API Project 4 Final Report

Background

In bioinformatics, it is often important to be able to get data from various organisms' **genomes**. Every individual organism has a genome, which could be written down as a string of bases. Of course, there is no single "human genome", but in most cases all humans have very similar genomes. Accordingly, biologists produce **reference alignments**, which are a string of bases representing a sample genome[1]. Genomes come in **chromosomes** (for example, the human genome has 23 different chromosomes[2]), so reference alignments are split by chromosome.

Most researchers aren't looking at the entire human (or mouse, or nematode) genome, but substrings. These are called **sequences**, and they're accessed via **coordinates**. The coordinates include the chromosome, start index, and end index, and tells you exactly where the sequence is located within the reference alignment. For each organism, there are several reference alignments, as new, better alignments are always being produced. For example, the human genome has hg38, hg19, etc. For this project, we will refer to a reference alignment as a genome, in keeping with the domain terminology (e.g. "the genome hg38").

A huge quantity of genome data is accessible publicly to researchers. One of the main vendors is the UCSC Genome Browser, which has a comprehensive graphical user interface as well as a REST API that performs programmatic actions and an assorted collection of tools. While the UCSC Genome Browser provides invaluable data, it isn't particularly easy to use programmatically, since it can only be accessed manually or via web request.

One of the most useful functions the genome browser provides is getting sequence data. Since sequences can be long, researchers often prefer to store and share coordinates rather than the actual sequence. However, when they then need the sequences, getting that data can be arduous. As a closely related task, researchers tend to share coordinates for a given alignment. When alignments are updated, these files are usually not. It is easy to end up with two sets of coordinates, where both are for human but one might be hg38 and the other hg19. To work with these coordinates, the researcher needs to **liftover** the coordinates from hg19 to hg38. UCSC Genome Browser also provides a command line tool to perform liftover.

[1] For the curious, variation within organisms is represented through single nucleotide polymorphisms (SNPs), which are locations where we know that different people have different bases

[2] You have 46 pairs, one from each parent

Given the amount of data contained in the UCSC Genome Browser, we could certainly not have provided all of its functionality, nor should we. However, since getting and lifting over sequence data is so essential to bioinformatics, we hoped that providing this functionality would be a useful project.

Our API

We created a Python wrapper for the UCSC Genome Browser functions we planned to support, choosing Python because it is a common scripting language for biologists. Our goal was to create a smooth experience.

The biggest problem with the current situation is the need to connect the output from multiple tools, each of which has its own overhead. In particular, using the REST API programmatically requires integrating code that sends requests and using liftover requires downloading chain files (files which specify how liftover should be performed) for the liftover. Both methods also may require the user to create intermediate files they will never use otherwise. Our API successfully eliminated this need in the cases we were targeting. It provided a clean, intuitive syntax to get and liftover sequences and eliminated the need to make your own intermediate files.

As an example scenario: suppose you have many files of candidate intervals, given in hg19 (an alignment of the human genome), and another file of regulatory sequences, given in hg38 (a more recent alignment of the human genome). You would like to only consider the candidate intervals that include at least one regulatory sequence.

Without our API, you would need to piece together several tools. First, you would write a script to combine the candidate interval files into one. Then, you would run liftover on this file (either command line or website would work). Next, you would need a script to pull sequence data (or download it otherwise). Finally, you would be able to perform your analysis and then output your data.

Our API made this process substantially simpler. You would create a SequenceSet from all the interval files, make a function call to liftover, and then you could use the strings in your analysis. Other than the analysis, the whole process is about 10 lines of code.

Overall, our API allowed for 4 main functionalities:

- Downloading sequences (whole genome, whole chromosome, or a list of sequences from coordinates)
- Lifting over coordinates

- Listing the genomes (either all the UCSC genomes or all the alignments given the colloquial name of an organism)
- Listing the chromosomes of a genome

We had three classes: Sequence, SequenceSet, and Genome. Our documentation describes each method in detail, but below is an overview of the classes and the decisions we made:

The Sequence class represented a single sequence, specified by its genome and coordinates. We considered calling this class "Coordinates" instead, and having a method "Coordinates.get_sequence()", but decided instead that, since coordinates specify a sequence, it would make more sense to create a sequence from the coordinates (chromosome, start, end) and genome. We get a genome sequence via "Sequence.string()", giving the illusion that a sequence object comes with the string. (In reality, we make a REST API call the first time string() is called, and cache it for later.)

The SequenceSet class represented a collection of sequences, which is the level most researchers work. We chose to take in a list of bed files (and a genome) to construct a SequenceSet, as bed files are the standard format in bioinformatics for sharing a list of coordinates, and reading a list made it easy to combine multiple related files into one set. A SequenceSet can be lifted over to another SequenceSet, which downloads and caches the necessary chain file, and iterates over it. Additionally, it can be outputted as a bed file of coordinates or a fasta file of sequences. The bed file output is necessary if the SequenceSet is created via liftover or filtering (or otherwise manipulating) another SequenceSet. The fasta file output allows sequences to be downloaded using a SequenceSet; fasta files are the standard file format to share sequences.

The Genome class represents a genome alignment and also provides static utility methods. We considered passing around genome alignments as strings, but decided to avoid the possibility of invalid genomes. Instead, a genome can be created from a string, and an exception is thrown if the genome is not in the UCSC database. We create all the genome objects we will need the first time a genome is created, and afterwards look them up, so that each genome is a unique object. In addition, the genome class makes it possible to download entire genomes/chromosomes, list the available genomes - all of them or for a specific organism (ex: "cow", "kangaroo rat"), and list the available chromosomes for a genome.

There is also a static wrapper for UCSC's liftover tool in the Genome class. We created this observing that, if a user calls our SequenceSet liftover purely to liftover, it reads from a bed file, then writes out those same coordinates, then runs liftover, which produces another bed file, then reads from the lifted over the bed file, which the user would then write to a bed file. This is needed because SequenceSets are not always produced from bed files, but for a user that simply wants to liftover from one file to another without downloading a chain file, it is unnecessary.

Since we were writing a wrapper for a REST API, we needed to be careful to obscure its idiosyncrasies. We automatically retry requests twice, in case the connection falters. The user is also able to set the number of retry requests if they wish to. Since the UCSC API had an infinite timeout, we also added one manually, so that users would not be left waiting indefinitely, and allowed them to set it. We also considered breaking up downloads into smaller pieces, but downloading a chromosome seemed reasonably efficient.

To demonstrate the versatility of the design we chose, we looked at two functions suggested by bioinformatics researchers. One was a bulk downloader for mammalian genomes, and another was a function to pull the sequence of one human gene and its orthologs (similar genes) in several species. We implemented both easily using our API, though in the former case the user would have to list the genomes they wanted.

Further Work Needed

The main improvement our API could use is flexibility. Since this API works with large quantities of data, interfacing with a REST API, and is intended for use by researchers who may have a large variety of environments, it is important that it can be set up and used easily for different operating systems and versions. It may also need more support for operations on very large sequence sets.

We can also work on additional tests to evaluate our API's behavior under client's bad network connection. We implemented a functionality to allow retries for the requests we make to the genome browser REST API. However, we were unable to generate tests that mocked a bad network connection. Adding tests for this behavior will allow us to see how the API behaves when it undergoes request retries.

In addition to bed files and fasta files, researchers may also use other file formats to share data. Though there are tools to convert between most file types to bed files and fasta files, since these are the most common types, we could be more useful by supporting some other common formats. Similarly, researchers may have other sources they prefer over UCSC Genome Browser, or they may have their own files. We should make it possible to substitute those.

Finally, the API currently supports a good set of functionality, but there may be more common usages users would commonly want. If this API becomes public and used, we would want to hear what might be useful to researchers.

Unanticipated Problems

A problem we encountered was dealing with possible retries and timeouts. In the beginning, there was no way for the user to specify how long the API request should take before timing out. The call would hang until the connection was closed, since the default for the web API is no timeout. Thus, we implemented a retry and timeout wrapper for use in the Genome and Sequence classes -- ones that may require long download times and the user should be able to set their own intended retry and timeout parameters. Different genomes have different download times, so we give the client the ability to set the timeout depending on what they want to download.

We also encountered a problem with downloading genome since we realized they can be really large. Since they can contain millions of DNA sequences, simply printing it to the terminal would cause it to freeze. We decided to output each chromosome of a genome to a separate generated file, that the user can decide what to prefix with (with the pattern fileprefix_chromosome_genome). This would allow the user to quickly identify which sequence(s) is the one they need, since it is also split up by individual chromosomes.

Another unanticipated problem we encountered was the uncertainty of which functionalities were necessary for our API to have. For example, we started off envisioning having a method to create a Genome object from a specified organism, but realized that this would not be necessarily what the user wants. They should know exactly which genome they are creating, as that is the most common use case. We also needed to decide on if we should give the users the ability to create Genomes, even though there should only be a set number. We ultimately decided on reducing the number of API calls and lazily populating an internal list of valid Genome objects that we create when the class is first used. We do not give the user the ability to create new Genome objects -- if the user tries to create a new one, it would simply reference the already created Genome object.

We also encountered a problem with the setup script. We realized that different OS's need different executables, and our current one works only for Mac OS. A proposed solution to this problem would be to create different setup scripts for each OS, but we noticed that there are only one or two commands to run depending on the OS. Therefore, we concluded that the best solution would be to elaborate on the set up steps in our API's README file.

Finally, a big unanticipated problem we encountered was having three of the members on our team not being the most familiar with Python. This led to a lot of trouble with referencing classes from different files. We were stuck for a rather long time on a “ImportError: No Module named …” that took much longer to debug than if we were more familiar with how package imports in Python work.

Work Distribution

SequenceSet class: Masha, Miranda
Liftover functionality: Emma, Minji
Genome class: Iris, Miranda
Sequence class: Iris, Miranda
Setup script: Emma
Client code: Emma
Unit Tests: Everyone
Unit Test Mocking: Masha, Minji
Retry and Timeout Wrapper: Iris, Minji
Multiple OS support: Emma, Miranda
Documentation: Everyone

External Resources Used

UCSC Genome API interface: api.genome.ucsc.edu
Thank you to Morgan Wirthlin and Irene Kaplow for your advice and suggestions!

Appendix

Link to API Artifacts

https://genome.ucsc.edu/goldenPath/help/api.html

Test Results

Sequence Tests

test_print_sequence	Create a sequence object and print sequence information.
test_sequence_string	Create a sequence object and check if the string is saved as a result of lazy evaluation.

Genome Tests

test_list_genome	Returns a list of all genomes available Makes input is case-insensitive
test_init	Ensures genomes with same string are the same object
test_download sequence	Downloads a chromosome for a genome with no errors Downloads entire genome with no errors
test_list_chromosome	Lists all chromosomes for a genome Ensure there are no duplicates

SequenceSet Tests

test_hg19_file_creation	Create a SequenceSet object and generate a bed file.
test_hg19_to_hg38	Download the chain file and perform liftover to a hg38 genome.
test_nonexistent_file	Correctly raise an error when the bed file given to SequenceSet constructor does not exist.
test_nonexistent_target	Correctly raise an error when the sequence name given to SequenceSet constructor is inconsistent.
test_bad_src_file	Correctly raise an error when the bed file given to SequenceSet constructor is malformed.
test_wrong_col_count	Correctly raise an error when the bed file given to SequenceSet

	constructor is malformed (wrong number of columns).
test_bad_chain_file	Correctly raise an error when the chain file given to liftover is malformed.