Discovering Groups

In previous sections, our datapoints have been labeled. What about when we **don't** have these labels? What happens if I want to discover **possible groups?**

k-means clustering is clustering where we tell the algorithm *how many clusters to make*. For example: "*cluster these people into 5 groups*"

Hierarchial clustering

In this approach, we don't tell the algorithm how many clusters to make.

This algorithm starts **with each instance in its own cluster**. Each iteration of the algorithm **combines the two most similar clusters into one**. Repeats until there is only **one cluster**.

This is called **hierarchial clustering.** The algorithm results in **one huge cluster** with one **sub-cluster**. Each subcluster has its own subcluster, etc., etc.

Remember, **AT EACH ITERATION, WE JOIN THE TWO NEAREST CLUSTERS!**

Single-linkage clustering

In this method, we define the distance between clusters as the **shortest distance between any member of one cluster to any member of the other.**

Complete-linkage clustering

In this method, we define the distance between two clusters as **the greatest distance between any member of one cluster to any member of the other**.

Average-linkage clustering

In average-linkage clustering we define the distance between two clusters as **the average distance between any member of one cluster to any member of another**.

Coding hierarchial clustering algorithm

In [18]:
```
#example of how priority queues work
from Queue import PriorityQueue

#queue that spits out elements based not on order in our out
#but rather, some parameter attached with the data
sQueue = PriorityQueue()
sQueue.put((15, "Kati Piri"))
sQueue.put((20, "Perseporino"))
sQueue.put((10, "Enrico Vittadini"))

#retrieves elements with the first element of the tuple as priority
s1 = sQueue.get()
s2 = sQueue.get()
s3 = sQueue.get()

print s1
print s2
print s3
```

```
(10, 'Enrico Vittadini')
(15, 'Kati Piri')
(20, 'Perseporino')
```

In [26]:
```
#below is an implementation of hierarchial clustering
import math
from Queue import PriorityQueue

#UTILITY FUNCS
def median(l):
    temp = list(l)
    temp.sort()
    length = len(temp)
    if (length % 2 == 1):
        return temp[int(length / 2)]
    else:
        return (temp[int(length / 2)] + temp[int(length / 2) - 1]) / 2

#all the columns are bound to be on different scales-
#normalize all that jazz!
def normalizeCol(col):
    med = median(col)
    adjSd = sum([abs(x - med) for x in col]) / len(col)
    return [(x - med) / adjSd for x in col]

class HierarchialClusterer:

    #accept a datafile to initialize the clusterer
    def __init__(self, dataFile):
        f = open(dataFile)
        self.data = {}
        self.counter = 0
        self.queue = PriorityQueue()
        lines = f.readlines()
        f.close()

        header = lines[0].split(',')
        self.cols = len(header)
        self.data = [[] for i in range(self.cols)]

        #exclude the first line w/ list splice magic!
        for line in lines[1:]:
            cells = line.split(',')
            t = 0
            for cell in range(self.cols):
                if t == 0:
                    self.data[cell].append(cells[cell])
                    t = 1
                else:
                    self.data[cell].append(float(cells[cell]))

        #GOTTA NORMALIZE THE DATA WITH THAT nifty FUNCTION WE WROTE!
        for i in range(1, self.cols):
            self.data[i] = normalizeCol(self.data[i])

        #Here we go. The described algorithm. This shall be brutal.
        #The major steps:
        #1) Calculate Euclidean Distance from i to each other element
        #store the result in 'neighbors', a dict.
        #2) Find nearest neighbor
        #3) Place on queue

        rows = len(self.data[0])

        for i in range(rows):
            minDist = 99999
            nn = 0
            neighbors = {}
            for j in range(rows):
                if i != j:
                    d = self.distance(i,j)
                    if i < j:
                        pair = (i,j)
                    else:
                        pair = (j,i)

                    #set j'th element of neighbors to dist and pair
                    neighbors[j] = (pair, d)

                    if d < minDist:
                        minDist = d
                        nn = j
                        nearestNum = j

            if i < nn:
                nearestPair = (i, nn)
            else:
                nearestPair = (nn, i)

            self.queue.put((minDist, self.counter, [[self.data[0][i]], n
earestPair, neighbors]))
            self.counter += 1

    def distance(self, i, j):
        ss = 0
        for k in range(1, self.cols):
            ss += (self.data[k][i] - self.data[k][j]) ** 2
        return math.sqrt(ss)

    def cluster(self):
        finished = False
        while not finished:
            top1 = self.queue.get()
            nearestPair = top1[2][1]
            if not self.queue.empty():
                next1 = self.queue.get()
                nearPair = next1[2][1]
                tmp = []

                #so obviously, if the closest distance is from i to j,
                #the next closest should be the same, from j to i
                #although there could be overlaps with other pairs
                #so go ahead and pop off until we hit a duplicate

                while nearPair != nearestPair:
                    tmp.append((next1[0], self.counter, next1[2]))
                    self.counter += 1
                    next1 = self.queue.get()
                    nearPair = next1[2][1]

                #clean up the mess we made :D
                for item in tmp:
                    self.queue.put(item)

                if len(top1[2][0]) == 1:
                    item1 = top1[2][0][0]
                else:
                    item1 = top1[2][0]

                if len(next1[2][0]) == 1:
                    item2 = next1[2][0][0]
                else:
                    item2 = next1[2][0]

                currentCluster = (item1, item2)

                #now find NN for this cluster, and build new neighbors l
ist

                minDist = 99999
                nearestPair = ()
                nearestNeighbor = ''
                merged = {}
                nNeighbors = next1[2][2]
                for(key, value) in top1[2][2].items():
                    if key in nNeighbors:
                        if nNeighbors[key][1] < value[1]:
                            dist = nNeighbors[key]
                        else:
                            dist = value
                        if dist[1] < minDist:
                            minDist = dist[1]
                            nearestPair = dist[0]
                            nearestNeighbor = key
                        #set merged element to current dist
                        merged[key] = dist

                #if empty
                if merged == {}:
                    return currentCluster
                else:
                    self.queue.put((minDist, self.counter, [currentClust
er, nearestPair, merged]))
                    self.counter += 1

fName = 'dogs.csv'
clusterer = HierarchialClusterer(fName)
cluster = clusterer.cluster()

print 'CLUSTERED GROUPINGS\n-=-=-=-=-=-=-=-=-=-=-=-=-=-'
print cluster
```

```
CLUSTERED GROUPINGS
-=-=-=-=-=-=-=-=-=-=-=-=-
(('Chihuahua', 'Yorkshire Terrier'), ('Great Dane', ('Bullmastiff',
 (('German Shepherd', 'Golden Retriever'), ('Standard Poodle', ('Boston
Terrier', ('Brittany Spaniel', ('Border Collie', 'Portuguese Water Do
g'))))))))
```

k-Means Clustering

k-Means is **the most popular** clustering algorithm.

Logical and intuitive process, the steps are as such:

1. **Step 1:** select k random instances to be the initial centroid
2. **Step 2:** REPEAT
3. **Step 3:** assign each instance to the nearest centroid, which preserves the amount of clusters
4. **Step 4:** update centroids by computing mean of each cluster
5. **Step 5:** UNTIL centroids don't change (significantly)

Algorithm is said to **converge** when the points in each group stop shifting around.

Possible to relax our criteria of *stop when no points shift* to *stop when less than 1% of points shift*, and maintain solid results.

K-MEANS is an instance of Expectation-Maximization (EM), which is a method that has two phases.

- **E:** use estimate to place points into their expected cluster
- **M:** use these expected values to adjust the estimate of the centroids

SSE or Scatter

We use the **sum of the squared error** to determine the quality of a set of clusters (we also call this **scatter**)

For each point, square the distance from that point to its centroid, then add those squared distances together.

$$\sum_{i=1}^{k} \sum_{x \in C_i} dist(c_i, x)^2$$

All this means is "iterate over the clusters, and sum up the squared differences from each point to the current cluster".

This metric lets us know how well our data is clustered. Smaller SSE means the clusters are better.

In [48]:
```
import math
import random

#UTILITY FUNCS, SAME AS BEFORE

def median(l):
    temp = list(l)
    temp.sort()
    length = len(temp)
    if (length % 2 == 1):
        return temp[int(length / 2)]
    else:
        return (temp[int(length / 2)] + temp[int(length / 2) - 1]) / 2

#all the columns are bound to be on different scales-
#normalize all that jazz!
def normalizeCol(col):
    med = median(col)
    adjSd = sum([abs(x - med) for x in col]) / len(col)
    return [(x - med) / adjSd for x in col]

class kMeansClusterer:
    #k-means clustering class

    def __init__(self, dataFile, k):
        f = open(dataFile)

        #init members
        self.data = {}
        self.k = k
        self.counter = 0
        self.iterationNumber = 0
        self.pointsChanged = 0
        self.sse = 0

        lines = f.readlines()
        f.close()

        header = lines[0].split(',')
        self.cols = len(header)
        self.data = [[] for i in range(self.cols)]

        #read file into data dict
        for line in lines[1:]:
            cells = line.split(',')
            t = 0
            for cell in range(self.cols):
                if t == 0:
                    self.data[cell].append(cells[cell])
                    t = 1
                else:
                    self.data[cell].append(float(cells[cell]))

        self.dSize = len(self.data[1])
        self.memberOf = [-1 for x in range(len(self.data[1]))]
        #normalize
        for i in range(1,self.cols):
            self.data[i] = normalizeCol(self.data[i])

        #seed randomizer and pick initial centroid
        random.seed()
        self.centroids = [[self.data[i][r] for i in range(1,len(self.dat
a))] for r in random.sample(range(len(self.data[0])),self.k)]
        self.assignPointsToCluster()

    def assignPointToCluster(self, i):
        minimum = 999999
        clusterNumber = -1
        #iterate over the clusters finding distance
        for centroid in range(self.k):
            dist = self.euclideanDistance(i, centroid)
            if dist < minimum:
                minimum = dist
                clusterNumber = centroid
        #changed points
        if clusterNumber != self.memberOf[i]:
            self.pointsChanged += 1

        self.sse = self.sse + minimum ** 2
        return clusterNumber

    def updateCentroids(self):
        members = [self.memberOf.count(i) for i in range(len(self.centro
ids))]
        self.centroids = [[sum([self.data[k][i] for i in
range(len(self.data[0])) if self.memberOf[i] == centroid])/members[centr
oid] for k in range(1,len(self.data))] for centroid in range(len(self.ce
ntroids))]

    def assignPointsToCluster(self):
        self.pointsChanged =0
        self.sse =0
        self.memberOf = [self.assignPointToCluster(i) for i in
range(len(self.data[1]))]

    def euclideanDistance(self, i, j):
        #comp dist from point i to the centroid j
        ss = 0
        for k in range(1, self.cols):
            ss += (self.data[k][i] - self.centroids[j][k-1]) ** 2
        return math.sqrt(ss)

    #actually cluster
    def cluster(self):
        done = False
        while not done:
            self.iterationNumber += 1
            self.updateCentroids()
            self.assignPointsToCluster()
            if float(self.pointsChanged) / len(self.memberOf) < 0.01:
                done = True
    def present(self):
        for centroid in range(len(self.centroids)):
            print "\n\nClass %i\n-=-=-=-=-" % (centroid + 1)
            for name in [self.data[0][i] for i in
range(len(self.data[0])) if self.memberOf[i] == centroid]:
                print name

klustahs = kMeansClusterer('dogs.csv', 3)
klustahs.cluster()
klustahs.present()
```

```
Class 1
-=-=-=-=-
Chihuahua
Yorkshire Terrier


Class 2
-=-=-=-=-
Bullmastiff
German Shepherd
Great Dane


Class 3
-=-=-=-=-
Border Collie
Boston Terrier
Brittany Spaniel
Golden Retriever
Portuguese Water Dog
Standard Poodle
```

k-means++

The major weakness of the original k-means algorithm is that it **randomly** picks *k* datapoints to be the initial centroids.

This randomness means that sometimes the initial centroids are great picks and lead to optimal clusters, and sometimes not.

This method **changes the way we choose our initial clusters**, and it works as follows:

1. Empty set of initial centroids
2. Select first centroid **randomly** from datapoints as before
3. Repeat until we have k initial centroids
    - **a)** Compute distance between each datapoint and nearest centroid.
    - **b)** In a probability with proportional to the distance, select one datapoint at random to be a new centroid and add it to set of centroids
    - **c)** Repeat!

The main idea is this: **whlie we still pick the initial centroids randomly, we prefer centroids that are far away from one another**

Recap

Clustering is about **discovery!**

Examples:

- Useful to cluster search results!
- Marketing teams might cluster users into demographics and target ads to each cluster

When to use k-means over hierarchial clustering?

- **k-means:** simple and fast algorithm. Perfect first-step method to identify features of data.
- **hierarchial:** when we want to create a taxonomy or hierarchy in our data. not as memory efficient as our other method.

The ENRON DATASET HAS SOME COOL PATTERNS! Over 600,000 emails were leaked after the Enron scandal, and they comprise a famous dataset.