An Enhanced Approach to Dynamic Unsupervised Clustering

Christopher Heinlen, Mark Volpi
Lone Star Analysis
Addison, TX
cheinlen@lone-star.com, mvolpi@lone-star.com

Randal Allen
Lone Star Analysis
Orlando, FL
rallen@lone-star.com

ABSTRACT

Many approaches to Artificial Intelligence and Machine Learning (AI/ML) rely on the premise that there is well-labeled training data. In many scenarios, however, such reliance is not feasible or even possible. In those cases, unsupervised learning approaches attempt to automatically discern patterns or class types within the data. Applications that deal with streaming or batched data that contain unknown or evolving class types typically fall into this category. A problem with many off-the-shelf algorithms is that they are designed to look at a single batch of data in isolation. This can cause inefficiencies and inaccuracies when applied in streaming environments. Implementers are forced to incorporate some form of reconciliation or deduping of classes. Knowledge learned from historic data is not leveraged when analyzing current data.

Dynamic, Unsupervised Clustering by Algorithmic Thresholding (DUCAT), first presented at I/ITSEC 2023, is a system designed to handle noisy, streaming data. When DUCAT identifies new class types, it codifies them so new data entering the system can be quickly checked for inclusion in previously identified classes. This removes the need for an external mechanism to reconcile classes and reduces the amount of data that needs to be searched for emergent clusters. Additionally, DUCAT is designed for noisy environments and can maintain tight cluster definitions even in situations with large amounts of noise.

In the year since DUCAT was first presented, the algorithm has been improved to allow for higher dimensional clustering and increased overall performance. This paper describes those improvements and presents accuracy and timing comparisons to other clustering approaches.

ABOUT THE AUTHORS

Christopher Heinlen is a System Engineer at Lone Star Analysis with experience in AI/ML development, particularly within the realm of unsupervised clustering. He has also conducted significant research into signal processing within the domains of radar, PNT, seismology, and others. He earned a B.S. in Mechanical Engineering from the University of Texas at Arlington.

Randal Allen is the Chief Scientist of Lone Star Analysis. He is responsible for applied research and technology development across a wide range of M&S disciplines and manages intellectual property. He maintains a CMSP with NTSA. He has published and presented technical papers and is co-author of the textbook, “Simulation of Dynamic Systems with MATLAB and Simulink.” He holds a Ph.D. in Mechanical Engineering (University of Central Florida), an Engineer’s Degree in Aeronautical and Astronautical Engineering (Stanford University), an M.S. in Applied Mathematics and a B.S. in Engineering Physics (University of Illinois, Urbana-Champaign). He serves as an Adjunct Professor/Faculty Advisor in the MAE department at UCF where he has taught over 20 aerospace-related courses.

Mark Volpi is the Director of Intelligence Solutions for Lone Star Analysis. He is focused on the growth, management, and development of programs intended to solve problems faced by the United States Intelligence Community and their allies. He has twenty years’ experience in problem solving, system engineering, software development, real-time systems, and SIGINT processing systems. He has worked with international teams, and his experience has ranged from placing antennas in the desert to being PM on a multi-million-dollar contract. Mark received both his B.S. and his M.S. in Physics from Texas A&M, with an interest in information theory.

An Enhanced Approach to Dynamic Unsupervised Clustering

Christopher Heinlen, Mark Volpi
Lone Star Analysis
Addison, TX
cheinlen@lone-star.com, mvolpi@lone-star.com

Randal Allen
Lone Star Analysis
Orlando, FL
rallen@lone-star.com

INTRODUCTION & BACKGROUND

This paper presents significant advancements to Dynamic, Unsupervised Clustering by Algorithmic Thresholding (DUCAT), a novel system designed for unsupervised clustering in dynamic and noisy environments, highlighting its applications in real-time data analysis and anomaly detection. During IITSEC 2023, we presented *A Novel Approach to Dynamic Unsupervised Clustering* (Heinlen, Volpi, and Allen, 2023), which described our system for classifying noisy data streams without human intervention. Our system is designed to efficiently and accurately identify and define an unknown number of clusters, each with unknown features. Additionally, the system was developed to maintain high performance in a noisy environment. In an ideal world, we operate with training data, and we are able to look at a dataset in its entirety. However, we do not always have that luxury. In many real-time streaming applications, training data is not feasible due to the unpredictability of future clusters, and real-time classifications prevent the luxury of analyzing the data in aggregate. Our approach enables us to make classification decisions based on historical data and the clusters identified therein, while also allowing for the detection of emerging clusters in current and future data streams.

The system is comprised of four modules shown below in Figure 1. Data entering the system is first checked against previously found clusters. Data that does not belong to any existing classes is added to an unlabeled pool that is periodically processed with our unsupervised clustering algorithm to search for emergent clusters. When a new cluster is found within the unclassified pool, that cluster is given a concrete definition by the shell creation module which is then added to the library that new data is checked against upon entry to the system. Labeled data is aggregated and given a final review in an optional second pass module. This module serves as an opportunity to refine cluster definitions, merge or split classes, or leverage any *a priori* knowledge known about the data stream.

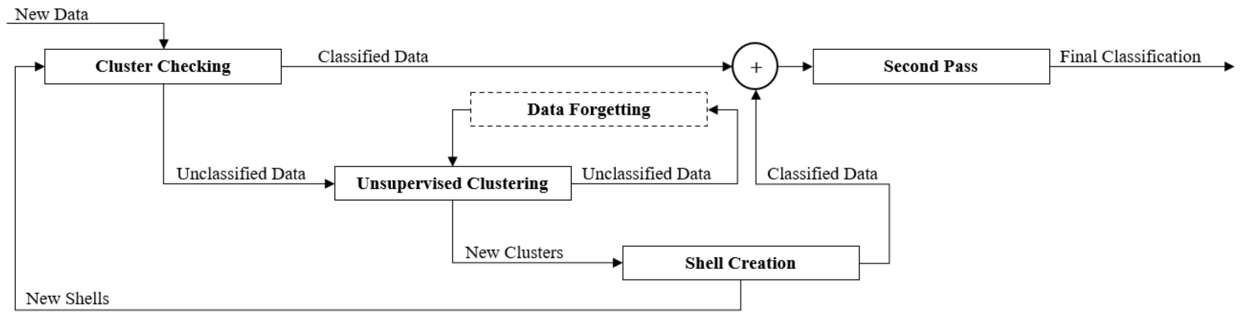

Figure 1. Data flow.

This system aims to tackle challenges observed in existing algorithms by incorporating features such as:

- memory of previously found classes,
- handling noisy data streams and maintaining accurate cluster definitions,
- the ability to find many, one, or no clusters in each search, and
- supporting real-time processing.

This system is applicable to a variety of domains, including education, simulation analysis, and performance monitoring. Our system could be leveraged in healthcare domains to cluster on patient symptoms and demographics to find macro trends or on the individual level it could be used to cluster on sensor data from a patient's vitals to detect anomalous behavior or find unobvious, high-dimensional trends. This system could be used by the intelligence community to automatically detect and categorize adversary's radar reserved modes by clustering on features from radar pulse descriptor words. At a high level, this system is designed for situations in which the categories present in the user's data are subject to change or the user does not have any insight into what the data may contain.

Our development over the past year has extended the capabilities of the system, specifically with respect to our unsupervised clustering algorithm. The first, and most obvious, limitation of the original implementation was the reliance on Delaunay triangulation.[1] Generating the triangulation can be done efficiently in the low dimensional spaces we worked in during our preliminary tests, but quickly becomes unfeasible as the dimensionality of the dataset increases. Our initial development from the version presented last year was aimed at finding a drop-in replacement for Delaunay triangulations that would generate similarly dynamic neighbor relationships but also scaled well to high dimensions. We successfully repurposed and extended computations from our Parzen Window Density Estimation[2] to achieve the same functionality previously provided by triangulation. With this, we were able to start doing real tests in high dimensions and with those tests more cracks began to show.

One of our primary concerns with this algorithm is a minimal and stable set of hyperparameters. That is, we want our default settings to produce good results on a wide range of datasets. Ideally, when we run our system on a data stream, we do not need to do any tuning to get good results. We allow for the fact that we can get *better* results with some tuning, but we want our defaults to perform well in any environment. The algorithm we presented last year had a stable set of hyperparameters in ~2–5 dimensions but that stability deteriorated when we extended to higher dimensions. As we worked to improve our algorithm, it began to resemble our original version less and less and began to converge, through conscious and unconscious efforts, on an algorithm that closely resembles Campello, Moulavi, and Sander's HDBSCAN (Campello, Moulavi, and Sander, 2013). While our current implementation shares a lot in common with HDBSCAN, we have not forgotten the reasons we began developing our own approach in the first place. HDBSCAN has shortcomings that become pronounced when used in a system like ours, and we have made additions and modifications to the algorithm to address those shortcomings. Additionally, we have spent a significant amount of time working to make our implementation as efficient as possible so that we can handle high data throughput in real time. To this end, the algorithm is implemented in C++ and much of it is parallelized. We use ParlayLib (Blelloch, Anderson, Dhulipala, 2020), a highly efficient scheduler and parallel algorithm implementation, as the backbone to our parallel infrastructure. Because the bulk of the development work over the past year has been directed toward advancing our unsupervised clustering algorithm, this paper focuses on that aspect of our system. For a description of the rest of the system architecture, please refer to our paper *A Novel Approach to Dynamic Unsupervised Clustering* from the 2023 I/ITSEC proceedings.

UNSUPERVISED CLUSTERING ALGORITHM

The core of our system is its unsupervised clustering algorithm. The logic of the algorithm builds off HDBSCAN and the implementation of the algorithm builds on the implementation described in *Fast Parallel Algorithms for Euclidean Minimum Spanning Tree and Hierarchical Spatial Clustering* (Wang, Yu, Gu, and Shun, 2021). At a high level HDBSCAN, and by extension our algorithm, uses a minimum spanning tree (MST)[3] to determine critical distance thresholds that correspond to candidate clusters gaining or losing data points and candidate clusters splitting or merging with each other. With the MST, a dendrogram,[4] or binary tree of potential clusters, can be constructed and then culled into our final designations. Our approach to each of these parts of the unsupervised clustering process, how the MST and dendrogram are used to make our cluster definitions, as well as how our process differs from the existing algorithms we have built upon, is discussed in detail below.

[1] At a high level, a triangulation subdivides the space occupied by a point set into simplices (in 2-dimensions, triangles) whose vertices are the points within the set. The Delaunay triangulation is a triangulation whose simplices' smallest interior angle is maximized. This avoids slivers and encourages equilateral simplices.
[2] The Parzen Window Density Estimation provides a way to determine a continuous density for a discrete set of points using a sliding window function.
[3] In graph theory, a graph refers to a group of vertices and the edges that link them. In our context the vertices are individual data points, and the edges are the distances between two data points. A connected graph is a graph in which there is a continuous path between any pair of vertices. A spanning tree is a subset of connected graphs in which there exists one, and only one, path between any pair of vertices. That is, there are no cycles or loops in the graph. Further, a minimum spanning tree is the spanning tree with the lowest combined edge weight possible.
[4] A dendrogram is a tree representation often used in hierarchical clustering to represent similarity in potential clusters. With a dendrogram it is possible to determine threshold(s) below which clusters are defined.

Minimum Spanning Tree

In the context of clustering, an MST is a useful structure because each edge represents the precise distance at which a point or group of points split away from a cluster. There is no need to guess at distance thresholds—every useful threshold is baked into the tree. To illustrate this, Figure 2 shows a simple, three-cluster dataset and its MST. To make cluster determinations, we can look at the edges of the MST, from largest to smallest, and analyze the points each edge connects. In the simple case below, it is easy to see that edges A and B are the critical edges whose removals make the three clusters distinct.

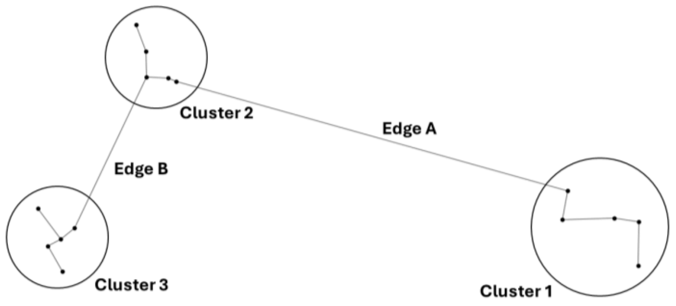

Figure 2 Simple dataset with MST.

From a computational standpoint, building the MST is by far the most expensive part of our clustering process, and increasing the efficiency of that process has been a critical focus in the development of the system. Like HDBSCAN, we use the concept of core distances and mutual reachability in our MST construction. This behavior is controlled by the *minimum points* hyperparameter. A data point's core distance is determined by the distance between itself and its *k*th nearest neighbor where k = *minimum points* and the mutual reachability of two data points is defined as the maximum value between each of the point's core distances and the actual distance between the two points or $|ab| = max(dist(a, b), a_{core}, b_{core})$.

The first step in the MST construction is to determine the core distance for each point within the dataset. To do this we follow Wang's implementation. We construct a *k*-d tree whose splits correspond to the median of the widest dimension of the subtree at that split. We can then use the *k*-d tree to efficiently find the *k*th nearest neighbor of each point and with that the core distance of each point.

Next, we use well-separated pair decomposition (WSPD)[5] to inform our edge search, this allows us to be judicious in our distance calculations and greatly reduce the amount of computation needed to generate the MST. During the tree building process we track three variables to limit our edge search for a particular iteration, β, ρ_{low}, and ρ_{high}. The algorithm flow to build the MST is as follows:

```
rho_low = 0; beta = 4;
while(tree.Incomplete) {
        rho_high = CalcRhoLimit(beta);
        while (edges > 0) {
                edges = GetEdges(beta, rho_low, rho_high);
                AddEdges(edges, tree);
        }
        beta *= 2.0;
        rho_low = rho_high;
}
```

[5] A set of points *A* is well-separated from point set *B* if for each set there exists a hypersphere of radius *r* such that each hypersphere fully encapsulates their point set and the hyperspheres are at least $s * r$ distance apart where $s > 0$. The well-separated pair decomposition of point set *S* is the series of well-separated pairs, $(A_1, B_1), (A_2, B_2), (A_3, B_3), \dots, (A_m, B_m)$, such that for any pair of points in *S* there exists only one pair, (A_i, B_i), in which A_i contains one of the points and B_i contains the other.

Each outer loop we double our β value and replace ρ_{low} with the value of ρ_{high} determined during *CalcRhoLimit*. *GetEdges* returns edges that connect two well-separated pairs whose combined size is less than or equal to β and whose edge weights are between ρ_{low} and ρ_{high}. These edges are then added to the MST.

CalcRhoLimit determines a new ρ_{high} value by finding the minimum possible bichromatic closest pair (BCP)[6] distance between any well-separated pairs whose combined size is greater than β. Because an exact BCP calculation is expensive, we use a lower bound given by the distance between the bounding boxes that enclose each set in the well-separated pair. In *GetEdges*, we look at all the well-separated pairs whose combined size is at most β and whose constituents belong to separate components[7] of the in-progress graph. While searching, we keep a record of the minimum connecting edge found so far in the current round for each component of the graph. If the minimum possible BCP distance between the two pairs is less than ρ_{high} and less than the current edge for any of the components contained in the pair, we calculate the exact BCP. If the exact BCP distances is less than ρ_{high} and the current best for either of the components that it connects we record it. We limit ourselves to calculating BCPs for pairs smaller than β because it is cheaper to find the BCP for smaller well-separated pairs. We limit ourselves to evaluating pairs whose potential BCP distance is less than ρ_{high} to ensure that we only add valid edges to the tree. Our MST building process uses the implementation presented by Wang as a starting point (a more detailed look at this approach can be found in that reference). We do stray from Wang's implementation in a few places and these deviations provide a significant reduction in computation time.

We are much more conservative in the number of full BCP calculations we do. During our edge search we maintain the current best edge for every component in the graph. With this information, when we consider a well-separated pair, we check whether its minimum possible BCP is less than the current best edge for any component within either set. At the end of *GetEdges*, we have a set of edges that *all* belong to the MST, whereas the implementation presented by Wang oversamples exact BCP calculations and each edge search returns edges that may or may not actually belong to the MST. Our implementation resembles Boruvka's algorithm[8] while Wang's implementation utilizes a batched Kruskal's algorithm[9] approach.

We also automatically add each point's nearest neighbor to the MST after calculating the core distance values for each point. We get the nearest neighbor for free with the core distance calculation and the nearest neighbor is always guaranteed to be a part of the MST, even when *minimum points* is greater than one. We have also streamlined some code paths, cache repeatedly used and expensive values, and made other similar bookkeeping enhancements.

Figure 3 shows timing comparisons[10] between Wang's implementation and our implementation for structured and unstructured data[11] with datasets of varying sizes. The code for the Wang implementation was run without modification except for a minor fix to its parallel buffer code that resulted in a segmentation fault and the removal of print statements. It should be noted that there are concepts described in their paper that are not implemented in the code, notably a method for caching previously calculated BCP's. Additionally, we were limited to running tests on

[6] The bichromatic closest pair of two disjoint (nonoverlapping) sets is the pair of points, one from each set, which are minimally separated.

[7] During the MST build process, the in-progress tree will be a spanning forest, or set of disjoint trees, and each tree is a referred to as a component. If each set in a well-separated pair is part of the same component, there is no need to further analyze the pair as the points within them are already fully connected.

[8] Boruvka's algorithm generates an MST in rounds by finding the minimum edge connection for each component in the existing graph. Through this approach, the number of components will decrease each round by at least half and the algorithm terminates when there is only a single component.

[9] At a high level, Kruskal's algorithm sorts candidate edges by weight, then iterates through each, from smallest to largest, checking whether adding that edge to the graph will result in a loop and adding the edge if it does not or discarding the edge if it does. Once the graph is fully connected, the algorithm terminates.

[10] These tests, and all other timing tests in this paper, were performed on a c5.4xlarge Amazon AWS EC2 instance running Amazon Linux 2023. This instance type has 16 vCPU and 32 GiB of memory.

[11] The structured dataset consists of three semi-overlapping gaussian clusters with varying standard distributions. These datasets were generated using sklearn's *make_blobs* function. The unstructured datasets are comprised of uniform distributions of noise. Five versions of each point-dimension pair for each dataset type were generated with different random seeds. Each point on the plot represents the average time to execute across ten trials for each of the five datasets for that point-dimension pair.

datasets with twenty or fewer dimensions as Wang's implementation hardcodes the number of dimensions and is not extensible to *n*-dimensions as written.

Our algorithm was at least 1.5 times faster (12,000 points, 5 dimensions unstructured data) and at most 24 times faster (1,500 points, 20 dimensions unstructured data). On average our implementation is 8 times faster for unstructured data, 5 times faster for structured data, 4 times faster in 5 dimensions and 9 times faster in 20 dimensions.

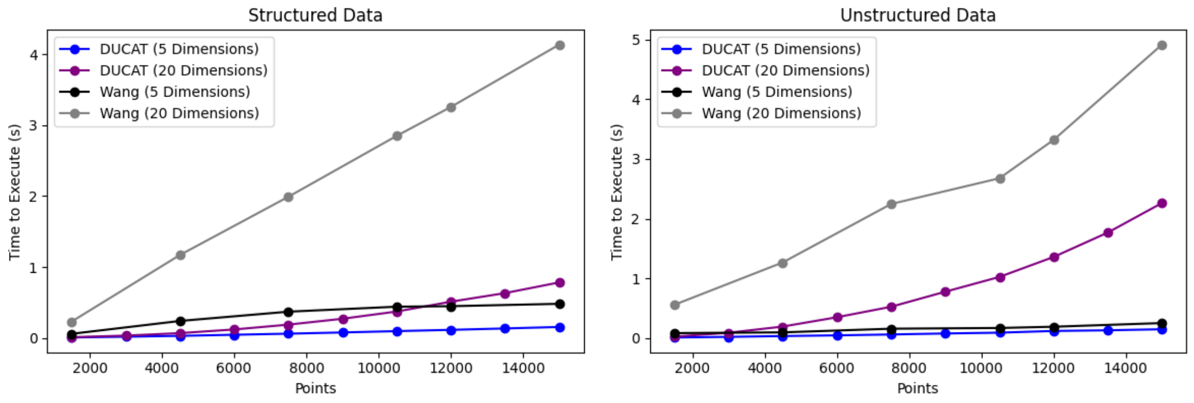

Figure 3 Execution time comparisons between the implementation provided by Wang and our revised algorithm.

Dendrogram & Cluster Culling

Once the MST is built, a dendrogram is formed from which we can make our cluster definitions. Taking the simple data set from Figure 2, we can build a dendrogram consisting of five candidate clusters. Figure 4 shows the dataset's dendrogram on the left, and the points which make up each of the five clusters to the right. Cluster A is the root node and consists of the entire dataset, it splits into clusters B and C, and B eventually splits into D and E. The dendrogram gives us a way to easily navigate and reason about a hierarchy of potential clusters within a dataset.

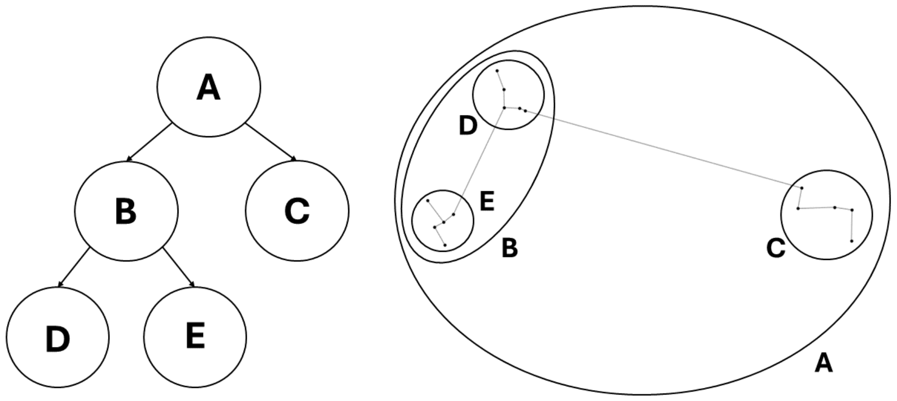

Figure 4 Dendrogram (left), and the original dataset (right).

Building the dendrogram is relatively straightforward and follows the implementation provided by Wang. First, the edges of the MST are sorted by weight. Then, we iterate through each edge and for each we determine the two components that edge connects using a union-find data structure.[12] We record the newly formed cluster, the two

[12] A union-find data structure stores a collection of disjoint sets. For our purposes, we use an implementation that allows us to (1) efficiently determine the set a point belongs to, and (2) efficiently merge two sets. This data structure is used throughout our algorithm and allows easy tracking of components during the MST and dendrogram build processes.

children that edge merged, the size of the new cluster, and the edge weight. With the dendrogram, we can begin to make cluster determinations.

We first condense the dendrogram based on a *minimum cluster* hyperparameter that defines the minimum number of points needed for a cluster to be reportable. The remaining nodes represent the dataset's candidate clusters. We use the concept of stability defined by HDBSCAN and the excess of mass algorithm to cull these clusters. Every node in the dendrogram has a stability, and we determine whether a cluster is valid by iterating through the dendrogram from the bottom up and comparing each node's stability to the combined stability of its children. If the parent's stability is greater than the combined stability of its children, we keep the parent and discard every node below the parent. If the children's combined stability is greater than the parent's, we set the parent's stability to the combined stability of the children and discard the parent cluster. We can define λ as $\lambda = \frac{1}{split\ distance}$, where the split distance corresponds to an edge weight in the MST. Every cluster in the dendrogram will have a λ_{death} corresponding to the λ at which the cluster's children merge and a λ_{birth} corresponding to the λ at which the cluster merges with its sibling into their parent cluster.[13] Additionally, each point within the cluster will have a λ value, $\lambda_{birth} \leq \lambda \leq \lambda_{death}$, that corresponds to the edge that joins the point to the cluster. The cluster's stability can then be defined as $\sum_{p \in cluster}(\lambda_p - \lambda_{birth})$. To understand what this culling approach is doing, and why it's called "excess of mass" it is useful to visualize the dendrogram with the graph below in Figure 5.

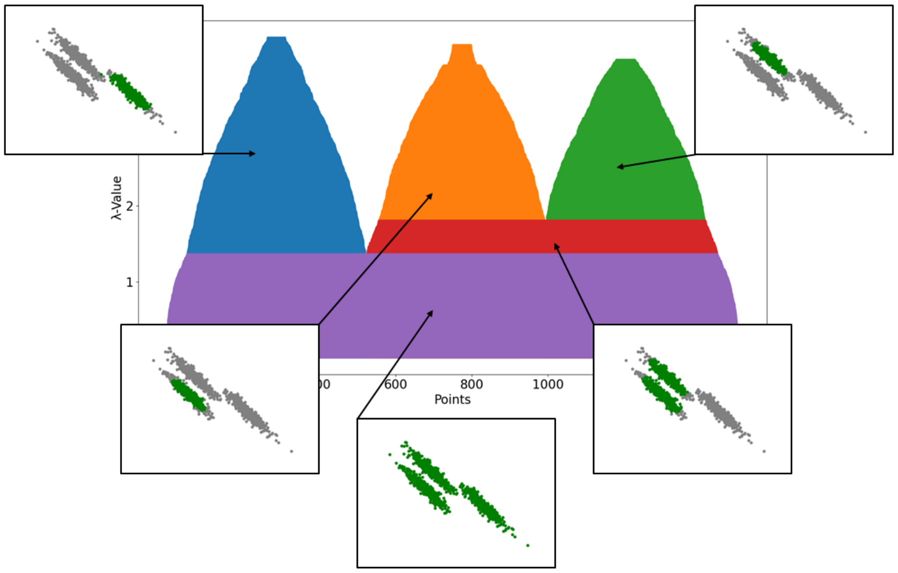

Figure 5 Cluster size as a function of λ.

This plot shows the relationship between λ and the resulting clusters for that λ-value for a simple, three-cluster dataset. Each unit on the x-axis corresponds to a point within the data set and the y-axis corresponds λ, each color in the plot represents a different cluster in the dendrogram hierarchy. The number of points in a cluster at a given λ is given by the cluster's width at that y-value. With this plot we can easily see the effect raising and lowering λ thresholds have on the number and size of the resulting clusters. The stability of each cluster is the area ("mass") of their colored region. When we make our stability comparisons to cull candidate clusters, we are comparing the area each cluster occupies on this graph.

The culling process described up to this point matches the process employed by HDBSCAN, but we identified three shortcomings of that approach. The first is that the root node of the dendrogram tends to have a much greater stability than the rest of the cluster tree. HDBSCAN's response to this is a toggle that allows the user to automatically discount

[13] The root node does not have a "birth" in the same way that the other nodes in the dendrogram have—its birth is $\lambda = 0$. Similarly, the leaf nodes do not have a "death" in the same way that other nodes do, their death occurs when they become smaller than the *minimum cluster* hyperparameter.

the root node. The effect of this toggle is that if we discount the root node the algorithm finds clusters and structure in datasets which have none, and if we don't discount the root node the algorithm tends toward defining the entire dataset as a single cluster. The problem with this approach is that it either requires some *a priori* knowledge that clusters exist in the dataset, or it requires some kind of feedback, human or automated, to determine whether the toggle should be on or off on a per-dataset basis. We are targeting real-time applications where human intervention is not an option. Additionally, if the number of clusters in a data stream is constant, and they have all been identified by our system, our unsupervised clustering algorithm will only be looking at noise and it needs to be able to consistently identify unstructured datasets as containing only noise. Our response to this problem is to limit the root node by defining a floor for the root node and only including the area above that floor in its stability calculation. We currently implement a floor at 60% of the root node's λ_{death}. This value has been an effective level that allows for unstructured data to be identified as such while still allowing cluster definitions when they should occur.

The second shortcoming of HDBSCAN's approach to labeling is that clusters are defined just before their birth, or right before they merge into their parent. The problem with this approach becomes evident in noisy datasets. Clusters will pick up noise points in the interstitial space between each other before merging. If we define our cluster definitions right before they merge, we end up with very loose definitions that encompass much more than is appropriate. Our solution to this can be best visualized with a similar λ vs. cluster size plot shown in Figure 5. This time we focus on a single cluster, and instead of mirroring the cluster growth we simply look at the relationship between λ and the size of the cluster. We can then determine what the plot would look like if it had the same stability and initial and final sizes but grew linearly with respect to λ. The λ value of the right-most intersection between the linear plot and the actual plot gives us a threshold to cut the cluster growth. The logic behind this approach is that a cluster's growth will be regular while expanding within itself, plateau once the cluster is fully encompassed, and then experience sudden growth after expanding into the surrounding noise. By reducing our λ threshold in this way we can step over the concavity associated with a loosely defined cluster. Figure 6 shows what this looks like for a noisy dataset containing two distinct clusters. The dotted line shows the calculated linear growth line, and the insets show the concavity that appears when a cluster begins to expand into surrounding noise. Each scatter plot shows the points to the left of the intersection between the linear growth line and the actual growth curve in green and the points to the right of that intersection in red.

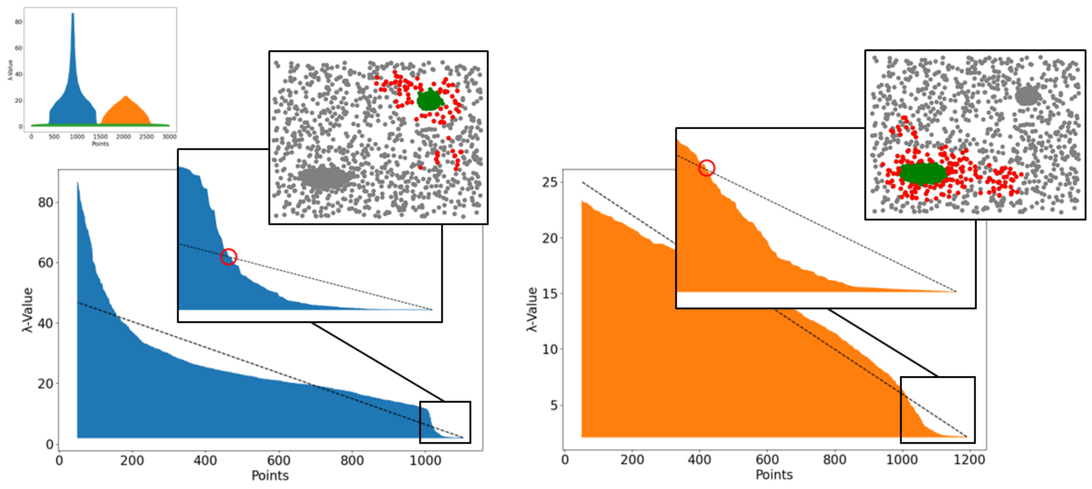

Figure 6 Cluster reduction strategy.

The third issue we have encountered is that when making a cluster determination, HDBSCAN will either keep the node or both of its children. It does not have a mechanism for only keeping one of the node's children. In a well-formed dendrogram, this is not usually a problem, but having a well-formed dendrogram requires hyperparameters that fit the particular dataset being analyzed. Our goal is to have an algorithm whose default hyperparameters have a high level of stability. When we have the luxury of parameter tuning, we can optimize for a particular data stream but when we do not, we are still performant. As such, we need a way to successfully deal with ill formed dendrograms. An exaggerated example can be seen below in Figure 7.

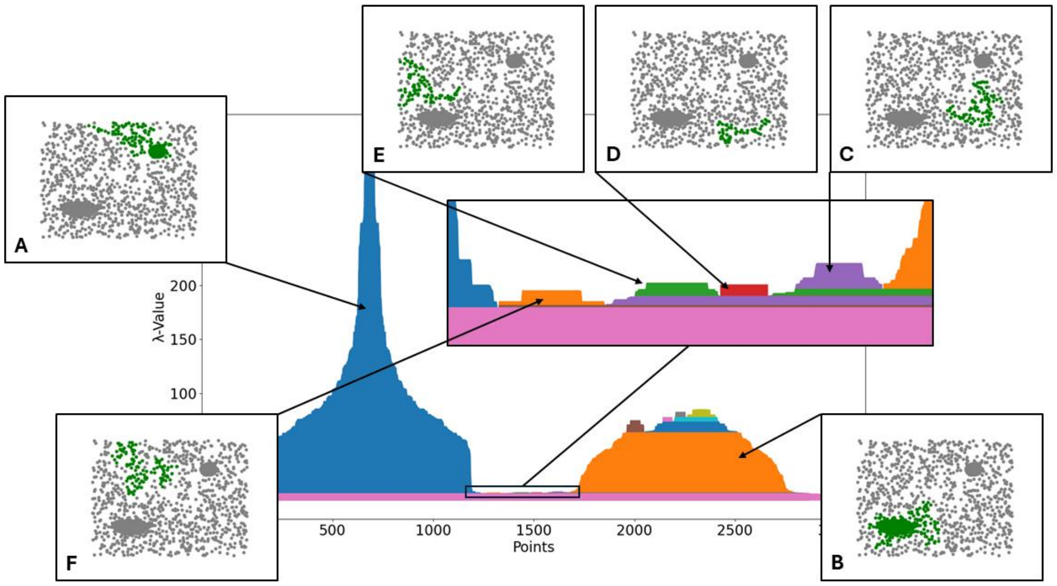

Figure 7 Ill-formed dendrogram.

Cluster A and B are the clusters we care about, but HDBSCAN's culling strategy would result in cluster's C, D, E, and F being counted as valid. Our approach to this problem is to allow a third option when considering a node in the dendrogram: we can keep the parent, both children, or only a single child. We make the decision regarding the parent in the same way described above. If the parent's stability is greater than the combined stability of the children, the parent is kept. If not, we determine whether the children represent two distinct clusters or a single distinct cluster with a hanger-on. We do this by first calculating the average stability contribution of a single point within each cluster ($stability/size$), we then take the ratio of the smaller cluster's average stability contribution to that of the larger cluster, $\frac{(stability/size)_{smaller}}{(stability/size)_{larger}}$. If that ratio is greater than a preset threshold we keep both clusters, if not we only keep the larger cluster. We look at the average stability contribution to allow small, but distinct, clusters in close proximity to much larger clusters to be identified and we look at a ratio of the two values so that our threshold is able to generalize to a wide range of datasets. In our tests, we have found that there is a wide band between the ratios of two legitimate clusters and the ratios of a single legitimate cluster and a hanger-on and a simple threshold value has been sufficient to make the determination.

With this approach to unsupervised clustering, we are able to handle structured and unstructured datasets, maintain tight cluster definitions, and reduce the number of extraneous cluster definitions generated all while executing very rapidly and leveraging parallel processes.

PERFORMANCE EVALUATION

The natural point of comparison to our system is HDBSCAN. The premise of our unsupervised clustering algorithm mirrors theirs and HDBSCAN is, in many ways, the current state of the art. As such, this paper will present benchmark comparisons between our algorithm and HDBSCAN. Comparisons between our algorithm and other unsupervised clustering algorithms can be seen transitively through the benchmark comparisons presented in HDBSCAN's documentation (McInnes, Healy, and Astels, 2016). As we have developed our own approach, we are mindful to make sure that we provide a tangible benefit over HDBSCAN. We must have an edge over their algorithm in key places, while generally performing at parity with HDBSCAN. Our two focuses have been noise rejection and speed. The effect of the former can be seen below in Figure 8.

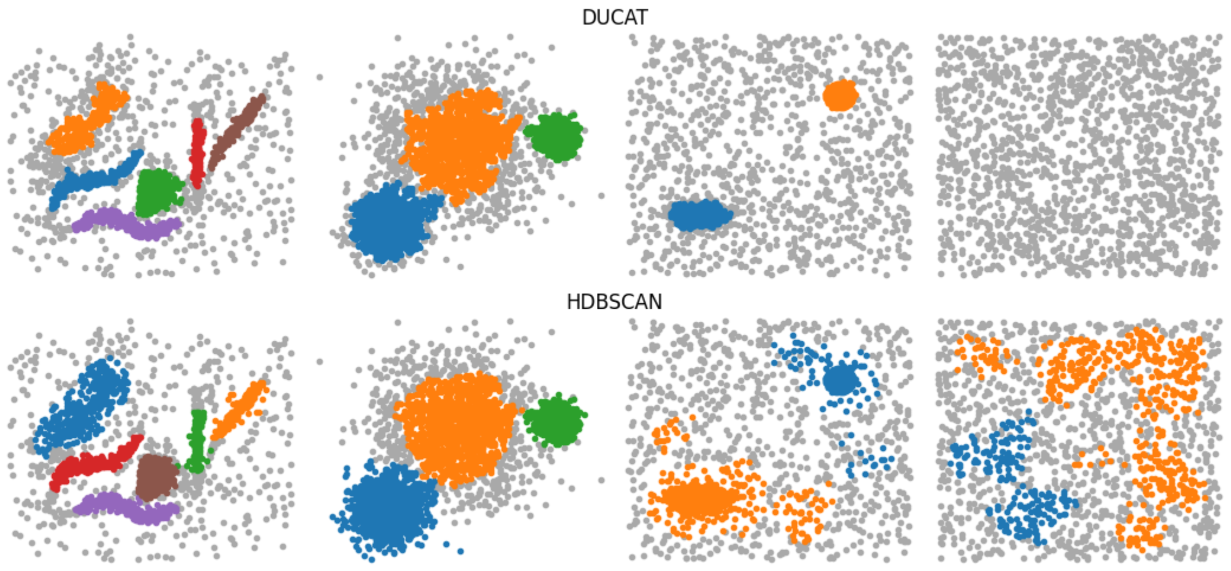

Figure 8 Labeling comparison.

The top row shows the labels that our algorithm generated and the bottom row the labels generated by HDBSCAN. Points labeled as noise are colored gray. Both algorithms could produce better results for each individual dataset if they were tuned specifically for that dataset, but a static hyperparameter set was selected for each algorithm based on the parameters that gave the best, overall results.[14] Note that both algorithms can find arbitrarily shaped clusters, clusters of varying densities, and clusters with significant overlap. But our algorithm is able to do all of that *and* maintain tight cluster definitions in the face of significant background noise *and* effectively determine when no clusters exist. All with a single set of hyperparameters.

On the surface, a timing comparison between our fully compiled algorithm and HDBSCAN's Python library seems unfair but (1) their implementation is largely written in Cython, which can produce performance comparable to C, and (2) the Python library is the official implementation. For our timing benchmarks we used the same unstructured and structured datasets used for benchmarking the performance difference between our implementation and Wang's implementation. Across all 100 dimension-point pairs we tested, our implementation was on average 4.1 times faster than HDBSCAN, we are at least 1.4 times faster and at most 8.1 times faster. For the structured data sets, we were on average 4.9 times faster, and for the unstructured data sets, we were on average 3.3 times faster. Plots of the time to execute for both the structured and unstructured data sets can be seen with respect to the data set length in Figure 9 and with respect to dimensionality in Figure 10.

[14] For DUCAT: *minimum cluster* = 60, *minimum points* = 10, for HDBSCAN: *minimum cluster* = 80, *minimum points* = 35

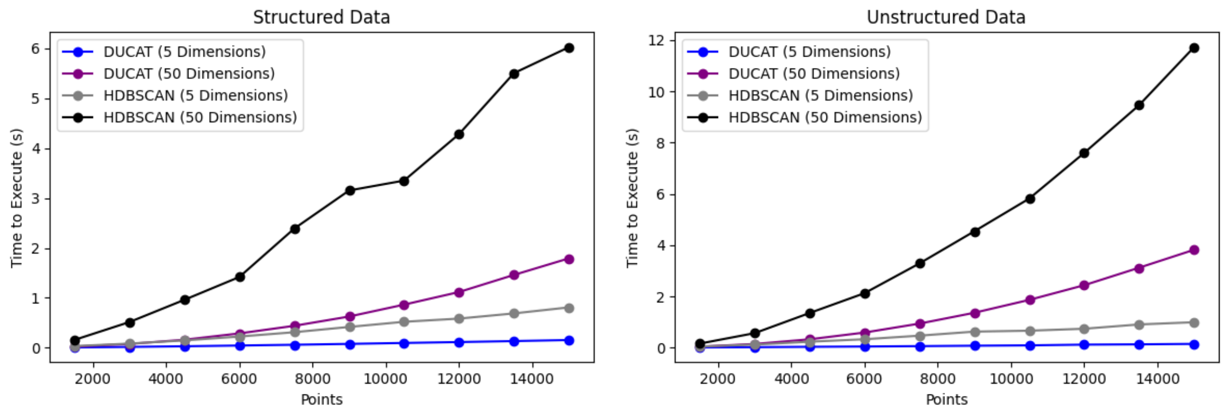

Figure 9 Time to execute comparisons across data set lengths with a fixed dimension.

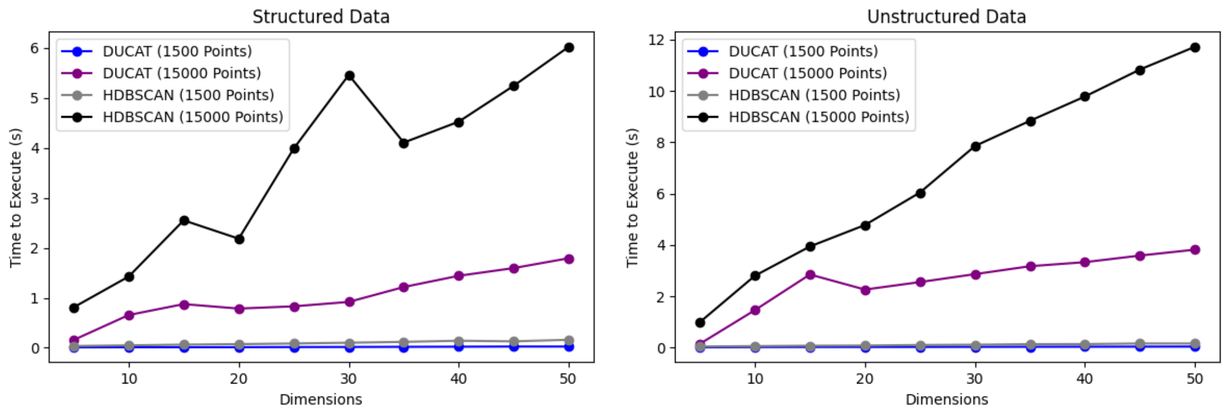

Figure 10 Time to execute comparisons across dimensions with a fixed data set length.

FUTURE WORK

Since its introduction, our system has undergone enhancements to better handle noisy and high-dimensional data. We have a much-improved unsupervised clustering algorithm that not only is highly capable as a stand-alone algorithm but also slots into the system architecture we presented last year. Our algorithm's ability to handle totally unstructured data as well as noisy structured data all with a single set of hyperparameters offer distinct benefits for clustering in dynamic and noisy environments.

Our system is actively being developed, and future work will focus on refining algorithmic efficiency and exploring new applications for our clustering system. While this paper has focused on the unsupervised algorithm, we still have work to do in the rest of the system. We would like to develop our cluster definition process and add sophistication to the way we codify found clusters. Additionally, we would like to develop concrete implementations for the second-pass module. These would be general approaches that could be leveraged for any data stream as well as approaches specific to a particular domain or data type. With respect to the unsupervised clustering algorithm, we believe there is still room for improvement in our wall-clock performance, particularly with respect to unstructured data. Ongoing efforts will focus on optimizing performance and refining clustering techniques. We are also continuing to work on how we address the labeling problems we laid out in this paper. The solutions we presented have given us very good results, but there is always room for improvement. We would ideally like to move away from thresholding techniques towards metrics derived from the data being analyzed, and we are actively exploring ways that we can do that.

REFERENCES

Blelloch, G.E., Anderson, D., Dhulipala, L. (2020). *ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines*. 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20). Association for Computing Machinery, New York, NY, USA, 507–509. https://doi.org/10.1145/3350755.3400254

Campello, R.J.G.B., Moulavi, D., Sander, J. (2013). *Density-Based Clustering Based on Hierarchical Density Estimates*. In: Pei, J., Tseng, V.S., Cao, L., Motoda, H., Xu, G. (eds) Advances in Knowledge Discovery and Data Mining. PAKDD 2013. Lecture Notes in Computer Science, vol 7819. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-37456-2_14

Heinlen, C., Volpi, M., Allen, R. (2023). *A Novel Approach to Dynamic Unsupervised Clustering*. 2023 Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC 2023). https://www.xcdsystem.com/iitsec/proceedings/index.cfm?Year=2023&CID=1001&AbID=120927

McInnes, L., Healy, J., & Astels, S. (2016). *HDBSCAN Documentation*. The HDBSCAN Clustering Library. https://hdbscan.readthedocs.io/en/latest/index.html

Wang, Y., Yu, S., Gu, Y., & Shun, J. (2021). *Fast Parallel Algorithms for Euclidean Minimum Spanning Tree and Hierarchical Spatial Clustering*. 2021 International Conference on Management of Data (SIGMOD '21). Association for Computing Machinery, New York, NY, USA, 1982–1995. https://doi.org/10.1145/3448016.3457296