Chapter 3: Implicit ratings and item-based filtering

Explicit Ratings

- Pandora/ YouTube thumbs up/down
- Amazon 1-5 star rating system

Implicit Ratings

- User clicks (news articles/ Amazon products)
- Music listened to (iTunes/ Spotify), including amount of listens

Problems with Explicit Ratings

- People are lazy, don't always rate items!
- People lie/ give only partial information
- People do not UPDATE RATINGS!

What metrics can we use for implicit ratings?

- **Web pages:** clicking link to page, time spent looking at page, repeated visits, referring page to others, what person watches on Hulu
- **Music players:** what the person plays, skipping songs, number of listens

Issues with too many users

CALCULATIONS GET COMPLICATED QUICKLY!

- To calculate millions of neighbors is burdensome
- Recalculate each time a new user is added!

Problems with user-based filtering

- **Scalability:** computation increases as number of users increases
- **Sparsity:** many users that rate *small fraction* of total products

Possible solution? ITEM-BASED FILTERING!

- Instead of calculating recommendations for each **user**, we find similar **items**
- Find similar items, combine that with user's rating for items, and use that to recommend!
- **DON'T NEED TO STORE ALL USER RATINGS** FOR ITEM-BASED FILTERING

Adjusted cosine similarity

$$\frac{\sum(R_{u,i} - \bar{R})(R_{u,j} - \bar{R})}{\sqrt{\sum(R_{u,i} - \bar{R})^2}(\sqrt{\sum(R_{u,j} - \bar{R})^2}}$$

- Perform normal cosine similarity, except on normalized (value - avg) form of the values

THIS GIVES YOU HOW SIMILAR TWO *ITEMS* ARE! How can we apply this to predict user ratings?

$$p(u,i) = \frac{\sum_{\in similarTo(i)}(S_{i,N} \times R_{u,N})}{\sum_{\in similarTo(i)}(|S_{i,N}|)}$$

^THAT'S how! Pretty much weights the similarity between two bands into our accounting for the user's rating of that band...

LIGHTBULB MOMENT!!!!!! PEARSON CORRELATION COEFFICIENT IS JUST COSINE SIMILARITY WITH NORMALIZED VALUES! HA! -also the cosine similarity we saw before was used between users rather than items

BUT WAIT, THERE'S MORE! We need to normalize the user ratings

$$R_{u,n}$$

should be normalized from -1 to 1

THIS REQUIRES THE NORMALIZATION/DENORMALIZATION FORMULAE THAT I WILL MAKE SERIOUS EFFORTS TO UNDERSTAND (covered in next chapter)

To use this method we need

- Similarity matrix, calculated in Pearson correlation
- Normalized user ratings (from -1 to 1 for Pearson)

Slope One

Use database of deviations (which are computed **ahead of time**) to compute the actual predictions.

1. **Done ahead of time:** compute deviations between every pair of items
2. Use deviations to make predictions

The beauty of this method?? We don't have to recalculate entire matrix each time new user added to system.

We use that trick where we say ((9 * 2) + 4) / 10 = 2.2 ^because we used to have 9 ratings w/ deviation of two, and we need to add the new ratings and divide by the new number

Deviation: average distance between bands across all users who have rated both bands

In [4]:
```
#user data
users2 = {"Amy": {"Taylor Swift": 4, "PSY": 3, "Whitney Houston": 4},
 "Ben": {"Taylor Swift": 5, "PSY": 2},
 "Clara": {"PSY": 3.5, "Whitney Houston": 4},
 "Daisy": {"Taylor Swift": 5, "Whitney Houston": 3}}
```

In [138]:
```
def computeDeviations(userData):
    frequencies = {}
    deviations = {}
    #devations are average distance between items
    for ratings in userData.values():
        for (item1, rating1) in ratings.items():
            frequencies.setdefault(item1, {})
            deviations.setdefault(item1, {})
            for (item2, rating2) in ratings.items():
                if item1 != item2:
                    #use default value trick to initialize to zero
                    frequencies[item1].setdefault(item2, 0.0)
                    deviations[item1].setdefault(item2, 0.0)
                    frequencies[item1][item2] += 1
                    deviations[item1][item2] += rating1 - rating2
    for (item, ratings) in deviations.items():
        for item2 in ratings:
            ratings[item2] /= frequencies[item][item2]

    return (deviations, frequencies)

(devs, freqs) = computeDeviations(users2)
print devs
print freqs
```

```
{'PSY': {'Taylor Swift': -2.0, 'Whitney Houston': -0.75}, 'Taylor Swif
t': {'PSY': 2.0, 'Whitney Houston': 1.0}, 'Whitney Houston': {'PSY': 0.
75, 'Taylor Swift': -1.0}}
{'PSY': {'Taylor Swift': 2.0, 'Whitney Houston': 2.0}, 'Taylor Swift':
 {'PSY': 2.0, 'Whitney Houston': 2.0}, 'Whitney Houston': {'PSY': 2.0,
 'Taylor Swift': 2.0}}
```

```
In [105]: def slopeOneRecommender(userRatings):
              #since abandoned class notion, will have local freqs
              #and the global freqs
              recommendations = {}
              (deviations,frequencies) = computeDeviations(users2)
              localFrequencies = {}
              #iterate over all items user has NOT rated
              for (item, userRating) in userRatings.items():
                  for (dItem, diffRatings) in deviations.items():
                      if dItem not in userRatings and item in deviations[dItem]:
                          #factor these into the weight
                          freq = frequencies[dItem][item]
                          recommendations.setdefault(dItem, 0.0)
                          localFrequencies.setdefault(dItem, 0)
                          recommendations[dItem] += (diffRatings[item] + userRatin
          g) * freq
                          localFrequencies[dItem] += freq

              recommendations = [(p, q / localFrequencies[p]) for (p, q) in recomm
          endations.items()]
              #magic sort algorithm we have been using the whole time
              recommendations.sort(key=lambda artistTuple: artistTuple[1], reverse
           = True)
              return recommendations

          testUser = users2["Ben"]
          rec = slopeOneRecommender(testUser)
          print rec
```

```
[('Whitney Houston', 3.375)]
```

Movie Lens

In [8]:
```
#NOTE: RAN INTO PROBLEMS LOADING DATASET, AND CALCULATING VALUES WITHOUT
 IMPLEMENTING CLASS
#WHICH IS ALREADY DONE BELOW- WILL CONTINUE CHAPTER BY ALTERING EXAMPLE
 CODE


#!/usr/bin/python
# -*- coding: utf-8 -*-
import codecs
from math import sqrt

users2 = {
    'Amy': {'Taylor Swift': 4, 'PSY': 3, 'Whitney Houston': 4},
    'Ben': {'Taylor Swift': 5, 'PSY': 2},
    'Clara': {'PSY': 3.5, 'Whitney Houston': 4},
    'Daisy': {'Taylor Swift': 5, 'Whitney Houston': 3},
    }

users = {
    'Angelica': {
        'Blues Traveler': 3.5,
        'Broken Bells': 2.0,
        'Norah Jones': 4.5,
        'Phoenix': 5.0,
        'Slightly Stoopid': 1.5,
        'The Strokes': 2.5,
        'Vampire Weekend': 2.0,
        },
    'Bill': {
        'Blues Traveler': 2.0,
        'Broken Bells': 3.5,
        'Deadmau5': 4.0,
        'Phoenix': 2.0,
        'Slightly Stoopid': 3.5,
        'Vampire Weekend': 3.0,
        },
    'Chan': {
        'Blues Traveler': 5.0,
        'Broken Bells': 1.0,
        'Deadmau5': 1.0,
        'Norah Jones': 3.0,
        'Phoenix': 5,
        'Slightly Stoopid': 1.0,
        },
    'Dan': {
        'Blues Traveler': 3.0,
        'Broken Bells': 4.0,
        'Deadmau5': 4.5,
        'Phoenix': 3.0,
        'Slightly Stoopid': 4.5,
        'The Strokes': 4.0,
        'Vampire Weekend': 2.0,
        },
    'Hailey': {
        'Broken Bells': 4.0,
        'Deadmau5': 1.0,
        'Norah Jones': 4.0,
        'The Strokes': 4.0,
        'Vampire Weekend': 1.0,
        },
    'Jordyn': {
        'Broken Bells': 4.5,
        'Deadmau5': 4.0,
        'Norah Jones': 5.0,
        'Phoenix': 5.0,
        'Slightly Stoopid': 4.5,
        'The Strokes': 4.0,
        'Vampire Weekend': 4.0,
        },
    'Sam': {
        'Blues Traveler': 5.0,
        'Broken Bells': 2.0,
        'Norah Jones': 3.0,
        'Phoenix': 5.0,
        'Slightly Stoopid': 4.0,
        'The Strokes': 5.0,
        },
    'Veronica': {
        'Blues Traveler': 3.0,
        'Norah Jones': 5.0,
        'Phoenix': 4.0,
        'Slightly Stoopid': 2.5,
        'The Strokes': 3.0,
        },
    }


class recommender:

    def __init__(
        self,
        data,
        k=1,
        metric='pearson',
        n=5,
        ):
        """ initialize recommender
      currently, if data is dictionary the recommender is initialized
      to it.
      For all other data types of data, no initialization occurs
      k is the k value for k nearest neighbor
      metric is which distance formula to use
      n is the maximum number of recommendations to make"""

        self.k = k
        self.n = n
        self.username2id = {}
        self.userid2name = {}
        self.productid2name = {}

      #
      # The following two variables are used for Slope One
      #

        self.frequencies = {}
        self.deviations = {}

      # for some reason I want to save the name of the metric

        self.metric = metric
        if self.metric == 'pearson':
            self.fn = self.pearson

      #
      # if data is dictionary set recommender data to it
      #

        if type(data).__name__ == 'dict':
            self.data = data

    def convertProductID2name(self, id):
        """Given product id number return product name"""

        if id in self.productid2name:
            return self.productid2name[id]
        else:
            return id

    def userRatings(self, id, n):
        """Return n top ratings for user with id"""

        print 'Ratings for ' + self.userid2name[id]
        ratings = self.data[id]
        print len(ratings)
        ratings = list(ratings.items())[:n]
        ratings = [(self.convertProductID2name(k), v) for (k, v) in
                   ratings]

      # finally sort and return

        ratings.sort(key=lambda artistTuple: artistTuple[1],
                     reverse=True)
        for rating in ratings:
            print '%s\t%i' % (rating[0], rating[1])

    def showUserTopItems(self, user, n):
        """ show top n items for user"""

        items = list(self.data[user].items())
        items.sort(key=lambda itemTuple: itemTuple[1], reverse=True)
        for i in range(n):
            print '%s\t%i' % (self.convertProductID2name(items[i][0]),
                              items[i][1])

    def loadMovieLens(self, path=''):
        self.data = {}

      #
      # first load movie ratings
      #

        i = 0

      #
      # First load book ratings into self.data
      #
      # f = codecs.open(path + "u.data", 'r', 'utf8')

        f = codecs.open(path + 'u.data', 'r', 'ascii')

      #  f = open(path + "u.data")

        for line in f:
            i += 1

         # separate line into fields

            fields = line.split('\t')
            user = fields[0]
            movie = fields[1]
            rating = int(fields[2].strip().strip('"'))
            if user in self.data:
                currentRatings = self.data[user]
            else:
                currentRatings = {}
            currentRatings[movie] = rating
            self.data[user] = currentRatings
        f.close()

    #
    # Now load movie into self.productid2name
    # the file u.item contains movie id, title, release date among
    # other fields
    #
    # f = codecs.open(path + "u.item", 'r', 'utf8')

        f = codecs.open(path + 'u.item', 'r', 'iso8859-1', 'ignore')

    # f = open(path + "u.item")

        for line in f:
            i += 1

         # separate line into fields

            fields = line.split('|')
            mid = fields[0].strip()
            title = fields[1].strip()
            self.productid2name[mid] = title
        f.close()

    #
    #  Now load user info into both self.userid2name
    #  and self.username2id
    #
    # f = codecs.open(path + "u.user", 'r', 'utf8')

        f = open(path + 'u.user')
        for line in f:
            i += 1
            fields = line.split('|')
            userid = fields[0].strip('"')
            self.userid2name[userid] = line
            self.username2id[line] = userid
        f.close()
        print i

  def loadBookDB(self, path=''):
        """loads the BX book dataset. Path is where the BX files are
    located"""

        self.data = {}
        i = 0

    #
    # First load book ratings into self.data
    #

        f = codecs.open(path + 'u.data', 'r', 'utf8')
```

```
          for line in f:
              i += 1

           # separate line into fields

              fields = line.split(';')
              user = fields[0].strip('"')
              book = fields[1].strip('"')
              rating = int(fields[2].strip().strip('"'))
              if rating > 5:
                  print ('EXCEEDING ', rating)
              if user in self.data:
                  currentRatings = self.data[user]
              else:
                  currentRatings = {}
              currentRatings[book] = rating
              self.data[user] = currentRatings
          f.close()

        #
        # Now load books into self.productid2name
        # Books contains isbn, title, and author among other fields
        #

          f = codecs.open(path + 'BX-Books.csv', 'r', 'utf8')
          for line in f:
              i += 1

           # separate line into fields

              fields = line.split(';')
              isbn = fields[0].strip('"')
              title = fields[1].strip('"')
              author = fields[2].strip().strip('"')
              title = title + ' by ' + author
              self.productid2name[isbn] = title
          f.close()

        #
        #  Now load user info into both self.userid2name and
        #  self.username2id
        #

          f = codecs.open(path + 'BX-Users.csv', 'r', 'utf8')
          for line in f:
              i += 1

           # separate line into fields

              fields = line.split(';')
              userid = fields[0].strip('"')
              location = fields[1].strip('"')
              if len(fields) > 3:
                  age = fields[2].strip().strip('"')
              else:
                  age = 'NULL'
              if age != 'NULL':
                  value = location + '  (age: ' + age + ')'
              else:
                  value = location
              self.userid2name[userid] = value
              self.username2id[location] = userid
          f.close()
          print i

      def computeDeviations(self):

        # for each person in the data:
        #    get their ratings

          for ratings in self.data.values():

           # for each item & rating in that set of ratings:

              for (item, rating) in ratings.items():
                  self.frequencies.setdefault(item, {})
                  self.deviations.setdefault(item, {})

              # for each item2 & rating2 in that set of ratings:

                  for (item2, rating2) in ratings.items():
                      if item != item2:

                    # add the difference between the ratings to our
                    # computation

                          self.frequencies[item].setdefault(item2, 0)
                          self.deviations[item].setdefault(item2, 0.0)
                          self.frequencies[item][item2] += 1
                          self.deviations[item][item2] += rating - rating2

          for (item, ratings) in self.deviations.items():
              for item2 in ratings:
                  ratings[item2] /= self.frequencies[item][item2]

      def slopeOneRecommendations(self, userRatings):
          recommendations = {}
          frequencies = {}

        # for every item and rating in the user's recommendations

          for (userItem, userRating) in userRatings.items():

           # for every item in our dataset that the user didn't rate

              for (diffItem, diffRatings) in self.deviations.items():
                  if diffItem not in userRatings and userItem \
                      in self.deviations[diffItem]:
                      freq = self.frequencies[diffItem][userItem]
                      recommendations.setdefault(diffItem, 0.0)
                      frequencies.setdefault(diffItem, 0)

                 # add to the running sum representing the numerator
                 # of the formula
```

```
                recommendations[diffItem] += (diffRatings[userItem]
                        + userRating) * freq

            # keep a running sum of the frequency of diffitem

                frequencies[diffItem] += freq
        recommendations = [(self.convertProductID2name(k), v
                           / frequencies[k]) for (k, v) in
                           recommendations.items()]

      # finally sort and return

        recommendations.sort(key=lambda artistTuple: artistTuple[1],
                             reverse=True)

      # I am only going to return the first 50 recommendations

        return recommendations[:50]

    def pearson(self, rating1, rating2):
        sum_xy = 0
        sum_x = 0
        sum_y = 0
        sum_x2 = 0
        sum_y2 = 0
        n = 0
        for key in rating1:
            if key in rating2:
                n += 1
                x = rating1[key]
                y = rating2[key]
                sum_xy += x * y
                sum_x += x
                sum_y += y
                sum_x2 += pow(x, 2)
                sum_y2 += pow(y, 2)
        if n == 0:
            return 0

      # now compute denominator

        denominator = sqrt(sum_x2 - pow(sum_x, 2) / n) * sqrt(sum_y2
                - pow(sum_y, 2) / n)
        if denominator == 0:
            return 0
        else:
            return (sum_xy - sum_x * sum_y / n) / denominator

    def computeNearestNeighbor(self, username):
        """creates a sorted list of users based on their distance
      to username"""

        distances = []
        for instance in self.data:
            if instance != username:
                distance = self.fn(self.data[username],
                              self.data[instance])
                distances.append((instance, distance))

      # sort based on distance -- closest first

        distances.sort(key=lambda artistTuple: artistTuple[1],
                       reverse=True)
        return distances

    def recommend(self, user):
        """Give list of recommendations"""

        recommendations = {}

      # first get list of users  ordered by nearness

        nearest = self.computeNearestNeighbor(user)

      #
      # now get the ratings for the user
      #

        userRatings = self.data[user]

      #
      # determine the total distance

        totalDistance = 0.0
        for i in range(self.k):
            totalDistance += nearest[i][1]

      # now iterate through the k nearest neighbors
      # accumulating their ratings

        for i in range(self.k):

         # compute slice of pie

            weight = nearest[i][1] / totalDistance

         # get the name of the person

            name = nearest[i][0]

         # get the ratings for this person

            neighborRatings = self.data[name]

         # get the name of the person
         # now find bands neighbor rated that user didn't

            for artist in neighborRatings:
                if not artist in userRatings:
                    if artist not in recommendations:
                        recommendations[artist] = \
                            neighborRatings[artist] * weight
                    else:
                        recommendations[artist] = \
                            recommendations[artist] \
                            + neighborRatings[artist] * weight

      # now make list from dictionary and only get the first n items

        recommendations = list(recommendations.items())[:self.n]
        recommendations = [(self.convertProductID2name(k), v) for (k,
                           v) in recommendations]

      # finally sort and return

        recommendations.sort(key=lambda artistTuple: artistTuple[1],
                             reverse=True)
        return recommendations

r = recommender(0)
r.loadMovieLens('./ml-100k/')

#input personal preferences
personalRatings = {u'214': 5, u'3': 4, u'102': 4, u'31': 3}
r.computeDeviations()
print r.slopeOneRecommendations(personalRatings)
```

```
102625
[(u"I Don't Want to Talk About It (De eso no se habla) (1993)", 6.5),
 (u'Bewegte Mann, Der (1994)', 6.333333333333333), (u'Year of the Horse
(1997)', 6.333333333333333), (u'Santa with Muscles (1996)', 6.0), (u'An
na (1996)', 6.0), (u'Object of My Affection, The (1998)', 6.0), (u'Witn
ess (1985)', 6.0), (u"Someone Else's America (1995)", 6.0), (u'Telling
 Lies in America (1997)', 6.0), (u'Sweet Nothing (1995)', 6.0), (u'Path
er Panchali (1955)', 5.714285714285714), (u'Boys, Les (1997)', 5.5),
 (u'Celestial Clockwork (1994)', 5.5), (u'The Deadly Cure (1996)', 5.
5), (u'Wings of Courage (1995)', 5.5), (u'Mina Tannenbaum (1994)', 5.
4), (u'Switchback (1997)', 5.333333333333333), (u'Hearts and Minds (199
6)', 5.333333333333333), (u'Crossfire (1947)', 5.333333333333333), (u'A
ngel Baby (1995)', 5.333333333333333), (u'World of Apu, The (Apur Sansa
r) (1959)', 5.333333333333333), (u'Kaspar Hauser (1993)', 5.28571428571
4286), (u'Wings of Desire (1987)', 5.276595744680851), (u'Nico Icon (19
95)', 5.25), (u'Close Shave, A (1995)', 5.25), (u'Wrong Trousers, The
 (1993)', 5.247863247863248), (u'Shall We Dance? (1996)', 5.20833333333
3333), (u'Saint of Fort Washington, The (1993)', 5.2), (u'Wallace & Gro
mit: The Best of Aardman Animation (1996)', 5.17910447761194), (u'Manon
of the Spring (Manon des sources) (1986)', 5.162790697674419), (u'Kundu
n (1997)', 5.153846153846154), (u'Casablanca (1942)', 5.11979166666666
7), (u'Paradise Lost: The Child Murders at Robin Hood Hills (1996)', 5.
117647058823529), (u'Jean de Florette (1986)', 5.113636363636363), (u'I
nspector General, The (1949)', 5.111111111111111), (u'Star Wars (197
7)', 5.043927648578811), (u'My Man Godfrey (1936)', 5.037037037037037),
(u'Rebecca (1940)', 5.035087719298246), (u'Promesse, La (1996)', 5.0),
 (u'Stranger, The (1994)', 5.0), (u'Cement Garden, The (1993)', 5.0),
 (u'Wife, The (1995)', 5.0), (u'Welcome To Sarajevo (1997)', 5.0), (u'N
aked in New York (1994)', 5.0), (u'Intimate Relations (1996)', 5.0),
 (u'Of Human Bondage (1934)', 5.0), (u'Ridicule (1996)', 5.0), (u'Windo
w to Paris (1994)', 5.0), (u'Farmer & Chase (1995)', 5.0), (u'Faces (19
68)', 5.0)]
```