COMPSCI 260, Introduction to Computational Genomics **28 October 2016**

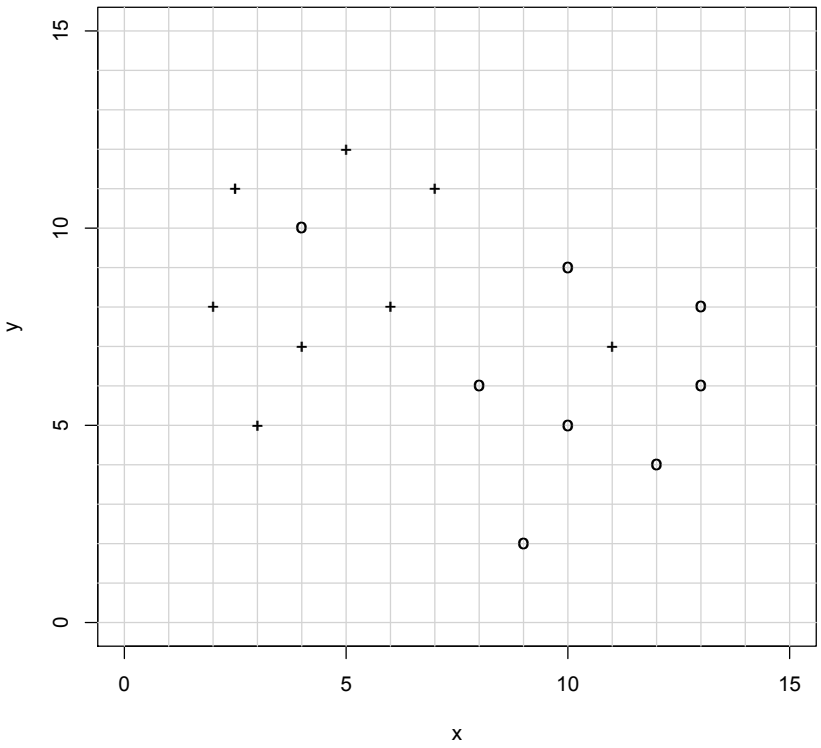
Problem Set 5

Due: 5pm on 11 November 2016

Problem 1: Love thy neighbor as thyself (25 points)

One simple but powerful way of classifying is by way of the non-parametric classification strategy called 'k-nearest-neighbor'. In this approach, unlabeled samples are assigned class labels based on the labels of the k labeled samples that are closest, taking a majority vote (for this reason, when there are two classes, we always choose k to be odd so that there won't be ties).

a) Consider the 16 labeled, two-dimensional training samples presented in the Cartesian plot below. Eight samples are labeled '+' and eight samples are labeled '$\circ$'. For example, these could represent the gene expression levels of two key biomarker genes (plotted on the x and y axes) from 16 patients, half of whom responded to a particular treatment ('+', the responders) and half of whom did not ('$\circ$', the non-responders).



Imagine five new patients walk into your clinic, and you measure the expression levels of the two biomarker genes for each patient. When the results come back from the lab, this is what you see:

Name	Gene expression profile	Class when $k = 1$	Class when $k = 3$
Alice	(2, 7)		
Bobby	(10, 7)		
Cindy	(8, 4)		
Donny	(4, 9)		
Ellen	(8, 8.5)		

What prognosis are you going to give each patient in terms of their response to the treatment? Consider both $k = 1$ and $k = 3$. Although k-nearest-neighbor just assigns a simple label to each case, comment on how confident you are in your prognosis in each case.

b) Let's scale this up a little bit. Imagine that you now have 10 biomarker genes (so the points are in 10-dimensional space, or $\mathbb{R}^{10}$; we won't ask you to draw this). Imagine also that you now have a database of gene expression values for all 10 biomarker genes for 1000 patients who responded to treatment (class R) and 1000 patients who were non-responsive (class N). This data is all contained in the file `gene_expression_training_set.txt`.

This time, ten new patients walk into your clinic, and you measure the expression levels of the ten biomarker genes for each patient. When the results come back from the lab, they are contained in the file `gene_expression_test_set.txt`.

Building on the skeleton code we provide in `KNearestNeighbors.py`, write Python code to give a prognosis for the ten new patients using $k = 5$.

Extra Challenge: For a little extra challenge, you may wish to write your code to take k as a parameter and see how sensitive the result is to your choice of k (you should require k to be odd). If you do this, one benefit is that you can confirm your answers to part (a) pretty quickly. For a different extra challenge, you can report not only the final class label assigned to the new patients (R or N), but also the distances and labels of the $k = 5$ closest labeled patients in the database: this will help you assess the confidence of your prognoses.

Problem 2: Sampling from a discrete event space (15 points)

Imagine that we have a sample space which is a set of n mutually exclusive events $e_1, \ldots, e_n$ with associated probabilities $p_1, \ldots, p_n \geq 0$, such that

$$\sum_i p_i = 1$$

Now consider the process of repeatedly sampling an event from this set, and doing so a total of m times. In each of the m samples, exactly one event from the set is selected according to its probability, and each sample is independent of the others.

a) Write a Python program `sampling.py` where it takes time linear in n to draw one sample from this event space according to the discrete probability distribution given by the p_i's. If drawing a single sample takes time linear in n, then drawing m samples will require $\Theta(mn)$ time. You should make use of Python's built-in pseudorandom number generator `random.random()`, which generates a new value distributed uniformly between 0 and 1 each time it is called.

You used Python's built-in `random.random()` function in the first problem set, but then the events were all equally likely; here you are given the probabilities of events as input (and they need not be equal) and every time you call your program, it should return one of the events with the appropriate probability.

For example, if you input $\{0.5, 0.3, 0.2\}$ as the event probabilities (p_1, p_2, and p_3) to your program and generate $m = 1000$ trials, you would expect to see around 500 results for e_1, 300 for e_2, and 200 for e_3.

b) Add a function to the Python program `sampling.py` that accomplishes the same task as in the previous problem, but requires only $\log n$ time per sample rather than linear time, which would be important if you have to sample from a very large set. Initialization time is not counted against the $\log n$ requirement, but initialization time shouldn't scale with m (for instance, you can't pre-compute the result of m samples and just call that initialization time). This means the total runtime to generate m samples (after initialization is done) should now be $\Theta(m \log n)$. What is the asymptotic running time of your initialization step?

Problem 3: Ultrametricity and additivity (25 points)

In class we discussed two algorithms for building phylogenetic trees: UPGMA (unweighted pair group method using arithmetic averages) and NJ (neighbor joining). Both algorithms use distances between sequences to reconstruct the phylogenetic trees. UPGMA is guaranteed to reconstruct the correct (rooted) tree if the distance metric is an *ultrametric*, while NJ is guaranteed to reconstruct the correct (unrooted) tree if the distance metric is *additive*.

Given four sequences X_1, X_2, X_3, and X_4, Table 1 shows the distance between each pair of sequences. Let's call this distance metric D_1. Similarly, for the five sequences Y_1, Y_2, Y_3, Y_4, and Y_5, Table 2 shows a distance metric D_2.

D_1	X_1	X_2	X_3	X_4
X_1	0	0.3	0.6	0.5
X_2		0	0.5	0.6
X_3			0	0.9
X_4				0

Table 1

D_2	Y_1	Y_2	Y_3	Y_4	Y_5
Y_1	0	0.8	0.6	0.8	0.6
Y_2		0	0.8	0.4	0.8
Y_3			0	0.8	0.2
Y_4				0	0.8
Y_5					0

Table 2

a) Write two Python functions, `is_ultrametric()` and `is_additive()` in the file `UltrametricAdditive.py`, that receive as input the pairwise distances for a set of sequences and then verify whether the given distances satisfy the ultrametricity and additivity criteria, respectively.

Do *not* use a matrix structure to represent the distances between two sequences; instead use the following dictionary representation, as illustrated in the script `UltrametricAdditive.py`:

dist = {"1,2" : $d_{1,2}$, "1,3" : $d_{1,3}$, "1,4" : $d_{1,4}$,
"2,3" : $d_{2,3}$, "2,4" : $d_{2,4}$,
"3,4" : $d_{3,4}$}

where $d_{i,j}$ is the distance between the i^{th} and j^{th} sequences.

b) Use your Python functions to check if the given distance metrics satisfy the ultrametricity and the additivity criteria. For each of the two distance metrics D_1 and D_2:

1. Test the ultrametricity criterion.
   - If one or both of them is not ultrametric, give an example of a triplet that does not satisfy the ultrametricity criterion.
   - If one or both of them is ultrametric, build the (rooted) phylogenetic tree that would be reconstructed by UPGMA by hand (i.e., on paper).
2. Test the additivity criterion.

- If one or both of them is not additive, give an example of a quadruplet that does not satisfy the additivity criterion.
- If one or both of them is additive but not ultrametric, build the (unrooted) phylogenetic tree that would be reconstructed by NJ by hand (i.e., on paper).

c) Recall the superoxide dismutase proteins we considered in the previous problem set. The table below contains, roughly, the distance metric for those four protein sequences. Using the code you've written in `UltrametricAdditive.py`, determine whether this distance metric satisfies the ultrametricity and additivity criteria. Based on the results of your check, select an appropriate tree construction algorithm and build the phylogenetic tree that would be reconstructed by that algorithm by hand. What do you observe? Compare the relationships between protein sequences described by the reconstructed tree with the relationships you inferred between protein sequences based on their pairwise alignments in the previous problem set.

	SODM_HUMAN	SODM_ECOLI	SODM_BACSU	SODM_MOUSE
SODM_HUMAN	0	0.5	0.5	0.1
SODM_ECOLI		0	0.3	0.5
SODM_BACSU			0	0.5
SODM_MOUSE				0

Problem 4: The origin of SARS (35 points)

a) It's early 2003 and you are enjoying a well-deserved vacation at the Metropole Hotel in Hong Kong. You're resting in the spa with cucumbers on your eyes when you are jolted into consciousness by a sharp knock on the door. Hotel staff are streaming into the room and—owing to your reputation as an eminent scientist—are asking urgently for you to weigh in on the likely origin of SARS: how did this particular deadly viral genome sequence arise? Was it a mutation in an existing human coronavirus? Or did an animal coronavirus make the jump into a human host?

Armed with nothing but your wits, endless fluffy spa towels, and a portable Python interpreter, you set to work. You request a set of orthologous 'spike' protein sequences from a range of coronaviruses isolated from different animal species. Within seconds, the hotel bellhop waltzes in with the `CoV.fasta` file on a silver platter; you tell yourself this guy deserves a big tip.

The table below shows the Genbank IDs of all 11 spike proteins, but in his hurry, the bellhop neglected to notate the virus species from which each protein came. Fill in the table with the names of the corresponding viruses, as reported in Genbank in the "protein" database. You should also note the animal host for each species of virus.

Index	Protein ID	Virus name	Animal host
1	AAL57308.1	Bovine coronavirus BCoV-LUN	cow
2	AAK83356.1		
3	AAR01015.1		
4	AAL80031	Porcine hemagglutinating encephalomyelitis virus (HEV)	
5	YP_209233.1	Murine hepatitis virus strain JHM	
6	NP_045300.1		
7	NP_040831.1		
8	NP_598310.1		
9	BAA06805		
10	AAP41037.1		
11	AAV49723		

b) Just as you finish filling in the table with the names of the viruses, the bellhop rushes back in to bring you another file, `CoV.aligned.fasta`, which contains a multiple alignment of the 11 spike proteins (apparently, the bellhop knows how to use ClustalW). You give him another big tip for his efforts.

Write a Python function that takes as input the indices of two proteins, as given in the table above, and returns the distance between the two proteins, computed from the given alignment. Here, you decide to define the distance between two aligned proteins to be the number of mismatches in the alignment, divided by the total number of matches and mismatches (*i.e.*, all positions where there are no gaps in either of the sequences): it's quick-and-dirty, but time is of the essence here. Include this function in the `BuildTree.py` file.

Compute the pairwise distances for the 11 proteins, and store them in a dictionary data structure similar to the one used in Problem 1.

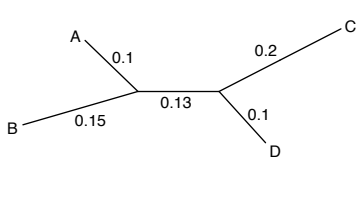
c) Use the Python functions `is_ultrametric()` and `is_additive()` from Problem 1 to verify whether the distance metric computed above satisfies the ultrametricity and/or additivity criteria. Since you're working with real protein sequences, it is rarely the case that two distances (or two sums of distances) will be exactly equal. So in the two functions `is_ultrametric()` and `is_additive()`, this time you should substitute conditions of the form `dist["i,j"] == dist["j,k"]` with: `is_almost_equal(dist["i,j"],dist["j,k"])`. The function `is_almost_equal` is defined in the Python script `BuildTree.py`, and it simply verifies whether the difference between the two parameters is negligible enough that the parameters can be considered equal.

Are the distances computed in part (b) ultrametric? Are they additive? Can we apply the UPGMA and/or NJ algorithms? Why or why not?

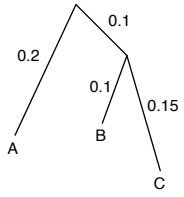
d) Luckily you find in an obscure folder on your computer a Python file called `BuildTree.py` which seems to contain skeleton code for implementing the NJ algorithm. You can definitely use it to reconstruct the (unrooted) phylogenetic tree of the 11 proteins. Unfortunately the program is missing certain important lines that actually make it work. In the section marked by the comments `#Begin - your code#` and `#End - your code#` fill in the lines as instructed by the comments, to make this program work.

The program already contains code to output the reconstructed tree in the Newick format. This output will be required in the next part of the problem.

The Newick format is described in detail at http://evolution.genetics.washington.edu/phylip/newicktree.html. Briefly, the Newick format is a computer-readable format for representing trees using nested parentheses. Each leaf node is represented by its name (*i.e.*, a string of characters except blanks, colons, semicolons, parentheses, and square brackets). Each internal node is represented by a pair of matched parentheses, with the children nodes between the two parentheses, separated by commas. The length of a branch connecting a node to its parent can be specified by putting a colon after the node, followed by a real number representing the length of the branch. The terminating semicolon specifies where the tree representation ends. Please see the examples below.



Newick representation:
((A:0.1,B:0.15):0.13,C:0.2, D:0.1);



Newick representation:
(A:0.2,(B:0.1,C:0.15):0.1);

Note that for an unrooted tree, the Newick representation is not unique. The unrooted tree on the left above can also be represented as: ((A:0.1,B:0.15):0.13,(C:0.2,D:0.1):0);

e) As soon as you finish reconstructing the phylogenetic tree of the spike proteins, you call the hotel staff to show them your findings. They look at the phylogenetic tree in the Newick format, but seem perplexed.

They do not understand what the set of parentheses, numbers, and letters means. Luckily, you remember the "Phylip drawtree" program that you used in your Computational Genomics class. You quickly Google the name of the program and find the web server: http://mobyle.pasteur.fr/cgi-bin/MobylePortal/portal.py?form=drawtree.

Use the online tool "Phylip drawtree" to visualize the phylogenetic tree built in part (d). You may wish to make use of the advanced options "Angle of labels: Along" and "Try to avoid label overlap". You may need to refresh your browser when the job is finished. Save the tree in a PostScript file named `SpikeTree.ps`, and study it.

Now, what are you going to tell the staff at the Metropole Hotel: How do you think the SARS virus arose?