Chapter 2- Recommender Systems

Samuel Naser ¶

Basic techniques in recommendation

- Manhattan Distance (aka Taxi Cab Distance)
- Euclidean Distance

Example Distances

In [17]:

```
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}}
```

In [18]:
```
#implementation for manhattan distance
def manhattanDistance(user1, user2):
    manhattanDistance = 0
    for band in user1:
        if band in user2:
            difference = user1[band] - user2[band]
            manhattanDistance += abs(difference)
    return manhattanDistance

#implementation for euclidean distance
def euclideanDistance(user1, user2):
    euclideanDistance = 0
    for band in user1:
        if band in user2:
            difference = user1[band] - user2[band]
            euclideanDistance += difference ** 2
    return round(euclideanDistance ** (0.5), 2)


print("Manhattan Distance between Angelica and Chan:",
manhattanDistance(users["Angelica"], users["Chan"]))
print("Euclidean Distance between Angelica and Chan:",
euclideanDistance(users["Angelica"], users["Chan"]))
```

```
Manhattan Distance between Angelica and Chan: 4.5
Euclidean Distance between Angelica and Chan: 2.4
```

NOW COMPUTE CLOSEST MATCH FOR USER!

In [35]:
```
def nearestMatches(username, allUserData):
    allDistances = []
    for currentUser in allUserData:
        if currentUser != username:
            currentDistance = manhattanDistance(allUserData[username], a
llUserData[currentUser])
            allDistances.append((currentDistance, currentUser))
    allDistances.sort()
    return allDistances

print("All distances for user Veronica:", nearestMatches("Veronica", use
rs))
```

```
All distances for user Veronica: [(2.0, 'Hailey'), (3.5, 'Angelica'),
 (4.0, 'Bill'), (4.0, 'Dan'), (4.0, 'Jordyn'), (6.5, 'Chan'), (8.5, 'Sa
m')]
```

In [53]:
```
def makeReccomendations(username, allUserData):
    recs = []
    nearestMatch = nearestMatches(username,allUserData)[0][1]
    nearestMatchRatings = allUserData[nearestMatch]
    userRatings = allUserData[username]
    for band in nearestMatchRatings:
        if not band in userRatings:
            recs.append((band, nearestMatchRatings[band]))
    return sorted(recs, key=lambda tup: tup[1], reverse=True) #don't ful
ly understand this line

print(makeReccomendations("Veronica", users))
```

```
[('Broken Bells', 4.0), ('Deadmau5', 1.0), ('Vampire Weekend', 1.0)]
```

ALGORITHM, AS IT IS, WILL NOT RECOMMEND USERS BANDS IF THEIR CLOSEST MATCH HAS IS SUPERSET OF THEIR PREFERENCES

Assignment: Implement Minkowski Distance

In [55]:
```
def minkowskiDistance(user1, user2, r):
    minkowskiDistance = 0
    for band in user1:
        if band in user2:
            difference = user1[band] - user2[band]
            minkowskiDistance += difference ** r
    return round(minkowskiDistance ** (1.0/r), 2)

print("Minkowski Distance between Angelica and Chan:",
minkowskiDistance(users["Angelica"], users["Chan"], 3))
```

```
Minkowski Distance between Angelica and Chan: 1.04
```

How to normalize user preferences?

Some users WAY more willing to give high ratings, other users are reluctant to drop a 5.

In data mining, we call this **GRADE INFLATION**

Pearson correlation coefficient to address this problem

Takes into account difference in user rating patterns.

In [7]:
```
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}}
```

In [16]:
```
from math import sqrt
def pearsonCorrelation(r1, r2):
    sxy = 0
    sx = 0
    sy = 0
    sx2 = 0
    sy2 = 0
    n = 0
    for key in r1:
        if key in r2:
            n += 1
            x = r1[key]
            y = r2[key]
            sxy += x * y
            sx += x
            sy += y
            sx2 += x ** 2
            sy2 += y ** 2
    #if the denominator is zero, might as well return 0
    lower = sqrt(sx2 - sx ** 2 / n) * sqrt(sy2 - sy ** 2 / n)
    if lower == 0:
        return 0
    else:
        return (sxy - (sx * sy) / n) / lower

print("Pearson correlation between Angelican and Chan:", round(pearsonCo
rrelation(users["Angelica"], users["Chan"]),2))
```

```
Pearson correlation between Angelican and Chan: 0.82
```

Cosine Similarity

Used more frequently in text mining, but also used in collaborative filtering

In some cases, like text mining, shared *zeroes* do not indicate similarity (e.g. just because two things do **NOT** include word doesn't mean they are similar)

In [20]:
```
#my cosine impl
def cosine(r1, r2):
    #calcaulate magnitudes of vectors
    m1 = sum([r1[key] ** 2 for key in r1]) ** 0.5
    m2 = sum([r2[key] ** 2 for key in r2]) ** 0.5
    #iterate over one vector checking for values for same key in other
    #NOTE: non-shared keys do not contribute to dot product
    dotProduct = 0
    for key in r1:
        if key in r2:
            dotProduct += r1[key] * r2[key]
    return dotProduct / (m1 * m2)

print(cosine(users["Angelica"], users["Chan"]))
```

```
0.8784261605942703
```

HUGE POINT: WHEN TO USE WHICH METHOD??

- With sparse data, where most values are zero, use Cosine similarity
- With dense data, where most data has values, and magnitude matters, use Manhattan or Euclidean distance
- With data subject to users rating on different scales, use a correlation value (Pearson coefficient)

Cannot just add zeroes to make sparse datasets work with Manhattan/Euclidean! The zeroes tend to dominate any calculations (you really just shouldn't take into account songs they haven't both rated).

- One possible workaround is to take a pseudo-average by finding the distance and then dividing by number of terms in common. This would, as you could see, result in lower distance (closer match) with *more* ratings in common

K-Nearest Neighbors

Takes out some of the quirks of basing recommendations of one (most) similar person

Use top k neighbors, and each neighbor influences the predictions by a factor of their similarity match to the user in question

In [99]:
```
import csv
from enum import Enum

class Metric(Enum):
    PEARSON = 0
```

```
    COSINE = 1
    EUCLIDEAN = 2
    MANHATTAN = 3

class Recommender:
    def __init__(self, _data, _metric):
        self.data = _data

        #assign appropriate correlation method
        if _metric == Metric.PEARSON:
            self.similarityMethod = self.__pearsonCorrelation
        if _metric == Metric.COSINE:
            self.similarityMethod = self.__cosine
        if _metric == Metric.EUCLIDEAN:
            self.similarityMethod = self.__euclidean
        if _metric == Metric.MANHATTAN:
            self.similarityMethod = self.__manhattan

    def recommend(self, user, amount):

        recommendations = {}
        #retrieve 3 nearest neighbors
        nearestUsers = self.kNearestNeighbors(user,3)
        userRatings = self.data[user]

        aggregateDistance = 0.0
        for user in nearestUsers:
            aggregateDistance += user[1]

        for user in nearestUsers:
            currentUsername = user[0]
            currentWeight = user[1] / aggregateDistance
            currentUserRatings = self.data[currentUsername]

            for band in currentUserRatings:
                if not band in userRatings:
                    if not band in recommendations:
                        recommendations[band] = currentUserRatings[band]
 * currentWeight
                    else:
                        recommendations[band] = recommendations[band] +
currentUserRatings[band] * currentWeight

        recommendations = list(recommendations.items())
        recommendations.sort(key=lambda x: x[1], reverse = True)

        return recommendations[:amount]

    def similarity(self, u1, u2):
        return self.similarityMethod(self.data[u1], self.data[u2])

    #method accepts username
    def kNearestNeighbors(self, user, k):
        similarities = []
        for currentUsername in self.data:
            if not currentUsername == user:
                similarities.append((currentUsername, self.similarity(us
```

```
er, currentUsername)))
        similarities = sorted(similarities, key = lambda x: x[1])
        return similarities[:k]

    #definitions of similarity measures
    #each takes two user rating dictionaries, and returns similarity
    #lower = more similar

    def __cosine(self, r1, r2):
        #calcaulate magnitudes of vectors
        m1 = sum([r1[key] ** 2 for key in r1]) ** 0.5
        m2 = sum([r2[key] ** 2 for key in r2]) ** 0.5
        #iterate over one vector checking for values for same key in oth
er
        #NOTE: non-shared keys do not contribute to dot product
        dotProduct = 0
        for key in r1:
            if key in r2:
                dotProduct += r1[key] * r2[key]
        return dotProduct / (m1 * m2)

    def __pearsonCorrelation(self, r1, r2):
        sxy = 0
        sx = 0
        sy = 0
        sx2 = 0
        sy2 = 0
        n = 0
        for key in r1:
            if key in r2:
                n += 1
                x = r1[key]
                y = r2[key]
                sxy += x * y
                sx += x
                sy += y
                sx2 += x ** 2
                sy2 += y ** 2
        #if the denominator is zero, might as well return 0
        lower = sqrt(sx2 - sx ** 2 / n) * sqrt(sy2 - sy ** 2 / n)
        if lower == 0:
            return 0
        else:
            return abs((sxy - (sx * sy) / n) / lower)

    #implementation for manhattan distance
    def __manhattanDistance(self, user1, user2):
        manhattanDistance = 0
        for band in user1:
            if band in user2:
                difference = user1[band] - user2[band]
                manhattanDistance += abs(difference)
        return manhattanDistance

    #implementation for euclidean distance
    def __euclideanDistance(self, user1, user2):
        euclideanDistance = 0
```

```
        for band in user1:
            if band in user2:
                difference = user1[band] - user2[band]
                euclideanDistance += difference ** 2
        return round(euclideanDistance ** (0.5), 2)

#main section
movie_file_rows = []
with open("movie_ratings.csv", 'r+') as movies:
    reader = csv.reader(movies)
    for rows in reader:
        movie_file_rows.append(rows)

#let's parse this data
userData = {}

movie_count = len(movie_file_rows) - 1
user_count = len(movie_file_rows[0]) - 1

#get list of usernames
usernames = []
for i in range(user_count):
    usernames.append(movie_file_rows[0][i+1])

#get list of movie names
movie_names = []
for i in range(movie_count):
    movie_names.append(movie_file_rows[i+1][0])

#set each entry to zero, at first
for username in usernames:
    userData[username] = {}

#fill matrix with data
dataMatrix = []
for i in range(movie_count):
    currentMovieRatings = []
    for j in range(user_count):
        currentMovieRatings.append(movie_file_rows[i+1][j+1])
    dataMatrix.append(currentMovieRatings)

for movieIndex in range(movie_count):
    currentMovie = movie_names[movieIndex]
    for userIndex in range(user_count):
        currentValue = dataMatrix[movieIndex][userIndex]
        if not currentValue == '':
            userData[usernames[userIndex]][movie_names[movieIndex]] = in
t(currentValue)

rcmd = Recommender(userData, Metric.PEARSON)
recommendation = rcmd.recommend("Bryan", 2)

print("RECOMMENDATIONS FOR BRYAN: ", recommendation)
```

```
RECOMMENDATIONS FOR BRYAN:  [('Scarface', 4.3182870505659885), ('Blade
 Runner', 3.4269113897024672)]
```