6.00 Quiz 3, Spring 2012 ----------

Name

1. _____/15
2. _____/10
3. _____/15
4. _____/5
5. _____/15
6. _____/15
7. _____/15
8. _____/10

Total _____ /100

---

Athena User Name

---

Recitation hour

This quiz is open book and open notes, but do not use a computer (or cell phone!). You have 120 minutes.

Please **write your name on the top of each page**, and your user name and the hour of the recitation you attend on the first page. Answer all questions in the boxes provided.

1) Are each of the following True or False? (15 points)

| | |
|---|---|
| T | 1.1. Agglomerative hierarchical clustering is $O(n^3)$, where n is the number of data points. |
| F | 1.2. Dynamic programming can be used to reduce the order of algorithmic complexity of sorting a list of integers to something **below** n log n, where n is the length of the list to be sorted. |
| T | 1.3. `pylab.polyfit` **cannot** be used to find a fit for the function $f(x) = 5^x$. |
| T | 1.4. Newton's method can be used to find an approximate value of the fourth root of a floating point number. |
| F | 1.5. In Python, instances of classes **cannot** be used as arguments to a function. |

3) Consider the following code.

```
import random, pylab

def throwNeedles(f, xMin, xMax, numNeedles = 10000):
     under = 0.0
     yMax = max(1, f(xMax + 1))
     for x in range(xMin, xMax):
          if f(x) > yMax:
               yMax = f(x)
     for Needles in range(1, numNeedles + 1):
          x = random.choice(range(xMin, xMax + 1))
          if random.choice(range(0, int(round(yMax)))) < f(x):
              under += 1
     return (under/numNeedles)

def performSim(f, numTrials, numNeedles):
    res = []
    for t in range(numTrials):
        res.append(throwNeedles(f, 0, 100))
    mean = sum(res)/float(len(res))
    print mean
    print max(res) - min(res)
    pylab.hist(res)

def f(x):
     return x

performSim(f, 1000, 100)
```

b

3.1. With high probability, the first number printed will be approximately,

a. `0`
b. `0.5`
c. `50`
d. `100`
e. none of the above

a

3.2. Which of the following values is likely to be closest to the second number printed?

a. 0
b. 100
c. the first number printed
d. half the first number printed
e. twice the first number printed

b

3.3. With high probability the histogram generated will depict,

a. a uniform distribution
b. a normal distribution
c. an exponential distribution
d. none of the above

(15 points)

4) Many Course 20 students take 6.00 each term. Over the last five years, Course 20 students taking 6.00 have significantly out performed the rest of the class in the fall terms and significantly under performed the rest of the class in the spring terms. The distribution of grades for the entire class does not differ in the spring and fall terms. Based on these facts, Course 20 advisors have been telling students that they are likely to get a better grade if they choose to take 6.00 in the fall. Does this follow from the evidence? Why or why not? (5 points)

No. From the evidence we cannot say that for sure.
It seems that "taking 6.00 in the fall terms" and "out performed the rest of the class" has some correlation, but correlation does not imply causation.
The observed correlation can be an effect of a common cause.
For example:
maybe in the past years, good students in Course 20 tend to take 6.00 in the fall terms, and good students tend to get good grades.
Then "good students" is the common cause for the correlation.
In this case, "taking 6.00 in the fall terms" and "out performed the rest of the class" are consequences of a common cause, but do not cause each other.
Because the distribution of grades for the entire class does not differ in the spring and fall terms, it is very unlikely that taking 6.00 in the spring term would give Course 20 students any advantage.
So it does not follow from the evidence that choosing to take 6.00 in the fall would let Course 20 get better grades.

6) Next to each item in the left column write the letter labeling the item in the right column that best matches the item in the left column. No item in the right column should be used more than once. (15 points)

| | | |
|---|---|---|
| b | polymorphism | a) fast |
| c | random walk | b) inheritance |
| a | greedy algorithm | c) non-deterministic |
| d | hierarchical clustering | d) deterministic |
| h | training and test sets | e) unit testing |
| | | f) bell curve |
| | | g) supervised learning |
| | | h) avoid over fitting |
| | | i) linear regression |

**The following questions all refer to the code you were asked to study in preparation for this exam. A copy of the posted code is at the end of this quiz. Feel free to detach it.**

7) Give the asymptotic complexity of each of the following functions in the code. Assume that for all the questions `self.dimensionality` = N.

a) `len(self.attrs)` (2 points)

O(1)
(for python built-in list, len() operation has O(1) time cost)

b) `Point.distance(self, other)` (2 points)

O(N)

c) `Cluster.singleLinkageDist(self, other)`
Assume that the number of points in `self.points` is P1 and in `other.points` is P2. (3 points)

O(N * P1 * P2)

d) `Cluster.update(self, points)`
Assume that the number of points in `points` is P. (4 points)

O( max{1, N*P} )
( I think just saying O(N*P) is OK, max{1, N*P} just to cover the corner case when P=0. Note that when P>0, computeCentroid() is the key part for time complexity, and it iterates over P points, each taking O(N) time. )

e) `ClusterSet.findClosest(self, metric)`
Assume that the number of members in `self.members` is M, metric is `singleLinkageDist`, and the maximum number of points in any cluster is P. (4 points)

$O( N M^2 P^2 )$

8) We wish to speed up hierarchical clustering by speeding up the `findClosest()` function.

Write a new function that finds the closest pair of clusters in `self` by randomly comparing R pairs of clusters and choosing the closest pair. Hint: ensure that your code does not compare a cluster to itself. (10 points)

```
# the code below assumes that all the interesting
# metrics are symmetric

import random
R = ... # assume that there is a global constant R

class ClusterSet(object):
  ...
  def findClosest(self, metric):
    M = len(self.members) # M must >= 2
    minDist = -1
    toMerge = None
    for i in xrange(R):
      x = random.randint(0, M-2)    ## (*)
      y = random.randint(x+1, M-1)  ## (*)
      dist = metric(self.members[x], self.members[y])
      if toMerge==None or dist<minDist:
        minDist = dist
        toMerge = (self.members[x], self.members[y])
    return toMerge
  ... # other definitions in class ClusterSet


If the metrics are not all symmetric, we can change
the two lines labeled by (*) to below:
      x = random.randint(0, M-1)
      y = random.randint(0, M-2)
      if y >= x:
        y += 1
```

9) The following questions relate to the code corresponding to `kmeans()`.

(a) Give one possible effect of removing `numIters < maxIters` in the while loop of `kmeans()`. (5 points)

kmeans() might take much longer to finish, because now it only finishes when the clustering converges (biggestChange < cutoff), and the number of iterations is no longer bounded by maxIters.

The time complexity is no longer bounded by a polynomial expression of maxIters, number of points, and the dimension of each point, because now the number of iterations is related to the spread of the data.

(b) Rather than checking to see if the `biggestChange` is >= `cutoff` as in the current `kmeans()`, we checked if the `averageChange` is >= `cutoff`, would the number of iterations increase or decrease? Would the value of `maxDist` increase or decrease at the end of the algorithm? Explain briefly. (5 points)

The number of iterations would decrease.

Because biggestChange is guaranteed to be >= averageChange,
and usually averageChange is less than biggestChange.

So the condition averageChange>=cutoff is usually easier to be violated,
hence the modified loop is usually quicker to finish.

maxDist would increase.
Because the modified loop is usually quicker to finish, the resulting clustering is less close to the "perfect" or "ideal" clustering, thus maxDist would usually increase.

Attached: code for study

```
#Code shared across examples
import pylab, random, string, copy

class Point(object):
    def __init__(self, name, originalAttrs, normalizedAttrs = None):
        """normalizedAttrs and originalAttrs are both arrays"""
        self.name = name
        self.unNormalized = originalAttrs
        if normalizedAttrs == None:
            self.attrs = originalAttrs
        else:
            self.attrs = normalizedAttrs
    def dimensionality(self):
        return len(self.attrs)
    def getAttrs(self):
        return self.attrs
    def getOriginalAttrs(self):
        return self.unNormalized
    def distance(self, other):
        #Euclidean distance metric
        result = 0.0
        for i in range(self.dimensionality()):
            result += (self.attrs[i] - other.attrs[i])**2
        return result**0.5
    def getName(self):
        return self.name
    def toStr(self):
        return self.name + str(self.attrs)
    def __str__(self):
        return self.name

class Cluster(object):
    def __init__(self, points, pointType):
        self.points = points
        self.pointType = pointType
        self.centroid = self.computeCentroid()
    def singleLinkageDist(self, other):
        minDist = self.points[0].distance(other.points[0])
        for p1 in self.points:
            for p2 in other.points:
                if p1.distance(p2) < minDist:
                    minDist = p1.distance(p2)
        return minDist
    def maxLinkageDist(self, other):
        maxDist = self.points[0].distance(other.points[0])
        for p1 in self.points:
            for p2 in other.points:
                if p1.distance(p2) > maxDist:
                    maxDist = p1.distance(p2)
        return maxDist
    def averageLinkageDist(self, other):
        totDist = 0.0
        for p1 in self.points:
            for p2 in other.points:
                totDist += p1.distance(p2)
        return totDist/(len(self.points)*len(other.points))
    def update(self, points):
        oldCentroid = self.centroid
        self.points = points
        if len(points) > 0:
            self.centroid = self.computeCentroid()
            return oldCentroid.distance(self.centroid)
        else:
            return 0.0
    def members(self):
        return self.points[:]
    def isIn(self, name):
        for p in self.points:
            if p.getName() == name:
                return True
        return False
    def toStr(self):
        result = ''
        for p in self.points:
```

```
            result = result + p.toStr() + ', '
        return result[:-2]
    def __str__(self):
        names = []
        for p in self.points:
            names.append(p.getName())
        names.sort()
        result = ''
        for p in names:
            result = result + p + ', '
        return result[:-2]
    def getCentroid(self):
        return self.centroid
    def computeCentroid(self):
        dim = self.points[0].dimensionality()
        totVals = pylab.array([0.0]*dim)
        for p in self.points:
            totVals += p.getAttrs()
        centroid = self.pointType('mean',
                                  totVals/float(len(self.points)),
                                  totVals/float(len(self.points)))
        return centroid

class ClusterSet(object):
    def __init__(self, pointType):
        self.members = []
    def add(self, c):
        if c in self.members:
            raise ValueError
        self.members.append(c)
    def getClusters(self):
        return self.members[:]
    def mergeClusters(self, c1, c2):
        points = []
        for p in c1.members():
            points.append(p)
        for p in c2.members():
            points.append(p)
        newC = Cluster(points, type(p))
        self.members.remove(c1)
        self.members.remove(c2)
        self.add(newC)
        return c1, c2
    def findClosest(self, metric):
        minDistance = metric(self.members[0], self.members[1])
        toMerge = (self.members[0], self.members[1])
        for c1 in self.members:
            for c2 in self.members:
                if c1 == c2:
                    continue
                if metric(c1, c2) < minDistance:
                    minDistance = metric(c1, c2)
                    toMerge = (c1, c2)
        return toMerge
    def mergeOne(self, metric, toPrint = False):
        if len(self.members) == 1:
            return None
        if len(self.members) == 2:
            return self.mergeClusters(self.members[0],
                                      self.members[1])
        toMerge = self.findClosest(metric)
        if toPrint:
            print 'Merged'
            print '  ' + str(toMerge[0])
            print 'with'
            print '  ' + str(toMerge[1])
        self.mergeClusters(toMerge[0], toMerge[1])
        return toMerge
    def mergeN(self, metric, numClusters = 1, history = [],
               toPrint = False):
        assert numClusters >= 1
        while len(self.members) > numClusters:
            merged = self.mergeOne(metric, toPrint)
            history.append(merged)
```

```
        return history
    def numClusters(self):
        return len(self.members) + 1
    def __str__(self):
        result = ''
        for c in self.members:
            result = result + str(c) + '\n'
        return result

#Mammal's teeth example
class Mammal(Point):
    def __init__(self, name, originalAttrs, scaledAttrs = None):
        Point.__init__(self, name, originalAttrs, originalAttrs)
    def scaleFeatures(self, key):
        scaleDict = {'identity': [1,1,1,1,1,1,1,1],
                     '1/max': [1/3.0,1/4.0,1.0,1.0,1/4.0,1/4.0,1/6.0,1/6.0],
                     '1/range': [1/3.0,1/3.0,1.0,1.0,1/4.0,1/4.0,1/5.0,1/5.0]}
        scaledFeatures = []
        features = self.getOriginalAttrs()
        for i in range(len(features)):
            scaledFeatures.append(features[i]*scaleDict[key][i])
        self.attrs = scaledFeatures

def readMammalData(fName):
    dataFile = open(fName, 'r')
    teethList = []
    nameList = []
    for line in dataFile:
        if len(line) == 0 or line[0] == '#':
            continue
        dataLine = string.split(line)
        teeth = dataLine.pop(-1)
        features = []
        for t in teeth:
            features.append(float(t))
        name = ''
        for w in dataLine:
            name = name + w + ' '
        name = name[:-1]
        teethList.append(features)
        nameList.append(name)
    return nameList, teethList

def buildMammalPoints(fName, scaling):
    nameList, featureList = readMammalData(fName)
    points = []
    for i in range(len(nameList)):
        point = Mammal(nameList[i], pylab.array(featureList[i]))
        point.scaleFeatures(scaling)
        points.append(point)
    return points

#Use hierarchical clustering for mammals teeth
def test0(numClusters = 2, scaling = 'identity', printSteps = False,
          printHistory = True):
    points = buildMammalPoints('mammalTeeth.txt', scaling)
    cS = ClusterSet(Mammal)
    for p in points:
        cS.add(Cluster([p], Mammal))
    history = cS.mergeN(Cluster.maxLinkageDist, numClusters,
                        toPrint = printSteps)
    if printHistory:
        print ''
        for i in range(len(history)):
            names1 = []
            for p in history[i][0].members():
                names1.append(p.getName())
            names2 = []
            for p in history[i][1].members():
                names2.append(p.getName())
            print 'Step', i, 'Merged', names1, 'with', names2
            print ''
    clusters = cS.getClusters()
    print 'Final set of clusters:'
```

```
    index = 0
    for c in clusters:
        print '  C' + str(index) + ':', c
        index += 1

def kmeans(points, k, cutoff, pointType, maxIters = 100,
           toPrint = False):
    #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 = cutoff
    while biggestChange >= cutoff and numIters < maxIters:
        #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(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)
        #Upate each cluster and 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:
        for p in c.members():
            if p.distance(c.getCentroid()) > maxDist:
                maxDist = p.distance(c.getCentroid())
    print 'Number of iterations =', numIters, 'Max Diameter =', maxDist
    return clusters, maxDist

def test1(k = 2, cutoff = 0.0001, numTrials = 1, printSteps = False,
          printHistory = False):
    points = buildMammalPoints('mammalTeeth.txt', '1/max')
    if printSteps:
        print 'Points:'
        for p in points:
            attrs = p.getOriginalAttrs()
            for i in range(len(attrs)):
                attrs[i] = round(attrs[i], 2)
            print '  ', p, attrs
    numClusterings = 0
    bestDiameter = None
    while numClusterings < numTrials:
        clusters, maxDiameter = kmeans(points, k, cutoff, Mammal)
        if bestDiameter == None or maxDiameter < bestDiameter:
            bestDiameter = maxDiameter
            bestClustering = copy.deepcopy(clusters) #Note deepcopy
        if printHistory:
            print 'Clusters:'
            for i in range(len(clusters)):
                print '  C' + str(i) + ':', clusters[i]
        numClusterings += 1
    print '\nBest Clustering'
    for i in range(len(bestClustering)):
        print '  C' + str(i) + ':', bestClustering[i]
```