I. What does it mean to "cluster" gene expression data? And why would we want to?

Imagine that we have use DNA microarrays to determine the gene expression profiles associated with patient samples where we know beforehand the disease status of an individual. *supervised learning* method that was used for class prediction of tumor types based on microarray gene expression data. Using training sets of microarray data derived from patients with either acute lymphoblastic leukemia (ALL) or acute myeloid leukemia (AML), Golub *et al.* showed that for patients with leukemia of unknown etiology, microarray data was useful for classifying their disease as either AML or ALL. In other words, they had enough *a priori* knowledge of their system that they could easily establish separate ALL and AML training sets. But what if pathologists using other methods had not yet established a method to distinguish between ALL and AML? Or what if the pathologists had not yet realized that they were dealing with two separate, although admittedly related, diseases? Very often in biology, we find ourselves in this boat. This is particularly true for researchers using microarrays. Since microarrays give you an unbiased and synoptic view of gene expression, they are often used as an exploratory tool - and the scientist might have no preconceptions about what to expect. Therefore, we need tools that allow us to see patterns and underlying structure in the data that do not depend on a training set, or even knowledge of the class labels. Algorithms of this nature are called *unsupervised*, and clustering is a well-known method of this class that has found application in many scientific disciplines. In fact, there are dozens of variants of clustering methods. We will focus on just one such method – so-called *hierarchical* (or *agglomerative*) clustering. These methods are motivated by the fact that if we can group similar items together, the human mind is better able to see the basic structure of the larger whole, and we can conceive of organizing principals that let us think about the data.

If we perform several microarray experiments, each of which exposes cells to some different condition or stimulus, we end up with an estimation of the gene expression level for each gene under each condition. An important assumption here is that genes with similar expression patterns are presumed (and these has frequently been demonstrated) to share either direct or indirect regulatory commonalities. The basic goal therefore is simply to reorder the linear list of gene names so that ones behaving similarly to one another are close together in the list, and ones behaving differently are farther apart. Then, using visualization software, such as Mike Eisen's original *TreeView* or successor packages such as *Java TreeView*, we can colorimetrically encode the associated gene expression data, and see what there is to see. How do we actually do this?

II. A brief description of the algorithm for clustering gene expression data.

The description of the basic algorithm for hierarchical clustering is straightforward. Over a set of microarray experiments, for each gene we get a *gene expression vector* over the set of conditions (c) that might look something like this:
Hierarchical Clustering - 2

c1  c2  c3  c4

| geneA | 0.1 | 2.3 | -3  | 1.8 | etc. |

Here are the steps:

1. We normally start with a pool of $K$ such vectors, $K$ corresponding to the number of genes represented on the microarray (i.e., the rows), or some number of genes that were differentially expressed over the set of conditions.

2. We next do a pairwise comparison of all the combinations of vectors, and decide which pair is "most alike" or has the highest degree of similarity.

SQ1. If you have $K$ vectors, how many possibilities do you need to consider if you look at all the pairwise combinations? Assuming you have a Python function $\text{Score}(\text{vectorX, vectorY})$ that calculates the similarity between vectors X and Y, bear in mind that $\text{Score}(\text{vectorX, vectorY}) = \text{Score}(\text{vectorY, vectorX})$. Also, we know that vectorX is perfectly similar (i.e. has zero distance) to vectorX (and we can't join them into a node anyway), so there is no need to calculate self-similarity scores...

3. We then join the two "most similar" gene vectors into a node, which then replaces the two original genes in the pool. This node (or, occasionally and perhaps confusingly, pseudogene) has characteristics combining the properties of the two entities from which it was made.

4. The pairwise similarity process is then iterated, this time allowing the possibility that a gene might be more similar to a node (and therefore paired with it to make a new node) than to another gene. Later on, nodes can also be paired with nodes.

This process of making larger and larger clusters (thus agglomerative) by repeatedly joining genes and nodes results in a dendrogram or clustergram, since it is easy to see that if we continue this process, we will eventually run out of nodes to join and will end up with one big node.

SQ2. Draw a few sample trees with a low starting $K$ to convince yourself that you will always need to calculate $K-1$ nodes before you are finished. If you happen to be mathematically inclined, could you formally prove that it is always $K-1$?
An example of such a dendrogram might look like this:

```
N5
  / \    
 N4  N1   \  
    / \   |
  N3   N2 |
    /   \
 G4 G2 G6 G1 G3 G5
```

Output from TreeView, showing the clustering dendrogram on the far left, along with colorimetrically encoded gene expression vectors. The box on the right is a zoom of the highlighted pink subcluster.
SQ3 Using the *Final_with_zeros.txt* file, experiment with TreeView by clustering by gene using different similarity metrics to see how that affects the output. Notice how you can highlight groups of gene within a given subnode. Also, try altering the contrast under the "settings" tab ("Pixel settings" I think in Java TreeView). What do you suppose this does?

### III. What concepts do we require in order to actually implement this algorithm?

There are a couple of concepts here that we've only talked about in generalities that we will need to make more concrete before we can actually implement the algorithm.

**i) Distance and similarity Metrics.** So far we have talked loosely about the concept of similarity or distance between two gene expression vectors. There are actually many different measures of distance between two vectors; we will discuss only two of these, the Euclidean distance, and the Pearson correlation coefficient. Because of its basis in geometry, the Euclidean distance is perhaps the easiest to understand. Imagine first that we have only microarrays. We can then think of our gene expression vectors as co-ordinates in two dimensional space, and for some genes $geneA$ and $geneB$, we can plot each of these points and draw a line joining them. Since the shortest distance between two points is a straight line, the length of this line is the distance between these two gene expression vectors. And thanks to Pythagoras, we know how long the line is. We simply make a right-angle triangle and trot out the Pythagorean formula:

$$d = \sqrt{(X_{geneA} - X_{geneB})^2 + (Y_{geneA} - Y_{geneB})^2}$$

Similarly, the distance between two points in 3D space is:

$$d = \sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}$$

The really good news here is that this formula easily generalizes to $N$-dimensional space:

$$d = \sqrt{\sum_{i=1}^{N} (x_i - y_i)^2}$$

Even though we can't visualize it geometrically anymore, the formula below works just fine if we imagine that our gene expression vector is actually a co-ordinate in some $N$-dimensional "gene expression space" (remember $N =$ number of microarrays). It is for this reason that you will sometimes hear gene expression data referred to as having "high dimensionality". That being the case, we can now consider our original goal in a bit of a different light. What we
are really trying to do is to project our high dimensionality dataset onto a low dimensionality space (an ordered list of genes). Since we can't visualize $N$-dimensional space for non-trivial $N$ (at least I can't!), one way to think about our final clustergram is that what we will be looking at when we are done is rather like a shadow cast by the actual high-dimensional data set. Like a shadow of a three-dimensional object that looks quite differently depending on the angle of the light, our shadow might appear differently depending on the assumptions that went into our projection. As we shall see, there is no single "right way" to perform such a projection, and no one clustering method will reveal all of the structure inherent in the dataset. Therefore, to gain the most possible insight from the data, it is usually necessary to cluster data using several different methods.

SQ4. Use the Excel to experiment with calculating the Euclidean distance between a few made up gene expression vectors. What happens to the distance as you make the vectors increasingly dissimilar? What about if you make the direction of changes the same between the two vectors, but with different magnitudes (as might be the case if two genes regulated by the same transcription factor were expressed at different levels because of unequal strength of their promoters).

Another similarity metric is the Pearson correlation coefficient. I'm unconvinced that going through the derivation of the Pearson coefficient is a useful exercise for our purposes, so we'll just look at the formula and discuss a few of its properties:

Pearson Correlation Coefficient:
$$ r = \frac{\sum_{i=1}^{N} (x_i - u_x)(y_i - u_y)}{N\sigma_y\sigma_x} $$

Note that there is also a Computational form of the formula that is very useful if you ever need to implement a Python method to calculate the Pearson:

$$ r = \frac{\sum_{i=1}^{N} x_i y_i - (\sum_{i=1}^{N} x_i)(\sum_{i=1}^{N} y_i)}{\sqrt{\sum_{i=1}^{N}(x_i - \sum_{j=1}^{N} x_j)^2} \sqrt{\sum_{i=1}^{N}(y_i - \sum_{j=1}^{N} y_i)^2}} $$

Notice that in this version, there are multiple summations - and whenever you see a summation, you should immediately think of a loop. In fact, we should be able to use just one loop...all of these summations are from 1..$N$, where $N$ is the number of microarrays. The Pearson is bounded by the values -1 to 1.

SQ5. Use the Excel Pearson function to experiment with calculating the Pearson correlation coefficient between a few made up gene expression vectors. What happens to the distance as you make the vectors increasingly dissimilar? What about if you make the direction of changes
the same between the two vectors, but with different magnitudes. What about if the magnitudes are the same, but with the signs reversed (so that \textit{geneX} goes up, \textit{geneY} goes down).

SQ6. Our version of the clustering algorithm, \textit{MyCluster.py}, seems to calculate its metric using the Euclidean distance. But \textit{TreeView} seems to rely on files created using the Pearson. Convert the \textit{MyCluster.py} so that it calculates $r$. Confirm that it is working correctly by printing some output and comparing to the answer from Excel.

\textbf{ii) Clustering methods.} When we discussed calculating the distance between genes, everything is pretty straightforward - we can (for example) just calculate the Pearson correlation between the two expression vectors, and we are done. But what does it mean to say how similar a node is to another node, or to a gene? It turns out that there are multiple procedures that can be used to decide how to build clusters of nodes. We will discuss three of the most commonly encountered.

a) \textbf{Single linkage clustering.} In single linkage clustering, to calculate the similarity between two nodes, we take the distance between the closest two genes in the cluster as representative of the similarity between the two nodes:

\begin{itemize}
\item \textbf{Complete linkage clustering.} In complete linkage clustering, to calculate the similarity between two nodes, we take the distance between the farthest two genes in the cluster as representative of the similarity between the two nodes:
\end{itemize}

\begin{center}
\textbf{Single Linkage:} \hspace{2cm} \textbf{Complete Linkage:}
\end{center}

Calculating the similarity between two nodes with either of these methods is therefore a bit of a pain, since for each node you have to go back and figure out what subnodes made up those nodes, and ultimately, what genes make up the cluster (the stars in the diagram above all refer to gene nodes). Then all of the pairwise distances of the genes in both nodes must be considered. However, the good news is that since the distance that is reported as the distance between two nodes is actually always a gene-to-gene distance, it is never necessary to calculate additional Pearson or Euclidian values after the initial round of building the pairwise gene table.

b) \textbf{Average linkage clustering.} In average linkage, a node simply takes on the mean of the gene expression vector of all of its component vectors (to average vector $Z$ from vectors $X$ and $Y$, simply let the $Z_i = \text{mean} (X_i, Y_i)$ [$i = 1...N$].
Hierarchical Clustering - 7

Average Linkage Clustering:

This procedure minimizes the average pairwise distance between two clusters. Our initial version of Cluster actually does a cheat variant of average linkage... rather than average all the component vectors, it right now just averages the vectors of the two items being joined. Then this new nodes vector is treated exactly like it was itself a gene. Still, when we create nodes we will need to keep track of the two nodes or genes that were used to create that node (usually these daughter nodes are called left_node and right_node or something similar).

SQ7. Have a crack at modifying MyCluster.py to accommodate true average linkage, single linkage or complete linkage (their implementations are almost identical, pick one). To do this, you will need to keep track somehow of the weight of a particular node (i.e., how many genes does it actually represent), and use a weighted version of the averaging procedure (if you don’t know how to calculate a weighted average, google is your friend).

iii) Dicts and Trees. Since we seem to have gotten into the business of making dendrograms, we need some suitable data structure keep track of all the genes and nodes. Fortunately, a common data structure called a tree is a natural abstraction of a dendrogram. As it turns out, trees are relatively easy to implement in Python using dicts. The key point is that by using dicts, we can associate a particular unique key (such as "NODE15") with a value that is a node object that in turn contains a bunch of attributes that we want to keep track of (usually instance variables are used to do this). To implement a tree structure in this way, we minimally need to keep track of the keys leading to the left and right daughter nodes (which themselves have the same structure). Another great trick in Python is that it is possible to make a class that is aware of all of its own instances using so-called “class variables”, which are variables defined in a class outside of the scope of any method of the class. These variables will be shared by all instances of that class. Indeed, the Tree class of MyCluster.py implements just such functionality. It keeps track of nodes that are instances of Tree by storing references to Tree instances in a class variable (Tree.nodes)! It may seem confusing that instances of the Tree class are actually nodes, but remember, a node is just the root of a subtree, and a subtree is, well, actually a tree. Tree.nodes is just a plain old dict with keys that are node or gene names (taking the form “NODEX” or “GENEX” where X is a number). Note that new nodes are added to this dict right in the initializer for the Tree class (does it make sense that the value assigned is just self?)

In any case, you can see how I’ve tucked everything I might need to know about a node into one convenient structure. Pretty much all of the instance variables for the tree class correspond to attributes that are potentially of interest for tree nodes. This includes instance variables to hold the identity of the left and right daughter nodes (which of course, if we know their name, we can look up the object for them in the master Tree.nodes class variable!)
But wait! Something kind of mind-blowing is going on! Because a node has access to its left and right subnodes, and these, in turn, will know the key to their left and right subnodes (if they have any) we in fact have access to the entire subtree below any node that we happen to know the key for!

One other important property to keep in mind is that in a proper tree representation, subnodes always eventually turn into genes, which are really just special nodes that have no further left and right daughters. An interesting further consequence of this is that if we know the name of the top-level node, we can access the entire tree. All we need is some method of tree traversal to systematically visit every node!

iv) Tree traversals, recursion, and the depth-first search. A tree traversal is a method for visiting all of the nodes of a tree in some ordered fashion. But first of all, why do we need to traverse this dendrogram that we’ve just built anyway?

SQ8. Run MyCluster.py with ratiodata.txt, and try and draw by hand the dendrogram. Start by writing gene1 through gene 16, and then connect them with nodes the way that the MyCluster.pl output suggests is correct. What problem do you run into? Does an analysis of the output from the Cluster program run with ratiodata.txt (ratiodata.cdt and ratiodata.gtr will be automatically created) suggest a solution (check the .cdt file especially)?

It turns out that if we keep the gene order the way it was in the input file, we really haven't accomplished much. Trying to draw a dendrogram inevitably leads to crossed wires for any non-trivial dendrogram. What we need to do is to reorder the gene list in such a way that we can draw a dendrogram without getting wires crossed, while still using the clusters specified by the MyCluster.py output. How will we find an ordering that works? Luckily, while building our dendrogram, it has been defining its own structure as we went along (note, for instance, the ordering of the genes in the example on pg. 3 here). In other words, we already have an internal representation of what the tree should look like, we just need some way for the tree to go from left to right and spit out the order of the genes. But remember, since we know the name of the key for the root node, we have access to the entire tree. One way we could have it print the genes from left to right would be to give it the key to the root node, and tell it to burrow its way down into the tree by always going down the left path for each subtree until coming to a gene (which is then printed). Then it would go back up to the node above, and explore the subtree under the right node (always exploring left subtrees first) until a gene was encountered and printed. Only when the entire left and right subtrees are exhausted would the algorithm proceed to the next higher node. This is called a depth-first search (DFS), and it is a classic operation on tree-like data structures. Do you see why this works?

SQ9. It is much easier to convince yourself that this procedure works, and to understand it, if you try it by hand. Once again, draw a dendrogram based on the output of MyCluster.py and ratiotext.txt, but this time start at the root note and follow the DFS procedure, drawing nodes and genes as you come to them. Do you end up with any crossed wires this time?
OK, perhaps you have convinced yourself that this works on paper, but how in the world would you implement this in Python? The key observation is that the procedure works exactly the same way for every subtree - even subtrees of subtrees. It's doing the same operations over and over and over again, which can be summarized as:

1. Does self.name of the current object correspond to a NODE or a GENE? If it is a GENE print out its required information to the .cdt file and return.

2. If it is a NODE:
   a. Does it have a left daughter?
      i. Yes, visit the left subtree
   b. Does it have a right daughter?
      i. Yes, visit the right subtree

3. We are done with this subtree, so go back up to the next highest node (return)

The way that this is actually accomplished in Python is by using recursion. This is, amazingly, when a method or function calls itself. Recursion is a natural (although not always the most efficient) way to solve many problems, especially those that have a representation as trees. The key point to keep in mind is that in order to avoid an infinite loop, there must always be a termination condition that stops the function from calling itself forever. In the case of our tree traversal, this comes from the fact that there are not an infinite number of subtrees...eventually we are guaranteed to come to genes in both the left and right subtrees, and accordingly will have to return up a level. Eventually, we will have exhausted all possible subtrees and will wind up at the root node again (or at whatever node we made the original call). In order to help you better understand recursion, I have provided a program called Mergesort.py, which implements a recursive algorithm for sorting a list. We will review this program in class.

The essential idea is that the list to be sorted can be repeatedly broken down in size by one half until there each sublist has only one character in the right node and one in the left node. The right and left halves are merged in sort order to make a sorted sublist. Sorted sublists are then repeatedly merged back into one another until we are left with a sorted list. Although described very early in the history of computer science (by John von Neumann, in 1945), this is actually one of the more efficient sorting algorithms, typically requiring fewer comparisons than other algorithms, such as quicksort. Note that in this case, since we are starting at the top node and working downwards (i.e. working divisively rather than agglomeratively), there is no need to explicitly keep track of the node structure...this happens auto-magically. This is because when one of the nested calls of Mergesort returns, it goes back to the line immediately following where it was called from - so once it reaches a gene it automatically unwinds itself until reaching a node with an unvisited subtree. When all subtrees have been visited, it will return from the original call of Mergesort, and we are done. Very cool, and it makes for very compact code.

SQ10. Based on the output from Mergesort.py, follow through by hand how the example string is progressively sorted by merging sorted substrings. Once you feel comfortable with the sorting procedure at an intuitive level, study the code, observing how a tree data
structure applies to such a 'divide and conquer' approach. See then if you can apply this same logic to flush out the DFS() method in MyCluster.py in order to print the required information to your .cdt file. This one is not as hard as it might seem at first! Again, recursive code is quite satisfyingly compact – if you are writing pages of code you are definitely overthinking this.

SQ 11. Once you have finished the goals of SQ6 and SQ10, we should have produced a program that is compatible with TreeView. Once it seems to be working properly with ratiodata.txt, try it on the BacillusData2.txt file, and try loading the resulting files in Java TreeView. Now we can try and decide which genes are downstream of the Spo0A transcription factor (required for entry into sporulation), and those that also require activation of the later transcription factor $\sigma^F$. The columns in BacillusData2.txt are interpreted as follows:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wt0A T-1.5</td>
<td>Wt (red channel) vs. Spo0A mutant (green channel) at 1.5 hours BEFORE sporulation (Spo0A is not active at this time)</td>
</tr>
<tr>
<td>Wt0A T0</td>
<td>Wt (red channel) vs. Spo0A mutant (green channel) at T0, the time when Wt commits to sporulating and Spo0A becomes active.</td>
</tr>
<tr>
<td>Wt0A T2</td>
<td>Wt (red channel) vs. Spo0A mutant (green channel) 2 hours after Spo0A first became active.</td>
</tr>
<tr>
<td>WtsF T2</td>
<td>Wt (red channel) vs. $\sigma^F$ mutant (green channel) at the time when $\sigma^F$ has just become active.</td>
</tr>
</tbody>
</table>

SQ 12. I mentioned that there are actually many ways that we could display the same dataset. In fact, there are many representations of even one tree - any or all of the nodes in our final dendrogram could be flipped (their left and right subnodes could be swapped). This would produce an equally valid tree. Can you convince yourself that there are $2^{N-1}$ valid orderings for any tree containing $N$ items?