Classification Based on Item Attributes

- Collaborative filtering has "rich get richer" effect, neglecting to recognize newer bands
- Music Genome Project broke songs down into quantitive attributes
- For example: might have "Genre" attribute with 1 = Rock, 2 = Pop, etc..

Flaw with our usual methods: **DISTANCE MEANS NOTHING WITH CATEGORICAL DATA!** Instead, we have to **split the categories out** and put them on numerical scale (e.g. 2/5 Rock, 4/5 Rap)

Once we have done this, we can find **distance** between any two songs with normal distance methods (Manhattan, Cosine sim., etc.)- this is effectively "rating" the items themselves across a series of attributes

We also gain the ability to **EXPLAIN** our recommendations! The attributes (vocal style, keyboard intensity, etc.) which are closest can be said to "explain" the recommendation.

Problem of Scale

Certain variables exist on different **scales**- for example: $100,000 net worth vs 3 cars owned. The net worth would dominate our distance calculations

How can we fix this tremendous problem??

Normalization!

One common method of normalization is **bringing data between 0 and 1**. Formula looks like this:

$$\frac{x - Min_x}{Max_x - Min_x}$$

This method, however, is **NAIVE!** Ok, maybe it's appropriate sometimes, but **STANDARD SCORE (Z-SCORE)** IS BETTER! and the formula is **SHOWN BELOW** (denominator is std. deviation)

$$Z_s = \frac{x - \bar{x}}{\sqrt{\frac{\sum x_i - \bar{x}^2}{card(x)}}}$$

Outliers often throw off standard deviation, so we use **modified standard score**

$$asd = \frac{1}{card(x)} \sum_i |x_i - median|$$

Modified standard score is **(EACH VALUE - MEDIAN) / (Absolute Standard Deviation)**

When to normalize?

Should normalize when:

- **Method calculates distance based on values of their features**
- **Scale of the different features is different**

TRADEOFFS: Normalization isn't always necessary, and in fact sometimes *reduces* accuracy. There is also a computational cost involved with normalization to consider.

Back to Pandora! :D

"Likes" and "Dislikes" can oftentimes group themselves along parameters. For example: if we plot "driving beat" against "dirty guitar" (1-5 scale), we may find the Likes and Dislikes cluster together.

Simplest method is to assume mystery class (like vs dislike) will be same as nearest neighbor! This is arguably the most rudimentary form of classification.

A classifier is a program that uses an object's attributes to figure out which class it belongs to!

Possible applications (note: classification we have done so far is item-based):

- **Twitter sentiment classification**
- **Automatic identification of people in photographs**
- **Targeted political ads (classifying people into demographics)**
- **Targeted marketing (likely buyers)**
- **Health and the Quantified Self**
- **Terrorist vs Non-Terrorist (these algorithms could use some work since I get pulled aside every flight I take)**

  Modified standard scores are commonly applied devices in classification.

In [1]:
```
#my implementation of nearest neighbor classifier
class Classifier(object):

    def __init__(self, inputFile, nMethod):
        self.nMethod = nMethod
        self.medianDeviation = []
        self.minMax = []
        f = open(inputFile)
        lines = f.readlines()
        f.close()
```

```
        self.format = lines[0].strip().split('\t')
        self.data = []
        for line in lines[1:]:
            fields = line.strip().split('\t')
            ignore = []
            vector = []
            for i in range(len(fields)):
                if self.format[i] == 'num':
                    vector.append(int(fields[i]))
                elif self.format[i] == 'comment':
                    ignore.append(fields[i])
                elif self.format[i] == 'class':
                    classification = fields[i]
            self.data.append((classification, vector, ignore))

        self.vectorLength = len(self.data[0][1])
        for v in range(self.vectorLength):
            self.normalizeColumn(v)

    def median(self, attribs):
        if attribs == []:
            return None

        attribs = sorted(attribs)
        if len(attribs) % 2 == 1:
            return attribs[(len(attribs)+1)/2 - 1]
        else:
            return float(sum(attribs[(len(attribs)/2)-1: len(attribs)/2
+1]))/2

    def absStandardDeviation(self, attribs, median):
        total = 0.0
        for x in attribs:
            total += abs(x - median)

        return total / len(attribs)

    def normalizeColumn(self, columnIndex):
        columnValues = []
        columnValues = [v[1][columnIndex] for v in self.data]

        median = self.median(columnValues)
        asd = self.absStandardDeviation(columnValues, median)
        mx = max(columnValues)
        mn = min(columnValues)
        #we store this so we can convert any NEW data to its standard-sc
ore equivalent
        self.medianDeviation.append((median, asd))
        self.minMax.append((mn,mx))
        for v in self.data:
            if self.nMethod == 'asd':
                v[1][columnIndex] = self.normalize(v[1][columnIndex], me
dian, asd)
            elif self.nMethod == 'range':
                v[1][columnIndex] = self.boring(v[1][columnIndex], mn, m
x)
```

```
    def normalize(self, value, median, asd):
        return (value - median) / asd

    def boring(self, v, mn, mx):
        return float((v - mn))/(mx - mn)

    def normalizeInputVector(self, vec):
        vector = list(vec)
        for i in xrange(len(vec)):
            (med, asd) = self.medianDeviation[i]
            vector[i] = (vector[i] - med) / asd
        return vector

    def boringNormalizeVector(self, vec):
        vector = list(vec)
        for i in xrange(len(vec)):
            vector[i] = float((vector[i] - self.minMax[i][0])) / (self.m
inMax[i][1] - self.minMax[i][0])
        return vector


    def manhattan(self, vec1, vec2):
        sum = 0.0
        for i in xrange(len(vec1)):
            sum += abs(vec1[i] - vec2[i])
        return sum

    def nearestNeighbor(self, vec1):
        distances = []
        for items in self.data:
            distances.append((self.manhattan(vec1, items[1]), items[0]))
        return sorted(distances)[0]

    def classify(self, vec):
        return self.nearestNeighbor(vec)

    def testDataFromFile(self, testPath):
        correct = 0.0
        incorrect = 0.0

        f = open(testPath, 'r')
        lines = [x.strip() for x in f.readlines()]
        f.close()

        for athlete in lines:
            s = athlete.split('\t')
            currentClass = s[1]
            cVec = [int(s[2]),int(s[3])]
            if self.nMethod == 'asd':
                ncVec = self.normalizeInputVector(cVec)
            elif self.nMethod == 'range':
                ncVec = self.boringNormalizeVector(cVec)
            prediction = self.classify(ncVec)
            if prediction[1] == currentClass:
                correct += 1
            else:
                incorrect += 1
```

```
        print "PERCENT ACCURACY:", (correct / (correct +
incorrect))*100, " WITH METHOD", self.nMethod

c = Classifier("athletesTrainingSet.txt", "range")
c.testDataFromFile("athletesTestSet.txt")

d = Classifier("athletesTrainingSet.txt", "asd")
d.testDataFromFile("athletesTestSet.txt")

#I cannot believe how long this code took to write.
```

```
PERCENT ACCURACY: 80.0  WITH METHOD range
PERCENT ACCURACY: 80.0  WITH METHOD asd
```

assert keyword useful in Python to test code and ensure certain conditions are met at runtime.

> **“it is important that each part of the specification be turned into a piece of code that implements it and a test that tests it. If you don’t have tests like these then you don’t know when you are done, you don’t know if you got it right, and you don’t know that any future changes might be breaking something.” - Peter Norvig**

Normalization: transforming data from its original scale to 0-1

Standardization: transforming data so that 0 = average, and either side are proportions of std dev from average