def kmeans(points, pointType, k):
    #Get k randomly chosen initial centroids
    initialCentroids = random.sample(points, k)
    clusters = []
    #Create a singleton cluster for each centroid
    for p in initialCentroids:
        clusters.append(Cluster([p], pointType))
    numIters = 0
    biggestChange = 1.0 #anything bigger than 0 is ok
    while biggestChange > 0.0:
        #Create a list containing k empty lists
        newClusters = []
        for i in range(k):
            newClusters.append([])
        for p in points:
            #Find the centroid closest to p
            smallestDistance = p.distance(clusters[0].getCentroid())
            index = 0
            for i in range(1,k):
                distance = p.distance(clusters[i].getCentroid())
                if distance < smallestDistance:
                    smallestDistance = distance
                    index = i
            #Add p to the list of points for the appropriate cluster
            newClusters[index].append(p)
        #Update each cluster; record how much the centroid has changed
        biggestChange = 0.0
        for i in range(len(clusters)):
            change = clusters[i].update(newClusters[i])
            biggestChange = max(biggestChange, change)
        numIters += 1
        #Calculate the coherence of the least coherent cluster
        maxDist = 0.0
        for c in clusters:
            totDist = 0.0
            numPts = 0.0
            for p in c.members():
                totDist = p.distance(c.getCentroid())
                numPts += 1
            if numPts > 0:
                aveDist = totDist/numPts
                if aveDist > maxDist:
                    maxDist = aveDist
        return clusters, maxDist
```python
def tryKMeans(points, pointType, numClusters, numTrials):
    for trial in range(numTrials):
        clusters, maxDist = kmeans(points, pointType, numClusters)
        #minimize the maximum distance
        if trial == 0:
            #clusters[0] is set of clusters
            bestClustering = clusters
            #clusters[1] is maximum variance
            minDistance = maxDist
        else:
            if maxDist < minDistance:
                minDistance = maxDist
                bestClustering = clusters
    return bestClustering, minDistance

P000:[ 89.  1.  0.  66.]:1
P001:[ 59.  0.  0.  72.]:0
P002:[ 73.  0.  0.  73.]:0
P003:[ 56.  1.  0.  65.]:0
P004:[ 75.  1.  1.  68.]:1
P005:[ 68.  1.  0.  56.]:0
P006:[ 73.  1.  0.  75.]:1
P007:[ 72.  0.  0.  65.]:0
P008:[ 73.  1.  0.  64.]:1
P009:[ 73.  0.  0.  58.]:0
P010:[ 100.  0.  0.  75.]:0
P011:[ 79.  0.  0.  31.]:0
P012:[ 81.  0.  0.  58.]:0
P013:[ 89.  1.  0.  50.]:1
P014:[ 81.  0.  0.  70.]:0

def scaleAttrs(vals):
    vals = pylab.array(vals)
    mean = sum(vals)/float(len(vals))
    sd = stdDev(vals)
    vals = vals - mean
    return vals/sd

def testClustering(numPatients, numClusters, scale = False):
    numTrials = 5
    patients, positives = genData(numPatients, scale)
    fracPos = len(positives)/float(numPatients)
    print 'Fraction of positives =', fracPos
    bestClustering, maxDist = tryKMeans(patients, Patient, numClusters, numTrials)
    printClustering(bestClustering)
```

```