Recommended Sistem

In this project i use a dataset about 904765 Musical Instruments that had been sold in Amazon, and i'm going to design a Similarity-Based Recommender based on the users who purchased the products and the items that had been purchased.

The dataset is able [here (https://s3.amazonaws.com/amazon-reviews-pds/tsv/index.txt)](https://s3.amazonaws.com/amazon-reviews-pds/tsv/index.txt) downloading the document named: [https://s3.amazonaws.com/amazon-reviews-pds/tsv/amazon_reviews_us_Musical_Instruments_v1_00.tsv.gz (https://s3.amazonaws.com/amazon-reviews-pds/tsv/amazon_reviews_us_Musical_Instruments_v1_00.tsv.gz)](https://s3.amazonaws.com/amazon-reviews-pds/tsv/amazon_reviews_us_Musical_Instruments_v1_00.tsv.gz)

Useful Libraries

In [2]:
```
import gzip
from collections import defaultdict
import random
import numpy as np
import scipy.optimize
```

Reading de data

In [3]:
```
path = "amazon_reviews_us_Musical_Instruments_v1_00.tsv.gz"
```

In [4]:
```
file = gzip.open(path, 'rt', encoding="utf-8")
```

In [5]:
```
header = file.readline()
header = header.strip().split('\t')
```

Now, let's see the content of the dataset

In [6]:
```
header
```

Out[6]:
```
['marketplace',
 'customer_id',
 'review_id',
 'product_id',
 'product_parent',
 'product_title',
 'product_category',
 'star_rating',
 'helpful_votes',
 'total_votes',
 'vine',
 'verified_purchase',
 'review_headline',
 'review_body',
 'review_date']
```

In [7]:
```
dataset = []
```

In [8]:
```
for line in file:
    fields = line.strip().split('\t')
    d = dict(zip(header, fields))
    d['star_rating'] = int(d['star_rating'])
    d['helpful_votes'] = int(d['helpful_votes'])
    d['total_votes'] = int(d['total_votes'])
    dataset.append(d)
```

In [9]:
```
len(dataset)#[0]
```

Out[9]: 904765

Useful data Structures

To perform set intersections/unions efficiently, we first build data structures representing the set of items for each user and users for each item

In [9]:
```
usersPerItem = defaultdict(set)
itemsPerUser = defaultdict(set)
```

In [10]:
```
itemNames = {}
```

In [11]:
```
for d in dataset:
    user, item = d['customer_id'], d['product_id']
    usersPerItem[item].add(user) # Ui: Save the item and add the users that purchased that
    itemsPerUser[user].add(item) # Iu: Save the user and add the items that have been purch
    itemNames[item] = d['product_title']
```

Recommendation

Jaccard Similarity

$$\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

In [13]:
```
def Jaccard(s1, s2):
    numerator = len(s1.intersection(s2))
    denominator = len(s1.union(s2))
    return numerator/denominator
```

We want a recommendation function that return **items similar to a candidate item i**.

- Find the set of users who purchased i
- Iterate over all other items other than i
- For all other items, compute their similarity with i (and store it)
- Sort all other items by (Jaccard) similarity
- Return the most similar

In [15]:
```
def mostSimilar(i):
    similarities = []
    users = usersPerItem[i] # Scroll the users in each item
    for i2 in usersPerItem: # Scroll the items that have been purchased by this user
        if i2 == i: continue # Skip if is the current item
        sim = Jaccard(users, usersPerItem[i2])
        similarities.append((sim,i2))
    similarities.sort(reverse = True)
    return similarities[:10]
```

In [16]:
```
dataset[2]
```

Out[16]:
```
{'marketplace': 'US',
 'customer_id': '6111003',
 'review_id': 'RIZR67JKUDBI0',
 'product_id': 'B0006VMBHI',
 'product_parent': '603261968',
 'product_title': 'AudioQuest LP record clean brush',
 'product_category': 'Musical Instruments',
 'star_rating': 3,
 'helpful_votes': 0,
 'total_votes': 1,
 'vine': 'N',
 'verified_purchase': 'Y',
 'review_headline': 'Three Stars',
 'review_body': 'removes dust. does not clean',
 'review_date': '2015-08-31'}
```

In [18]:
```
# The query is just a product ID
query = dataset[2]['product_id']
query
```

Out[18]:
```
'B0006VMBHI'
```

In [19]:
```
mostSimilar(query)
```

Out[19]:
```
[(0.028446389496717725, 'B00006I5SD'),
 (0.01694915254237288, 'B00006I5SB'),
 (0.015065913370998116, 'B000AJR482'),
 (0.014204545454545454, 'B00E7MVP3S'),
 (0.008955223880597015, 'B001255YL2'),
 (0.008849557522123894, 'B003EIRVO8'),
 (0.008333333333333333, 'B0015VEZ22'),
 (0.00821917808219178, 'B00006I5UH'),
 (0.008021390374331552, 'B00008BWM7'),
 (0.007656967840735069, 'B000H2BC4E')]
```

Finally, let´s look at the items that were recommended

In [20]:
```python
itemNames[query]
```

Out[20]:
```
'AudioQuest LP record clean brush'
```

In [21]:
```python
[itemNames[x[1]] for x in mostSimilar(query)]
```

Out[21]:
```
['Shure SFG-2 Stylus Tracking Force Gauge',
 'Shure M97xE High-Performance Magnetic Phono Cartridge',
 'ART Pro Audio DJPRE II Phono Turntable Preamplifier',
 'Signstek Blue LCD Backlight Digital Long-Playing LP Turntable Stylus Force Scale Gauge Tester',
 'Audio Technica AT120E/T Standard Mount Phono Cartridge',
 'Technics: 45 Adaptor for Technics 1200 (SFWE010)',
 'GruvGlide GRUVGLIDE DJ Package',
 'STANTON MAGNETICS Record Cleaner Kit',
 'Shure M97xE High-Performance Magnetic Phono Cartridge',
 'Behringer PP400 Ultra Compact Phono Preamplifier']
```

Inefficient:

The slowest component is the iteration over all other items.

In fact is sufficient to iterate over those items purchased by one of the users who purchased i

Efficient Implementation

In [24]:
```python
def mostSimilarFast(i):
    similarities = []
    users = usersPerItem[i] # Scroll the users in each item
    candidateItems = set()
    for u in users:
        candidateItems = candidateItems.union(itemsPerUser[u])
    for i2 in usersPerItem: # Scroll the items that have been purchased by this user
        if i2 == i: continue # Skip if is the current item
        sim = Jaccard(users, usersPerItem[i2])
        similarities.append((sim,i2))
    similarities.sort(reverse = True)
    return similarities[:10]
```

In [25]:
```python
mostSimilarFast(query)
```

Out[25]:
```
[(0.028446389496717725, 'B00006I5SD'),
 (0.01694915254237288, 'B00006I5SB'),
 (0.015065913370998116, 'B000AJR482'),
 (0.014204545454545454, 'B00E7MVP3S'),
 (0.008955223880597015, 'B001255YL2'),
 (0.008849557522123894, 'B003EIRVO8'),
 (0.008333333333333333, 'B0015VEZ22'),
 (0.00821917808219178, 'B00006I5UH'),
 (0.008021390374331552, 'B00008BWM7'),
 (0.007656967840735069, 'B000H2BC4E')]
```

```
In [26]: [itemNames[x[1]] for x in mostSimilarFast(query)]
```

```
Out[26]: ['Shure SFG-2 Stylus Tracking Force Gauge',
 'Shure M97xE High-Performance Magnetic Phono Cartridge',
 'ART Pro Audio DJPRE II Phono Turntable Preamplifier',
 'Signstek Blue LCD Backlight Digital Long-Playing LP Turntable Stylus Force Scale Gauge Tester',
 'Audio Technica AT120E/T Standard Mount Phono Cartridge',
 'Technics: 45 Adaptor for Technics 1200 (SFWE010)',
 'GruvGlide GRUVGLIDE DJ Package',
 'STANTON MAGNETICS Record Cleaner Kit',
 'Shure M97xE High-Performance Magnetic Phono Cartridge',
 'Behringer PP400 Ultra Compact Phono Preamplifier']
```

Rating Predictor based on Similarity

Heuristic:

- The user (u)'s rating for an item *i* is a weighted combination of all of their previous ratings for items *j*
- The weight for each rating is given by the Jaccard similarity between *i* and *j*

This can be written as:

$$r(u,i) = \frac{1}{Z} \sum_{\underbrace{j \in I_u \setminus \{i\}}_{\text{All items the user has rated other than } i}} r_{u,j} \cdot \mathrm{sim}(i,j)$$

Normalization constant

$$Z = \sum_{j \in I_u \setminus \{i\}} \mathrm{sim}(i,j)$$

```
In [27]: reviewsPerUser = defaultdict(list)
         reviewsPerItem = defaultdict(list)
```

```
In [28]: for d in dataset:
             user, item = d['customer_id'], d['product_id']
             reviewsPerUser[user].append(d)
             reviewsPerItem[item].append(d)
```

```
In [29]: # This will be our baseline for comparison
         ratingMean = sum(d['star_rating'] for d in dataset) / len(dataset)
         ratingMean
```

```
Out[29]: 4.251102772543146
```

```
In [30]: def predictRating(user, item):
             ratings = []
             similarities = []
             for d in reviewsPerUser[user]:
                 i2 = d['product_id']
                 if i2 == item: continue
                 ratings.append(d['star_rating']) # r_u,j
                 similarities.append(Jaccard(usersPerItem[item], usersPerItem[i2])) # sim(i,j)
             if (sum(similarities) > 0):
                 weightedRatings = [(x*y) for x,y in zip(ratings, similarities)]
                 return sum(weightedRatings) / sum(similarities)
             else:
                 # User hasn't rated any similar items
                 return ratingMean
```

As an example, select a rating for prediction

```
In [31]: dataset[1]
```

```
Out[31]: {'marketplace': 'US',
          'customer_id': '14640079',
          'review_id': 'RZSL0BALIYUNU',
          'product_id': 'B003LRN53I',
          'product_parent': '986692292',
          'product_title': 'Sennheiser HD203 Closed-Back DJ Headphones',
          'product_category': 'Musical Instruments',
          'star_rating': 5,
          'helpful_votes': 0,
          'total_votes': 0,
          'vine': 'N',
          'verified_purchase': 'Y',
          'review_headline': 'Five Stars',
          'review_body': 'Nice headphones at a reasonable price.',
          'review_date': '2015-08-31'}
```

```
In [32]: u,i = dataset[1]['customer_id'], dataset[1]['product_id']
```

```
In [33]: predictRating(u, i)
```

```
Out[33]: 5.0
```

Similarly, we can evaluate accuracy across the entire corpus

```
In [34]: def MSE(predictions, labels):
             differences = [(x-y)**2 for x,y in zip(predictions, labels)]
             return sum(differences) / len(differences)
```

In [35]:
```
alwaysPredictMean = [ratingMean for d in dataset]
```

In [36]:
```
cfPredictions = [predictRating(d['customer_id'], d['product_id']) for d in dataset]
```

In [37]:
```
labels = [d['star_rating'] for d in dataset]
```

In [38]:
```
MSE(alwaysPredictMean, labels)
```

Out[38]: 1.4796142779564334

In [39]:
```
MSE(cfPredictions, labels)
```

Out[39]: 1.6146130004291603

- In fact in this case it did worse (in terms of the MSE) than always predicting the mean
- We could adapt this to use:
  1) A different similarity function (e.g. cosine)
  2) Similarity based on users ratiher thar than items
  3) A different weighting scheme

Latent Factor Model.

Bias Only Model

$$\arg\min_{\alpha,\beta} \frac{1}{N} \sum_{u,i}(\alpha + \beta_u + \beta_i - R_{u,i})^2 + \lambda \left[\sum_u \beta_u^2 + \sum_i \beta_i^2\right]$$

In this section is the code for the gradient equations; and after that a library is used to optimize the model

In [40]:
```
N = len(dataset)
nUsers = len(reviewsPerUser)
nItems = len(reviewsPerItem)
users = list(reviewsPerUser.keys())
items = list(reviewsPerItem.keys())
```

Note: Alpha and Beta (userbiases) are parameter we'll fit. This code sets their initial values (alpha to the mean rating, and beta_u / beta_i to zero)

In [ ]:
```
alpha = ratingMean
```

In [41]:
```
# Beta_u & Beta_i
userBiases = defaultdict(float)
itemBiases = defaultdict(float)
```

Our prediction function in this case just implements the bias only model:

$$f(\underset{\text{user}}{u}, \underset{\text{item}}{i}) = \alpha + \underset{\text{user bias}}{\beta_u} + \underset{\text{item bias}}{\beta_i}$$

function = offset + UserBias + ItemBias

In [42]:
```
def prediction(user, item):
    return alpha + userBiases[user] + itemBiases[item]
```

The first complex function to implement is this "unpack" function. The gradient descent library we'll use expects a single vector of parameters (θ), which we have to unpack to produce alpha and beta:

In [43]:
```
def unpack(theta):
    global alpha
    global userBiases
    global itemBiases
    alpha = theta[0]
    userBiases = dict(zip(users, theta[1:nUsers+1]))
    itemBiases = dict(zip(items, theta[1+nUsers:]))
```

Full cost function

$$\frac{1}{N}\sum_{u,i}(\alpha + \beta_u + \beta_i - R_{u,i})^2 + \lambda\left[\sum_u \beta_u^2 + \sum_i \beta_i^2\right]$$

In [44]:
```
def cost(theta, labels, lamb):
    unpack(theta)
    predictions = [prediction(d['customer_id'], d['product_id']) for d in dataset]
    cost = MSE(predictions, labels)
    print("MSE = " + str(cost))
    for u in userBiases:
        cost += lamb*userBiases[u]**2
    for i in itemBiases:
        cost += lamb*itemBiases[i]**2
    return cost
```

Derivative term for each parameter

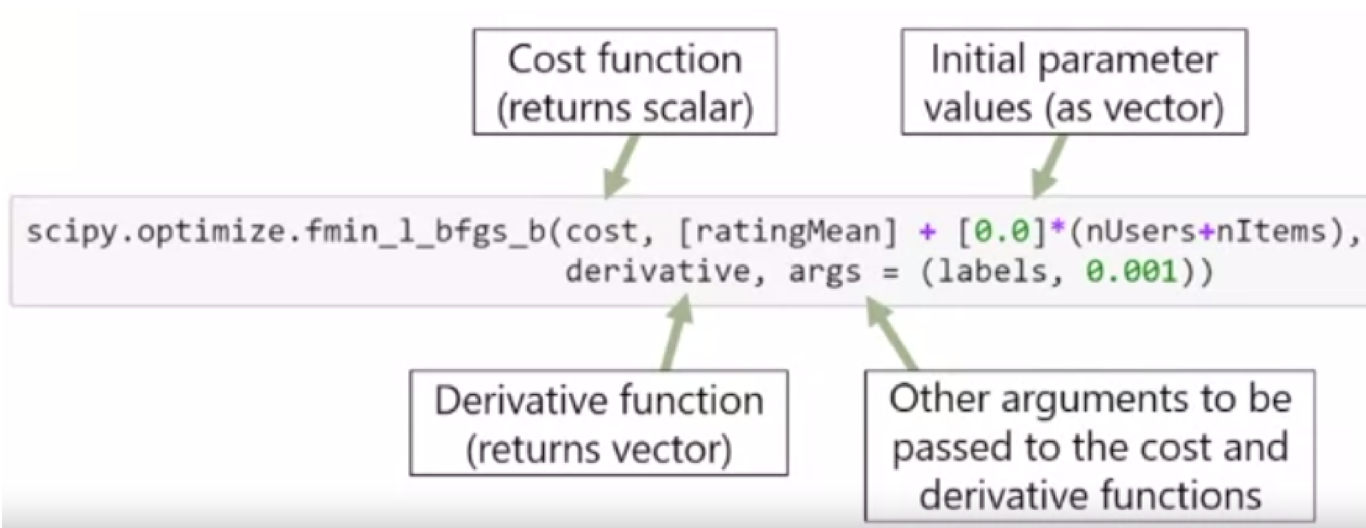
$$\frac{\partial \text{obj}}{\partial \beta_u} = \frac{1}{N} \sum_{u,i} 2(\alpha + \beta_u + \beta_i - R_{u,i}) + 2\lambda\beta_u$$

In [46]:
```python
def derivative(theta, labels, lamb):
    unpack(theta)
    N = len(dataset)
    d_alpha = 0
    d_UserBiases = defaultdict(float)
    d_ItemBiases = defaultdict(float)
    for d in dataset:
        u,i = d['customer_id'], d['product_id']
        pred = prediction(u, i)
        diff = pred - d['star_rating']
        d_alpha += 2/N*diff
        d_UserBiases[u] += 2/N*diff
        d_ItemBiases[i] += 2/N*diff
    for u in userBiases:
        d_UserBiases[u] += 2*lamb*userBiases[u]
    for i in itemBiases:
        d_ItemBiases[i] += 2*lamb*itemBiases[i]
    d_theta = [d_alpha] + [d_UserBiases[u] for u in users] + [d_ItemBiases[i] for i in iter
    return np.array(d_theta)
```

In [47]:
```python
MSE(alwaysPredictMean, labels)
```

Out[47]: 1.4796142779564334

Using Library from Scipy (lbfgs)



In [48]:
```
scipy.optimize.fmin_l_bfgs_b(cost, [ratingMean] + [0.0]*(nUsers+nItems), derivative, args
```

```
MSE = 1.4796142779564334
MSE = 1.468686355953835
MSE = 2.6961687181992064
MSE = 1.4681419018494124
MSE = 1.4523523347391192
MSE = 1.4513575397272933
MSE = 1.4476987674765316
MSE = 1.4421925605951182
MSE = 1.4415262672088056
MSE = 1.4413460037417523
MSE = 1.441397612244047
MSE = 1.4414066017099236
```

Out[48]:
```
(array([ 4.24278450e+00, -1.37216332e-03,  5.73953696e-03, ...,
         8.31051679e-04, -2.15966373e-03, -2.67061773e-04]),
 1.4574364057349305,
 {'grad': array([-4.52665785e-07,  8.64056712e-09, -2.76548829e-08, ...,
         -5.06227031e-09,  1.10178064e-08,  1.37544741e-09]),
  'task': b'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL',
  'funcalls': 12,
  'nit': 9,
  'warnflag': 0})
```

Complete Latent Factor Model

$$\arg\min_{\alpha,\beta,\gamma} \frac{1}{N} \sum_{u,i} (\alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i - R_{u,i})^2 + \lambda \left[\sum_u \beta_u^2 + \sum_i \beta_i^2 + \sum_i \|\gamma_i\|_2^2 + \sum_u \|\gamma_u\|_2^2 \right]$$

In [50]:
```
alpha = ratingMean
alpha
```

Out[50]:
```
4.251102772543146
```

In [51]:
```
userBiases = defaultdict(float)
itemBiases = defaultdict(float)
```

In [52]:
```
userGamma = {}
itemGamma = {}
```

In [53]:
```
K = 2 # Numbre of latent factors(i.e. dimensionality of gamma)
```

In [54]:
```
for u in reviewsPerUser:
    userGamma[u] = [random.random()*0.1 - 0.05 for k in range(K)]
```

In [55]:
```
for i in reviewsPerItem:
    itemGamma[i] = [random.random()*0.1 - 0.05 for k in range(K)]
```

From Theta to Alpha, Beta and Gamma

In [56]:
```python
def unpack(theta):
    global alpha
    global userBiases
    global itemBiases
    index = 0
    alpha = theta[index]
    index += 1
    userBiases = dict(zip(users, theta[index:index+nUsers]))
    index += nUsers
    itemBiases = dict(zip(items, theta[index:index+nItems]))
    index += nItems
    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 [57]:
```python
def inner(x, y):
    return sum([a*b for a,b in zip(x, y)])
```

In [59]:
```python
def prediction(user, item):
    return alpha + userBiases[user] + itemBiases[item] + inner(userGamma[user], itemGamma[i
```

In [60]:
```python
def cost(theta, lafbels, lamb):
    unpack(theta)
    predictions = [prediction(d['customer_id'], d['product_id']) for d in dataset]
    cost = MSE(predictions, labels)
    print("MSE = " + str(cost))

    for u in users:
        cost += lamb*userBiases[u]**2
        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
```

the video have a really bad Quality so de derivative can't be seen.

In [ ]: