Nextstrain lab

Helpful links:

Nextstrain documentation page: https://docs.nextstrain.org/en/latest/index.html Nextstrain augur sub-commands documentation: https://docs.nextstrain.org/projects/augur/en/stable/usage/usage.html Github repository for Nextstrain: https://github.com/nextstrain Treetime documetation: https://treetime.readthedocs.io/en/latest/index.html

Introduction and background

In this lab, you will be using Nextstrain to build a set of phylogenetic trees describing non-human H3 influenza virus evolution. In human populations, 2 influenza A subtypes circulate seasonally, causing yearly epidemics: H3N2 and H1N1. Both viruses caused pandemics in the past, and then continued circulating in human populations ever since. The H3N2 virus that circulates today in humans first emerged in the pandemic of 1968, during which HA and PB1 genes from an avian influenza virus reassorted with the H2N2 seasonal virus that was circulating in humans. The introduction of a novel H3 HA gene, against which humans did not have pre-existing immunity, caused widespread transmission and a pandemic. Following initial transmission in humans, H3N2 continued to circulate seasonally, evolving yearly in a process of continual immune selection. The H1N1 virus that circulates in humans today is the result of the 2009 swine flu pandemic.

All influenza viruses naturally circulate in wild, aquatic birds, which harbor every HA and NA subtype (except for H17 and 18, which only infect bats). Avian origin H3 viruses have repeatedly spilled over into other host species, resulting in long-term transmission in humans, pigs, horses, and dogs. In this tutorial, we will build phylogenetic trees to investigate the following questions:

1. How many times have H3 viruses been introduced into new species?
2. When did these events occur?
3. Does reassortment contribute to patterns of circulation in different hosts?
4. How have these viruses spread geographically?

We will begin our tutorial by running individual Nextstrain commands one by one and observing what they are doing. Next, we will automate this process using a tool called Snakemake. Finally, we will introduce the concept of wildcards, which allow automatic pipelines to be run for a wide range of input files.

Activity 1: Manually building a tree.

1. Navigate to the course github page, and clone the repository. Navigate to `exercises/Nextstrain-lab/activity-1` and open up the folder.

2. Let's start by taking a look at our input data. This data was downloaded from Genbank's influenza database, and cleaned up to include cleaned and annotated host sequences, dates, and geographic location data. Open up the file `h3nx_ha.fa` in BBEdit. Our input file is a fasta file, where metadata has been added to the headers, separated by a "|" character. Nextstrain can accomodate multiple input filetypes, but requires data be in one of two basic formats: a. a fasta file where metadata fields are specified in the fasta header, separated by an accepted separator character. The first field must

always be a strain name, and each strain name must be unique. b. 2 separate files: a fasta file, containing for each sequence, a strain name and sequence; a metadata file, consisting of a tab-separated text file where the first column specifies the strain name and each additional column specifies a metadata field.

In this tutorial we will start with format a and generate format b, so you will see both. The main, important caveat is that the first column must always be the strain name, each strain name must be unique, and each sequence must have the same number of metadata partitions. Fields can be blank/empty, but the partition must exist.

In our fasta file, we have several pieces of metadata, in the following format:

```
strain_name|Genbank accession number|influenza subtype|sample collection date|host category (avian, human, dog, swine, horse)|country of sample collection|region of sample collection|host species|cleaned host species|host order
```

The first field is the strain name. Influenza strain names are organized by the following: influenza type (A,B,C)/ host species, if not human/ a random identifier/ year of sample collection.

Host order represents some cleaning of the various avian species and categorization into the Anseriformes (ducks, geese, swans), Galliformes (turkeys, chickens, quails), and everything else (Neoaves).

3. Now, open up your Nextstrain build environment and test that it works. Nextstrain can be installed in a few different ways, detailed in the documentation here (https://docs.nextstrain.org/en/latest/index.html). For this workshop, it should be installed via Conda. Go ahead and check that this works by running: `nextstrain check-setup`. You should see a series of check marks showing that the conda runtime is working, but not docker. If you do not see this, let me know, and we will set up your conda runtime environment.

4. If step 3 worked, open your terminal window, and type `conda activate nextstrain`. This will use Conda to activate a pre-built environment, called `nextstrain`, that already has all the software installed. For the first part of this tutorial, we want to run commands one by one. To do this, type `nextstrain shell .`, which will open up an interactive display that looks like this:

Within thisi window, type `nextstrain --help`, which should print out a series of help commands/documentation. Nextstrain has quite good documentation for all of its commands, and they are all available via the `--help` flag. I use these all the time when I forgot the required arguments for various commands. These are also described and documented on the website.

5. We are now ready to start running Nextstrain! Let's begin by parsing our input fasta file into sequences and metadata. To do so, we can use the `augur parse command`. To see the list of options for `augur parse`, plus a description of what it's doing, type out `augur parse --help`. Augur parse will split our fasta file into 2: a sequences file, and a metadata file. We need to tell Nextstrain what the fields in our fasta file are, which file is our input, and which filename to write our output to. Type the following:

```
augur parse --sequences h3nx_ha.fa --output-sequences h3nx_ha_sequences.fasta --output-metadata h3nx_ha_metadata.tsv --fields strain accession subtype date host country region species broad order
```

Open up the output files and check them out.

6. Next, let's filter our data. Sequence datasets usually require a fair bit of cleaning up before using them to build a tree. In addition to removing unwanted sequences from our dataset, we will also want to do a bit of work to subsample our data. Phylogenetic models are sometimes heavily impacted by sampling bias, and sampling bias is persistent in most sequence databases. One good way to combat this is even out the groups in your analysis with subsampling. By forcing the number of sequences per group to be equal, we can help to mediate some of the impacts of sampling bias. In this step, we will filter out sequences from ferrets (which likely represent experimental infections), sequences that are too short, sequences that contain erroneous, non-nucleotide characters, and subsample down to a small, manageable number of genomes per group. We will also exclude the strains specified in `exclude_strains.txt`, which represent highly divergent swine lineages. The computational time it takes to run this pipeline depends heavily on how many sequences are included, with the most computationally intensive steps being alignment, tree generation with IQTree, and tree refinement with Treetime. Feel free to play around with adding more sequences later, but for expediency, I've tried to keep each step to under 5 minutes.

```
augur filter --sequences h3nx_ha_sequences.fasta --metadata h3nx_ha_metadata.tsv --output filtered_h3nx_ha.fasta --group-by region host subtype year --sequences-per-group 2 --exclude-where host=ferret --min-length 1600 --non-nucleotide --exclude exclude_strains.txt
```

You should get a printout that tells you how many sequences were removed from the data for each subsampling argument you included, and how many total sequences remain in your dataset.

7. The next step is to align our sequences. We will do this with `augur align`, which calls mafft. We will use a reference sequence, in genbank format, to help with the alignment. This step will take ~30 seconds to a minute.

```
augur align --sequences filtered_h3nx_ha.fasta --reference-sequence reference_sequence_ha_A_mallard_Alberta_114_1997.gb --output aligned_h3nx_ha.fasta --remove-reference --nthreads auto
```

Once that completes, take a look at in Seaview to make sure it all looks good (i.e., no weird gaps or obviously misaligned regions).

8. Once we have our alignment, we are ready to build a tree. We will initially build our tree using IQtree, but you could instead choose to use raxml or fasttree. Explore `augur tree --help` to see all the options here. This step will take ~2 minutes.

```
augur tree --alignment aligned_h3nx_ha.fasta --output tree-raw_h3nx_ha.nwk --method iqtree --nthreads auto
```

9. Refine our tree to build a time-resolved phylogeny using TreeTime. Here, we will build a time-resolved tree using a coalescent model with a constant population size. We will infer dates using the "marginal" method, and we will employ a clock filter to remove samples which fall > 4 standard deviations from the inferred clock rate. These types of sequences are often the result of sequencing error or mislabeled dates. Finally, Nextstrain stores data to layer on top of trees (like inferred mutations and branch lengths) in json format, and calls these types of data "node_data". So here, the

output node data is branch lengths, which we save to a json file. JSON files can be opened in any text editor, and are essentially massive, printed out dictionary.

```
augur refine --tree tree-raw_h3nx_ha.nwk --alignment aligned_h3nx_ha.fasta --
metadata h3nx_ha_metadata.tsv --output-tree tree_h3nx_ha.nwk --output-node-data
branch-lengths.json --timetree --coalescent "const" --date-confidence --date-
inference marginal --clock-filter-iqd 4
```

Treetime will print out a lot of information during this, and will go through multiple iterations of refining the tree. This step took ~3-4 minutes on my laptop.

10. As part of this exercise, we would like to infer mutations onto each branch and internal node. To do so, we can use `augur ancestral`:

```
augur ancestral --tree tree_h3nx_ha.nwk --alignment aligned_h3nx_ha.fasta --
output-node-data nt-muts.json --inference joint --keep-ambiguous
```

Also, we would like to translate these mutations into amino acids: `augur translate --tree tree_h3nx_ha.nwk --ancestral-sequences nt-muts.json --reference-sequence reference_sequence_ha_A_mallard_Alberta_114_1997.gb --output aa-muts.json`

11. As our last analytic step, let's perform a discrete trait reconstruction to infer ancestral states for geographic region, host group, and subtype. These models are very common in phylogeography, and treat metadata columns as "discrete traits". The transitions between these states are modeled in the exact same way that a substitution process is modeled, ie., each internal node will be assigned a probability of being in a particular state, and will be annotated with the most likely ancestral state.

```
augur traits --tree tree_h3nx_ha.nwk --metadata h3nx_ha_metadata.tsv --output
traits.json --columns host order region subtype --confidence
```

12. Nextstrain is made up of several, distinct components. Auspice is the software that performs the visualizations. In order to visualize our final phylogenetic tree in auspice, we need to export the tree + all of the node data encoded in JSON files into a single tree JSON object. We can accomplish that with `augur export`. The specifics of how the JSON is formatted, and which fields are displayed as drop-down options and filtering options are specified in `auspice_config.json`. Finally, Nextstrain will automatically assign colors to any metadata field visualized in the tree. However, the defaults are not always easy to interpret, so we can also pass our own colors. We will pass our own colors for subtypes and hosts in the `colors.tsv` file.

```
augur export v2 --tree tree_h3nx_ha.nwk --metadata h3nx_ha_metadata.tsv --node-
data branch-lengths.json nt-muts.json aa-muts.json traits.json --auspice-config
auspice_config.json --include-root-sequence --output auspice_h3nx_ha.json --
colors colors.tsv
```

13. To visualize the resulting tree, you have a couple options. One is to navigate to auspice.us and drag and drop your auspice json file onto it. This option always works. The other is to exit out of the shell runtime by typing `exit`, and then running `nextstrain view .` in your terminal window. Then, navigate to `localhost:4000` in your browser, and it should load! Take some time to explore the tree.

Visualizing results

Now that we have a tree, let's take a look! The Nextstrain interface has a lot of fun and useful features. We'll walk through a few of them, and then you should take some time to explore.

1. First, let's check out the clock rate. One the lefthand panel of the screen, you'll see lots of options for how to visualize the tree. Under the "Tree Options" -> "Layout", you should see a "Clock" option. Click on that option, and take a look. What is the estimated clock rate for these viruses? Do these viruses exhibit "clock-like" evolution?

2. One interesting question is whether these viruses might evolve at different rates in different host species. One way we can take a look at that question is by comparing the clock rates by host. We can look at this visually, but it is a little challenging to see. So now let's try filtering the data by host. In the lefthand panel, under "Filter Data", click in the box. You should see some options auto-fill. Select `Host -> Equine` and `Host -> Swine`. Is there any noticeable difference in clock rates among these hosts? How about other combinations of hosts? Now try filtering on `Host -> Canine`. What is different about canine evolution?

3. Now, let's go back to the tree view. Under "Tree Options" -> "Layout", click on "Rectangular". We now get to choose whether we want to view the tree with branch lengths scaled by Time or Divergence. Toggle between the divergence tree and the timetree. Do they look different? If so, how? If not, why would that be?

4. Look at the tree, colored by host. You'll notice that we have inferred "host" traits onto internal nodes on the phylogeny. We can infer host switches on the tree by looking at the internal nodes at the which a host switch occurred. How many host switch events have there been over the history of this tree? Which host groups are major donors and which hosts are recipients? Which host switches have been the most successful, i.e., circulated for the longest or with the most diversity? Do different avian orders play different roles in H3 evolution and transmission? Which host order (anseriformes, galliformes, or neoaves) act more frequently as sources?

5. As part of our reconstruction, we also inferred nucleotide and amino acid mutations on the internal nodes of the phylogeny. Zoom in on different branches of the tree by clicking on them. Take a look at some of the branches leading to host switches. We see that many of these host switch branches have amino acid changing mutations. If we want to filter the tree to color by mutations, under "Color By", select "Genotype" -> "HA", then the amino acid. Take for example site 179. Look at this site in the tree view and the scatterplot view. We can see that it is highly polymorphic in different host species. Without further data on phenotype, it is impossible to know whether this illicits an important change evolutionarily, but it is an interesting hypothesis that could be tested.

6. Now let's look at geographic regions. Change the colorby view to "region" and take a look at the tree. How geographically structured are these populations? Are particular host groups restricted to particular geographic regions? Which host groups circulate the most widely across geographic space? As another fun way to visualize this, play around with the scatterplot view of the tree. You can select which variables to plot on the x and y axes.

7. Finally, let's look at subtypes. Influenza viruses shuffle their genomes through a process called reassortment, which occurs when co-infecting virions co-infect the same cell. Reassortment is how HA and NA segments gets shuffled into new combinations (e.g., h3n1, h3n5, h3n8, etc...). Using the scatter plot view and the tree view, do particular hosts harbor more subtype diversity than others? If so, what does that tell us about infection frequency?

Some useful extra notes about Auspice:

1. You can download the tree as well as screenshots of the tree in whatever view you format it into. This can be really useful for generating figures for a paper or for a presentation. If you scroll down to the very bottom of the screen, click on "Download data", and you will see some options for downloading the trees and screenshots.

2. One cool extra feature that is a bit secret is that you can add extra metadata on top of these trees and visualize them in Auspice by dragging and dropping a csv file onto the web page. This will actually populate the "Color By" dropdown menu with the new features. All that is required is that this annotation file is called "annotations.csv" and that the first column is the strain name, exactly matched to the strain names in the build. This can be useful if for example you have a collaborator who has extra data/metadata they would like to visualize, but can't be shared publicly.

Analysis 2: Automating your build with Snakemake

Now that we've run through all the commands in augur one by one, and laid out all the steps in the pipeline, we can now automate this. Nextstrain builds rely on a pipeline tool called Snakemake to generate and run builds, without having to manually specify each step. Snakemake has extensive documentation, and is a little bit confusing. However, once you get used to it, it can be quite useful. Here, I'll provide a brief overview of how Snakemake works with regard to Nextstrain and how the Snakefile is organized. Then, we'll take a look at the Snakefile and launch a build by calling that file.

Snakemake is a tool that was built for pipelines. Snakemake works by reading in a Snakefile which specifies that pipeline in a set of rules. Rules can inherit from each other and chain together to make a complete pipeline. The user defines the rules, and input and output files, along with instructions for how each rule is connected to each other one. Snakemake uses this information to construct a directed, acyclic graph that describes the workflow, and determines which steps need to be run.

1. To get started with this analysis, navigate to the folder `activity-2-snakemake-ha-only`. In your terminal, change into that directory with `cd ../analysis-2-snakemake-ha-only`. Test that the build will launch by running `snakemake -n`. You should get the following printout:

If that succeeds, run: `snakemake -p --cores 2`. This will launch the build. While the build is running, we will take a look at our snakefile. This build will take ~10-15 minutes to run. Because it will produce the same results as Activity 1, it's not necessary to run to completion. The main goal is to see that the exact same things are run, but in an automated fashion.

Alternatively, you can launch the build with `nextstrain build .`. The instructions you use primarily depend on how Nextstrain was installed. They are both equivalent, in that they simply run the Snakefile that is in the directory you launch the analysis from.

2. Open up the `Snakefile` in a text editor of your choice (Atom if installed, BBedit is fine if not). Take a look at how the file is organized. This Snakefile includes all the rules to run the build we just ran manually, and includes a ton of extra notes and comments about the structure of the file, user-defined inputs, and places where I tend to make changes. Also, instead of outputting everything into a single directory, we now put things into folders. We have `data`, `results`,`auspice`, and `config`.

3. Now, scroll though the Snakefile and take a look at the order of the rules. You'll notice that the first rule specified is the `rule all`. Every Snakefile has this. `Rule all` specifies what the overall goal of the snakemake pipeline is. This line tells Snakemake that at the end of this pipeline, we want to generate a single, final output file, `auspice/h3nx_ha.json`. Next, we have our `files` rule. These specify files that must be present at the outset of the pipeline for Snakemake to work. These are our input files.

4. After that, we get into our pipeline rules. Notice that the rules that we are running are the exact same analyses that we ran in analysis 1. We start with `rule parse`, which calls `augur parse`, using the same arguments we ran earlier. The only exception is `rule filter`, which includes slightly different sampling parameters (1 sequences per group rather than 10, simply to make this run faster), and has include and exclude files, and a minimum date filter. Each rule has a few different components:

a. `message`: this is a string of text that you can change to anything you'd like. This is imply what will print while the rule is running.

b. `input`: These define the inputs for your rule. These inputs are meant to reference other rules. For example, our first rule, `rule parse` calls `files.input_sequences` as its `sequences` input variable. In the next rule, `rule filter`, we use the output of `rule parse` as our input variable. We call that by specifying `rules.parse.output.sequences`.

c. `output`: For each rule, the output simply specifies where the output file should be written and what it should be called. Often, outputs in 1 rule will serve as inputs for later rules.

d. `params`: The `params` portion of the rule specifies arguments for that rule that aren't inputs or outputs. For example, `params` includes all of our filtering arguments in `rule filter`

e. `shell`: The `shell` argument is the actual syntax/text that is fed to the terminal to execute the rule. You'll notice that these shell commands are exactly equivalent to what we ran in activity 1.

If this is all still a little confusing, that is normal! Snakemake is really flexible, but structured in a way that is very verbose and confusing. My hope is that having these examples can provide a template that you can look back at later, and use to structure your own pipelines.

5. At this point, you should be noticing that your pipeline is running in the background in the same order that we ran things activity 1. It may even be done by now. If all went well, you will have generated essentially the same analysis, only with way fewer, with your output in the `auspice` folder.

Activity 3: Using wildcards

In this last activity, we will introduce wildcards as a way of making your analysis more flexible and running your trees on multiple viruses, genes, or both. Wildcards add yet more complexity to Snakemake, but are really wonderful for increasing complexity. In this example, we will use the `segments` wildcard to repeat this whole pipeline on each individual influenza gene segment, all at once.

1. Let's get this last analysis running, and then we can look at our trees while it runs in the background. Navigate into the folder, `analysis-3-snakemake-wildcards` and launch the build the same way again. This time, let's do a preview first to have it print out what the analysis will be doing before we launch it. In your terminal, type `snakemake -n`, which will perform a dry-run. You should see the printout below:

Notice that we are now going to be running 3x as many jobs! That is because we have used wildcards to specify that we want to run this pipeline on each influenza gene segment, and there are 8 gene segments. Go ahead and launch the build with the same commands as before: `snakemake -n --cores 2`

2. While the build is running, open up the Snakefile in a text editor. You'll notice some new features here right away: wildcards! Wildcards are specified as a list of strings, and tell Snakemake that we want to do the entire pipeline for every wildcard. The wildcard names are then appended to the various input and output files as specified in the Snakefile. For example, our input rule has been transformed from this: `auspice_json = expand("auspice/h3nx_ha.json")`

to this: `auspice_json = expand("auspice/h3nx_{segment}.json", segment=SEGMENTS)`

The lowercase `segment` in curly braces `{}` is a way of calling the wildcard variable, and essentially expands the list of expected files to be one for each wildcard in the wildcards list. For example, the above syntax tells Snakemake that we now expect 8 output files: `auspice/h3nx_pb2.json, auspice/h3nx_pb1.json, auspice/h3nx_ha.json, auspice/h3nx_pa.json` etc..., one for each wildcard.

As you scroll down through the Snakefile, notice that wildcards propagate to each input and output file as well, guaranteeing that write separate outputs for each wildcard.

Wildcards can be finicky, but you can do a lot with them. You can even have multiple wildcards at the same time: in the public avian flu build, we have wildcards for subtypes and segments, so you can see an example there.

3. A final thing I included in this example is an example for how to use custom functions to make this arguments flexible for each wildcard. In this example, I've defined a function called `min_length`, which specifies a minimum sequence length for each gene segment. In `rule filter`, instead of specifying one length filter cutoff for all the genes, we instead call the `min_length` function to set the minimum lengths separately for each gene.

A final note: you can include shell calls to other, custom scripts and modules as well. They are not restricted only to nextstrain commands. The avian-flu builds also have examples of a few custom functions, like annotating cleavage sites and adding clade information, that you can look at if you are interested at https://github.com/nextstrain/builds/avian-flu.

Visualizing results

Once everything is run, again visualize results by exiting the nextstrain shell runtime with `exit` and then run `nextstrain view auspice`. This will open up an auspice window where you can visualize the jsons in the auspice folder. If this fails (it sometimes doesn't work if Docker isn't installed), simply drag and drop the jsons onto auspice.us.

Some questions with the internal gene segments: Do the internal gene segments show similar rates of cross-species transmission, or are they distinct? How similar or different are the tree topologies and host reconstructions? Hint: you can look at 2 trees simultaneously by selecting `2nd tree` in the auspice drop-down menu.