Clusters Galore!: Three Algorithms for Microarray Dataset Clustering

Triton Pitassi
University of Portland, School of Engineering
5000 N. Willamette Blvd.
503-943-8000
pitassi17@up.edu

Caleb Piekstra
University of Portland, School of Engineering
5000 N. Willamette Blvd.
503-943-8000
piekstra17@up.edu

Matthew Farr
University of Portland, School of Engineering
5000 N. Willamette Blvd.
503-943-8000
farr16@up.edu

ABSTRACT
In this paper, we describe the process of analyzing data from microarray experiments. Microarrays are a method of performing experiments on sample DNA sequences to determine if similar genes are expressed in the samples. There are many different algorithms which can be used group experimental data with similar gene expression profiles. We implemented k-means clustering, quality threshold (QT clustering), and hierarchical clustering in a Python program. Each of the three algorithms uses a heuristic approach to produce cluster results which are not necessarily optimal. The program outputs the clusters resulting from each of the three algorithms, including printing a table of the clusters produced by k-means when the table would be two-dimensional.

Categories and Subject Descriptors
J.J.3 [**Computer Applications**]: Biology and genetics

General Terms
Algorithms, Experimentation.

Keywords
< "Microarray", "Clustering", "K-Means", "Quality Threshold (QT)", "Hierarchical", >

1. INTRODUCTION
Clustering is a data analysis tool of study that helps scientists in a variety of fields. Clustering algorithms are applied to microarray data and form clusters showing similar expression across genes. The results of clustering can help scientists determine the function of newly discovered genes, group similar organisms together, and determine if a person has a certain gene-related disease. There are many other applications for clustering and there are multiple algorithms that exist for clustering data. We chose to implement three different clustering algorithms: K-Means, QT, and Hierarchical clustering in order to cluster microarray data. Following this introduction, we provide background information on the clustering problem, the methods we used for clustering, and results and discussion comparing the results of our clustering algorithms to an established tool.

2. BACKGROUND

2.1 Biological Background
DNA Microarrays are small chips used to show the expression levels of genes. While every cell of an organism contains its whole genome, some of these genes are turned off in specific cells. Expression level of a gene in a sample refers to whether the mRNA of that gene is actually produced. A DNA microarray consists of an array of spots, each of which contains many identical copies of a single stranded DNA sequence corresponding to a specific gene's mRNA.

A microarray is organized so that the rows and columns represent genes and experiments. The experiment sample, dyed fluorescent green, a control sample, dyed fluorescent red, are washed over the microarray. The color on the spot can be used to determine the expression level of that gene for that experiment, with more green being a higher score, and more red being a lower score.

Experiment samples with similar scores for the same genes have similar expression profiles. Similarity in expression profiles can be used for many purposes, such as determining whether a patient's cells have a genetic disorder.

2.2 Computational Background
Making use of microarray data requires a method for determining how similar two expression profiles are. The three techniques used to show similarity in expression profiles for this paper are K-means clustering, quality threshold (or QT) clustering, and hierarchical clustering. Each of these is a greedy algorithm, meaning they make the obvious best choice at each step, but may not produce the best possible result.

Each of the three algorithms uses the expression scores as dimensions, and uses those scores to calculate distances, either between genes or between a gene and its cluster center.

K-means clustering assigns each gene to a random cluster, and then calculates the center for each cluster. The algorithm then reassigns each gene to the cluster with the nearest center, and computes new cluster centers. This process repeats until the clusters are stable.

QT clustering computes clusters starting with each gene, containing as many genes as possible without exceeding a certain maximum distance between genes. It then selects the largest cluster, removes those genes from the pool, and repeats until every gene is in a cluster.

Hierarchical clustering generates a tree grouping of the genes using the same process as UPGMA. It assigns each gene to its own cluster, and then calculates the distance between each cluster. The algorithm then combines the two closest clusters into one cluster, and computes the distance from this new cluster to each of the other clusters. This process repeats until there is only one cluster remaining.

3. METHODS

3.1 Data
The data used for this project was the ALL-AML microarray data from Computational Biology Lab 9. This dataset includes expression levels of 7070 human genes for 38 patients, each with

either acute myeloid leukemia (AML) or acute lymphoblastic leukemia (ALL).

3.2 Methods

To complete our project, we coded and used three separate clustering algorithms: *K-Means*, *Quality Threshold*, and *Hierarchical clustering*. Each algorithm uses a different method to determine clusters of similarly expressed genes.

3.2.1 K-Means Clustering

K-Means starts with a 'k' value that represents the maximum number of potential clusters to end up with. The algorithm then randomly assigns each gene in the microarray dataset to one of the k clusters. After this has been completed, the algorithm calculates the center of each cluster and finds the square distance between every gene and every cluster center. It then reassigns each gene to the cluster with the closest center. A gene may be reassigned to the same cluster it was in initially. This process of reassignment continues indefinitely until no gene is reassigned to a new cluster. Consequently, as our algorithm does not make any attempt to determine if one or more genes are switching back and forth between two or more clusters, the algorithm may fail to complete and the program will run infinitely. Once the final assignment of genes has been determined, the algorithm stops and returns the current list of clusters.

3.2.2 Quality Threshold Clustering

QT clustering begins by setting a 'diameter' value which is the maximum distance between genes in a cluster. When there are only two experiments per gene, the algorithm can be compared to drawing circles of the specified diameter with every gene inside the circle being a cluster. The algorithm begins by determining the distance between each gene every other gene by comparing vector distances between them. To optimize the algorithm, a boolean table is created using the values in the distance table in order to keep track of whether the distance between any two genes is within the specified diameter. Once these tables have been completed, the algorithm finds each gene's cluster using a greedy approach. After every gene has been assigned to a cluster, the largest cluster is saved as a final cluster and its genes are removed from future clustering iterations. The algorithm continues as such until every gene is in some final cluster in the set of final clusters. In other words, the intersection of the sets of clusters is the empty set.

3.2.3 Hierarchical Clustering

Hierarchical clustering follows the same process that the UPGMA uses for creating trees. The difference being that hierarchical clustering uses pairwise distances as opposed to alignment scores as distances. The algorithm starts out by assigning each gene to its own cluster. Once this has finished, the algorithm determines the distances between each cluster and every other cluster and finds the two closest clusters. These two clusters are then merged into a new group and the distances between clusters are recomputed in order to account for the two merged clusters being a combined group. This process continues until there is only one group remaining. The result is a list of joined clusters that represents a hierarchical distance tree of the genes.

4. RESULTS

The following results were gathered from the console output of a single run of our program using the ALL-AML microarray data and constants k = 3 and diameter = 100,000.

Number of rows of microarray data (patients): 38

Number of columns of microarray data (human genes): 7070

Row labels (patients' ID number and cancer types): {ALL1, ALL2, ALL3, ALL4, ALL5, ALL6, ALL7, ALL8, ALL9, ALL10, ALL11, ALL12, ALL13, ALL14, ALL15, ALL16, ALL17, ALL18, ALL19, ALL20, ALL21, ALL22, ALL23, ALL24, ALL25, ALL26, ALL27, AML28, AML29, AML30, AML31, AML32, AML33, AML34, AML35, AML36, AML37, AML38}

Gene clusters using K-Means clustering algorithm with k = 3:

Cluster Number	**Cluster Contents**
1	ALL1, ALL2, ALL3, ALL5, ALL9, ALL10, ALL11, ALL13, ALL14, ALL15, ALL16, ALL17, ALL20, ALL24
2	ALL4, ALL6, ALL7, ALL8, ALL12, ALL18, ALL19, ALL21, ALL22, ALL23, ALL26, ALL27, AML35
3	ALL25, AML28, AML29, AML30, AML31, AML32, AML33, AML34, AML36, AML37, AML38

Gene clusters using QT clustering algorithm with diameter = 100000:

Cluster Number	**Cluster Contents**
1	ALL1, ALL2, ALL4, ALL5, ALL10, ALL11, ALL12, ALL13, ALL14, ALL15, ALL16, ALL18, ALL19, ALL22, ALL24, ALL25, ALL26, ALL27, AML28, AML29, AML31, AML32, AML34
2	ALL3, ALL6, ALL7, ALL8, ALL23
3	AML30, AML33, AML36, AML37
4	ALL9, ALL17
5	ALL20
6	ALL21
7	AML35
8	AML38

Distance 'tree' using Hierarchical clustering algorithm:

{ALL21,{ALL20,{{{AML33,AML37},{AML30,AML36}},{AL L17,{{{ALL6,ALL23},{ALL9,{ALL3,{ALL10,ALL11}}}},{{{ ALL7,AML35},{{ALL8,ALL27},{ALL22,{ALL12,ALL25}}}}, {{AML29,AML38},{{ALL18,{ALL15,{{ALL13,{ALL1,ALL26 }},{ALL19,{ALL4,{ALL16,{ALL5,ALL24}}}}}}},{{ALL2,AL L14},{{AML28,AML32},{AML31,AML34}}}}}}}}}}

5. DISCUSSION

5.1 Algorithm Complexities

The algorithms used by our program had slightly varied runtimes, though none of them had a better time complexity than $O(N^2)$.

Time and space complexity of our implementations of the algorithms:

Clustering Algorithm	Time Complexity	Space Complexity
K-Means	$O(N^2)$	O(n*m) where n is the number genes and m is the number of experiments in the microarray dataset.
QT	$O(N^3)$	$O(n^2)$ where n is the number of genes in the microarray dataset.
Hierarchical	$O(N^2)$	$O(n^2)$ where n is the number of genes in the microarray dataset.

5.2 Analysis of Results

5.2.1 Evaluation

5.2.1.1 QT

The results of the algorithms vary substantially. The QT algorithm does not make very much sense for this dataset. The clusters it generated mostly contained exclusively AML or ALL patients, but having 8 clusters doesn't make much since and it extremely dependent on the diameter passed to the algorithm.

5.2.1.2 K-Means

In contrast, the K-Means algorithm makes quite a bit more sense. You can choose to limit the number of clusters to a desired number based on the k value. In our tests, we tried setting k to two as we hoped that all of the ALL patients and all of the AML patients would end up in their own clusters but that did not happen as expected. This is due to the randomness of the algorithm when performing the initial cluster assignment of each gene. Once we upped the k value to three as seen in our results, the clusters made a lot more sense. After running the program a few times to help account for the initial randomness in the K-Means algorithm, it generated results that were quite satisfactory. Each cluster was almost exclusively AML or ALL patients except for cluster two which had only a single AML patient out of 13 total and cluster three which had only a single ALL patient out of 11 total in the cluster. However, it is important to note that had we not known which patients were AML or ALL, in order to determine which cluster results were 'good' we would have had to code the algorithm to run repeatedly and determine which result out of multiple results was the most accurate in terms of cluster assignments based on expression levels.

5.2.1.3 Hierarchical

The results of the Hierarchical clustering algorithm show the groupings of the patients into clusters. Everything between a left curly brace and its corresponding right curly brace is in the same group. Each ALL or AML patient is an element in the group. The unfortunately difficult to read results of the Hierarchical clustering algorithm show that for the most part, AML patients are connected with other AML patients and the same for ALL patients. However, in some areas of the 'tree' the AML patients, while mostly grouped with each other, are interspersed among ALL patients. Conclusively, the K-Means algorithm is the best choice for analyzing our ALL-AML dataset.

5.2.2 Comparison

In order to help gauge the accuracy of the results of the algorithm that worked the best with our dataset, K-Means, we pitted it up against Java TreeView. Java TreeView allows for a k value as expected, but it also allows for specifying the number of runs, which are the number of times it runs the K-Means algorithm before picking the best result from all of the runs. This is to help alleviate any bad results that would occur due to the initial randomness of the algorithm. While our algorithm does not account for this, by running our program manually multiple times, we were able to visually see which run had the best results and keep that one. Doing it by hand is much slower than doing it programmatically as Java TreeView does it.

Using the same k value of three and with 20 runs, the results were as follows:

Cluster Number	**Cluster Contents**
1	ALL2, ALL17, ALL20, AML28, AML30, AML32, AML33, AML36, AML37
2	ALL3, ALL6, ALL9, ALL10, ALL11, ALL23
3	ALL1, ALL4, ALL5, ALL7, ALL8, ALL12, ALL13, ALL14, ALL15, ALL16, ALL18, ALL19, ALL21, ALL22, ALL24, ALL25, ALL26, ALL27, AML29, AML31, AML34, AML35, AML38

The results given by Java TreeView were far less successful at separating out the ALL and AML patients into clusters containing patients with only the same type of leukemia when compared to the cluster assignments determined by our implementation of K-Means. However, while we also performed 20 runs (manually), we were not using a programmatic evaluation of the cluster assignments, but rather a biased personal evaluation. Our personal evaluation was heavily impacted by our ability to know that the desired result was for ALL patients to be clustered with other ALL patients and likewise for AML. In consequence, while our results seem to be far more accurate, we would need to establish a way to evaluate the clusters based on expression levels in order to achieve a more unbiased selection from among our 20 runs.

5.2.3 Accuracy

In regards to which clustering algorithm of the three we implemented is best suited to the problem presented by our dataset, K-Means is definitely our first choice. Primarily, since we can set the number of desired clusters manually based on how many we are looking for, that makes it the best choice. The results from Quality Threshold clustering are not very helpful in determining which type of cancer each patient has and the results from Hierarchical clustering are mixed around so that it is also hard to cluster effectively.

The K-Means clustering is quite accurate. While only one of the three clusters contained patients all with the same type of cancer, the other two clusters only had a single out of place patient each.

If a new patient were introduced and clustered into one of those clusters, we could likely tell which kind of cancer this new patient had. However, due to the K-Means algorithm having a degree of randomness to it, requiring it to be run manually many times and evaluated objectively, it is less than ideal since the clusters found in one run with k = 3 will be different from another ten runs with the same k value.

All-in-all, these clustering algorithms are not reliable enough for datasets where the expected results are not known in advance. While the K-Means algorithm is best suited clustering the known cancer data, because of the randomness involved in the algorithm, it would still not be reliable enough to make a definite conclusion.

5.2.4 Future Work

The next steps in this project come in two parts, the first being to explore different types of clustering algorithms. There are several other well-known clustering algorithms that we did not implement that may be useful in clustering this type of dataset.

The other step is to determine ways to refine the algorithms that we have already implemented. There may be ways to modify any of the three algorithms that we created in order to decrease runtime and increase accuracy, at least in the average case.

6. RELATED WORK

There are multiple tools and research projects that make use of clustering to analyze data. There are numerous applications in biological fields, so our work is tied to research in those other areas. The methods we used are used in some of this other research and other microarray data with cancer information is used in a plethora of other studies.

The K-Means algorithm was created by Stuart Lloyd in 1957. We modeled our algorithm after the K-Means clustering pseudocode in the Computational Biology course pack. This algorithm still follows the general format of Lloyd's implementation, but has been refined to work specifically with microarray datasets. Our implementation also creates a scatterplot for data with only 2 experiments and colors the clusters to give a visual representation of the clustered data.

QT Clustering was created by Heyer, Kruglyak, and Yooseph in 1999. We modeled our algorithm after the QT clustering pseudocode in the Computational Biology course pack. Our implementation creates a look-up table with the distance between each pair of genes and another look-up table with containing booleans that tell whether the distance between a pair of genes is within the diameter. Essentially, it does some preprocessing in order to increase efficiency.

S.C. Johnson defined the process of Hierarchical clustering in 1967. We modeled our algorithm after the Hierarchical clustering pseudocode in the Computational Biology course pack. The algorithm follows the general format of Johnson's implementation, but has been refined to work specifically with microarray datasets.

7. CONCLUSIONS

Clustering microarray data has important applications across many different fields in biology. There are different clustering algorithms that are better for certain applications. We implemented three algorithms that all have advantages and disadvantages.

K-Means clustering is useful when we have an idea about how many clusters there likely should be, such as with the data set that we tested: We knew that some patients had ALL and some had AML, so we could base the number of clusters off of this information. K-Means is at a disadvantage when we do not have any idea how many clusters there should be and when the dataset contains outliers.

Quality Threshold clustering is useful when we know how large a distance can be between genes in the same cluster. This can be useful if we have a newly discovered gene and want to determine what process it's a part of; we can cluster it with other well-known genes and see which cluster it falls into. The disadvantages of this algorithm are that we have to know the diameter of the clusters beforehand and that it can be computationally intensive compared to K-Means clustering.

Hierarchical clustering is useful when we do not know how many clusters we want, but instead want to look at several different numbers of clusters and determine which is the best. This type of clustering can also be useful if we want to see a taxonomic relationship between data in the microarray. The disadvantages are that it is sensitive to outliers and that it has a slow runtime.

Describe the conclusions you draw from your project. Make sure all points you want the reader to remember are included in this section. (In fact, the abstract and conclusions are probably the most important sections in research papers for readers who are scanning for related work.)

The full paper (including references but NOT including appendices should be at least 4 pages in length and no more than 8 pages in length).

8. ACKNOWLEDGMENTS

We would like to thank Dr. VanDeGrift for her contribution to this project.

9. REFERENCES

[1] VanDeGrift, T., 2015. BIO/CS 423 Computational Biology. Course pack for Computational Biology class at the University of Portland.

[2] Pond , S. L. K., 2011. Clustering in Bioinformatics. Presentation for the CSE department at the University of San Diego.

[3] Matteucci, M. A Tutorial in Clustering Algorithms: K-Means Clustering. Webpage created for students at the Politecnico di Milano.

[4] opticks.org. Quality Threshold Clustering Algorithm. Webpage that describes the use of the Quality Threshold Clustering algorithm.

[5] Borgatti, S. P., 1994. How to Explain Hierarchical Clustering. INSNA 1994. University of South Carolina

[6] Gao, J. 2012. Clustering Lecture 3: Hierarchical Clustering. University at Buffalo, 2012. Lecture for university course.

10. APPENDICES

10.1 User Manual: Readme

System Requirements:

- Python 3.X
- matplotlib python package
- possibly numpy package

To install missing packages run the following:

1. pip install matplotlib
2. pip install numpy

Note: pip is python's package manager

To run the program, simply run clusterMaster.py using Python 3.X

Example:

$ python3 clusterMaster.py

To edit the k value for k-means clustering and the diameter for QT clustering you have to edit the 'k' and 'diameter' variables in clusterMaster.py

The clustering results as well as a tree of the genes is printed out to the console and written to the file 'results.txt'

To modify the microarray data, edit the 'microarraydata.csv' file. Note that the program expects that each column will be labeled with an expression header and each row's first column will contain the gene name. The values contained in the rest of the file are comma-separated integers.

10.2 Source Code

Include the code you wrote/modified (paste it here as an appendix to your paper in 8-pt Courier font). Your code should be formatted cleanly and appropriately commented. If the code is really long, put sample code in the Appendix and refer to the name of the zip file containing your code here.

CLUSTERMASTER.PY:

```
# graphing imports!
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import re

from csvReader import CSVReader
# k-means clustering
from k_means import KMeans
# QT clustering
from QT import QT
# Hierarchical clustering
from hierarchical import Hierarchical

# Print lots of stuff
VERBOSE = False

# the input file
inputFile = "ALL-AML-TRANSPOSED.csv"
# the output file
```

```
outputFile = "results.txt"
# A CSV file reader
csvReader = CSVReader()

# get the microarray data from the csv file
microarrayData = csvReader.read(inputFile)
microarrayLabels = csvReader.getLabels(inputFile)
print ("File %s parsed succesfully!\n\tRows:\t\t%d\n\tColumns:\t%d" % (inputFile, len(microarrayData), len(microarrayData[0])))
print ("\nLabels: {%s}" % (', '.join(microarrayLabels)))

## k-means algorithm!
# set the k-value (max potential clusters)
k = 3
# holds a reference to a KMeans object
kmeans = KMeans(verbose=VERBOSE)
# get the clusters determined by the algorithm
kMeansFinalClusters = kmeans.kmeans(microarrayData, k)

## QT algorithm!
# set the diameter for the QT algorithm
diameter = 100000# holds a reference to a QT object
qt = QT(verbose=VERBOSE)
# get the clusters determined by the algorithm
qtFinalClusters = qt.QTClustering(microarrayData, diameter)

## Hierarchical Clustering algorithm!
# holds a reference to a QT object
hc = Hierarchical(microarrayData, VERBOSE)
# get the clusters determined by the algorithm
hcFinalClusters = hc.hierarchicalCluster()

# print out the data in the microarray if in VERBOSE mode
if VERBOSE:
    print ("\nParsed Microarray Data:\n")
    for idx, gene in enumerate(microarrayData):
        print ("Gene %d:" % (idx+1), gene)

# print out the k-means clusters
print ("\n\nFinal sets of gene clusters:\n")
print ("Using K-Means Algorithm:")
for clusterIdx, cluster in enumerate(kMeansFinalClusters):
    print ("\tCluster %d: {%s}" % (clusterIdx+1, ', '.join([microarrayLabels[idx] for gene, idx in cluster])))
print ("")
```

```
# print out the QT clusters
print ("Using QT Algorithm:")
for clusterIdx, cluster in enumerate(qtFinalClusters):
    print ("\tCluster %d: {%s}" % (clusterIdx+1, ', '.join([microarrayLabels[geneNum] for geneNum in cluster])))
print ("")

# print out the hierarchical tree
print ("Using Hierarchical Clustering Algorithm:")
# establish a regular expression to match the genes
geneRegex = re.compile('G\d+')
# get the gene indexes
geneNumbers = [int(gene[1:]) - 1 for gene in geneRegex.findall(hcFinalClusters)]
# use the gene indexes to get the labels for the genes
# and replace the genes with their labels
labeledGenes = re.sub(geneRegex, lambda match: microarrayLabels[geneNumbers.pop(0)], hcFinalClusters)
# print out the labeled 'tree'
print ("Tree: %s" % (labeledGenes))

with open(outputFile, 'w') as out:
    # print out the k-means clusters
    out.write("\n\nProgram results for gene clusters:\n\n")
    out.write("\nUsing K-Means Algorithm with k = %d:\n" % (k))
    for clusterIdx, cluster in enumerate(kMeansFinalClusters):
        out.write("\tCluster %d: {%s}\n" % (clusterIdx+1, ', '.join([microarrayLabels[idx] for gene, idx in cluster])))
    out.write("")

    # print out the QT clusters
    out.write("\nUsing QT Algorithm with diameter %d:\n" % (diameter))
    for clusterIdx, cluster in enumerate(qtFinalClusters):
        out.write("\tCluster %d: {%s}\n" % (clusterIdx+1, ', '.join([microarrayLabels[geneNum] for geneNum in cluster])))
    out.write("")

    # print out the hierarchical tree
    out.write("\nUsing Hierarchical Clustering Algorithm:\n")
    out.write("Tree: %s\n" % hcFinalClusters)
#============================================================================================
K_MEANS.PY:
import random
# Graphing imports
import matplotlib.pyplot as plt
import matplotlib.colors as colors

##
```

```
#
# Author: Caleb Piekstra
#
# Last Modified: 12/11/2015
#
# Description: Runs the KMeans clustering algorithm
#
# K-means Clustering: Given N items with known distances between
#   items and K clusters, assigns the N items into one of K clusters
#   such that the total distance from each item to its cluster
#   is minimized.
#
class KMeans:

    ##
    # constructor
    #
    # Description: Initializes instance variables used for the algorithm
    #
    # Parameters:
    #   self - The object pointer
    #   microarray - A 2D list where each sublist is a gene containing
    #       its expression data
    #   k - The upper bound on the number of initial clusters
    #   verbose - Optional parameter to print out information
    #
    def __init__ (self, verbose=False):
        # whether or not to print extra information to console
        self.verbose = verbose


    ##
    # center
    #
    # Description: Given a cluster of genes, calculates and
    #   returns the center of the cluster
    #
    # Parameters:
    #   cluster - A cluster of genes
    #
    # Returns:
    #   The calculated center of the cluster
    #
    def center (self, cluster):
        # get the length of the cluster
```

```
    listLen = float(len(cluster))
    # return a list containing the center of the cluster
    return [sum(exp)/listLen for exp in list(zip(*cluster))]


##
# sqDist
#
# Description: Given two genes, calculates
#   the square distance between them
#
# Parameters:
#   self - the object pointer
#   initial - The source gene
#   dest - The destination gene
#
# Returns:
#   The square distance between the initial gene and dest
#   gene
#
def sqDist (self, initial, dest):
    # holds the square distance between two genes
    sumSquares = 0

    # loop for the length of the initial gene
    for i in range(len(initial)):
        # square the distance between each expression in
        # the destination and initial genes and add it
        # to the total sum
        sumSquares += abs(dest[i] - initial[i])**2

    # return the square distance
    return sumSquares


##
# newClusters
#
# Description: Creates a new empty cluster list
#   of a certain size
#
# Parameters:
#   self - The object pointer
#   len - The number of clusters
#
```

```
# Returns:
#   An emtpy 2D list with len inner lists
#
def newClusters(self, len):
    return [[] for _ in range(len)]


##
# plot
#
# Description: Thickens
#
# Parameters:
#   self - The object pointer
#   clusters - Gene clusters
#   figure - The figure number
#
def plot(self, clusters, figureNum):
    # only allow plotting of clusters with
    # genes with 2 expressions (2D graph)
    if self.numExps != 2:
        return

    # remove the gene label from each gene
    refinedClusters = [[gene[0] for gene in cluster] for cluster in clusters]

    # modify clusters to have all x's in one tuple and all
    # y's in the other tuple (seperate exp1 and exp2)
    xyClusters = [list(zip(*cluster)) for cluster in refinedClusters]

    # get a unique set of colors to use for the clusters
    clusterColors = colors.cnames.values()[:len(xyClusters)]

    # create a new figure
    plt.figure(figureNum)
    # create a scatter-plot for the figure
    for idx, cluster in enumerate(xyClusters):
        plt.scatter(cluster[0], cluster[1], color=clusterColors[idx])


##
# kmeans
#
# Description:
#   Runs the K-Means clustering algorithm on microarray
```

```
    #   data.
    #
    # Parameters:
    #   self - The object pointer
    #   mdata - The microarray data
    #   k - The k-value of the clustering algorithm
    #       (max potential clusters)
    #
    # Returns:
    #   The final cluster assignment
    #
    def kmeans (self, mdata, k):
        # the number of genes in the microarray
        numGenes = len(mdata)
        # the number of expressions per gene
        numExps = len(mdata[0])

        if self.verbose:
            print ("K-means Clustering:\n\tk = %d\n\tnum genes: %d\n\tnum expressions: %d" %
(k, numGenes, numExps))

        # create a new empty list of clusters
        clusters = self.newClusters(k)

        # randomly assign each gene in the microarray data to one of the k clusters
        for geneIdx, gene in enumerate(mdata):
            clusters[random.randint(0,k-1)].append((gene, geneIdx))

        # remove any empty clusters
        clusters = list(filter(None, clusters))

        # keeps track of the number of cluster arrangements
        counter = 0

        # the algorithm runs until the cluster assignments stop changing
        while True:
            # keep track of the number of unique cluster arrangements
            counter += 1

            # only plot the clusters if there are 2 expressions
            if numExps == 2:
                self.plot(clusters, counter)

            # create an empty 2D list to hold the next iteration of clusters
            newClusters = self.newClusters(len(clusters))
            # calculate the centers of the current clusters
```

```
            centers = [self.center(list(zip(*cluster))[0]) for cluster in clusters]

            if self.verbose:
                print ("\ncluster arrangement %d:" % (counter))
                print ("Gene%s\t%s\t cluster assignment" % (' '.join(['    ' for _ in range(2,numExps)]), '\t'.join(["sq dist to center C%d" % i for i in range(1, len(centers)+1)])))

            # loop through the microarray data
            for geneIdx, gene in enumerate(mdata):
                # determine the distance from the gene to each cluster center
                distancesToCenters = [self.sqDist(gene, center) for center in centers]

                # figure out which cluster to assign the gene to (minimum distance to center)
                assignedClusterIdx = min(range(len(distancesToCenters)), key=distancesToCenters.__getitem__)

                # assign the gene to the new cluster
                newClusters[assignedClusterIdx].append((gene, geneIdx))

                if self.verbose:
                    print ("%s\t%s\t\t\t %s" % (gene, '\t\t\t'.join(["%0.2f" % dist for dist in distancesToCenters]), ("C%d" % (assignedClusterIdx+1))))

            # remove any empty clusters
            newClusters = list(filter(None, newClusters))
            # if the new cluster assignment is the same as the previous one
            # then the algorithm has finished
            if newClusters == clusters:
                break
            # otherwise loop again for new cluster assignments
            else:
                clusters = newClusters

        # display the data in figures if there are 2 expressions
        # (2D plot!)
        if numExps == 2:
            plt.show()

        # return the final cluster arrangement
        return clusters


# Test code for directly running the file
if __name__ == "__main__":
    # preprocessing (input)
    exp1 = [1, 6, 5, 2, 6, 0, 1, 1, 2, 4, 5, 6, 7]
```

```
    exp2 = [4, 2, 3, 5, 4, 5, 3, 1, 2, 4, 5, 6, 7]
    exps = (exp1, exp2)

    # preliminary data
    microarrayData = list(zip(*exps))
    k = 4

    # initialize the kmeans object
    kmeans = KMeans(k, verbose=True)

    # run the algorithm!
    finalClusters = kmeans.kmeans(microarrayData)

    # print out the results
    print ("\nFinal set of gene clusters:")
    for clusterIdx, cluster in enumerate(finalClusters):
        print ("\tCluster %d: %s" % (clusterIdx+1, ["gene" + str(idx+1) for gene, idx in
cluster]))
    print ("")

    # show the plots
    if kmeans.numExps == 2:
        plt.show()
#================================================================================
QT.py
import math

#############################################################################
# QT (class)
#
# Author: Triton Pitassi
#
# Last Modified: 12/2/2015
#
# Description: This class contains the necessary funtions to run the
# QT Clustering algorithm.
#############################################################################
class QT:

    def __init__(self, verbose=False):
        self.verbose = verbose
        self.divider = "=============================================================="

    ###########################################################################
    # QTClustering
    #
```

```
    # Description:
    #       This is the main function for implementing the QT Clustering
    # algorithm. It uses the indicies of the genes in order to perform the
    # clustering. Creates and uses a distance table and a diameter truth
    # table in order to determine the clusters. It loops through the set S
    # containing all indicies until S is empty then returns the final
    # clusters that were found
    #
    # Parameters:
    #       self - the object pointer
    #       data - a list of lists where each inner list contains the
    #              experiment data for one gene
    #       diameter - the maximum diameter of each cluster
    #
    # Return:
    #       a list of lists where each inner list is a cluster
    ###########################################################################
    def QTClustering(self, data, diameter):

        # number of genes
        numGenes = len(data)

        # make set S
        setS = self.createSetS(numGenes)

        # initialize list of final clusters
        finalClusters = []

        # create distance table
        distTable = self.createDistanceTable(self.createTable(numGenes, numGenes, 0), data,
numGenes)

        # create table to tell if genes are within diameter d
        diameterTable = self.createDiameterTable(self.createTable(numGenes, numGenes, True),
distTable, diameter, numGenes)

        if self.verbose:
            print("Total number of genes:", numGenes, "\n")
            print("Distance Table")
            self.printDistTable(distTable)
            print("\t")
            print("Diameter Table")
            self.printDiameterTable(diameterTable)
            print("\n" + self.divider + "\n")

        # while there are still items left to cluster
```

```
    while setS:

        # find each gene's personal cluster
        clusters = self.findClusters(diameterTable, distTable, setS)

        # determine the biggest cluster
        biggestCluster = self.findBiggestCluster(clusters)

        # add list to list of final clusters
        finalClusters.append(biggestCluster)

        if self.verbose:
            print("The set of genes left to cluster:", setS, "\n")

            print("The clusters for this iteration are:")
            for cluster in clusters:
                print(str(clusters[0]) + ":" + str(cluster))

            print("\nThe biggest cluster for this iteration is:", biggestCluster, "\n")
            print(self.divider, "\n")
        # remove genes that have been clustered
        setS = self.removeClusteredGenes(setS, biggestCluster)

    # return the final clusters
    return [sorted(cluster) for cluster in finalClusters]


##############################################################################
# createSetS
#
# Description: creates a list of the indicies of each gene to use in
# the QT algorithm
#
# Parameters:
#       self - the object pointer
#       numGenes - the number of genes to cluster
#
# Return:
#       a list of ints representing gene indicies
##############################################################################
def createSetS(self, numGenes):

    # initialize the list
    setS = []
```

```
    # put the indicies in setS
    for i in range(0, numGenes):

        setS.append(i)

    return setS


##########################################################################
# createDistanceTable
#
# Description: make a table that holds the distances between each pair
# of genes. The indicies represent genes, so the distance at [i][j] is
# the same as the distance at [j][i].
#
# Parameters:
#       self - the object pointer
#       distTable - the table to fill
#       data - the data set represented by a list of lists
#       numGenes - the number of genes to cluster
#
# Return:
#       a list of lists representing a table of distances
##########################################################################
def createDistanceTable(self, distTable, data, numGenes):

    #find all of the distances
    for i in range(0, numGenes):
        for j in range(0, numGenes):

            # calculate the distance
            distTable[i][j] = self.dist(data[i], data[j])

    # return the distance table
    return distTable


##########################################################################
# createDiameterTable
#
# Description: make a table that holds a boolean telling whether the
# distance between a pair of genes is less than or equal to the
# diameter i.e. if they could possibly be in the same cluster. The
# indicies represent genes, so the boolean at [i][j] is the same as the
# distance at [j][i].
```

```
    #
    # Parameters:
    #       self - the object pointer
    #       diameterTable - the table to fill
    #       distTable - the table that contains distances between pairs of
    #                   genes
    #       diameter - the diameter of each cluster
    #       numGenes - the number of genes to cluster
    #
    # Return:
    #       a list of lists representing a table of booleans
    ##########################################################################
    def createDiameterTable(self, diameterTable, distTable, diameter, numGenes):

        for i in range(0, numGenes):
            for j in range(0, numGenes):

                # if the distance between the genes is greater than the diameter, set to
False
                if distTable[i][j] > diameter:

                    diameterTable[i][j] = False

        # return the diameter table
        return diameterTable


    ##########################################################################
    # findClusters
    #
    # Description: finds each gene's personal clusters. Starts with the
    # closest gene and adds the next closest until the distance exceeds the
    # diameter.
    #
    # Parameters:
    #       self - the object pointer
    #       diameterTable - the table telling whether the distance between
    #                       two genes is less than the diameter
    #       distTable - the table that contains distances between pairs of
    #                   genes
    #       setS - a list of indicies representing the genes to cluster
    #
    # Return:
    #       a list of lists where each inner list is a cluster
    ##########################################################################
    def findClusters(self, diameterTable, distTable, setS):
```

```
        # initialize list of clusters
        clusters = []

        # for every gene index left in S
        for i in range(0, len(setS)):

            # copy S to a list so that we can remove items from it without changing S
            tempSetS = list(setS)

            # remove the gene
            tempSetS.remove(setS[i])
            clusters.append([setS[i]])

            # initialize boolean to keep track of whether we are done finding this gene's
personal cluster
            fitInCluster = True

            # while there is a gene that fits in the diameter of the cluster
            while fitInCluster:

                # if there aren't any genes left to possibly put in a cluster,
                # then we're done with this cluster
                if not tempSetS:

                    break

                # initialize variables for finding the closest gene
                closestGene = -1
                closestGeneDist = float("inf")

                # find closest gene
                for gene in tempSetS:

                    # if we haven't found a gene yet or this gene is closer than the closest
we've found, make it the closest gene
                    if closestGene == -1 or distTable[setS[i]][gene] < closestGeneDist:

                        closestGene = gene
                        closestGeneDist = distTable[setS[i]][gene]

                # determine if the closest gene fits within the diameter of the cluster
                for gene in clusters[i]:

                    # if the distance is greater than the diameter, this gene doesn't fit
                    if not diameterTable[gene][closestGene]:
```

```
                        fitInCluster = False

                        # since we know the gene doesn't fit, we don't have to check the rest
of the genes in the cluster
                        break

                # if the next closest gene fits, put it in the cluster and remove it from our
temporary S
                if fitInCluster:

                    clusters[i].append(closestGene)
                    tempSetS.remove(closestGene)

        # return a list with each gene's personal cluster
        return clusters


    ##########################################################################
    # findBiggestCluster
    #
    # Description: finds the biggest cluster in a group of clusters
    #
    # Parameters:
    #       self - the object pointer
    #       clusters - a list of each gene's personal cluster (only includes
    #                  genes left to cluster)
    #
    # Return:
    #       a list representing a cluster
    ##########################################################################
    def findBiggestCluster(self, clusters):

        # initialize variables for finding biggest cluster
        biggestCluster = []
        lenBiggestCluster = 0

        # for each cluster, determine if it is larger than the biggest cluster we've found so
far
        for cluster in clusters:

            if len(cluster) > lenBiggestCluster:

                lenBiggestCluster = len(cluster)
                biggestCluster = cluster
```

```
    # return the biggest cluster found
    return biggestCluster


#############################################################################
# removeClusteredGenes
#
# Description: removes genes from set S if they have been put in the
# group of final clusters
#
# Parameters:
#       self - the object pointer
#       setS - a list of indicies representing the genes to cluster
#       biggestCluster - the biggest cluster found in this iteration
#
# Return:
#       set S with the genes in the biggest cluster removed
#############################################################################
# we've found the biggest cluster for this iteration so remove those gene indicies from S
def removeClusteredGenes(self, setS, biggestCluster):

    for geneIdx in biggestCluster:

        setS.remove(geneIdx)

    return setS


#############################################################################
# createTable
#
# Description: Create a 2D table with the given number of rows and columns
# and fills all entries with value given as a parameter
#
# Parameters:
#       self - the object pointer
#       numRows - the number of rows to put in the table
#       numCols - the number of columns to put in the table
#       value - the temporary value to store in each cell
#
# Return:
#       a table with with the specified number of rows and columns
#############################################################################
def createTable(self, numRows, numCols, value):
```

```
    return [list([value]*numCols) for _ in range(numRows)]


#############################################################################
# printDistTable
#
# Description: print the distance table to the console
#
# Parameters:
#       self - the object pointer
#       table - the distance table to print
#
# Return:
#       void
#############################################################################
def printDistTable(self, table):
    # get table dimensions
    NUM_ROWS = len(table)
    NUM_COLS = len(table[0])

    # print top row
    print(" \t", end="")
    for i in range(0, NUM_ROWS):
        print(repr(i).ljust(1), "\t", end="")

    print("\t")

    # print table contents
    for i in range(0, NUM_ROWS):
        print(i, "\t", end="")
        for j in range(0, NUM_COLS):
            print(repr(float("{0:.2f}".format(table[i][j]))).ljust(1), "\t", end="")

        print("\t")

    return


#############################################################################
# printDiameterTable
#
# Description: print the diameter table to the console
#
# Parameters:
#       self - the object pointer
```

```
#       table - the diameter table to print
#
# Return:
#       void
##############################################################################
def printDiameterTable(self, table):
    # get table dimensions
    NUM_ROWS = len(table)
    NUM_COLS = len(table[0])

    # print top row
    print(" \t", end="")
    for i in range(0, NUM_ROWS):
        print(repr(i).ljust(2), "\t", end="")

    print("\t")

    # print table contents
    for i in range(0, NUM_ROWS):
        print(i, "\t", end="")
        for j in range(0, NUM_COLS):
            print(repr(table[i][j]).ljust(2), "\t", end="")

        print("\t")

    return

##############################################################################
# dist
#
# Description: Calculates the distance between two gene vectors.
#
# Parameters:
#       self - the object pointer
#       gene1 - the first gene vector
#       gene2 - the second gene vector
#
# Return:
#       the distance between the two gene vectors
##############################################################################
def dist(self, gene1, gene2):

    # initialize distance
    distance = 0
```

```
        # calculate (x2 - x1)^2 for each term
        for i in range(0, len(gene1)):

            distance += (gene1[i] - gene2[i])**2

        # calculate square root and return
        return math.sqrt(distance)


if __name__ == "__main__":

    # UNIT TESTS
    gene1 = [2, 5, 1]
    gene2 = [6, 3, 2]
    gene3 = [1, 1, 3]
    gene4 = [5, 3, 6]
    gene5 = [2, 3, 3]
    gene6 = [4, 4, 2]

    genes = []
    genes.append(gene1)
    genes.append(gene2)
    genes.append(gene3)
    genes.append(gene4)
    genes.append(gene5)
    genes.append(gene6)

    qt = QT(True)
    print("The final clusters are:", qt.QTClustering(genes, 5))
#===============================================================================
HIERARCHICAL.py
import random
import math
import sys

##
# Class for performing hierarchical clustering to generate trees.
# Uses microarray data to output a tree where each leaf is a gene.
#
class Hierarchical:

# Constructor
#
    def __init__ (self, microarray, verbose=False):
        self.verbose = verbose
```

```
    self.mdata = microarray
    self.numGenes = len(microarray)
    self.numExps = len(microarray[0])
    self.distTable = self.initTable()

##
# calcDist
#
# Method for calculating distances between two genes based on
# microarray experiment data. Takes two genes' experiment results
# as input and returns the distance between them as a float
#
def calcDist (self, initial, dest):
    sumSquares = 0
    for i in range(len(initial)):
        sumSquares += abs(dest[i] - initial[i])**2
    return math.sqrt(sumSquares)

##
# initTable
#
# Creates the initial distance table. Calculates the distances
# between each pair of genes and places it in the appropriate entry
# in the table.
#
def initTable(self):
    # Size of square table containing distances and gene group labels
    tableDim = self.numGenes+1

    # Initialize table as a table of placeholder values
    distTable = [['X' for x in range(tableDim)] for x in range(tableDim)]

    # Fill in each of the gene group labels
    for i in range(self.numGenes):
        label = "G" + str(i+1)
        distTable[0][i] = label
        distTable[i+1][self.numGenes] = label

    # Place the distances between pairs of genes in the appropriate
    # table cells
    for i in range(self.numGenes-1):
        for j in range(i+1,self.numGenes):
            distTable[i+1][j] = self.calcDist(self.mdata[i],self.mdata[j])

    return distTable
```

```
##
# getMinDistance
#
# returns the coordinates in the table of the minimum distance in
# the table
#
def getMinDistance(self):

    tableDim = len(self.distTable)

    # Arbitrarily large so anything will be smaller
    minVal = sys.maxsize

    coords = (-1,-1)

    # Loop over all distance values in the array
    for i in range(len(self.distTable)-2):
        for j in range(i+1,len(self.distTable[0])-1):
            if self.distTable[i+1][j] < minVal:
                minVal = self.distTable[i+1][j]
                coords = (i+1,j)
    return coords

##
# mergeClusters
#
# Combines two clusters in the table to create a new cluster.
# Creates a new distance table with the merged clusters
#
def mergeClusters(self):
    # Size of a table 1 smaller than the current distance table
    tableDim = len(self.distTable)-1

    # Size of old distance table
    oldDim = tableDim + 1

    # Coordinates of minimum distance
    minimum = self.getMinDistance()
    minCol = minimum[1]
    minRow = minimum[0]

    # Initialize table as a table of placeholder values
    newTable = [['X' for x in range(tableDim)] for x in range(tableDim)]
```

```
        # Label rows and columns of new table
        col = 0
        row = 1
        for i in range(oldDim):
            if i != minCol and i != minRow - 1:
                label = self.distTable[0][i]
                newTable[0][col] = label
                newTable[row][tableDim-1] = label
                col += 1
                row += 1
        label = '{' + self.distTable[minRow][oldDim-1] + ',' + self.distTable[0][minCol] +
'}'
        newTable[0][tableDim-2] = newTable[tableDim-1][tableDim-1] = label

        row = col = 1
        # Iterate over rows
        for i in range(1,oldDim-1):
            if i != minRow and i != minCol + 1:
                col = row
                # Iterate over columns
                for j in range(i,oldDim-1):
                    if j != minCol and j != minRow - 1:
                        newTable[row][col] = self.distTable[i][j]
                        col += 1
                row += 1

        clustDist1 = []
        clustDist2 = []

        # Get distances from one of our clusters to the others
        for i in range(1, minCol+1):
            if i != minRow:
                clustDist1.append(self.distTable[i][minCol])
        for i in range(minCol+1, oldDim-1):
            if i != minRow-1:
                clustDist1.append(self.distTable[minCol+1][i])

        # Get distances from the other cluster to the others
        for i in range(1, minRow):
            if i != minCol + 1:
                clustDist2.append(self.distTable[i][minRow-1])
        for i in range(minRow, oldDim-1):
            if i != minCol:
                clustDist2.append(self.distTable[minRow][i])

        for i in range(1,tableDim-1):
```

```
            newTable[i][tableDim-2] = (clustDist1[i-1] + clustDist2[i-1])/2

        self.distTable = newTable
        return


    ##
    # hierarchicalCluster
    #
    # Performs hierarchical clustering
    #
    def hierarchicalCluster(self):
        while len(self.distTable) > 2:
            self.mergeClusters()
            #for i in range(len(self.distTable)):
                #print(self.distTable[i])

#print("================================================================================")
        return self.distTable[0][0]

# TESTS
if __name__ == "__main__":
    exp1 = []
    exp2 = []
    for i in range(12):
        exp1.append(random.randint(0,10))
        exp2.append(random.randint(0,10))

    exps = (exp1, exp2)

    microarrayData = list(zip(*exps))
    print(microarrayData)
    h = Hierarchical(microarrayData, False)
    #print(h.mdata)
    #print("================================================================================")
    print(h.hierarchicalCluster())
#============================================================================================
CSVREADER.py
##
#
# Author: Caleb Piekstra
#
# Last Modified: 12/11/2015
#
# Description: Parases a CSV file that contains microarray data
#   in the form of genes on rows and expression values on columns
```

```
#
class CSVReader:


    ##
    # read
    #
    # Description: Given a csv file name, opens the file
    #   and parses out the microarray data
    #
    # Parameters:
    #   fileName - The name of the file to read
    #   verbose - Optional parameter to print out
    #       extra information
    #
    # Returns:
    #   The microarray data that was parsed from the file
    #
    def read(self, fileName, verbose=False):
        # create an array to hold the microarray data
        microArrayData = []

        # open the file for reading
        with open(fileName, 'r') as csv:
            # go through each line
            for lineNo, line in enumerate(csv):
                # skip the experiment column headers
                if lineNo == 0:
                    if verbose:
                        print (filter(None, line.rstrip().split(',')))
                else:
                    # gather the gene data skipping the first column (gene label)
                    gene = line.rstrip().split(',')[1:]
                    if verbose:
                        print (gene)
                    # add the gene to the microarray data
                    microArrayData.append(list(map(int, gene)))

        # return the microarray data in the form of a 2D list
        # where each inner list is a gene's expression data
        return microArrayData


    ##
    # getLabels
```

```
    #
    # Description: Given a csv file name, opens the file
    #   and parses out the labels
    #
    # Parameters:
    #   fileName - The name of the file to read
    #   verbose - Optional parameter to print out
    #       extra information
    #
    # Returns:
    #   The labels
    #
    def getLabels(self, fileName, verbose=False):
        # create an array to hold the labels
        labels = []

        # open the file for reading
        with open(fileName, 'r') as csv:
            # return the labels (column 0)
            labels = [line.rstrip().split(',')[0] for line in csv][1:]

        # return the labels
        return labels


# Test code for directly running the file
if __name__ == "__main__":
    # initialize the reader
    csv = CSVReader()
    # read the data and printo out the results
    microArrayData = csv.read("microarraydata.csv", verbose=True)
```

10.3 Anything Else

Dataset: The dataset was obtained from lab 9 of the University of Portland Computational Biology class. The file is called “ALL-AML.txt.” The dataset is too large to add to this paper.