Biothon: A dependency-free bioinformatics library, written in Python.

Daniel Famiyeh

Abstract

This paper details the design and development of Biothon, a project in the field of bioinformatics, written in Python 3. The toolkit provides a library of functions and structures that facilitate the analysis of biological data, namely DNA, RNA and protein sequences. Numerous functions for global and local pairwise sequence alignments allow for regions of similarity and conserved domains to be identified between pairs of biosequences. Multiple sequence alignment is integrated into the library, using a distance-based, neighbour-joining implementation of the ClustalW algorithm. This same idea of hierarchical cluster analysis also facilitates the construction of phylogenetic dendrograms from lists of biosequences allowing for potential homologies between closely and distantly-related taxa to be shown visually. The library is standalone, requiring only the Python Standard Library to use.

I. Introduction

Bioinformatics is a field of study that can be described as the union between the fields of mathematics, computer science and biology. It entails using mathematical techniques, to derive exact or heuristic algorithms that reduce the dimensionality of biological data allowing for meaningful, optimal analysis with reasonable space and time complexity. Biologists in the present-day interface with bioinformatics tools and software all the time. The power of modern computation means that toolkits can be hosted on web servers or take the form of computer programs that do not require insane levels of technical expertise to use. Those with programming experience may opt to use languages suited to data analysis such as R, Scala, Julia or Python. Using a programming language gives the bioinformatician freedom to create general or highly-specific tools suited to the task at hand. Biothon extends the Python Standard Library by providing a layer of abstraction between the language, biological data and common functions that one would wish to perform on said data allowing for users to perform computation on sequences with ease.

II. Sequence Representation

The Seq Class

The Seq class is the most fundamental structure in Biothon, through which all functional computation and analyses are performed. Each sequence constructed as a Seq object must have, at least, a string of characters representing biological data and a *SeqType*. The SeqType is an enum with enumerables *SeqType.DNA*, *SeqType.RNA* and *SeqType.PROTEIN*. This SeqType is used initially to assert that no invalid characters are present in the string passed during construction. A character is deemed invalid if it is not part of the respective sequence type's alphabet. For all sequences the IUPAC's unambiguous 1-letter alphabet is used. SeqType is also used to ensure that only like-sequences are aligned together and to limit functionality as required. Optional keyword arguments that can be passed during construction include a name and description, the former used when printing the sequence to the terminal and both are used when writing sequences to a fasta file. The Seq class has two more attributes, a weight and a label, these attributes are used during phylogenetic tree construction and ClustalW and serve no purpose to the user.

Seqs share much of the same functionality as strings. They can be sliced in the using the same [start_index:stop_index] syntax and iterating over a Seq iterates over the characters in its sequence. Seqs sharing SeqTypes can be concatenated using the '+' operator and Seq equality can be tested by using the '==' operator. Two Seq objects are equal if their sequence strings are equal. Since SeqTypes enforce an alphabet, checking that these strings are equal implicitly checks that they share SeqTypes too. Concatenation can be performed in place using the '+=' operator and applying the *len()* function on a Seq object returns the length of its sequence.

Functional Methods

comp() *and* ***rev_comp()****:* Complement and reverse complement methods, defined for DNA and RNA sequences. Both methods replace sequence residues with their complement, however, *rev_comp()* also reverses the direction.

- A $\iff$ T (U for RNA)
- C $\iff$ G

The inversion operator '$\sim$' also returns the complement of a sequence, out-of-place.

scribe(*args) *and* ***back_scribe()****:* Transcription and back-transcription methods; The former is defined for DNA sequences and the latter for RNA sequences. The *scribe()* method replaces thymine bases with uracil and the latter performs the inverse operation. Arguments can be passed to *scribe()* in the form of strings representing introns. These will be spliced from the resulting sequence as per the function of a spliceosome.

slate(*args)*:* Translation method. Although physically defined only on RNA, calling *slate()* on a DNA sequence calls *scribe()* first to transform the Seq into an RNA Seq which is then translated. The result, either way, is a protein sequence of 1-letter amino acids generated by reading through the RNA sequence one codon at a time.

Statistical Methods

gc_content()*:* Defined only for DNA sequences, this method returns the base composition (G-C content) of the sequence.

transit(seq)*:* Transition-transervsion ratio method. This is defined only for DNA sequences and requires another DNA sequence to be passed as an argument. This method returns the transition-transversion mutation ratio between the two sequences. Note that substitutions (mutations) are calculated via hamming distance. A requirement of this distance metric is that the sequences be of the same length.

point_mutations(seq)*:* Point mutation method. Defined for all sequences, and requiring another like-sequence of equal length as an argument, this method returns the number of point mutations between the two sequences.

find_motif(subseq)*:* Defined for all sequence types, the method returns a list of the indices where the subsequence passed as an argument can be found.

Probabilistic Sequence Generation (Markov Model)

Sequences can be randomly generated to an arbitrary length using the *MarkovModel* class. During construction a Markov-Model object can be passed a sequence type, transformation matrix and initial probability distribution vector denoted by the keyword arguments *type*, *tm* and *pi* respectively. If either the matrix or vector is not passed during construction a random one will be generated using private *_rand_tm()* and *_rand_pi()* methods and if no type is passed a DNA SeqType is assumed. The model has only one public method called *sequence()* taking, the length of the desired sequence as a parameter. It returns a Seq object of said length with random residues based on the probabilities contained within pi and tm.

III. Alignment

Sequence alignment entails the arranging of biological sequences in order to identify correlations that could indicate functional or structural similarities as well as homology. In global alignment we seek to align all residues in different sequences with one another, useful for genomes that inherently share similarities. Local alignment deals with finding small regions of similarities in otherwise large divergent sequences, usually to identify conserved regions that would indicate functional conservation. Pairwise alignment deals with aligning pairs of sequences together, while multiple sequence alignment deals with aligning a set of sequences together. The latter is performed for the same reasons as the former, but tends to be better at showing evolutionary relationships and shared patterns across families of sequences that would otherwise seem divergent using a pairwise approach.

The Basic Idea and the Gap Penalty

Take the two sequences $S_1 = ATCGT$ and $S_2 = TGGTG$, what is the most optimal way to align these sequences? First we need to define a scoring system to distinguish a good alignment from a bad one. The simplest scoring system is identity, meaning we score +1 for a match and 0 for anything else. Lining these two sequences yields:

$$ATCGT$$
$$TGGTG$$

No letters match up, so this has a score of 0. If we allow for the idea of 'gaps' to shift sequences though we could arrive at:

$$A\mathbf{T}C\mathbf{GT}-$$
$$-\mathbf{T}G\mathbf{GT}G$$

Which is a better alignment in our defined system, with a score of 3. We do need to penalise gaps to some degree however. For instance, if we allow gaps without penalising them, then the above alignment and this sub-optimal alignment

$$--A--\mathbf{T}-C-\mathbf{GT}--$$
$$-----\mathbf{T}-G-\mathbf{GT}-G$$

will have the same scores. Clearly these alignments are not equal. The latter is more 'noisy' with unnecessary gaps obfuscating the sequence. From an evolutionary standpoint, we generally consider gaps to be point mutations that mean that residues have been inserted or removed from a sequence. Bearing this in mind, the optimal alignment suggests each sequence has only experienced one insertion/deletion (indel) event. The sequences in the last sub-optimal alignment suggest that more of this mutation event has taken place, suggesting that this pair are more divergent than the optimal pair. So, once again, we arrive at the conclusion that these alignments are not equal, with some biological theory to support this notion. To mitigate this, we introduce a variable, say δ, which denotes a constant gap penalty applied every time a gap is placed in either sequence. So that the optimal sequence has a score of $3-2\delta$ and the last alignment has a score of $3-16\delta$, indicating that the former is the more optimal of the two.

Affine Gap Penalty

Most modern approaches do not use a constant gap penalty, opting for what is known as an *affine gap penalty*. The idea stems from the fact that chains of consecutive gaps likely stem from a single evolutionary event. Consider the following two alignments:

$$AT---GC$$
$$ATTGAGC$$

And the alignment,

$$A-TG--C$$
$$ATTGAGC$$

Applying a constant gap penalty would result in both alignments receiving the same the score. The second alignment is less likely to have occurred than the first one however as two chains of gaps suggest that two evolutionary events took place to cause them. The first alignment has one chain of gaps that separate the first sequence from the second, which is likely to have occurred from one event, and is overall more likely. It stands to reason that a more accurate gap penalty will penalise gap openings more than gap extensions, and this is the affine gap penalty; Defined by,

$$w(k) = -\alpha - \beta(k-1)$$

where α denotes the gap open penalty, β the gap extension penalty and k is the gap length.

Substitution Matrices

Substitution matrices or scoring matrices provide a way of scoring matches/mismatches that is more true to experimentally observed biology. The two types of the two types of substiution matrices used in Biothon are the transition/transversion matrix for DNA sequences and the BLOSUM matrix for amino acid sequences.

***Transition-Transversion Matrix**:* This scoring matrix takes into account the fact that transition mutations occur approximately twice as often than transversions. Transition mutations are purine-purine ($A \iff G$) or pyrimidine-pyrimidine ($T \iff C$) mutations that involve residues of a similar shape. These types of mutations hence preserve structure to some degree and are not penalised as much as transversion mutations from single-ring pyrimidines to double-ring purines and vice-versa.

***BLOSUM-62 Matrix**:* The BLOSUM matrix is a substitution matrix for the aligning protein sequences, based on experimental data from the BLOCKS database. There are different versions of the BLOSUM matrix with BLOSUM80 being used for more closely-related proteins and BLOSUM45 for more divergent sequences. The BLOSUM62 matrix is generally accepted as the midrange matrix for protein amino substitution and is hence the one used in Biothon.

Pairwise Sequence Alignment

PairAligner: The structure responsible for performing pairwise alignment is contained within the PairAligner class. During construction a PairAligner object takes two parameters: score_matrix and gap_open corresponding to the scoring matrix to be used for alignment and the gap penalty respectively. The PairAligner class offers the following alignment algorithms:

- **Needleman-Wunsch (Global)** - The standard dynamic programming approach to computing pairwise, global alignment. A matrix is constructed with rows that represent the letters of one sequence and columns that represent the letters of the other. A path through the matrix, starting at the top-left corner and ending at the bottom-right corner denotes an alignment of both sequences. Horizontal movement adds gaps to the sequence represented by rows and vertical movement adds gaps to the sequence represented by columns. Diagonal movement can either represent a match or a mismatch depending on whether the i^{th} letter of one sequence is the same (or differs from) the j^{th} letter of the other. For every point in the matrix we choose a predecessor that yields the best score. Upon reaching the end we then backtrack via the most optimal path until reaching the start, at which point, the path traced back yields the optimal alignment albeit with letters reversed. We hence reverse the resultant strings and we have the alignment.

- **Smith-Waterman (Local)** - The standard DP approach for pairwise, local alignment. The approach is very similar to the previous, except we cap the minimum value that any entry in the matrix can have at zero. This means that alignment paths can start anywhere in the matrix (and thus anywhere in either sequence) and end once a 0-entry or the last entry in the matrix has been reached. To find the alignments, we find the entry with the highest score and backtrack over optimal predecessors, reversing the strings formed by this path yields the optimal, local alignments.
- **Hirschberg's (Global)** An algorithm that provides a layer of optimisation over Needleman-Wunsch via a divide-and-conquer approach. We use a heuristic, noting that the predecessor of every entry in the matrix is either the entry immediately to the left of it, the entry above it, or the entry towards the top-left corner of it. This means that there are, at most, two rows that need to be considered at any given point in the algorithm. Updating stored entries every iteration, halves the storage requirement to a single row. Either way, we do not need to store all values in the matrix at once and either approach provides spatial optimisation over the naive approach. The matrix is recursively split in half, until we arrive at single entries and built back up. During this process, Needleman-Wunsch is performed in both directions and an optimal point in the path is found, however, every time this process is performed, the search space required for the next iteration is halved. The algorithm still runs in quadratic time, but operates in linear space and is generally computed faster. The approach that I used in Biothon uses the slightly lazier approach of storing two rows at at time, however, this hardly affected optimisation and during testing Hirschberg's, at times, ran twice as a fast as NW.
- **FASTA (Local)** - An optimised algorithm for performing local sequence alignments between a query sequence and a database of sequences. Between the query and each database sequence we:
  - Find the common k-words between the query and database sequence. A k-word is a subsequence of length 'k', and is decided by the user. We then find the indices of these common k-words and record them.
  - Use the indices to identify diagonal runs of common k-words in a matrix. In practice, we do not need to actually construct the matrix if we index each diagonal by the value given by $i - j$. In Biothon, a hash map is used with $i - j$ denoting the key and the value being used as a counter to indicate the length of the diagonal. This hash map only has as many entries as diagonals and is hence a significant reduction on the quadratic complexity of storing a matrix. Once diagonals have been identified, we take note of the top 10.
  - The best are then re-scored using an appropriate scoring matrix.
  - Non-overlapping diagonals can be joined if they result in a better alignment provided that they are penalised for the gaps that are used to link them. Finding the optimal configuration of joined diagonals requires some graph theory. The problem is tantamount to computing the heaviest path in a directed acyclic graph. We convert the information about the diagonals into a graph, with edges denoting the L1 distance between between valid pairs multiplied by the gap penalty. To find the configuration, we add arbitrary source and destination vertices to the graph connecting all of the functional vertices to them and perform a reverse-optimised Bellman Ford algorithm, returning the indices of the diagonals in the optimal chain.
  - We then make the heuristic assumption that the optimal alignment will tend towards this high-scoring diagonal and perform a banded Smith-Waterman algorithm. Banded because we only apply it to the matrix indices that are on the diagonal, allowing for some offset denoted by a user-defined parameter w.

  The results are then compiled into a list of alignments and returned from the function.
- **BLAST (Local)** - The **B**asic **L**ocal **A**lignment **S**earch **T**ool algorithm begins by constructing a neighbourhood of k-words of the query sequence. The neighbourhood should contain *all* possible strings of length k, given the corresponding alphabet, that when aligned with the actual k-length subsequence contained within the query, score at least the user-defined threshold, θ. Any word in this neighbourhood that is found in the database sequence is called a *seed hit*. By considering all possible strings, with respect to a threshold, we are allowing for a certain degree of mismatch between the k-word in the query sequence and the k-word identified in the database sequence. These initial seed hits are extended in both directions to form high-scoring segment pairs and continue to be extended until they fall below the highest attained score, minus an arbitrary threshold. We then return a list of statistically significant HSPs.

PairAlignment: The PairAlignment class represents a template for a pairwise alignment and allows for the alignment to be printed to the screen in a clear manner. Upon construction, Two aligned sequence strings are passed as well as the sequences names' in the form of the keyword arguments *s1_name* and *s2_name*. This PairAlignment object can also be written to a fasta file using the BioIO module.

Multiple Sequence Alignment

Pairwise sequence alignment can identify conserved domains between pairs of sequences but what about a set of sequences? It turns out that directly applying pairwise methods, in this scenario, will not yield an optimal alignment and will fail to identify conserved domains and homology across the family of sequences. Multiple sequence alignment serves to tackle this problem through the process of progressive alignment, computing pairwise alignments for more similar pairs and proceding to more divergent ones. Distinguishing closely-related sequences from divergent ones requires a phylogenetic approach via hierarchical

clustering.

The MSAProfile Class: The MSAProfile class is a representation of a multiple sequence alignment that also has its own *align()* method. This is because unlike in pairwise sequence alignment, where once we've computed the alignment, the job is done, in progressive alignment we must align sequences with sequences, sequences with alignments of multiple sequences and alignments of multiple sequences with each other. The common approach in representing alignments of multiple sequences as a single object to use a profile. A profile is a structure with as many rows as possible symbols in the alphabet of the sequence and as many columns as letters in the alignment itself. Each entry ij denotes the proportion of sequences in the MSA that have symbol i as their j^{th} letter. Therefore each column represent a distribution for a certain letter in the alignment, and as such sums to 1. Generally speaking, each letter in the MSAProfile will have a symbol in the profile that will have the greatest proportion of representation in the individual sequences. Gaps are not counted. If we take each symbol with the greatest representation for every letter in the alignment and form a string out of it, we arrive at the *consensus* of the profile. The methods in the MSAProfile are best explained with respect to the MultiAligner class and the process of multiple sequence alignment as a whole.

The MultiAligner Class: The MSAProfile is capable of performing alignments but as previously stated, to perform progressive alignment, we must align closely related taxa first and progress onto the more distantly-related. This requires, some extra functionality that is separated from the MSAProfile class, exposing only the necessary features as requried to the MSAProfile is handled by the MultiAligner class. The process of (ClustalW) multiple sequence alignment in Biothon is as follows, note that each taxon represents a sequence:

- Take the set of all sequences and compute all pairwise alignment scores and hence construct a distance map.
- Perform UPGMA cluster analysis on the distance map to construct a rooted guide tree with distances equally represented.
- Perform pre-order traversal and assign weights to each taxon, to account for sampling error.
- Perform post-order traversal and perform the align method, to align groups of closely related taxa first progressing onto more distant ones.

During every alignment we make sure to weight the scores computed using the weight of all each taxon represented by each sequence. After every alignment an MSAProfile will make a call to a *reprofile()* method to calculate the new distribution of characters in represented by the MSA.
The result is a profile, with the set sequences aligned in a hierarchical manner.

IV. Cluster Analysis

Phylogenetic Dendrograms

The guide tree created via UPGMA is an example of a dendrogram, a tree in which the length of branches represent evolutionary distance. The guide tree cannot be printed to the terminal, however the *PhyloTree* class has internal methods to *_parse()* the string representation of the tree by progressively adding leaves and branches using +'s, -'s and |'s to visually represent the structure. This is achieved by storing the result of each step of the UPGMA algorithm and making calls to private *_gen_leaf()*, *_add_branch()* and *_shift_subtree()* methods. The result is a visual representation of a phylogenetic tree that can be viewed from within the terminal.

The k-Medoid Algorithm

The k-Medioid algorithm, is a clustering algorithm closely related to k-Means but is more robust to outliers. This is acheived by using data points as centroids (medoids) rather than the mean between groups, as mean is inherently sensitive to outliers. A KMedoid object takes a group of sequences and a distance metric function to use when clustering them. After the running the $solve()$ method, the $.clusters$ attribute of a KMedoid object will contain a list of clusters containing the sequences used as centroids and the sequences related to them. This provides another way to group sequences together.

V. File IO

The BioIO class provides static methods that allow for sequences and alignments to be read from and written to file. Sequences and pairwise alignments are written to *.fasta* files while MSAs are read from/written to *.clustal* files. Lists of sequences can also be read from/written to a single multi-fasta file in the same way that pairwise alignments can.

VI. Conclusion

When I began this project, I only expected to implement a few rudimentary alignment algorithms and statistical functions for biosequences. The original intention was to use this as an applied way to practice dynamic programming while learning about a new field along the way. The more I learned, the more I became engrossed in computational genomics and sought to create a more fleshed out library. Creating Biothon has taught me about the importance of DP and heuristics in data reduction and how reduction can be applied to real-world problems. The most complex algorithm that I had programmed before this point was

the backpropagation in neural networks. Although initially conceptually difficult, the algorithm itself is quite intuitive and can be summarised in four equations. The algorithms I was required to program for Biothon, in particular the ClustalW algorithm, were hence both conceptually and practically more complex than anything I had come across before. They often required knowledge of complex data structures and analysis techniques, often requiring a lot more abstract thinking to implement. Frequently, I would find myself not writing any code, rather having to speak or draw out algorithms in order to understand the flow of information in the algorithms and how best to implement them.

The future of Biothon

The first step in the future of Biothon involves finding a way to implement an affine gap penalty for all alignment algorithms. Although the library is capable of computing accurate alignment scores, especially when using exact methods, for more serious uses of Biothon, an affine gap penalty would offer more accuracy. The next step is to integrate it with online databases allowing for data to be downloaded directly from the console. Entrez is a popular database system that many Pythonic bionformatics libraries have managed to integrate and as such will be my database of choice. I would like the library to have the possibility of invoking a GUI that provides full functionality opening up the potential for Biothon being used by non-programmers and programmers alike. Before I can open up Biothon for external users and contribution, code must be documented and robustly tested to ensure that it cannot be broken. After adding greater support for file types and substitution matrices, I would like to add a graphics engine capable of rendering phylogenetic trees and macromolecular structures from parsed PDB files.

Final Thoughts

There already exists many stellar tools and libraries for bioinformatics in numerous languages, and as such I am not expecting this be much more than a personal project. Regardless, the process of creating Biothon has been an extremely rewarding experience, developing my skills in problem-solving, dynamic programming and Python. All while learning about a field of study that I did not know existed, but have grown fond of all the same.

ACKNOWLEDGMENT

I would like to thank E. Pitkänen from the University of Helsinki and Dr. G Klau from the Free University of Berlin for their notes on bioinformatics. These set of notes, were the only resources that I was able to consistently understand and, in conjunction with other resources, made Biothon possible. I would also like to thank my friend Bethan for discussing my work with me and showing genuine interest and excitement, it was exactly the kind of encouragement I needed to finish the initial stage of this project. Lastly, I would like to thank you for reading this, even if you just skipped to the end, at least you took some time out of your day to show some interest. I appreciate it.