COMP 9 Lab 4: Simian similarity

Due: Thursday, February 10th, 12:00 PM

March 3, 2011

1 Introduction

As we discussed in class on Tuesday, in order to reject or verify the claim that humans (*Homo sapiens*) and chimpanzees (*Pan troglodytes*) are 98% genetically identical, we must actually compare genetic similarity gene-by-gene in the two organisms. This is exactly what you are going to do. Along the way, you will learn how to read and write files, use the BioRuby library (which makes it easy to run BLAST queries), and compute some statistics.

You are conducting a scientific experiment! The specifics of how you compute the statistics on genetic similarity are *up to you*. However, it is of the utmost importance that *you* understand your choices, and that they are justifiable.

2 Housekeeping

You may wish to create a `lab4` directory within your `comp9` directory to keep things organized, as you will have a data file you have to read as well as the code you write. Remember to use the `mkdir` and `mv` commands.

In addition, you will need to use a specific version of Ruby for this assignment, or the BioRuby modules will not work. All you have to do is type `use comp9` in the terminal to set this version of Ruby as the default for that session, but you'll have to do this every time you log in. For convenience, I suggest you use Kate to edit the file `.cshrc` in your home directory:

```
kate ~/.cshrc
```

and put the following into it:

```
use comp9
```

Then save and close that file. You'll need to reopen the terminal or still type `use comp9` this once in the terminal, but you won't have to do it in the future. The file `.cshrc` in your home directory is a *shell script* that is executed every time you open a terminal window; it will there for run the `use comp9` command for you automatically.

3 Design document

As always, please have the course staff check over your *design document* before you begin writing code. You may collaborate with classmates on the design document; please put everyone's name at the top who was involved in discussions. Read the specifications and new information below before beginning the design document.

In your design document, answer the following questions:

1. What sequence of events does your program need to perform?
2. What command-line arguments should it handle, and what error checking should there be?
3. What file I/O operations should it perform?
4. What BLAST queries will it make, and what will it do with the results?
5. How will you represent the sequence identity measures you compute for each gene?
6. How will you turn these values into a final measure of similarity?
7. What data will you need to store to write to the summary file, and how will you store it?

4 Specifications

Write a Ruby program `genome.rb` that does the following:

- Reads a list of *Homo sapiens* genes from a file specified *on the command line*. I will give you one such file.
  - The input file can be found in `/comp/9/data/human_genes.fasta`. You may use that path directly in your program or copy the file into your home directory; it's up to you.
- For each gene in that file, performs a BLAST query against a chimpanzee genome database and finds the 'best' hit

  - computes some similarity measure for that best hit

- Computes the overall similarity
- Prints a summary to the screen indicating the overall similarity of the entire set of genes as a single number (such as 98%)
- Prints each gene's *sequence alignment* to a file specified *on the command line*.

Remember, the specifics of how you compute the statistics on genetic similarity are *up to you*. However, it is of the utmost importance that *you* understand your choices, and that they are justifiable. It would be a great idea to have some comments explaining the *intent* and *justification* of your similarity measure. In other words, make sure what you are computing means what you want it to mean!

In addition to correct functionality, you will be graded on the readability of your code, sensible naming of variables and methods, and abstraction of *repetitive* code into methods.

This is the first time you have done many of these things (File I/O, handling command-line arguments, using BioRuby). Later in this document are some guidelines that will help you.

5 Input file format

The file you will be reading the Human protein sequences from is in a format called *FASTA* (it's a legacy from an early alignment program whose name meant 'fast alignment').

The file looks like this:

```
>sp|P16442|BGAT_HUMAN Histo-blood group ABO system
MAEVLRTLAGKPKCHALRPMILFLIMLVLVLFGYGVLSPRSLMPGSLERGFCMAVREPDH
LQRVSLPRMVYPQPKVLTPCRKDVLVVTPWLAPIVWEGTFNIDILNEQFRLQNTTIGLTV
FAIKKYVAFLKLFLETAEKHFMVGHRVHYYVFTDQPAAVPRVTLGTGRQLSVLEVRAYKR
WQDVSMRRMEMISDFCERRFLSEVDYLVCVDVDMEFRDHVGVEILTPLFGTLHPGFYGSS
REAFTYERRPQSQAYIPKDEGDFYYLGGFFGGSVQEVQRLTRACHQAMMVDQANGIEAVW
HDESHLNKYLLRHKPTKVLSPEYLWDQQLLGWPAVLRKLRFTAVPKNHQAVRNP
>sp|P13647|K2C5_HUMAN Keratin, type II cytoskeletal 5
MSRQSSVSFRSGGSRSFSTASAITPSVSRTSFTSVSRSGGGGGGGFGRVSLAGACGVGGY
GSRSLYNLGGSKRISISTSGGSFRNRFGAGAGGGYGFGGGAGSGFGFGGGAGGGFGLGGG
AGFGGGFGGPGFPVCPPGGIQEVTVNQSLLTPLNLQIDPSIQRVRTEEREQIKTLNNKFA
```

Each line that begins with the character `>` indicates the start of a new entry. That line contains some information about each gene I've given you, but that information is only important for the extra credit part of the assignment. Every subsequent line contains only amino acids. Basically, after a line beginning with `>` come one or more lines representing the sequence for that gene, which continue until you reach another `>` or a blank line; the last sequence ends when you see a blank line by itself.

How might you parse a file like this? Consider (after reading the section on File I/O):

```
sequences = []
sequence = ''
File.open(filename) do |f|
    f.each_line do |line|
        if line[0] == '>' or line.empty?
            # we're at either a > or the final empty line
            # if sequence isn't empty, then we have an existing
            # sequence that should be appended to sequences
            if not sequence.empty?
                sequences << sequence
                # now clear out sequence for the next sequence
                sequence = ''
            end
        else
            sequence += line.chomp
        end
    end
end
```

6 Extra Credit

Should you choose, you may gain up to 10% extra credit by keeping track of the query sequence information. Earlier it was stated that the information about each Human gene provided to you is in a line that starts with a `>` and may be ignored. For the extra credit, preface each entry in the output file with information about the query gene as well as the hit gene. For this, you will want to use the `Bio::FastaFormat` class, described at the end of this document.

If you do the extra credit, save the extra credit version of the assignment as `genome_extra.rb`.

7 Hints

1. Write a piece at a time; test your program early and often. For example, write the code that handles command-line arguments and test that. Then read from an input file and test that by simply printing what you read to the screen via `puts`.
2. Don't repeat yourself! Any code you find yourself typing (or copying and pasting) in more than one place *belongs in a method*. Any code that distracts from the ease of reading your program *belongs in a method*. The main part of your program should be *very* simple, though it will rely on methods you write.
3. Don't panic. This program may seem intimidating, but it can be accomplished with very little code. Think before you write!
4. Remember that *variable* names are arbitrary but should make sense, as are method names for methods that you write. However, method names for existing methods are *not* arbitrary.
5. The specifics of how *you* decide to calculate the similarity of genomes is up to you! However, it should make sense.
6. If you want to compute an average, perhaps it would make sense to write a method for it? Is a simple average what you'd want?

8 How to submit

Please only submit the program you write, `genome.rb`

```
provide comp9 lab4 genome.rb
```

Do *not* submit the file I give you (I already have a copy!) or the output summary you generate. I will test your program in part by generating a new output summary.

If you did the extra credit, submit it as a separate file at the same time:

```
provide comp9 lab4 genome.rb genome_extra.rb
```

9 New things in Ruby

9.1 File I/O

The `File` module allows you to read and write from and to files.

9.1.1 File reading

```
lines = []
filename = 'genes.txt'
File.open(filename) do |f|
    f.each_line do |line|
        lines << line.chomp
    end
end
```

In this example, `filename` represents the name of the file we wish to read.

`File.open` opens a file for reading (specifically, it opens the file whose name is passed as an argument, in this case `filename`). It also takes a *block* (much like the `each` method on an `Array`). Within that block, the *block argument* (`f` in this case) represents the *file itself*.

Since `f` represents the *file itself* (not the filename any more, but actually a way to access the *contents* of the file), we can now read from the file.

The method `each_line` takes *another* block, and the block argument to that (in this case, `line`) represents a single line of input. Just like with `gets`, each line will have a *newline* character at the end, so we can remove that with `chomp`.

So, `each_line` gives us access to a line of the file at a time; in this example we have `chomp`ed that line and appended it to an array, called `lines`.

The file is automatically closed when the `File.open` block `end`s.

9.1.2 File writing

Similarly, you can *write to* a file. When you open a file with `File.open`, you can give it an optional argument that says *how* you want to open the file: for *reading*, for *writing*, or for *appending*. Appending is a special case of writing; if you open a file for writing, any preexisting contents are *erased*; when you open a file for *appending*, anything you write to the file is *appended* after any preexisting contents.

Reading is the default mechanism; the `File.open` example above opens a file for reading. It would be equivalent to say `File.open(filename, 'r') do |f|` in the example above; the second argument `'r'` indicates reading. If you want to open a file for writing, you pass the string `'w'` as the argument; for appending, use `'a'`:

```
File.open(output_file, 'w') do |f|
    f.puts "This is the first line of the file!"
end
```

```
stuff = [2,3,4]
File.open(output_file, 'a') do |f|
    stuff.each do |num|
        f.puts "This is now line #{num} of the file"
    end
end
```

In this example, we open the file `output_file` for *writing*, and simply write a single line to it. In this case, rather than the `Kernel` method `puts`, we're using the `File` method `puts`, which writes to a file rather than to the screen.

We then *re-open* the file `output_file`, this time for appending, and for each thing in `stuff`, we write another string to the file.

With the same end result, we could have written the following; the above example was just to illustrate appending. It is unlikely that you will want to use appending for this assignment.

```
stuff = [2,3,4]
File.open(output_file, 'w') do |f|
    f.puts "This is the first line of the file!"
    stuff.each do |num|
        f.puts "This is now line #{num} of the file"
    end
end
```

9.2 Handling command-line arguments

Do you recall how you can move a file from one location to another by typing `mv oldfilename newfilename`? The *program* you are running is actually `mv`, and you are giving it two *command-line arguments*. Much like the arguments to a method in Ruby, these tell the program `mv` the specifics of what to move to where.

Similarly, you can write a Ruby program that can take *command-line arguments* so that it can operate differently based upon the user's needs, without having to prompt for input with `puts` and `gets`.

The special 'variable' (actually a *constant* as it cannot be changed once the program is running) `ARGV` is an `Array` of command-line arguments given to your program. Since `ARGV` is an array, we can check its `length` and access its `elements`. Consider the following Ruby program:

```
#!/usr/bin/env ruby
input_file = ARGV[0]
output_file = ARGV[1]
```

This program doesn't do much, other than assign the arguments it's given to two variables, `input_file` and `output_file`. What if the input file doesn't actually exist, or the user doesn't specify it? We might want some error checking:

```
#!/usr/bin/env ruby
input_file = ARGV[0]
if input_file.nil?
    puts "You must specify an input file as the first argument!"
    exit
elsif not File.exist?(input_file)
    puts "The file #{input_file} does not exist"
end

output_file = ARGV[1]
if output_file.nil?
    puts "You must specify an output file as the second argument!"
    exit
end
```

Now, this program still doesn't do much useful, but if you saved it as `file_test.rb`, made it executable with `chmod 755 file_test.rb`, and then ran it from the terminal, it would check the arguments you gave it. Assume file1.txt exists and badfile.txt does not.

```
./file_test.rb file1.txt file2.txt
```

No actual output will be produced because the program doesn't yet do anything with its input and output files. However, if we give it a nonexistent file for the first argument:

```
./file_test.rb badfile.txt file2.txt
You must specify an input file as the first argument!
```

It exits with that error message. Finally, what if we forget to give it a second argument?

```
./file_test.rb file1.txt
You must specify an output file as the second argument!
```

Note that the program doesn't check that the *output* file exists, because that file doesn't need to exist yet in order to write to it; it will be *created* when when we do `File.open(output_file, 'w')`

Note that you can also *count* the number of command-line arguments presented with `ARGV.length`

```
#!/usr/bin/env ruby
if ARGV.length != 2
    puts "Usage: #{$0} input_file output_file"
    exit
end
```

The special variable `$0` always contains the *name* of the program; it can be handy for error messages. This little stub of a program behaves as follows:

```
./file_test.rb file1.txt file2.txt file3.txt
Usage: file_test.rb input_file output_file
```

9.3 BioRuby

`BioRuby` is a Ruby toolkit that contains methods for dealing with various biological file formats, tools, programs, web services, and databases. We will use it to run `BLAST` queries.

To use the `BioRuby` toolkit, you first have to load it with `require` at the beginning of your program:

```
#!/usr/bin/env ruby
require 'bio'
```

To run a BLAST query, we first have to jump through a few hoops to create what BioRuby calls a *BLAST factory*. This is just a helper that allows us to run many BLAST queries against the same genetic database with the same options.

```
chimp_factory = Bio::Blast.local('blastp', '/comp/9/databases/chimp.db')
```

This indicates that we want to run a *local* (as opposed to *remote* via a web service) BLAST query. The first argument, `blastp`, indicates that we want to use *protein* BLAST (since we're working with protein sequences). The second argument is the *file path* to the database of chimpanzee genes that I have set up. This creates a BLAST factory, which we might as well assign to the amusingly-named variable `chimp_factory`.

Before we can use our chimp factory to run any BLAST queries, we have to tell it one additional piece of information: where the BLAST program itself lives:

```
chimp_factory.blastall = '/comp/9/bin/blast-2.2.17/bin/blastall'
```

Now, we can query this BLAST factory with a protein sequence; when we do so, it'll give us a *report* that we can do all sorts of things with.

```
query1 = "MAVMPPRTLLLLLSGALALTQTWAGSHSMRYFFTSVSRPGRGEPRFIAVGYVDDTQFVRF"
report = chimp_factory.query(query1)
```

The `report` is actually a Ruby type (or *class*) specific to BioRuby, called a `Bio::Blast::Report`. Just like the `String` or `Array` or `Float` classes, a `Bio::Blast::Report` has lots of methods that can be called on it. Like a `String` or `Array`, it contains things, though what it contains are `Bio::Blast::Report::Hit` objects. Importantly, it contains these hits in *order of significance*. The *evalue* (expectation value) of a hit indicates how likely it is to be an unrelated sequence (occurring at random) so a smaller number is better:

```
report.hits.map{|h| h.evalue}

=> [2.8047e-20, 2.29639e-19, 8.23835e-16, 1.54033e-15, 0.00279514, 0.0549597, 2.28988]
```

On the other hand, if we want to look at identity, we can use the *identity* method:

```
report.hits.map{|h| h.identity}

=> [45, 44, 39, 32, 13, 15, 13]
```

And of course, if we wanted to print a few things about the first hit, we can do just that:

```
first_hit = report.hits.first
puts first_hit.identity
puts first_hit.len
puts first_hit.definition
```

Now, what if we wanted to print the query sequence and its alignment to the hit?

```
puts first_hit.query_seq
MPPRTXXXXXXXXXXXTQTWAGSHSMRYFFTSVSRPGRGEPRFIAVGYVDDTQFVRF

puts first_hit.midline
MPPRT           TQTWAGSHSMRYF+TSVSRPGRGEPRFIAVGYVDDTQFVRF

puts first_hit.target_seq
MPPRTLLLLLSGALALTQTWAGSHSMRYFYTSVSRPGRGEPRFIAVGYVDDTQFVRF
```

Note that the repeated `X`s indicate the *gaps* because the alignment is not perfect. The mid-line also shows this.

9.4 The Bio::FastaFormat class (for extra credit)

The `Bio::FastaFormat` class gives you a way to take a string containing a line starting with a `>` and additional lines of amino acids, and retain information about the gene description contained on that first line. Rather than give away the store, I direct you to the BioRuby documentation for this class: http://bioruby.open-bio.org/rdoc/classes/Bio/FastaFormat.html

The important thing is that you can pass a `Bio::FastaFormat` object to your BLAST factory, and it will preserve the header information in the report:

```
fasta_query = Bio::FastaFormat.new(f_string)
report = chimp_factory.query(fasta_query)
report.hits.first.query_def
=> "sce:YBR160W  CDC28, SRM5; cyclin-dependent protein
kinase catalytic subunit [EC:2.7.1.-] [SP:CC28_YEAST]"
```

You will find this useful should you choose to do the extra credit. Note that this header information (called the `definition`) is also available in the Bio::FastaFormat object itself:

```
fasta_query.definition
=> "sce:YBR160W  CDC28, SRM5; cyclin-dependent protein
kinase catalytic subunit [EC:2.7.1.-] [SP:CC28_YEAST]"
```