In [1]:
```
import gzip
import math
import random
from collections import defaultdict
```

Data is available at http://cseweb.ucsd.edu/~jmcauley/pml/data/ (http://cseweb.ucsd.edu/~jmcauley/pml/data/). Download and save to your own directory

In [2]:
```
dataDir = "/home/jmcauley/pml_data/"
```

Amazon musical instrument review data. Originally from https://s3.amazonaws.com/amazon-reviews-pds/tsv/index.txt (https://s3.amazonaws.com/amazon-reviews-pds/tsv/index.txt)

In [3]:
```
path = dataDir + "amazon_reviews_us_Musical_Instruments_v1_00.tsv.gz"
f = gzip.open(path, 'rt', encoding="utf8")

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

Dataset contains the following fields

In [4]:
```
header
```

Out[4]:
```
['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']
```

Parse the data and convert fields to integers where needed

In [5]:
```
dataset = []

for line in f:
    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)
```

One row of the dataset (as a python dictionary)

In [6]:
```
dataset[0]
```

Out[6]:
```
{'customer_id': '45610553',
 'helpful_votes': 0,
 'marketplace': 'US',
 'product_category': 'Musical Instruments',
 'product_id': 'B00HH62VB6',
 'product_parent': '618218723',
 'product_title': 'AGPtek® 10 Isolated Output 9V 12V 18V Guitar Pedal B
oard Power Supply Effect Pedals with Isolated Short Cricuit / Overcurre
nt Protection',
 'review_body': 'Works very good, but induces ALOT of noise.',
 'review_date': '2015-08-31',
 'review_headline': 'Three Stars',
 'review_id': 'RMDCHWD0Y5OZ9',
 'star_rating': 3,
 'total_votes': 1,
 'verified_purchase': 'N',
 'vine': 'N'}
```

Extract a few utility data structures

In [7]:
```
usersPerItem = defaultdict(set) # Maps an item to the users who rated it
itemsPerUser = defaultdict(set) # Maps a user to the items that they rat
ed
itemNames = {}
ratingDict = {} # To retrieve a rating for a specific user/item pair

for d in dataset:
    user,item = d['customer_id'], d['product_id']
    usersPerItem[item].add(user)
    itemsPerUser[user].add(item)
    ratingDict[(user,item)] = d['star_rating']
    itemNames[item] = d['product_title']
```

Extract per-user and per-item averages (useful later for rating prediction)

In [8]:
```
userAverages = {}
itemAverages = {}

for u in itemsPerUser:
    rs = [ratingDict[(u,i)] for i in itemsPerUser[u]]
    userAverages[u] = sum(rs) / len(rs)

for i in usersPerItem:
    rs = [ratingDict[(u,i)] for u in usersPerItem[i]]
    itemAverages[i] = sum(rs) / len(rs)
```

Similarity metrics

Jaccard

In [9]:
```
def Jaccard(s1, s2):
    numer = len(s1.intersection(s2))
    denom = len(s1.union(s2))
    if denom == 0:
        return 0
    return numer / denom
```

Cosine

Simple implementation for set-structured data

In [10]:
```
def CosineSet(s1, s2):
    # Not a proper implementation, operates on sets so correct for inter
actions only
    numer = len(s1.intersection(s2))
    denom = math.sqrt(len(s1)) * math.sqrt(len(s2))
    if denom == 0:
        return 0
    return numer / denom
```

Or for real values (e.g. ratings). Note that this implementation uses global variables (usersPerItem, ratingDict), which ideally should be passed as parameters.

```
In [11]: def Cosine(i1, i2):
             # Between two items
             inter = usersPerItem[i1].intersection(usersPerItem[i2])
             numer = 0
             denom1 = 0
             denom2 = 0
             for u in inter:
                 numer += ratingDict[(u,i1)]*ratingDict[(u,i2)]
             for u in usersPerItem[i1]:
                 denom1 += ratingDict[(u,i1)]**2
             for u in usersPerItem[i2]:
                 denom2 += ratingDict[(u,i2)]**2
             denom = math.sqrt(denom1) * math.sqrt(denom2)
             if denom == 0: return 0
             return numer / denom
```

Pearson

```
In [12]: def Pearson(i1, i2):
             # Between two items
             iBar1 = itemAverages[i1]
             iBar2 = itemAverages[i2]
             inter = usersPerItem[i1].intersection(usersPerItem[i2])
             numer = 0
             denom1 = 0
             denom2 = 0
             for u in inter:
                 numer += (ratingDict[(u,i1)] - iBar1)*(ratingDict[(u,i2)] - iBar
         2)
             for u in inter: #usersPerItem[i1]:
                 denom1 += (ratingDict[(u,i1)] - iBar1)**2
             #for u in usersPerItem[i2]:
                 denom2 += (ratingDict[(u,i2)] - iBar2)**2
             denom = math.sqrt(denom1) * math.sqrt(denom2)
             if denom == 0: return 0
             return numer / denom
```

Retrieve the most similar items to a given query

In this case, based on the Jaccard similarity

In [13]:
```
def mostSimilar(i, N):
    similarities = []
    users = usersPerItem[i]
    for i2 in usersPerItem:
        if i2 == i: continue
        sim = Jaccard(users, usersPerItem[i2])
        #sim = Pearson(i, i2) # Could use alternate similarity metrics s
traightforwardly
        similarities.append((sim,i2))
    similarities.sort(reverse=True)
    return similarities[:10]
```

Choose an item to use as a query

In [14]:
```
dataset[2]
```

Out[14]:
```
{'customer_id': '6111003',
 'helpful_votes': 0,
 'marketplace': 'US',
 'product_category': 'Musical Instruments',
 'product_id': 'B0006VMBHI',
 'product_parent': '603261968',
 'product_title': 'AudioQuest LP record clean brush',
 'review_body': 'removes dust. does not clean',
 'review_date': '2015-08-31',
 'review_headline': 'Three Stars',
 'review_id': 'RIZR67JKUDBI0',
 'star_rating': 3,
 'total_votes': 1,
 'verified_purchase': 'Y',
 'vine': 'N'}
```

In [15]:
```
query = dataset[2]['product_id']
```

Retrieve the most similary items

In [16]:
```
ms = mostSimilar(query, 10)
```

In [17]:
```
ms
```

Out[17]:
```
[(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')]
```

Print names of query and recommended items

In [18]:
```
itemNames[query]
```

Out[18]:
```
'AudioQuest LP record clean brush'
```

In [19]:
```
[itemNames[x[1]] for x in ms]
```

Out[19]:
```
['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']
```

Faster implementation

In [20]:
```
def mostSimilarFast(i, N):
    similarities = []
    users = usersPerItem[i]
    candidateItems = set()
    for u in users:
        candidateItems = candidateItems.union(itemsPerUser[u])
    for i2 in candidateItems:
        if i2 == i: continue
        sim = Jaccard(users, usersPerItem[i2])
        similarities.append((sim,i2))
    similarities.sort(reverse=True)
    return similarities[:N]
```

Confirm that results are the same...

In [21]:
```
mostSimilarFast(query, 10)
```

Out[21]:
```
[(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')]
```

Similarity-based rating estimation

Use our similarity functions to estimate ratings. Start by building a few utility data structures.

In [22]:
```
reviewsPerUser = defaultdict(list)
reviewsPerItem = defaultdict(list)
```

In [23]:
```
for d in dataset:
    user,item = d['customer_id'], d['product_id']
    reviewsPerUser[user].append(d)
    reviewsPerItem[item].append(d)
```

In [24]:
```
ratingMean = sum([d['star_rating'] for d in dataset]) / len(dataset)
```

In [25]:
```
ratingMean
```

Out[25]: 4.251102772543146

Rating prediction heuristic (several alternatives from Chapter 4 could be used)

In [26]:
```
def predictRating(user,item):
    ratings = []
    similarities = []
    for d in reviewsPerUser[user]:
        i2 = d['product_id']
        if i2 == item: continue
        ratings.append(d['star_rating'] - itemAverages[i2])
        similarities.append(Jaccard(usersPerItem[item],usersPerItem[i2
]))
    if (sum(similarities) > 0):
        weightedRatings = [(x*y) for x,y in zip(ratings,similarities)]
        return itemAverages[item] + sum(weightedRatings) / sum(similarit
ies)
    else:
        # User hasn't rated any similar items
        return ratingMean
```

In [27]:
```
dataset[1]
```

Out[27]:
```
{'customer_id': '14640079',
 'helpful_votes': 0,
 'marketplace': 'US',
 'product_category': 'Musical Instruments',
 'product_id': 'B003LRN53I',
 'product_parent': '986692292',
 'product_title': 'Sennheiser HD203 Closed-Back DJ Headphones',
 'review_body': 'Nice headphones at a reasonable price.',
 'review_date': '2015-08-31',
 'review_headline': 'Five Stars',
 'review_id': 'RZSL0BALIYUNU',
 'star_rating': 5,
 'total_votes': 0,
 'verified_purchase': 'Y',
 'vine': 'N'}
```

Predict a rating for a particular user/item pair

In [28]:
```
u,i = dataset[1]['customer_id'], dataset[1]['product_id']
```

In [29]:
```
predictRating(u, i)
```

Out[29]:
```
4.509357030989021
```

Compute the MSE for a model based on this heuristic

In [30]:
```
def MSE(predictions, labels):
    differences = [(x-y)**2 for x,y in zip(predictions,labels)]
    return sum(differences) / len(differences)
```

Compared to a trivial predictor which always predicts the mean

In [31]:
```
alwaysPredictMean = [ratingMean for d in dataset]
```

Get predictions for all instances (fairly slow!)

In [32]:
```
simPredictions = [predictRating(d['customer_id'], d['product_id']) for d
in dataset]
```

In [33]:
```
labels = [d['star_rating'] for d in dataset]
```

In [34]:
```
MSE(alwaysPredictMean, labels)
```

Out[34]:
```
1.4796142779564334
```

In [35]:
```
MSE(simPredictions, labels)
```

Out[35]: 1.44672577948388

Exercises

4.1

(implementation is provided via the function mostSimilarFast above)

4.2

(using Amazon musical instruments data from examples above)

In [36]:
```
def simTest(simFunction, nUserSamples):
    sims = []
    randomSims = []

    items = set(usersPerItem.keys())
    users = list(itemsPerUser.keys())

    for u in random.sample(users, nUserSamples):
        itemsU = set(itemsPerUser[u])
        if len(itemsU) < 2: continue # User needs at least two interacti
ons
        (i,j) = random.sample(itemsU, 2)
        k = random.sample(items.difference(itemsU),1)[0]
        usersi = usersPerItem[i].difference(set([u]))
        usersj = usersPerItem[j].difference(set([u]))
        usersk = usersPerItem[k].difference(set([u]))
        sims.append(simFunction(usersi,usersj))
        randomSims.append(simFunction(usersi,usersk))

    print("Average similarity = " + str(sum(sims)/len(sims)))
    print("Average similarity (with random item) = " + str(sum(randomSim
s)/len(randomSims)))
```

In [37]:
```
simTest(Jaccard, 1000)
```

```
Average similarity = 0.0019330961239460492
Average similarity (with random item) = 0.0
```

In [38]:
```
simTest(CosineSet, 1000)
```

```
Average similarity = 0.005634438126569325
Average similarity (with random item) = 0.0
```

4.3

In [39]:
```
items = set(usersPerItem.keys())
users = set(itemsPerUser.keys())
```

In [40]:
```
# 1: Average cosine similarity between i and items in u's history
def rec1score(u, i, userHistory):
    if len(userHistory) == 0:
        return 0
    averageSim = []
    s1 = usersPerItem[i].difference(set([u]))
    for h in userHistory:
        s2 = usersPerItem[h].difference(set([u]))
        averageSim.append(Jaccard(s1,s2))
    averageSim = sum(averageSim)/len(averageSim)
    return averageSim

# 2: Jaccard similarity with most similar user who has consumed i
def rec2score(u, i, userHistory):
    bestSim = None
    for v in usersPerItem[i]:
        if u == v:
            continue
        sim = Jaccard(userHistory, itemsPerUser[v])
        if bestSim == None or sim > bestSim:
            bestSim = sim
    if bestSim == None:
        return 0
    return bestSim

# Generate a recommendation for a user based on a given scoring function
def rec(u, score):
    history = itemsPerUser[u]
    if len(history) > 5: # If the history is too long, just take a sampl
e
        history = random.sample(history,5)
    bestItem = None
    bestScore = None
    for i in items:
        if i in itemsPerUser[u]: continue
        s = score(u, i, history)
        if bestItem == None or s > bestScore:
            bestItem = i
            bestScore = s

    return bestItem, bestScore
```

In [41]:
```
u = random.sample(users,1)[0]
```

In [42]:
```
rec(u, rec1score)
```

Out[42]:
```
('B002KYLGT8', 0.043478260869565216)
```

In [43]:
```
rec(u, rec2score)
```

Out[43]: ('B00HCPTXJA', 0.5)

In [44]:
```
def recTest(simFunction, nUserSamples):
    items = set(usersPerItem.keys())
    users = list(itemsPerUser.keys())

    better = 0
    worse = 0

    for u in random.sample(users, nUserSamples):
        itemsU = set(itemsPerUser[u])
        if len(itemsU) < 2:
            continue
        i = random.sample(itemsU, 1)[0]
        uWithheld = itemsU.difference(set([i]))
        j = random.sample(items,1)[0]

        si = simFunction(u,i,uWithheld)
        sj = simFunction(u,j,uWithheld)

        if si > sj:
            better += 1
        if sj > si:
            worse += 1

    print("Better than random " + str(better) + " times")
    print("Worse than random " + str(worse) + " times")
```

Results on this dataset aren't particularly interesting. Could try with a denser dataset (so that many items have non-zero similarity) to get more interesting results.

In [45]:
```
recTest(rec1score,5000)
```

```
Better than random 306 times
Worse than random 1 times
```

In [46]:
```
recTest(rec2score,5000)
```

```
Better than random 278 times
Worse than random 4 times
```

4.4

(following code and auxiliary data structures from the examples above)

Equation 4.20

In [47]:
```
def predictRating1(user,item):
    ratings = []
    similarities = []
    for d in reviewsPerUser[user]:
        i2 = d['product_id']
        if i2 == item: continue
        ratings.append(d['star_rating'])
        similarities.append(Jaccard(usersPerItem[item],usersPerItem[i2
]))
    if (sum(similarities) > 0):
        weightedRatings = [(x*y) for x,y in zip(ratings,similarities)]
        return sum(weightedRatings) / sum(similarities)
    else:
        return ratingMean
```

Equation 4.21

In [48]:
```
def predictRating2(user,item):
    ratings = []
    similarities = []
    for d in reviewsPerItem[item]:
        u2 = d['customer_id']
        if u2 == user: continue
        ratings.append(d['star_rating'])
        similarities.append(Jaccard(itemsPerUser[user],itemsPerUser[u2
]))
    if (sum(similarities) > 0):
        weightedRatings = [(x*y) for x,y in zip(ratings,similarities)]
        return sum(weightedRatings) / sum(similarities)
    else:
        return ratingMean
```

Equation 4.22

In [49]:
```
def predictRating3(user,item):
    ratings = []
    similarities = []
    for d in reviewsPerUser[user]:
        i2 = d['product_id']
        if i2 == item: continue
        ratings.append(d['star_rating'] - itemAverages[i2])
        similarities.append(Jaccard(usersPerItem[item],usersPerItem[i2
]))
    if (sum(similarities) > 0):
        weightedRatings = [(x*y) for x,y in zip(ratings,similarities)]
        return itemAverages[item] + sum(weightedRatings) / sum(similarit
ies)
    else:
        return ratingMean
```

In [50]:
```
simPredictions1 = [predictRating1(d['customer_id'], d['product_id']) for
d in dataset]
simPredictions2 = [predictRating2(d['customer_id'], d['product_id']) for
d in dataset]
simPredictions3 = [predictRating3(d['customer_id'], d['product_id']) for
d in dataset]
```

In [51]:
```
MSE(alwaysPredictMean, labels)
```

Out[51]: 1.4796142779564334

In [52]:
```
MSE(simPredictions1, labels)
```

Out[52]: 1.6146130004291603

In [53]:
```
MSE(simPredictions2, labels)
```

Out[53]: 1.4540822838636853

In [54]:
```
MSE(simPredictions3, labels)
```

Out[54]: 1.44672577948388