

Biclustering and Visualization of High Dimensional Data using Visual Statistical Data Analyzer

Patrick M. Blake

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Yue Wang, Chair
Jianhua Xuan
Guoqiang Yu

December 7th, 2018
Arlington, Virginia

Keywords: high dimensional data, biclustering, VISDA, VISDApy

Copyright 2018, Patrick M. Blake

Biclustering and Visualization of High Dimensional Data using Visual Statistical Data Analyzer

Patrick M. Blake

ABSTRACT

The proliferation of high dimensional data (HDD), where the number of features is much greater than the number of samples, due to technological improvements in sensing and computation precipitates the need for tools to handle its unique challenges. In general, biclustering methods, which focus on clustering in subspaces within the data, can help alleviate the Curse of Dimensionality, but are subject to limitations in that the user must have thorough knowledge of the data. This includes knowing how many biclusters to expect and what structure they hold. This thesis set out to further develop the Visual Statistical Data Analyzer (VISDA) into VISDApy, and to propose a method for using it to explore HDD and inform the biclustering process. VISDApy was developed and tested, and a few excursions were run on a muscular dystrophy microassay dataset as well as two synthetic datasets. The next steps for this research would be to increase the speed and improve the user experience for VISDApy, and to look at an even wider range of datasets.

Biclustering and Visualization of High Dimensional Data using Visual Statistical Data Analyzer

Patrick M. Blake

GENERAL AUDIENCE ABSTRACT

Many data sets have too many features for conventional pattern recognition techniques to work properly. This thesis investigates techniques that alleviate these difficulties. One such technique, biclustering, clusters data in both dimensions and is inherently resistant to the challenges posed by having too many features. However, the algorithms that implement biclustering have limitations in that the user must know at least the structure of the data and how many biclusters to expect. This is where the Visual Statistical Data Analyzer, or VISDA, can help. It is a visualization tool that successively and progressively explores the structure of the data, identifying clusters along the way. This thesis proposes coupling VISDA with biclustering to overcome some of the challenges of data sets with too many features. Further, to increase the performance, usability, and maintainability as well as reduce costs, VISDA was translated from Matlab to a Python version called VISDApy. Both VISDApy and the overall process were demonstrated with real and synthetic data sets. The results of this work have the potential to improve analysts' understanding of the relationships within complex data sets and their ability to make informed decisions from such data.

Dedication

To: My parents, who raised me to value and pursue my education at all levels, and who supported me in many ways throughout the journey.

Acknowledgments

I would like to Acknowledge Dr. Joseph Wang for both his tutelage in the classroom and his assiduous guidance during the research for the thesis. His course on Advanced Machine Learning originally got me interested in this topic. Also, he was very responsive whenever I was at an impasse during the thesis. I would also like to Acknowledge the other Committee Members, Dr. Jason Xuan and Dr. Guoqiang Yu, for their valued insights and suggestions to help make the thesis stronger. Finally, I would like to Acknowledge Brian Koch and Julia Packer of the MITRE Corporation, who supported my pursuit of an MSEE while working full time.

Contents

1	Introduction	1
2	Development of VISDApy	13
2.1	Design Principles	13
2.1.1	Complementary Projections	13
2.1.2	Hierarchical Exploration	15
2.2	Design of VISDApy	17
2.3	Demonstration of VISDApy	32
2.4	Testing VISDApy	43
2.5	Future Improvements to VISDApy	46
3	VISDApy for Bicluster Algorithm Heuristic Selection and Structure Identification	49

List of Figures

1.1	Example Dataset with 1000 features and 100 Samples	2
1.2	Impact of the Curse of Dimensionality on Predictive Classification Error, Adapted from [3]	3
1.3	Feature Extraction	4
1.4	Clustering in Different Dimensions, Adapted from [1]	6
1.5	Clustering (Global) vs. Biclustering (Local)	7
1.6	Example of Exclusive Biclusters, Adapted from [1]	8
1.7	Example of Overlapping Biclusters, Adapted from [1]	9
1.8	Example of Hierarchical Bicluster Structure, Adapted from [1]	9
1.9	Example of Checkerboard Bicluster Structure, Adapted from [1]	10
2.1	VISDApy's Program Flow Diagram, Adapted from [9]	17
2.2	VISDApy's Functional Architecture	19

2.3	HC Dendrogram for Level 1	32
2.4	Cluster Projections for Level 1	33
2.5	Subcluster Model Selections for Level 1	33
2.6	Minimum Description Length of Each Subcluster Model of Level 1	34
2.7	Total Visualization for Level 1 of the Hierarchy	35
2.8	HC Dendrogram for Level 2 Cluster 1	35
2.9	Projections for Level 2 Cluster 1	36
2.10	Minimum Description Length of Each Subcluster Model of Level 2 Cluster 1	36
2.11	HC Dendrogram for Level 2 Cluster 2	37
2.12	Projections for Level 2 Cluster 2	37
2.13	Minimum Description Length of Each Subcluster Model of Level 2 Cluster 2	38
2.14	Level 2 Cluster 2 Subcluster Model Selections	38
2.15	HC Dendrogram for Level 2 Cluster 3	39
2.16	Projections for Level 2 Cluster 3	39
2.17	Level 2 Hierarchy Found by VISDApy	40
2.18	Level 2 Hierarchy with Cluster Images	40
2.19	Level 3 Hierarchy Found by VISDApy	41

2.20	Total Hierarchy with Cluster Images	41
2.21	Alternate Hierarchy Found by VISDApy	42
2.22	Alternate Total Hierarchy	42
2.23	Cluster Projections for Level 1	45
2.24	Alternate Hierarchy Found by VISDApy	46
3.1	Synthetic Data with 5 Exclusive Biclusters	50
3.2	Shuffled Synthetic Data	50
3.3	VISDApy Finds Flat Structure of 5 Biclusters	50
3.4	Biclustered Data from Spectral Coclustering Algorithm with VISDApy's Out-puts	50
3.5	Synthetic Hierarchical Biclusters Mixed with Single Exclusive Bicluster	51
3.6	Synthetic Hierarchical Biclusters Mixed with Single Exclusive Bicluster and AWGN	51
3.7	VISDApy Finds Hierarchical Structure	52

List of Tables

2.1	Dependency Table for Main and Coordinating Functions.	20
2.2	Dependency Table for Visualization Functions.	22
2.3	Dependency Table for Data Projection Functions.	25
2.4	Dependency Table for Mathematical Support Functions.	28

Chapter 1

Introduction

Researchers and analysts across fields need to understand the relationships within increasingly complex, high-dimensional datasets. Generally, a dataset contains features, or variables, and samples, or separate instances of a set of features. For example, a car can have many features like color, engine volume, horsepower, number of doors, fuel efficiency, or any other attribute. Samples could be individual models like Civic, Accord, Camry, Focus, Dart, or any other model of car. A dataset is considered high-dimensional when the number of features, p , is much greater than the number of samples, n (ie: $p \gg n$) [4]. Figure 1.1 shows an example of such a dataset. The thousands of features to only tens of samples classify this dataset as high-dimensional. Improvements in sensing and measurement techniques, understanding of physical phenomena, and capacity of data storage have facilitated an increase in collectible features and thus high dimensional data (HDD).

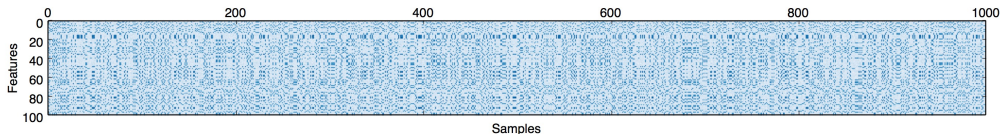


Figure 1.1: Example Dataset with 1000 features and 100 Samples

The analytics that support solving high-dimensional, Big Data problems face many challenges in making sense of the data and extracting meaningful relationships from it. HDD can have very complex structures [7]. Clarke, et. al. [3] detail how it may contain numerous potentially relevant subspaces and correlated data points, which can occur at both a local and global scope. Many of these subspaces and correlations can be spurious, depending on the data. Further, distance-based analytic methods tend to have difficulty with HDD, as nearest and farthest neighbor points tend to be equidistant. Neural network-based classification approaches have to contend with very large search radii when used on HDD. Many irrelevant subspaces in the data can seriously compound the matter. The network must allocate limited resources to all of them, as the algorithms are not able to distinguish between what is relevant and what is not [3]. Overfitting and overly-complex models are common problems with analyzing HDD [4]. In general, obtaining a correct, or even acceptably correct, solution is very computationally intensive [3]. Collectively, these challenges are known as the Curse of Dimensionality.

Figure 1.2 from [3] depicts the impact of the Curse of Dimensionality, specifically as it relates to the bias-variance dilemma. The x-axis is the dimension of the data, and the y-axis is the prediction error. The orange curve is the variance of the data, while the green curve is the

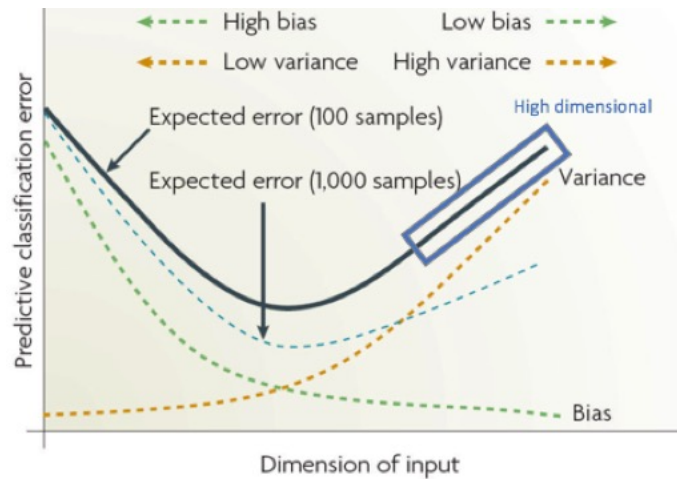


Figure 1.2: Impact of the Curse of Dimensionality on Predictive Classification Error, Adapted from [3]

bias of the data. With a low dimensionality, or number of features with respect to samples, the bias toward the few features included is the principal driver of the prediction error. With a high dimensionality, the variance of the many features with respect to the smaller number of samples leads to overfitting of predictive models to the data. High variance and overfitting drive increased prediction error with models of HDD [4, 3]. Methods for decreasing this prediction error must target the variance in some way. The conventional approach aims to do this by directly reducing the overall dimensionality of the data, pushing the prediction error from its right-hand slope back into the trough of the curve in Figure 1.2. The first solution in this family of methods would be to increase the number of samples in the dataset to reduce the dimensionality. However, this is rarely possible or practical, and may not relieve all of the complications of having so many features. More common approaches are feature extraction, feature selection, or some combination of the two.

Feature extraction attempts to form non-redundant features from the original or down-

selected feature set. This typically involves some combination of the features. For example, Principal Component Analysis (PCA) maps the features to a new space where principal components are some linear combination of the original feature set and represent the directions with greatest variance in the data. Feature extraction methods like PCA also induce information loss because the new features are combinations of the original feature set mapped to a new space as Figure 1.3 demonstrates. The features in the new set do not have any real-world meaning and cannot be mapped back to the original set. Feature extraction produces feature sets that may be difficult for analysts and researchers, who usually deal with real-world variables, to understand.

$$\begin{pmatrix} gene_1 \\ gene_2 \\ gene_3 \\ \dots \\ gene_n \end{pmatrix} \rightarrow \begin{pmatrix} pc_1 \\ pc_2 \\ \dots \\ pc_m \end{pmatrix}$$

Figure 1.3: Feature Extraction

Feature selection attempts to reduce the overall feature set to the subset that is most relevant globally. This induces information loss, as the non-selected features may have relevance at a local level. However, the selected features remain real-world variables. The approach for conventional methods might be to perform feature selection globally on the entire dataset, perform feature extraction globally on the dataset with down-selected features, and finally perform the chosen clustering method globally on the feature-extracted data. This conventional approach assumes the only relevant space is defined by the global feature set or some down-selection of it, discounting potentially relevant subspaces.

Another approach for handling HDD uses learning methods that resist the Curse of Dimensionality [3]. The investigation of these methods begins with clustering. Clustering is a very useful method for extracting information when dealing with limited knowledge of a dataset [9]. Clustering is generally used to find patterns based on similarities within data. Clustering techniques include Hierarchical Clustering, K-Means Clustering, Self-Organizing Maps, and Support Vector Machines [2]. Some Hierarchical Clustering approaches avoid the computational heft required by other methods to work with HDD, but are only useful for visualization [3]. Support Vector Machines handle high-dimensionality very well for binary classification, but are somewhat limited outside this scope. Zhu, et. al. [9] points out that while a good starting point, conventional clustering approaches have shortcomings that prevent them from adequately addressing the Curse of Dimensionality. Most, if not all, conventional clustering methods require heuristics for algorithm initialization, or just use a random initialization. As such, they are susceptible to the selection of poor model initialization parameters, causing them to converge on poor local optima and yield inaccurate results. HDD exacerbates this weakness with its many potentially relevant subspaces. Lack of cluster number detection is also a significant limitation of conventional approaches [9]. Conventional Clustering techniques by themselves cannot adapt to expert knowledge or results from other methods.

The concept of clustering across the whole dataset has some additional limitations related to those of the dimension reduction techniques. Conventional clustering techniques focus on finding global patterns within the data. That is, they aim to find patterns in the samples

across all features [7]. Alternatively, conventional clustering methods could characterize groups of features across all samples. They cannot do both simultaneously [6]. Figure 1.4 demonstrates that traditional clustering methods would detect two clusters in the X1 dimension, and three clusters in the X2 dimension. In many applications, like gene expression analysis, it is desirable to identify how subsets of samples behave according to subsets of features. This is not possible with a global approach [1]. Improvements must address both the limitations of the overall conventional clustering approach and those of the individual algorithms.

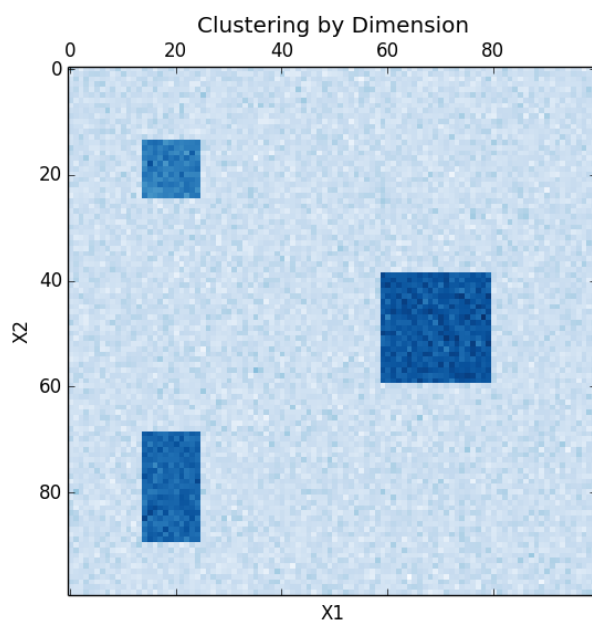


Figure 1.4: Clustering in Different Dimensions, Adapted from [1]

This investigation leads to a relatively new method of clustering called biclustering. Biclustering is a general method for detecting local patterns in data [7]. Biclustering is a two-dimensional technique, as it clusters by both rows and columns, while global clustering

does so by just rows or columns. Biclustering is considered a local, or subspace, technique because it seeks to cluster by both samples and features. Clustering by subspaces of the total dimension is inherently resistant to the Curse of Dimensionality. Instead of looking for similarities of samples across all features, biclustering looks for similarities across a subset of features [7]. Figure 1.5 shows the visual differences in what an analyst might see between global and local clusters. Biclusters visualize complex relationships very well, and if there are global patterns, can show the mappings from local to global clusters. Biclustering on the original feature set induces minimal information loss compared to that incurred with dimension reduction and global clustering. Avoidance of dimension reduction techniques allows the original feature set to be maintained for the analyst or researcher. Features that correspond to real-world variables are easier to understand than mathematical projections. This approach addresses the conceptual limitations of global dimension reduction and clustering.

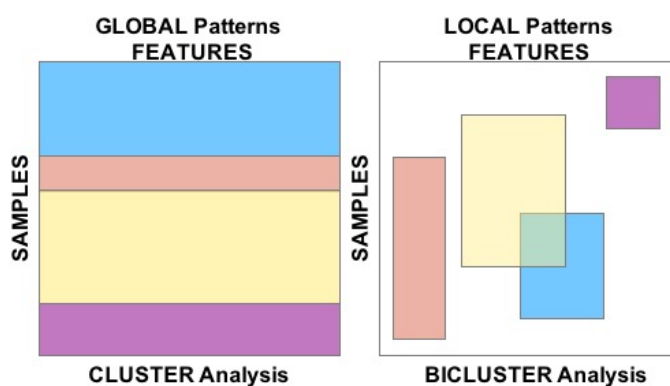


Figure 1.5: Clustering (Global) vs. Biclustering (Local)

Biclusters may occur in a few different configurations. Figures 1.6 through 1.9 depict four different structures of biclusters. Figure 1.6 shows a nonoverlapping, exclusive configuration, 1.7 shows an overlapping configuration, 1.8 shows a hierarchical configuration, and 1.9 shows

a checkerboard configuration [1]. Nonoverlapping exclusive structures tend to appear in clustered text documents, where documents are the samples and the features are the total vocabulary contained in them. This is useful for an analyst to understand the relationships between documents, their authors, and the language contained therein. Hierarchical and overlapping structures tend to appear in genetic expression data. The mapping of gene to condition is rarely one-to-one. More often there are combinations of genes that may be active for a given condition, while a subset of that combination is also active for another condition. The checkerboard pattern seems to be an artifact of the algorithm used. It may be used as a coarse approximation of the other structures. The structures present in the data influence the selection of the best biclustering algorithm to find them.

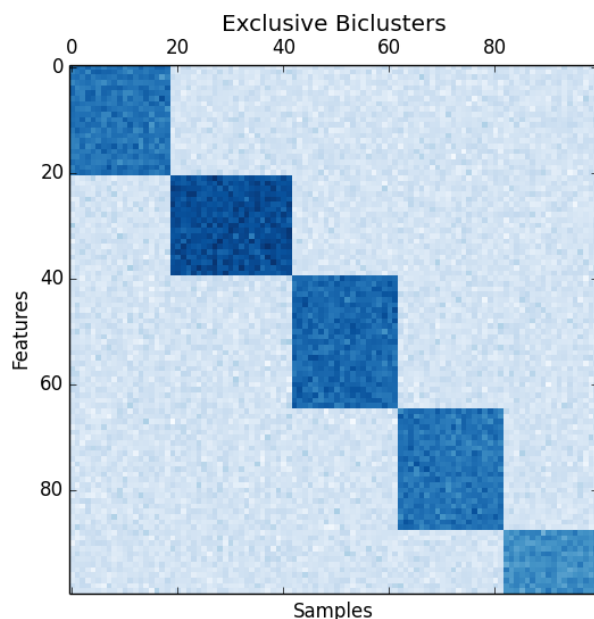


Figure 1.6: Example of Exclusive Biclusters, Adapted from [1]

Biclustering retains many of the limitations of conventional clustering approaches at the

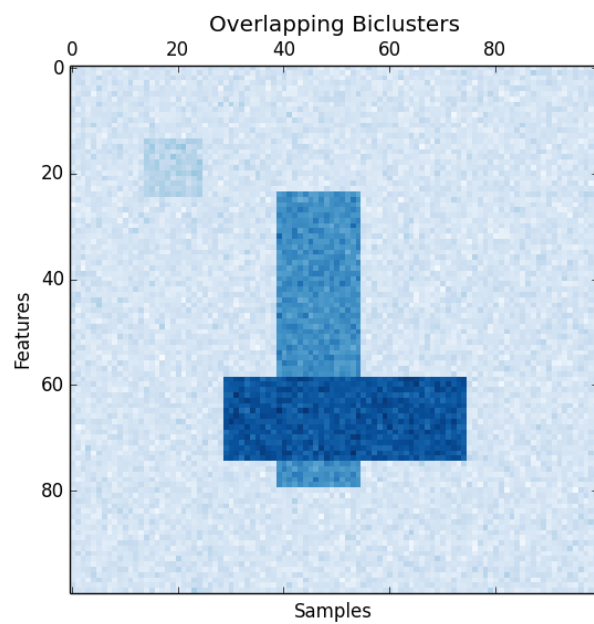


Figure 1.7: Example of Overlapping Biclusters, Adapted from [1]

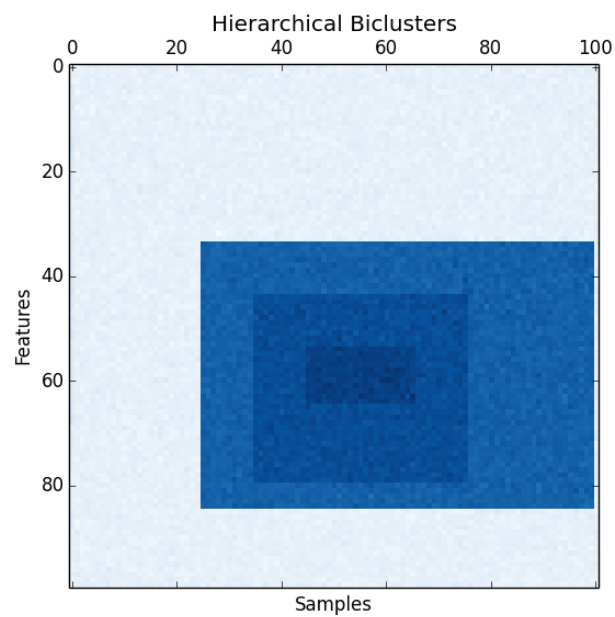


Figure 1.8: Example of Hierarchical Bicluster Structure, Adapted from [1]

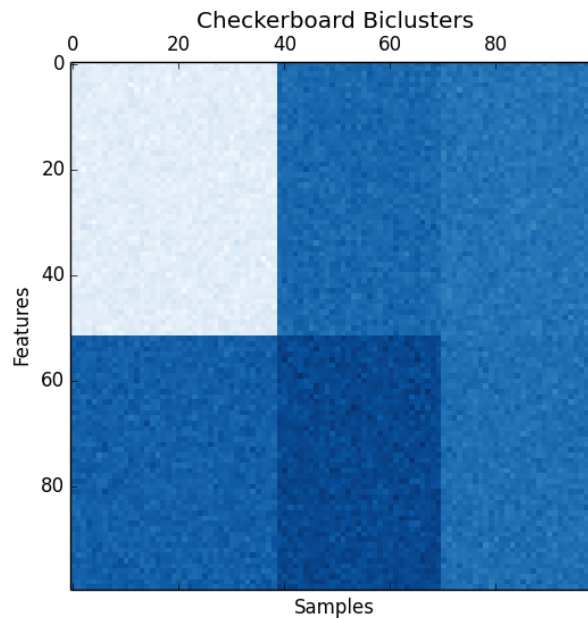


Figure 1.9: Example of Checkerboard Bicluster Structure, Adapted from [1]

algorithm level. The general biclustering algorithm is an NP-hard problem. Most implementations of the algorithm rely on heuristics to speed up the processing time. Knowledge of the expected number of biclusters remains one of the key heuristics across global clustering and biclustering algorithms. Many still use random initialization for other parameters as well. Biclustering algorithms also heavily depend on the data input. The type of data used, whether binary, numerical, categorical, or other, and the expected structure (e.g. hierarchical, overlapping, etc. . .) of the biclustered data heavily impact algorithm and parameter selection. Biclustering algorithms are sensitive to noise in the data, especially at higher dimensionalities. In most cases, the analyst's domain knowledge plays an important role in learning about the overall structure and parameters.

Selection of the proper algorithm and associated set of meaningful parameters requires ex-

ploratory analysis. The Visual Statistical Data Analyzer (VISDA), described by Zhu, et. al in [9] was developed to address the shortcomings of traditional clustering methods, specifically for genomic data analysis. VISDA is designed as a comprehensive and flexible clustering tool for modeling, visualization, and exploration. The high-level approach of VISDA is to perform progressive and divisive hierarchical clustering. It uses a combination of hierarchical mixture modeling, guidance from user knowledge, and both supervised and unsupervised feature selection and data projections. Hierarchical mixture modeling is an approach for gleanng statistical information about the subclusters within a larger cluster assuming said subclusters exhibit a hierarchical structure within the overall cluster. The incorporation of user knowledge in the selection of cluster centers and appropriate projections ensures the visualisations seem less like the output of some arcane black box and more representative of the real-world structure of the data. VISDA's approach allows it to explore the often hidden cluster structures so commonly present in complex HDD [9]. VISDA may also be employed to discover non-hierarchical flat structures, as this thesis will demonstrate. VISDA offers its end user a detailed visualization of the structure of their data, made more accessible to them through their expertise-backed input.

Previous versions of VISDA exist in C++ with Java for the visualization pieces, and Matlab [8, 9]. While fast if optimized properly, C++ is a lower-level language than Matlab and requires much more programming expertise. Additionally, updates to the C++ version would potentially require changes in two different programming languages. The Matlab version is the more recent and is a single-language implementation. Matlab is an incredibly

powerful high-level language and development environment for modeling and simulation, rapid prototyping, algorithm development, and many other use cases. The relatively low bar of programming expertise required, robust and well-documented libraries, and fully contained visualization capabilities make it a very attractive option. The main drawbacks are that it is much slower than C++ and can be prohibitively expensive to obtain a license, with the many often required toolboxes only available at added cost. Finally, Matlab is not widely used outside of the engineering disciplines, ensuring that users would need much more justification for the cost than the use of a single tool. Migration of VISDA to a more comprehensive, accessible language would greatly improve its usability and maintainability.

The purpose of this thesis research is to develop a more deployable and useable version of VISDA and to propose a new way to use VISDA in tandem with biclustering methods. The rest of the thesis covers the development of a new version of VISDA—including a detailed discussion of its main components, operation, code structure, individual functions, testing, and proof-of-concept results—and a demonstration of using exploratory analysis for bicluster algorithm and heuristic selection. The code developed for the project is available upon request.

Chapter 2

Development of VISDApy

2.1 Design Principles

The new version of VISDA retains the previous versions' design principles of offering a variety of complementary projections and a hierarchical exploration process.

2.1.1 Complementary Projections

VISDA provides five complementary projection methods for exploring the data. They are Principal Component Analysis (PCA), PCA-Projection Pursuit Method (PCA-PPM), Locality Preserving Projection (LPP), Hierarchical Clustering/K-means Clustering/Standard Finite Normal Mixture Discriminatory Component Analysis (HC-KMC-SFNM-DCA), and Affinity Propagation Clustering-DCA (APC-DCA). The user sets the availability of each

method at initialization of VISDA, and determines which one of these will be used at each level for each upper-level cluster [9]. For each method, the projection is given by

$$x_i = W_k^T(t_i - u_{t,k}) \quad (2.1)$$

where x is the projected data point in the new space, u is the mean of cluster k from the original space, and W_k is the projection matrix from the chosen method [9]. VISDA uses the first two components from each projection for the subspace in question, with the second orthogonal to the first [9]. These two components account for the greatest separability in the data as defined by each projection.

Each projection method has its own strengths in revealing different structures. As the most basic method, PCA projects in the directions of greatest variance within the data to reveal the possible subcluster structure [9]. PCA-PPM refines the results of PCA to favor distributions without well-defined peaks (i.e. flatter with shallow tails) [9]. It keys on the kurtosis of the projected data from PCA, which may better reveal the subcluster structure than simply the variance. PCA-PPM selects the projections with the two smallest kurtoses, which indicate they are the most platykurtic (i.e. have the shallowest tails). This indicates the presence of additional subcluster structure, as opposed to a single peak with deep tails [9, 5]. LPP reveals the neighborhood structure of the data, which emphasizes the data points with the greatest number of nearest neighbors [9]. LPP minimizes a compactness cost function that is obtained from summing the square distances of each possible pair of points, applying a

larger weight to smaller distances and vice versa. This preserves local structures better than the other methods [9].

The DCA-based methods both reveal apparent discrimination among subclusters, but using different unsupervised clustering approaches [9]. With HC-KMC-SFNM-DCA the user determines the subcluster number, while APC-DCA automatically determines it [9]. HC-KMC-SFNM-DCA uses the unsupervised methods of agglomerative Hierarchical Clustering, K-means Clustering and Standard Finite Normal Mixtures sequentially to generate a partition of the data, which is then used with the supervised method of Discriminatory Component Analysis to visually validate partition separability. The idea is to confirm whether the partition given by the unsupervised methods accurately describes the subcluster structure [9]. Affinity Propagation Clustering-DCA (APC-DCA) uses a similar unsupervised/supervised hybrid approach, but uses a different clustering method. With APC, each data point represents a node in a network that recursively passes messages to other nodes. Messages communicate the affinity a node has for another node as a subcluster center, while minimizing the dissimilarity between nodes and centers [9].

2.1.2 Hierarchical Exploration

At initialization, VISDA treats every dataset in its entirety as the top level of a hierarchy. This means that level 1 has a single cluster. It then successively subdivides the clusters at each level until the full structure is complete [9]. For each cluster or subcluster at a particular

level, the user will select one of the presented projections. An effective user will have domain knowledge and some knowledge of data science [9]. Such a user would know to select the projection that shows clearly separated clusters, that does not have clusters only composed of outliers, and that coincides with domain knowledge.

Figure 2.1 depicts a detailed program flow diagram for VISDA. For each cluster in the current level of the hierarchy, the user selects the best projection of the data. Then, the user must select the set of candidate clusters by clicking where their centers lie in the data, determining the number of clusters, k , with their first selection. VISDA then encourages the consideration of $k-1$ and $k+1$ clusters. The user selects the proper cluster model based on a Minimum Description Length (MDL) calculation for each model. The smallest MDL is typically the best choice, although VISDA provides the flexibility for the user to choose otherwise. If multiple subclusters appear the user advances to the next level, which generates one new projection per subcluster. The process repeats until the user decides to stop.

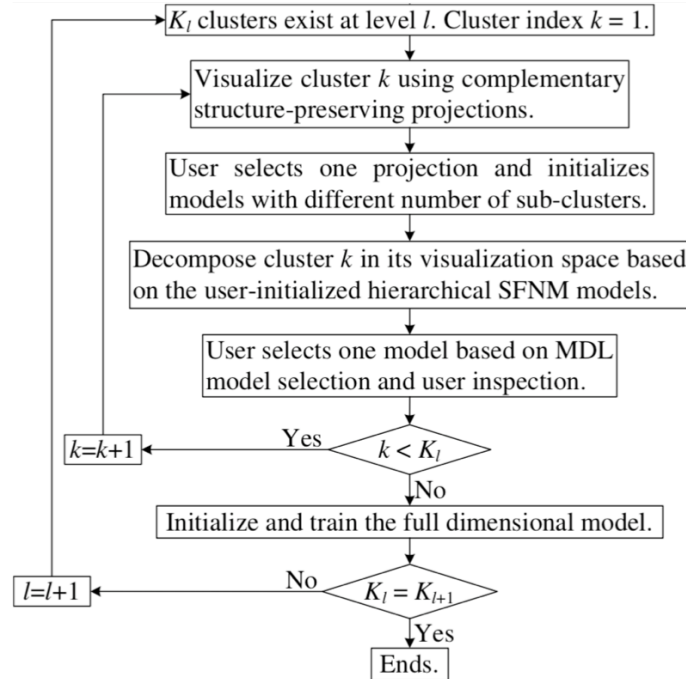


Figure 2.1: VISDApy's Program Flow Diagram, Adapted from [9]

2.2 Design of VISDApy

VISDApy fulfills this thesis project's research objective of developing a more deployable, maintainable, and usable version of VISDA. It is based on the Matlab version of VISDA, which comes with three different modules:

- VISDA_for_unsupervised_clustering
- VISDA_for_phenotype_clustering
- Unsupervised_GS

Currently, VISDApy only implements VISDA_for_unsupervised_clustering, but can easily integrate the other two modules if they are translated into Python. VISDA_for_phenotype_clustering is a supervised clustering technique using known classes (i.e. phenotypes) while Unsupervised_GS is essentially unsupervised feature selection. While these techniques have

yet to be implemented in VISDApy, they use some common functions that have already been implemented as part of the main module.

VISDApy is written exclusively in Python 2.7 and its supporting libraries, which are all open source. Even the interactive visualization capabilities are exclusively Python. VISDApy relies on a few extra libraries beyond those included in basic Python 2.7. This implementation uses Numpy and Scipy for matrix math, Sklearn and Scipy for various machine learning algorithms, and the pyplot and widgets modules of Matplotlib for visualization and interactions.

Miscellaneous libraries include:

- math
- copy
- time
- pickle (for image storage)
- os
- sys
- division

As VISDApy was translated from the Matlab version of VISDA, much of the overall architecture in terms of function dependencies and algorithmic operations remains the same. The major exception is that all the functions for one technique (i.e. Unsupervised vs. Supervised vs. Feature Selection) are organized within that technique's module. Figure 2.2 describes VISDApy's architecture as consisting of four classes of functions. The Main and Utility functions initialize VISDApy and maintain the state of the data if clusters vanish. The Visualization functions provide visual output and receive user input. The Projection

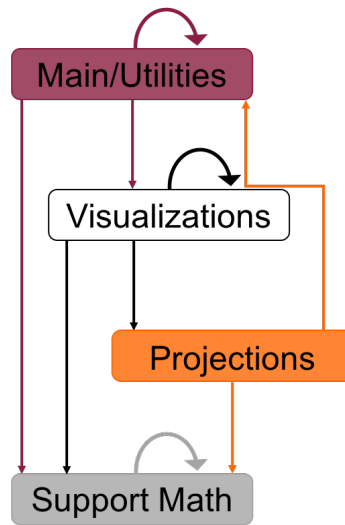


Figure 2.2: VISDApy's Functional Architecture

functions calculate all of the data projections the user has enabled. The Supporting Mathematical functions provide any necessary and repeated mathematical support too complicated to include as part of any other function. An example of this would be the expectation maximization algorithms. Function descriptions and dependencies are included to aid the reader in understanding how VISDApy works from both a software and mathematical perspective.

A close reading of [9] and [8] will further benefit the reader in this endeavor.

The Main and Utility functions are itemized here for the reader. Table 2.1 contains the dependencies for these functions.

- $ve_run(D, thre_Zjk, projection1, projection2, projection3, projection4, projection5, filename, min_num_point)$ is the main function of VISDApy that runs sample clustering and gene clustering. D is the data matrix to be clustered, the rows of which are data points to be clustered. The input $thre_Zjk$ is the threshold which a point's posterior probability of belonging to a particular cluster must exceed for that point to be dis-

Table 2.1: Dependency Table for Main and Coordinating Functions.

Function	Parent Functions	Child Functions
<i>ve_run()</i>	(Calling script or command line)	<i>ve_3to1proj()</i> <i>ve_mselect()</i> <i>SL-EM()</i> <i>Clear_Up_vk()</i> <i>Draw_Hierarchy()</i> <i>ve_mvis()</i> <i>ve_mselect2()</i>
<i>Clear_Up_vk()</i>	<i>ve_run()</i>	
<i>Process_Label_For_Empty_Class()</i>	<i>ve_hc_dca()</i>	

played in the plot of said cluster's projection. It takes a value between 0 and 1. The *projection* inputs correspond to the projection set the user wishes to explore. From 1 through 5, the projections that may be enabled are PCA, PCA-PPM, HC-KMC-SFNM-DCA, LPP, and APC-DCA. Values for these inputs are 1 for enabled and 0 for not enabled. The input *filename* is a string that will name the folder into which all results will be saved, and *min_num_point* is the minimum number of points that must be in a cluster. If cluster size is reduced below *min_num_point*, the cluster vanishes. This function coordinates VISDApy's activities using *ve_3to1proj()*, *ve_mselect()*, *SL-EM()*, *Clear_up_vk()*, *Draw_Hierarchy()*, *ve_mvis()*, and *ve_mselect2()*. It should be called from within a Python script after importing the module `VISDA_for_unsupervised_clustering`.

- *Clear_Up_vk(varargin)* clears the *vk* variables of discontinuities created by vanishing clusters. The *vk* variables are responsible for preserving the information about the structure of the data while traversing the levels of the hierarchy. If, for example, cluster 2 of 3 vanishes, the *vk* variables must be transformed from *v1* and *v3* to *v1* and

v2 for the *Draw_Hierarchy()* function to work properly. This function takes a Python dictionary as *varargin* for input and is called by *ve_run()*.

- *Process_Label_For_Empty_Class(label_in)* is essentially a clear-up function used in the successive clustering process for the HC-KMC-SFNM-DCA projection that ensures the cluster labels are continuous in the event one of them has vanished. The input *label_in* may contain empty classes, and *label_out* has them removed and subcluster labels adjusted to reflect the removal. For example, if hierarchical clustering gives results for 5 subclusters, but k-means assigns samples to only 4, this function updates the list of subcluster labels for each sample to reflect only 4 valid subclusters. This function is called by *ve_hc_dca()*.

The Visualization functions are itemized here for the reader. Table 2.2 contains the dependencies for these functions.

- *ve_vis_t(D, w, labels, Zjk, thre_Zjk, blobsize)* plots a single visualization projection. *D* is the data, *w* is the projection matrix composed of the top two components, *labels* contains which sample belongs to which cluster, *Zjk* contains the posterior probabilities of a sample belonging to a particular cluster, *thre_Zjk* determines whether a point will be displayed based on its *Zjk*, and *blobsize* is not used. The function does not return anything, but plots to the current axes. It is called by *ve_3to1proj()*, *ve_mselect()*, *ve_mselect2()*, and *ve_mvis()*.
- *ve_3to1proj(level, data, Zjk, thre_Zjk, cov_cluster, cluster_index, Zjk_whole)* plots all

Table 2.2: Dependency Table for Visualization Functions.

Function	Parent Functions	Child Functions
<i>ve_vis_t()</i>	<i>ve_3to1proj()</i> <i>ve_mselect()</i> <i>ve_mselect2()</i> <i>ve_mvis()</i>	
<i>ve_3to1proj()</i>	<i>ve_run()</i>	<i>ve_vis_t()</i> <i>ve_pca()</i> <i>ve_sub_pca()</i> <i>ve_pca_ppm()</i> <i>ve_sub_pca_ppm()</i> <i>ve_hc_dca()</i> <i>ve_lpp()</i> <i>ve_hc_dca-APC()</i>
<i>ve_mselect()</i>	<i>ve_run()</i>	<i>SL-EM()</i> <i>ve_vis_t()</i> <i>elipsnorm()</i>
<i>ve_mselect2()</i>	<i>ve_run()</i>	<i>Two-L-EM()</i> <i>ve_vis_t()</i> <i>elipsnorm()</i>
<i>elipsnorm()</i>	<i>ve_mselect()</i> <i>ve_mselect2()</i>	
<i>ve_mvis()</i>	<i>ve_run()</i>	<i>ve_vis_t()</i>
<i>Draw_Hierarchy()</i>	<i>ve_run()</i>	

of the selected projections for each cluster at each level so the user may select the best one. The input *level* corresponds to the hierarchical level, *data* is the input samples, *Zjk* is the posterior probabilities of the samples belonging to the current cluster, *thre_Zjk* is the threshold posterior probability a sample must have for it to display, *cov_cluster* is the covariance matrix of the cluster, *cluster_index* is the index of the cluster at the current level, and *Zjk_whole* contains the posterior probabilities of the points belonging in any of the clusters on the current level. The function uses all the projection calculation functions as well as *ve_vis_t()* to plot them. It is called by *ve_run()* and returns *top2components*, which contains the top two components of each projection calculation.

- *ve_mselect(level, D, W, Zk, thre_Zjk)* executes the top level model selection. The input *level* is always set to 1, *D* is the data, *w* is the projection vector selected by the user, *Zk* is the posterior probability that a sample belongs to a particular cluster, and *thre_Zjk* is the threshold of posterior probability a sample must have to be displayed in the projection. For the top-level selection, *Zk* is set to 1 for all samples because the top level is assumed to be a single cluster to which all samples belong. This function is called by *ve_run()*. It relies on *SL_EM()* to find the maximum likelihood estimates of cluster statistics based on user-selected cluster centers, *ve_vis_t()* to visualize each cluster, and *elipsnorm()* to plot the predicted boundaries of each cluster. It returns the indices of the supercluster to which each subcluster at the current level belong as *vk*. For this function, *vk* is always 1 because it is only used at the top level. It also returns

the estimates of the means, covariances, and proportion mixes as *estmu*, *estcov*, and *weight*, respectively. This function is called by *ve_run()*.

- *ve_mselect2(level, D, W, MU, Zjk, thre_Zjk)* executes the sub-level model selection. The input *level* is the current level in the hierarchy, *D* is the data, *W* is the projection matrix for each cluster, *MU* contains the cluster centers of the upper level, *Zjk* contains the posterior probabilities of the upper level, and *thre_Zjk* is the threshold of posterior probability a sample must have to be displayed in the projection. This function relies on *Two_L_EM()* to find the maximum likelihood estimates of cluster statistics based on user-selected centers, *ve_vist_t()* to visualize the subcluster, and *elipsnorm()* to plot the predicted boundaries of each cluster. It returns the indices of the supercluster to which each subcluster at the current level belongs as *vk0*. It also returns the estimates of the means, covariances, and proportion mixes as *estmu*, *estcov*, and *weight*, respectively. This function is called by *ve_run()*.
- *elipsnorm(m, cov, level)* plots a contour of a bivariate gaussian density with mean *m* and covariance *cov*. The input *level* corresponds to the density level desired for plotting. This function is called by *ve_mselect()* and *ve_mselect2()*.
- *ve_mvis(level, D, WW, MU, Zjk, thre_Zjk)* generates subplots of all the subclusters at a single level. The input *level* determines the particular level of the plots while the inputs *D*, *WW*, *MU*, *Zjk*, and *thre_Zjk* are all passed to *ve_vist_t()* to plot each subcluster separately. This function is called by *ve_run()*.

Table 2.3: Dependency Table for Data Projection Functions.

Function	Parent Functions	Child Functions
<i>ve_pca()</i>	<i>ve_3to1proj()</i>	
<i>ve_sub_pca()</i>	<i>ve_3to1proj()</i>	
<i>ve_pca_ppm()</i>	<i>ve_3to1proj()</i>	
<i>ve_sub_pca_ppm()</i>	<i>ve_3to1proj()</i>	
<i>ve_hc_dca()</i>	<i>ve_3to1proj()</i>	<i>SL_EM()</i> <i>Two_L_EM()</i> <i>Process_Label_For_Empty_Class()</i> <i>ve_cov_Jain()</i>
<i>ve_lpp()</i>	<i>ve_3to1proj()</i>	
<i>ve_hc_dca_APC()</i>	<i>ve_3to1proj()</i>	<i>SL_EM()</i> <i>Two_L_EM()</i> <i>ve_cov_Jain()</i>

- *Draw_Hierarchy(vk_dict, Zjk, level)* plots the hierarchical structure of the data based on the projections and clustering chosen. The input *vk_dict* contains the indications of which upper level cluster owns a particular subcluster. *Zjk*, the posterior probability of a subcluster belonging to that cluster, is used to establish and display a weight, or intensity, of membership. Finally, *level* is used only to save off the visualization under the appropriate filename. This function is called by *ve_run()*.

The Projection functions are itemized here for the reader. Table 2.3 contains the dependencies for these functions.

- *ve_pca(data)* calculates the principal components for the top level using the conventional, variance-based PCA algorithm. The function returns the projected *data* as *data_pca*, the coefficients of the principal components as *pcs*, and the variances of the principal components as *latent*. This function is called by the main visualization rou-

tine, *ve_3to1proj()*.

- *ve_sub_pca(data, cov_cluster)* calculates the conventional PCA projection of the *data* for sublevels within the hierarchy. The main difference from *ve_pca()* is that this function uses the covariance *cov_cluster* estimated by expectation maximization corresponding to the subcluster immediately above the current level. This function generates the projection as *data_pca*, the coefficients as *pcs*, and the eigenvalues (i.e. variances) as *eig_pca*. This function is called by *ve_3to1proj()*.
- *ve_pca_ppm(data)* calculates the PCA-PPM projection of data. The calling function, *ve_3to1proj()*, passes the projected data from *ve_pca* so that this function can take the kurtosis of the components and rank order them. The output is *kurtosis_sorted_pca*.
- *ve_sub_pca_ppm(data, Zjk)* is similar in function to *ve_pca_ppm*, but is meant for sublevels within the hierarchy. Instead, the PCA projected *data*, which is taken from the *ve_sub_pca()* function, must be centered to the local cluster. The sample posterior probabilities, *Zjk*, are used for this calculation as well as for the kurtosis calculation. The main visualization function, *ve_3to1proj()*, also calls this function.
- *ve_hc_dca(level, data, cluster_index, Zjk_whole)* calculates the HC-KMC-SFNM-DCA projection. The input *level* is the current level in the hierarchy, *data* is the data to be clustered, *cluster_index* is the current cluster in the level, and *Zjk_whole* contains the posterior probabilities of the data points belonging to all the clusters at the current level. This function displays an interactive dendrogram of HC for the user to “cut”

a partition, thus selecting the desired subcluster number. It then performs KMC and SFNM fitting to obtain the partition as a known classification, and performs DCA to obtain the separability between the classes by maximizing the weighted Fisher criterion. This maximization is achieved via eigenvalue decomposition. As in the other projection methods, the two eigenvectors with largest eigenvalues are orthogonalized with the Gram-Schmidt Process. The output, *pro_vector*, contains these orthogonal vectors. This function relies on *SL_EM()* or *Two_L_EM()*, depending on the level, *Process_Label_For_Empty_Class()* to ensure cluster label continuity, and *ve_cov_Jain()* for SFNM fitting. It is called by *ve_3to1proj()*.

- *ve_lpp(level, data, cluster_index, Zjk_whole)* calculates the Locality Preserving Projection of the data at the hierarchical level *level* for cluster *cluster_index* by minimizing the weighted sum of the squared distances between each pair of points. Larger weights are assigned to smaller square distances (i.e. closer neighbors) and vice versa. *Zjk* contains the posterior probabilities of each sample belonging to cluster *cluster_index*. The function produces *pro_vector*, an affine projection matrix from the Gram-Schmidt process. It is called by *ve_3to1proj()*.
- *ve_hc_dca_APC(level, data, cluster_index, Zjk_whole)* calculates the APC-DCA projection. The input *level* is the current hierarchical level, *data* is the data points, *cluster_index* is the index of the current cluster for projection, and *Zjk_whole* contains the posterior probabilities that the samples belong to any of the clusters at the current level. This function relies on *apcluster()* to automatically determine subcluster

Table 2.4: Dependency Table for Mathematical Support Functions.

Function	Parent Functions	Child Functions
<i>Cal_Covar2()</i>	<i>ve_hc_dca()</i> <i>ve_hc_dca_APC()</i>	
<i>Cal_Mean2()</i>	<i>ve_hc_dca()</i> <i>ve_hc_dca_APC()</i>	
<i>multinorm()</i>	<i>SL_EM()</i> <i>Two_L_EM()</i>	
<i>SL_EM()</i>	<i>ve_run()</i> <i>ve_mselect()</i> <i>ve_hc_dca()</i> <i>ve_hc_dca_APC()</i>	<i>multinorm()</i>
<i>Two_L_EM()</i>	<i>ve_mselect2()</i> <i>ve_hc_dca()</i> <i>ve_hc_dca_APC()</i>	<i>multinorm()</i>
<i>ve_cov_Jain()</i>	<i>multinorm()</i> <i>ve_hc_dca()</i> <i>ve_hc_dca_APC()</i>	
<i>apcluster()</i>	<i>ve_hc_dca_APC()</i>	

number instead of a user-selected dendrogram partition. The rest of the process is to that of *ve_hc_dca()*. This function relies on either *SL_EM()* or *Two_L_EM()* depending on the current level, and *ve_cov_Jain()*. It is called by *ve_3to1proj()* and returns the orthogonal eigenvectors with the two greatest eigenvalues.

The Supporting Mathematical functions are itemized here for the reader. Table 2.4 contains the dependencies for these functions.

- *Cal_Covar2(data, label)* calculates *p_covar*, the covariance of each of the subclusters.

The input *data* contains the values for each sample and *label* contains the assigned subcluster for the particular samples. This function is called by *ve_hc_dca()* and *ve_hc_dca_APC()*.

- *Cal_Mean2(data, label)* calculates *p_mean*, the means of each of the subclusters. The input *data* contains the values for each sample and *label* contains the corresponding subclusters for each sample. This function is called by *ve_hc_dca()* and *ve_hc_dca_APC()*.
- *multinorm(x, m, covar)* calculates the value of a multidimensional Gaussian of mean *m* and covariance *covar* at the specific points in *x*. This function is called by the single and two-level expectation maximization functions *SL_EM()* and *Two_L_EM()*.
- *SL_EM(y, estmu, estcov, estpp, min_num_point, flag_allow_vanish)* calculates the maximum likelihood a posteriori estimates of the means, covariances, and mix proportions of each cluster. The input are as follows: *y* contains the data, *estmu* contains the initial mean estimates, *estcov* contains the initial covariance estimates, *min_num_point* is the minimum number of samples that can be in a cluster, and *flag_allow_vanish* indicates whether a cluster will be allowed to vanish in the EM process. There are two steps in the EM algorithm. The E step calculates the updated posterior probabilities by applying the latest mixing proportions to the Gaussian evaluated with *multinorm()* using the latest *estmu* and *estcov*. The M step then updates *estmu*, *estcov*, and *estpp* using these probabilities according to maximum likelihood. The current probabilities are compared to their previous values and if they exceed a set threshold, the algorithm iterates to another E step. Otherwise, the algorithm returns the parameter estimates *estmu*, *estcov*, and *estpp*, along with the latest posterior probabilities *indic* and *normindic* (normalized). The function also returns *id_record*, the record of the remaining clusters after EM, and flag *flag_allow_vanish* indicating a vanished cluster was not al-

lowed to vanish or nonconvergence. This function is widely used throughout VISDApy in *ve_mselect()*, *ve_run()*, *ve_hca_dca()*, and *ve_hca_dca-APC()*.

- *Two-L-EM(y, estmu, estcov, estpp, min_num_point, flag_allow_vanish, normindic_up)* runs the Expectation Maximization algorithm for *estmu*, *estcov*, and *estpp* at the sub-levels within the hierarchy. The main difference from *SL-EM()* is that this function uses the previous level's posterior probabilities, *normindic_up*, instead of calculating its own first. The output *normindic* is the normalized posterior probabilities, *indic* is the original posterior probabilities, *estmu* is the mean, *estcov* is the covariance, *estpp* is the mix proportions, *id_record* is the remaining clusters after any have vanished, and the *return_flag* indicates whether a cluster should have vanished or if the algorithm did not converge. This function is called by *ve_mselect2()*, *ve_hc_dca-APC()*, and *ve_hc-DCA()*. It relies on *multinorm()* for Gaussian probability evaluations.

- *ve_cov-Jain(cov_in)* calculates the inverse and determinant of the covariance matrix *cov_in* passed to it. There are two problems it seeks to solve in these calculations. The first is for singular matrices whose determinants are zero and thus do not have inverses. This is solved using an iterative energy assignment procedure to generate a new set of eigenvalues and, with the eigenvectors of the original input, construct a new and invertible covariance matrix. The second issue is a quirk of the computing environment in which some incoming values can be less than the minimum normalized floating point number. So, the function sets these to the *real_min* of the environment. This function supplies *inv_cov*, *cov_mat*, and *det_cov* to the *multinorm()*, *ve_hc_dca()*,

and *ve_hc_dca_APC()* functions.

- *apcluster(apclust_input)* performs affinity propagation clustering to automatically determine subcluster number for DCA. The input *apclust_input* is a Python dictionary containing similarities, S , in the form of pair-wise distances and preferences, P , which indicate the preference that a data point be chosen as the cluster center. The preferences are chosen in the calling function, *ve_hc_dca_APC()*, as the points with minimum pairwise distances, but the median is also a good choice. The similarity matrix, S , is an $N \times N$ matrix (for N data points) where $S(i,k)$ is the similarity between points i and k . The similarity $S(i,k)$ is not necessarily equal to the similarity $S(k,i)$. Each cluster is represented by its center, and this function searches for cluster to maximize a net similarity function. Net similarity is determined by the sum of the preferences of the cluster centers added to the sum of the rest of the data points' similarities to their cluster centers. The function returns the cluster centers and their assigned data points in *idx*, the sum of the similarities in *dpsim*, the sum of the preferences of each cluster center in *expref*, and the net similarity in *netsim*. This function has a host of other input parameters that modify functionality for sparsity, plot the intermediate results, etc... but are not used by the calling functions in VISDApy. It is called by *ve_hc_dca_APC()* only.

2.3 Demonstration of VISDApy

This demonstration uses VISDApy to perform sample clustering on a subset of muscular dystrophy data with 110 genes and 121 samples. To start VISDApy, run the following commands in a Python script:

```
import VISDA_for_unsupervised_clustering as vuc

vuc.ve_run(sampleData, 0.5, 1, 1, 1, 1, 0, 'resultFiles', 5)
```

This call declares that a point will display for a particular cluster if it has 50% chance of belonging to that cluster, and that a cluster will have at least 5 points. It also omits the APC-DCA projection, which will be examined in Section 2.4.

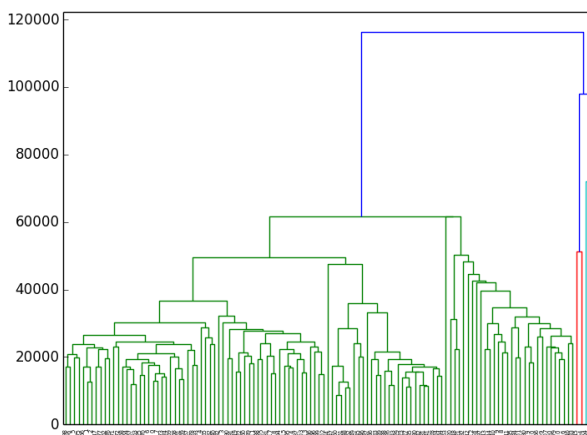


Figure 2.3: HC Dendrogram for Level 1

Figure 2.3 shows the top level dendrogram for HC-KMC-SFNM-DCA partition selection, which is the first image the user sees if they have enabled this projection. This dendrogram displays the links between nonsingleton clusters and their children. The length of each link

represents the distance between clusters and children. The cursor controls a horizontal red line allowing the user to cut the hierarchy into subclusters based on the distance at which the cut is made. The colors are added by Scipy's dendrogram package to indicate what it thinks the partitions should be.

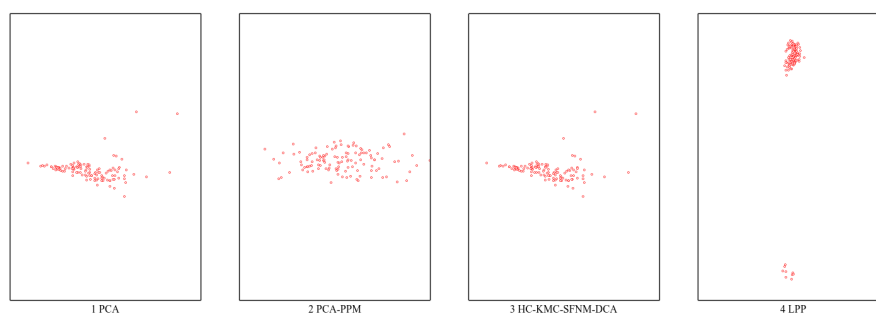


Figure 2.4: Cluster Projections for Level 1

Figure 2.4 displays the projections of the data at the top level for each projection type enabled. It acts as a menu so the user can select which best reveals cluster structure. The user enters the number of the selection to continue with the process. For this demonstration, the user has selected projection 4, LPP.

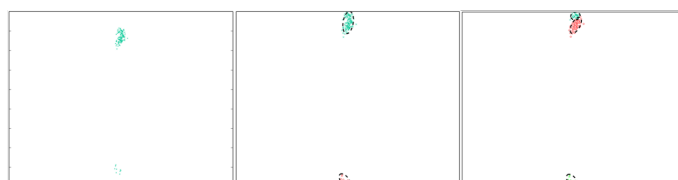


Figure 2.5: Subcluster Model Selections for Level 1

VISDApy requires the user to establish a subcluster model for the top level by selecting the number and locations of subcluster centers. The user has selected two clusters in the middle of Figure 2.5. To aid the user in thoroughly exploring the data, VISDApy requires selection

of one less cluster and one more cluster than the original selection. The left and right ends of Figure 2.5 show these selections. Future configurations of VISDApy might additionally require two less and two more clusters to better help the user discover the best subcluster number.

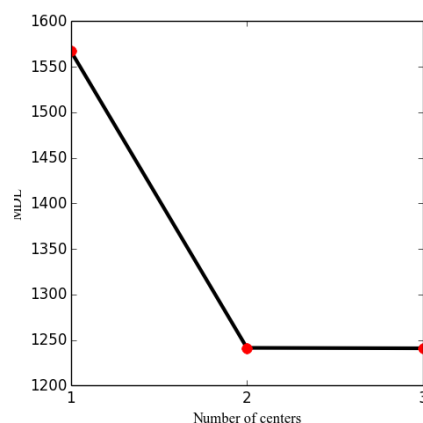


Figure 2.6: Minimum Description Length of Each Subcluster Model of Level 1

VISDApy aids the user in selecting the proper subcluster model by calculating the MDL of each set of cluster centers. As the smallest MDL is typically the best choice, Figure 2.6 shows that the three-cluster selection is the best choice.

VISDApy generates a summary visualization at each level depicting the projection selections for each upper level cluster. Figure 2.7 depicts the top level total visualization, which is just the single projection selection made. The data at the top level is treated as one cluster.

Since the top level is always one cluster, the program moves to the next level of the hierarchy. Figure 2.8 displays the HC dendrogram for just the first subcluster selected in the previous level. The user continues to make partitions for each subcluster and level, which gradually

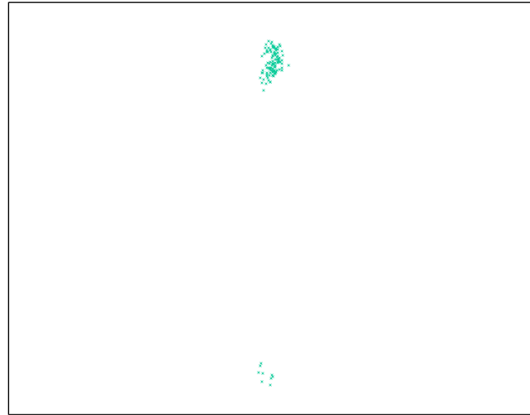


Figure 2.7: Total Visualization for Level 1 of the Hierarchy

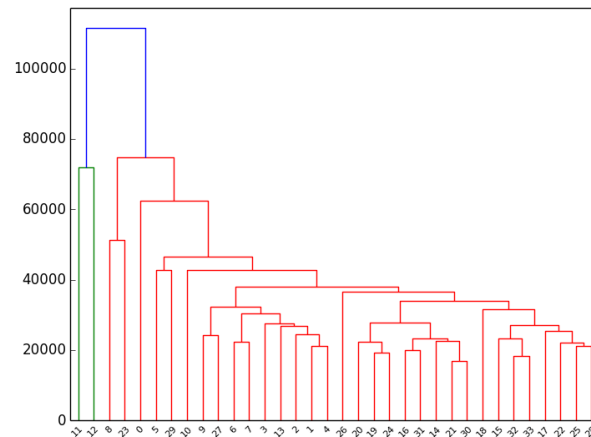


Figure 2.8: HC Dendrogram for Level 2 Cluster 1

reduces the span and distances of dendrograms within the same branch of the hierarchy.

Figure 2.9 displays the projections for level 2, cluster 1. The user has selected projection 2, PCA-PPM. This is likely not a good selection, as there were other projections with tighter clustering of points.

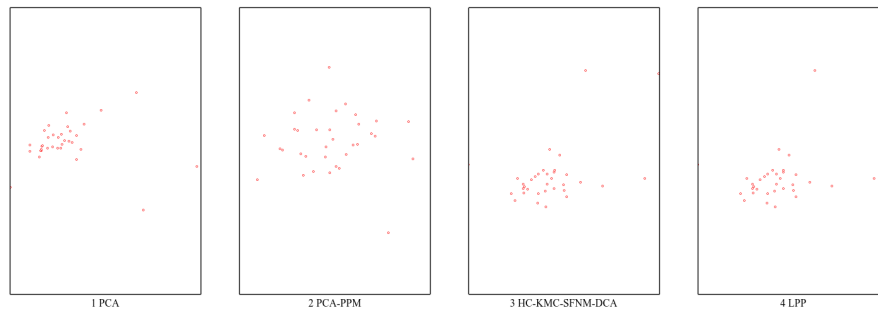


Figure 2.9: Projections for Level 2 Cluster 1

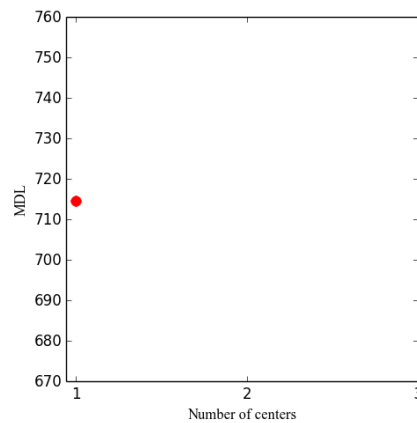


Figure 2.10: Minimum Description Length of Each Subcluster Model of Level 2 Cluster 1

The user selected only one cluster, which Figure 2.10 confirms.

Figure 2.11 displays the HC dendrogram for the second subcluster selected in the previous level.

Figure 2.12 shows the projections for the second cluster in level 2. The user has selected PCA, as it shows two or three closer clusters of points.

The user has selected three clusters, but the MDL plot in Figure 2.13 suggests two clusters are the best. The two clusters in Figure 2.14 reflect the user's selection. They overlap

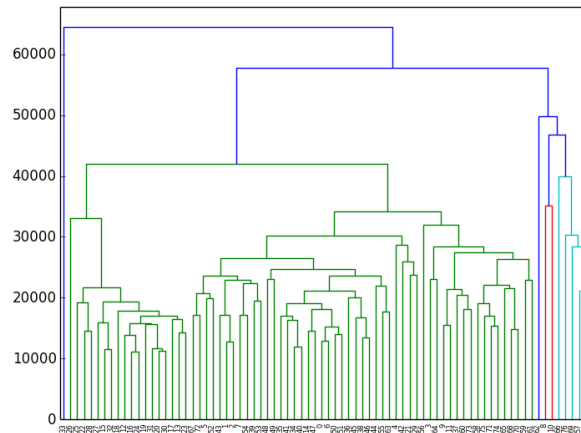


Figure 2.11: HC Dendrogram for Level 2 Cluster 2

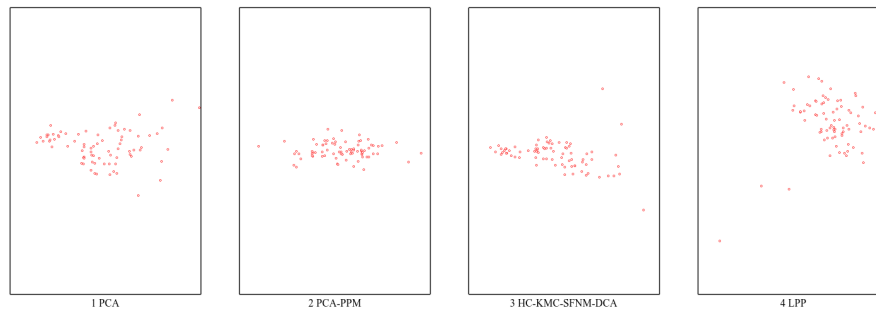


Figure 2.12: Projections for Level 2 Cluster 2

significantly.

Figure 2.15 displays the HC dendrogram for the third subcluster selected in the previous level. This seems to be the smallest of the three and will likely not yield any more interesting structure.

Figure 2.16 displays the projections for level 2, cluster 3. No projections show any meaningful structure. The user has selected PCA as the points seem closest together.

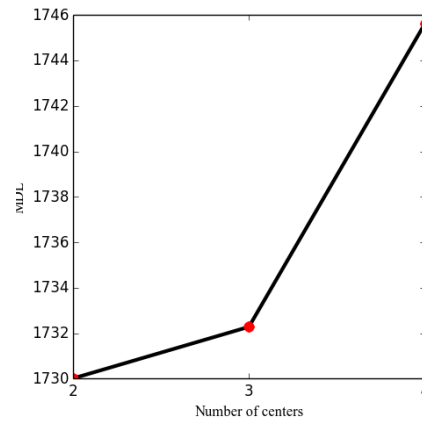


Figure 2.13: Minimum Description Length of Each Subcluster Model of Level 2 Cluster 2

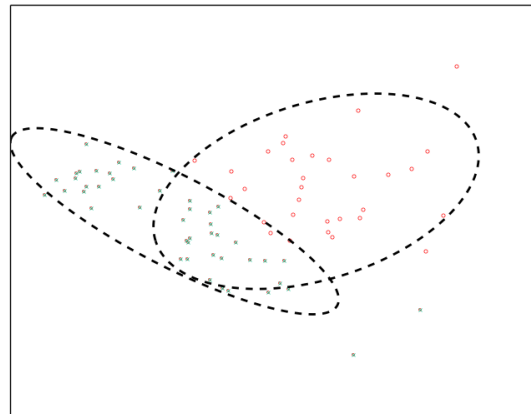


Figure 2.14: Level 2 Cluster 2 Subcluster Model Selections

Figure 2.17 displays plots the representative level 2 hierarchy found by VISDApy.

Figure 2.18 is a user-generated figure based on Figure 2.17 and all the selected projections in level 2.

After the user runs another round of projections and cluster model selections for level three, VISDApy produces the hierarchy in Figure 2.19. Figure 2.20 fully displays the hierarchy

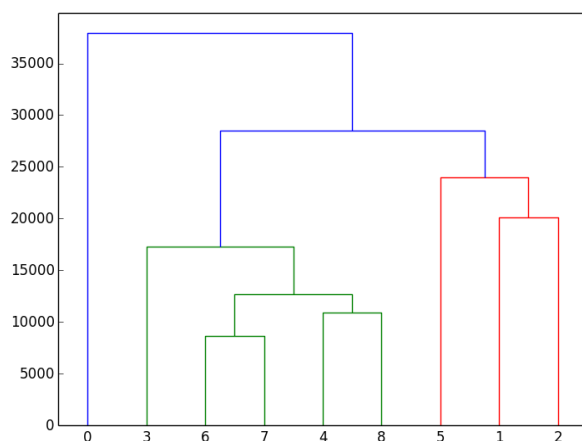


Figure 2.15: HC Dendrogram for Level 2 Cluster 3

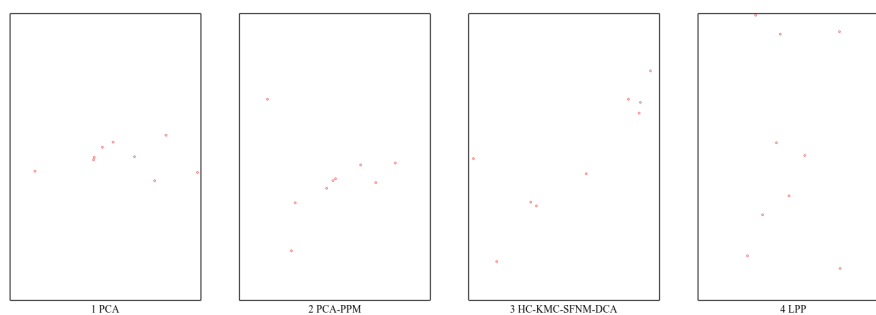


Figure 2.16: Projections for Level 2 Cluster 3

assembled by the user. This run of VISDApy has discovered four total clusters.

Note that projection and subcluster model selection matter when exploring a hierarchy. A dataset might require multiple runs through VISDApy to determine the most likely hierarchy. If the user had selected PCA instead of LPP in the first step and made other choices throughout the process, the results would look slightly different.

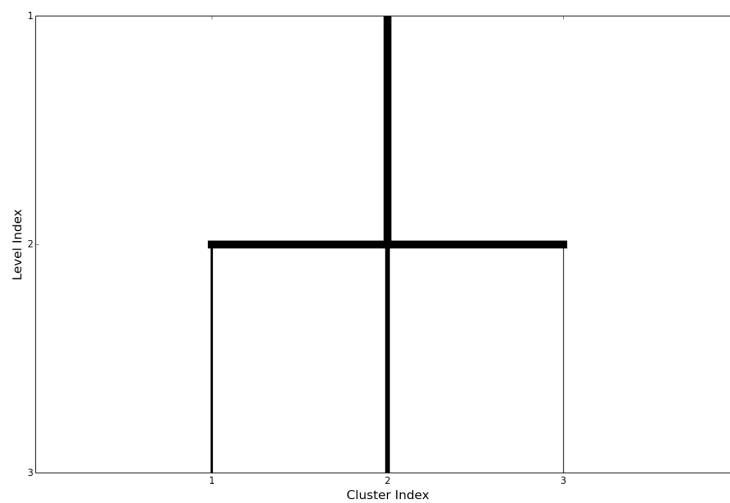


Figure 2.17: Level 2 Hierarchy Found by VISDApy

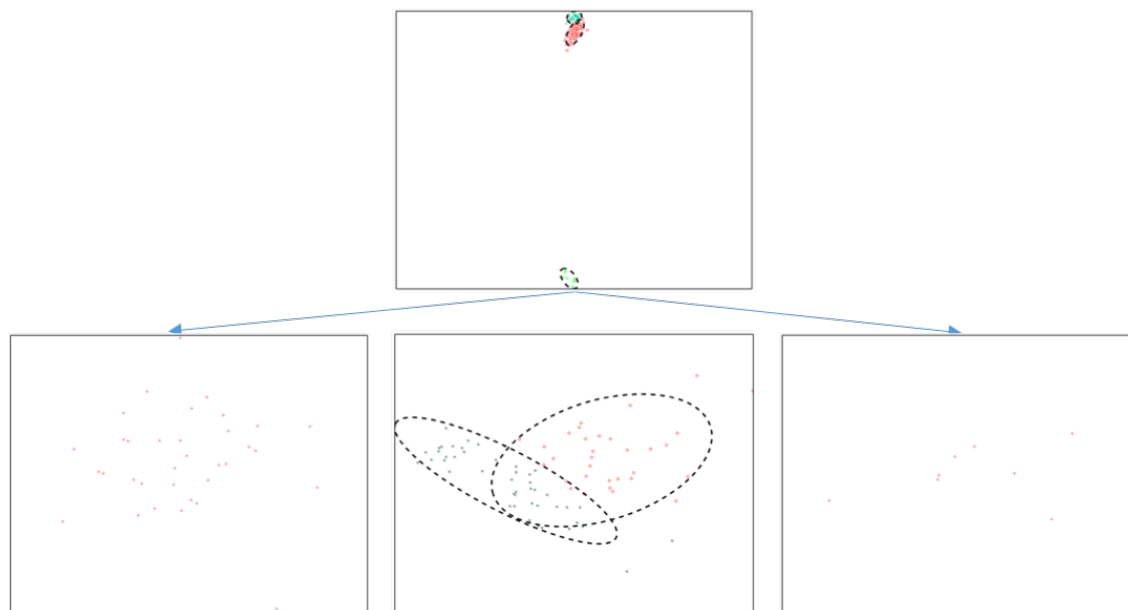


Figure 2.18: Level 2 Hierarchy with Cluster Images

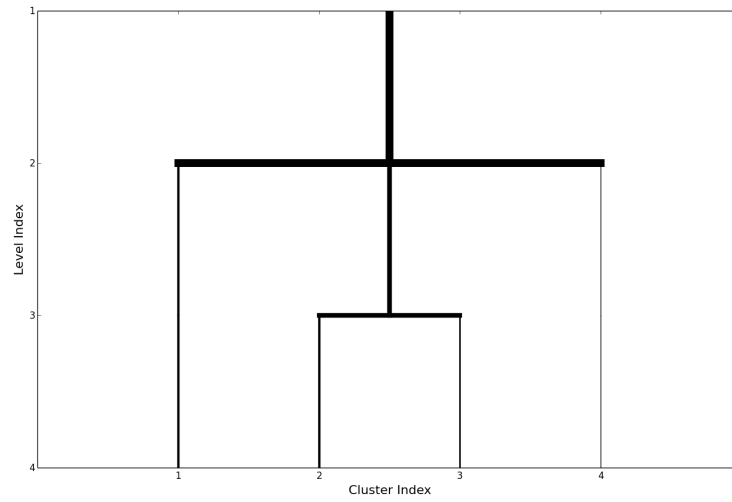


Figure 2.19: Level 3 Hierarchy Found by VISDApy

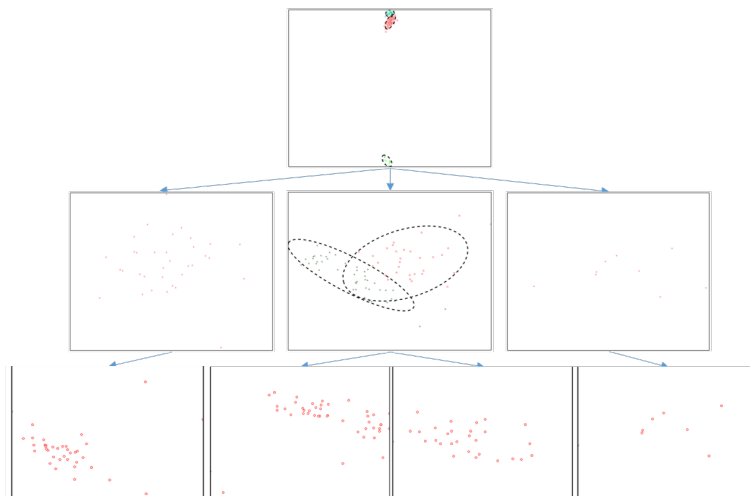


Figure 2.20: Total Hierarchy with Cluster Images

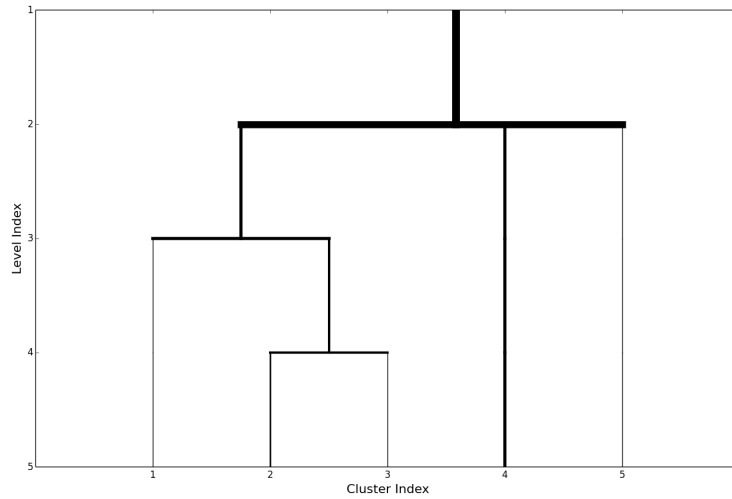


Figure 2.21: Alternate Hierarchy Found by VISDApy

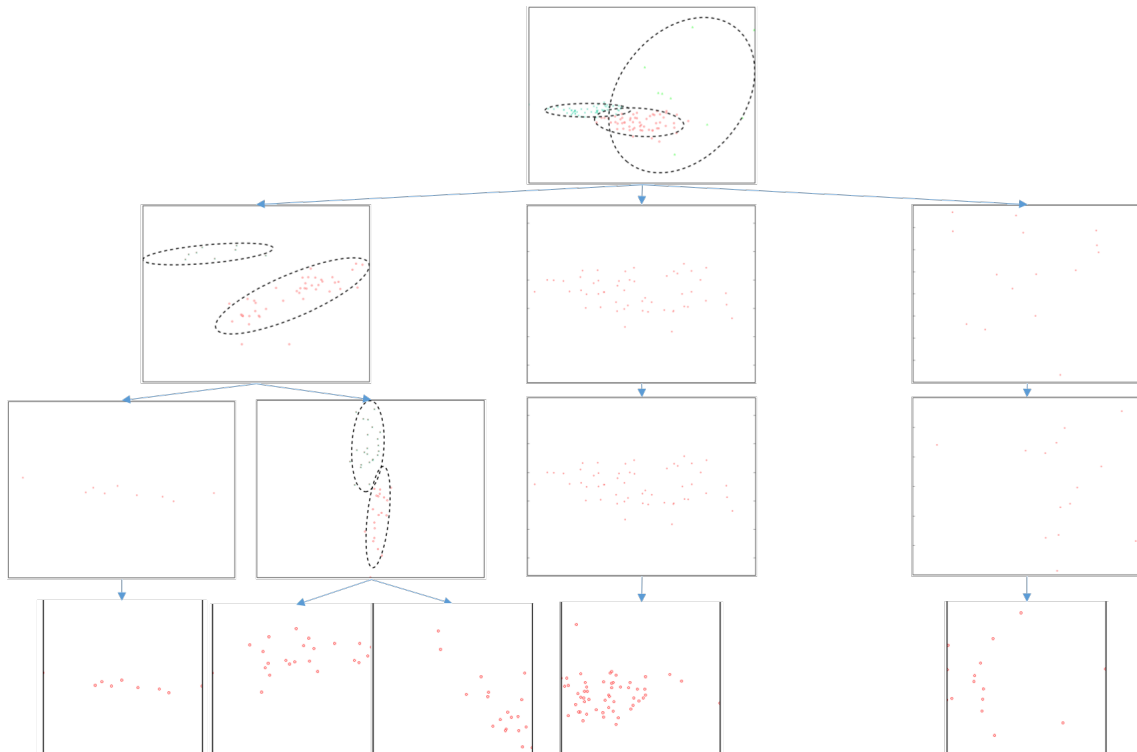


Figure 2.22: Alternate Total Hierarchy

Figures 2.21 and 2.22 show the alternate hierarchy found by VISDApy. Note that it extends into four levels and that the subclusters in the lower layers are more defined without overlapping with one another, while the top level model has a cluster with large overlap of one of the other clusters. VISDApy has found a total of 5 clusters, but with a large amount of overlap for the fifth cluster it may be safe to assume that only four true clusters have been found.

2.4 Testing VISDApy

The testing of VISDApy included both unit and integration testing against the results obtained from runs of the Matlab version of VISDA using sample data sets. The validity of the test results depends on proper functioning of the Matlab version of VISDA. The data for the unit tests was obtained by saving the inputs and outputs of each Matlab function to a persistent data structure and running VISDA on sample data sets. Just one run yielded multiple test cases for most functions on a variety of inputs. This also helps engage multiple logical paths within VISDApy that may not be explored with just one input. One significant finding during unit testing involved the use of eigendecomposition in many of the functions. The eigenvectors produced by the *eig()* function in Matlab differ from those produced by the *eig()* functions in both Numpy and Scipy by their normalization factors. The eigenvalues produced are mostly the same. These different results required closer manual inspection of the results of many functions beyond automated value comparison.

While unit testing focused on obtaining correct mathematical results, integration testing focused more on ensuring proper functionality of VISDApy. The two main reasons for this were the inconsistency in eigendecompositions cascading down the chain of functions, and the general insufficiency of rote value comparison in assessing visual outputs and program flow. Inconsistency of output format when using open source functions in Python caused many problems during integration testing. Matlab almost exclusively uses matrices. Python differentiates between scalars, arrays, and matrices. A function could produce a $(N,1)$ matrix, but the code operating on that result might require a $(N,)$ array. Ensuring proper indexing dominated integration testing for this reason.

There are two outstanding issues revealed through testing. Though *ve_hc_dca-APC()* passed unit testing, integration testing revealed that choosing this projection for some data sets caused posterior probabilities and thus data points to shift, or jump, hierarchical branches as the user traversed more levels of the hierarchy. This issue can be observed in running the demo in Section 2.3 with the APC-DCA option enabled. The level 1 cluster projections in Figure 2.23 for the same data include the APC-DCA projection. Note this projection has the best clusters of all the projections, with four clearly separated clusters. Assume that the user continues running VISDApy through its process, choosing the best projections along the way.

Figure 2.24 displays the final hierarchy found by VISDApy after including the use of APC-DCA where it provided the best clusters. It starts with four clusters at level 2 and finishes with 10 clusters in level 6. The thickness of the lines connecting parent clusters to their

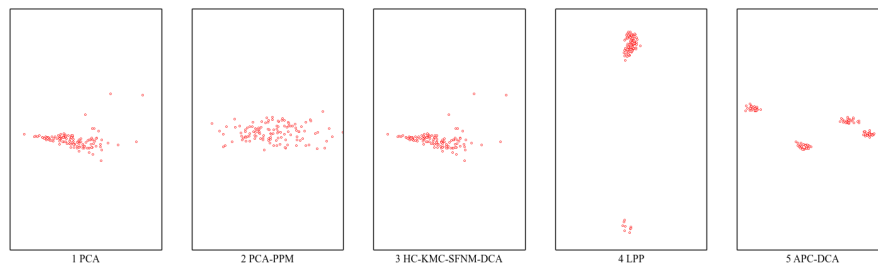


Figure 2.23: Cluster Projections for Level 1

children indicate the posterior probabilities of the children belonging to that parent. Note that the line for the second level is the thickest, while the fourth cluster takes on most of the probability. The fourth branch also siphons most of the data points away from the others as the hierarchy grows deeper. Finally, the fourth branch terminates with four clusters itself, which is the amount the initial APC-DCA projection found at the second level. It is also the same number that the other two iterations in Section 2.3 found. Furthermore, running the intermediate, level 2, data through VISDA for Matlab shows all of the sub-clusters vanishing. This most likely means that the APC-DCA has found all the salient clusters in its first attempt, and that VISDApy's EM algorithm needs further tuning for detection of vanished clusters.

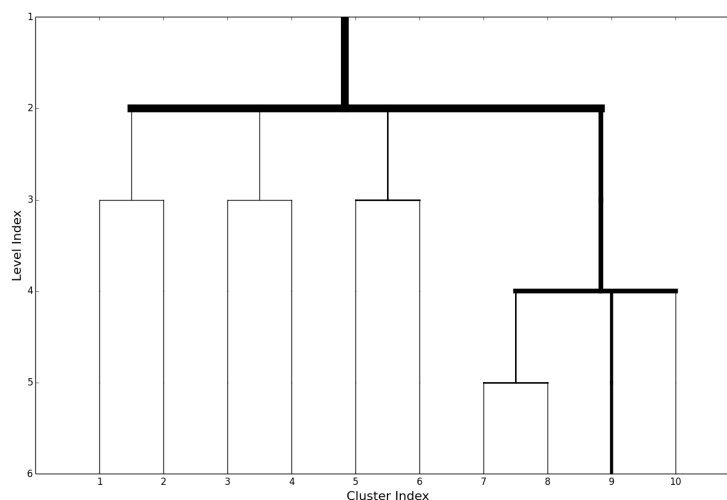


Figure 2.24: Alternate Hierarchy Found by VISDApy

The second issue is more minor. Both the Matlab version of VISDA and VISDApy shed the imaginary components from many of their calculations. This may need to be addressed in future iterations.

2.5 Future Improvements to VISDApy

There are a few near-term improvements that can be made to VISDApy. The first priority would be to further validate the use of APC-DCA in exploration of hierarchical data. This projection seems useful in finding tight clusters quickly without too much exploration and is a powerful component to VISDApy. Unsupervised Gene Selection and Supervised VISDA also still need to be implemented in VISDApy. The Gene Selection will especially help in picking the most relevant genes. Of course, this is a tradeoff in terms of information loss.

Nevertheless, it would make a valuable tool in a data analyst's toolbox. Finally, a user must run VISDApy through a Python script, which requires a minimal amount of experience using Python. A seamless interface, even if still through a command line interface, would improve VISDApy's usability by data analysts.

While translating from Matlab to Python has its own theoretical improvements in computational performance, additional opportunities exist for faster processing. Currently, VISDApy does not take advantage of multicore processing, even though it contains many loops and thus potential for parallelization. Integrating Python's *multiprocessing* module into VISDApy could be a quick win in improving performance on multicore machines. It uses distributed processing to separately calculate different iterations of loops, so care must be taken not to share data between separate processes. Multithreading may be another option, but this may be more of an intensive change.

During VISDApy's development, a web-based visualization library for Python called Bokeh was investigated. This had the potential to make VISDApy more seamless, especially if the algorithms could be hosted for online access. It would lower the bar for using VISDApy even further as all potential users would have to do is upload their data to a web server instead of downloading VISDApy, installing Python, and running everything locally. Unfortunately, Bokeh proved too slow for handling even moderately large data sets with VISDApy. Another near term, but more significant effort might be to find a way to use web-based visualization that is not prohibitively slow.

A longer-term consideration is that VISDApy was written in Python 2.7 instead of 3.x. This

was due to the ubiquity and library support of 2.7. Macs and Linux distributions come with 2.7 preinstalled. If in the future 3.x becomes the more preferred option, VISDApy might require translation into 3.x. There would likely be a moderate amount of syntactical changes necessary to do this. Luckily, many if not all of the libraries used by VISDApy are supported in 3.x.

Chapter 3

VISDApy for Bicluster Algorithm

Heuristic Selection and Structure

Identification

Biclustering algorithms depend on heuristics, such as estimated number of biclusters, to find biclusters in HDD. Additionally, the selection of algorithm itself depends on the expected structure of the data. The synthetic data set in Figure 3.1 has 100 samples and 1000 features. VISDApy takes in a randomly shuffled version of the data, Figure 3.2, and discovers a flat structure with five clusters in Figure 3.3. This information drives the selection of a spectral co-clustering algorithm with 5 biclusters. Figure 3.4 shows the results of that algorithm and configuration on the shuffled dataset.

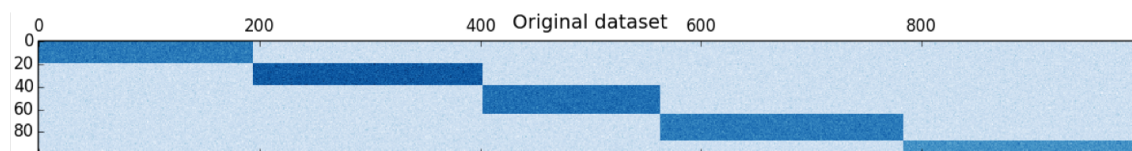


Figure 3.1: Synthetic Data with 5 Exclusive Biclusters

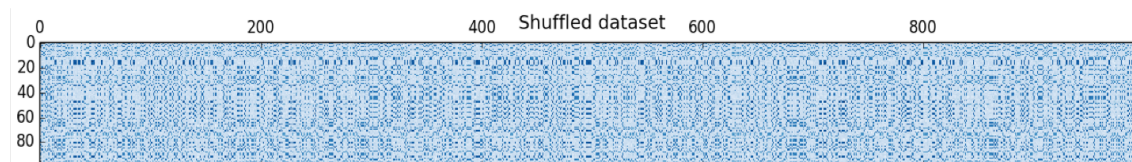


Figure 3.2: Shuffled Synthetic Data

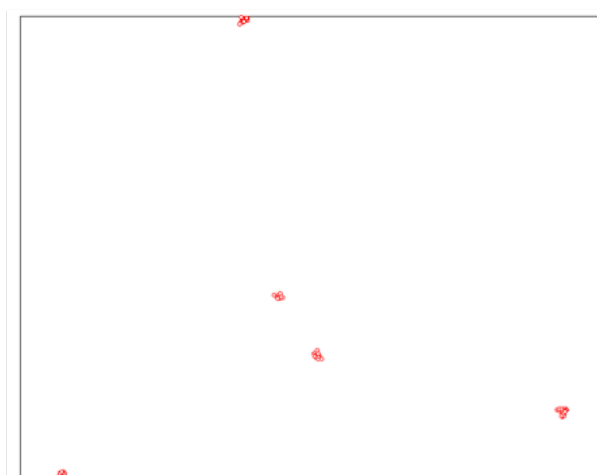


Figure 3.3: VISDApy Finds Flat Structure of 5 Biclusters

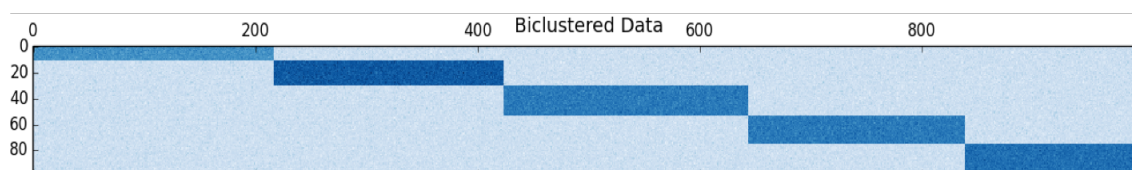


Figure 3.4: Biclustered Data from Spectral Coclustering Algorithm with VISDApy's Outputs

Figure 3.5 depicts a synthetic dataset with hierarchical biclusters of three levels and a lone exclusive bicluster. The exclusive bicluster is light on the color scale, as its value is close to zero. This will give an indication of VISDApy's ability to detect clusters in noise, which is

added in Figure 3.6. The noise is Gaussian with standard deviation 10. Figure 3.7 depicts the hierarchy found by VISDApy, which shows five total clusters. While it is able to pick out the weak bicluster, the noise is also pulled out as a cluster. It detects the hierarchy of the main bicluster fairly well, although the hierarchy might potentially be a little deeper.

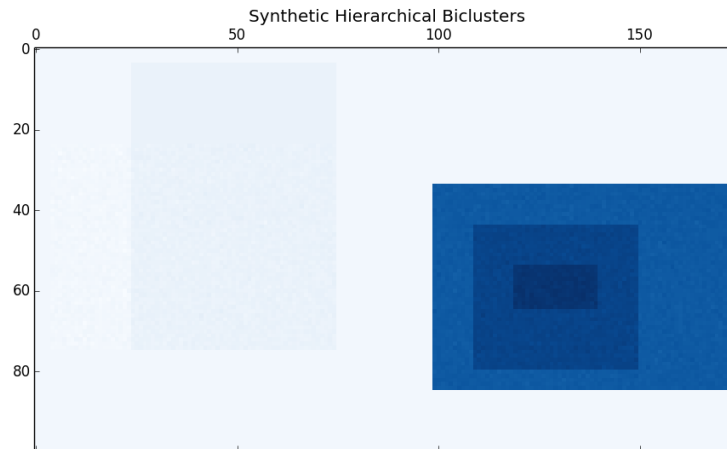


Figure 3.5: Synthetic Hierarchical Biclusters Mixed with Single Exclusive Bicluster

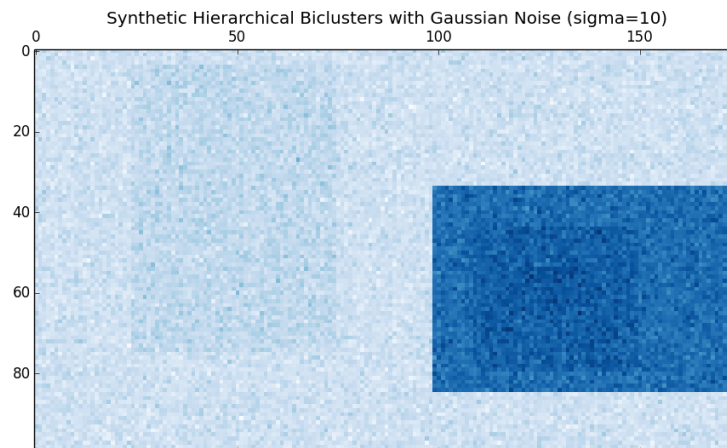


Figure 3.6: Synthetic Hierarchical Biclusters Mixed with Single Exclusive Bicluster and AWGN

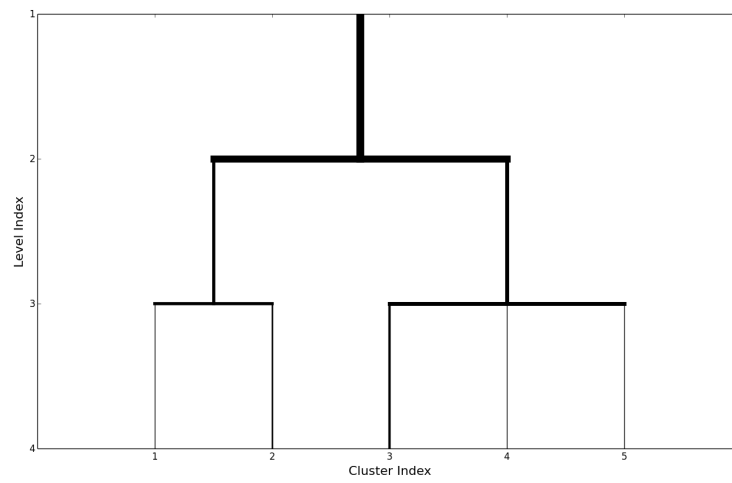


Figure 3.7: VISDApy Finds Hierarchical Structure

Chapter 4

Contributions

This thesis project accomplished a translation of VISDA from Matlab into Python, making it more deployable, maintainable, updateable, and usable. The open source nature of Python and its libraries as well as its level of programming made this possible. The project also proposed and demonstrated a way of using the information provided by VISDApy to aid in bicluster algorithm and heuristic selection. There is certainly more to be done with this work. In addition to the improvements to VISDApy outlined in 2.5, these are some questions for further study:

1. How well does VISDApy capture the various bicluster structures (i.e. hierarchical, overlapping, checkerboard, and nonoverlapping exclusive)?
2. How useful is the data VISDApy provides in giving the full range of heuristics required for biclustering algorithms to work well?

3. What are some other datasets to use with VISDApy?
4. Can VISDApy be adapted to handle data with missing values?

References

- [1] Dhammika Amaratunga, Javier Cabrera, Nolan Joy Perualila, Adetayo Kasim, and Ziv Shkedy. From Cluster Analysis to Biclustering. In Adetayo Kasim, Ziv Shkedy, Sebastian Kaiser, Sepp Hochreiter, and Willem Talloen, editors, *Applied Biclustering Methods for Big and High-Dimensional Data Using R*, pages 11–33. Taylor & Francis Group, Boca Raton, 2017.
- [2] Stanislav Busygin, Oleg Prokopyev, and Panos M Pardalos. Biclustering in data mining. *Computers & Operations Research*, 35(9):2964–2987, September 2008.
- [3] Robert Clarke, Habtom W Resson, Antai Wang, Jianhua Xuan, Minetta C Liu, Edmund A Gehan, and Yue Wang. The properties of high-dimensional data spaces: implications for exploring gene and protein expression data. *Nature Reviews Cancer*, 8(1):37–49, January 2008.
- [4] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Data Mining, Inference, and Prediction. Springer Science+Business Media, LLC, New York, second edition, 2009.

- [5] Aapo Hyvärinen, Juha Karhunen, and Erkki Oja. *Independent Component Analysis*. Adaptive and Learning Systems for Signal Processing, Communications, and Control. John Wiley & Sons, Inc., New York, USA, May 2001.
- [6] S C Madeira and A L Oliveira. Biclustering algorithms for biological data analysis: a survey. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1(1):24–45, January 2004.
- [7] Ziv Shkedy, Adetayo Kasim, Sepp Hochreiter, Sebastian Kaiser, and Willem Talloen. Introduction. In Adetayo Kasim, Ziv Shkedy, Sebastian Kaiser, Sepp Hochreiter, and Willem Talloen, editors, *Applied Biclustering Methods for Big and High-Dimensional Data Using R*, pages 1–10. Taylor & Francis Group, Boca Raton, 2017.
- [8] Jiajing Wang, Huai Li, Yitan Zhu, Malik Yousef, Michael Nebozhyn, Michael Showe, Louise Showe, Jianhua Xuan, Robert Clarke, and Yue Wang. VISDA: an open-source caBIGTM analytical tool for data clustering and beyond. *Bioinformatics*, 23(15):2024–2027, May 2007.
- [9] Yitan Zhu, Huai Li, David J Miller, Zuyi Wang, Jianhua Xuan, Robert Clarke, Eric P Hoffman, and Yue Wang. caBIGTM VISDA: Modeling, visualization, and discovery for cluster analysis of genomic data. *BMC Bioinformatics*, 9(1):383, 2008.