WHEATON COLLEGE
Norton, Massachusetts

This is to certify that Donald Bass has fulfilled the requirements for graduation with Departmental Honors in Computer Science.

The degree of Bachelor of Arts was awarded on 5-19-12

Registrar

Credit: 2

Director: Mark D. Leblanc
Michael Kahn

Cluster Validation Using the Non-parametric Bootstrap and Parallel Processing: Applications in Unsupervised Machine Learning of Shimodaira's Method to Text Mining and Genomics

BY

Donald Bass

A Study

Presented to the Faculty

of

Wheaton College

in Partial Fulfillment of the Requirements

for

Graduation with Departmental Honors

in Computer Science

Norton, Massachusetts

May 23, 2012

Contents

1 Introduction

Cluster analysis finds patterns in large sets of objects by organizing the set into groups of similar objects. The technique has been applied to many diverse areas of research, including Anglo-Saxon literature (Drout *et al.*, 2011), business management (Joseph *et al.*, 2005), software design (Adnan *et al.*, 2008), and genomics (Buckley *et al.*, 2001). While the technique will always find a possible grouping among a set of objects, cluster analysis by itself does not provide any measure of certainty that the pattern is an accurate reflection of the underlying data. To generate such a measure an additional type of technique known as cluster validation is necessary. trueTree is a tool that will automate the process of performing cluster validation.

The Lexomics group at Wheaton College has been using cluster analysis in their research on Anglo-Saxon literature for several years. During that time they used an R script to run a hierarchical clustering algorithm on their data. During the summer of 2011, this script was connected to a web based interface called treeView which can be found at `http://cs.wheatoncollege.edu/treeview/` to make it easier for people to use. The script however had no capacity for performing cluster validation, and it is this lack of capability that trueTree is intended to address.

1.1 Guide to Chapters

This section describes how the rest of the thesis is organized.

Chapter 2, Literature Review gives a general explanation of the core techniques used in trueTree. As such, the chapter contains a brief overview of both Artificial Intelligence and Machine Learning to provide context, an explanation on how cluster analysis works, the basics of performing cluster validation, and finally the concepts behind parallel processing.

Chapter 3, Methodology, gives an indepth explanation of how trueTree works, walking through the entire process from generating an initial data set to returning final results. This explanation includes a look at the actual software, as well as covering all the different options trueTree provides the user.

Chapter 4, Use Cases, presents results of running data from several different fields of research through trueTree. The section explains how to interpret the results, and shows how trueTree can be applied in other domains. It also contains some benchmarking numbers to show how the performance of trueTree changes based on the datasets used, and the machines trueTree is run on.

Chapter 5, Conclusion, summarizes how trueTree can be used, as well as suggesting how it might be expanded in the future.

Appendix A, Dendrogram Coloring, explores some of trueTree's additional functionality designed to add color to dendrograms created from certain types of datasets in order to make the large dendrograms easier to comprehend.

Appendix B, Setting Up a Machine Cluster in Linux, gives detailed instructions on how to combine a number of Linux machines into a single cluster capable of running trueTree.

Appendix C, Lab Setup, describes the setup of the machines used to run trueTree for this research.

2 Literature Review

Artificial Intelligence (AI) is a field of computer science dedicated to making computers more intelligent. The exact definition of what it means for a computer to be intelligent is disputed. There are many different definitions which can be divided into four categories. Each definition represents a different approach towards developing intelligent machines. These four categories of definitions involve systems that (a) think like humans, (b) act like humans, (c) think rationally, or (d) act rationally (Russell and Norvig, 2003).

The first category of definitions consists of definitions which state that artificial intelligence creates systems that think like humans. Under this definition the approach to developing an intelligent machine is to study and emulate the workings on the human mind. However, in order to emulate the human mind, it is necessary to understand how the mind works, and this led to the creation of

the field of cognitive science, which combines AI and psychology to develop a better understanding of the inner workings of the human mind (Russell and Norvig, 2003).

The second category of definitions states that artificial intelligence creates systems that act like humans. The best example of this line of thought is the Turing Test. The Turing Test was a method proposed by Alan Turing to test if a computer is actually intelligent. The test says that if a human interrogator after giving the computer a series of written questions and receiving a series of written responses, cannot tell whether the responses came from a human or a computer, the computer can be considered intelligent (Russell and Norvig, 2003).

The third category of definitions say that artificial intelligence creates systems that think rationally. Thinking rationally means that the system obeys the rules of logic. The system would start with a database of facts, and make logical inferences from those facts. For instance if two of the facts were "all mammals are warm-blooded" and "dogs are mammals" the computer can infer that since dogs are mammals and all mammals are warm-blooded, dogs must be warm-blooded (Russell and Norvig, 2003).

The final category of definitions asserts that artificial intelligence creates systems that act rationally. Acting rationally means that the system acts to achieve the best outcome. As such, judging if a system is intelligent under this type of definition involves examining the actions it takes. A system that is acting rationally will be able to act in a manner that allows it to achieve its goals, and respond appropriately to any changes in its environment (Russell and Norvig, 2003).

There are many different disciplines that are part of AI. One such discipline is natural language processing, which is focused on the ability to decipher the meaning of written and spoken text. Another is computer vision, which involves extracting information from images so a computer can recognize objects within those images. A third discipline is robotics, focused on making devices that can move around and interact with a physical environment (Russell and Norvig, 2003). Finally the discipline most relevant to this thesis is machine learning. Machine learning is the ability for systems to take in data and "learn" from that data.

2.1 Machine Learning

Machine learning is focused on analyzing large data sets in order to create a model of the underlying processes that formed them. The data sets, analysis, and resulting model can take many forms. As an example, there has been a great deal of interest in using machine learning techniques to write a program that can assist in the diagnosis of medical ailments (Kononenko and Kukar, 2007). The data set might be a collection of medical records detailing the various ailments patients suffered from and the symptoms they displayed. The program would analyze these records to match symptoms with ailments and create a model that can be used to relate a patient's symptoms to an ailment. The underlying processes in the model include relationships such as the "flu gives people high fevers", "asthma makes people short of breath", and so forth. The model would allow the program to go through all these various processes linking ailments to symptoms, in order to determine what ailment best matches a patient's symptoms.

There are countless other applications for machine learning. Astronomers at the Jet Propulsion Lab at the Californian Technical University are using machine learning techniques to create a computer that can correctly classify various types of astronomical objects based on images taken of those objects. Financial institutions create models that can be used to assess the risk of giving people loans. Industries use machine learning techniques to analyze their production process and determine the variables that have the most affect on their overall efficiency (Kononenko and Kukar, 2007).

There are a number of different types of machine learning methods. The most frequently used category is supervised learning. Supervised learning methods are focused on determining a single supervised variable. One type of supervised learning methods is classification. Classification methods work with objects that can be described by a set of attributes the objects possess. These attributes can be discrete, meaning there is a finite set of possible values the attribute can have, or continuous meaning that there is an infinite set of possible values. Returning to the example of a program that diagnoses medical ailments, an example of a discrete attribute would be whether or not a patient

reports a headache, since there are only two possible values for that attribute: yes or no. An example of a continuous attribute would be the patient's weight which could theoretically be any number and as such has an infinite number of possible values. All of the objects used by any particular method will share the same set of attributes, but the values of those attributes may differ between objects. There is also a set of categories, also known as classes, which these objects can be assigned to, with each object belonging in a single one of those classes (Larose, 2005). As an example, the objects might consist of a number of different organisms, and the attributes those objects posses are types of information about their genome. The classes then might be a particular trait of the organism such as the temperature range in which they live (Dyer *et al.*, 2007).

Continuous Data Set

Attribute	Value
Weight	195 lb
Blood Pressure	112/64 mmHg
Pulse	80 BPM
Temperature	98.8° F

Discrete Data Set

Attribute	Value
Headache Reported	Yes
Dizziness Reported	No
Gender	Male
Joint Pain Reported	No

Figure 2.1: Two sample objects for a program that diagnoses medical ailments. The top object contains only continuous attributes, the bottom object contains only discrete attributes.

Classification methods create a type of model called a classifier which can be used to take an object that belongs to an unknown class and based upon its attributes determine the class where it most likely belongs. The classifier is created from a set of objects known as a training set with each object having a known classification. The exact form the classifier takes and the method used to create it can vary widely based upon the classification method used. A decision tree based method creates a model that takes the form of a tree. When an object is run through the model it starts at the root of the tree. Each node in the tree examines a particular attribute of the object and sends it down one of several branches based on the particular value of that attribute. When the object reaches one of the leaves of the tree, that leaf specifies which class where it most likely belongs (Kononenko and Kukar, 2007).

A method using a nearest neighbors classifier works differently. Running an object through a nearest neighbors classifier involves examining all the objects in each class used to create the classifier and assigning the object to the class with the most similarity to the object. There are also numerous other types of classification methods, such as Bayesian classifiers, discriminant functions, and support vector machines (Larose, 2005).

Another category of supervised methods is regression. Regression is similar to classification but does not work with objects that can be divided into a discrete set of classes. Instead regression works with objects that have a continuous variable with an unknown value. The actual regression method is used to create a model called a regressional predictor that can be used to predict the value of the unknown continuous variable. The whole process is similar to the process of creating a classifier. There is an initial training set which is filled with objects which have a known value for the continuous variable. This set is fed into an algorithm and that algorithm creates the regressional predictor. Many of the methods used for classification like decision trees and support vector machines

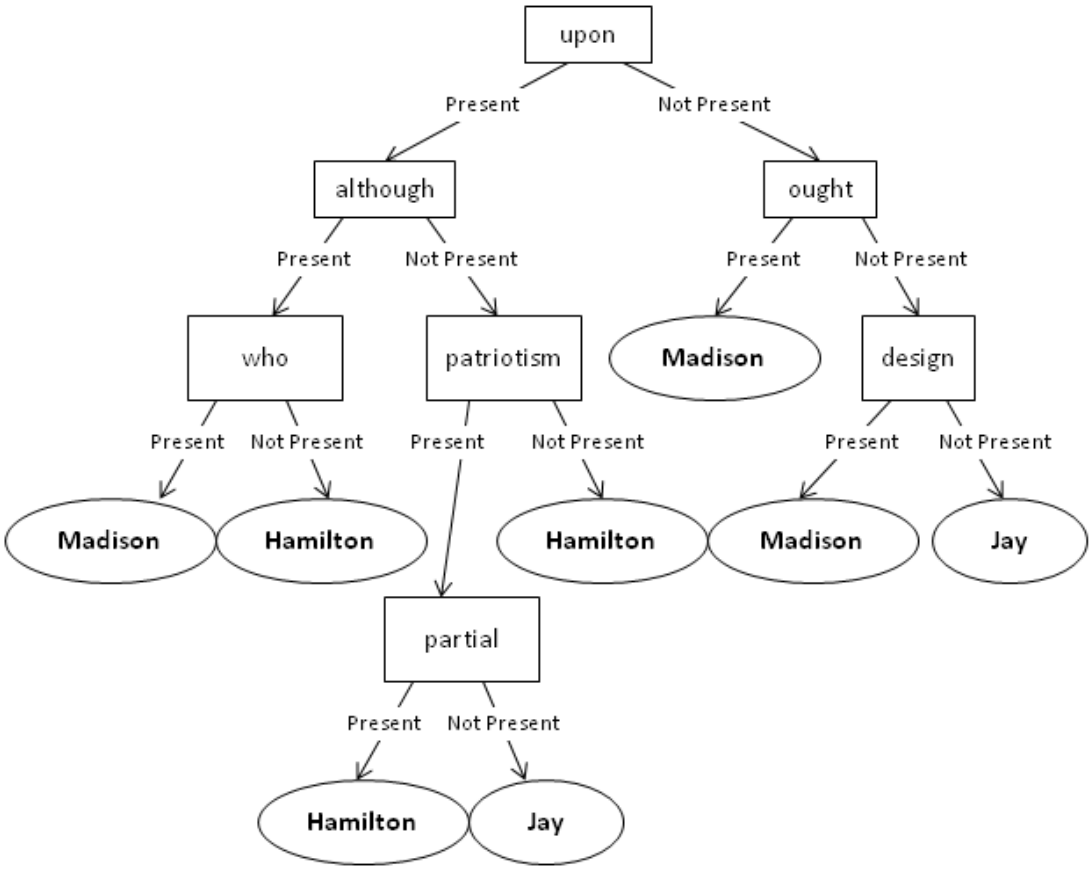

Figure 2.2: Sample decision tree, generated using Mallet's (McCallum, 2002) decisiontree trainer to determine authorship of one of the Federalist Papers. Rectangular nodes check for the presence of that particular word in the paper. Oval shaped nodes denote the predicted author based on preceding nodes in the decision tree.

can be altered so they can be used as regression methods (Kononenko and Kukar, 2007).

As a side note, the objects used in supervised learning methods as well as those in unsupervised learning methods use a number of different names in the literature. The objects have been referred to in various texts as objects, examples, observations and chunks. This thesis is mainly focused in the area of text mining in which the name that is most commonly used is "chunks" (for chunks or sections of a text), so from here forth all such objects will be called chunks.

Another field of machine learning is reinforcement learning which involves repeatedly performing the same task over and over again to determine the best way to undertake it. Reinforcement learning focuses on the actions of some autonomous agent that has to make a series of decisions about what actions to perform. For instance, reinforcement learning might be used to teach a computer how to best play a game like chess, where the autonomous agent would be the program deciding what moves to make, and the decisions it is making being the particular moves to make. Every time the agent performs an action it is given either a positive or a negative reward based on the results of either that action, or that action and all the actions leading up to it. To go back to the chess example, the program might be given a positive reward if a move improves its position on the board by letting it capture an enemy piece, and similarly the program would be given a negative reward if the move worsens its position by letting a piece get captured. The overall goal of reinforcement learning is to determine what actions can be taken which will maximize the cumulative reward that it receives (Kononenko and Kukar, 2007).

Unlike the supervised learning methods of machine learning discussed above, this thesis applies clustering, a type of unsupervised learning. Unsupervised learning methods determine general relationships instead of finding the value of a single specific variable like supervised learning techniques methods do. Clustering methods in particular take in a set of objects as input that is similar to the sets that classification and regression techniques use. However instead of using this set of objects to create a model, clustering methods divide all the objects into a number of different groups, sometimes referred to as either clusters or clades, with each clade containing objects that are "similar" to each other. There are two main categories of clustering methods partitional clustering and hierarchical clustering (Larose, 2005).

2.1.1 Partitional Clustering

Partitional clustering algorithms divide a set of chunks into a specified number of clades. Usually the algorithms either try to make sure each clade is compact, meaning that all the chunks in a clade are similar to each other, or make sure the clades are separated, meaning that the chunks in any one clade are dissimilar to the chunks in the other clades. Partitional algorithms are well suited for applications that involve large data sets, as using certain other types of algorithms such as hierarchical algorithms can be computationally prohibitive.

The most popular Partitional Clustering algorithm is the k-means algorithm. The algorithm inputs a set of chunks and a number k which is the number of clades that will be created. The algorithm first randomly divides the chunks into k clades. For each clade, it finds the average of all the chunks inside that clade. Each chunk is then moved into the clade which has the average closest to that chunk, and new averages are calculated for the new clades. This is repeated until a set of averages are computed that cause no changes in the clades topography (Kononenko and Kukar, 2007).

2.1.2 Hierarchical Clustering

Hierarchical clustering algorithms build a hierarchy of clades showing how similar all the items in the input set are to each other. There are two categories of Hierarchical Clustering Algorithms, agglomerative, and divisive. Agglomerative algorithms put all the items in the dataset into separate groups and joins those groups together. Divisive algorithms put all the items in the same group and divide that group into smaller groups (Everitt *et al.*, 2011). The research in this thesis was done using agglomerative methods.

Agglomerative algorithms start by finding the two most similar chunks and joining them together into a clade. The similarity of two chunks is more commonly referred to as the distance between

them, and the higher that distance is the less similar they are, and as such the smaller the distance between two chunks the more similar they are. This original clade is then treated as a single chunk whose attributes are then "merged" into a joint measure such as the average of all the chunks in the clade. The algorithm then finds the next two most similar chunks and joins those into a clade. This process is continued until in the end a single clade is formed containing all the smaller clades. (Kononenko and Kukar, 2007).

Divisive algorithms in comparison can work in two different fashions. One way is to use polythetic divisive methods which, starting with an initial clade containing all the items in the dataset, finds the chunk least similar to the other chunks in the group. This chunk is then made part of a splinter clade, and all the other chunks more similar to the splinter clade than the original clade are moved into the splinter clade. This process is repeated on the new smaller clade until every chunk is contained in a separate clade (Everitt *et al.*, 2011).

The second type of divisive algorithm uses monothetic divisive methods. These methods split clades using certain attributes. This means that if a clade is split using attribute A all chunks that contain that attribute would be put in one clade and all the chunks that lack that attribute would be put in a separate clade. The algorithm will decide which attribute to use by finding the attribute which creates the two most homogeneous clades, meaning that all the chunks in each clade will be as similar as possible to all the other chunks in the clade. Again the algorithm will start with a single clade containing all the chunks and keep splitting clades until every chunk is in its own separate clade (Everitt *et al.*, 2011).

No matter what method is used to obtain the final result, the result will be depicted as a tree. The tree will have all the chunks as leaves on the bottom and join them together in order from most similar to least similar. Chunks that are similar are connected into a clade towards the bottom of the tree, while chunks that greatly differ are connected towards the top. This vertical positioning of a clade is also known as the clade's height. This tree is commonly referred to as a dendrogram (Kononenko and Kukar, 2007). A rather simple example is displayed in Figure 2.3.

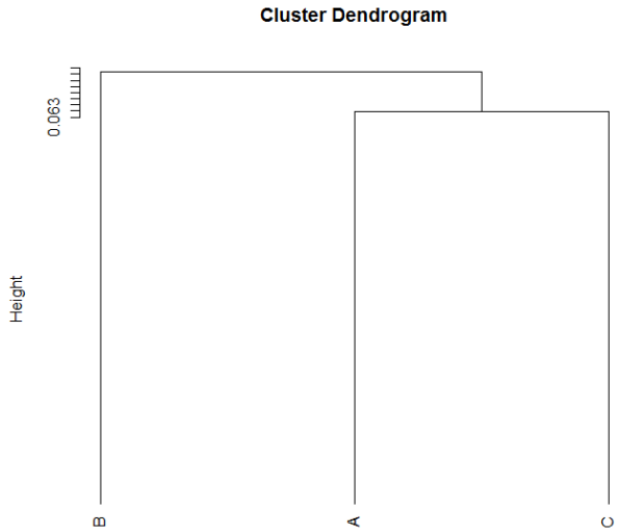

Figure 2.3: A trivial three-leaf dendogram as output from a hierarchical clustering algorithm where chunks A and C are most "like" and joined first.

Figure 2.3 shows the resulting tree from dividing a text file into three chunks labeled in order A, B and C and running them through a clustering algorithm. The first clade contains chunks A and C, meaning those two parts are more similar to each other than to chunk B. This clade formed fairly high up on the tree meaning that the two chunks are not all that similar. Next a clade is formed containing this initial clade and chunk B. Since this clade is formed only a little higher up on the

graph than the first clade, chunk B is not much more different from the average of chunks A and C than chunks A and C are from each other.

There are a number of different methods for finding the distance between two chunks, such as two parts of a text (Kraus, 2010). One general category of methods are known as Minkowski metrics where the distance between chunks i and j is

$$d_{ij} = (\sum_{k=1}^{a} w_k^{\lambda} |v_{ik} - v_{jk}|^{\lambda})^{\frac{1}{\lambda}} \qquad \text{(Formula 2.1)}$$

where v_{ik} is the value of the kth attribute in the ith chunk, a is the number of attributes, w_k is the weight of the kth attribute and λ is a parameter that varies depending on the particular metric. The metric most commonly used in this research is the Euclidean metric which is just a Minkowski metric where $\lambda = 2$ (Kononenko and Kukar, 2007).

To demonstrate how this works suppose the following table contains a set of two chunks A and B each of which has two attributes.

Chunk	A	B
Attribute 1	2	5
Attribute 2	4	8

Table 2.1: Dataset with values of attributes 1 and 2 for two chunks, A and B

Using the Euclidean metric and assuming all of the attributes have a weight(w) of 1 the distance between chunks A and B is calculated below.

$$\begin{aligned}
d_{AB} &= (\sum_{k=1}^{a} |v_{Ak} - v_{Bk}|^2)^{\frac{1}{2}} \\
d_{AB} &= ((|2-5|^2) + (|4-8|^2))^{\frac{1}{2}} \\
d_{AB} &= ((|-3|^2) + (|-4|^2))^{\frac{1}{2}} \\
d_{AB} &= ((3^2) + (4^2))^{\frac{1}{2}} \\
d_{AB} &= (9+16)^{\frac{1}{2}} \\
d_{AB} &= 25^{\frac{1}{2}} \\
d_{AB} &= 5
\end{aligned}$$

Thus, using the data shown in Table 2.1 the Euclidean distance between chunks A and B is 5.

Once the first clade is formed, measuring the distance between chunks becomes more complicated, since the clade counts as a single chunk. There are a number of different techniques devised to calculate the distance between two clades. These techniques can also be used to find the distance between a clade and an individual chunk since an individual chunk can be treated as a clade that only contains that one chunk. The technique used in this thesis for the distance between two clades takes the mean of the distances between each chunk in the first clade and each chunk in the second. The formula to find the mean distance between two clades is

$$d(X,Y) = \frac{1}{|X||Y|} \sum_{i \in X} \sum_{j \in Y} d_{ij} \qquad \text{(Formula 2.2)}$$

where X and Y are two clades, $|X|$ is the number of chunks in clade X, and d_{ij} is the distance between chunk i in clade X and chunk j in clade Y. As such the mean distance is simply the average of the distances for every pair of chunks in the two clades (Kononenko and Kukar, 2007).

To demonstrate suppose the following table contains a set of three chunks each of which has two attributes.

Chunk	A	B	C
Attribute 1	2	5	9
Attribute 2	4	8	13

Table 2.2: Example dataset showing values of attributes 1 and 2 for three chunks(A, B, and C)

If chunks A and B are first joined together into a clade AB the mean distance between that clade and chunk C is calculated below, where i and j here refer to each chunk within clade AB, and C respectively:

$$d(AB, C) = \frac{1}{|AB||C|} \sum_{i \in AB} \sum_{j \in C} d_{ij}$$
$$d(AB, C) = \frac{1}{2 * 1}(d_{AC} + d_{BC})$$
$$d(AB, C) = \frac{1}{2}(11.4 + 6.4)$$
$$d(AB, C) = \frac{1}{2}(17.8)$$
$$d(AB, C) = 8.9$$

Thus using the data in Table 2.2 the distance between the clade of AB and C is 8.9.

Calculating the distances between clades is central to hierarchical clustering. The basic algorithm for agglomerative clustering is as follows:

1. Assign each chunk to a clade containing just that chunk.
2. Compute the distances between all the clades.
3. Take the two closest clades and merge them into a single clade.
4. Compute the distance between this new clade and all of the remaining old clades.
5. Repeat steps 3-4 until all the chunks are contained in a single clade (Kononenko and Kukar, 2007).

For example using the data in Table 2.2, the following would occur. The algorithm would create three clades containing respectively chunks A, B, and C. It would calculate the distances between these three clades and store them in an array called a distance matrix that would contain the following information.

	A	B	C
A	0	5	11.4
B	5	0	6.4
C	11.4	6.4	0

Table 2.3: Initial distance matrix for the chunks in Table 2.2

Each cell in the matrix contains the distance between the clades specified in the row and column labels. The algorithm would then look at these distances and see that A and B are the closest pair of clades with a distance of only 5 so it would merge the two clades. A new distance matrix would then be generated that would look like this

	AB	C
AB	0	8.9
C	8.9	0

Table 2.4: New distance matrix after joining chunks A and B into a clade

Clades AB and C are now the closest clades so they get merged into a single clade. Now all three chunks are part of a single clade so the algorithm ends. The final dendrogram is shown in Figure 2.4.

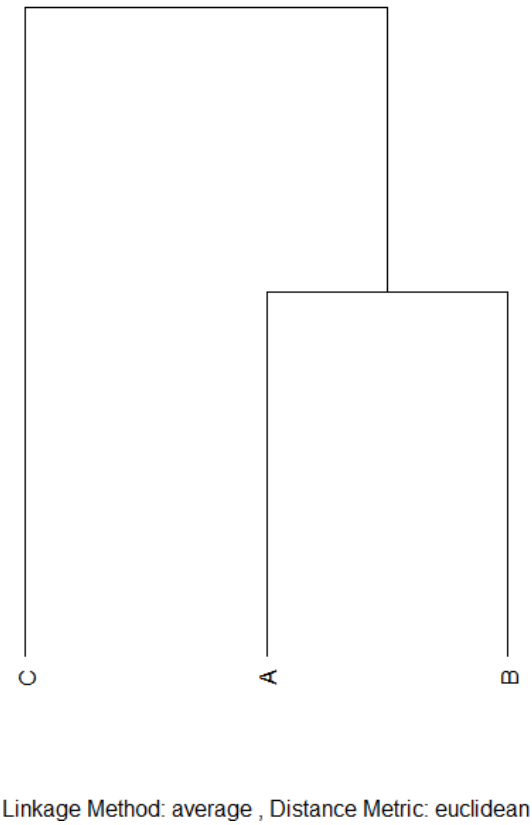

Figure 2.4: Results of hierarchical clustering algorithm.

One field in which these techniques have been applied is in determining the authorship of Anglo-Saxon literature. There are a number of different works whose true author is unknown. By running those works and other pieces of writing with known authors through a hierarchical clustering algorithm it is possible to determine which pieces of literature appear to be similar. A similarity between two different works usually occurs because they were either both written by the same person, or both were based on the same source material. For more information see Drout *et al.* (2011).

2.2 Cluster Validation

Cluster Validation is a process that involves analyzing the results of a previous cluster analysis to test how likely it is that the clustering represent a structure that actually exists in the original source. Cluster Validation is important to address two potential problems that may arise during the initial cluster analysis. The first is that some clustering algorithms possess biases towards certain partitions and as such the results of the analysis can show the bias instead of the true structure. The second issue is that clustering algorithms will always cluster data even when no actual pattern exists inside that data. Cluster Validation can be used to distinguish cases where the algorithm found an actual structure within the data and when the algorithm invented a structure that never existed before.

There are two different categories of cluster validation methods: external measures and internal measures. External methods are used to evaluate the performance of algorithms. They involve

taking a well known data set that already has a commonly agreed on "true" clustering and running it through a clustering algorithm. The results are then compared to the "true" results and the closer an algorithm's results are, the better the algorithm. As these methods require knowledge of the correct clade arrangement they are outside the scope of this paper. Internal measures, on the other hand, analyze the output of a clustering algorithm to determine the quality of the clustering. Unlike external methods, there is no reliance on knowledge of the proper outcome (Handl *et al.*, 2005). Instead the only information needed is the result of the clustering algorithm and the contents of the initial data set. The various types of internal measures are discussed next.

2.2.1 Compactness, Connectedness, and Spatial Separation Methods

One method of validating the quality of a clustering is to see how well the clustering exhibits one of several desirable traits. The three main traits are Compactness, Connectedness, and Spatial Separation. Algorithms focusing on compactness look to see if the clades are "compact", meaning that there is little variation between the different chunks within a cluster. Algorithms focusing on connectedness check to see if chunks that are similar to each other share the same clade. Finally, methods based on spatial separation check to see if the clades greatly differ from one another. Researchers frequently combine multiple methods from these categories, especially Compactness and Spatial Separation methods when performing validation. A particular clustering can be said to be better than an alternative if it does at least as well as the alternative for all the methods being used, and does better for at least one (Handl *et al.*, 2005).

2.2.2 Cophenetic Correlation Coefficient

The Cophenetic Correlation Coefficient is another measure to determine the quality of a given clustering. It is calculated by creating a distance matrix showing how far apart the different chunks are in the final clustering. This distance is simply the height of the first clade to contain both chunks. This distance matrix is then compared to a distance matrix created from the original data showing how far apart the actual chunks are from each other. The more correlated the two matrices are the more likely the clustering is valid, since it means the clustering is preserving the original distances (Farris, 1969).

As an example consider the following dataset.

Chunk	A	B	C
Attribute 1	1	4	3
Attribute 2	1	3	3
Attribute 3	1	0	1

Table 2.5: Example dataset showing values of attributes 1 and 2 and 3 for three chunks(A, B, and C)

The distance matrix for this dataset would be

Chunk	A	B	C
A	0	0.091	0.64
B	0.091	0	0.56
C	0.64	0.56	0

Table 2.6: Initial distance matrix for the chunks in Table 2.5

and the dendrogram created by running the data through a hierarchical clustering algorithm would look appear as follows.

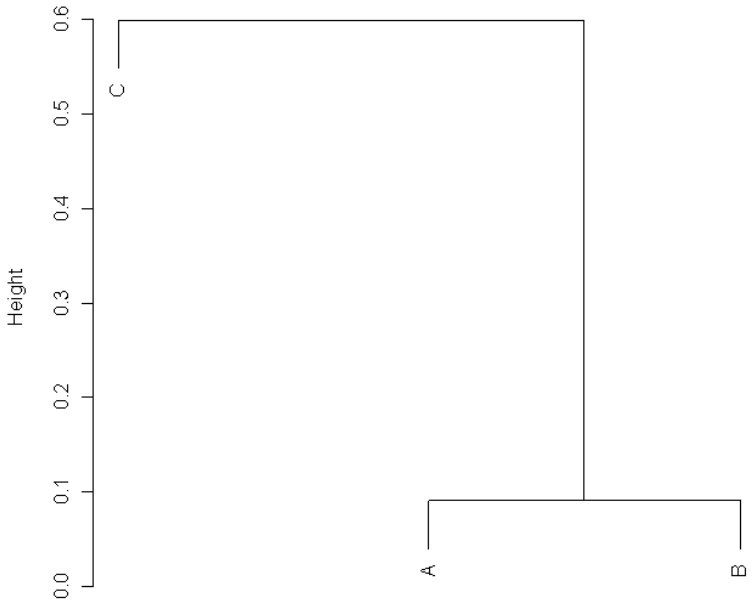

Figure 2.5: Dendrogram created using the dataset in Table 2.5 as input for an hierarchical clustering algorithm. A and B were merged at height 0.091 and AB and C were merged at height 0.60.

The cophenetic distance matrix for this clustering is created by finding the height at which the different chunks were first put in the same clade. A and B were merged at height 0.091 so the distance between them is 0.091. A and C were first put in the same clade when clade AB merged with clade C at height 0.60 so the distance between A and C is 0.60. B's first appearance in the same clade as C occurred at the same point as A, so the distance between B and C is also 0.60.

Chunk	A	B	C
A	0	0.091	0.60
B	0.091	0	0.60
C	0.60	0.60	0

Table 2.7: Cophenetic distance matrix for clustering seen in Figure 2.5

The Cophenetic Correlation Coefficient for this example is the correlation between the original distance matrix seen in Table 2.6 and the new Cophenetic distance matrix seen in Table 2.7. The Coefficient is calculated by comparing the columns of the first original distance matrix with the columns of the cophenetic distance matrix using the Pearson method. For more information see Hogg *et al.* (2005). In this example the coefficient can be calculated to be 0.6101.

2.2.3 The Bootstrap

The bootstrap method is a computer-intensive method for using a dataset to measure distributional properties of a statistic based on that dataset. In simple cases this method is used to determine how similar a sample is from the original population. The ideal method to determine the similarity would be to take every possible sample of a population that is same size as the original sample and then compare all the samples. However this method is usually infeasible. The formula for the number of unique samples in a population is $\frac{n!}{r!(n-r)!}$ where n is the size of population, and r is the size of a sample. As such even with a small population of size 100, and a similarly small sample size of 10 there are $1.73 * 10^{13}$ unique samples of size 10 that can be taken from the population. If

the information on the entire population had already been collected it would be computationally intensive to compute all the unique samples. If, on the other hand, the only information possessed on a population was the original sample collected, collecting all unique samples would be next to impossible. This is where bootstrapping can be applied.

The bootstrap involves creating new samples using only the original sample and pretending that the sample is actually the entire population. The new samples are created by using a technique known as resampling with replacement. The resampling part means that we are picking a set of new data points that is most commonly the same size as the original sample out of the values in the original sample. The replacement part means that the same data point from the original set can be selected multiple times, and this is what prevents the new sample from being identical to the original. So if the original sample had the values 5, 8, 9, and 10, when the new sample is created it might contain the values 5, 8, 9, 8, with the 8 having been picked twice due to replacement. If this process is repeated enough times, the distribution of the bootstrap will be close to the distribution of all the actual samples of that size that might be picked from the entire population (Hogg *et al.*, 2005).

This process can be applied to cluster validation by resampling the original dataset used to create the original clustering in order to create a large number of new datasets. Each of these datasets is then used to find a new clustering which is compared to the original clustering. The general idea is yo see what clades are stable and consistently reappear despite the clusterings being created from the resampled datasets and what clades are volatile and don't appear often in the resampled datasets. The "real" clades should be stable since they should reflect actual patterns in the data that won't be removed by minor changes from resampling while clades that aren't "real" are formed from the noise in the dataset and should prove volatile when the dataset is resampled. As such if the original clustering accurately reflects the original dataset these new clusterings should be similar to the original one, and if not the new clusterings should differ.

2.2.3.1 AU and BP Values There are a number of different methods that have been devised to take the results of using bootstrapping and turn them into a numerical value that measures how reliable the original clustering was. The method used by trueTree is the approximately unbiased (AU) test which requires the results of another method, the Bootstrap Probability(BP) to calculate. Both methods produce a number that gives the percentage chance that a clustering is "valid" which makes it essentially a measure of how confident once can be about the results. Both AU and BP measures can be used on a per clade basis, giving a measure of how likely each clade in the clustering is a valid clade. This is in comparison to some other measures like the Cophenetic Correlation Coefficient which only give a measure of confidence for an entire dendrogram (Shimodaira, 2004).

The bootstrap probability is a widely used method for cluster validation. The bootstrap probability for a clade is calculated by finding the percentage of times while running the bootstrap that the clade was present in the clustering from a resampled dataset. If 10 new datasets were generated while bootstrapping, and clade 5 from the clustering of the original dataset was present in the clusterings of five of those new datasets, clade 5 would have a BP of 50 (Shimodaira, 2004).

The approximately unbiased test is a variant of the BP test, created by Hidetoshi Shimodira to correct some of the inherent bias in using BP values. The AU value is found by changing the size of the resampled datasets over the course of running the bootstrap. So there would be some datasets generated that were smaller then the original dataset, some the same size, and some that are larger. For each of these sizes of the resampled datasets, a BP value is calculated from just the datasets of that size. These BP values are plotted and a least squares estimate is found, which can then be used to calculate the AU value (Shimodaira, 2004). The exact details of how this calculation works are discused in section 3.4. Due to the AU measures being less biased then the BP measures only the AU measures are used in this research.

2.3 Parallel Processing

Parallel Processing involves solving a computational problem using multiple processors. When used on a small scale this often involves using the multiple processors found in most modern Central Pro-

cessing Units (CPUs), but for computationally intensive problems, it is often necessary to combine the computing power of multiple computers across a network. This usually requires some special software to handle the proper distribution of work to all the processors at hand.

CPU's that have multiple processors are also known as multicore processors. Each core/processor is an independent piece of the hardware capable of independently executing a sequential program. Each processor has its own cache which is a small store of memory located near the processor for quick access. However the processors share the rest of the computers resources including the vast majority of the memory. The operating system of the computer handles distributing work across the different processors (Herlihy and Shavit, 2008).

Not all problems can be solved in parallel. Some algorithms, and by extension some problems, are inherently sequential meaning that each step in the algorithm relies on the results from the step that came before. Even problems that can be run in parallel often contain certain steps that are sequential, which limits the reduction in total run time that may be achieved when adding additional processors. For instance, if an algorithm starts with 4 sequential steps that take 20 seconds to run, when the final program is run, no matter the distribution of work across processors, those first 4 steps will still take 20 seconds to run, since these steps cannot be run simultaneously.

In general, there are several ways to run algorithms in parallel. One way involves working with algorithms that perform a large number of independent computations. A simple example would be adding a long list of numbers. This can be run in parallel by giving every processor a portion of that list. Each processor then finds the sum of its portion of the list, and once all the processors are finished, those partial sums are added together for the final sum. Since the order of additions does not matter, the computations can be considered independent. This is illustrated in pseudocode in Listing 2.1.

```
1
2  function addParallel(numProcessors, list, listSize)
3    sublistSize = listSize / numProcessors #split work across
4                                           #processors
5    for(i = 0 to numProcessors - 1)
6      addParallelNode(i, list[sublistSize * i ... sublistSize * i + 1],
7          results[i])
8
9    wait until Processors(0 .. numProcessors - 1) are done
10   totalSum = 0
11
12   for(i = 0 to numProcessors - 1)
13     totalSum += results[i]
14
15 function addParallelNode(ProcessorNumber, sublist, answerDestination)
16   have Processor ProcessorNumber do:
17     sum = 0
18     for i in subList
19       sum += i
20     answerDestination = sum
```

Listing 2.1 Pseudocode to add a list of numbers in parallel

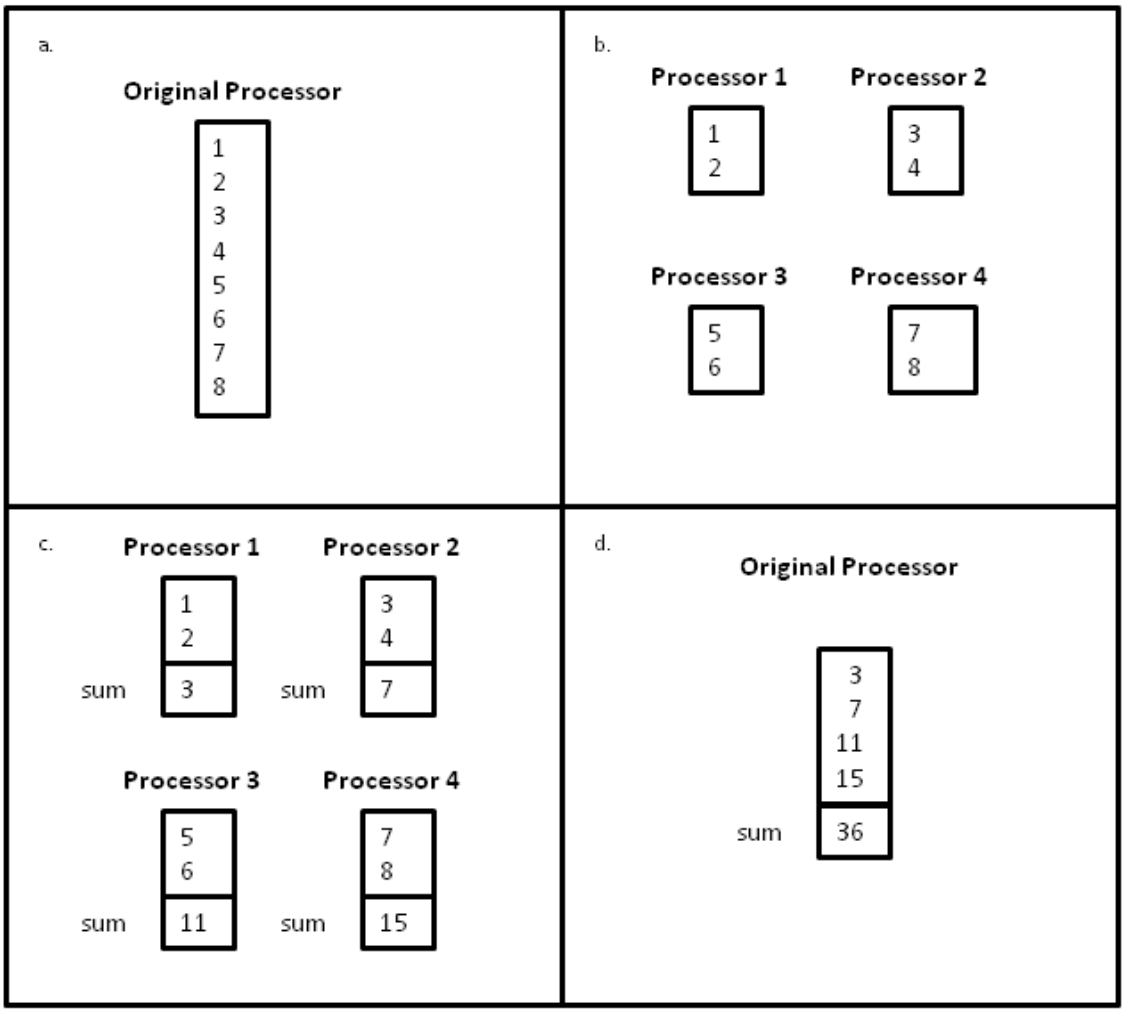

Figure 2.6: The process of running the code in Listing 2.1 in order to find the sum of integers 1-8 using 4 processors. (a) Shows the initial list of 8 numbers to add. (b) Shows the 4 lists sent to the four processors. (c) Shows the four processors finding the sum of their individual lists. (d) Shows the 4 sums being returned to the original processor and added together to arrive at the final sum.

Listing 2.1 is actually slightly simplified compared to the algorithm that would be necessary to actually solve this problem, since it assumes a list where the length can be evenly divided by the number of processors. The algorithm assigns a function to each processor, and that function is then run on that processor. In each instantiate the function is given a subsection of the list, finds the sum, and then stores it in a location where the original function can access the partial sum. Once all the processors have finished their workload the original function takes those partials sums and calculates the final sum.

Another way to parallelize an algorithm that has to be run on multiple pieces of data is to divide the algorithm into a number of distinct stages, each of which will then be handled by a separate processor. The first processor will run the initial stage on the first piece of data and then pass the results on to the second processor, and afterwards will run the same stage on the second piece of data, passing that result on as well and so forth. The second processor will run the second stage on the data it receives and pass on its results to the third processor. This technique is also known as pipelining.

2.4 Existing Tools

2.4.1 pvclust

trueTree is built on top of pvclust an R package written by Ryota Suzuki, and Hidetoshi Shimodaira for performing multiscale bootstrap resampling to calculate AU and BP values for hierarchical clusterings. pvclust was originally designed for various types of genetic research such as DNA microarray data analysis. pvclust provides functions to run the bootstrap, find AU and BP values for every single clade in a hierarchical clustering, and produce a dendrogram showing both the hierarchical clustering and all the AU and BP values calculated. In addition pvclust includes a number of functions to help analyze the results such as plotting the curve fitting done to produce the AU values and plotting the standard error for the calculated values (Suzuki and Shimodaira, 2011).

2.4.2 snow and snowfall

snow is a package for the language R designed to add support for some basic parallel processing (R Development Core Team, 2011), (Tierney *et al.*, 2011). snow creates a cluster of processors. To avoid confusion with clustering algorithms, this type of cluster will be referred to as a machine cluster. A machine cluster is just a data structure that stores how many processors should be used, and the information on how to access them. These processors can be multiple cores on a single computer or distributed across a number of different computers. Work is then performed by using a variant of R's `apply` function. The `apply` function reads in a list as input and applies a function to each element of the data structure. The results are stored in a new list which is the same size as the original and each element is the result of applying the function to the corresponding element in the original list. The version of the `apply` function snow provides operates similarly, but instead of passing the function a single list, it's passed a list containing one list for each processor in the machine cluster. Each sublist is then passed to a different processor. The processors then apply the function to sublist they received, and return a list containing the results. Once all the results are returned to the original caller they are each stored in part of a larger list. snow also provides functions to help ensure that all the processors can access the code they need to run and perform some simple performance analysis.

snowfall is an additional R package that serves as a wrapper for snow as well as providing some additional functionality (Knaus, 2010). snowfall simplifies the process of creating machine clusters in snow and also provides the ability to save the results of the intermediate run of an algorithm. This means if code using this feature is stopped in the middle of execution then when the code is next run it can pick up from where it left off. A simple example program written using snowfall is provided below.

```
1  #This program adds the numbers 1 through 10000 together using
2  #the snow library to preform parallel computations
3  library(snowfall)
4  sfInit(parallel=TRUE, cpus=3,type='SOCK') #create a cluster of
5                                            #3 cpus. SOCK tells
6                                            #snow to use a raw
7                                            #socket machine
8                                            #cluster.
9  nums <- (1:10000) #create a list of numbers from 1 to 10000
10
11 splitList <- sfClusterSplit(nums) #split the list up a
12                                   #separate sublist
13                                   #for each processor in the
14                                   #machine cluster
15 results <- sfClusterApply(splitList, sum) #have each processer
16                                           #find the sum of
17                                           #it's own sublist
```

```
18
19 #add the sums of each sublist together
20 total = 0
21 for(i in 1:length(results))
22 {
23     total = total + results[[i]]
24 }
25
26 print(total) #print results
```

Listing 2.2 R script using the snowfall library to add 10,000 numbers together using 3 processors in parallel.

The script shown in Listing 2.2 finds the sum of all the numbers between 1 and 10000, and does the calculations in parallel across three CPUs. The `sfInit` function creates a raw socket machine cluster of 3 processors. The next line creates a list containing the numbers from one to ten thousand. The `sfClusterSplit` function tries to evenly divide the list into a number of smaller lists of equal size. The exact number of smaller lists is equal to the number of processors in the machine cluster. Since there are 3 processors, the result would be three lists one with 3334 numbers in it and the other two with 3333. These sublists are then stored as the individual elements of a larger list. Next the `sfClusterApply` function takes that larger list and sends each processor one of its elements. Since it has three lists as elements and there are 3 processors, each processor gets one of the lists. Each processor then individually applies the function listed as the second parameter (in this case the `sum`) function to the element they received, and returns the result. In this case the result is the sum of all the numbers in the sublist. These results are stored as elements in a list. Finally the program goes through the list that was just created and finds the total of the sums of the elements in the three smaller lists, and that total is equal to the sum of all the numbers being added.

snow supports four different protocols for generating machine clusters and handling communications with the various processors within that cluster. First it can use raw sockets to form a direct connection to the machines containing the processors to use (Microsoft, 2011). Alternatively it can interface with one of three different types of third party programs and let those programs handle the work of managing the processors. The first program is the NetWorkSpaces(NWS) software package offered by Scientific Computing Associates, Inc (Scientific Computing Associates, 2009). The second option is to use Parallel virtual machine(PVM) package developed at Oak Ridge National Laboratory (Oak Ridge Natural Laboratory, 2011). The final option is to use one of the several different implementations of the Message Passing Interface(MPI) library standard. snowfall includes tools to help manage large clusters of machines that only work when using MPI to form a machine cluster, and as such that is the protocol that is used in this research (Knaus, 2010).

2.4.3 MPI

Message Passing Interface (MPI) is a library standard created in 1994. It is designed to handle communication and passing of data between multiple processors. It is one of the most popular standards for use in parallel programming (Quinn, 2003). There are a number of different implementations of MPI available for use. The implementation used in this research is LAM/MPI (Squyres and Lumsdaine, 2003).

Use of LAM/MPI requires the installation of the software on every computer that is going to be used as part of the machine cluster. One computer needs to be chosen to be the master node from which all the other nodes will be controlled. The master node is provided with a file containing the Internet Protocol (IP) addresses of all the other computers in the cluster as well as the number of processors available to use. Once the program in installed on all the computers, it is necessary to launch the runtime environment on all the computers in the cluster. This can be done using the `lamboot` command on the master node, which causes the node to form a secure shell (ssh) session with all the machines in the cluster which is used to launch the necessary program on each one, as

well as issue future commands (Squyres and Lumsdaine, 2003).

Once the environments are launched any program that utilizes the MPI Application Programmer's Interface (API) can utilize the machine cluster assuming it is run on one of the machines in the cluster. The API allows a programmer to tell the computer to run code on any given number of processors. LAM/MPI accepts that request, determines which processors should be given the workload, passes the request to each of the processors, and finally waits for results to be returned from the processors (Quinn, 2003). See Appendix B for help with setting up a machine cluster in Linux.

3 Methodology

This chapter details how trueTree works; from creating the initial dataset to calculating the final results and includes a number of pieces of code. For the benefit of readers who might not be familiar with the details of how R works, whenever there is code presented it will be followed by brief documentation for the relevant function calls, and an explanation of what the code actually does. Any function calls that appear in multiple code samples will only be explained the first time they appear.

3.1 Preparing The Input Data

trueTree requires its input to come from a file that has a specific format. Since trueTree was primarily written for use when text mining, the explanation given in this section on how to generate an input file will be specific to text mining. In text mining, the input file lists all the unique words contained within the chunks used, as well as the number of times each word appeared in each chunk. A tool called diviText (Jones, 2011) is available for use to automate the process of generating such a file. However it is possible to use trueTree in other areas of research as long as the necessary data for the research can be expressed in terms of a number of attributes with each chunk having a numerical value for all of the attributes. If those two conditions can be met it is possible to use trueTree by generating an input file that stores the attributes and their values in the same format as the input file described for the domain of text mining.

3.1.1 Initial Texts

The first step in generating an input file for trueTree is to select the texts that are going to be compared. It is possible to compare complete texts against each other, to divide one text into smaller pieces and compare those pieces against each other, or to do some combination of the above. Since the comparisons are based on the number of times different words appear in each text or chunk, all of the texts must be written in the same language to ensure the texts have enough words in common to make meaningful comparisons. Once the texts are selected, each individual text needs be saved as a separate text file, so they can be imported into diviText.

3.1.2 diviText

diviText is a tool developed by Amos Jones for the Lexomics Group at Wheaton College for automating the process of calculating the number of times each word appears in a text. This open source tool is available at `http://cs.wheatoncollege.edu/divitext/`.

Using diviText is a four step process. The first step is to upload the files containing each text using the upload/download button on the bottom left. Secondly each text needs to be divided into individual chunks. This process can be automated using the cutting tool window on the bottom right, or done manually be clicking on words in the center window to create new chunks starting from those words. Once the chunks have been made, they need to be saved using the save chunkset button on the bottom right. Finally all the chunksets need to be merged into a single file and downloaded using the merge chunksets button on the bottom left. For more information see (Jones, 2011).

3.1.3 Input Format

trueTree accepts as input a tab separated value (tsv) file. A tsv file stores a table, as show in Table 3.1. Each line in the file is one row of the table. Tabs mark the transitions between columns. So a line that goes A tab B has two columns, one column which contains A and one which contains B. The first row of the table contained in the file specifies the names of the attributes, which in the case of text mining would be the different words contained within the text. There is one attribute named in each column, except the first column in the table which specifies the names of the chunks with there being one chunk being named on each row row. The value in a cell is the number of times the word specified by the column the cell is in appears in the chunk specified by the row the cell is in.

	The	Cat	Dog
A	9	7	10
B	0	3	5
C	2	4	6

Table 3.1: A sample input file as it would appear when viewed in a program like Microsoft Excel^{TM}

\t**The**\t**Cat**\t**Dog**
A\t**9**\t**7**\t**10**
B\t**0**\t**3**\t**5**
C\t**2**\t**4**\t**6**

Listing 3.1 Actual contents of a sample input file as shown in table format in Table 3.1. \t represents a tab character

3.2 Initial Clustering

After the input is prepared trueTree can run the bootstrap on that dataset. However before it can do that it needs to read in the input file, and find an inital clustering that the ones created latter in the bootstrap can be compared to.

3.2.1 Reading in the Input Data

The code for reading in the input file is fairly simple and is shown in Listing 3.2

```
1 input.data <- read.table(file=as.character(input.file), header=T,
2      row.names=1, sep="\t")
3 tTable <- t(input.data)
```

Listing 3.2 R code to read in an input file.

`read.table(file, header, comment.char, row.names, sep, quote)`. R command to read in a file as input and store it as data.

file - Character vector that holds the name of the file to read

header - Logical Value that specifies whether or not the first line of the input file is a header specifying the names of the variables it contains

row.names - This parameter specifies the row names for the table. When set to a number; it specifies which column contains the row names.

sep - Character that specifies that the character separating each column in the input file

`t(x)`. R command that takes a table as input and returns it's transpose

x - The table to transpose

The first line of this code reads in the input file and stores it as a table, which the third line then transposes. Transposing the table is necessary because pvclust expects its input in a different format than diviText provides by default. pvclust wants the rows to specify the different attributes and the columns to specify the different chunks, which is the transpose of the format diviText gives output in by default.

3.2.2 Normalizing the Input Data

There is one more step required before the initial clustering algorithm can be run, and that step is to normalize the data. When the different chunks are compared to determine how similar they are to each other, the numerical values of all their attributes, which in this case is the word counts, are compared. The problem with such a comparison is that if two chunks have different total number of words, the word counts cannot be compared fairly. A word that appears five times in a six word chunk is more significant than a word that appears five times in a thousand word long chunk. This problem can be fixed normalizing the word counts by turning them into relative frequencies. The relative frequency of a word is the percentage of a chunk composed of that particular word. These frequencies can then be compared, since as percentages there meaning isn't dependent on the size of the chunk. The code to normalize the input data is shown in Listing 3.3

```
1  colSums <- apply(data, 2, sum) #each example/observation/object is
2                                 #one column, so find the sums of the
3                                 #columns
4
5  #compute matrix to divide current matrix by
6  #to normalize the word counts. Each entry in a column
7  #is the sum of the column
8  denoms <- matrix(rep(colSums, dim(data)[1]), byrow=T,
9      ncol=dim(data)[2])
10 relFreq <- data/denoms
```

Listing 3.3 R code to normalize the input data. data is a table containing the contents of the input file

apply(X, MARGIN, FUN). R command that applies a function to every entry in an array or matrix and returns a vector containing the results from each application of the function
X - An array or matrix to apply the function to
MARGIN - A vector that determines what subscript of X to apply FUN to. If X is a matrix and MARGIN = 1, FUN is applied to each row. IF MARGIN = 2 FUN is applied to each column
FUN - The function to apply

sum(...). R command that finds the sum of all the arguments that are passed
... - Any number of numeric vectors to add together

matrix(data, nrow, byRow, ncol). R command that generates a new matrix
data - An optional vector that contains the data that should be entered into the new matrix
nrow - Vector containing the number of rows in the new matrix. If no value is provided R will try and infer it from the size of Data
byrow - Logical value specifying which way the matrix is filled. If true it is filled by rows meaning that data is entered into the first row until it is fill then into the second and so forth. If false the matrix is filled by columns instead.
ncol - Vector containing the number of columns in the new matrix. If no value is provided R will try and infer it from the size of Data

dim(x). R command that retrieves the dimensions of a object. Returns a vector with the first number being the number of rows and the second number the number of columns
x - The object to retrieve the dimensions of

In Listing 3.3, line one finds the sums of all the word counts in each column. Since each column stores a single chunk due to taking the transpose of the input data in Listing 3.2, this results in finding the total number of words in each chunk. Line eight converts these sums into a table the same size of the original table. Each column in the table holds the total number of words in the chunk that column represents with that sum being repeated so it's present in every row. The result is that when the original table is divided by this newly created table in the line ten, which causes each number in the first table to be divided by the number in the same position in the second, every word count is divided by the total number of words in its chunk, which converts all the counts into relative frequencies.

3.2.3 hclust

Once the data has been converted into the relative frequencies the next step is to run a hierarchical clustering algorithm on that data, to generate a clustering to compare the results of the bootstrap to. R has a built in function, `hclust` that will run a hierarchical clustering algorithm on data. It operates very similarly to the method outlined in section 2.1.2, but offers support for a number of additional methods for determining the distance between a pair of chunks or clades. The code to run hclust is shown in Listing 3.4

```
1 distance <- dist.pvclust(relFreq, method= "euclidean")
2 data.hclust <- hclust(distance, method="average")
```

Listing 3.4. R code to compute the hierarchical clustering of a dataset. relFreq contains the table of normalized input generated in Listing 3.3 Suzuki and Shimodaira (2011)

`dist.pvclust(x, method)`. Function from the pvclust package (Suzuki and Shimodaira, 2011) that implements a custom verson of R's dist command, and creates a distance matrix that contains the distances between every pair of columns. This function supports several the "correlation", "uncentered", and "abscor" distance metrics while the original dist command does not.

x - Table to compute distances for.

method - Method to use when calculating distances between chunks. Supports "euclidean", "maximum", "manhattan", "canberra", "binary", "minkowski", "correlation", "uncentered", and "abscor" methods.

`hclust(d, method)`. R command that runs a hierarchical clustering algorithm on it's input.

d - distance matrix to run hierarchical clustering on.

method - Method to use when calculating distances between clades with more then one chunk. Supports "ward", "single", "complete", "average", "mcquitty", "median" or "centroid")

Line one of Listing 3.4 creates a distance matrix for the chunks stored in relFreq (See Listing 3.3). This matrix is then used by the hclust function in line two to find a hierarchical clustering for the dataset. The result is stored as an hclust object in the data.hclust variable. The hclust object has a number of components, the most important being the merge table.

The merge table stores the contents of the actual clades. The table has two columns and one row for each clade that is formed. The rows are sorted in the order that their respective clades formed. The first value in a row specifies the first chunk/clade that was used to form that row's clade, and the second value specifies the second chunk/clade that was joined with the first. These values can be positive or negative. A positive value refers to a previously formed clade, with the value being the row in the merge table in which the clade was formed. A negative value refers to a chunk, with the inverse of the value being the number of the chunk in question. For instance if the value in the first column is -3, then the 3rd chunk is being joined. An example merge table is show in Table 3.2

-2	-4
-1	1
-3	2

Table 3.2: Sample Merge table from an hclust object

The first row contains the values -2, and -4. These values are negative so they are specifying chunks. As such the first clade contains chunks number 2 and 4. The second row has the values -1 and 1. The -1 is negative meaning that chunk 1 is in clade 2. The 1 being positive means that the clade in row 1 is in clade 2. So that second clade joins chunk 1 and the clade already containing chunks 2 and 4. Similarly the final row contains the values -3 and 2, meaning that it joins chunk 3 and the second clade together to form the final clade. The resulting dendrogram is shown in Figure 3.1

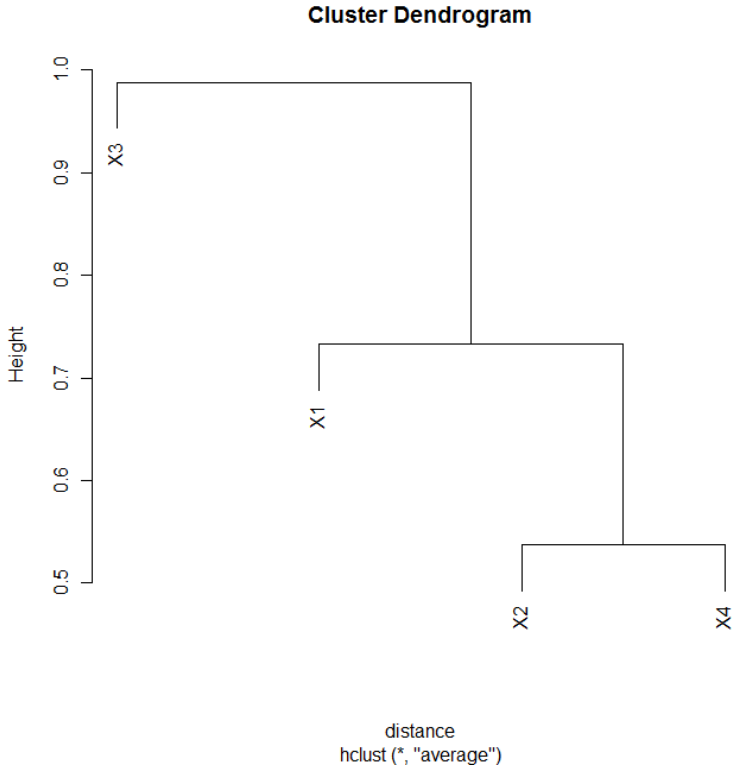

Figure 3.1: Dendrogram represented in Table 3.2

3.3 Bootstrap Resampling

Once the relative frequencies are calculated for the dataset and the initial hierarchical clustering is run the bootstrap can begin. Bootstrapping is a two step process. First the word counts are resampled with replacement in order to create a new data set, then a new hierarchical clustering is generated which is compared to the original. These two steps are then repeated thousands of times to produce the final results.

The resampling method used by trueTree is one of it's major divergences with the pvclust R code it is based on. Pvclust was originally designed for certain types of genetic research, and its resampling method reflects this. It can be described as interrow resampling since it resamples by picking rows of data to place in the new dataset (Suzuki and Shimodaira, 2011). If this method was applied to text mining the process would generate new datasets by picking from the different unique

words present in the dataset and carrying over the number of times each word appeared in each chunk from the original dataset. This method proved ill-suited for text mining since different words have different levels of importance towards determining how chunks should cluster. For example a word that only appears once across all the chunks is insignificant, while a word that appears dozens of times in some chunks and only a few times in others can be a major factor in a clustering. Interrow based resampling introduces the possibility of ending up with too many words from one of those two groups, either eliminating all the differences, or greatly magnifying them, either way generating a dataset that has been distorted more then desired when running the bootstrap. As such trueTree implements it's own resampling function.

trueTree includes two different resampling methods, intra-chunk resampling, and intra-clade resampling both of which are designed primarily for use in the field of text mining. Both resample based on word counts instead of unique words. The difference is that intra-chunk resampling looks at the word counts within each chunk, while intra-clade resampling looks at the word counts within clades comprised of multiple chunks. Conceptually both methods utilize a list of all the words in a chunk or clade, with each word appearing the same number of times in the list that it appeared in the chunk or clade, and resampling by randomly choosing words from that list. These two methods are explained more fully in the next sections.

3.3.1 Intra-chunk Resampling

For performance reasons the actual resampling is not done by generating a long list of words but by using a multinomial distribution. A multinomial distribution takes a list of possible outcomes, in this case the different words in a chunk, and a list of the probability that each outcome would be picked when an outcome is chosen at random, and then uses those probabilities to pick an outcome. The probability used for each is the probability that if a word was selected from the previously mentioned list at random it would prove to be that word. As such the probabilities are the percentage of the list composed of each word which is equivalent to the relative frequencies calculated in Listing 3.3. This process is done once for each chunk with the number of words picked using the multinomial distribution for each chunk equaling the number of words originally in the chunk. The code to run the multinomial distribution is shown in Listing 3.5

```
bootStrapData <- matrix(data = 0, nrow = nrow(data),
    ncol = ncol(data), dimnames=list(rownames(data),
    colnames(data))) #set up empty table to hold word counts
                     #for bootstrapped data

for(j in 1:ncol(data)){ #for each chunk
    size <- chunkSize[[j]]     #calculate the number of words in
                               #the new bootstrapped chunk based
                               #on the original size
    counts <- rmultinom(1, size, relFreq[,j])
    bootStrapData[,j] <- counts #store the word counts
}

#normalize the new data
colSums <- apply(bootStrapData, 2, sum)
denoms <- matrix(rep(colSums, dim(bootStrapData)[1]), byrow=T,
    ncol=dim(bootStrapData)[2])
bootStrapData <- bootStrapData/denoms
```

Listing 3.5. R code to run intra-chunk resampling with replacement using a multinomial distribution in order to create a resampled dataset. The object named data contains the original data table read in Listing 3.2, chunkSize is a vector containing the number of words in each chunk, and relFreq contains the table of normalized input generated in Listing 3.3

rmultinom(n, size, prob). R function that generates a multinomial distribution. Returns a vector(s) showing the number of times each object was picked.

n - Number of vectors of length size to generate

size - Number of objects to pick for each vector

prob - Vector containing the probability that each object is selected

Line 1 creates a matrix the same size as the original data set to hold the new resampled data set. Each column in this matrix stores the word counts for a chunk, while each row stores the number of times a unique word appeared in each chunk. Line 6 is a loop that runs once for each chunk in the dataset. The index variable j is used to store the chunk currently of interest. Line 7 looks up the number of words in that chunk. That number is used in line 10 as part of the resampling using the multinomial distribution. The output of the multinomial function is a vector that contains the number of times each word appears in the chunk in the resampled dataset. Line 11 stores the counts in the column of the new dataset corresponding to the chunk in question. Finally lines 15-18 normalize the new dataset, since the resampling computes raw word counts instead of relative frequencies. These lines work identically to the code in Listing 3.3.

3.3.2 Intra-clade Resampling

Intra-clade based resampling works similar to intra-chunk based resampling, except that instead of basing the resampling on the number of times a word appears in a chunk, it bases the resampling on the number of times a word appears in a clade across multiple chunks. Compared to intra-chunk resampling, intra-clade resampling requires additional preparation to use since it is necessary to determine the number of times each word appears in each clade used in the resampling. The clades used are defined by the user and while in normal usage they will be based on the clades in the initial clustering, it is possible to define clades that do not exist in the original clustering since this provides the maximum flexibility in how trueTree is used. The clades are defined in a vector called cladeChunkIn that has the same number of elements as there are chunks. Each element is an integer that gives the number of the clade the chunk corresponding to that elements index is in. For example if the vector defines the clades as:

`1,2,2,3,1`

it would mean that there are 3 clades being used in the resampling, the first of which contains chunks 1 and 5, the second of which contains chunks 2 and 3, and the last of which contains just chunk 4.

3.3.2.1 Preparations The first step towards running Intra-clade resampling is to create a matrix showing the relative frequencies for each unique word inside each clade. The code to do so is shown in Listing 3.6

```
1  n <- nrow(data); p <- ncol(data)
2  cladeCounts <- matrix(rep(0, max(cladeChunkIn * n)),
3      ncol = max(cladeChunkIn), nrow = n)
4  for(i in 1:p) #for each chunk
5  {
6          curClade <- cladeChunkIn[i] #get which clade the
7                                      #chunk is in
8          #add counts to total
9          cladeCounts[,curClade] <- cladeCounts[,curClade] + data[,i]
10 }
11
12 #convert to relative frequency
13 colSums <- apply(cladeCounts, 2, sum) #each clade is one column, so
14                                        #find the sums of the columns
15 denoms <- matrix(rep(colSums, dim(cladeCounts)[1]), byrow=T,
```

```
16      ncol=dim(cladeCounts)[2]) #compute matrix to divide current
17                                #matrix by to normalize matrix. Each
18                                #entry in a column is the sum of
19                                #the column
20  cladeRelFreq <- cladeCounts/denoms
```

Listing 3.6 R code to determine the relative frequencies of words inside a set of user defined clades. The vector cladeChunkIn contains the clade each chunk is in. The table data contains all the word counts for all the chunks that was generated in Listing 3.2.

`max(...)`. R command that finds the max of all the arguments that are passed
... - Works on any number of numeric vectors

Line 1 gets the number of rows in the data table, which is also the number of unique words in the dataset, and the number of columns, which is also the number of chunks in the dataset. Lines 2 and 3 create a matrix to store the word counts for each clade. The matrix has one row for each unique word, and one column for each clade being used in the resampling. All the values in the matrix are set to 0, because at this point no instances of any of the words have been seen. The code makes the assumption that there will be no gaps in the range of numbers assigned to clades by the user. For instance if the user puts some chunks in clade 4, then there must be clades 3, 2, and 1. As such the highest clade number is also the total number of different clades being used. However if this assumption proves false, the code will still work, but there will be a few empty columns in cladeCounts that never get used.

The loop starting at line 4 runs once for each chunk in the dataset. As such for each chunk line 6 checks what clade the chunk is part of, then line 9 adds it's word counts to the total wordcounts for that clade. Once this has been done for all the clades, lines 15-20 normalize the word counts into relative frequencies. These lines work identically to the code in Listing 3.3.

3.3.2.2 Resampling Once the relative frequencies have been calculated for the different clades the actual resampling works similarly to the method used for intra-chunk resampling. The code to run the intra-clade resampling is shown in Listing 3.7

```
1   bootStrapData <- matrix(data = 0, nrow = nrow(data),
2       ncol = ncol(data), dimnames=list(rownames(data),
3       colnames(data))) #set up empty table to hold word counts
4                        #for bootstrapped data
5
6   for(j in 1:ncol(data)){ #for each chunk
7       size <- chunkSize[[j]]     #calculate the number of words in
8                                  #the new bootstrapped chunk based
9                                  #on the original size
10      chunklistIndex <- cladeChunkIn[j] #get clade the chunk is in
11                                        #to know what sublist of
12                                        #the wordlist to use
13      counts <- rmultinom(1, size, cladeRelFreq[,chunklistIndex])
14      bootStrapData[,j] <- counts #store the word counts
15  }
16
17  #normalize the new data
18  colSums <- apply(bootStrapData, 2, sum)
19  denoms <- matrix(rep(colSums, dim(bootStrapData)[1]), byrow=T,
20      ncol=dim(bootStrapData)[2])
21  bootStrapData <- bootStrapData/denoms
```

Listing 3.7. R code to run intra-clade resampling with replacement using a multinomial distribution in order to create a resampled dataset. cladeChunkIn is a vector containing the clade number where

each chunk resides. data contains the original data table, chunkSize is a vector containing the number of words in each chunk, and cladeRelFreq contains the table of normalized input generated in Listing 3.6

This code functions almost identically to the code in Listing 3.5 except for lines 10 and 13. Line 10 determines the clade where the chunk resides. Line 13 uses the results of line 10 to get the proper column of cladeRelFreq to use to resample. It is important to note that while the relative frequencies used may be the frequencies for the entire clade, the size of the resampled chunk is still based on the chunk's original size, not the size of the entire clade.

trueTree actually uses the code in Listing 3.7 when it runs intra-chunk resampling to reduce the amount of duplicated code. It is able to do so by generating a `cladeChunkIn` vector that places every chunk in a different clade. As such the code resamples using a set of clades that each contain a single chunk, which is equivalent to resampling based on individual chunks.

3.3.3 Comparing the new samples to the original data

Once a new data set has been generated through resampling, the next step is to run hierarchical clustering on the new dataset and compare the results to the clustering generated for the original dataset. The comparison being made tests whether the same clades exist in both clusterings. A clade can be said to exist in both, if each clustering has a clade containing the exact same set of chunks. The order in which the clades are generated in each clustering does not matter, so if a particular clade was the first clade generated in one clustering, and the second clade generated in the other, they still count as being in the same clade.

In R, hierarchical clusterings are stored as hclust objects. The merge table that stores the contents of the clades is difficult to work with, so the first step towards being able to compare clusterings is to convert the data stored in the table into a more convenient form. pvclust provides a function `hc2split` that converts an hclust object into a vector of binary strings, with each string containing the contents of one clade. Each string has one digit corresponding to each chunk in the dataset. That digit is set to 1 if the chunk is in the clade, and 0 otherwise. To give an example figure 2.2 shows a simple dendrogram.

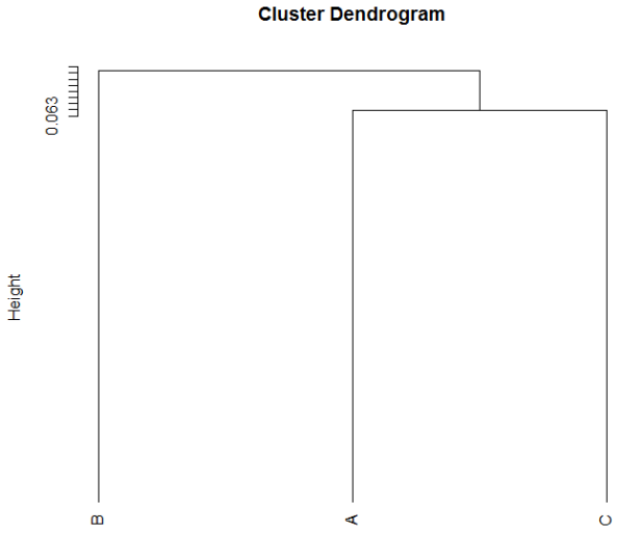

Figure 2.2: Trival Dendrogram intended to show the results of the `hc2split` function. A is the first chunk in the dataset, B is the second, C is the third

This dendrogram has two clades so if `hc2split` was run on it the function would return 2

binary strings. The first clade contains chunks A and C, so the string representing that clade would be `101`. The second clade contains all three chunks so it's string would be `111`

In addition to converting the clusterings into these binary strings, a table needs to be constructed that can store the results of multiple comparisons since during the bootstrap thousands of new datasets will be generated. This table will need to hold the number of times each clade is created during the bootstrap. As such during the initial setup for bootstrapping, the clustering for the original dataset needs to be run through `hc2split` and the results of that function need to be used to set up a suitable table. The code to do so is shown in Listing 3.8

```
1 pattern   <- hc2split(data.hclust)$pattern #get the pattern (the
2                                             #contents of each clade)
3                                             #from the original
4                                             #hclust object
5
6 #create a table that shows the number of times each clade is
7 #formed in the bootstraped clusters
8 edges.cnt <- table(factor(pattern)) - table(factor(pattern))
```

Listing 3.8 R code to generate a table to store the results of comparisons of clusterings. The hclust object named data.hclust is the object created from the original dataset created in Listing 3.4 (Suzuki and Shimodaira, 2011)

`hc2split(x)`. pvclust function that converts an hclust object into a vector of binary strings that document each clade's contents.

x - hclust object to convert

`factor(x)`. R function that encodes a vector as a factor. A factor is a type of vector that also contains a discrete list of all the possible values that can be in the vector, which are called levels.

x - vector to convert into a factor

levels - optional vector containing the levels for the factor being created. Any value in x that is not in levels will be replaced with NA meaning not available

`table(...)`. R function that builds a table out of factors. table takes each of the levels in the factor and makes them columns in a table, and then creates a row of data containing the number of times each level appears in the vector portion of the factor. Ignores any elements of the factor which equal NA

... - factors to convert into table

Line 1 takes the original clustering and converts it into a vector of binary strings containing one string for each clade in the clustering. Line 8 builds a table out of this data that has a single row and one column for each clade. The clades are gotten by the `factor(pattern)` call that turns the vector constructed on line 1 into a factor. This adds a list of the values that are possible in the vector by listing all the unique elements in the vector. Since no two clades in a clustering can have the exact same contents every string in pattern is unique so the list of possible values is identical to the contents of pattern. `table(factor(pattern))` turns the factor into the previously mentioned table. Each clade is given a count of 1 since they appeared once in the original clustering, and as such are contained in pattern. As such it is necessary to create the same table a second time and subtract it's contents from the first to set the counts to zero, because the original clustering doesn't count towards how many times each clade shows up in the resampled datasets.

Once the `edges.cnt` table is created, it is then possible to run comparisons between the original clusterings and those generated using resampled datasets. The code to do so is shown in Listing 3.9.

```
1 #create the new distance matrix
2 distance  <- dist.pvclust(bootStrapData,method="euclidean")
```

```
3
4  #create the hclust object
5  x.hclust  <- hclust(distance,method="average")
6
7  #get the contents of the clades
8  pattern.i <- hc2split(x.hclust)$pattern
9
10 #add the clades identical to the clades in the original object to the
11 #count of the number of times those clades appeared
12 edges.cnt <- edges.cnt + table(factor(pattern.i,  levels=pattern))
```

Listing 3.9 R code to run hierarchical clustering on a resampled dataset and compare the results to the original clustering. bootStrapData contains the relative frequencies for the new dataset computed in Listing 3.7. edges.cnt is a table storing the number of times each clade in the original clustering has appeared and was generated in Listing 3.8 pattern is a vector of binary strings representing each clade in the original hierarchical clustering generated in Listing 3.8 (Suzuki and Shimodaira, 2011)

Lines 2 and 5 take the resampled dataset and generates a hierarchical clustering from it just like in Listing 3.4. Line 8 then turns the clustering into a vector of binary strings. Line 12 runs the comparison against the original clustering and updates the table containing number of times each clade has appeared at that point during the bootstrap process. It does so by using the `factor(pattern.i, levels=pattern)` command which converts the binary strings generated from the new clustering into a factor that only contains the strings for clades that were in the original clustering, with all other clades being lost. As such when the factor is converted into a table it will have one column for each clade present in the original clustering each of which will have a value of 1 due to appearing once, and when that table is added to `edges.cnt` it updates the counts of clades present in both the old and new clusterings.

In addition to comparing the clustering of the resampled dataset to that of the original, a cophenetic correlation coefficient needs to be calculated. As mentioned in Chapter 2 the cophenetic correlation coefficient measures how similar the cophenetic distance matrix created from a clustering is to the original distance matrix used to make it. trueTree compares the cophenetic distance matrix created from the clustering of the resampled dataset with the distance matrix of the original dataset. This is because the point of running the bootstrap is to see if the results are stable even when making minor changes to the dataset by resampling it, under the assumption that stable results are more likely to be "real". As such the cophenetic distance matrix needs to be compared to the original distance matrix, because the result of that comparison, shows how much the clustering has changed from the original data.

The code for calculating the Cophenetic Correlation Coefficient is shown in Listing 3.10

```
1  bootCop <- cophenetic(x.hclust) #get the cophenetic distance matrix
2  stc[[i]] <- cor(copDistance, bootCop) #find the coefficient
```

Listing 3.10. Code for calculating the Cophenetic Correlation Coefficient. x.hclust is the hclust object created in Listing 3.9. copDistance is the distance matrix from the original dataset. i is the number of resampled datasets that have been generated so far

`cophenetic(x)`. Finds the cophenetic distance matrix of an hclust object
x - hclust object to find the distance matrix of

`cor(x, y)`. R function that finds the pearson correlation between two objects
x - A numerical vector or matrix
y - object to find the correlation of x to

The first line creates the cophenetic distance matrix. The second line finds the correlation between that matrix and the original distance matrix, and stores that number as one element of a list that will end up holding all the coefficients for the resampled datasets.

3.3.4 Multiscale Bootstrap Resampling

The methods shown in the previous sections are for running the bootstrap that can be used to calculate BP values. Calculating AU values requires a multiscale bootstrap to be run. In a multiscale bootstrap each time a resampled dataset is created, an r value is provided and the size of the resampled dataset is set to the original size of the dataset times the r value. For instance if a r value of 1.5 is provided, the resampled dataset will be 1.5 times larger then the original dataset being resampled. Integrating the use of r values requires only changing one line in the resampling code. Line 7 in Listing 3.7 is `size <- chunkSize[[j]]`. Changing that line to `size <- chunkSize[[j]] * r` where r is the r value being used for that particular resampled dataset, will ensure that the new dataset is of the proper size.

3.3.5 Combining the Steps

The preceding sections have shown all the steps necessary to generate and compare a single resampled data set. However running the bootstrap requires thousands of resampled datasets to be generated in order to get reliable results. trueTree has a parameter `nboot` that specifies how many resampled data sets should be generated. The code in Listings 3.7 and 3.9 are put inside a loop that runs `nboot` times with the loop itself being placed in a function called `boot.pvclust` which runs the code Listing 3.8, then goes through the loop before finally returning the `edges.cnt` table from Listing 3.9 when run.

trueTree also has another parameter `r` that contains all the r values to use for the multiscale bootstrapping. By default it contains the values `.5,.6,.7,.8,.9,1,1.1,1.2,1.3,1.4`. There are nboot resampled datasets generated for each r value used, so with the default of 10 r values, a total of $10 * nboot$ resampled datasets are generated whenever trueTree is run. The code to run the bootstrap is shown in Listing 3.11.

```
1 mboot <- lapply(r, boot.hclust, data=data, object.hclust=data.hclust,
2     nboot=nboot, cladeChunkIn=cladeChunkIn, chunkSize=chunkSize,
3     relFreq = relFreq) #do the actual bootstrapping
```

Listing 3.11 R code to run multiscale bootstrap resampling. r is a list of r values to use Suzuki and Shimodaira (2011)

`lapply(X, FUN, ....)`. Variant of R's apply command that applies a function to every item in an list and returns a list containing the results for each application of the function

X - A list to apply the function to

FUN - The function to apply

.... - List of parameters to use when running FUN

`boot.pvclust(r, data, object.hclust, nboot, cladeChunkIn, chunkSize, relFreq)`. pvclust function (Suzuki and Shimodaira, 2011) that runs bootstrap resampling on the data in data nboot times and returns a table with the results

r - r value to use when generating resampled datasets

nboot - number of resampled datasets to generated

.... - all other parameters contain the contents of the identically named variables seen in earlier Listings.

This code runs the bootstrap once for each r value due to the lapply function being called on a list of r values. The final result is a list of tables, one table per r value. Each table has one column for each clade in the clustering of the original dataset, and each column contains the total number of times that clade showed up in the resampled datasets generated using the r value corresponding to that table. As an example if a dataset was used that has four chunks and that dataset was resampled 10 times for each r value the table the first table in the list would look something like Table 3.3

1001	0110	1111
9	7	4

Table 3.3: Example output from running the bootstrap on a dataset containing 4 chunks

The column names are binary strings encoding the contents of the clade as was discussed in Section 3.3.3. Table 3.3 shows that the first clade containing the first and fourth chunks was created nine times, the second clade seven times, and so forth. For each additional r value used, another table will be created that is identical in format to the one in Table 3.3, but has different values.

3.4 Analyzing the Results

Once the bootstrap is complete the next step is to analyze the results in order to calculate the AU, BP, and Cophenetic Correlation Coefficient values that give the results of the cluster validation. Calculating the BP values is required in order to calculate the AU values, but the Cophenetic Correlation Coefficients are calculated independently of those two. The code for calculating the BP and AU values are taken from the pvclust package for R Suzuki and Shimodaira (2011)

3.4.1 Calculating BP values

Calculating the BP values is simple in concept. The BP value for any given clade, is simply the percentage of times that clade appears in the clusterings from the resampled datasets. A different BP value is generated for each r value that was used in running the bootstrap, so if there were 10 r values used, each clade will have 10 BP values associated with it.

The code for calculating the BP values is show in Listing 3.12

```
edges.bp <- edges.cnt <- data.frame(matrix(rep(0,ne*rl),
        nrow=ne,ncol=rl)) #Creates two big tables, with each
                          #row being a clade, and each column
                          #being an r value. An individual entry
                          #denotes either the number of times
                          #that clade occured or the bp value
                          #for that clade for a particular r value.
row.names(edges.bp) <- pattern
names(edges.cnt) <- paste("r", 1:rl, sep="")

for(j in 1:rl) { #for each r value
    #find the number of times each clade is formed for this
    #particular r value
        edges.cnt[,j] <- as.vector(mboot[[j]]$edges.cnt)
    #find the bp value for each clade for this particular r value
        edges.bp[,j]  <- edges.cnt[,j] / nboot[j]
}
```

Listing 3.12 Code for calculating the BP values for each clade for each r value. ne is the number of clades, rl is the number of r values, nboot is a list containg the number of resampled datasets generated for each r value, pattern is a vector containing each clade in the original clustering as a binary string as seen in Listing 3.8 and mboot is the list containing the results of the bootstrap generated in Listing 3.11 (Suzuki and Shimodaira, 2011).

`data.frame(....)`. R Function that converts the objects passed into a data frame. A data frame is a more complicated form of a matrix, which supports having columns that store differing types of data.

.... - Objects to convert into a data frame

paste(...., sep). R Function that converts it's arguments into character vectors then concatenates those vectors

.... Objects to convert into character vectors

sep Character string that will separate each object in the final output.

as.vector(x). R Function that converts the object passed into a vector

x - Object to convert into a vector

The first line of the code in Listing 3.12 creates two data tables, one of which will hold the number of times each clade was created for each r value, while the other will hold the BP values calculated. In both tables each column represents an r value and each row represents a clade. Line 8 sets the names of the rows in the table holding the BP values to the binary strings for each clade. Line 9 gives the columns in the table holding the counts names representing what the r value the column represents. The first column representing the first r value is named r1, the second r2 and so forth. Finally the code from 11 to 17 calculate the actual BP values. The loop runs once per r value used. Inside the loop, line 14 gets the number of times each clade appeared in the resampled datasets out of that data returned from the boostrapping, and adds the counts to the appropriate column in the table. The numbers in that column are then divided on line 16 by the number of resampled datasets generated to get the percentage of datasets that contained those clades, and those results are stored in the appropriate column of the second table. Once all the BP values are calculated they can then be used to calculate the AU values.

3.4.2 Calculating AU values

Calculating the AU values is a three step process. The general idea behind the calculations is, as was mentioned in Chapter 2, to plot the BP values calculated for each r value onto a graph, and find a curve that fits them. The first step in this process is to plot the BP values which require several conversions to get the numbers into the right form. In the graph the Y axis measures $\tilde{Z}$ and the X axis measures $\frac{1}{\tau}$ (Shimodaira, 2004).

τ is derived from the r value used to calculate a BP value and as such is a measure of scale. τ equals $\sqrt{\frac{n}{n_1}}$ where n is the size of the original dataset, and n_1 is the size of the resampled dataset. Only the relative proportions matter here, so n can be set to 1, and n_1 can be set to the r value since those two numbers have the same relative proportions as the sizes of the original and resampled datasets. Using that definition of τ, $\frac{1}{\tau}$ is equal to $\sqrt{\frac{r}{1}}$ (Shimodaira, 2004).

$\tilde{Z}$ is derived from the BP value, and is equal to $\Phi^{-1}(\tilde{\alpha}(y, \tau_1))$. $\tilde{\alpha}(y, \tau_1)$ is equal to the percentage of times an observation y falls into a specific region for the given τ_1. The y in this case is the specific clade the au value is being calculated for. This means $\tilde{\alpha}(y, \tau_1)$ is equal to the percentage of times a certain clade is formed for a specific r value, which is in effect the BP value of that clade for that r value. $\Phi^{-1}(x)$ is an operation that finds at what point on the normal curve there is x percent of the area under the curve lying to the right of that point. (Shimodaira, 2004)

Given these definitions, plotting a BP value requires taking the r value and using it to calculate $\frac{1}{\tau}$ while the BP value itself is used to calculate $\tilde{Z}$ with the results of these two calculations being used to determine where the value falls on the graph. (Shimodaira, 2004)

The second step is to find a regression curve that fits the points plotted. Specifically the equation for the curve should be in the form of $\tilde{Z}(y, \tau) \approx \frac{\hat{v}}{\tau_1} + \hat{c}\tau_1$. $\hat{v}$ and $\hat{c}$ are the regression coefficients that need to be calculated to find the curve. The curve itself is found by running a least squares fit (Shimodaira, 2004)

The final step is to convert the coefficients into an AU value. The formula for the AU value is $\Phi(\hat{v} - \hat{c})$. $\Phi(x)$ is an operator that finds where x is on the normal curve and finds the percentage of the area under the curve that is to the left of that point. For more details about how the calculations are preformed and the rational for why they produce the desired results see (Shimodaira, 2004).

The code for calculating the AU values is shown in Listing 3.13. The code is called once for every clade in the dendrogram of the original clustering.

```
zz <- -qnorm(bp) #find where the bp values lie phi inverse

vv <- ((1 - bp) * bp) / (dnorm(zz)^2 * nboot)

X   <- cbind(sqrt(r), 1/sqrt(r)); dimnames(X)
    <- list(NULL, c("v","c"))
fit <- lsfit(X, zz, 1/vv, intercept=FALSE) #fit the curve
a$coef <- coef <- fit$coef #get the coefficients

h.au <- c(1, -1)

z.au <- drop(h.au %*% coef); #%*% is matrix multiplication
                             #au is v - c

au <- pnorm(-z.au); #phi
```

Listing 3.13. Code for calculating AU values. bp is a list of the bp values for each r value for a given clade. r is a list of the r values, and nboot is a list of the nboot values used (Suzuki and Shimodaira, 2011).

The first line performs the Φ^{-1} operation on all the BP values. The fifth line calculates τ and $\frac{1}{\tau}$ for each r value, and puts them together into a matrix. It needs both because the equation for the curve $\tilde{Z}(y,\tau) \approx \frac{\hat{v}}{\tau_1} + \hat{c}\tau_1$ requires both values. The seventh line performs the actual least squares curve fitting. The coefficients found for $\hat{v}$ and $\hat{c}$ are then taken from the results on the next line. Line twelve finds the value of $\hat{v} - \hat{c}$ by using matrix multiplication. The matrix created on line ten is $[1, -1]$ so when the coefficients are multiplied by that matrix the result is the sum of the first coefficient minus the second coefficient. Finally line fifteen performs the Φ operator on the result of $\hat{v} - \hat{c}$ giving the au value.

3.4.3 Analyzing the Cophenetic Correlation Coefficients

After running the bootstrap, the cophenetic correlation coefficients for all the resampled datasets will be stored in a list. Given that there may be thousands of coefficients, the list must be augmented to provide useful information. As such trueTree runs a number of basic tests on the coefficients to provide some basic statistics on how the coefficients are distributed. trueTree determines the minimum, mean, median, and maximum values in the set, as well as the standard deviation. In addition trueTree can be provided with a confidence interval and it will determine the values that lie at the upper and lower bounds of that interval. Finally trueTree will plot a histogram showing the distribution of values, and add a line plotting where the cophenetic correlation coefficient for the original dataset lies within that plot. The code to perform the calculations is shown below in Listing 3.14.

```
print(paste("Original Cophenetic correlation", originalCor, sep=" "))
print(paste("Number of Cophenetic correlation values",
    length(copValues), sep=" "))
print(paste("Minimum Cophenetic correlation",
    min(copValues), sep=" "))
print(paste(lowerBound * 100, "% interval Cophenetic correlation ",
    copValues[round(copSize * lowerBound)], sep=""))
print(paste("Median Cophenetic correlation",
    median(copValues), sep=" "))
print(paste(upperBound * 100, "% interval Cophenetic correlation ",
     copValues[round(copSize * upperBound)], sep=""))
print(paste("Maximum Cophenetic correlation",
```

```
13      max(copValues), sep=" "))
14  print(paste("Mean Cophenetic correlation", mean(copValues), sep=" "))
15  print(paste("Standard Deviation Cophenetic correlation",
16      sd(copValues), sep=" "))
17
18  hist(copValues)
19  abline(v = originalCor, col = "red") #add a line showing where
20                                       #original coefficient lies
```

Listing 3.14. R code for analyzing the results of finding the cophenetic correlation coefficients during the bootstrap process. orginalCor is the cophenetic correlation coefficient for the dendrogram created from the original dataset. copValues is a list containing all the cophenetic correlation coefficients found during bootstrapping. lowerBound and upperBound are the percents as decimals of the lower and upper bound of the desired confidence interval. copSize is the total number of coefficients contained in copValues.

`print(x)`. R command that prints the argument passed to it
x - Argument to print

`min(...)`. R command that finds the min of all the arguments that are passed
... - Works on any number of numeric vectors

`round(x)`. R command that rounds a number to the nearest integer
x - numeric vector containing number to round

`median(x)`. R command that finds the median of the vector passed to it
x - vector to find the median of

`mean(x)`. R command that finds the mean of the vector passed to it
x - vector to find the mean of

`sd(x)`. R command that finds the standard deviation of the vector passed to it
x - vector to find the standard deviation of

`hist(x)`. R command that plots a histogram from data in a list
x - List to plot histogram from.

`abline(a, col)`. R command that draws a vertical line on a plot
x - position along x axis to draw line
col - color of the line to draw.

The code in Listing 3.14 is simple. The majority of the lines of code run a single function on the list of cophenetic correlation coefficients, joins the results together with a string to label what the result is and then prints out the result for the user to see. Lines 6 and 7 which find the lower end of the confidence interval multiples the percent where the confidence interval lies with the total number of coefficients to get a usable index. For example if the lower end of the confidence interval is 2.5% and there are 100 coefficents, it will find that the confidence interval starts at the 2.5th coefficient which will get rounded up into the 3rd coefficient. The code on lines 10 and 11 for finding the upper bound works similarly.

3.5 Differences When Running the Code in Parallel

Running trueTree in parallel instead of on a single processor requires only a few changes to the code. The only part of trueTree that is actually suited to be run in parallel is the bootstrap process. All

the steps done during the initial setup and final analysis are either inherently sequential making that step impossible to run in parallel, or can be calculated so quickly that the extra overhead from running code in parallel would eclipse any savings in runtime gained by executing the code in parallel. The bootstrapping in comparison is not inherently sequential since the thousands of resampled datasets that need to be generated can be generated in any order, and each one takes a significant length of time to run meaning the reduction in time that can be achieved by running the code in parallel vastly outweighs the extra overhead.

The snow package (Tierney *et al.*, 2011) is used to handle the actual task of running the code in parallel. The main method for distributing work across processors in the snow package is to use different variants of R's apply functions that instead of just applying a function to each element in a list or other datastructure, farms out the work to all the processors in a cluster, so each processor is applying the function to a different element in the list. The contents of this list are the only unique input given to each processor, and as such needs to contain all the information necessary to tell each processor it's specific task. The process of running the bootstrap is the same for every processor, so in trueTree this list is used to perform load balancing. The list contains an integer for each processor, with the integer representing the number of resampled datasets to generate on that processor for each r value, and that number is generated by dividing the number of resampled datasets specified by the nboot value across the processors. The code to do so is shown in Listing 3.15.

```
1  #Divide nboot up evenly across the processers in the cluster
2  nbl <- as.list(rep(nboot %/% ncl,times=ncl)) # %/% is integer
3                                               #division.
4
5  if((rem <- nboot %% ncl) > 0) #if there are some nboots remaining
6          nbl[1:rem] <- (nbl[1:rem], "+", 1) #add 1 nboot to each
7                                             #cluster upto the number
8                                             #of remaining bootstraps
```

Listing 3.15 R code to divide the workload of generating resampled datasets across processors. nboot is the number of datasets to generate in total. ncl is the number of processors in the machine cluster being used (Suzuki and Shimodaira, 2011).

Line 2 divides the number of datasets to generate by the number of processors being used to get a baseline for the number of datasets each processor should generate. As such the initial list containing the load balancing uses that number as the initial amount for each processor. Integer division is used here because it is impossible to meaningfully generate a fraction of a dataset. Line 5 checks if there was a remainder when performing the earlier division. If so it means that there are a few more datasets that need to be generated then are currently accounted for. If there is a remainder it is stored in rem and line 6 takes the first rem numbers in the list and increases each by 1. This makes the number of datasets that will be generated equal to the value of nboot.

Once the load balance is determined the bootstrap can be started. The code that actually executes the bootstrap is shown in Listing 3.16

```
1  pvclust.node <- function(x, r,...) #this does the bootstrap for
2                                     #a single processor
3  {
4      #do the bootstrap once for each r value
5      mboot.node <- lapply(r, boot.hclust, nboot=x, ...)
6      return(mboot.node)
7  }
8
9  mlist <- parLapply(cl, nbl, pvclust.node, r=r, data=data,
10                    object.hclust=data.hclust,  nboot=nboot,
```

```
11                      cladeChunkIn=cladeChunkIn, chunkSize=chunkSize,
12                       relFreq = relFreq) #do the bootstrap
```

Listing 3.16 R code to run the bootstrap in parallel using the snow package. cl is a snow object representing a machine cluster to use. nbl is the list generated in Listing 3.15 (Suzuki and Shimodaira, 2011)

parLapply(X, FUN, ...). Variant of R's lapply command provided by the snow package. It applies a function to every item in a list, with the function being applied to each item on a different processor meaning that all the applications are run simultaneously. Returns a list containing the results for each application of the function

cl - machine cluster to run function on

X - A list to apply the function to

FUN - The function to apply

.... - List of parameters to use when running FUN

Line 9 runs the `pvclust.node` function on each processor. This function does the exact same thing as shown in Listing 3.11 with the only difference in execution being that instead of generating nboot resampled datasets it generates the number determined in Listing 3.15. This number is passed in as the x variable by the parLapply function.

The only other difference in running the code in parallel is that once the parLapply function on line 9 is finished, the results from all the processor have to be merged into a single result. Since the data passed back by each processor is a table showing the number of times each clade was formed in the resampled datasets generated on that processor, these tables can be merged by adding the counts for each processor together into one final table.

3.6 trueTree API

There are three functions available when running trueTree. The first is named `trueTree` and it runs the main bootstrapping. The second, `varianceTest`, is intended to help determine the best set of parameters to use. The third is a custom plot command to allow the usage of trueTree's graph coloring code. This section contains the details on calling these functions. Information on how to apply them to research is given in next chapter.

trueTree(input.file, outputFilename = NULL, main = NULL, textlabs = NULL , chunksize = NULL , labelFileName = NULL, distMetric = "euclidean", clustMethod = "average", input.transposed = TRUE, nboot = 100, runParallel = FALSE, numCPUs = 2, clusterType = 'SOCK', confidenceInterval = .95, seed = NULL, cladeChunkIn=NULL, storehClust=FALSE, storeChunks=FALSE, rowSample=FALSE, r=seq(.5,1.4,by=.1), height=800, width=800, highlightFileName=NULL, metadata = FALSE, plotOut = TRUE, logFileName = "").

The main function in trueTree that is used to perform cluster validation. trueTree will output a dendrogram as well as an analysis of the cophenetic correlation coefficients from the bootstrap process and return an object that stores all of the important information for later use.

input.file - Character vector containing the name of the file containing the input data

outputFilename - Character vector containing the name of the file to save the dendrogram showing the results. The dendrogram will be saved as a .png file but the file extension should be left out of the file name as trueTree will add it automatically. Additionally a histogram showing the distribution of cophenetic correlation coefficients will be saved to outputFilename-histogram.png. If outputFilename is set to NULL, which it is by default, R will instead create a window to display the results.

main - Optional character vector containing a title for the dendrogram. This title will be displayed at the top of the dendrogram.
textLabs - See description for chunksize
chunkSize - textLabs and chunkSize are a pair of optional vectors that are used to relabel all the chunks in the dataset. These parameters are designed to be used when there the dataset contains a number of texts which have been split into multiple chunks. textLabs is a character vector and should contain the names of all the texts in the order in which they appear in the dataset. chunksize is a vector that should contain then number of chunks in each of those texts in the same order as the texts were listed. The function will then give the initial chunk the label for first chunk in the first text, the second chunk as being the second chunk in the first text and so forth until the number of chunks listed in chunkSize has been seen, at which point it will start labeling chunks as being part of the second text. For instance if there were five chunks total and textLabs = ("Bob", "George") and chunkSize = (2,3) the chunks will be labeled Bob 1, Bob 2, George 1, George 2, George 3. textLabs and chunkSize need to have the same length and the sum of the chunksizes needs to be equal to the total number of chunks.
distMetric - Character vector containing the method to use when calculating distances between chunks. Supports "euclidean", "maximum", "manhattan", "canberra", "binary", "minkowski", "correlation", "uncentered", and "abscor" methods.
clustMethod - Character vector containing the method to use when calculating distancesbetween clades containing multiple chunks. Supports "ward", "single", "complete", "average", "mcquitty", "median", and "centroid" methods.
input.transposed - Boolean that tells trueTree what input format to expect. input.transposed should equal TRUE if each row holds a chunk and each column a word, or FALSE if each column holds a chunk and each row a word.
nboot - Integer that says how many resampled datasets should be generated for each r value
runParallel - Boolean that whether to run the bootstrap in parallel. If TRUE the bootstrap will be run over multiple processors with the details of how specified by the numCPUs and clusterType parameters. If FALSE then the bootstrap will be run on a single processor.
numCPUs - Integer denoting how many processors should the code be run over. This parameter has no effect if runParallel is set to false.
clusterType - Character vector containing what type of machine cluster is being used. Valid types are "sock" for raw sockets, "pvm" for parallel virtual machine, "nws" for NetWorkSpaces and "mpi" for Message Passing Interface. "sock" is the only option that can be used without any initial setup but only allows the bootstrap to be split across the cores in the machine the code is being run on. "mpi" supports running the bootstrap across multiple machines but requires an existing mpi runtime. "pvm" and "nws" are untested. See the snow documentation Tierney *et al.* (2011) for more information on how to use these options.
confidenceInterval - Numeric value containing is the confidence interval to use for the cophenetic correlation coefficient analysis. The function will return the cophenetic correlation coefficients that lie at the upper and lower bounds of this interval. The interval should be given as a decimal not a percentage.
seed - Parameter that gives a seed to use for the random number generator. If runParallel is FALSE, then seed should be a single integer. If runParallel is true then seed should be a vector containing one seed for each processor with each seed being an integer. If seed is left to NULL then a seed or seeds will be automatically generated based on the current time.
cladeChunkIn - Vector that allows the use of intra-clade based resampling. cladeChunkIn should contain 1 integer for each chunk, with the integer giving the number of the clade that chunk is in. If cladeChunkIn is left to NULL then intra-chunk based resampling will be used instead.
storehClust - Boolean specifying whether or not to store all the generated hclust objects from the bootstrap. If TRUE the hclust objects will be saved, if FALSE they won't. WARNING: Storing the hclust objects consumes a lot of memory. If this option is used in conjunction with a high nboot value, or a large number of r values, the computer running trueTree may run out of RAM, causing the code to crash.

storeChunks - Boolean specifying whether or not to store all the resampled datasets from the bootstrap. If TRUE the resampled datasets will be saved, if FALSE they won't. WARNING: Storing the resampled datasets consumes a lot of memory. If this option is used in conjunction with a high nboot value, or a large number of r values, the computer running trueTree may run out of RAM, causing the code to crash.
rowSample - Boolean specifying whether or not to use pvclusts original resampling method. If TRUE interrow based resampling will be used, where rows are picked from the original dataset to place in the new resampled dataset. WARNING: This resampling method is poorly suited to some domains like text mining. Before this option is used the user should determine whether it is appropriate for the domain where it is being applied.
r - Vector specifying what r values should be used to run the multiscale bootstrapping. r must contain at least two values for any AU values to be generated.
height - Vector specifying the height in pixels of the dendrogram that will be created. This parameter only works when outputting to a file.
width - Vector specifying the width in pixels of the dendrogram that will be created. This parameter only works when outputting to a file.
highlightFileName - Character vector containing the name of a file containing a list of chunks that should be highlighted in the dendrogram. The file should list one chunk per line, and the line should contain the exact label of the chunk to highlight.
metadata - Boolean specifying whether the input file contains metadata. Metadata is used to color a dendrogram. Currently only one metadata format is recognized. See Appendix A for details.
plotOut - Boolean that specifies whether the dendrogram containing the results of the cluster validation should be displayed. If true the dendrogram will be displayed/saved to a file. If false no action will be taken.
logFileName - Optional character vector containing the name of a file to write the results when running trueTree. The results printed are various pieces of timing data, as well as the cophenetic correlation analysis. If the file specified already exists the results of the current run will be appended to the end of the file. If left to the default of `""` all output will be displayed in the console.

trueTree returns a trueTree object that is a list with the following components

hclust The hclust object formed from the original dataset
edges Table containing AU and BP values for every clade in the order that the clades were formed, as well as the standard error for both, the v and c found during the calculations.
count Table containing the number of times each clade in the original clustering was formed during the bootstrap for each r value used.
msfit Msfit object created containing the results of the curve fitting used to calculate the au values. See pvclusts documentation for more information (Suzuki and Shimodaira, 2011)
nboot The number of resampled databases generated for each r value
r Vector containing all the r values used in the multiscale resampling
storehClust List containing all the hclust objects generating while running the bootstrap. This will be NULL if store was set to false in trueTree's parameters. The list will contain one sublist for each r value used, and each sublist has all the hclust objects generated for that r value.
distance The distance matrix generated from the original dataset
seed A vector containing all the seeds for random number generator used to run the bootstrap. There will be one seed for each processor used.
storeChunks List containing all the resampled datasets objects generating while running the bootstrap. This will be NULL if storeChunks was set to false in trueTree's parameters. The list will contain one sublist for each r value used, and each sublist has all the resampled datasets generated for that r value.

`plot.trueTree(x, outputFilename=NULL, height=800, width=800, specialLabels=NULL, showBP=FALSE)` Function that plots the dendrogram contained

in a trueTree object and shows the AU values. If the object contains metadata the dendrogram will automatically be colored.

x - trueTree object to plot

outputFilename - Name of file to output to. If set to NULL R will create a window to draw the plot in instead. The output will be saved as a .png file, but the filename should not include the extension as it will be added automatically.

height - How tall the plotted dendrogram should be. This parameter only works when outputting to a file, but is useful for plotting large dendrograms that won't easily fit on a single screen.

width - How wide the plotted dendrogram should be. This parameter only works when outputting to a file, but is useful for plotting large dendrograms that won't easily fit on a single screen.

specialLabels] - A character vector containing the labels of chunks that should be highlighted in the plotted dendrogram. Any chunk which label is in the vector will be colored yellow instead of it's normal color.

showBP - Boolean dictating whether or not to display BP values for each clade. By default trueTree only shows the AU values because they are more accurate.

```
varianceTest(input.file, distMetric = "euclidean" , clustMethod =
"average" , input.transposed = TRUE, nboot = 10, runParallel = FALSE,
numCPUs = 2, clusterType = 'SOCK', cladeChunkIn=NULL, rowSample=FALSE,
r=seq(.5,1.4,by=.1), metadata = FALSE, testRuns = 5)
```

Function for testing how great the AU and BP values can vary for a particular dataset when using a specific set of parameters. varianceTest runs trueTree a user specified number of times, and finds the largest amount an AU value differs over all those runs and the largest amount a BP differs over all those runs.

testRuns How many times should trueTree be run.

... All other parameters modify the behavior of trueTree when it is run and function identically to the parameters with the same name for the trueTree function.

4 Use Cases

trueTree can be applied to a number of different fields of research. The three examples presented in this section are intended to show both the proper usage of trueTree and give a small sample of the areas where trueTree can be applied.

4.1 *Daniel* and *Azarias*: Old English Text Mining

Daniel and *Azarias* are two old Anglo Saxon poems. The goal of this test is to see if the relationship between the two texts can be detected. These two poems are notable because scholars currently believe that the middle portion of *Daniel* and *Azarias* were both derived from a common source, due to similarities in both the word choice and structure of those sections (Drout *et al.*, 2011). Due to this relationship between the two works these texts make an ideal test case to see if trueTree works properly and is able to produce results that match the known situation.

The first step towards running trueTree on these texts was to create an input file. *Daniel* is a longer text then *Azarias* and was thus divided into ten chunks of roughly even length. Each chunk will be referred to by the name of the text followed by the chunk number so the first chunk of *Daniel* is *Daniel*1. Of those ten chunks, *Daniel*4 through *Daniel*6 are the most important. The section of *Daniel* contained in *Daniel*5 is the section taken from the same source as *Azarias*. *Daniel*4 and *Daniel*6 contain the start and end of this section respectively, but also contain portions of *Daniel* that don't draw from that particular source. *Azarias* was kept as one chunk since the goal of this test is to see if the relationship between the two texts can be detected, and dividing *Azarias* into multiple chunks would make the results less easily interpreted. These specific divisions were taken

from previous research done on *Daniel* and *Azarias* using cluster analysis in Drout *et al.* (2011). These chunks were then converted into an input file holding the word counts for each chunk.

A successful test of trueTree in this use case will detect the similarities between *Azarias* and the chunk *Daniel*5. *Daniel*4 and *Daniel*6 which each only have a portion that draws from that source should be found most similar to each other but still similar to *Azarias* and *Daniel*5. Finally the other chunks in *Daniel* should be found to be quite different from those 4 chunks but similar to each other. In addition the cluster validation performed should confirm these results with a significant degree of confidence.

The nboot number was one very important parameter that needed to be decided before trueTree was run. The nboot number decides how many resampled datasets are generated while running the bootstrap. Since the bootstrap involves randomness it is important that a large number of datasets be generated to "even" out the randomness and get accurate results. However the more datasets that are generated the longer it takes for trueTree to run. As such it is important to find a good balance between accuracy and runtime when choosing an nboot number.

trueTree's `varianceTest` function is designed to help find the right nboot number. It runs trueTree on a dataset using the same set of parameters a given number of times keeping track of the results each time. After it's done for each clade the function looks at all the AU values given to that clade over the course of it's test runs, and sees how much the lowest and the highest values differ. It then reports the greatest difference found between those two values in any of the clades. This number measures how much variance is possible in the results for a given parameter set. It should be noted that this shows the worst case scenario not the average, so the number being high doesn't mean that the results for every clade greatly will vary across runs, rather just that the results in one of the clades did. As such the number represents the most a particular value is likely to be off from the correct result. Table 4.1 shows the results of using `varianceTest` on the *Daniel Azarias* dataset for various nboot values, with trueTree being run 20 times for each nboot value.

nboot	Greatest difference in AU values
10	99.250648819396
100	59.9866535843093
500	25.6238885031469
1000	18.4479759393262
5000	8.04831836972173
10000	5.45965455886498
20000	3.94242458209143

Table 4.1: Worst case results of using the `varianceTest` to run trueTree on the *Daniel Azarias* dataset twenty times for various nboot values

The results show that when nboot is low the results are highly unreliable since it's possible for them to greatly differ between runs. Once nboot reachs 5000 the results become somewhat reliable, with the worst difference being only 8 points with the actual difference between runs likely being less the vast majority of the time. The gain in accuracy from increasing nboot gets smaller the larger nboot gets. Going from 5,000 to 10,000 decreases the greatest difference by roughly 2.5 points although it adds 5000 more datasets to generate. Going from 10 to 100 added only 90 more datasets but reduced the greatest difference by roughly 40 points. Combining this fact with the fact that adding more datasets to generate takes more time to run, it means that trying to get almost perfect accuracy will inevitably lead to lengthy runtimes, but getting accuracy that is good enough can usually be achieved while keeping a reasonable run time. Ultimately 10000 was selected for the nboot. The goal of this particular experiment is just to determine whether it's likely or not that a clade is valid, and that doesn't require a huge amount of accuracy, and as such mistaking a clade which has an 80 percent likelihood of being valid for having an 85 percent likelihood doesn't greatly impact the final results. As such an nboot that could be run in a reasonably brief time period was chosen.

There are three other parameters that can affect the outcome of running trueTree. The first is

the set of r values to run the multiscale bootstrapping. By default pvclust, upon which trueTree was built, uses the values [.5, .6, .7, .8, .9, 1, 1.1, 1.2, 1.3, 1.4]. Testing has shown little reason to change this. The other two parameters are the distance metric used to find the distance between individual chunks, and the linkage metric which is used to find the distance between clades containing multiple chunks. Past research using cluster analysis on *Daniel* and *Azarias* has used the euclidean distance metric and the average linkage metric (Drout *et al.*, 2011), which are described in chapter 2.1.2, and as such are the metrics that will be used for this example.

The actual call to the trueTree function will vary slightly depending on whether it's run on a cluster of machines or a single machine. In addition the input files for trueTree may be in one of two formats, either having words as rows and chunks as columns, or having chunks as rows and words as columns. As such the trueTree function also has a parameter, `input.tranposed` to deal with both types of files. The file containing the input data for *Daniel* and *Azarias* has words as rows and chunks as columns so `input.tranposed` gets set to FALSE. The two functions calls are shown below in Listings 4.1 and 4.2

```
#result <- trueTree("danile-azarius.txt", nboot=10000,
    distMetric = "euclidean", clustMethod = "average",
    runParallel = TRUE, input.transposed = FALSE,
    numCPUs = 2, clusterType = "SOCK")
```

Listing 4.1. Command to run trueTree on *Daniel* and *Azarias* on a single machine with a dual-core processor.

```
#result <- trueTree("danile-azarius.txt", nboot=10000,
    distMetric = "euclidean", clustMethod = "average",
    runParallel = TRUE, input.transposed = FALSE,
    numCPUs = 22, clusterType = "MPI")
```

Listing 4.2 Command to run trueTree on *Daniel* and *Azarias* on a cluster of 11 machines with dual-core processors

These calls can be modified to support other types of computers. If trueTree is running on a single machine with more then 2 cores in it's processor, numCPUs in Listing 4.1 can be changed to the actual number of cores, and similarly numCPUs in Listing 4.2 can be changed to account for a cluster with access to a greater or lesser number of cores. Finally runParallel can be set to FALSE to make trueTree only run on a single core, ideal for letting trueTree run in the background of a computer used for multiple purposes.

The first piece of output produced is a dendrogram showing the results of running trueTree. The dendrogram is shown in Figure 4.1.

The dendrogram is the result of running hierachical clustering on the *Daniel Azarias* dataset. The results of the bootstrap are seen in the numbers in red which show the AU values calculated for each clade. *Daniel*5 and *Azarias* clustered together as was expected, and that clade has an AU value of 84 meaning there is an 84 percent chance that that clade is correct. Also as expected *Daniel*4 and *Daniel*6 clustered together, and the AU value of that clade strongly supports the validity of that clustering. The clade with containing all four of those chunks however only has an AU value of 65. That says that while there is a strong possibility that the clade is correct there is some uncertainty. Likely it means there is a chance that *Daniel*4 and *Daniel*6 belong with the rest of *Daniel* instead of with *Daniel*5 and *Azarias* since only part of them draw from the same source as the latter two. The other main clade of interest is the ninth clade that contains the other 7 chunks of *Daniel*. That clade has an AU value of 90 saying that there is a 90 percent chance that a clade containing only those chunks exists. This is equivalent to saying that there is a 90 percent chance that the chunks in clade 9 are separate from the 4 chunks on the left side of the dendrogram. These results all support what was already known about the two texts. *Daniel*5 and *Azarias* are definitely similar, *Daniel*4 and *Daniel*6 are similar to each other, and the rest of *Daniel* is different from those sections. The

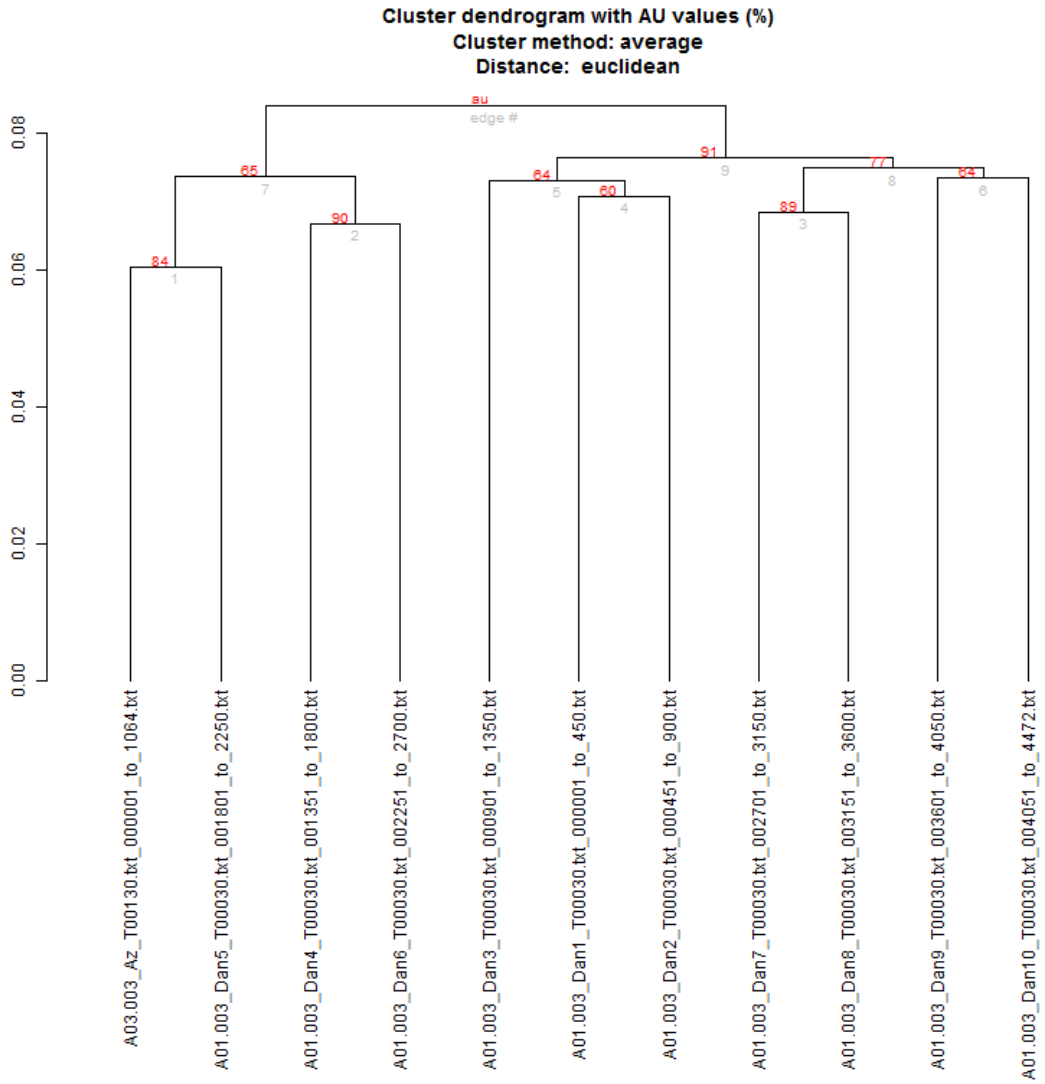

Figure 4.1: Dendrogram made using trueTree from ten chunks of *Daniel* and *Azarias*

only thing remotely questionable about the dendrogram is whether *Daniel*4 and *Daniel*6 are more similar to *Daniel*5 and *Azarias* or to the rest of *Daniel*, and even there the evidence weighs in the favor of the first of those two options.

The second piece of output received is the results of analyzing the cophenetic correlation coefficients found during bootstrapping, shown in Listing 4.3

```
Original Cophenetic correlation 0.591120277912255
Number of Cophenetic correlation values 100000
Minimum Cophenetic correlation -0.101363301731442
2.5% interval Cophenetic correlation 0.289437347300661
Median Cophenetic correlation 0.52215378885636
97.5% interval Cophenetic correlation 0.628478451792611
Maximum Cophenetic correlation 0.657791044746264
Mean Cophenetic correlation 0.5024860611722
Standard Deviation Cophenetic correlation 0.092783822771156
```

Listing 4.3 Results of analyzing the cophenetic correlation coefficients while running trueTree on *Daniel* and *Azarias*.

The original cophenetic correlation coefficient was 0.591120277912255. This indicates some correlation between the distances in the dendrogram, and the original distance matrix, but the relationship is somewhat weak. The additional numbers look at how much variance existed in the Cophenetic correlation coefficient over the course of the bootstrap. The original value was greater than mean and median values, and the lower end of the 95 percent confidence interval is 0.289437347300661 which is much lower. trueTree also provides a histogram of this data that confirms this result, shown in Figure 4.2.

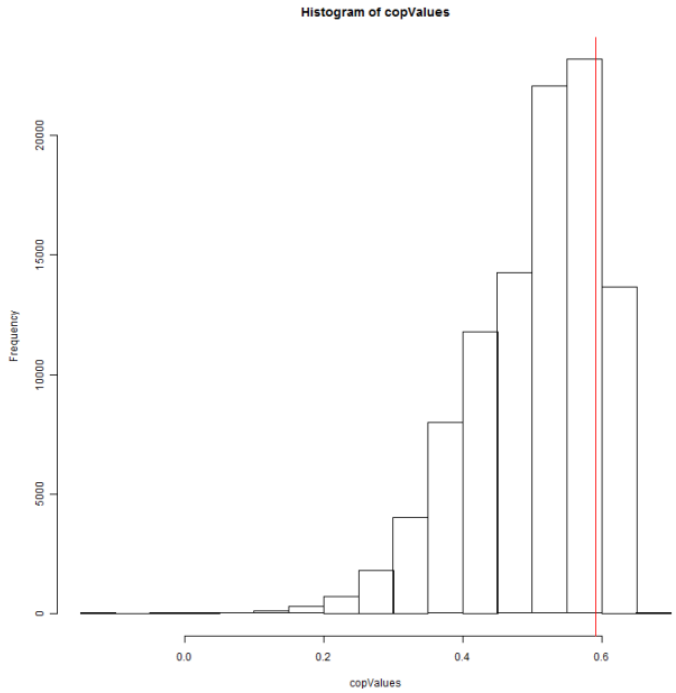

Figure 4.2: Histogram of cophenetic correlation coefficients calculated while running trueTree on *Daniel* and *Azarias*

The red line in the histogram shows where the original Cophenetic correlation coefficient lies.

The histogram shows that there is a wide distribution of values and that the original value lies at the upper end of the range of values. This indicates a degree of noise being encountered when the Cophenetic correlation coefficient is measured for this dataset resulting in the range of values when small changes are made to the dataset. As such the measurement is fairly imprecise, so we conclude for this example that the cophenetic correlation coefficient does not give much useful information.

Finally there are some performance numbers to consider. The dataset has 11 chunks and 1911 unique words with the majority appearing only a few times across the two texts. The majority of the chunks are only 450 words long with *Azarias* being about 1000 words long. Overall this is a relatively small dataset compared to some of the others trueTree might cluster. The time it takes to run trueTree on the dataset is reasonably quick and is shown in Table 4.2. Technical specifications for the machines used can be found in Appendix C.

Machine used	Runtime
Lab Machine Cluster (22 processors)	1.22 mins
Lab Machine 2 cores	11.35 mins
Lab Machine 1 core	18.25 mins

Table 4.2: Total time taken to run trueTree on *Daniel* and *Azarias* with an nboot of 10000

These results show that as expected trueTree runs faster when run on multiple processors. What is important to note is that doubling the number of cores does not cut the runtime in half. This is because of the added overhead of running code in parallel. The vast majority of the work trueTree does can be done in parallel, and as such the length of time spent on that portion of the code is roughly halved but the extra overhead, and the portions of the code that are inherently sequential increases the runtime.

4.2 Federalist Papers: Text Mining

The Federalist papers are a series of 85 articles written from October 1787 to August 1788 advocating for the state of New York to ratify the constitution. They were written by three people, Alexander Hamilton, James Madison, and John Jay, but for several of the papers it is currently unknown which of the three was the author. As such the Federalist Papers are one of the classic examples of an authorship attribution problem and much research has been done on them (Jockers and Witten, 2010).

The individual articles can be divided into five categories. The first is papers that are known to be written by Hamilton of which there are 51. Similarly there 14 papers known to be have been written by Madison, and 5 by Jay. Of the remaining articles there are 3 that were coauthored by Madison and Hamilton and finally there are 12 whose authorship is disputed. (Jockers and Witten, 2010), (Forsyth, 1997).

Matthew Jockers performed a study (Jockers and Witten, 2010) comparing how ten different classification techniques attributed the disputed and coauthored papers to the three different authors. For all of those papers the majority of the ten different techniques he tested attributed the paper to Madison. Most of the papers were attributed to Hamilton by one to three of the techniques used, but for a few the decision to attribute the paper to Madison was unanimous. None of the papers were attributed to Jay using any of the techniques.

Since trueTree does cluster analysis instead of classification, it does not directly attribute the disputed papers to a specific author. However, articles written by the same author should have a number of underlying similarities so the disputed papers should cluster together with other papers written by the same author. In addition based on the previous research, it is expected that the majority of the disputed papers should cluster with papers written by Madison.

The dataset given to trueTree consisted of 85 chunks, one for each paper. trueTree was run with an nboot of 10,000. Each chunk was given a label that contained the number of the paper as well as a letter denoting which category it fell into. As such a paper labeled H was written by Hamilton, M

by Madison, and J by Jay. Papers labeled C were coauthored, and papers labeled D had disputed authorship. The results are shown in Figure 4.3.

The 3 coauthored texts, papers 18, 19 and 20, are clustered together. The AU value for the clade containing all 3 is 95, strongly suggesting that they belong together. The clade they are in is next joined with a giant clade where the vast majority of the papers were written by Hamilton and Madison. This makes sense. Due to being coauthored, the three papers wouldn't resemble papers written by Hamilton or Madison individually but would have elements of the style of each and should match a clade that contains papers from both. The AU value for that clade is 90 which again means the clade is quite likely to be "real".

The disputed papers ended up in several spots in the dendrogram. Papers 49, 51, and 54 appear in a clade with an AU value of 96 and three papers written by Madison, strongly suggesting they were written by Madison. Paper 50 ended up in a clade with one other paper by Madison and an AU value of 87, which again suggests Madison is the likely author. Papers 52, 53, 55, 57, 58 and 63 all clustered together alongside two papers by Madison. That clade has an AU value of 98, yet again leaving little reason to doubt that Madison was the author. Finally paper 62 clustered together with 3 of Madison's papers and as such given the high AU value for that clade of 88 the most likely author of the paper is Madison. The only disputed paper left is paper 56, for which the results are inconclusive. The clade the paper joins contains a number of papers by both Madison and Hamilton, so there is nothing to suggest it belongs to one of the two over the other. These results match Jockers' results that found all the disputed papers being assigned to Madison the majority of the time (Jockers and Witten, 2010).

Also of note is that the five papers written by Jay were put in a cluster entirely separate from the rest of the dendrogram. The fact that they tightly clustered together is to be expected given they share a common author, and the fact they are separate from all the disputed papers supports Jocker's findings that none of the disputed papers were ever assigned to Jay (Jockers and Witten, 2010). Also of note is a number of the larger clades had AU values of 0. This says that the clade isn't an accurate representation of the underlying data. The most likely explanation is there are several chunks that are similar to those in the clade, but the clade is too small to include all of them. As such when resampled datasets are generated the small variations in the data cause some of those similar chunks to replace other chunks in the clade meaning that specific clade never forms.

As for the cophenetic correlation coefficient, the results suggest that it was somewhat more reliable then the one seen for *Daniel* and *Azarias* as seen in Listing 4.4

```
Original Cophenetic correlation 0.830189163986368
Number of Cophenetic correlation values 100000
Minimum Cophenetic correlation 0.531131243500471
2.5% interval Cophenetic correlation 0.713275080936535
Median Cophenetic correlation 0.789584790617164
97.5% interval Cophenetic correlation 0.82105925698192
Maximum Cophenetic correlation 0.850274664527484
Mean Cophenetic correlation 0.783192900184687
Standard Deviation Cophenetic correlation 0.0281907300892871
```

Listing 4.4 Results of analyzing the cophenetic correlation coefficients while running trueTree on the 85 Federalist papers.

The original cophenetic correlation coefficient is quite high at 0.830189163986368. This actually puts it past the 95 percent confidence interval, and thus makes it higher than the mean and median values. However the total range of coefficients found is significantly smaller than those seen for *Daniel* and *Azarias* with the confidence interval ranging from 0.713275080936535 to 0.82105925698192. As such, while the original value is likely too high given it's position outside the main portion of the distribution, the fact that the overall distribution is smaller means there is less noise skewing the results. The histogram in Figure 4.4 confirms this.

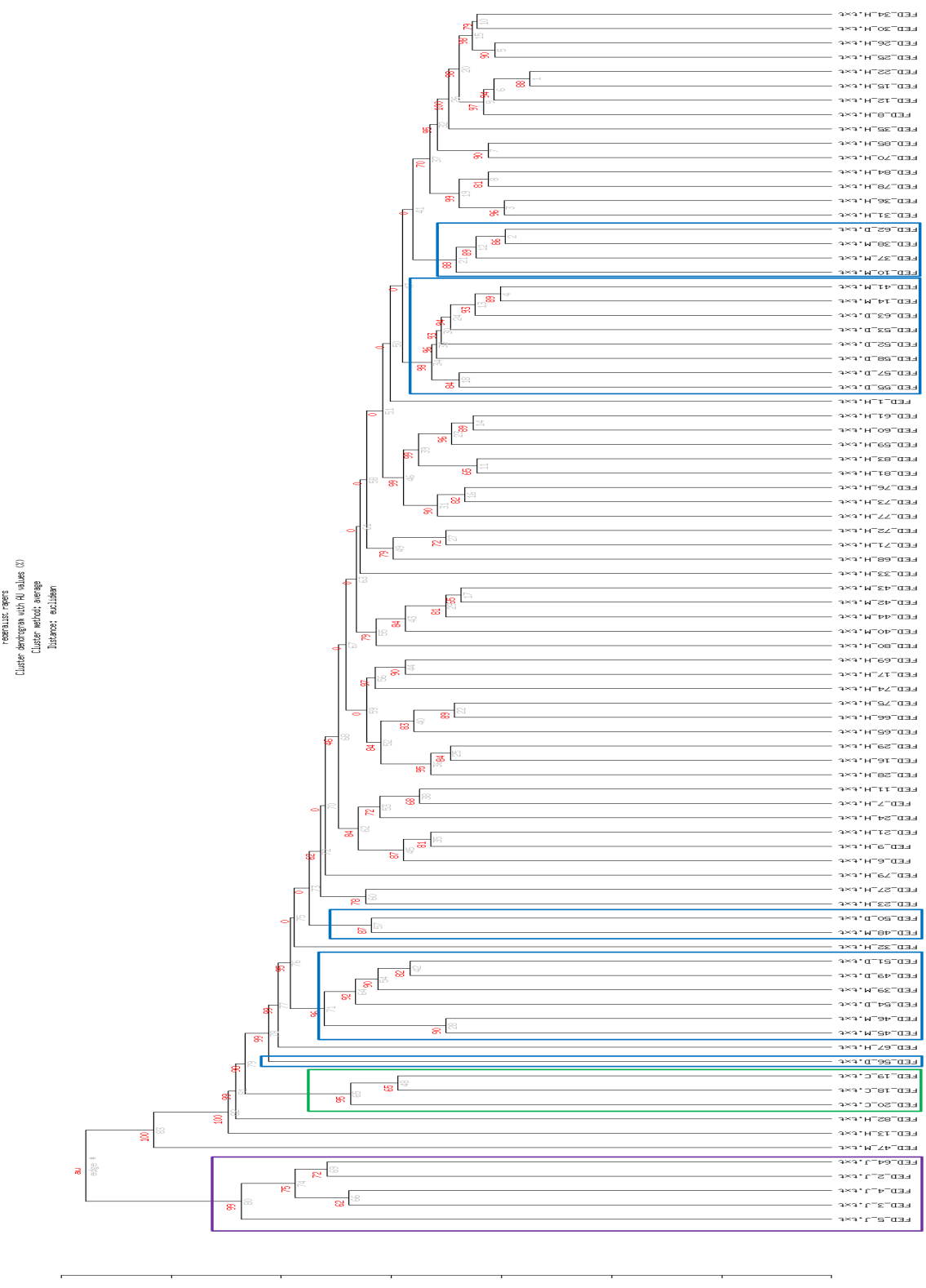

Figure 4.3: Dendrogram made using trueTree with the 85 Federalist Papers. The clade containing papers written by Jay is in the purple box. The clade containing papers that were coauthored is in the green box. The clades containing papers with disputed authorship are in the blue boxes.

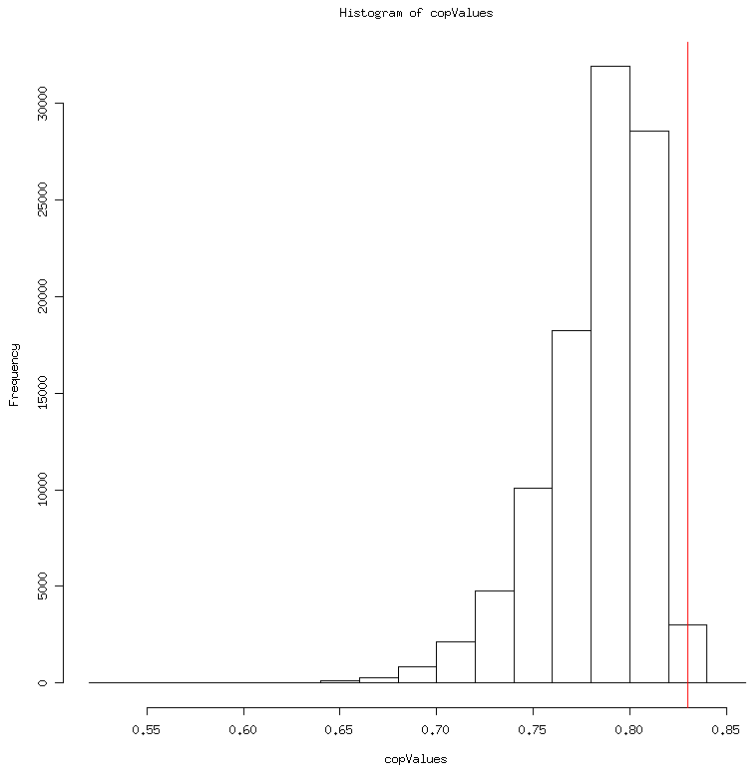

Figure 4.4: Histogram of cophenetic correlation coefficients calculated while running trueTree on the Federalist Papers. The red line shows the cophenetic correlation coefficient for the original dendrogram

The Federalist Papers dataset contained 85 chunks with a total of 8819 unique words. The length of each paper varies but they are usually between one to three thousand words long. Most of the unique words only appear one or two times and only in a few select papers, although there are a few words like "the" that can appear dozens of times. By every measure this is a significantly larger dataset than *Daniel* and *Azarias* and it comes as no surprise that the run time ends up being significantly longer than the time for the earlier dataset. The time it takes to run trueTree on the dataset is shown in Table 4.3. Technical specifications for the machines used can be found in Appendix C.

Machine used	Runtime
Lab Machine Cluster (22 processors)	1.60 hours
Lab Machine 2 cores	16.73 hours
Lab Machine 1 core	20.06 hours mins

Table 4.3: Total time taken to run trueTree on The Federalist Papers with an nboot of 10000

There are two things noteworthy about these results. First is that this dataset starts to demonstrate why the ability to run the bootstrap in parallel is so important. It takes a large cluster of machines more than a hour and a half to process the dataset. With a single machine running the same test, it can easily last most of a day which can get quite inconvenient especially if multiple tests need to be performed. The other item of note is the rather underwhelming decrease in runtime from going from 1 to 2 cores on the lab machine. The reason is fairly simple. The Lab Machines have only 2 cores and those cores not only have to run the experiment but also have to run the rest of the computer. As such one of the cores is going to be splitting work between the experiment and

the other processes on the computer such as maintaining the operating system, meaning less CPU time is given to the experiment then would initially be expected. As such while it might be expected that going from running trueTree on 1 core to 2 cores would cause the code to run in roughly half the time it actually ends up taking a lot longer for the code to finish.

4.3 Genomics

At the time of this writing this has been some ongoing attempts to apply trueTree towards genomics research. Since this research is ongoing there are no final results to report, but a look at the current progress does help reveal the wide range of uses trueTree has. The goal is to build from research done by Dyer *et al.* (2007) and LeBlanc *et al.* (2012) in using classification methods to detect evidence of horizontal transfer in microorganisms. Horizontal transfer occurs when two microorganisms swap sections of their DNA with each other. The research was done using several datasets the latest of which consists of the genomes of 74 different organisms: 35 of the organisms belonging to the Archaea Kingdom, 39 to the Bacteria Kingdom. The genomes were divided into chunks of 10,000 base pairs, and a Classification and Regression Tree (CART) Algorithm, as well as a Support Vector Machines(SVM) algorithm, both of which are types of classification methods, were used to create models that could classify chunks as being from Bacteria or Archaea organisms. Chunks which were incorrectly classified as belonging to the wrong kingdom were possible candidates for being the result of horizontal transfer.

Currently trueTree is being used to help support previous results (LeBlanc *et al.*, 2012). Most of the organisms in the dataset have lengthy genomes that are split into several hundred chunks. The overall dataset consists of 18,637 chunks too many to run trueTree on in full even when using a machine cluster, due to a set of that size both taking too long to run, and requiring more RAM then the computers in the machine cluster used for research have access to. Instead a dendrogram has been created from the full dataset without using any of the computationally intensive cluster validation, and work is ongoing to select a smaller subset of that data to run cluster validation on to confirm the most important results. The dendrogram makes use of an algorithm in trueTree's designed to color a dendrogram to help distinguish between chunks belonging to bacteria and archaea. More information about how this algorithm works can be found in Appendix A.

The dendrogram as a whole is too large to include in this document being 200,000 pixels in width. However it is possible to show the most relevant sections contained within it. There are six chunks that both CART and SVM always classified as belonging to the wrong kingdom. By looking at where those chunks end up in the dendrogram, it is possible to see if they appear to group together with organisms from a different kingdom then they belong to which would support previous results. In the dendrogram clades are colored green if they only contain chunks belonging to bacteria, and red if they only contain chunks belonging to archaea. In addition the six chunks that were always classified wrong are colored yellow. Clades that contained chunks from both types of organisms, or contained chunks from one of the two types as well as one of the six clades that always classified wrong are colored blue.

Four of the six chunks that were examined were clearly clustered with organisms belonging to a different kingdom then their own. The clearest of these cases came from chunk 41 of the genome of the *Anaerocellum Thermophilum* organisms, a type of bacteria. As the 41st chunk of the genome it contains base pairs 400000 to 409999 of the genome. Figure 4.5 shows that it clustered with a large number of archaea chunks.

The clade the *Anaerocellum Thermophilum* chunk gets put in was actually to large to fit on the page, and in the actual dendrogram it goes on further, but the remaining portion looks the same as what is seen in Figure 4.5 consisting only of chunks from organisms belonging to the archaea kingdom. The most ambiguous of the four chunks that appeared to cluster with the wrong kingdom was chunk 351 of the *Methanosarcina Acetivorans* organism, a member of the archaea kingdom. Since the chunk is the 351st chunk in the genome it contains base pairs 3500000 to 3509999 of the genome. As seen in Figure 4.6 the chunk clustered together with 15 other chunks from that organism, but those 15 chunks then clustered together with a large number of bacteria chunks, which is an anomaly, as most archaea chunks ended up clustering together with hundreds of other chunks from

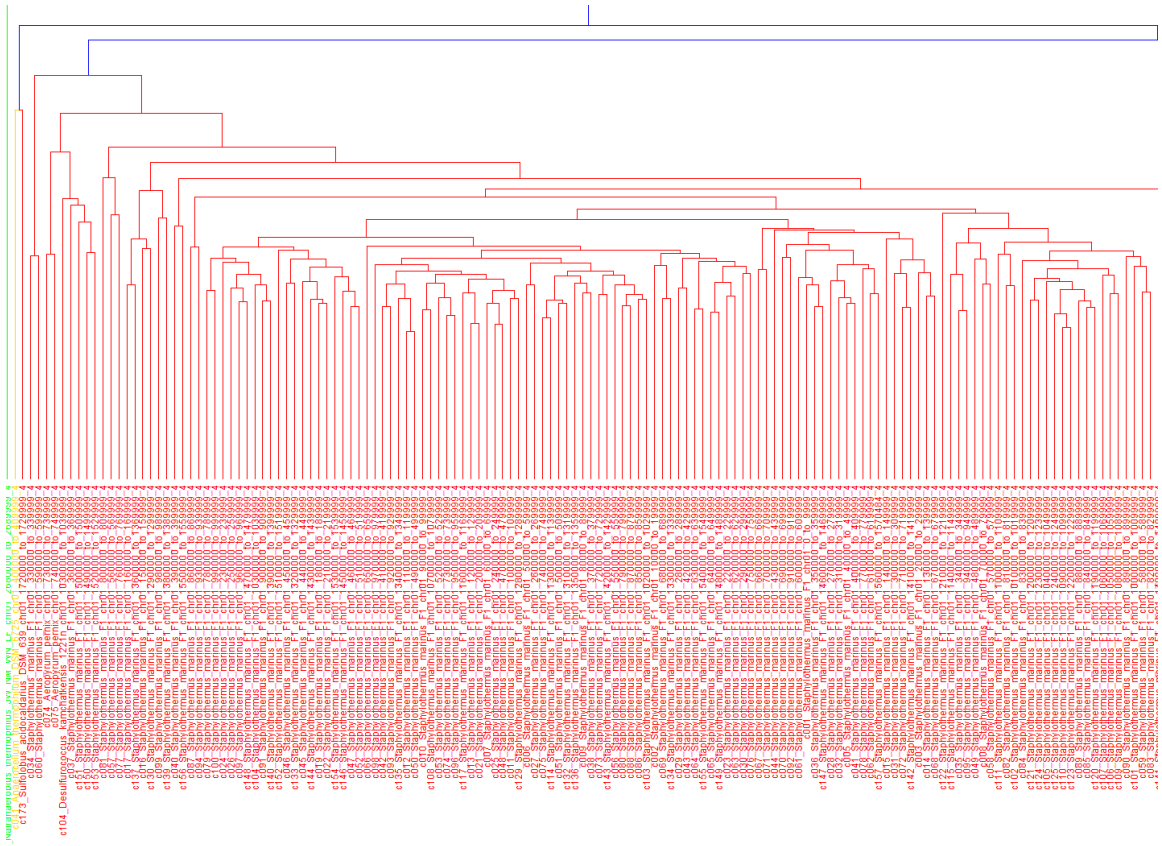

Figure 4.5: Section of the dendrogram of 18,637 genomics chunks, showing the location of chunk 41 from the genome of the *Anaerocellum thermophilum* organism. Red indicates that a chunk belongs to an organism in the archaea kingdom or that a clade only contains chunks belonging to organisms in the archaea kingdom. Green indicates that a chunk comes from a organism belonging to the bacteria kingdom or that a clade only contains chunks from organisms belonging to the bacteria kingdom. Blue indicates a clade containing chunks from more then one kingdom. Yellow indicates a chunk that both classifiers always classified wrong.

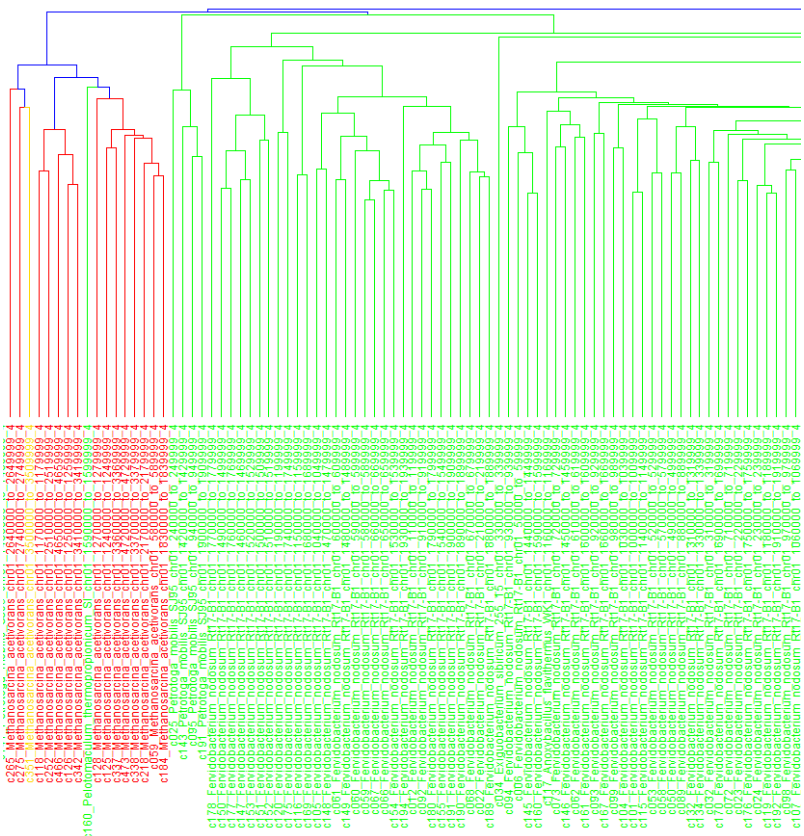

Figure 4.6: Section of the dendrogram of 18,637 genomics chunks, showing the location of chunk 351 from the genome of the *Methanosarcina acetivorans* organims. Red indicates that a chunk belongs to an organism in the archaea kingdom or that a clade only contains chunks belonging to organisms in the archaea kingdom. Green indicates that a chunk comes from a organism belonging to the bacteria kingdom or that a clade only contains chunks from organisms belonging to the bacteria kingdom. Blue indicates a clade containing chunks from more then one kingdom. Yellow indicates a chunk that both classifiers always classified wrong.

that kingdom.

Finally two of the chunks appeared to cluster together with organisms belonging to the same kingdom. For example chunk 314 of the *Methanosarcina Acetivorans* organism, which is a member of the archaea kingdom. The chunk being the 314th chunk in the genome contains base pairs 3130000 to 3139999 of the genome. The chunk clustered together with other archaea chunks including a number of chunks from the *Methanosarcina Acetivorans* genome as seen in Figure 4.7. This suggests that the chunk belongs to an archaea organism and as such the dendrogram classifies it correctly.

To summarize the preliminary results, after looking at the six chunks past research has shown do not get classified correctly for either of the classification methods used, the findings from running a hierarchical clustering algorithm seem to support earlier results for four of them, and disputes those results for the other two. Since work is still ongoing to create a subset of the data to run cluster validation on it is difficult to say at this point how reliable these results are. The results of cluster validation could easily throw some of these findings into question.

5 Conclusion

The experiments in the use case section confirm several things. First of all cluster analysis works. Comparing the results of cluster analysis to existing research shows that cluster analysis confirms the earlier research. In addition, the AU value is effective as a method of cluster validation. Almost every clade that showed the dendrogram matching the expected results had high AU values and as such almost every time there was evidence a clade should be correct the cluster validation confirmed the clades correctness meaning the expected results matched the evidence. The cophenetic correlation coefficients analysis on the other hand were a mixed bag. For some datasets, it showed that there was too much noise in the values to be reliable, while other times the distribution of values was small and the results could be considered precise. As such the cophenetic correlation coefficient of a clustering can be useful at times but it is not consistently reliable and as such is not suited to be the primary means of validation in text mining. Finally, parallel processing proved to be key to running trueTree on certain datasets, as the runtime proved to be prohibitive otherwise.

While trueTree was designed to handle everything necessary to run cluster validation, there are several possible improvements that could be implemented at a later point in time. The main two areas are expanding the graph coloring code and adding dynamic load balancing. The graph coloring code as it currently stands is only designed to be used on genomics data that is a specific file format, as that was the only dataset large enough to require the coloring to produce an easily readable dendrogram. However there are many other types of data that could benefit from this type of coloring. Ideally the person using trueTree could give it an input file with metadata in any format they desire, specify where in the input file the metadata is contained, and map specific values that could be contained in the metadata to colors to use in the dendrogram. The big obstacle to implementing this within the current implementation is finding a good way to actually get this data as input from the user given the complexity of what is required. The code to assign the colors lies in the `R` function and should be easily modifiable if proper data can be provided.

Implementing dynamic load balancing would be trickier. trueTree currently does static load balancing by dividing up all the work evenly between processors at the start of the bootstrap process and then waiting for all the processors to finish. This means that the length taken to run the bootstrap is determined by the slowest processor as even if all the others finish their tasks trueTree still has to wait for the slowest one to complete it's workload. Ideally trueTree would handle the load-balancing dynamically and reassign the workload as necessary so that no processor is ever left idle. For example, if four processors finish their jobs while the fifth still has 100 resampled datasets to generate, the work of generating those 100 datasets should be distributed across all the remaining idle processors. If this were implemented it should speed up run time and eliminate some of the anomalies seen when comparing the length of time it takes to run trueTree using different numbers of cores on a single machine.

Finally over the course of this research there have been a number of suggestions made about possible changes in how various components of the validation process should work. One suggestion

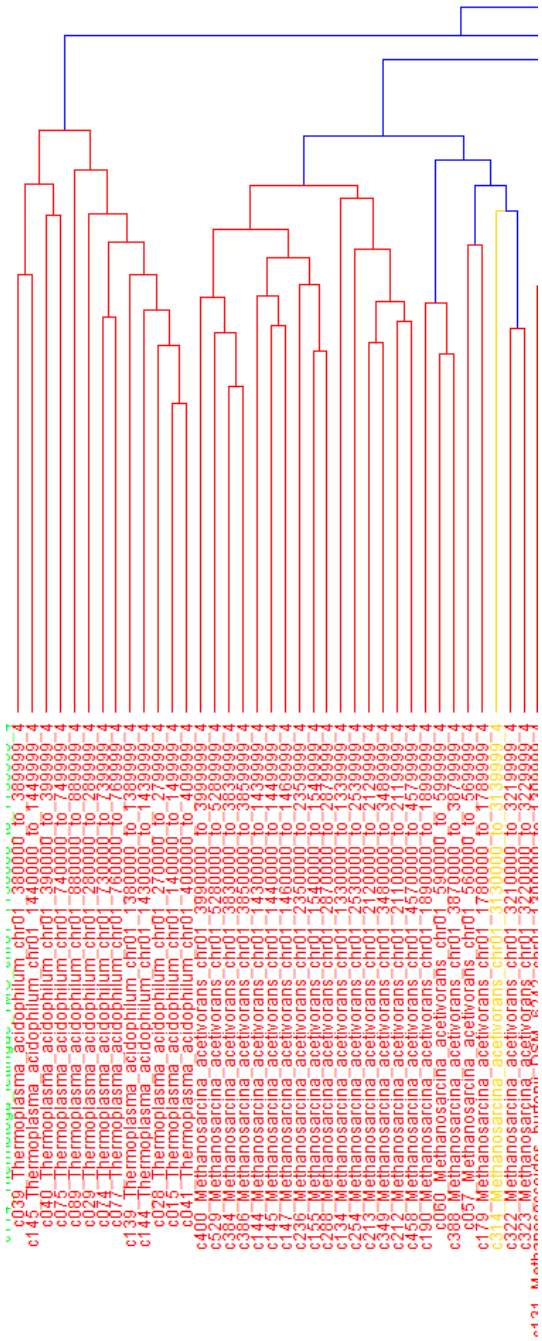

Figure 4.7: Section of the dendrogram of 18,637 genomics chunks, showing the location of chunk 314 from the genome of the *Methanosarcina acetivorans* organism. Red indicates that a chunk belongs to an organism in the archaea kingdom or that a clade only contains chunks belonging to organisms in the archaea kingdom. Green indicates that a chunk comes from a organism belonging to the bacteria kingdom or that a clade only contains chunks from organisms belonging to the bacteria kingdom. Blue indicates a clade containing chunks from more then one kingdom. Yellow indicates a chunk that both classifiers always classified wrong.

was to try permuting the datasets instead of resampling them to generate new datasets during the bootstrap process. This means that for each unique word in the dataset the word counts for all the chunks would be subject to permutation, so they would be shuffled. Resampling tests the theory that the dendrogram initially generated is the correct representation of the data. Permutation in contrast would test the hypothesis that the dendrogram has no connection to the underlying dataset and as such shuffling the word counts would not impact the results.

Appendix A Dendrogram Coloring

When working with datasets with large numbers of chunks the resulting dendrogram can be difficult to read. To help solve this problem trueTree includes an algorithm for coloring the dendrogram in order to make it easier to pinpoint the important information. Currently the code is only designed to handle a single specific type of genomics dataset. This dataset contains counts of the number of times each possible tetramer or 4mer, an ordered sequence of 4 base pairs, was found in a sample of an organism's DNA. In our trials the specific organisms used are microorganisms from the Bacteria and Archaea kingdoms. The information for each organism is divided into a number of chunks with each chunk containing the "genomic signature" for that chunk, that is frequencies of 4mers over a sequence of ten thousand base pairs from the organism's genome. The algorithm is designed to show the difference in how the chunks belonging to organisms in the Archaea kingdom and the chunks belonging to organisms in the Bacteria kingdom cluster together. As such each clade in the dendrogram is given a color, red if it only contains Archaea chunks, green if it only contains Bacteria chunks, and Blue if it contains both types of chunks.

The format for the input file containing the genomics data is similar to the one used when using trueTree for text mining. The first column in the input file contains the names for each chunk, followed by one column for each word with the words in this case being the 4mers. However after the last 4mer is listed, there are 8 additional columns containing metadata that give additional information about the organisms. Most importantly the third such column gives the kingdom (Archaea or Bacteria) of the organism.

After the input file is read by trueTree the metadata is removed from the dataset and stored in a separate table so it can be used when the dendrogram is plotted.

pvclust comes with a function to plot a dendrogram that shows the au values that have been calculated for every clade. The dendrogram itself however is drawn by calling the plot function for hclust objects using the hclust object that holds the original clustering. This is problematic because the hclust plot function does not provide the ability to change the color of individual clades. R however includes a dendrogram object which is another type of object that can store a dendrogram, and includes a function to convert from an hclust object to a dendrogram object. Each node in a dendrogram object can be given it's own individual color for display when plotted. However the only way to modify this color is using the `dendrapply` function which applies a function to each node in the dendrogram. This creates another problem since the `dendrapply` function does not provide enough information to determine what node the function is currently being applied to. This can be fixed by taking advantage of the fact that the `dendrapply` function recursively traverses the tree representing the dendrogram to determine the order of nodes to apply the function to, and as such the order is predictable. As such it is possible to create a list of instructions to implement for each node and then use an external counter to keep track of which node is currently being checked, allowing the dendrogram to be colored correctly. This process will be explained in more detail in the next few sections.

A.1 Creating the Color List

A node in a dendrogram object can represent a few different things. The leaves, containing each individual chunk, count as individual nodes. A clade containing two subclades consists of two nodes one node representing the line connecting the first subclade to the overall clade, and the other representing the line connecting the other subclade. A clade connecting a subclade and an individual chunk only has one node which connects the subclade to the overall clade. A clade connecting two individual chunks doesn't have any unique nodes of it's own. The only nodes present are the nodes representing the individual chunks. Each node in a clade is either a left node or a right node depending on whether they are connecting to a subclade to the left or the right of the overall clade in the dendrogram. Finally there is a node that holds a line going off the top of the topmost clade. The parameters trueTree uses when plotting a dendrogram make it so this node doesn't show up as part of the plotted dendrogram, but the node still exists.

The `dendrapply` function travels through the dendrogram recursively. It starts at the top of

the dendrogram. For each clade it visits the left node if it exists, recursively visits the chunk or clade to the left, visits the right node if it exists, and then visits the chunk or clade to the right. Figure A.1 shows an example dendrogram marking each node and the order in which dendrapply would visit them.

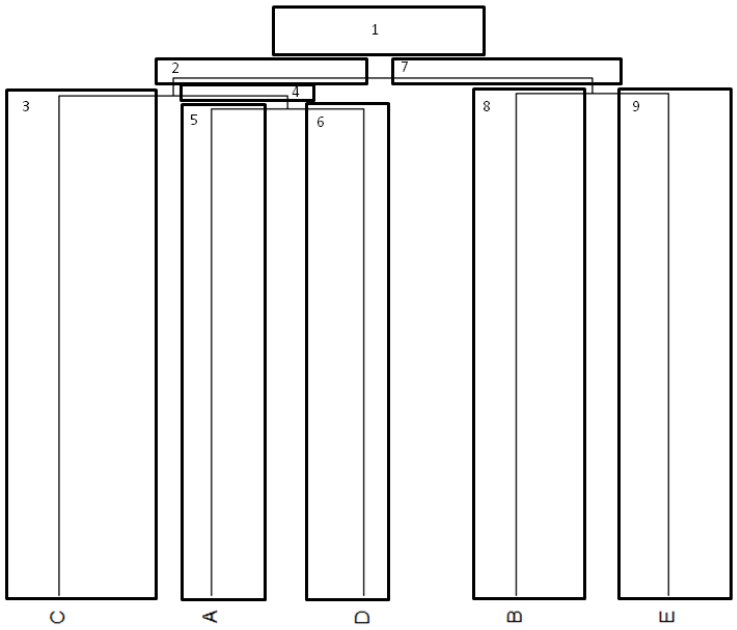

Figure A.1: Sample dendrogram showing how `dendrapply` goes through the nodes. Each node is outline in a box, and the numbers in the boxes give the order `dendrapply` visits the nodes

Each node needs to be individually assigned a color, and the first step is to create a list with an entry for each node noting the color for that node. In trueTree this list is generated using a recursive function that traverses through the dendrogram in a fashion similar to that used by `dendrapply` by using the merge table of the hclust object, discussed in section 3.2.2 that holds the structure of the dendrogram. The code to do so is shown below is Listing A.1.

```
#generates a list containing the color for every node in the tree
#the rules are a node is red if it only contains Archaea, green if it
#only contains Bacteria, Gold if it was specially selected for
#highlighting, and blue if it contains chunks that match multiple
#categories. The list is ordered in the order that nodes are visited
#by dendrapply.
generateLineColorList <- function(x, mergeTableRow,
    metaTable = NULL)
{
    colorlist <- list()

    #find the color of the left half of the clade
    if(x$merge[mergeTableRow,1] < 0) #if the left node is a chunk
                                     #determine the chunk's color
    {
        leftColor <- getColor(
            x$labels[-x$merge[mergeTableRow,1]],
            metaTable = metaTable) #the color of the chunk
        leftList <- list(leftColor) #list of the colors of all
```

```
                                    #the nodes to the left
    }

    else #if the left node is a clade recursively run the
         #function on that clade
    {
        result <- generateLineColorList(x,
            x$merge[mergeTableRow,1],
            metaTable = metaTable)
        leftColor <- result$color #the overall color of the
                                  #subclade
        leftList <- result$colorList #list of the colors of all
                                     #the nodes to the left
    }

    #find the color of the right half of the clade
    if(x$merge[mergeTableRow,2] < 0) #if the right node is a
                                     #chunk determine the
                                     #chunk's color
    {
        rightColor <- getColor(
            x$labels[-x$merge[mergeTableRow,2]],
            metaTable = metaTable) #the color of the chunk
        rightList <- list(rightColor) #list of the colors of all
                                      #the nodes to the right
    }

    else #if the right node is a clade recursively run the
    #function on that clade
    {
        result <- generateLineColorList(x,
            x$merge[mergeTableRow,2],
            metaTable = metaTable)
        rightColor <- result$color #the overall color of the
                                   #subclade
        rightList <- result$colorList #list of the colors of all
                                      #the nodes to the right
    }

    #check if the colors of the two subclades of the current
    #clade are the same
    if(leftColor == rightColor)
    {
    color <- leftColor #if so use the color they share
    }

    else #if the colors are different the subclades
         #have different contents
    {
    color <- "blue" #set the clade to blue to mark it's
                    #mixed contents
    }

    #the colors found need to be put together in the proper .
```

```
    #order The current clade has one node for each of it's
    #children which contain a clade instead of just a chunk.
    #Those nodes need to be given the color of the current
    #clade, but only if they exist. These nodes will appear
    #in the list of colors before all the colors for the
    #nodes in the respective subclades

    #check if both childern are subclades
    if(x$merge[mergeTableRow,1] > 0  && x$merge[mergeTableRow,2]
        > 0)
    {
    #both nodes in the current clade exist so add them
    #into the color list
    colorList <- c(color, leftList, color, rightList)
    }

    #check if the right child is a chunk
    else if(x$merge[mergeTableRow,1] > 0)
    {
    #there is only a node for the left clade
    #so add that to the color list
    colorList <- c(color, leftList, rightList)
    }

    else if{x$merge[mergeTableRow,2] > 0} #the left child is a chunk
    {
    #there is only a node for the right clade
    #so add that to the color list
    colorList <- c(leftList, color, rightList)
    }

    else #both children are individual chunks
    {
                colorList <- append(leftColor, rightColor)
    }

result <- list(colorList=colorList, color=color)
return(result)
}

#figure out what color each node should be
colorList <- generateLineColorList(hclustObj, dim(hclustObj$merge)[1],
    metaTable = metaTable)
#the first node checked be dendrapply is not part of the visible
#dendrogram so add a dummy value at the start of the list
colorList <- c(0, colorList$colorList)
```

Listing A.1 Code to generate a list of colors to use to color a dendrogram. hclustObj is the hclust object holding the dendrogram to color.

`getColor(x, metatable)`. trueTree function that takes the number of a chunk, and looks up the chunks metadata to determine what color the chunk should be displayed on the dendrogram. The current implementation colors a chunk green if it's from an organism in the bacteria kingdom, and red if it's from an organism in the archaea kingdom.

x - the chunk number to find the color for
metatable - Table containing metadata for every chunk

The generateLineColorList function is run once per line of the merge table. The initial function call seen on line 115 runs the function on the last row of the merge table which stores the topmost clade in the dendrogram. When generateLineColor is run the first part of the function from lines 12 to 57 looks at the specified row of the merge table to see if the children of that clade are individual chunks, or subclades. When the function finds a subclade it recursively calls itself on the row of the merge table holding that subclade. This recursive call will return a list of colors for all the nodes within the subclade, as well as the color of the node directly connected to the current clade. If instead one of the children of the current clade is an individual chunk, the color for that chunk is determined directly.

Next from lines 59 to 71 the colors from the left and right children need to be compared. The colors are an indication of the contents of the clades, so if they are the same color then both of the children have the same type of content, meaning the overall clade consists of that type of content, and the color of those clades can be safely used. If on the other hand the two colors are different then one clade contains chunks from Bacteria, and the other from Arcaea and the clade needs to be colored blue to reflect the mixed contents.

Finally the last section of the function from lines 85 to 108 puts together an actual list of colors for the current clade. Since the number of nodes that are part of the current clade varies depending on the contents of that clade's children there are four separate cases necessary to handle all the possible combination of subclades and individual chunks. Whenever there is a subclade the current clade has a node that is visited before the subclade. When there is an individual chunk instead of a subclade, the current clade has no node that is visited before that chunk.

The last line of Listing A.1 adds a dummy value to the start of the colors list. As was previously mentioned a dendrogram object has a node above the top of the topmost clade that is not actually drawn by trueTree. This node is the first to be visited so it needs to be assigned a color so the colorList directly corresponds to the order the nodes are visited, but since it is not drawn the color does not matter.

A.2 Applying The Colors

The `dendrapply` visits every node in the dendrogram. Each of those nodes has a set of parameters for the edge connecting it to the rest of the dendrogram, and a set of parameters for the node itself. The edges have one parameter that needs to be changed, col, that dictates the color of the edge. The node itself has two parameters that need to be updated. First is the pch parameter. This parameter dictates a shape which will be drawn on the dendrogram at the location of each node. This behavior is not desired so it needs to be set to NA. Secondly there is the lab.col that sets the color of the labels which need to be changed to match the edges. The code that handles these modifications is shown in Listing A.2

```
1  #set currentNode to 1 a global variable
2  #to keep track of where in the tree we are
3  assign("currentNode",  1, envir = .GlobalEnv)
4
5  lineColor <- function(x, colorOfNodes)
6  {
7          attr(x, "nodePar") <- list("pch"  = NA, "lab.col"  =
8              colorOfNodes[[currentNode]])
9          attr(x, "edgePar") <- list("col"  =
10             colorOfNodes[[currentNode]])
11         assign("currentNode",  currentNode + 1, envir
12             = .GlobalEnv)
13
```

```
14          x #this line is necessary for the dendrapply function
15            #that calls this to work
16  }
17
18  dend <- dendrapply(dend, lineColor,
19       colorList) #add color to all the nodes in the tree
```

Listing A.2 R code to change the colors of a dendrogram. dend is a dendrogram object holding the dendrogram to be plotted, colorList is a list of colors generated in Listing A.1

dendrapply(x, FUN, ...). R function that applies the FUN function to every node in the dendrogram x

x - the dendrogram to apply the function to.

FUN - The function to apply

... - Additional parameters for the FUN function

assign(name, value, environment). R function that assigns a value to a name in a given environment. Can be used to create global variables.

name - the name to assign the value to

value - the value to assign

envir - The environment the assignment should take place in. .GlobalEnv means the environment is global.

The third line sets up a global variable that keeps track of the progress made going through the dendrogram since `dendrapply` does not provide enough information to determine the exact location from within the lineColor function call. Then for each node the global variable is used to determine which color the node should be, the appropriate parameters are updated, after which the global variable is updated so it's ready for the next node. The last line shows the actual function call to `dendrapply`

A.3 Plotting the Dendrogram

Once the colors are set the dendrogram can be plotted. Dendrogram objects have a built in plot command, that does most of the work, and produces a plot similar to the one created from an equivalent hclust object. However Dendrogram objects do not automatically scale the size of the dendrogram to ensure that none of the labels for the chunks get cut off by the bottom of the plot. The solution is to change the margin settings. The margins are a set of 4 parameters that define the borders in which a plot can be drawn. By setting the margins so the bottom margin is substantially higher then the bottom of the display window, there will be plenty of empty space for the dendrogram to place the labels in when they inevitably overflow the bounds it tried to place them in. The code to do the plotting including the function calls seen in Listings A.1 and A.2 is shown in Listing A.3.

```
1   dend <- as.dendrogram(hclustObj) #convert the hclust object
2                                    #into a dendrogram object
3   #figure out what color each node should be
4   colorList <- generateLineColorList(hclustObj, dim(hclustObj$merge)[1]
5       , metaTable = metaTable)
6   #the first node checked be dendrapply is not part of the visible
7   #dendrogram so add a dummy value at the start of the list
8   colorList <- c(0, colorList$colorList)
9
10  #set currentNode to 1 a global variable
11  #to keep track of where in the tree we are
12  assign("currentNode",  1, envir = .GlobalEnv)
13
```

```
14 dend <- dendrapply(dend, lineColor,
15      colorList) #add color to all the nodes in the tree
16
17 #find length of longest chunk name
18 maxL <- max( nchar( x$hclust$labels ))
19
20 # set margins so there is just enough room for the labels
21 # The numbers measure margin size in line units
22 # The paramets are the size of the bottom,left,top,right margins
23 # On average a margin one line wide seems to have room for about 2.5
24 # characters so the margin on the bottom is set to the number of lines
25 # necessary to display the longest label if there was only 2
26 # characters per line which leave's a decent buffer
27 par( mar=c((maxL / 2.0), 2.1, 4.1, 2.1))
28
29 plot(dend)
```

Listing A.3 R code to plot a dendrogram utilizing trueTree's graph coloring capability. hclustObj is the initial hclust object. metaTable is the table holding the metadata for it's contents.

`as.dendrogram(x)`. R function that converts an object into a dendrogram object.
x - object to convert
`par(...)`. R function that sets graphical parameters given in the format tag = value where tag is the name of the parameter and value is the value to set it to. mar stands for margin and sets the bottom, left, top, and right margins in the graphical window.
... - List of parameters to set.

The code converts the initial hclust object into a dendrogram object. Then it determines the color for each edge and sets the colors of the dendrogram as seen in Listing A.1 and A.2. Next it finds the longest label out of all the chunk labels, and uses the length of that label to estimate the amount of room needed at the bottom of the graph and uses that value to set the margins. Finally the dendrogram is plotted, with the plot being appropriately colored.

Appendix B Setting Up a Machine Cluster in Linux

Setting up a machine cluster in Linux is a five step process. The first three steps need to be performed once on each computer in the cluster, while the last two only need to be performed on a single machine from which trueTree will be run. This single machine is the master node, while the other computers are slave nodes. All of the machines need to be running the same version of R, and share the same architecture i.e. all the machines must be running the 32-bit version of Linux or all the machines must be running the 64-bit version.

The first step is to install LAM/MPI (Squyres and Lumsdaine, 2003), an implementation of the MPI library standard for distributing workload across multiple computers. This needs to be done on all the machines in the cluster. This can be done on the command line with the following command

```
sudo apt-get install lam4-dev
```

This command tells Linux to install the package lam4-dev. Once the command is entered, Linux will first ask for the password for the current account being used, and then it will list additional packages that need to be installed alongside lam4-dev and ask if it should proceed, with the correct answer being "yes". It is possible that upon entering the password an error will be returned that says the account does not have the proper permissions in which case it would be necessary to log in as an administrator account.

The second step is to setup R (R Development Core Team, 2011) on all the machines in the cluster. If R is already installed upon a machine it can be launched by entering `R` on the command line. If it's not installed R can be installed by entering

```
sudo apt-get install r-base-core
```

Once R is launched there are three packages that need to be installed from within, with those being the snow, snowfall, and Rmpi libraries. The following R command will install all three.

```
install.packages(c("snow", "snowfall", "Rmpi"))
```

Upon entering this command, a window will pop up asking which mirror site to use. The packages will install properly no matter what mirror is selected but selecting a mirror located geographically close to where the Linux machine is located will provide the fastest speed. If this is the first time a package has been installed in R, R will ask if it can create a personal library to store the packages, and should be told "yes". Once R is set up, it possible to return to the Linux command line by entering `q()`. R will ask if it should save the current workspace, but it is safe to discard the workspace.

The third step is to get the Internet Protocol(IP) addresses of all the computers in the cluster. In Linux, the IP address can be determined by entering the following on the command line

```
/sbin/ifconfig
```

This command causes Linux to give the current status of its network connection. The status is divided into several sections. One section will be labeled `lo` which stands for local loopback and can be ignored. The other sections are determined by how the machine is connected to the network. If the computer is connected by Ethernet, then there will be a section labeled `eth0` for that connection. Other methods of connection will generate their own similar sections. Once the section corresponding to the correct connection is found the IP address is the value labeled `inet addr:` in that section, as shown in in Listing B.1

```
eth0      Link encap:Ethernet  HWaddr 00:1a:a0:a7:52:44
          inet addr:155.47.45.51 Bcast:155.47.45.255  Mask:255.255.255.0
          inet6 addr: fe80::21a:a0ff:fea7:5244/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:267561 errors:0 dropped:0 overruns:0 frame:0
```

```
          TX packets:27180 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:54727329 (54.7 MB)  TX bytes:6377225 (6.3 MB)
          Interrupt:16

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:144 errors:0 dropped:0 overruns:0 frame:0
          TX packets:144 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:13710 (13.7 KB)  TX bytes:13710 (13.7 KB)
```

Listing B.1 Sample Output from /sbin/ifconfig. In this particular example the IP address is 155.47.45.51

Once the IP address is determined for each computer, that address needs to be stored in a host file so that LAM/MPI can use the address to start up the cluster. While the host file can be built on any machine it will eventually need to be moved to the master. The format of a host file is simple. Each line in the file corresponds to one of the machines in the cluster. Each line lists the IP address of the corresponding machine delimited by a space then cpu=N where N is the number of cores in that machine to use as part of the cluster. The host file needs to contain information on the master node as well as the slave nodes. The process is shown below in Figure B.1.

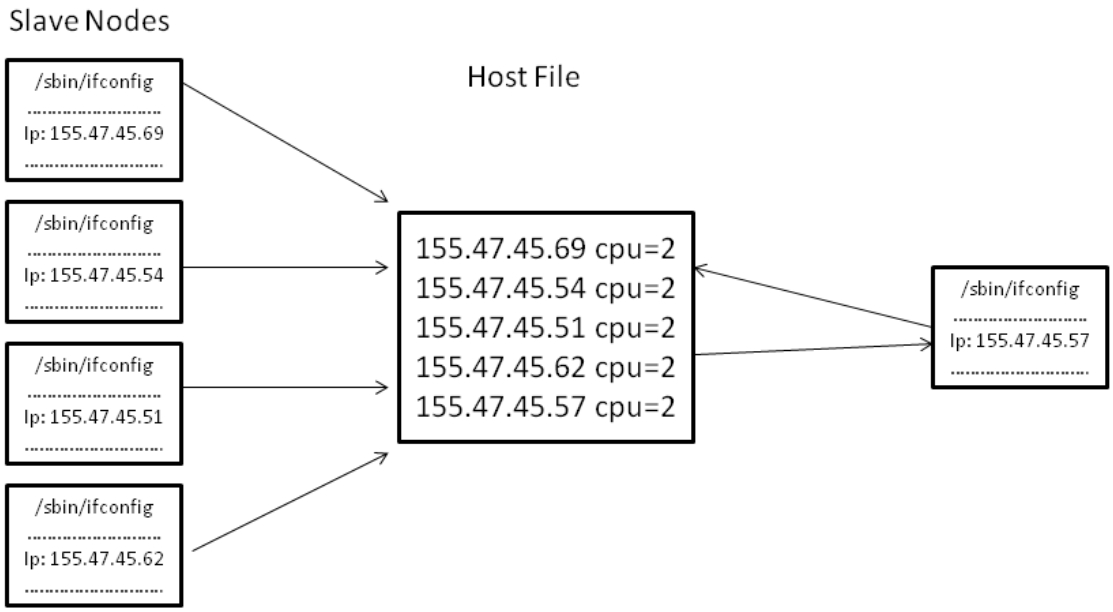

Figure B.1: Process of creating the host file. The host file takes the IP address of every machine in the cluster before finally being stored on the master node

Once the host file contains the information for all the machines in the cluster it is possible to boot up the cluster and run trueTree. However, at this point, booting up the cluster requires entering a password for every core of every processor used in the cluster. There is a way to get around this annoyance by setting up a shared key. The basic idea is to create a RSA private public key pair.

The machine that opens the ssh session will have the private key while the machine that the session is being opened on has the public key. The former machine signs a message with the private key that the latter machine can verify using the public key to confirm the former's identity and allow it to connect without prompting for a password. The exact procedure will depend on what ssh client is used. For open SSH 5.5 there is a three step procedure. First of all, the shared key needs to be created on the master node. The following command needs to be entered on the command line:

```
ssh-keygen -t rsa
```

The ssh client will ask for a location to store the key and a passphrase. At the location prompt just hit enter to store the key in the default location. If for some reason the location needs to be changed, some future commands will need to be modified to reflect the different location. The passphrase is a password that needs to be entered before the ssh client can access the file storing the private key. If the extra security is not desired it can be left blank. The next step is to copy over the public key to all the other computers in the cluster. This can be down with the following two commands

```
ssh accountName@IP mkdir -p .ssh
cat .ssh/id_rsa.pub | ssh accountName@IP 'cat >> .ssh/authorized_keys'
```

where accountName is the name of the user account the ssh client should access and IP is the IP address of the machine to be accessed. Upon entering these two commands there will be a prompt to enter the password for the machine being connected to. This is the only time entering the password will be necessary. The first command creates the default folder to store public keys for the ssh client if it hasn't already been created. The second command copies the file containing the public keys on the master and appends it to the end of the file containing public keys on the other machine creating said file if it doesn't already exist.

Finally the machine cluster can be booted up. On the master, navigate on the command line to the directory containing the host file and enter the following command:

```
lamboot -v hostfilename
```

where hostfilename is the name of the host file. This will start the LAM/MPI runtime on all the machines in the cluster. Once the runtime has been started, running trueTree will make use of the cluster assuming it has the proper parameters set. Those parameters are clusterType which has to be set to "MPI" and numCPUs which needs to be set to the number of processors in the cluster. After finishing using trueTree, the command `lamhalt` should be entered onto the command line of the master which shuts down the LAM/MPI runtime on all the computers in the cluster freeing up resources for other uses.

Appendix C Lab Setup

trueTree was designed to run on the computers in the CS lab at Wheaton College in Norton Massachusetts. The lab has 17 computers but only 11 are used to form the machine cluster due to the fact that some of the computers needed to be reserved for other uses. The computers all have identical technical specs shown in Listing C.1

```
Intel core 2 6600 @ 2.40 GHz
4 GB Ram 667 MHz
Barracuda 7200.10 160gb 7200rpm Hard Drive
256MB ATI Radeon X1300Pro Graphics Card
Ubuntu version 10.10
LAM/MPI 7.1.4
Open SSH 5.5
R 2.12.1
```

Listing C.1 Technical Specifications for Lab Machines

References

Adnan, R., Graaf, B., van Deursen, A., Zonneveld, J., and Zonneveld, J. (2008). Using cluster analysis to improve the design of component interfaces. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 383–386, Washington, DC, USA. IEEE Computer Society.

Buckley, T. R., Simon, C., Shimodaira, H., and Chambers, G. K. (2001). Evaluating hypotheses on the origin and evolution of the New Zealand alpine cicadas (*Maoricicada*) using multiple-comparison tests of tree topology. *Molecular Biology and Evolution*, 18(2):223–234.

Drout, M. D. C., Kahn, M. J., LeBlanc, M. D., and Nelson, C. (2011). Of dendrogrammatology: Lexomic methods for analyzing relationships among Old English poems. *Journal of English and Germanic Philology*, 110(3):301–336.

Dyer, B. D., Kahn, M. J., and LeBlanc, M. D. (2007). Classification and regression tree (CART) analyses of genomic signatures reveal sets of tetramers that discriminate temperature optima of archaea and bacteria. *Archaea*, 2(3):159–167.

Everitt, B. S., Landau, S., Leese, M., and Stahl, D. (2011). *Cluster Analysis*. John Wiley and Sons, Chichester, UK, 5th edition.

Farris, J. S. (1969). On the cophenetic correlation coefficient. *Systematic Biology*, 18(3):279–285.

Forsyth, R. S. (1997). Towards a text benchmark suite. In *Joint International Conference of the Association for Computers and the Humanities and the Association for Literary and Linguistic Computing*, page 73, Kingston, Ontario. Association for Computers and the Humanities.

Handl, J., Knowles, J., and Kell, D. B. (2005). Computational cluster validation in post-genomic data analysis. *Bioinformatics*, 21(15):3201–3212.

Herlihy, M. and Shavit, N. (2008). *Art of Multiprocessor Programming*. Morgan Kaufmann, Burlington MA.

Hogg, R. V., McKean, J. W., and Craig, A. T. (2005). *Introduction to Mathematicial Statistics*. Pearson Prentice Hall, Upper Saddle River NJ, 6th edition.

Jockers, M. L. and Witten, D. M. (2010). A comparative study of machine learning methods for authorship attribution. *Literary and Linguistic Computing*, 25(2):215–223.

Jones, A. C. (2011). divitext: Visualizing text segmentation for text mining. Undergraduate Honors Thesis, Wheaton College, Norton, MA.

Joseph, D., Ang, S., and Slaughter, S. (2005). Identifying the prototypical career paths of IT professionals: a sequence and cluster analysis. In *Proceedings of the 2005 ACM SIGMIS CPR conference on Computer personnel research*, SIGMIS CPR '05, pages 94–96, New York, NY, USA. ACM.

Knaus, J. (2010). *snowfall: Easier cluster computing (based on snow)*. R package version 1.84.

Kononenko, I. and Kukar, M. (2007). *Machine Learning And Data Mining*. Horwood Publishing, Chichester, UK.

Kraus, L. E. (2010). Using cluster analysis to identify relationships between Old English poems. Undergraduate Honors Thesis, Wheaton College, Norton, MA.

Larose, D. T. (2005). *Discovering Knowledge in Data: An Introduction to Data Mining*. Wiley, Hoboken, New Jersey.

LeBlanc, M. D., Baldwin, E., Hitchens, K., Bass, D., Kahn, M., and Dyer, B. (2012). Classifying stages of retention and loss of DNA acquired by horizontal transfer between bacteria and archaea. In *Proceedings of BIT's 3rd World DNA and Genome Day*, page 73, Xi'an, China. BIT.

McCallum, A. K. (2002). MALLET: A Machine Learning for Language Toolkit. `http://mallet.cs.umass.edu/`.

Microsoft (2011). Tcp/ip raw sockets. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms740548%28v=vs.85%29.aspx`.

Oak Ridge Natural Laboratory (2011). Pvm: Parallel virtual machine. `http://www.csm.ornl.gov/pvm/`.

Quinn, M. J. (2003). *Parallel Programming in C with MPI and OpenMP*. McGraw Hill, New York City, New York, International edition.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

Russell, S. and Norvig, P. (2003). *Artifical Intelligence A Modern Approach*. Prentice Hall. Pearson Education Inc, Upper Saddle River, New Jersey, 2nd edition.

Scientific Computing Associates (2009). Networkspaces. `http://www.lindaspaces.com/products/NWS_overview.html`.

Shimodaira, H. (2004). Approximately unbiased tests of regions using multistep-multiscale bootstrap resampling. *The Annals of Statistics*, 32(6):2616–2641.

Squyres, J. M. and Lumsdaine, A. (2003). A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy. Springer-Verlag.

Suzuki, R. and Shimodaira, H. (2011). *pvclust: Hierarchical Clustering with P-Values via Multiscale Bootstrap Resampling*. R package version 1.2-2.

Tierney, L., Rossini, A. J., Li, N., and Sevcikova, H. (2011). *snow: Simple Network of Workstations*. R package version 0.3-7.