alpha, will upda
te Beta_u,
                           #derivative of Beta_u, more precisely, and the derivative of Beta_i
        dUserBiases[u] += 2/N*diff
        dItemBiases[i] += 2/N*diff
    for u in userBiases:
        dUserBiases[u]+=2*lamb*userBiases[u] #encode a derivatives of our regularizers for each use
r and for each item
    for i in itemBiases:
        dItemBiases[i]+=2*lamb*itemBiases[i]
    dtheta=[dalpha] + [dUserBiases[u] for u in users] + [dItemBiases[i] for i in items]
    return numpy.array(dtheta)

     #convert those values back to a vector. So we'll take the derivative of Alpha, the derivative
of our
    #user bias for all users, and the derivative of our item biases for all items will return that
vector of derivatives.
```

In [47]:

```
MSE(alwaysPredictMean, labels)
```

Out[47]:

```
1.6992712804660657
```

gradient descent library called LBFGS, which is available as part of SciPy optimize fmin LBFGS a cost function we implemented that says, given some estimates of our parameters Alpha, Beta_u, and Beta_i, what is the total cost? So the MSE plus the regularizer it takes is the initial values for all of our parameters. So that's going to be the rating mean plus a bunch of zeros of the offset times for each user, and each item we're going to be initialized to zero there's functions also needed to see what are the labels and what is our regularization constant Lambda. We're going to pass in those two values to each of those functions every time they're called. That's a very general purpose interface to gradient descent

Analyze Your Results

In [48]:

```
scipy.optimize.fmin_l_bfgs_b(cost, [ratingmean]+[0.0]*(nUsers+nItems),
                             derivative, args=(labels, 0.001))
#It's going to print out the mean squared error during each iteration, because we wrote that code
inside the cost function to print the MSE
```

```
MSE::1.6992712804660657
MSE::1.693781606688868
MSE::1.7472332111212827
MSE::1.6935245788016036
MSE::1.6863042302701934
MSE::1.6861931203536156
MSE::1.6857522112428713
MSE::1.6857339553610422
MSE::1.685700019558462
```

Out[48]:

```
(array([ 4.14423463e+00, -2.40421992e-03,  4.80465535e-04, ...,
        4.82487643e-04, -8.13389846e-05, -8.13389846e-05]), 1.6921680739149174, {'grad': array([-8
.23065516e-06,  5.97531682e-09, -1.78511163e-09, ...,
       -2.12859045e-09,  3.22129723e-10,  3.22129723e-10]), 'task': b'CONVERGENCE: NORM_OF_PROJECTE
D_GRADIENT_<=_PGTOL', 'funcalls': 9, 'nit': 7, 'warnflag': 0})
```

presented was a general purpose gradient descent algorithm. It simply requires that we provide this cost function, which is really our f of x and derivative which is really just f' of x or the derivative of f of x, and you can relatively straightforwardly apply this to other gradient descent problems

Implementation of a fully-fledged latent factor model

In [49]:

```
alpha=ratingmean #offset term, Alpha
```

In [50]:

```
userBiases=defaultdict(float) #a bias terms for each user
itemBiases=defaultdict(float) #a bias terms for each item
```

In [51]:

```
#latent factors so we have additional parameters corresponding to user Gamma for each user and the
item Gamma for each item
userGamma={}
itemGamma={}
```

In [52]:

```
k=2 #number of latent factor dimentionality of gamma
```

In [53]:

```
for u in reviewsperuser:
    userGamma[u]=[random.random()* 0.1 - 0.05 for k in range(k)]
```

In [54]:

```
for i in reviewsperitem:
    itemGamma[i]=[random.random()* 0.1 - 0.05 for k in range(k)]
```

In [55]:

```
def unpack(theta):
    global alpha #extract from that contiguous vector the offset parameter
    global userBiases
    global itemBiases
    global userGamma #the preference vector to each user
    global itemGamma #the preference vector to each item.
    index=0
    alpha=theta[index]
    index+=1
    userBiases=dict(zip(users, theta[index:index+nUsers])) #iterate through all of the positions in
Theta
    index+= nUsers
    itemBiases=dict(zip(items, theta[index:index+nItems]))
    index+= nItems
    #iterate through all users in all dimensions and extract the user preference for that
preference dimension and then
    #the item preference for that preference dimension.
    for u in users:
        userGamma[u]=theta[index:index+k]
        index+=k
    for i in items:
        itemGamma[i]=theta[index:index+k]
        index+=k
```

In [56]:

```
def inner(x, y):
```

```
def inner(x, y):
    return sum([a*b for a,b in zip(x,y)])
```

In [57]:

```
def prediction(user, item):
    return alpha+userBiases[user] + itemBiases[item] +inner(userGamma[user],itemGamma[item])
```

In [58]:

```
def cost(theta, labels, lamb): #take the current estimate of Theta the labels value Lambda, which
is going to be our
                               #regularization strength
    unpack(theta) # extract the values of Alpha, Beta_u, and Beta_i
    predictions=[prediction(d['customer_id'], d['product_id']) for d in dataset] #use those values
to make predictions
                                                                                #or the data point
data set or training set
    cost=MSE(predictions, labels)
    print("MSE::"+str(cost)) #compute our mean squared error
    for u in users:
        cost+=lamb*userBiases[u]**2 #Lambda times each user biases squared and Lambda times for
each
                                    #item biases squared, and we'll return that total cost.
        for k in range(k):
            cost +=lamb*userGamma[u][k]**2

    for i in items:
        cost+=lamb*itemBiases[i]**2
        for k in range(k):
            cost+=lamb*itemGamma[i][k]**2
    return cost
```

In [ ]: