University of Rhode Island
DigitalCommons@URI

Open Access Master's Theses

2021

DATA DIMENSIONALITY REDUCTION THROUGH CLUSTER TREES AND MANIFOLD LEARNING

Ali Amani
University of Rhode Island, ali_amani@uri.edu

Follow this and additional works at: https://digitalcommons.uri.edu/theses

Terms of Use
All rights reserved under copyright.

Recommended Citation
Amani, Ali, "DATA DIMENSIONALITY REDUCTION THROUGH CLUSTER TREES AND MANIFOLD LEARNING" (2021). *Open Access Master's Theses.* Paper 1941.
https://digitalcommons.uri.edu/theses/1941

This Thesis is brought to you by the University of Rhode Island. It has been accepted for inclusion in Open Access Master's Theses by an authorized administrator of DigitalCommons@URI. For more information, please contact digitalcommons-group@uri.edu. For permission to reuse copyrighted content, contact the author directly.

DATA DIMENSIONALITY REDUCTION THROUGH CLUSTER TREES AND MANIFOLD LEARNING

BY

ALI AMANI

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2021

MASTER OF SCIENCE THESIS

OF

ALI AMANI

APPROVED:

Thesis Committee:

Major Professor

Noah Daniels

Gretchen Macht

Marco Alvarez

Brenton DeBoef
DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND
2021

ABSTRACT

Dimensionality reduction algorithms are a commonly used solution to create a visual summary of high dimensional data in a way that makes identification of patterns and trends easier. Algorithms that are used to visualize data as 2 or 3 dimensional plots are popular options, even more so due to clustering and manifold learning. There already exist many tools, both linear and nonlinear, that are used in visualizing high dimensional data, three of the most popular being PCA, t-SNE and UMAP. PCA has low memory requirements and is efficient in low dimensions, t-SNE captures much of the local structure of high dimensional data while also revealing factors like presence of clusters, and UMAP has no computational restrictions on embedding dimension.

Despite each of their respective advantages, all three of these tools have noticeable drawbacks. t-SNE and UMAP both have hyperparameters which require tuning to get visualizations of any value. PCA cannot recover nonlinear structure, so there can be significant loss of the global structure when applying that algorithm to data. These drawbacks prompt the development of new (mostly nonlinear) tools for visualizing high dimensional data. The reason for which we would want to visualize high dimensional data in the first place is because humans are incapable of seeing in more than three dimensions. Reducing the dimension of high dimensional data enables us not only to view the data, but to notice patterns and easier detect anomalous data points.

Manifold learning is one approach to getting a simplified low dimensional version of higher dimensional data. This machine learning tool is used in the visualization of high dimensional data by describing these datasets as low dimensional manifolds embedded into

higher dimensional space. Clustering is a machine learning approach that groups together individual data points in a way that provides value. Clustering simplifies a large high dimensional dataset by showing clusters, or organized groups of data points, rather than all the data points individually. Hierarchical clustering applies this principle by first organizing datasets into one large cluster, and then recursively dividing the current cluster(s) until a specific criteria is met that finds the optimal "level" of this process, or the optimal clusters which represent the dataset.

Clustering algorithms are usually more effective in lower dimensions due to the "curse of dimensionality", or the issues which arise when analyzing high dimensional data that do not occur in lower dimensions. For this reason, if we want to apply clustering algorithms to high dimensional data, we will be required to use dimensionality reduction first. This is a reason for which we would use manifold learning in tandem with hierarchical clustering, as it reduces the dimension of the data first to maximize the effectiveness of clustering.

When manifold learning and hierarchical clustering are used in unison, the result is a set of clusters from a dataset brought down to a lower dimension through manifold learning. These clusters, when taken from the manifold, are then able to be visualized easily in graph form. In this study, we will develop a tool to visualize high dimensional data by using hierarchical clustering and manifold learning together, but without actually reducing the dimension. Instead of using dimensionality reduction traditionally, we will visualize low dimensional summaries of high dimensional data. The summaries inferred from the data will give information about the manifold, such as connectedness between different parts of the manifold and how this connectedness changes through different stages of the

hierarchical clustering algorithm. These summaries will also give factors indicating the presence of possible anomalous data points.

To create and access these summaries, we will use Pyclam, the Python implementation of CLAM (Clustered Learning of Approximate Manifolds). CLAM is an existing dimensionality reduction tool that uses manifold learning and hierarchical clustering, and made primarily for anomaly detection. From the manifolds produced by CLAM, we will be able to access all the necessary properties needed to infer graphs. These graphs will be returned in our implementation in the form of a DOT file, a file format read by various software to produce a graphical representation. After we are able to produce working DOT files, we will use a visualization tool of our own design, implemented in Rust, to read these DOT files and display these graphs in a force-directed layout.

ACKNOWLEDGMENTS

I would like to express my gratitude to my Major Advisor, Dr. Noah Daniels, for his support, encouragement, and patience throughout this project. I am also grateful to both my Outside Committee Member, Dr. Gretchen Macht, and my Internal Committee Member, Dr. Marco Alvarez, for agreeing to be part of my Thesis and Defense Committees. A special thanks to Najib Ishaq, who helped me with implementation. I also would like to thank Dr. Bill Kinnersley for chairing my defense.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Dimensionality Reduction

Many datasets in machine learning almost always have a high dimensionality, meaning these datasets possess thousands of features. This also includes real-world data, such as speech signals, digital photographs, or fMRI scans. Having such a large number of features causes issues such as an extremely slow training process and difficulty in finding an effective solution. These issues are together known as the "curse of dimensionality", which dimensionality reduction techniques aim to solve. Dimensionality reduction is the transformation of high dimensional data into a lower dimensional representation that is typically more meaningful. Dimensionality reduction is important to many fields, since it uses the classification, visualization, and compression of high-dimensional data by reducing or simplifying the properties of high-dimensional spaces [3, 22].

1.2 Data Visualization

Data visualization is arguably one of the most important parts of dimensionality reduction, since it has the dimensionality of the given data drop down to two or three, making it possible to visualize the data on a plot. This opens the possibility of gaining important insights by analyzing the patterns and trends these plots can give us. Two major approaches exist for data visualization, projection and manifold learning. Projection translates every high dimensional data point onto a low dimensional subspace, approximately preserving the distances between the points. Manifold learning, which will

be explained in further depth in Subsection 1.2.1, relies on the assumption that most real-world high dimensional datasets lie close to manifold with a much lower dimension [22].

1.3 Importance of Graphs

The purpose of a graph is to present data, often too numerous and complicated to be described in text, in a way that visually illustrates relationships in the data [23]. Graphs are used extensively in computer science, as well as many other related fields. Many domains, such as social networks, molecular graph structures, biological protein-protein networks, recommender systems can be modeled as graphs. Relationships and interactions between individual units can be represented as edges connected between nodes representing the units. Graphs also play a key role in machine learning. When making predictions and discovering new, graph-structured data is used as feature information [24].

1.4 Our Solution

Using CLAM, an existing dimensionality reduction tool, we have been able to extract low dimensional summaries of high dimensional data in the form of manifolds. From these manifolds, we were able to iterate over the clusters recorded into it, and thus infer two types of graphs to display the data. The first graph, the Cluster Tree graph, visualizes the clusters in the manifold made by hierarchical clustering as a binary tree. The Manifold Layer graph, the second graph, visualizes a specified layer of the Cluster Tree, representing the dataset at a specified stage of the hierarchical clustering algorithm.

The algorithms used to produce these graphs were implemented in Python and added to Pyclam, the Python implementation of CLAM. These implementations would

return the graph in the form a DOT file, which is a file format able to be read by software to produce visualized graphs [12]. After the DOT files were produced, we would then use Graphviz, a DOT file visualization software, to test the file format for errors. After the DOT files were written to a correct format, we would observe their visualization and determine what features to add in order to make the graph more meaningful in terms of anomaly and pattern detection.

After being able to return usable DOT files from our implementations, we were able to use them to test our own graph visualization tool, which we implemented in Rust. This tool was made with the functionality of Graphviz in mind, such that it would read DOT files and produce a visual representation of the graph. Where we intended to make our Rust tool different in Graphviz was the graph's layout. Using an existing Rust library, force_graph [19], we would create a visually appealing force-directed graph from the DOT file.

CHAPTER 2
LITERATURE REVIEW

2.1 Current Data Visualization Tools

Reducing the dimensionality of data and visualizing it as a plot is not a new concept, as there already exist multiple tools for this purpose. Traditionally, dimensionality was reduced using linear techniques, of which PCA (Principal Components Analysis) was a popular option [3]. These tools reduce dimensionality through projection, translating the data into a linear subspace with minimum information loss [22]. However, if the data lies on a nonlinear submanifold of the feature space, then linear dimensionality reduction tools will overestimate the dimensionality, and thus give a much less accurate summary of the data. Since linear solutions are unable to adequately handle complex nonlinear data, there have been many recently proposed nonlinear tools for dimensionality reduction [1, 3, 15]. Existing tools used in dimensionality reduction includes PCA, t-SNE, and UMAP. Another and more recent one is CLAM, which we will be using in this project. CLAM uses a combination of hierarchical clustering and manifold learning to get a dimensionally reduced set of clusters, each cluster representing a group of similar data points.

2.1.1 PCA

PCA (Principal Component Analysis) is a linear dimensionality reduction tool that has found applications in fields such as facial recognition and image compression, and covers standard deviation, covariance, and eigenvectors. PCA works by identifying the hyperplane to which the data lies closest to, and then projects the data on that hyperplane, retaining most of the properties of the original dataset in the process [22]. PCA's strengths

include low sensitivity to noise, low capacity and memory requirements, and high efficiency in smaller dimensions. PCA's key weaknesses are its covariance matrix being difficult to evaluate in an accurate manner, and even simple invariances could not be captured by the PCA unless this information is supplied by the training data [18]. Another weakness of PCA comes with its nature as a linear tool for dimensionality reduction, as studies have shown that nonlinear techniques outperform their linear ones on complex tasks [3].

2.1.2 t-SNE

t-SNE is a nonlinear tool for visualizing high dimensional data, capable of capturing most of the local structure of high dimensional data while revealing global structure, such as the presence of clusters, at several scales. t-SNE takes a high dimensional data set and reduces it to a low dimensional graph while retaining much of the original data's information. This is made possible by giving each data point a location on a two or three-dimensional map, and finding clusters in data, making sure that this embedding preserves the original data's information [22]. Testing t-SNE on a variety of real-world data sets has been shown that it outperforms existing state-of-the-art techniques for visualizing data. t-SNE's weaknesses lie in its unclear performance on general dimensionality reduction tasks, its local nature making it sensitive to the differences between the inherent high dimensionality of data and its reduced dimensionality, and not guaranteeing convergence to a global optimum of its cost function [17].

2.1.3 UMAP

UMAP (Uniform Manifold Approximation and Projection) is a nonlinear dimensionality reduction tool, and is very effective in visualizing clusters data points along with their relative proximities [22]. UMAP constructs a topological representation in two steps: approximating the manifold containing the data, and building a fuzzy simplicial set representation of the approximated manifold. Strengths which UMAP has over t-SNE lie in visualization quality, global structure preservation, runtime performance, and ability to scale to significantly larger dataset sizes than t-SNE can handle. UMAP also has no computational restrictions regarding embedding dimension. Weaknesses of UMAP include lacking the strong interpretability of Principal Component Analysis, and approximations that need to be made for the sake of computational efficiency can have a negative impact on results produced from small datasets [16].

2.2 Manifold Learning

Manifold learning is a component of many dimensionality reduction tools, and describes datasets as low-dimensional manifolds that are embedded into high-dimensional spaces. It relies on the manifold hypothesis, or the assumption which holds that most real-world high dimensional datasets are “close” to a manifold of a much lower dimension [22]. In manifold learning, the manifold can be compared to a flat sheet, or a low dimensional object embedded in a higher dimensional world [4]. The dataset of interest is in this low dimensional manifold, where the low dimensional space reflects the underlying parameters, and the high-dimensional space is the feature space [6].

The manifold can be rotated, reoriented, and stretched to fit all data points in high dimensional space. No matter how the manifold is moved or altered to fit the data points, it still retains its low dimensional geometry while still embedding high dimensional space. Manifold learning is essentially the process of uncovering this manifold structure in a dataset, and also helps to visualize data using the low dimensional nature of the manifold as it is contorted to fill high dimensional space [4, 7].

2.3 Hierarchical Clustering

Clustering techniques are widely used in the analysis of large datasets to group together samples with similar properties, with data points bearing distinctive similarities being grouped in the same cluster [5, 8]. Cluster analysis divides data into multiple clusters in a way that is useful and meaningful to the user. In most cases, cluster analysis is the starting point for other purposes, such as, in this study, data summarization [5].

Cluster trees are introduced with hierarchical clustering, where a tree-like structure is used to allocate data points into leaf nodes [7]. Hierarchical clustering creates clusters that have a predetermined ordering from top to bottom, and is divided into top-down (divisive) and bottom-up (agglomerative) clustering. Divisive clustering begins by assigning all data points to one large cluster, and recursively dividing all existing clusters into two clusters until either a user-specified condition is met, or each cluster has only one data point. Agglomerative clustering works in reverse, initially having each data point as its own cluster, and then recursively joining two clusters based on similarity until either a user-specified condition is met, or the entire dataset is in one cluster. Each “hierarchy” of these two methods are the clusters produced in each recursive step. Evidence shows that

divisive clustering algorithms give more accurate hierarchies than bottom-up algorithms in some cases, but are conceptually more complex [9].

2.4 CLAM

CLAM (Clustered Learning of Approximate Manifolds) is an approach to dimensionality reduction that uses hierarchical divisive clustering to find a manifold in high dimensional space. Simply put, CLAM applies hierarchical clustering to a manifold learned from high dimensional data, a task made easier since the algorithm will be looking for simplified clusters rather than a wide range of data points scattered across high dimensional space [7]. The manifold learning component of CLAM is derived from CHESS (Clustered Hierarchical Entropy-Scaling Search), a hierarchical search algorithm that is used here to accelerate approximate search on high dimensional datasets [8, 9].

CLAM also induces a graph at each stage of hierarchical clustering, where a collection of five algorithms are put to work, all of which make up CHAODA (Clustered Hierarchical Anomaly and Outlier Detection Algorithms). CHAODA is meant to explore the various properties of each graph in the different cluster hierarchies to detect anomalies and outliers, and relies on optimal graphs at nonuniform depths, using machine learning to determine these depths [7]. CLAM infers graphs from hierarchical clustering and can do so at uniform depths. The method in which these graphs are produced can be found in Figure 2 of the Appendix. Figure 2 highlights each stage of hierarchical clustering, graphs as they are induced at each stage or “level” of the algorithm, and how these graphs are induced in the first place.

CHAPTER 3

METHODOLOGY

3.1 Data Sources

Many of the datasets we used in this project to get manifolds from CLAM are some of the same datasets used to originally test the CLAM implementation and were sourced from Outlier Detection Datasets [7]. These datasets were taken from the UCI Machine Learning Repository [25]. The size of these datasets ranged from just over 100 records to thousands of records, and with dimensions ranging from a simple 3 dimensional space to over 2,000 dimensions [7].

Two especially large datasets were also used to measure our project's success with datasets of high dimension and size. The first is APOGEE (Apache Point Observatory Galactic Evolution Experiment) from the Sloan Digital Sky Survey, each data point a stellar spectrum in the infrared band, with 8,575 bins, translating into datasets with a dimension of 8,575 [26]. The second is SILVA, an 18S ribosomal RNA dataset, whose multiple sequence alignment gave each sequence 50,000 bases long, translating into datasets with a dimension of 50,000 [27]. Each dataset used in this project can be found in Figure 1 of the Appendix, which a chart showing the name of every dataset, the number of data points, and their dimensional space. We ran these datasets through CLAM to extract manifolds of a given depth, and from this manifold we were able to infer graphs.

3.2 Inferring Graphs from the Manifold

There are two types of graphs which we produced in this project. The first is the Cluster Tree graph, which visualizes the clusters of the manifold as a binary tree, starting from a "root" cluster up to "leaf" clusters. The second is the Manifold Layer graph, which visualizes the dataset as clusters at a specific layer of the manifold, or a specified stage of hierarchical clustering. The following subsections explain how these graphs are inferred from the manifold.

3.2.1 Cluster Tree Graphs

A Cluster Tree graph takes the form of a binary tree, starting at the top with a "root" node and ending with "leaf" nodes. The root node, and every other node in the graph except for the leaf nodes, share edges with at most two "child" nodes that are both on the level of the tree directly under their parent node and ending at the bottom with "leaf" nodes. The Cluster Tree has uniform "layers" holding its clusters. Figure 2 of the Appendix provides a visual explanation of a Cluster Tree with a depth of four, meaning four layers of child nodes which are descended from the root node. The graph in Figure 2 starts with the root node at the top, connecting to its two child nodes. Both of the child nodes are on the same layer of the tree, and the children of these two nodes are also on the same layer.

Inferring a Cluster Tree graph from the manifold starts as CLAM applies divisive clustering to the dataset. At the first step of hierarchical clustering, every point on the dataset is assigned to one large cluster. Next, this cluster is divided into two clusters, and both of the two new clusters are divided into two new clusters as well. This division of existing clusters continues recursively until either every cluster contains only one data

point, or until the algorithm has reached a user-given limit of recursions. This limit, whether user specified or not, is the "depth" of the Cluster Tree. Every cluster that was formed by the algorithm is saved to the manifold, which allows us to retrieve our Cluster Tree graph after the hierarchical clustering is finished [7, 10].

The nature of the clusters as they are saved to the manifold help us to draw the manifold as a binary tree. From the manifold, we can get a "root" cluster, which is the original cluster we started with in hierarchical clustering and contains all the data points. As we start from the root cluster, we are able to access each of its "child" clusters, or the two clusters formed from the division of the root cluster that happens in hierarchical clustering. We are then able to access the children of the root's child clusters, and this process continues iteratively until we reach the "leaf" clusters, or clusters at the end of the tree that have no children. Each "layer" of the Cluster Tree, or the set of clusters at a certain depth of the manifold, represent the entire dataset clustered at one stage of CLAM's hierarchical clustering process. Cluster Tree graphs don't always have to start from the root cluster. When inferring a graph based on Cluster Trees, we can also get a subtree, treating a cluster that's between the root and the leaf clusters as the root of this new subgraph. The binary tree displaying all this cluster's descendents up to the leaf nodes gives us a Cluster Subtree graph.

The purpose of the Cluster Tree graph is to give a clear visualization of how certain properties of clusters change through each step of hierarchical clustering. Also, it is from the layers of the Cluster Tree from which we are able to infer the next type of graph, which is the Manifold Layer graph. The view of the Cluster Tree graph can also help the user find what depth of the Cluster Tree is optimal for the Manifold Layer Graph [10].

3.2.2 Manifold Layer Graphs

The Manifold Layer graph represents the dataset clustered at a particular stage of the hierarchical clustering algorithm of CLAM. This graph is not as neat as the Cluster Tree graph, and instead takes the form of a set of nodes whose connectivity and arrangement are much less uniform. However, if each node in the graph is properly labeled, then the Manifold Layer graph can offer a wealth of information. The image in Figure 2 of the Appendix gives a "cartoon" representation of Manifold Layer graphs inferred from the Cluster Tree. Underneath the binary tree in Figure 2, the blue nodes represent nodes from the Cluster Tree at specific depths, and which nodes in the binary tree they correspond to are denoted by grey arrows. The nodes in the Manifold Layer graphs are connected based on whether or not the two nodes' volumes overlap, which is shown on the Cluster Tree by the circles around the nodes overlapping. This is useful because the connectivity of the Manifold Layer graph shows us both the global and local structure of the manifold as it is occupied by data. Some of what the Manifold Layer graph shows us is if the manifold is continuous, if "holes" in the data are present, and if outliers exist.

The Manifold Layer graph is extracted from a specific layer of the Cluster Tree, since each layer of the Cluster Tree graph represents the entire dataset organized by the clusters in that particular layer, which is at a particular phase of hierarchical clustering. As CLAM is building the Cluster Tree, the algorithm builds a graph for each layer of the tree by creating edges between clusters with overlapping volumes. The Manifold Layer graph is one of these graphs at a specified depth of the Cluster Tree. CLAM builds the Cluster Tree up to the user-specified depth, and also produces a heterogenous layer graph based on

different depths for different parts of the tree. These depths are useful in anomaly detection [7].

The purpose of Manifold Layer graphs is that they allow us to get a closer look at the varying properties of clusters at different depths. These properties include cardinality, connectivity of clusters, outliers, and how often a cluster is visited by random walks on the manifold. CLAM also uses these graphs to detect outliers and anomalies in small, disjoint connected components of the graph. These disjoint components, or "islands" represent isolated regions of the manifold. These islands are more likely to contain anomalies due to their isolation from a majority of the other clusters and can be caused by one or few anomalous data points close enough to non-anomalous data points to form a cluster.

At the level of a Manifold Layer graph, several other properties unique to graphs alone can help us better understand the data. Connectedness of the nodes and the shortest distance of traversal between interconnected nodes can help us define relationships between different clusters. Radii between connecting clusters can be used to determine their "closeness", or similarity. Essentially, all the properties which 2D graphs have to offer can help us find out the properties of various sets of high dimensional data, with each situation offering new and unique insights.

Manifold Layer graphs are also analogous to "filtration" in computational topology. Filtrations are one of the key components in persistent homology, a method used to understand such shapes and their persistence in point clouds and networks. Filtration can be imagined as an embedded sequence of networks with some form of geometrical shape built from the edges and nodes in each sequence step. Filtration can also make certain

components visible, components which are persistent across a wide range of distances along with those that are artifacts of a particular distance. Using this principle, we can look at graph properties to see how long-lived they are [7, 21].

3.3 Our Algorithms

The algorithms meant to infer graphs and produce DOT files were implemented in Python, and were then added to three different classes in Pyclam depending on whether they would produce a Cluster Tree graph, a Cluster Subtree graph, or a Manifold Layer graph. All of these graphs are inferred after the Manifold has been built up to a specified depth. Implemented in Python, these three algorithms return a string in DOT format, meant to be written as a DOT file. From there, the produced DOT file can be read by graph visualization software, such as Graphviz [10].

3.3.1 Extracting the Manifold Layer Graph

The algorithm that infers the Manifold Layer graph operates from the specified layer of the Cluster Tree, accessing the clusters and edges built between them from the given layer alone. This algorithm reads each cluster and edge in the layer of the manifold iteratively and creates a graph from it. For each cluster, a node is initialized and added to the graph, with this new node given the appropriate properties derived from the cluster.

After we have gone over all the clusters and filled the graph with nodes, we bring our attention to edges. Edges, in a Manifold Layer graph, are made between two nodes whose clusters overlap. Each cluster possesses a radius, which is defined by the distance between the cluster's center and its furthest point, treating them as hyperspheres in this

instance. Two clusters whose radii overlap have their respective nodes connected by edges, where the weight of this edge, in our project, is set as the distance between the cluster centers.

Algorithm 1: produce DOT string for a Manifold Layer graph

```
get_manifold_layer_graph():
        manifold_layer_graph.initialize()
        for cluster in self.clusters:
          |     //Creating a graph node from a cluster's data
          |     new_node = derive_node(cluster)
          |     manifold_layer_graph.add_node(new_node)
        end
        for edge in self.edges:
          |     //Creating a graph edge between two nodes whose clusters overlap
          |     new_edge = derive_edge(edge)
          |     manifold_layer_graph.add_edge(new_edge)
        end
        manifold_layer_graph.finish()
        return manifold_layer_graph
```

3.3.2 Extracting the Cluster Subtree Graph

The algorithm that infers the Cluster Subtree graph operates by going over several "layers" of the manifold, or each stage of hierarchical clustering. The Subtree algorithm starts from one specified starting cluster in the manifold, iteratively accessing all descendants of the given cluster. The algorithm reads every cluster (the starting cluster and all its descendants) and from it creates a graph. For each cluster, a node is initialized and added to the graph, this new node given the appropriate properties derived from the cluster. Edges are added after, and edges in a Cluster Subtree graph are made between two nodes

that have a parent/child relationship. All nodes in this graph, save for the leaf nodes, have edges going to their child nodes.

Algorithm 2: produce DOT string for a Cluster Subtree graph

```
get_cluster_subtree_graph():
        subtree_graph.initialize()
        for cluster in self.subgraph:
        |       //Creating a graph node from a cluster's data
        |       new_node = derive_node(cluster)
        |       subtree_graph.add_node(new_node)
        |       for child in cluster.children:
        |       |       //Creating a graph edge between a parent and child node
        |       |       new_edge = derive_edge(child, cluster)
        |       |       subtree_graph.add_edge(new_edge)
        |       end
        end
        subtree_graph.finish()
        return subtree_graph
```

3.3.3 Extracting the Cluster Tree Graph

The algorithm that infers the Cluster Tree graph of the entire manifold operates from the manifold itself, accessing the subtree algorithm highlighted in Subsection 2.3.2, except using the root cluster as the starting cluster. This gives us the Cluster Tree of the entire manifold, starting from the root cluster and accessing all its descendants. The algorithm reads every cluster (the root cluster and all its descendants) and from it creates a graph. For each cluster, a node is initialized and added to the graph, this new node given the appropriate properties derived from the cluster. Edges are added after, and edges in a Cluster Subtree graph are made between two nodes that have a parent/child relationship. All nodes in this graph, save for the leaf nodes, have edges going to their child nodes.

Algorithm 3: produce DOT string for a Cluster Tree graph

```
get_cluster_tree_graph():
        root = self.root
        return root.get_cluster_subtree_graph()
```

3.4 Writing the DOT File String

When reading the manifold, accessing all the required elements of the graph was simply a matter of reading every cluster and every edge. After extracting the graph from the manifold, the next step was writing the graph to a string in DOT format. DOT files, when used with software such as Graphviz, are able to draw graphs either as graph files or in a graphics format such as GIF, PNG, SVG, PDF, or PostScript. Its features include well-tuned layout algorithms for placing nodes and edges, as well as applying labels, shapes, and colors to nodes and edges [12].

We took advantage of the features DOT format had to offer, since both clusters (represented as nodes in a graph) and edges between the clusters would have distinct properties we wanted to visualize with the graph. DOT files produced by the implemented algorithms were tested and read using Graphviz, an open source graph visualization software. Graphviz takes descriptions of graphs in a simple text language, and uses them to make diagrams in various formats. This was ideal for the testing of DOT files of cluster trees and cluster tree layers produced by our code [12].

3.4.1 Cluster Properties

In the implementation of our algorithms, properties of clusters that are used in visualizing the graph are the clusters' name, cardinality, radius, and local fractal dimension (LFD). A cluster's radius, or the distance between the outermost point of the cluster and the cluster's center, is used in determining whether the cluster overlaps with any of the other clusters. Cluster cardinality is equated with its "anomalousness", or the likelihood of the cluster containing points that are not outliers. A higher cardinality indicates a lower probability of a cluster containing outliers, and a low cardinality indicates a higher chance of a cluster having outliers. A cluster that contains few points is more likely to have outliers, so the cardinality of a cluster is proportional to the number of points it contains [7].

The LFD of a cluster is calculated using the ratio of the number of data points within two spheres of the same center point, the first sphere having the radius of the whole cluster and the second sphere having the radius of half that cluster [8]. The equation used to calculate a cluster's LFD can be found at Figure 3 of the Appendix. This equation captures the effect a shape's radius has on its area has. For example, if double the length of a line, which is one dimensional, doubles the amount of space it takes up. With that same principle, doubling the diameter or radius of a circle quadruples the amount of space the circle takes up. Doubling the radius of a sphere increases its area by 8×2^3. If the LFD is 1, then that means the number of points in the whole cluster is equal to the number of points in a sphere with a radius half of that of the whole cluster. The closer the LFD of a cluster is to 1, the higher likelihood there is for that cluster to contain anomalous points.

3.4.2 Calculating the Node Color from the LFD

All of the cluster's properties are written on the node representing the cluster as labels in the visualized graph. The LFD, however, is also used to define the color of the node representing the cluster. The algorithm used to determine all the clusters' colors happens before any graphs are derived, and first normalizes the LFD's of every cluster in the graph to values between 0 (the lowest LFD) and 1 (the highest LFD). The normalized LFD's are then used to calculate an RGB value (green hard-coded to zero), that is on a color gradient between blue (representing the lowest LFD) and red (representing the highest LFD). This RGB value is then converted to a Hexadecimal color string, a format that is better able to be read by Graphviz.

Algorithm 4: calculating the color of each cluster based on its LFD

```
lfds = []
for cluster in graph:
    lfds.append(cluster.lfd)
end
normalized_lfds = normalize(lfds)


for cluster in graph:
    r = 255 * normalized_lfds[cluster.lfd]
    b = 255 * (1 - normalized_lfds[cluster.lfd])
    cluster.color = hex( (r, 0, b) )
end
```

3.4.3 Edge Properties

Unlike clusters, edges between clusters are different when it comes to different types of graphs. In the implemented algorithm for Cluster Trees and Subtrees, edges

defined are directed edges going from parent clusters to child clusters, and are not given any labels. These edges denote the binary division of clusters as hierarchical clustering is applied. In the Manifold Layer graphs, the edges we included are all undirected edges which represent the clusters of connected nodes having overlapping volumes. These edges are also labeled by their distance.

3.4.4 DOT File Format

In DOT format, each line of the file, after the opening bracket and before the closing bracket, denotes a feature of the graph. Figures 4 through 7 of the Appendix show simple DOT files compared with the graphs they visualize through Graphviz. Figures 5 and 7 both show DOT files which contain graphs named after the dataset that CLAM read to produce them. Each of the DOT files begin by declaring the "type" of graph, meaning whether or not they are a directed graph. Then the edge properties are defined, followed by the node properties, and then finishing up with the edges between nodes and, if applicable, labels for the edges.

Both DOT files begin with the "type" of graph they will draw. Figure 5 is a "digraph", or directed graph, and Figure 7 is a "graph", an undirected graph. This specification is followed by the name of the graph, or in this case, the name of the dataset used to infer the graph. Edge properties of the graph come next, denoted by "edge", which is followed by a set of properties enclosed in a bracket. Both Figure 5 and 7 have edges with a solid style, a pen width of 5, and a label distance of 10. All of these properties are constant throughout the graph, so these are only specified once at the start of the DOT files. Nodes are defined next, which are signified by their cluster's name, followed by a bracket

containing the node's properties. These properties include the node's label, which includes the cluster's name, cardinality, radius, and LFD. Color is defined next, and the properties are concluded with the coloring style "filled".

Finally, the edges are written to the DOT file, and are defined in two ways depending on the type of graph. If we are writing a Cluster Tree graph, which is a digraph with directed edges like in Figure 5, then we write "->" on the line and between the names of the two clusters meant to be connected, signifying a directed edge going from the node on the left and to the node on the right. But if we are writing a Manifold Layer graph, which is a graph of undirected edges like in Figure 7, then we write "--" on the line and between the names of the two connected clusters, signifying an undirected edge between the two nodes. This undirected edge is followed by the properties of that particular edge in brackets, which in this case is a label of the edge's distance.

All the properties defined in the DOT files are visualized in their corresponding graphs produced by Graphviz. Figure 4 shows the visualized graph of the DOT file in Figure 5, and Figure 6 shows the visualized graph of the DOT file in Figure 7.

3.5 Results

The wide set of capabilities allowed by DOT format ensured that we could add all the features we wanted to the visualizations of our inferred graphs. These features included node labels, node colors, edge labels, and directed/undirected edges. The end result for Cluster Tree graphs were DOT files that gave us binary trees representing the manifold, starting from the node representing the root cluster and ending in leaf clusters at a specified depth. Directed edges go from parent nodes to child nodes, representing the binary division

of clusters as hierarchical clustering is applied. The end result for the Manifold Layer graphs were DOT files that gave us a visualization of a specified layer of the Cluster Tree, with nodes representing clusters at the layer of interest. Undirected edges between nodes signify the two clusters, represented by the connected nodes, having overlapping volumes. In both types of graphs, all nodes are labeled with their names, cardinality, radius, and LFD. All nodes are also given colors calculated from their LFD, all within a gradient between red (a high LFD) and blue (a low LFD).

Because of the capabilities of DOT format, the DOT files produced by the implemented algorithms worked exactly as intended. DOT files were read by Graphviz and visualized without encountering any error. The colors of the nodes in most of the Cluster Tree graphs showed an expected trend of increasing LFD the further the clusters were from the root. This is an example of graphs of summarized data having more information and value than data in its raw form. Several examples of manifold data we visualized using Graphviz can be found in Figures 8 and 9 of the Appendix. Figure 8 shows Cluster Tree graphs given by DOT files which our implemented algorithms produced, based on several different datasets. Figure 9 shows Manifold Layer graphs given by DOT files which our implemented algorithms produced, based on different datasets.

Figure 9 also has several examples of graphs with multiple disjoint connected components, or “islands”. These islands can be used to find clusters containing anomalous points, as well as determining if the manifold is made up of multiple distributions, or multiple disjoint components [7]. Each of the smaller islands on the Manifold Layer graphs contains one or more cluster with an LFD of exactly 1. As shown in Figure 3, a cluster’s LFD is a ratio of the set of points within two radii on the cluster’s center. An LFD of 1

would mean that both sets of points have the same amount of points, which signifies a low amount of points in that cluster (these clusters do not have only one point, otherwise they would have a radius of 0). These clusters also consistently have a low cardinality of 2, which means a higher likelihood of containing anomalous points. This solidifies the notion that these islands contain anomalous points, and these visualizations make it easier to find the clusters containing such points.

3.6 Visualizing DOT Files with Rust

After being able to create usable DOT files from our implemented algorithms, we then started working towards making our own DOT file visualization tool with many of the same capabilities of Graphviz. This tool would access the DOT file and read each line as a string, parsing it to extract information for the cluster or edge that line represents. Our tool would save the data and use existing programming tools and libraries to create a force-directed visualization of the graph.

For this part of the project, we implemented in Rust. Rust is a multi-paradigm system programming language that runs similar to C++. We decided that, with its fast performance, memory efficiency, no runtime, and no garbage collector [13], it was a feasible option for graph visualization. After our Rust tool read the DOT file and created a graph, we would use an existing Rust library, force_graph, to apply the Rust implementation of Graphoon, a force-directed graph algorithm [19]. After these two steps, our tool would then visualize the graph using Nannou, an open sourced coding graphics framework for Rust [14].

3.6.1 Reading the Dot File

In our Rust tool, we began by reading each line of the DOT file iteratively to gather graph data. As we were reading the DOT file, we looked for indicators of whether the current line was a cluster, an edge, or neither. If the current line was either a cluster or an edge, we would parse the line as a string to extract that line's data, and save it as either a cluster or an edge. Newly extracted clusters or edges are then put into one of two lists, one full of clusters and the other full of edges [11].

Algorithm 5: reading a DOT file to extract graph data

```
get_graph_from_dot(dot_file):
        graph.initialize()
        for line in file:
        |       if is_cluster(line):
        |       |       new_node = get_node_from_line(line)
        |       |       graph.add_node(new_node )
        |       end
        |       if is_edge(line):
        |       |       new_edge = get_edge_from_line(line)
        |       |       graph.add_node(new_edge)
        |       end
        end
        graph.finish()
        return graph
```

3.6.2 Applying Force Direct

To visualize a force-directed graph with our Rust tool, we used an existing Rust library, force_graph. This library uses the Rust implementation of the Graphoon algorithm [19]. Graphoon creates a force-directed graph layout by simulating physical forces, pulling and pushing each node in the graph until a visually appealing layout is found. Graphoon

emerged from the graph calculation code used that is present in both LoGiVi and LoFiVi, both of which are data visualization tools that display force-directed graphs [20]. In our project, this Rust implementation of Graphoon gives us the ability to create graphs and calculate their force-directed layout based on physical attraction and repulsion forces.

Graphoon works by first creating a graph, to which we can add nodes and edges. In force_graph, nodes have several properties which must be defined upon initialization. The first are the node's original coordinates, or the node's starting position before force-direct is applied. The second is the node's data, which is a user defined object. The third property is the node's mass, which is a major factor in calculating one node's repulsion towards other nodes. The last property is Boolean, and it decides whether or not the given node is an anchor node or not. Anchor nodes, simply put, have their initial coordinates be constant through every update of the force-directed graph. Edges, once initialized, only have two properties: the indices of the two nodes the edge connects. After adding the nodes and edges to the graph, we can apply the force calculations by calling a single function, update(), from force_graph in our Rust tool a set number of times. The update() function takes one parameter, which is the amount of force, as a floating point, applied to the movement of the force-direct algorithm. The more update() is called, the closer the user gets to a visually appealed force-directed graph [19].

We were able to make use of force_graph in two major areas of our Rust tool. Firstly, as we were reading the DOT file, we created a new graph using force_graph. After this, we iteratively went over each line of the DOT file, and added nodes to the graph when we read lines representing nodes, and added edges to the graph when we read lines representing edges. As each node was created and added to the graph, the nodes' initial

coordinates were set to random values on given ranges. All the nodes were given the same value for their mass, and their data was an object possessing cluster properties which were read from the DOT file. For every graph produced by our Rust tool, there was only one anchor node: the first node of the DOT file, which would be the root node in the case of Cluster Trees.

After the creation of the graph is complete, we apply the update() function from force_graph for a set number of iterations in order to apply the force-direct algorithm to the existing graph. We have observed that even a small amount of force, below 0.5, can cause significant movement in even a single call from update(). Also, after much trial and error, we have found that it is necessary to apply several thousand iterations of updates to the graph in order to achieve the best results for the force-directed layout.

3.6.3 Visualizing the Graph

After applying Force Direct to the graph, drawing it using Nannou is a simple matter of drawing all edges and all clusters iteratively. First, accessing the list of edges, we draw each edge as simple lines with the coordinates of the clusters they connect. Then the clusters are drawn in their appropriate coordinates as ellipses, then giving them the appropriate color. One issue we faced was that Nannou is unable to color shapes based on the Hexadecimal colors in our DOT files. Because of this, we had to implement a function that converted the Hexadecimal color strings to RGB tuples. Labels of the cluster are drawn inside its ellipse. Examples of the visualized graphs our tool produced can be found in Figure 10 and 11 in the Appendix.

3.6.4 Analyzing Datasets at Different Depths

One measure of success we defined in our project was how our Rust tool visualized graphs made from significantly high dimensional data. Using the added modules to Pyclam, we made DOT files of six datasets, each of them Manifold Layer graphs taken at different depths, which we visualized using our Rust tool.

In Figure 12 of the appendix, we included a chart of DOT files produced by the modules added to Pyclam, including the dataset they were taken from, their depth, the total number of nodes in the graph, and the number of disconnected nodes. The two largest datasets we used are apogee-train, with 254,160 data points at a dimension of 8,575, and silva-SSU-Ref-train, with 2,214,740 data points and a dimension of 50,000 [26, 27]. Figure 13 shows Manifold Layer graphs for both APOGEE and SILVA at different depths, highlighting our project's ability to derive graphs from significantly large datasets in terms of dimension and number of data points.

In each Manifold Layer graph, we found that more the depth increased, the more nodes there were in the graph, in terms of both the total number of nodes and, in some cases, the number of disconnected nodes. This is consistent with the Manifold Layer graph representing different layers of the Cluster Tree graph, since the further we go down the Cluster Tree, the more nodes there are. A larger depth also means more of a chance there is that we find disconnected nodes, which would be leaf nodes on the Cluster Tree graph.

3.7 Results

Evaluating the success of our Rust tool for visualizing graphs was mainly based on comparing its performance with Graphviz. However, the reading of the DOT file, as well as the implementation of the force-directed graph layout were also procedures whose success needed to be evaluated. We carefully analyzed how effectively the DOT file was read, assessing any possible weaknesses.

As we were implementing our Rust tool, we used the visualizations produced by Graphviz as an outline for what features to add to our own visualizations. The intent was to have the graphs our tool gave us display all the node labels and colors, as well as directed/undirected edges that are labeled with their distance when appropriate, all of which were visible on Graphviz. The only aspect of Graphviz we didn't intend to add to our Rust tool was the layout it presented graphs in. Instead we wanted our Rust tool to apply a force-directed layout to the graphs it would visualize. Examples of the visualized graphs our tool produced can be found in Figure 10 and 11 in the Appendix. These graphs are similar to the output provided by Graphviz, such that they display similar labels, their name, cardinality, radius, and LFD.

Reading the DOT file was a simple matter for Rust. We were able to take advantage of common patterns and trends in DOT files to identify which lines represented clusters and which lines represented edges. Parsing each line of the DOT file as a string is how our Rust tool gathers data for clusters and nodes. However, the functions that parse the line all depend on each line following a consistent order and format, that they are in a similar order as they are in the DOT files produced by our algorithms. This creates the possibility that

our Rust tool could fail to successfully extract the DOT file information if the DOT file was not written in an identical format as those produced by our algorithms implemented in Pyclam. For example, in the DOT files produced by our algorithms, the cluster's labels are given in a precise order, first its name, followed by its cardinality, radius, and LFD. The cluster's color and color style is then included respectively. Our Rust tool relies on all those features being in that precise order. If it read a DOT file with nodes whose labels and other properties were in a different order, our Rust tool would very likely return an error.

Concerning graphics, one shortcoming of our Rust tool was its inability to color the nodes in a gradient ranging between red and blue the way Graphviz did. Instead of coloring each node with its exact color, the colors were approximated to the closest color that was coded into Nannou. For example, if a cluster's color, read from the DOT file and converted to an RGB tuple, gives us (250, 0, 5), this color would be approximated to (255, 0, 0) when coloring a node on our graph.

The implementation of the Graphoon algorithm gave us a force-directed layout for both the Cluster Tree graphs and the Manifold Layer graphs. Visualizations of Cluster Tree graphs from our Rust tool gave clearly successful implementations of force-directed layouts. Figure 10 of the Appendix shows force-directed Cluster Tree graphs our Rust tool visualized from given DOT files, all of which were made from our implemented algorithms. Visualizations of Manifold Layer graphs from our Rust tool also gave force-directed layouts of these graphs, which can be found in Figure 11 of the Appendix.

There were a few shortcomings of the force-direct visualization in our Rust tool. Firstly, there lies the possibility of part of the visualized graph extending beyond the

boundaries of the computer display. This can be solved, however, by implementing a "scrolling" or "zoom in/out" feature to the Rust tool. Second, the layouts of the Manifold Layer graphs were noticeably more complicated than that of Cluster Tree graphs, due to a major difference in the number of edges between the two graph types. The Manifold Layer graph has nodes with higher degrees (the number of edges connecting the node), which is a significant factor in the force-direct algorithm, both in the attraction and repulsion of other nodes. There was also some trouble faced from finding the value for the node mass. After much trial and error, we found that a higher node mass is required for a greater number of high degree nodes to create a reasonable distance between these nodes. Using the same node mass on a graph with low degree nodes (below 5) can push the nodes very far apart from each other.

The end result for our graph visualization was a force-directed version of the Graphviz visualizations. Nodes from the graphs produced by our Rust tool were still labeled in a similar way as their Graphviz counterparts, displaying the name of the cluster they represent, as well as cardinality, radius, and LFD. The node color, although approximated to stricter Nannou color values, still was able to show a clear shift of LFD throughout different layers of Cluster Tree graphs. Edges are still drawn between clusters as defined in DOT files, whether they are directed edges between parent and child clusters, or undirected edges between nodes whose clusters overlap.

CHAPTER 4

CONCLUSION

4.1 Analysis of Results

The purpose of this study was to develop a tool to visualize high dimensional data using summaries of the given data, which were gathered by using hierarchical clustering and manifold learning together. CLAM (Clustered Learning of Approximate Manifolds), which implements both these techniques [7], was used to extract manifolds from various datasets. Afterwards, we would infer different graphs from these manifolds, based on the clusters formed at different stages of hierarchical clustering, and write these graphs to a DOT file. The DOT file would then be read and visualized using Graphviz, and later with a tool of our own design that was implemented in Rust.

The algorithms meant to infer graphs from the manifolds produced by CLAM were largely successful. They were evaluated by the DOT files they produced after being implemented in Python and added to Pyclam, and then read and visualized through Graphviz. The result was the clear visualization of graphs extracted from manifolds, showing intended manifold features in a way that's easy to follow. The DOT file visualization tool we implemented in Rust, despite having its shortcomings, provided the framework for a Rust-based DOT file visualizer. It was able to read DOT files and retrieve graph data by reading and parsing each line of the file. It also provided a force-directed layout of the graphs it visualizes.

4.2 Contributions

One of the contributions of our work in this project are providing algorithmic framework for inferring graphs from high dimensional data. Code added to Pyclam allows the user to get a graph from dimensionally reduced data in DOT format, a method which can be adapted to existing dimensionality reduction tools. The additions to CLAM also include a tool for inferring graphs of cluster trees, which show the dataset at each stage of hierarchical clustering and aids in finding the optimal layer of the cluster tree to derive an additional graph from. Another contribution of our work is the creation of a Rust DOT file visualizer, which can be incorporated into existing data dimensionality reduction tools.

The impact of these contributions has to do with the importance of graph visualizations in data analysis, exploration of high dimensional datasets, and anomaly detection. Graphs themselves represent complex data in a way that visually illustrates relationships in the data, and graph-structured data is used as feature information in predictions made by machine learning [23, 24]. Our visualizations allow for easy anomaly detection by providing clusters labeled with cardinality and LFD, both of which can be used to determine the presence of outliers in a cluster.

In the Manifold Layer graphs, clusters containing anomalous points are found in small disjoint “islands” in the graph, which represent isolated regions of the manifold. These disconnected components are more likely to contain anomalies, indicated by their low cardinality, due to their isolation from a majority of the other clusters. The LFD of a cluster, calculated by a ratio of points within the whole cluster and half the cluster, can be

used to indicate the presence of outlier data points the closer the cluster's LFD is to 1. It is through displaying the cluster's cardinality, LFD, and nature as an "island", that Manifold Layer graphs can not only just determine the presence of anomalous data points, but also detect whether the entire manifold is made up of a series of disjoint components.

4.3 Future Work

The work we have done in this project creates one possible model for two tools working together: data visualization and summarization of high dimensional data. Along with being the start of deriving optimal graphs from CLAM, our work can be a base for future methods of visualizing data in DOT file form to be added in existing dimensionality reduction tools. Possible future work for this topic include creating 3D visualizations of the graphs, and showing them in a graphical user interface or even virtual reality environments. Another is to unify both the DOT file producing functions and the visualization tool and have them both work in the same program. The current Rust tool can also be built on even more, paving the way for a Rust-implemented DOT file visualizer comparable with Graphviz.

APPENDIX

Dataset	# Data Points	# Dimensions	Dataset	# Data Points	# Dimensions
lympho	148	18	**vertebral**	240	6
wbc	278	30	**fashion-mnist-test**	10000	784
glass	214	9	**fashion-mnist-train**	60000	784
vowels	1456	12	**gist-test**	1000	960
cardio	1831	21	**gist-train**	1000000	960
thyroid	3772	6	**glove-100-test**	10000	100
musk	3062	166	**glove-100-train**	1183514	100
satimage-2	5803	36	**glove-200-test**	10000	200
pima	768	8	**glove-200-train**	1183514	200
satellite	6435	36	**kosarak-test**	500	27983
shuttle	49097	9	**kosarak-train**	74962	27983
breastw	683	9	**lastfm-test**	50000	65
arrhythmia	452	274	**lastfm-train**	292385	65
ionosphere	351	33	**mnist-test**	10000	784
mnist	7063	100	**mnist-train**	60000	784
optdigits	5216	64	**nytimes-test**	10000	256
cover	286048	10	**nytimes-train**	290000	256
mammography	11183	6	**sift-test**	10000	128
annthyroid	7200	6	**sift-train**	1000000	128
pendigits	6870	16	**apogee-train**	254160	8575
wine	129	13	**silva-SSU-Ref-train**	2214740	50000

Figure 1: datasets used in this project, taken from the UCI Machine Learning Repository [25], APOGEE [26], and SILVA [27]

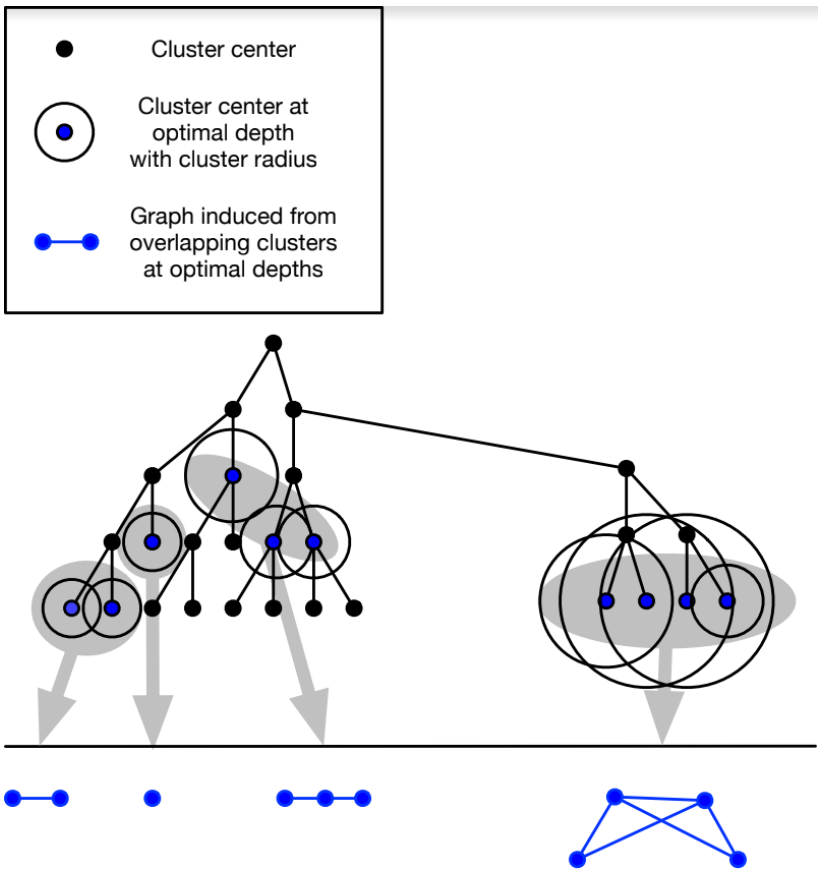

Figure 2: a "cartoon" explanation of how Cluster Tree graphs and Manifold Layer graphs are inferred from manifolds (purely a representation and not based on any existing data). The cluster center at the top of the tree is the "root" cluster, or the cluster containing all points of the dataset at the first stage of hierarchical clustering. All following "child" clusters are derived from splitting the previous "parent" cluster into two clusters. At each level of the tree, we can infer graphs of all the clusters at that level, or "layer", and form edges between clusters whose radii overlap with each other [7].

$$log_2\left(\frac{|B_D(q,r_1)|}{|B_D(q,r_2)|}\right)$$

Figure 3: Formula for a cluster's LFD (Local Fractal Dimension), where $B_D(q, r)$ is the set of points contained in a sphere on the dataset D of radius r centered on point q; fractal dimension is computed for radius r_2 and a smaller radius $r_1 = r_2/2$ [8]

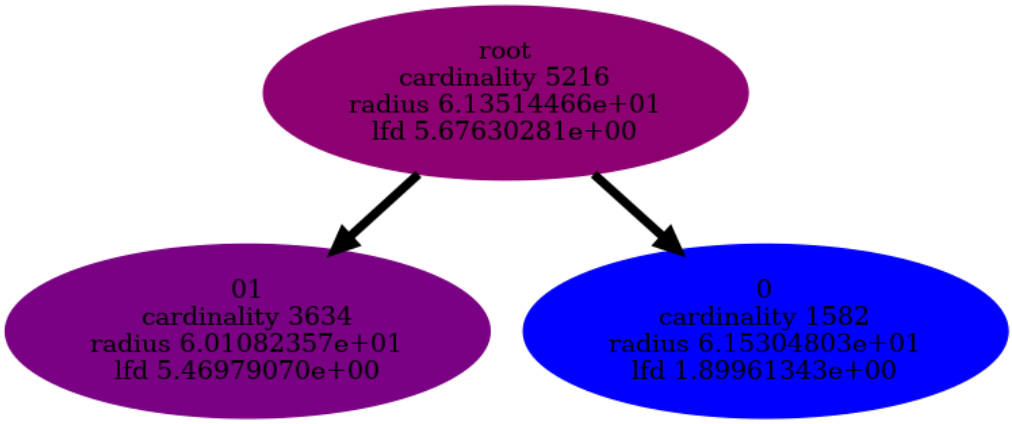

Figure 4: a small, close up view of a Cluster Tree graph visualized by Graphviz using a DOT file our algorithm produced. The dataset used was optdigits, and was extracted at the tree depth of 2. The node colors are indicative of their LFD (local fractal dimension), which gets closer to blue as we move down the tree, indicating a decrease.

```
digraph optdigits {
    edge[style=solid, penwidth="5", labeldistance="10"]
    root [label="root\ncardinality 5216\nradius 6.13514466e+01\nlfd 5.67630281e+00", color="#8D0071", style="filled"]
    01 [label="01\ncardinality 3634\nradius 6.01082357e+01\nlfd 5.46979070e+00", color="#7A0084", style="filled"]
    0 [label="0\ncardinality 1582\nradius 6.15304803e+01\nlfd 1.89961343e+00", color="#0000FF", style="filled"]
    root -> 01
    root -> 0
}
```

Figure 5: the DOT file our algorithm produced to make the Cluster Tree graph in Figure 4

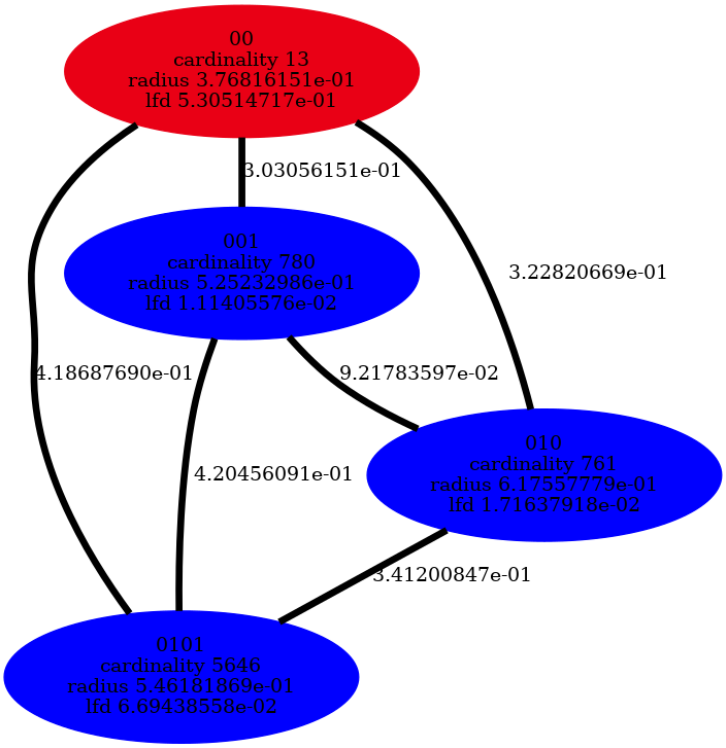

Figure 6: a small, close up view of a Manifold Layer graph visualized by Graphviz using a dot file our algorithm produced. The dataset used was annthyroid, and was extracted at the tree depth of 2. The node colors are indicative of their LFD (local fractal dimension), showing one red cluster (a high LFD compared to the other three nodes).

```
graph annthyroid {
    edge[style=solid, penwidth="5", labeldistance="10"]
    010 [label="010\ncardinality 761\nradius 6.15772504e-01\nlfd 1.90835460e-02", color="#0000FF", style="filled"]
    00 [label="00\ncardinality 13\nradius 3.76816151e-01\nlfd 5.30514717e-01", color="#E90015", style="filled"]
    001 [label="001\ncardinality 780\nradius 5.12896431e-01\nlfd 1.30057098e-02", color="#0000FF", style="filled"]
    0101 [label="0101\ncardinality 5646\nradius 5.52637313e-01\nlfd 2.21442458e-02", color="#0000FF", style="filled"]
    00 -- 010 [label="3.17249497e-01"]
    001 -- 010 [label="1.11464763e-01"]
    00 -- 0101 [label="4.11171498e-01"]
    00 -- 001 [label="3.29279538e-01"]
    001 -- 0101 [label="4.20028748e-01"]
    010 -- 0101 [label="3.10379327e-01"]
}
```

Figure 7: the DOT file our algorithm produced to make the Manifold Layer graph in Figure 6.

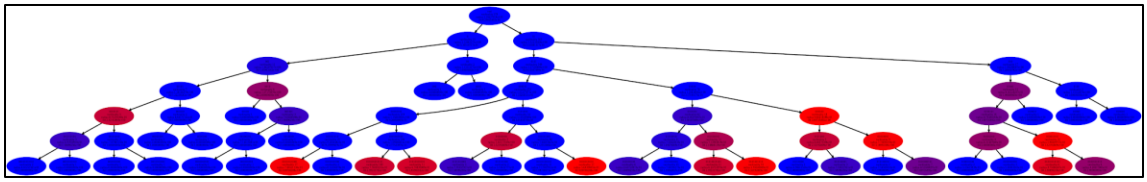

Figure 8.a

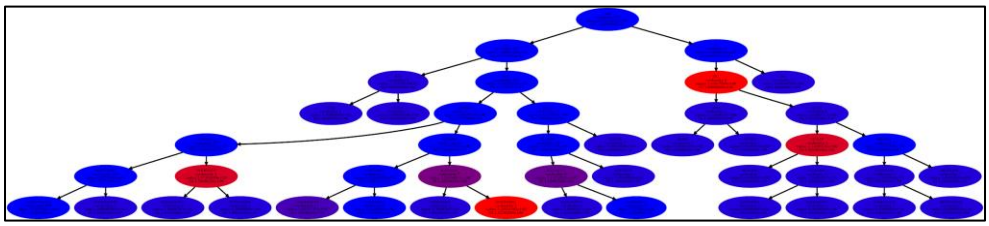

Figure 8.b

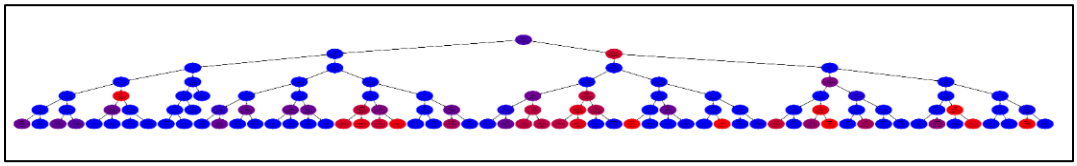

Figure 8.c

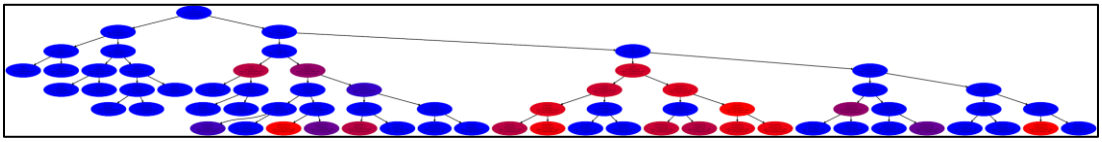

Figure 8.d

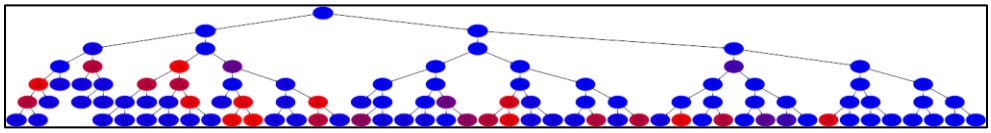

Figure 8.e

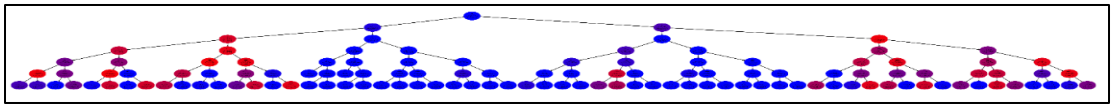

Figure 8.f

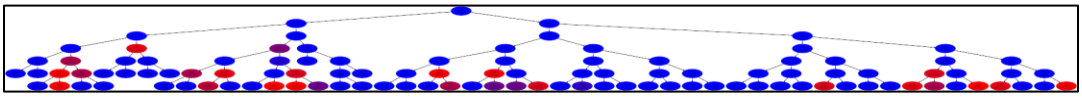

Figure 8.g

Figure 8: Cluster Tree graphs visualized by Graphviz using DOT files our algorithms produced. The datasets used are lympho (8.a), wbc (8.b), glass (8.c), vowels (8.d), cardio (8.e), thyroid (8.f), and musk (8.g), and all have a depth of 6. The node colors of each graph are indicative of their LFD (local fractal dimension), and give us an idea of the change in LFD without having to read the labels. These graphs show us what clusters have an increase in LFD (closer to red), a decrease in LFD (closer to blue), and no change.

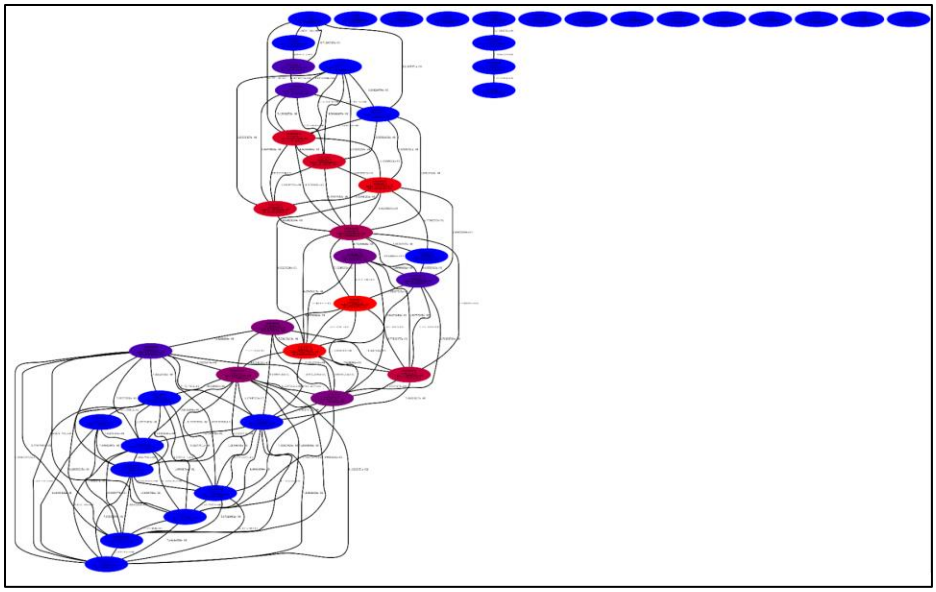

Figure 9.a

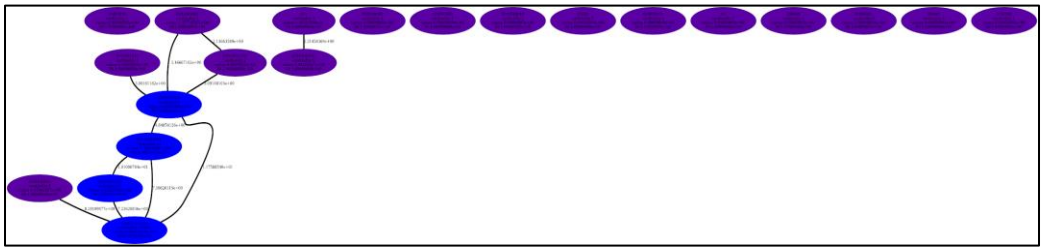

Figure 9.b

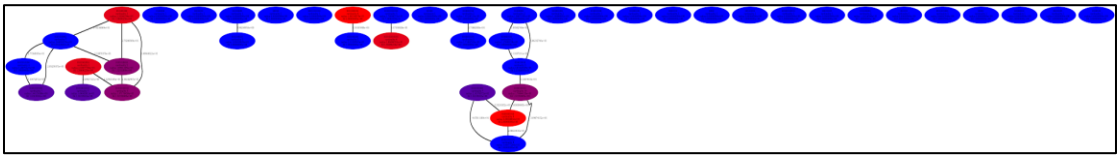

Figure 9.c

Figure 9: Manifold Layer graphs visualized by Graphviz using DOT files our algorithms produced. The datasets used are pima (9.a), mammography (9.b), and wine (9.c), all at a depth of 6. These graphs all share a feature of having multiple disjoint connected components, of which possess clusters that fit the criteria for outliers and anomalies (LFD of 1 and cardinality of 2).

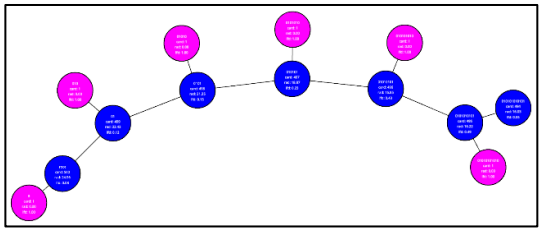

Figure 10.a

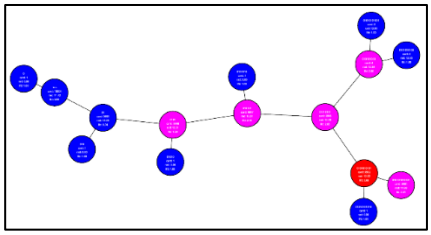

Figure 10.b

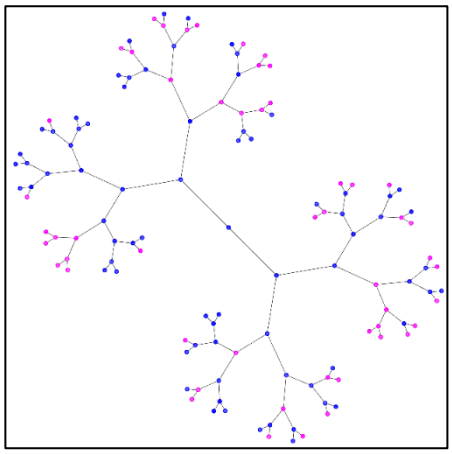

Figure 10.c

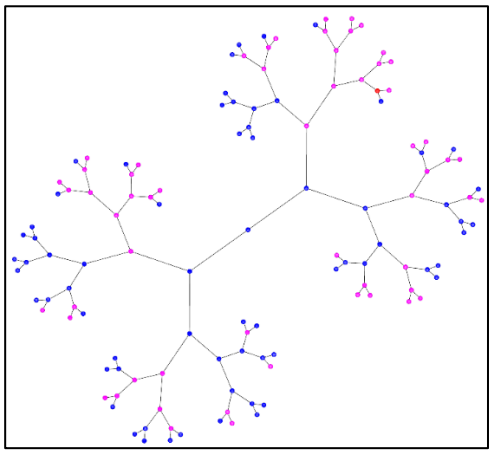

Figure 10.d

Figure 10: Cluster Tree graphs visualized by our Rust tool using DOT files our algorithms produced. The datasets used are kosarak-test (10.a), glove-200-test (10.b), sift-train (10.c), and mnist-train (10.d), and all have a depth of 6.

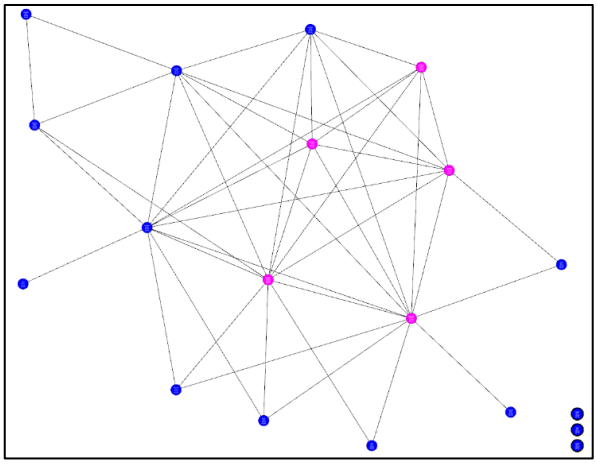

Figure 11.a

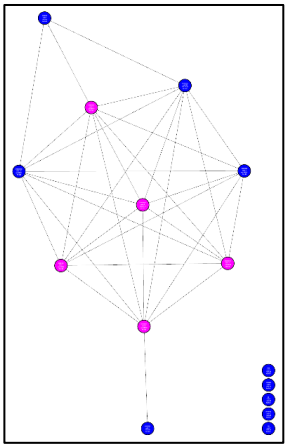

Figure 11.b

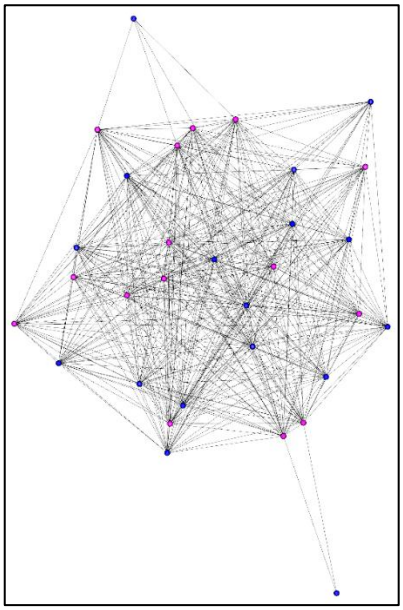

Figure 11.c

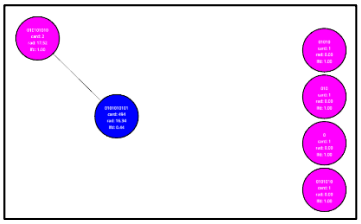

Figure 11.d

Figure 11: Manifold Layer graphs visualized by our Rust tool using DOT files our algorithms produced. The datasets used are gist-test (11.a), arrhythmia (11.b), fashion-mnist-test (11.c), and kosarak-test (11.d). All these graphs were extracted at the tree depth of 5. The top two and the bottom right graphs all have a set of disconnected nodes, which are pushed to the lower right side of the graph.

Dataset	# Data Points	# Dimensions	Depth	Total Nodes	Disconnected Nodes
arrhythmia	452	274	5	15	5
arrhythmia	452	274	6	23	8
arrhythmia	452	274	7	35	10
arrhythmia	452	274	11	95	31
arrhythmia	452	274	20	437	415
mnist	7063	100	5	32	0
mnist	7063	100	6	64	0
mnist	7063	100	7	126	0
mnist	7063	100	11	1094	32
mnist	7063	100	20	7045	5040
musk	3062	166	5	32	0
musk	3062	166	6	62	3
musk	3062	166	7	117	44
musk	3062	166	11	939	207
musk	3062	166	20	3062	3062
fashion-mnist-test	10000	784	5	32	0
fashion-mnist-test	10000	784	6	62	0
fashion-mnist-test	10000	784	7	117	2
fashion-mnist-test	10000	784	11	941	87
fashion-mnist-test	10000	784	20	6818	3255
gist-test	1000	960	5	19	3
gist-test	1000	960	6	28	13
gist-test	1000	960	7	37	17
gist-test	1000	960	11	96	39
gist-test	1000	960	20	627	152
kosarak-test	500	27983	5	6	4
kosarak-test	500	27983	6	8	5
kosarak-test	500	27983	7	10	7
kosarak-test	500	27983	11	16	10
kosarak-test	500	27983	20	61	30
apogee-train	254160	8575	5	15	5
apogee- train	254160	8575	6	25	16
apogee- train	254160	8575	7	34	23
apogee- train	254160	8575	11	123	76
apogee- train	254160	8575	20	1129	640
silva-SSU-Ref- train	2214740	50000	5	19	7
silva-SSU-Ref- train	2214740	50000	6	29	8
silva-SSU-Ref- train	2214740	50000	7	46	12
silva-SSU-Ref- train	2214740	50000	11	243	68
silva-SSU-Ref- train	2214740	50000	20	3436	1004

Figure 12: A chart of Manifold Layer graphs taken from different datasets at different depths (5, 6, 7, 11, 20), and the graphs' total number of nodes and disconnected nodes.

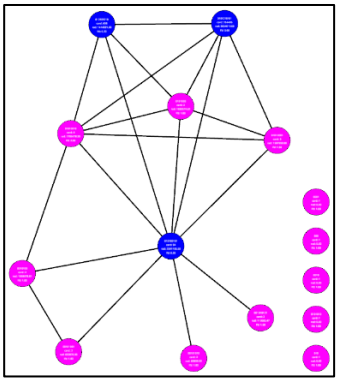

Figure 13.a

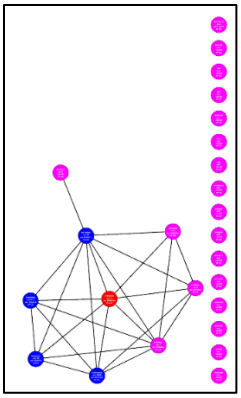

Figure 13.b

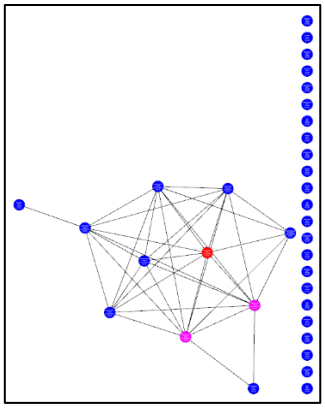

Figure 13.c

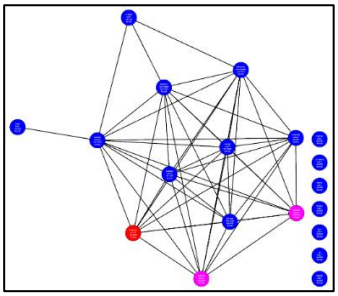

Figure 13.d

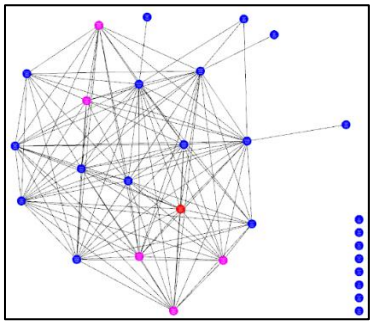

Figure 13.e

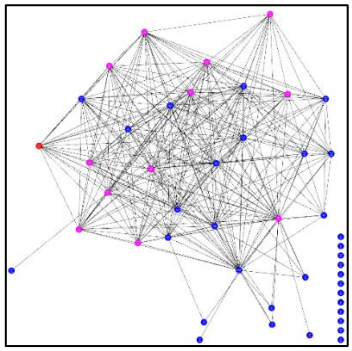

Figure 13.f

Figure 13: Manifold Layer graphs visualized by our Rust tool using DOT files our algorithms produced. The graphs are derived from apogee-train at depth 5 (13.a), apogee-train at depth 6 (13.b), apogee-train at depth 7 (13.c), silva-SSU-Ref-train at a depth of 5 (13.d), silva-SSU-Ref-train at a depth of 6 (13.e), and silva-SSU-Ref-train at a depth of 7 (13.f).

BIBLIOGRAPHY

[1] DeMers, D., & Cottrell, G. (1992). Non-linear dimensionality reduction. *Advances in neural information processing systems*, *5*, 580-587.

[2] Van der Maaten, L. J. P. (2007). An introduction to dimensionality reduction using matlab. Report, 1201(07-07), 62.

[3] Van Der Maaten, L., Postma, E., & Van den Herik, J. (2009). Dimensionality reduction: a comparative. *J Mach Learn Res*, *10*(66-71), 13.

[4] VanderPlas, J. (n.d.). Python Data Science Handbook. Retrieved November 16, 2020, from https://jakevdp.github.io/PythonDataScienceHandbook/

[5] Tan, P., Steinbach, M., Karpatne, A., & Kumar, V. (2005). *Introduction to Data Mining* (2nd ed.). Pearson.

[6] Cayton, L. (2005). Algorithms for manifold learning. Univ. of California at San Diego Tech. Rep, 12(1-17), 1.

[7] Howard, T. J., Ishaq, N., & Daniels, N. M. (n.d.). Clustered Hierarchal Anomaly and Outlier Detection Algorithms. Unpublished.

[8] Ishaq, N., Student, G., & Daniels, N. M. (2019, December). Clustered hierarchical entropy-scaling search of astronomical and biological data. In *2019 IEEE International Conference on Big Data (Big Data)* (pp. 780-789). IEEE.

[9] (n.d.). Retrieved November 16, 2020, from https://www.saedsayad.com/clustering_hierarchical.htm

[10] Uri-Abd. (n.d.). URI-ABD/clam. Retrieved November 20, 2020, from https://github.com/URI-ABD/clam/tree/master

[11] Amani, A. (n.d.). Ali-amani01/nannou_dot_file_visualization. Retrieved April 21, 2021, from https://github.com/ali-amani01/draw_force_graph

[12] Graph Visualization Software. (n.d.). Retrieved November 16, 2020, from https://www.graphviz.org/

[13] Rust. (n.d.). Retrieved November 16, 2020, from https://www.rust-lang.org/

[14] Nannou. (n.d.). Home. Retrieved November 16, 2020, from https://nannou.cc/

[15] Lee, J. A., & Verleysen, M. (2007). *Nonlinear dimensionality reduction*. Springer Science & Business Media.

[16] McInnes, L., Healy, J., & Melville, J. (2018). Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*.

[17] Maaten, L. V. D., & Hinton, G. (2008). Visualizing data using t-SNE. *Journal of machine learning research*, *9*(Nov), 2579-2605.

[18] Karamizadeh, S., Abdullah, S. M., Manaf, A. A., Zamani, M., & Hooman, A. (2013). An overview of principal component analysis. *Journal of Signal and Information Processing*, *4*(3B), 173.

[19] T-Mw. (n.d.). T-mw/force-graph-rs. Retrieved December 01, 2020, from https://github.com/t-mw/force-graph-rs

[20] Rm-Code. (n.d.). Rm-code/Graphoon. Retrieved December 01, 2020, from https://github.com/rm-code/Graphoon/

[21] Stolz, B. (2014). Computational topology in neuroscience. *Master's thesis (University of Oxford, 2014)*.

[22] Siva, S. (2020, November 12). Dimensionality Reduction for Data Visualization: PCA vs TSNE vs UMAP vs LDA. Retrieved November 30, 2020, from https://towardsdatascience.com/dimensionality-reduction-for-data-visualization-pca-vs-tsne-vs-umap-be4aa7b1cb29

[23] Slutsky D. J. (2014). The effective use of graphs. Journal of wrist surgery, 3(2), 67–68. https://doi.org/10.1055/s-0034-1375704

[24] Hamilton, W. L., Ying, R., & Leskovec, J. (2017). Representation learning on graphs: Methods and applications. arXiv preprint arXiv:1709.05584.

[25] Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[26] Prieto, C. Allende, et al. "APOGEE: the Apache point observatory galactic evolution experiment." *Astronomische Nachrichten: Astronomical Notes* 329.9-10 (2008): 1018-1021.

[27] Quast C, Pruesse E, Yilmaz P, Gerken J, Schweer T, Yarza P, Peplies J, Glöckner FO (2013) The SILVA ribosomal RNA gene database project: improved data processing and web-based tools.