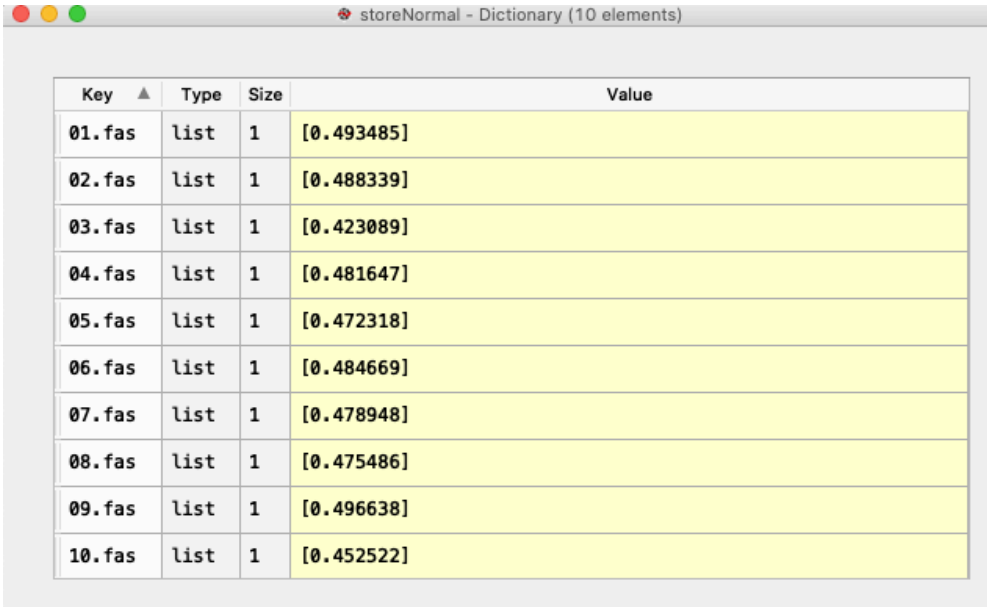


Homework 8 Writeup

For reference, please find this list of the p-value for all FASTA files with the example tree and parameters. The average is 0.4747141.



Key ▲	Type	Size	Value
01.fas	list	1	[0.493485]
02.fas	list	1	[0.488339]
03.fas	list	1	[0.423089]
04.fas	list	1	[0.481647]
05.fas	list	1	[0.472318]
06.fas	list	1	[0.484669]
07.fas	list	1	[0.478948]
08.fas	list	1	[0.475486]
09.fas	list	1	[0.496638]
10.fas	list	1	[0.452522]

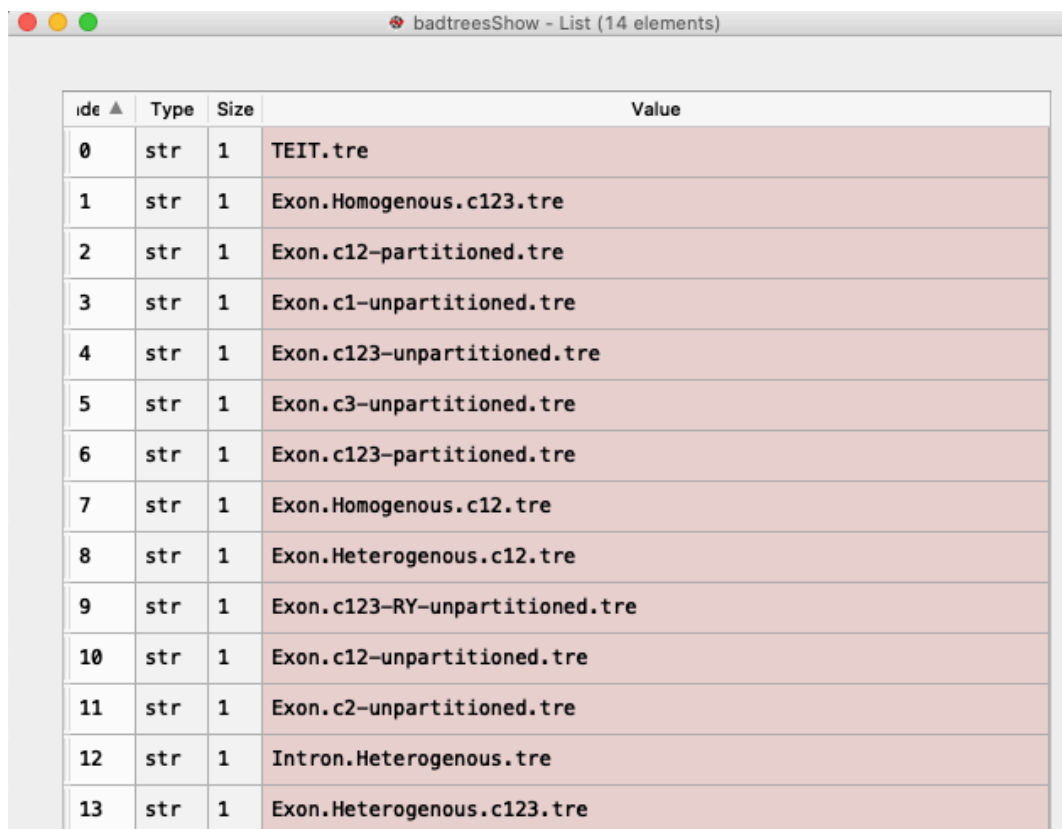
I.

Please see the visuals below, the first is the list of 'good trees', meaning trees that did not have a p-value of less than 0.05 for any of the 10 FASTA files. The second a list of 'bad trees', which had a p-value of less than 0.05 for one or more of the sequences. The third is the plot showing the average p-value over all 10 sequences for all the trees. It was determined that this was the best way to representant visually which trees are likely to have generated the data, which can be rejected and which are ambiguous.



idx ▲	Type	Size	Value
0	str	1	TENT.25%.tre
1	str	1	UCE.unpartitioned.tre
2	str	1	Intron-unpartitioned.tre
3	str	1	TENT.75%.tre
4	str	1	WGT-Alternative.tre
5	str	1	Intron.Homogenous.tre
6	str	1	WGT-Best.tre
7	str	1	TENT.tre
8	str	1	TENT.50%.tre
9	str	1	Exon.Amino_Acid.tre
10	str	1	TENT+C3.tre
11	str	1	TENT+outgroup.tre
12	str	1	Intron-partitioned.tre

Figure 1. The list of good trees.



idx ▲	Type	Size	Value
0	str	1	TEIT.tre
1	str	1	Exon.Homogenous.c123.tre
2	str	1	Exon.c12-partitioned.tre
3	str	1	Exon.c1-unpartitioned.tre
4	str	1	Exon.c123-unpartitioned.tre
5	str	1	Exon.c3-unpartitioned.tre
6	str	1	Exon.c123-partitioned.tre
7	str	1	Exon.Homogenous.c12.tre
8	str	1	Exon.Heterogenous.c12.tre
9	str	1	Exon.c123-RY-unpartitioned.tre
10	str	1	Exon.c12-unpartitioned.tre
11	str	1	Exon.c2-unpartitioned.tre
12	str	1	Intron.Heterogenous.tre
13	str	1	Exon.Heterogenous.c123.tre

Figure 2. The list of bad trees. All these may be rejected.

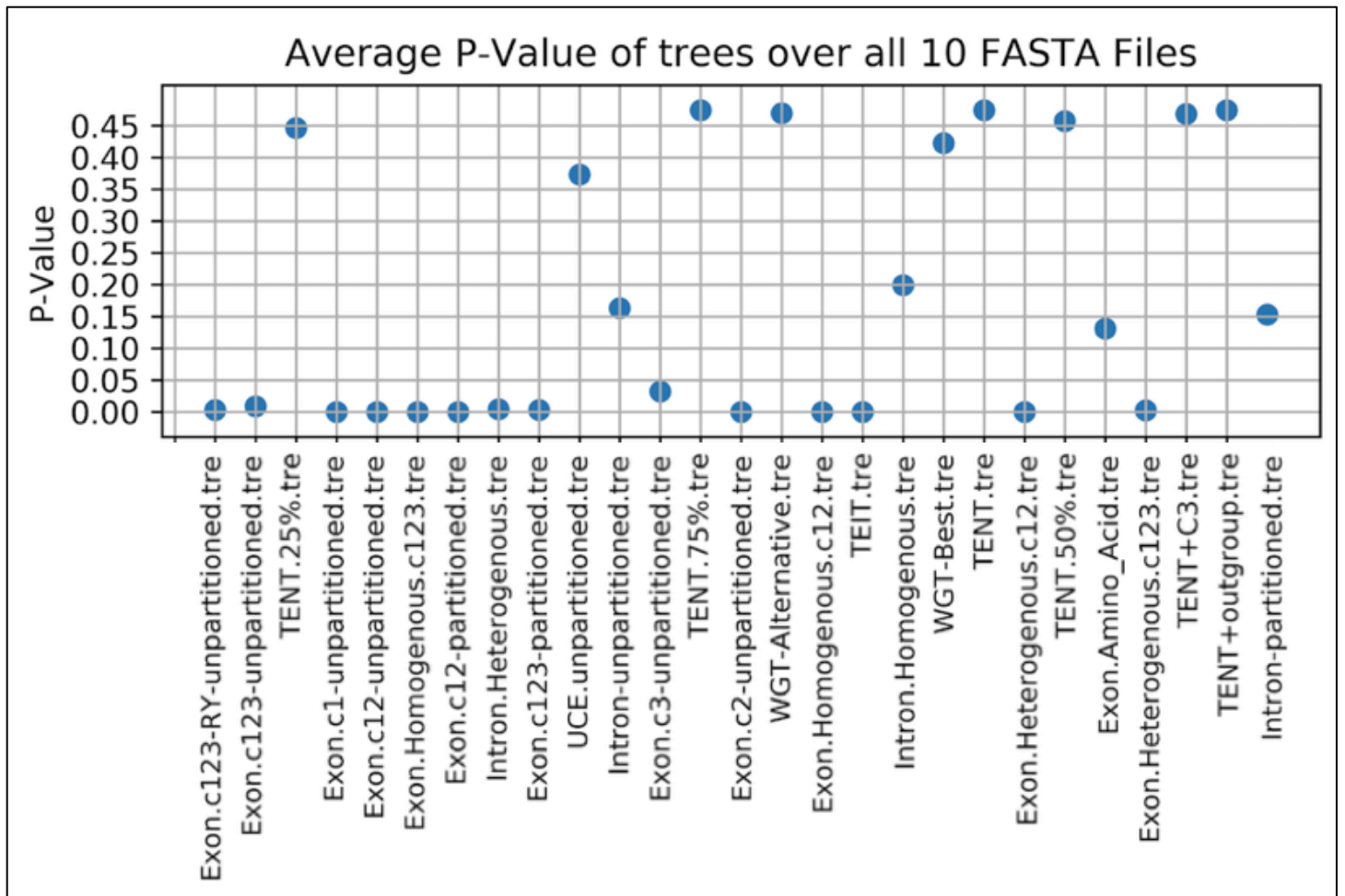
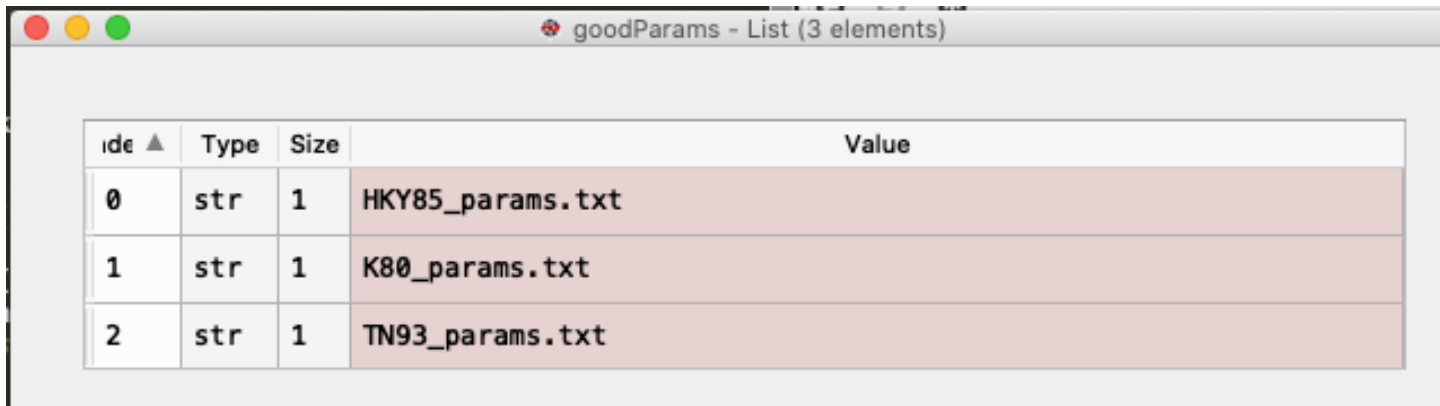


Figure 3, average p-value for all trees.

Clearly, all the trees in the list bad trees can be safely rejected due to a p-value of less than 0.05. The trees likely to have generated the data are those that are greatest (which are close to the base average of 0.4747141, given above): TENT+outgroup.tre, TENT.75%.tre, TENT.tre, TENT+C3.tre, and WGT-Alternative.tre. These are also seen as having high values in the figure above. The trees left ambiguous are those that remain (not having the top value but still in the good-trees list): Exon.Amino_Acid.tre, Intron-partitioned.tre, Intron-unpartitioned.tre, Intron.Homogenous.tre, TENT.25%.tre, TENT.50%.tre, UCE.unpartitioned.tre, WGT-Best.tre.

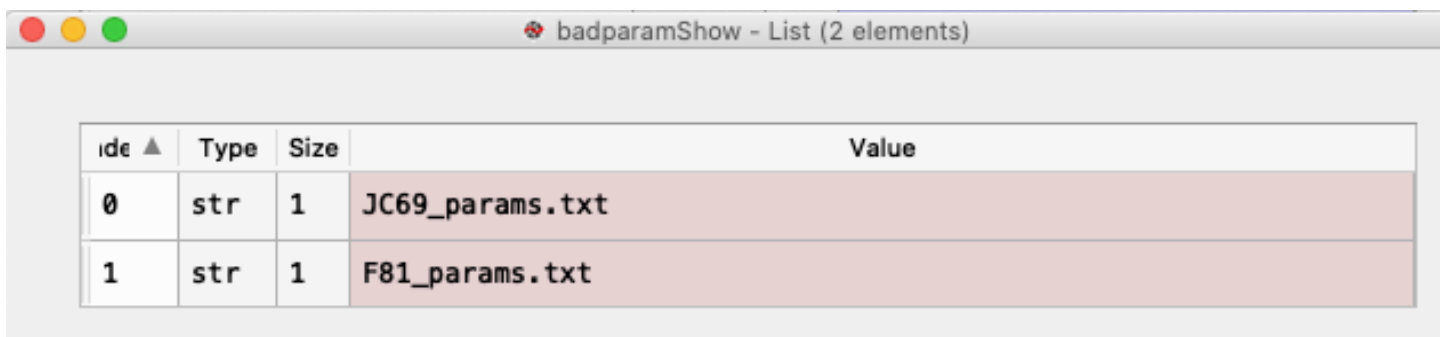
II.

The same figures corresponding to the parameters are now displayed below.



ide ▲	Type	Size	Value
0	str	1	HKY85_params.txt
1	str	1	K80_params.txt
2	str	1	TN93_params.txt

Figure 4. The list of good parameters, with a p-value of greater than 0.05 for all fasta files.



ide ▲	Type	Size	Value
0	str	1	JC69_params.txt
1	str	1	F81_params.txt

Figure 5. The list of bad parameters, having a p-value of less than 0.05 on at least one fasta file.

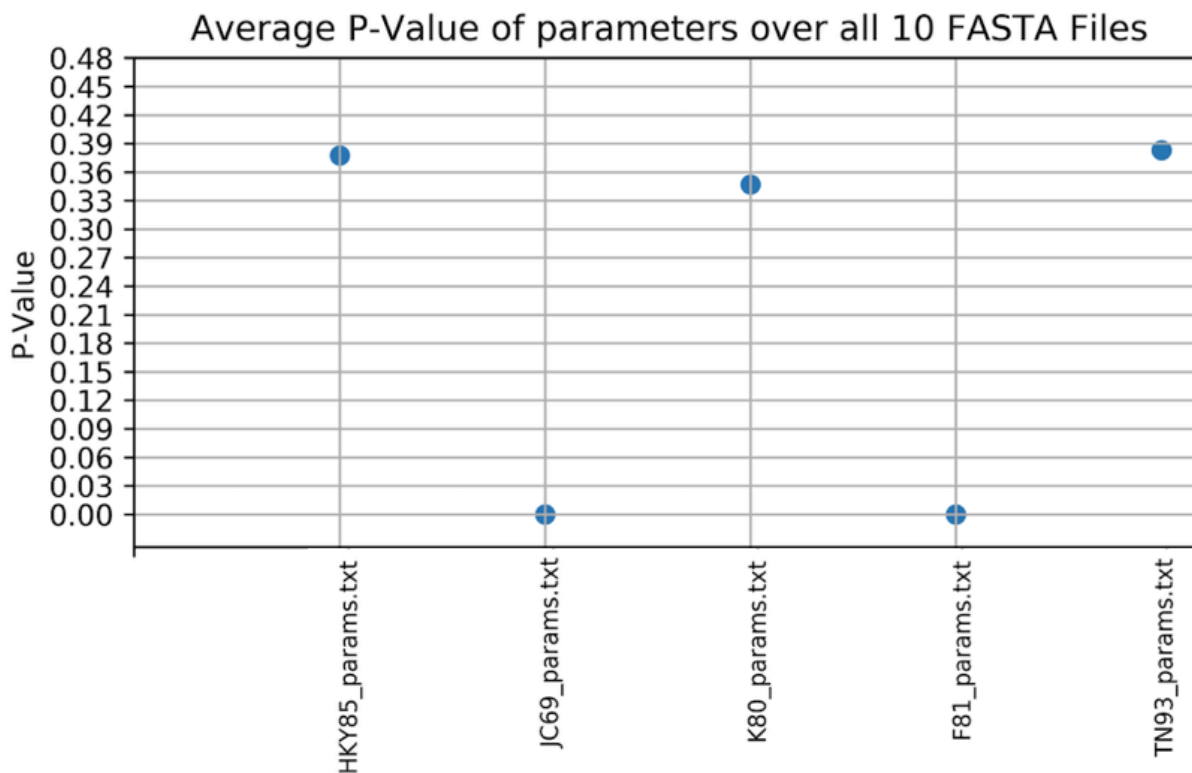


Figure 6, average p-value for all parameters.

Once again the parameters that can be rejected are those list in the bad parameters list above. None of the parameters have an explicitly higher average p-value than that of the GTR model, so none can be considered as good as GTR. However, TN93_params.txt has the highest overall average is likely the best parameter given the choices.

When a model has fewer parameters but cannot be rejected, generally, it is considered preferable. So I would choose K80 (-1)

III.

The best results would likely be given by the tree TENT.tre and the parameters TN93_params.txt, as those each have the highest respective p-value, therefore those would be the ones that this author would chose.

IV.

The bootstrapping was performed with the script generate_Nulls. The idea was to bootstrap sites, so the first thing done was to loop through the desired number of samples and bootstrap on each run-through, saving the result, which gave the desired number of bootstrapped samples. To bootstrap on each run, an array of the length of the sequence was generated with a random site with replacement at each cell, this essentially determined how the sites would be arranged in this bootstrapping. Next a dictionary for new sequences was created. Finally the random array was iterated through, and the character in the original sequences at the site designated was appended to each of the new respective sequences. Then the likelihood of this new batch of sequences was found and stored.

Instead of bootstrapping alignments, you could have easily bootstrapped site likelihoods (matrix L). You would get identical results but would be much much faster (-0)

V.

The author learned that this is a very time consuming process, that is to evaluate the potential for a given tree or parameter set to have generated a set of fasta sequences. It was observed that most of the trees and models did not result in particularly good performance when compared to the GTR model and example tree. It was also observed that due to the time consuming nature only 100 samples were used for the null distributions in this case, when in reality much accuracy could be gained from using more samples. Essentially the potential accuracy of this method is hampered by the significant runtime. Even after generating the bootstrapped sequences, it took a significant amount of time to create the csv file given the p-values for each tree or parameter set. The thing the bootstrapping and csv creation have in common is the compute_likelihood function – it takes a significant amount of time to run and having to run it over a thousand times means a long runtime for the program as a whole.

In a critique of this procedure, there is too much dependence on a slow compute_likelihood function. This could be somewhat mitigated by bootstrapping a different way. This author did not implement an alternative

method, but he imagines that there is a way to redo the bootstrapping such that it runs faster. The author theorizes that since by this bootstrapping method only the order changes, not any of the substitutions since the relative order between all the sequences is preserved, that by simply looking at the changed order and comparing it to the previous, un-bootstrapped order, one could compute the likelihood based on this change, instead of doing the whole likelihood over all the new sequences.

Description and Asymptotic running time for generate_Nulls.py:

Here the script will be described and the runtime will be given. This file is used to generate the null distributions for each of the 10 Fafsa files. It is called the same as the `compute_likelihood` function, with a tree, parameters, sequences, and output. It creates a list of 100 bootstrapped likelihoods, the null distribution, and saves this list in a pickle to be used by the other script for this homework – the one that finds the p-values and creates the CSV. It should be noted that each time this script is run it is called on a specific fafsa file (for example 03.fas), and the name of the pickle to which this distribution will be saved in line 156 of the code should be changed as well to "dist_03.pickle" to make sure it is saved with the correct name.

Since `generate_Nulls` is created from an edited version of `compute_likelihood`, it is assumed the author is familiar with the `compute_likelihood` function as it was created and explained in homework 6. The changes are as follows: there is a for-loop that runs through the number of times of how many samples are desired in the null distribution, calling the function `getSample` each time, which returns a likelihood. This likelihood is stored in an array. The `getSample` function first creates an array of random numbers with replacement of the same length of the sequence and numbers chosen from zero to the length of the sequence. Then it creates a dictionary with the same keys as the sequence dictionary, but with empty lists at each entry. Then the random array is looped through and new sequences are constructed using the site's given in the random array. Finally the function `compute_likelihood` is called to get the likelihood of the bootstrapped sequences and the given tree and parameters. Lastly in the main loop, once all the likelihoods have been found and stored, they are saved in a pickle.

The running time is as follows: in the "main" section of the code, the desired number of samples is looped through, in this case 100 were used to create the null distributions. In this loop, the function `getSample` was called and was added to the array storing each sampled bootstrapped log likelihood. Within `getSample`, there are 2 for-loops; the first loops over the keys in the dictionary of sequences, which will run k times, where k is the number of sequences. The second will be a nested for-loop, with the outside running l times, where l is the length of the longest sequence and the inside running k times, where k is the number of sequences passed in. Then the `compute_likelihood` function is called which has a running time of $O(nls^2)$, where n is number of nodes, l is length of longest sequence, and $s = 4$ for DNA and $s = 20$ for amino acids.

The total runtime for this function `getSample` is then $O(nls^2) + O(k) + O(lk) = O(l(ns^2+k))$

The total runtime of `generate_Nulls`, accounting for the for-loop in the main is then $O(jl((ns^2+k)))$, where j is the number of samples in the null distribution, n is number of nodes, l is length of longest sequence, and $s = 4$ for DNA and $s = 20$ for amino acids.

This can be reasonably simplified since k is the number of sequences but n is the number of nodes, which contains all the sequences, therefore the runtime of this script is simplified to: $O(jlns^2)$.

Description and Asymptotic running time for generate CSV.py:

This function takes in the null distributions found previously and creates the csv file. This function can be called without any arguments passed, but it requires that the pickles containing the null distributions, and the folders 'example', 'trees', and 'params' be in the same directory as the script is being run in. It uses a function called calcP to open the use the appropriate null distribution for calculating the p-value, done using the built in python module for stats. The first thing that happens in the main loop is the appropriate directories are found and searched to create lists of the fasta files, trees, and parameters that must be used to find the p-values in the csv file. Next, dictionaries are defined to store the p-values to be saved in pickles are the end of the program for use in creating visuals for the writeup. Next, all the fasta files are looped through. The first thing in the for-loop is then to get the files for the tree, sequences, and gtr parameters, which are passed to the likelihood function. Then this likelihood is passed to calcP and that line is added to the csv file. Next in the same for-loop is a nested for-loop that loops through all the parameters and does the same thing as before except this time passing the appropriate parameter to the likelihood function. Once again the p-value is found and that line is added to the csv file. Finally there is another nested for-loop which loops through the trees and finds the likelihood and p-value of each corresponding tree and adds it to the csv file. Once this outer for loop has been completed (meaning all fasta files combinations have been added to the csv files), the dictionaries storing the p-values are saved as pickles.

Based on the above description, the running time is affected by the number of fasta files, f , and the number of trees, t , and parameter files, p . Since the likelihood must be calculated for each combination of fasta files and trees and fasta files and parameter files, the asymptotic runtime is: $O(fnls^2(t + p))$, where n is number of nodes, l is length of longest sequence, and $s = 4$ for DNA and $s = 20$ for amino acids.