Programming for Bioinformatics | BIOL 7200

Ungraded extras

Assignment description

This document describes some example projects you may wish to set yourself in order to practice the concepts we have covered during the Python portion of this course. None of these assignments will be graded. However, if you have questsions about them, we can discuss any of these problems during review sessions or in the discussion section of Canvas.

1. Refactoring and extending your Needleman-Wunsch module

There are a number of improvements you might want to make to the modules you wrote to perform Needleman-Wunsch alignment. First, the code you wrote to implement the algorithm is likely split up into multiple functions which all work with some form of nested list structure. As noted in the session on Python classes: whenever you have multiple functions that are highly specific to a certain format of data, those functions may be more sensibly collected into a class. That is certainly the case here. Additionally, returning an instance of a class as the final result will be a tidier and more flexible way to organize the different outputs you might want to get out of the alignment. For example, at minimum, you would want an optimal alignment and the score. You may also want *all* optimal alignments and the score and pointer matrices to be returned. Returning an instance of a class with all of those data as attributes allows you to receive just a single object as an output and then reference the attributes corresponding to whichever data you need at that time. Using a class like this is much cleaner than returning a tuple of objects (some of which may be `None` depending on the arguments given). This is how some core Python packages work, for example `re` and `subprocess.run`.

Below are some suggestions for changes you may want to make to your Needleman-Wunsch alignment module:

- Refactor the functions into a single class
- Write a classmethod to perform Needleman-Wunsch alignment and return an instance of the class containing all the data
- write a method that can find *all* optimal alignments
- write a classmethod and associated methods to perform the Smith-Waterman algorithm. This algorithm is a modification of Needleman-Wunsch that performs local alignment (i.e., it doesn't need to include all of both sequences). You should be able to modify a lot of your Needleman-Wunsch functions to work for either, given an argument specifying which to do.
- Write a method to view the score or pointer matrix. This could simply draw a matrix in the terminal to allow you to see a representation that is clearer than printing a list of lists, or it could use matplotlib to create a proper visualization with arrows showing the pointers at each position. If you are feeling adventurous, you could write something that identifies all paths through the matrix and colours arrows that are part of the paths. Alternatively, you might only colour the arrows that are part of a specified path (e.g., the path that gives the Nth alignment returned).

2. Extending your magnum opus to be even greater

Part 1

In you isPCR and Needleman-Wunsch modules, you already have a useful bioinformatic tool. However, there is another reasonable extension that you may wish to perform to make it even more impressive.

As a necessary part of performing Needleman-Wunsch alignments, you calculate a score that represents how similar two sequences are. The higher that score, the more similar the sequences. Of course, longer sequences can have higher scores. If your match score is +1, then two identical sequences of 5bp will have a score of 5, while two 10bp sequences will have a score of 10. In order to compare between sequences of different lengths, you will need to normalize those scores somehow. However, you *do* basically have a tool that allows you to quantify the similarity or difference between any pair of sequences.

What if you wanted to assess the relatedness among a set of more than two sequences? That's a very common thing to want in bioinformatics. For example, if you had a selection of different organisms, you might want to know which are more related and which are less related. This task is to extend you Magnum Opus to tackle exactly that scenario.

As described above, you already have a score representing the *similarity* between two sequences. You also, therefore have a score representing their difference (i.e., the inverse of the similiarity). If you start with a set of multiple sequences you can generate a pairwise distance matrix (or similarity if you prefer) for those sequences by simply running your existing code and storing the normalized scores.

Once you have a distance matrix, there are numerous methods to identify hierarchical relationships among your samples by using clustering. A commonly used algorithm in cases where people want to identify relationships among organisms is UPGMA. Also the implimentation of UPGMA is similar to Needleman-Wunsch (in that is involves manipulating a matrix), but different enough that this should be a fun challenge. You should be able to figure out how to implement UPGMA based on the description on the Wikipedia page.

This exercise is to use your Magnum Opus package to create a distance matrix for a set of assemblies and then use UPGMA to infer a newick tree representing their relationships.

Think about how you can use the UPGMA algorithm to construct that tree as a newick string. This may seem like a very difficult puzzle, but the process of the UPGMA algorithm lends itself well to joining together strings into a tree.

You can use the provided assemblies and 16S primers in the "16s_tree/" dir in the provided tarball to test your UPGMA implementation. Note that the provided primers (which amplify part of the 16S ribosomal RNA gene) often produce multiple amplicons for a single assembly as multiple copies of the gene are frequently encoded. You should be safe just picking one of the amplicons from each assembly, but you might want to explore whether all the amplicons from a given assembly end up clustered in the tree (if not then picking one would be inappropriate). Whether to include all amplicons in the tree or pick one could be a command line option for your program. You should get something like the following tree (when picking a single amplicon).

```
((Ferroplasma_acidiphilum_Y:74.25,
(Nitrososphaera_viennensis_EN76:48.5,Sulfolobus_islandicus_M.16.27:48.5):25.75):11.
29,
((Mycobacterium_tuberculosis_H37Rv:53.0,Treponema_phagedenis_strain_B43.1:53.0):3.5
2,((Staphylococcus_aureus_NCTC_8325:46.88,(Pseudomonas_aeruginosa_UCBPP-PA14:35.25,
(Escherichia_coli_K12:18.5,Vibrio_cholerae_N16961:18.5):16.75):11.62):2.25,Wolbachi
:49.12):7.39):29.02);
```

In addition to the example data provided, you should be able to make a 16S tree of basically any prokaryotic organism, so feel free to download random assemblies off NCBI and try out your magnum opus on them!

Part 2

Some genes or regions of DNA (herein collectively called "loci"; singular "locus") accumulate mutations at different rates to others. There are numerous possible reasons for this phenomenon. For example, regions under purifying selection (such as genes that perform essential functions) do not tolerate mutations. i.e., mutations in essential genes greatly reduce the fitness of an organism and so that organism produces fewer progeny. Conversely, regions under diversifying selection (such as genes encoded by pathogens which are targets of the host immune system) tolerate mutations much better. Indeed, when regions are under negative frequency-dependent selection (i.e., selection that favours rare variants, such as in the case of immune escape mutations), new mutations provide the organism with a benefit.

Because different loci change at different rates, different regions can provide resolution of different time scales. A region that changes very quickly may saturate all possible mutations more rapidly than a region that changes very slowly. Therefore, if you are interested in comparing the relationships between very distantly related organisms, you would do better to select a region present in all the organisms that changes very slowly. The 16S rRNA gene used in the above part is an example of exactly this sort of region. The tree shown above includes Bacteria and Archaea, which diverged billions of years ago. The 16SrRNA gene is suitable for such an analysis as it is essential for both groups of organisms, and mutations in its sequence are rarely tolerated.

When comparing more closely related organisms, the 16S rRNA gene alone typically does not provide good resolution. For example (not real numbers), if you imagine that the 16S gene accumulates a mutation roughly every 100 million years (I'm just making up this rate), then you will be unable to resolve differences between any organisms who had a common ancestor more recently than that. Instead, different loci, that mutate more rapidly, would likely provide better resolution of differences.

Appropriate regions to use will depend on the organisms being compared. Specifically, how diverse are the organisms being compared, and which loci do they all have? Usually, it is best to only use regions shared by all organisms being compared as it is hard to quantify how different an organism is if it is lacking a region. This makes sense. How much more different is a missing gene than a gene with 5 mutations?

In some cases, a single region may still not provide good resolution. In general, the more sequence you use in your analysis, the more information you have, and the smaller a difference you can resolve. This is complicated by a feature of prokaryotes that is less of a concern in Eukaryoted - horizontal gene transfer (HGT). HGT is the process by which DNA can be transferred from one cell to another and incorporated into the genome of the recipient. In some cases the donor and recipient organism are closely related, but often they are distantly related. Horizontally acquired DNA has the sequence of the donor organism, not that of the recipient (of course - it came from the donor). This DNA is therefore not useful in an analysis that attempts to reconstruct the historical relationship between the recipient organism and some other organisms.

Due to the above considerations, it is common for the loci used for each scope of comparison to be different. Typically there will have been a long history of research and argument in the process of settling on the best set of loci and changes are very common. However, in all cases, once a set of loci have been chosen, an analysis is usually performed in the same way: by combining the sequences into a single sequence by stitching them together (concatenating them) and considering them as a whole.

This concatenation process is achieved by first aligning each locus separately using a global alignment. After each locus is aligned, the alignments are stitched together. Alignments are performed before stitching so that no sequence from each locus ends up aligned to a different locus. As the loci are typically not contiguous in the genome, such an alignment would make no sense. Once the alignments are concatenated, a tree can be inferred as in part 1.

In the example data provided with this document, I have provided primers used to amplify the typing loci currently used to compare members of the bacterial genus *Pseudomonas*. I have also provided you with various *Pseudomonas* assemblies. Your task for this part is to extend your magnum opus to support being provided with multiple sets of primers and to infer a tree using a concatenated alignment of the amplicons produced using those primers.