Building Environmentally-aware Classifiers on Streaming Data

A dissertation presented to the faculty of the graduate school at the

University of Missouri — Columbia

In partial fulfillment of the requirements for the degree

Doctor of Philosophy

Jeffrey Dale

Dr. James Keller, Dissertation Supervisor

July 2022

The undersigned, appointed by the dean of the graduate school, have examined the dissertation proposal entitled

BUILDING ENVIRONMENTALLY-AWARE CLASSIFIERS ON STREAMING DATA

presented by Jeffrey Dale,

a candidate for the degree of doctor of philosophy in Computer Science,

and hereby certify that, in their opinion, it is worthy of acceptance.

James Keller

Dr. James Keller

Derek Anderson

Dr. Derek Anderson

[illegible]

Dr. Giovanna Guidoboni

[illegible]

Dr. Mihail Popescu

[illegible]

Dr. Grant Scott

Acknowledgements

This work was partially funded by the Office of Naval Research grant number N00014-161-2323 to support the U.S. Naval Surface Warfare Center.

Table of Contents

List of Figures

Appendix

List of Algorithms

Abstract

The three biggest challenges currently faced in machine learning, in our estimation, are the staggering quantity of data we wish to analyze, the incredibly small proportion of these data that are labeled, and the apparent lack of interest in creating algorithms that continually learn during inference. An unsupervised streaming approach addresses all three of these challenges, storing only a finite amount of information to model an unbounded dataset and adapting to new structures as they arise.

Specifically, we are motivated by automated target recognition (ATR) in synthetic aperture sonar (SAS) imagery, the problem of finding explosive hazards on the seafloor. It has been shown that the performance of ATR can be improved by, instead of using a single classifier for the entire ATR task, creating several specialized classifers and fusing their predictions [44]. The prevailing opinion seems be that one should have different classifiers for varying complexity of seafloor [74], but we hypothesize that fusing classifiers based on sea bottom type will yield higher accuracy and better lend itself to making explainable classification decisions. The first step of building such a system is developing a robust framework for online texture classification, the topic of this research.

In this work, we improve upon StreamSoNG [85], an existing algorithm for streaming data analysis (SDA) that models each structure in the data with a neural gas [69] and detects new structures by clustering an outlier list with the possibilistic 1-means [62] (P1M) algorithm. We call the modified algorithm StreamSoNGv2, denoting that it is the second version, or verse, if you will, of StreamSoNG. Notable improvements include detection of arbitrarily-shaped clusters by using DBSCAN [37] instead of P1M, using growing neural gas [43] to model each structure with an adaptive number of prototypes, and an automated approach to estimate the η parameters.

Furthermore, we propose a novel algorithm called single-pass possibilistic clustering (SPC) for solving the same task. SPC maintains a fixed number of structures to model the data stream. These structures can be updated and merged based only on their "footprints", that is, summary statistics that contain all of the information from the stream needed by the algorithm without directly maintaining the entire stream. SPC is built on a damped window framework, allowing the user to balance the weight between old and new points in the stream with a decay factor parameter.

We evaluate the two algorithms under consideration against four state of the art SDA algorithms from the literature on several synthetic datasets and two texture datasets: one real (KTH-TIPS2b [68]) and

one simulated. The simulated dataset, a significant research effort in itself, is of our own construction in Unreal Engine and contains on the order of 6,000 images at 720×720 resolution from six different texture types.

Our hope is that the methodology developed here will be effective texture classifiers for use not only in underwater scene understanding, but also in improving performance of ATR algorithms by providing a context in which the potential target is embedded.

I. Problem Statement

Most modern machine learning approaches, both supervised (classification) and unsupervised (clustering), make the assumption that the test set contains only samples from classes present in the training set. For example, when training a classifier to distinguish cats from dogs, the assumption is often made that all samples fed to this classifier will be of either a cat or a dog. As former Secretary of Defense Donald Rumsfeld put it, this is a problem of known unknowns. We know that there are cats and dogs that we haven't seen before, and can develop algorithms accordingly.

Although closed-world models like the example described here are sufficient for certain tasks, many domains cannot rely on the assumption that all unknowns are known unknowns. Unless the union of all classes known to a classifier form the universe set under consideration, there are always unknown unknowns to handle. Continuing with the example of the cat vs. dog classifier, one would expect indeterminate behavior to occur if the classifier is given a picture of a human. Perhaps the classifier would assign the human some confidence in the cat and dog classes, especially if a sum-to-one constraint such as a softmax is imposed on the classifier output. More sophisticated classifiers might assign little to no confidence in both classes, indicating that the input is neither a cat nor a dog. While such behavior is more desirable than incorrectly forcing every sample into the cat or dog class, it still does not provide a mechanism for recognizing future samples from the human class.

A streaming framework is required to sufficiently model classification problems in the presence of unknown unknowns in order for learning to occur in the field. This entails developing a solution to the following problems:

1. *Offline initialization* – given a finite set of training samples, which may or may not be labeled, discover and model structure among the samples.
2. *Online classification* – given an unlabeled sample, produce a meaningful classification label vector with one element for each possible class. Entries of this vector can be binary, indicating either full or no membership of the sample in a class (*i.e.*, crisp classifiers), or real numbers in $[0, 1]$ to indicate partial belonging. For fuzzy and probabilistic classifiers, the entries of this vector sum to unity, whereas possibilistic classifiers relax this constraint.
3. *Outlier detection* – determine whether an unlabeled sample should be placed in an existing class or marked as anomalous. Anomalous points do not conform to our existing understanding of structures in the data, and may be either outliers or early samples from a

developing class — a distinction that algorithms operating in the streaming domain must be able to make.

4. *Online learning* – given an unlabeled sample that is deemed an inlier, update the classification model to better recognize samples like it in the future. This is typically accomplished by updating the model of one or more existing classes or by creating a new class.

Each of these problems are thoroughly discussed in the subsections that follow. Furthermore, the framework is subject to the constraints of:

1. *Finite storage* – when run for an infinite amount of time on an unbounded input stream, the model can be contained in a finite amount of space.
2. *Arbitrarily-shaped structures* – can model complex shapes in the input stream, as opposed to, for example, solely Gaussian clusters or hyperplane decision boundaries.

We study two models for streaming data analysis in this research: StreamSoNGv2, and improvement to the streaming soft neural gas (StreamSoNG) algorithm [85], and single-pass possibilistic clustering (SPC), a novel footprint-based possibilistic clustering algorithm. These models are applied in the domain of streaming texture classification.

Lastly, we would like to bring to attention the question by Bezdek and Keller on whether the nomenclature "streaming clustering" is a correct description of the task being performed by most of the algorithms that proclaim themselves as such [24]. Throughout this work, we refer to these algorithms, including those of our own design, by the more general category of "streaming data analysis" algorithms, or SDA algorithms for short.

A. Offline Initialization

The first step in building an algorithm for stream analysis, be it supervised or unsupervised, is to determine whether the algorithm will be initialized using a set of static data, that is, data can be retained and iterated over. It is exceedingly rare for machine learning algorithms to function properly on very small sample sizes, as would be the case if an algorithm were to immediately try and learn from scratch on a data stream with no initialization. Most commonly, an initialization set is provided *a priori*, though the algorithm could also have a so-called *burn-in period* where data from the stream is accumulated for a predetermined period of time to form an initialization set.

One must also decide whether the data used for initialization must be labeled. This decision is more nuanced than simply using labels if they are available. In our experience, training supervised classifiers on subjectively labeled textures has led to worse performance than using unsupervised methods. While research exists for learning with noisy labels [56, 67], achieving high accuracy with unsupervised methods allows the research to be applied to a wider swath of use cases, and hence is preferred in our research.

During offline initialization, we are allowed to maintain the data long enough to iterate over it as much as needed, subject to domain and time constraints. Thus, performing the offline initialization can be done using well-established and well-studied algorithms like k-means or fuzzy C-means [23] in the unsupervised case, or, in the supervised case, support vector machines or possibilistic K-nearest neighbors [42]. The main requirement is that the knowledge gained during initialization allows the algorithm to make intelligent decisions where it counts: as samples stream in during inference.

B. Online Classification

After initialization, the model should be prepared to accept and classify samples from the data stream. Upon receiving a new sample from the stream, the model must provide a label for the sample, and preferably a confidence associated with the classification. This label could potentially be "outlier", indicating that the sample doesn't fit any known class. Outlier detection is further discussed in the following section.

For soft classifiers, the principle of least commitment [59] is adhered to and a vector of confidences in each known class is provided in lieu of a single classification decision. The constraints applied to the vector of confidences depend on the particular type of soft classifier. Probabilistic and fuzzy classifiers require confidences in the range $[0, 1]$ with the sum of confidences over all classes to be unity, and possibilistic classifiers require only the former constraint. Our focus is on possibilistic classifiers for reasons that are made clear in Section IV-C.1 of this document.

C. Outlier Detection

We argue that it is crucial not only for the model to produce accurate labels for the stream, but also to have a robust mechanism of outlier detection. This work's application domain, streaming texture classification, all but guarantees the presence of outliers and the arrival of new classes in the stream.

Outlier detection is a very well-studied field, however the problem we encounter in streaming data analysis is that both the arrival of an outlier and the beginning of a new class in the stream appear identical until more data arrives. It is rarely possible to know if an incoming point is truly an outlier that can be discarded until much further in the stream. Thus, the ability to form new classes in an online setting necessitates finding a way to save *all* outliers as they arrive in some way, which is inherently at odds with the streaming philosophy of maintaining a constant amount of information to describe an unbounded data stream.

Several of the existing algorithms for streaming data analysis, such as CluStream [15], DenStream [32], and DBSTREAM [50], treat outliers as “micro-clusters” containing a single point, and then have a separate procedure for removing micro-clusters that do not contribute to the clustering. Others, like StreamSoNG [85], maintain a list of outliers and periodically search for clusters within that list.

The key challenge here is in deciding when an outlier can be safely discarded, that is, when we believe the likelihood of encountering more points like it from the stream is small enough. By a large margin, the most common approach for solving this problem is some form of temporal windowing. Sliding window approaches simply discard points older than a predefined threshold as outliers, whereas some more complicated approaches like pyramidal windowing attempt to maintain information about outliers in the whole stream, but in coarser and coarser detail the further back in time one looks.

D. Online Learning

A useful SDA algorithm must be able to learn in an online setting. Without online learning, SDA would be reduced to simply running inference with any static classifier. The learning process for SDA boils down to two primary tasks: updating existing class definitions and forming new class definitions.

Updating existing class definitions usually entails updating *cluster footprints*, that is, the set of summary statistics needed to fully define a cluster, to reflect new information from the stream. The update can be done stochastically, such as by moving a cluster centroid some amount in the direction of a new point, or analytically, such as by deriving update equations based on necessary conditions for convergence of the algorithm.

When the outlier detection methodology within the SDA algorithm identifies one or more samples that comprise a new class, the algorithm must be able to compute the same cluster footprint as for existing classes using only the known sample(s) in the new class. In many cases, these cluster footprints consist of some summary statistics of the data. BIRCH [89], for example, stores the summary statistics necessary to compute the variance of each cluster, namely the number, sum, and sum of squares of the data points which comprise it.

Both tasks must be solved when developing an algorithm for SDA, as they are fundamental to the streaming phenomenology.

E. Simulated Datasets

As the video game industry pushes simulation to previously unfathomed levels of photorealism, machine learning researchers have found that models can be trained solely on scenes created in game engines and achieve impressive performance on real data. Moreover, manually modeling a scene gives the researcher full control over the contents of the scene, the placement of objects, the lighting, and much more. Perhaps most alluring is the ability to create unbounded training sets with pixel-perfect ground truth, circumventing the time and money bottleneck of hiring data labelers to segment thousands or millions of images.

These simulated datasets do not come for free, however. Game engines, in our case Unreal Engine, have a steep learning curve and often require skill sets aligned more with graphic design than computer science. A substantial time investment is required before usable scenes are available in simulation, but fortunately, the availability of ultra high resolution scans of textures and objects for download online greatly eases this barrier to entry.

II. Literature Review

In this section, we provide a detailed survey of the relevant literature for the proposed study. The structure of this literature review will generally mimic that of Section I, but with offline initialization, online classification, and online learning merged into a single section on streaming data analysis (SDA) as a whole. This change is made to avoid splitting SDA papers across multiple sections, as they typically propose unique solutions to each of the three problems covered in Section I. We briefly summarize how each of these SDA algorithms address the problems of initialization, online classification, online learning, and outlier detection in Table 1, with more detail to follow in this section.

A. Streaming Data Analysis

Carnein and Trautmann have recently produced a phenomenal survey on 51 SDA algorithms [31]. Thanks to their efforts, we were able to quickly identify seven relevant algorithms for further examination as exemplars of effective techniques in SDA. We chose to describe three classical SDA algorithms (BIRCH [89], CluStream [15], and DenStream [29]), two competitive learning algorithms (G-Stream [45] and evoStream [30]), and two SDAs identified by Carnein and Trautmann

	Offline Initialization	**Online Classification**	**Online Learning**	**Outlier Detection**
BIRCH	N/A	N/A	Insert into CF Tree	Low density
CluStream	K-means	N/A	Update CF of nearest micro-cluster	Distance from nearest micro-cluster
DenStream	DBSCAN	N/A	Merge into nearest micro-cluster	Distance from nearest micro-cluster
D-Stream	N/A	N/A	Update characteristic vector of grid	Delete sporadic grid cells
DBSTREAM	N/A	N/A	Update weight, center, and time of nearest micro-cluster	Number of fixed-radius nearest neighbor micro-clusters
G-Stream	N/A	N/A	Growing neural gas	Distance to best matching unit
evoStream	N/A	N/A	Update weight, center, and time of nearest micro-cluster	Distance to nearest micro-cluster
StreamSoNG	Neural Gas	PKNN	Online extension of NG	Low typicality

Table 1: Brief summary of how each SDA algorithm reviewed here addresses the main challenges of the SDA problem.

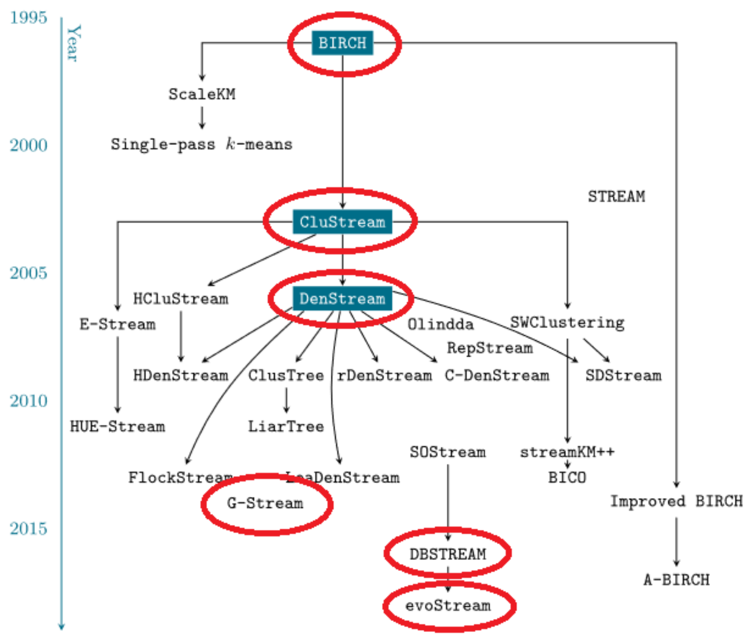


Figure 1: The taxonomy of distance-based SDA algorithms as interpreted by Carnein and Trautmann. This figure is reproduced from Figure 5 of Carnein and Trautmann [31], with algorithms we study here circled in red.

to be state-of-the-art (D-Stream [32] and DBSTREAM [50]).

The taxonomy of the distance-based SDA algorithms we have chosen to evaluate, based on that of Carnein and Trautmann [31], is shown in Figure 1. We have circled the SDA algorithms under consideration here in red, which include all except D-Stream (a density-based, rather than distance-based, algorithm) and StreamSoNG [85] (more recent than the survey). We provide a high level and intuitive overview of each algorithm in this section, though for precise details on any algorithm, we refer the reader to the original manuscript.

Algorithm	**Year**	**Parameters**[*]	**Cluster Shape**
BIRCH [89]	1996	3	Spherical
CluStream [15]	2003	4	Arbitrary
DenStream [29]	2006	8	Arbitrary
D-Stream [32]	2007	5	Arbitrary
DBSTREAM [50]	2016	7	Arbitrary
G-Stream [45]	2016	8	Arbitrary
evoStream [31]	2019	6	Spherical
StreamSoNG [85]	2020	$6 + 5C^{\dagger}$	Arbitrary

[*] Parameters that do not affect execution, *i.e.*, for visualization, are not counted.
[†] C is the number of classes in the data, as each class has a neural gas associated with it.

Table 2: Comparison of SDA algorithms.

1. *Classical SDAs*

The Balanced Iterative Reducing and Clustering using Hierarchies algorithm [89] (BIRCH) was a pioneer in streaming data analysis, among the first in both representing clusters using summary statistics that can be updated incrementally and in performing clustering in a single pass over the dataset. While the intent of BIRCH's single pass requirement was to improve upon the efficiency of existing clustering algorithms, it was later found to be valuable for ephemeral datasets such as streams.

BIRCH fully describes a cluster with a three-tuple containing the number of points in the cluster, the sum of the data points in the cluster, and the sum of the (element-wise) squares of the points in the cluster. This three-tuple, called the Clustering Feature (CF), contains all necessary information required to compute the centroid, radius, and diameter of a cluster and contains only an integer and two vectors of same dimensionality as the dataset. As points from the dataset arrive, a tree of CF vectors is produced, leading to a hierarchy of clusters. When requested, clusters are extracted from the CF tree using agglomerative hierarchical clustering.

CluStream [15] introduces the concept of a micro-cluster, described as “a temporal extension of the cluster feature vector” of BIRCH. The micro-clusters of CluStream contain the same information as the CF of BIRCH, but also include the sum of timestamps and sum of squared timestamps from each point in the stream. A fixed and finite number of micro-clusters are stored over a pyramidal time window as “snapshots”, providing high granularity over recent evolution of the dataset and low granularity far in the past. In order to keep the number of micro-clusters constant, a maintenance procedure deletes micro-clusters that are deemed outliers based on an estimation of the micro-cluster’s age.

At any time during the execution of CluStream, an offline clustering procedure can be run to obtain the “macro-clusters” present in the stream at the current time. The authors propose clustering the micro-clusters with a slightly modified k-means to produce macro-clusters over a time horizon $[t_1, t_2]$, with t_2 often being the current time.

A drawback of algorithms like BIRCH and CluStream that depend on the CF to summarize clusters is their inability to model non-Gaussian clusters. DenStream [29] builds upon CluStream's concept of the micro-cluster, leveraging a modified version of DBSCAN [37] to model arbitrarily shaped clusters.

Further extending the idea of micro-clusters, DenStream maintains core micro-clusters, potential micro-clusters, and outlier micro-clusters. Core micro-clusters serve the same purpose as the micro-clusters of CluStream and represent a subset of the stream that are sufficiently dense in feature space. Potential micro-clusters describe smaller structures in the stream that are not dense enough to for a core micro-cluster. Outlier micro-clusters are a set of outliers that are not dense enough to become a potential micro-cluster.

The general process of DenStream is as follows. First, a preliminary clustering is done by running DBSCAN on an initialization set. When a new point from the stream arrives, attempt to merge it into an existing potential micro-cluster. If no potential micro-clusters describe the new point well enough, either merge the new point into an outlier micro-cluster by similar logic or create a new outlier micro-cluster containing the new point. Periodically, a maintenance algorithm deletes any potential or outlier micro-clusters that are either too old or not sufficiently dense. When a clustering is requested, a variant of DBSCAN is performed on the core and potential micro-clusters.

2. *Related SDAs*

G-Stream [45] is an SDA that takes advantage of the growing neural gas (GNG) algorithm [43] to represent arbitrary topologies in the dataset. The key contribution of G-Stream in making GNG viable in a single pass over the dataset is the introduction of a "reservoir" to hold outliers. When a new sample is read from the stream, if it is further than a prescribed threshold from its nearest GNG neuron, it is added to the reservoir. After enough points have been added to the reservoir, all points in it are removed and placed at the front of the stream as if they are just arriving. The reservoir allows the learning component of GNG to take another look at what were previously thought to be outliers, but may now form a structure in the context of more data. Further, outdated and isolated nodes are periodically removed from the GNG.

Combined with GNG's mechanism for automatically detecting when new nodes are needed to model the topology of the dataset, G-Stream is one of the few SDAs that can truly represent any topology. A key difference between StreamSoNG and G-Stream is that, in G-Stream, each node of the neural gas represents a cluster, whereas, in StreamSoNG, each cluster is modeled by its own neural gas prototypes.

Evolutionary algorithms also lie in the domain of competitive learning along with neural gas, and so, we also consider evoStream [30] for comparison. The evoStream algorithm exhibits fairly standard behavior when a new point arrives from the stream. If the new point is close enough to an existing micro-cluster, that micro-cluster absorbs the point. Otherwise, a new micro-cluster is created to hold the point. Occasionally, a cleanup procedure is run that removes old micro-clusters and merges micro-clusters that are too close together.

The hallmark of evoStream is its evolutionary step, which form macro-clusters from the micro-clusters and is run while the stream is idle. The authors indicate that this idle time typically occurs in real-time applications where there is down time between the arrival of sequential points from the stream.

During the evolution step, roulette wheel selection, binary crossover, and mutation are applied to the population as is standard for evolutionary algorithms, where mutation is accomplished by perturbing the coordinates of selected individuals by a small random value. The population size P is maintained in a similar way to a $(P + 2)$–Evolution Strategies [26, 73]. After the two newly generated offspring are added to the population, the two least-fit members of the now $(P + 2)$-sized population (potentially the newly generated offspring) are discarded.

The fitness function f used in evoStream is the inverse of the sum of squared distances from each micro-cluster to its nearest macro-cluster, given by[1]

$$SSQ = \sum_{i=1}^{k} \sum_{mc \in C_i} d(mc[c], C_i)^2 \tag{1}$$

$$f = \frac{1}{SSQ} \tag{2}$$

where k is the number of macro-clusters, d is a distance function (Euclidean in this case), $mc[c]$ denotes the centroid of micro-cluster mc, and C_i denotes macro-cluster i.

3. *State-of-the-Art*

D-Stream [32] is a grid-based SDA that is often reported to perform well [31, 50]. The general idea of D-Stream is to maintain a sparse grid over the input space, storing incoming points in their appropriate grid cell and updating a summary statistic containing, among other clerical items, a density measure of the cell. After a certain number of points have arrived from the stream, an initial clustering is run. Furthermore, "sporadic" grid cells, those deemed to contain only outliers, are periodically removed from the grid list and existing clusters are updated to reflect the change.

The initial clustering of D-Stream is agglomerative in that it starts with every dense grid cell in its own cluster and iteratively merges neighboring clusters. The periodic cluster adjustment procedure looks

[1]This equation is slightly modified in notation from Equation 2 in Carnein and Trautmann [30].

at all grid cells and relabels them as sparse or dense, as appropriate, in addition to handling tasks like removing certain sporadic grid cells and splitting clusters of grid cells that have become disconnected.

As is typical with algorithms that attempt to maintain a grid over the input space, D-Stream can be quickly rendered intractable when given high dimensional data [18]. Maintaining a dense grid follows a power law with the number of dimensions and, even when the grid is sparse (as in D-Stream), the number of neighbors of a grid cell is exponential in dimensionality. D-Stream addresses this problem by aggressively removing sporadic grid cells to save memory.

Perhaps the most performant SDA algorithm as of this writing, DBSTREAM [50] adapts concepts from DBSCAN [37] into a streaming setting, similar to DenStream [29]. DBSTREAM maintains a shared density graph over the pairs of micro-clusters in order to later produce macro-clusters. The shared density concept is illustrated in Figure 2 of the original manuscript [50], which is also reproduced here as Figure 2 for convenience. The general idea of shared density between two micro-clusters is novel to DBSTREAM and models the idea that two micro-clusters should be in the same macro-cluster if they not only have a large overlap region, but also have many points assigned to that overlap region. Producing macro-clusters is done offline and on demand in a process involving finding connected components of a graph derived

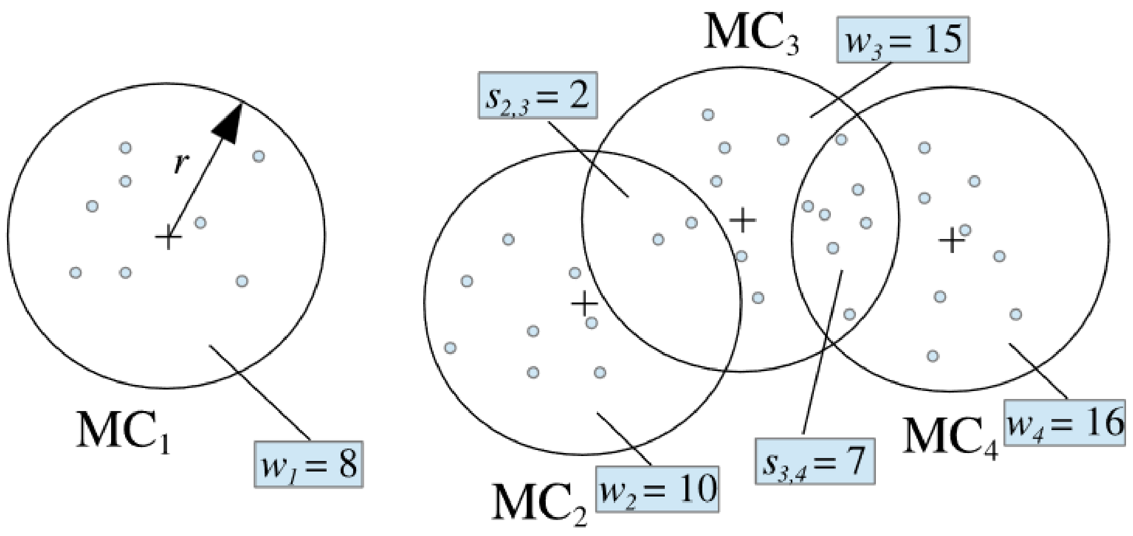


Figure 2: In DBSTREAM, each micro-cluster MC_i has a weight w_i, counting the number of points assigned to it, in addition to a shared weight to each other micro-cluster MC_j denoted $s_{i,j}$ that counts the number of points assigned to the overlap region of the two micro-clusters. This figure is reproduced from Figure 2 in Hahsler and Bolaños [50].

from the shared density graph.

When a new point arrives from the stream, DBSTREAM searches for all neighboring micro-clusters of the point. If none exist, a new micro-cluster is created around that point. Otherwise, *all* micro-clusters in the neighborhood of the new point have their center, weight, recency, and shared density updated. To prevent the problem of collapsing clusters, if two micro-clusters have drifted too close to one another, they are reverted to their previous positions.

B. Outlier Detection

Common to all SDA algorithms discussed in the previous section is the need to detect whether incoming points belong to existing structures. While not explicitly stated in some cases, this is a problem of outlier detection, which has a rich and vast literature in itself. Wang et al. [83] have recently published a comprehensive survey on the subject, dividing methods of outlier detection into six categories: distance-based, density-based, clustering-based, graph-based, ensemble-based, and learning-based. We will summarize some of the more relevant information from their survey to SDA, but direct the reader to the survey if more breadth of information is desired, or the manuscript that proposed the approach if more depth on a particular method is desired.

Distance-based outlier detection seems to be the most prevalent in SDA. Numerous algorithms in SDA have logic along the lines of "if a new point is *close enough* to a micro-cluster, include it in the micro-cluster," where the *close enough* concept means that the distance is smaller than some threshold. Notably, DBSTREAM [50] uses distance-based outlier detection (the ϵ-neighborhood of a point) to determine whether a new sample is absorbed into an existing micro-cluster or forms a new micro-cluster.

Specifically designed for the streaming domain, Angiulli and Fassetti propose a suite (one exact and two approximate algorithms) of distance-based outlier detectors called Stream Outlier Miners (STORM) that operate on windows of the dataset [21]. They create a specialized data structure called an indexed stream buffer that returns all objects in the window with distance less than or equal to a radius parameter. Yang et al. [87] also propose several window-based stream outlier detection algorithms called Abstract-C, Extra-N, and Exact-N. In short, Abstract-C and Extra-N have CPU time roughly linear in window size and Exact-N has linear memory usage in window size. Unfortunately, all methods discussed here require storing all of the data points in the window.

Density-based outlier detection is also common in SDA. Algorithms like D-Stream [32] define outliers as regions in feature space in which few to no previously seen samples reside. A number of density-based heuristics have been developed for outlier detection and outlined by Wang et al. [83]. The most popular seems to be local outlier factor (LOF) [25], a method that identifies outliers as points with relatively low reachability to their k-nearest neighbors, though LOF is not suitable for SDA.

Clustering-based outlier detection tends to take a contrarian approach, where small, sparse, or non-homogeneous clusters are considered outliers. One domain where such an approach has shown promise is in gene expression data analysis [34], in which the authors identified genes related to the progression of a disease in grape plants based on their inability to cluster. While in this particular example, node-based resilience clustering [70] is used, any clustering algorithm capable of modeling the cluster structure of the dataset is suitable for use in clustering-based outlier detection. In SDA, one gets cluster-based outlier detection almost for free having developed an online clustering algorithm.

III. StreamSoNGv2

Abstract — When dealing with unbounded streaming data, such as network packets or frames from a continuous live video feed, it is not feasible to retain all of the data, let alone to apply iterative algorithms over the full dataset. The streaming soft neural gas (StreamSoNG) algorithm proposed by Wu *et al.* is particularly appealing given its ability to model arbitrary topologies in the data, however there some shortcomings that make it difficult to apply in a practical setting. In this work, we identify these shortcomings, offer solutions to them, and compare the improved algorithm to several similar algorithms in streaming data analysis. Particularly, we offer, among other major enhancements, an automated, data-driven approach to determining the value of a parameter η to which the original algorithm is especially sensitive. We demonstrate that StreamSoNGv2 is competitive with related algorithms, shows improvement over the original, and that it provides useful soft labels for each streaming data point, rather than assigning it full membership in a single class.

A. Introduction

The Streaming Soft Neural Gas (StreamSoNG) algorithm [85] shows great promise in solving problems in the domain of SDA, and, after using this algorithm extensively in our research, we have found and addressed a number of shortcomings in StreamSoNG which has culminated in what we are calling StreamSoNGv2 (StreamSoNG Verse 2). Modifications presented here significantly improve the performance of the algorithm and make it both easier and more reliable to apply to new datasets by addressing the difficulty of parameter selection, updating some of the underlying models, and making the model more robust to overlapping classes. In our experiments, StreamSoNGv2 outperforms its predecessor and either beats or is comparable to several state of the art algorithms when considering crisp class assignment, but with the advantage of assigning soft class labels for confusing objects.

B. Background

1. *Neural Gas (NG)*

Neural gas [69] is a competitive learning algorithm that uses a fixed number of prototypes (also called "neurons" or "weights") to learn the topology of a feature space. The general idea of the algorithm is to repeatedly sample a point from the dataset and move each prototype toward this point by an amount inversely proportional to how close the prototype is to the point relative to the other prototypes. For example, the nearest prototype to a sampled point will move a large fraction of the distance toward the point, whereas the furthest prototype from the sampled point will have little to no movement. In addition, "connections" are dynamically formed between neurons, but they are only an output of the algorithm and play no role in the placement of prototypes. Algorithm 1 shows the full pseudocode for neural gas.

Two major limitations of neural gas for use in streaming data analysis (SDA) are that the number of prototypes is fixed and that it is, by design, an iterative algorithm. In SDA, we only take one look at the streaming data, making us unable to leverage multiple passes over the dataset. Furthermore, we cannot know in advance how many prototypes are required to effectively describe a structure in the data that has not been seen yet.

2. *Growing Neural Gas (GNG)*

Growing neural gas [43] addresses the problem of neural gas having a fixed number of prototypes. This is accomplished by tracking an error measurement for each prototype and periodically creating a new prototype between the highest error prototype and its highest error topological neighbor. Another change from the standard neural gas is

Algorithm 1 Neural Gas [69]

```
procedure NeuralGas($X$, $n$, $\varepsilon$, $\lambda$, $T$)
    // $X = \{x_j \in \mathbb{R}^d, j = 1, 2, \ldots, N\}$, N points in d dimensions
    // $n \in \mathbb{N}$, the number of neural gas prototypes to use
    // $\varepsilon \in \mathbb{R}^+$, the learning rate parameter
    // $\lambda \in \mathbb{R}^+$, the neighborhood parameter
    // $T \in \mathbb{N}$, the maximum age of connections between prototypes

    // We initialize $W$ by sampling $n$ points from the dataset
    Initialize $W$ as $n$ weights in $\mathbb{R}^d$ from the $N$ data samples in $X$

    repeat

        Sample a point $x_j \in X$
        for $i = 1, 2, \ldots, n$ do
            Let $w^{(i)}$ be the $i$-th closest prototype to $x_j$
            Update $w^{(i)}$ according to
                        $w^{(i)} \leftarrow w^{(i)} + \varepsilon \exp\{-k_i/\lambda\}(x_j - w^{(i)})$
        Mark the two closest prototypes to $x_j$, $w^{(1)}$ and $w^{(2)}$, as connected
        Set the age of the connection between $w^{(1)}$ and $w^{(2)}$ to zero
        Increment the age of all connections to $w^{(1)}$ by one
        Disconnect prototypes with connections older than $T$

    // We define convergence to be a fixed number of iterations
    until convergence
```

that connections between neurons are now utilized in moving prototypes. Instead of moving all prototypes toward new input samples, the best matching unit (BMU) is moved a large distance toward the new point and its topological neighbors, that is, prototypes connected to the BMU, are moved a smaller distance toward the new point. Prototypes not connected to the BMU do not move. Algorithm 2 shows the pseudocode for growing neural gas.

Growing neural gas is better suited to SDA than neural gas due primarily to removing the requirement that the number of prototypes

Algorithm 2 Growing Neural Gas [43]

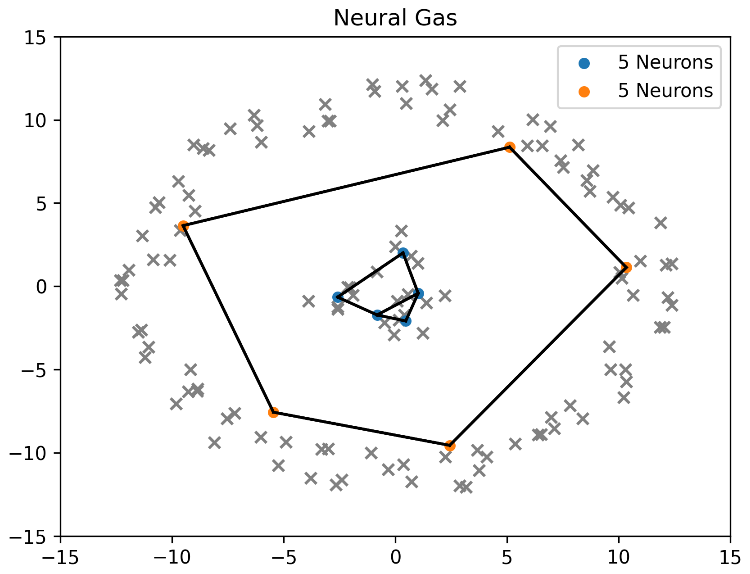
```
// $X = \{x_j \in \mathbb{R}^d, j = 1, 2, \ldots, N\}$, N points with d dimensions
// $\varepsilon_{bmu} \in \mathbb{R}^+$, the learning rate parameter for the best matching unit (BMU)
// $\varepsilon_{nbhd} \in \mathbb{R}^+$, the learning rate parameter for neighbors of the BMU
// $\lambda \in \mathbb{N}$, growth interval
// $\alpha_1 \in (0, 1)$, error scale factor of highest error neurons
// $\alpha_2 \in (0, 1)$, error scale factor of all neurons
// $T \in \mathbb{N}$, the maximum age of connections between prototypes

procedure TrainGNG($X$, $\varepsilon_{bmu}$, $\varepsilon_{nbhd}$, $\lambda$, $\alpha_1$, $\alpha_2$, $T$)
    Initialize $W$ as two weights in $\mathbb{R}^d$
    repeat
        Sample a point $x_j \in X$
        Call UpdateGNG($x_j$, $\varepsilon_{bmu}$, $\varepsilon_{nbhd}$, $\lambda$, $\alpha_1$, $\alpha_2$, $T$)
    until convergence

procedure UpdateGNG($x_j$, $\varepsilon_{bmu}$, $\varepsilon_{nbhd}$, $\lambda$, $\alpha_1$, $\alpha_2$, $T$)
    Let $w^{(1)}$ and $w^{(2)}$ be the closest and next closest prototype to $x_j$
    Increment the age of all connections to $w^{(1)}$ by one
    Increase the error associated with $w^{(1)}$ by $\left\|w^{(1)} - x_j\right\|^2$
    Set $w^{(1)} \leftarrow w^{(1)} + \varepsilon_{bmu}(x_j - w^{(1)})$
    for each $i = 2, 3, \ldots, |W|$ such that $w^{(i)}$ is connected to $w^{(1)}$ do
        Set $w^{(i)} \leftarrow w^{(i)} + \varepsilon_{nbhd}(x_j - w^{(i)})$
    Mark the two closest prototypes to $x_j$, $w^{(1)}$ and $w^{(2)}$, as connected
    Set the age of the connection between $w^{(1)}$ and $w^{(2)}$ to zero
    Disconnect prototypes with connections older than $T$
    Remove all prototypes that have no connections
    if the total number of iterations is $k\lambda$ for some
        $k \in \mathbb{N}$ then
        Call ExpandGNG( )
    Multiply all errors by $\alpha_2$

procedure ExpandGNG( )
    Let $w_p$ be the prototype with largest error
    Let $w_q$ be the prototype connected to $w_p$ with highest error
    Create a new prototype $w^{|W|+1} = 0.5(w_p + w_q)$
    Connect $w^{|W|+1}$ to $w_p$ and $w_q$; disconnect $w_p$ from $w_q$
    Multiply the error associated with $w_p$ and $w_q$ by $\alpha_1$
    Set the error associated with $w^{|W|+1}$ to the error associated with $w_p$
```

be specified *a priori*. It is still an iterative algorithm though, and was published with the intent of looping over the data many times to determine final prototype positions. We have found qualitatively that, by choosing learning rate parameters more aggressively than when running iteratively (*i.e.*, larger ε_{bmu} and ε_{nbhd}), growing neural gas performs well on a single pass over the data.

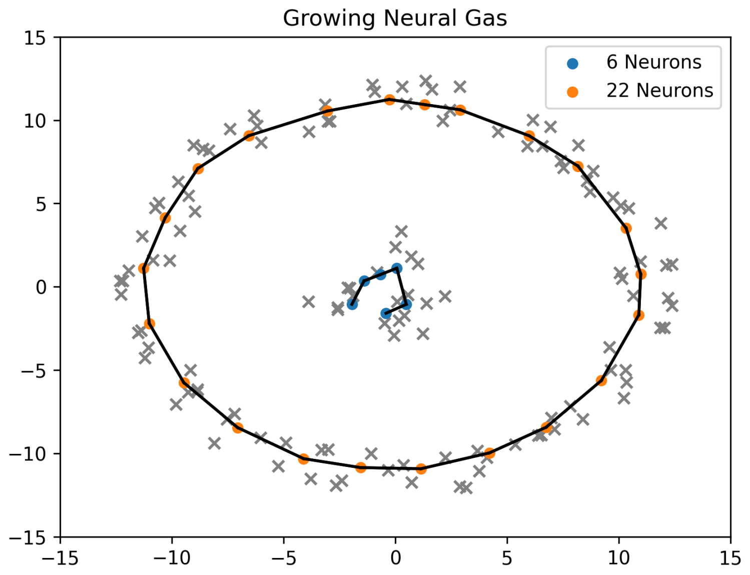
Figure 3 demonstrates why using a fixed number of prototypes, as in NG, is problematic in certain situations. One could imagine that the cluster in the center of this image is known from initialization data and the outer ring arose from streaming data. Five prototypes are sufficient to model the inner class, however more prototypes are clearly needed for the outer class. Growing neural gas ameliorates the issue suffered by neural gas in cases where different structures require different numbers of prototypes.

3. *Possibilistic K-nearest Neighbors (PKNN)*

PKNN [42] is a possibilistic extension of the fuzzy K-nearest neighbors algorithm [60], which is, in turn, an extension of the ubiquitous K-nearest neighbors algorithm [39]. The first step of PKNN is to soften the labels associated with the training data based on the labels of each



(a) Neural gas



(b) Growing neural gas

Figure 3: Demonstration of a case where growing neural gas is preferable to the standard neural gas. There are two structures in this toy dataset, both with similar point density but one having substantially more points than the other.

sample's neighbors according to

$$\tilde{\mu}^i(y) = \begin{cases} 0.51 + 0.49 \cdot \left(\frac{n_i}{K}\right), & \text{if } i = j \\ 0.49 \cdot \left(\frac{n_i}{K}\right), & \text{if } i \neq j \end{cases} \tag{3}$$

where $\tilde{\mu}^i(y)$ is the possibilistic label of y in class i, n_i is the number of neighbors of y in class i, and K is the number of neighbors under consideration. Intuitively, the "0.51" term ensures that each prototype has a majority membership in its assigned class, and the "0.49" term takes into account the labels of the neighbors of the prototype, which may disagree with the given class label. In the event that a prototype is surrounded entirely by members of other classes, it will have membership 0.51 in its assigned class and 0.49 in the surrounding class. Highly separable datasets will have soft labels that are effectively one-hot encodings of the hard labels.

When given a new sample x, its typicality in each class i is computed according to

$$\mu^i(x) = \frac{1}{K} \sum_{i=1}^{K} \tilde{\mu}^i(y_k) w_p(x, y_k), \tag{4}$$

where y_k is the k-th nearest neighbor to x. The possibilistic weight function w_p is given by

$$w_p(x, y_k) = \frac{1}{1 + \left[\max\left\{0, \|x - y_k\| - \xi\right\}\right]^{\frac{2}{m-1}}}, \tag{5}$$

where m and ξ (renamed from η in the original manuscript to prevent confusion) are parameters.

C. StreamSoNG Algorithm Description

Streaming soft neural gas (StreamSoNG) [85], explicitly referred to as StreamSoNGv1 for clarity, is a recently developed algorithm that leverages ideas from neural gas [69], possibilistic K-nearest neighbors (PKNN) [42], sequential possibilistic 1-means (SP1M) [76], and possibilistic C-means (PCM) [62] to model arbitrary topologies in feature space. In this section, we provide a detailed description of the algorithm, offering insights and criticism that are not present in the original manuscript. We then explain some shortcomings of StreamSoNGv1 and how we address them, culminating in an improved StreamSoNGv2, and evaluate its performance against the original StreamSoNGv1 and several other SDA algorithms.

StreamSoNGv1 is initialized with labeled data. First, it runs neural gas on each class to develop a set of per-class prototypes. When labels are not available for the initialization set, one could instead use the (hardened) cluster memberships obtained by running an offline clustering algorithm such as PCM. Additionally, StreamSoNGv1 could be adapted to skip initialization by simply putting all incoming points in the outlier list until a cluster is found, though this has not been tested.

The purpose of the initialization procedure is to seed the algorithm with good examples from some of the classes from the dataset in the hopes that new examples can be expressed by the degree to which they belong to these seed classes, which is not unlike the idea of hyperspectral endmember extraction [58].

During the streaming phase, new points are sequentially presented to the algorithm. First, the fuzzy memberships are computed for each neural gas prototype according to [2]

$$\mu^i(p) = \begin{cases} 0.51 + 0.49 \cdot \left(\frac{n_i}{K}\right), & \text{if } i = j \\ 0.49 \cdot \left(\frac{n_i}{K}\right), & \text{if } i \neq j \end{cases}, \tag{6}$$

where $\mu^i(p)$ is defined as the membership of prototype p in class i, j is the true class of p, n_i is the number of neighbors of p in class i, and K is the number of neighbors under consideration. We observe that Equation 6 is identical to Equation 3, but with notation adjusted for clarity in this context. The set of all $\mu^i(p)$ values are best understood as a row-stochastic matrix-valued function $U(p_{ik})$ containing membership values, with one row for each class and one column for each prototype.

The next step is to compute the typicality[3] of the incoming point x_t in each class i with respect to each of its K-nearest prototypes,

[2]This equation was originally presented in the context of the fuzzy K-nearest neighbors algorithm in Keller at al. [60].

[3]Memberships, like probabilities, are in $[0, 1]$ and sum to unity, whereas typicalities relax the latter constraint.

resulting in a matrix with a row for each class and a column for each of the K-nearest neighbors of x_t. In the usual case of Euclidean distance, typicality is computed as [4]

$$t_{ik}(x_t) = \frac{1}{1 + \left(\frac{\|x_t - p_{ik}\|^2}{\eta_i}\right)^{\frac{1}{m-1}}}, \tag{7}$$

where p_{ik} is the k-th closest prototype in class i to x_t. The variable $m > 1$ is known as the "fuzzifier" parameter and governs how quickly typicalities fall off to zero as distance increases, with higher values of m leading to slower transitions from regions of high typicality to regions of low typicality. We discuss the η parameters in great detail in Section 6, as it is notoriously difficult to estimate. Equation 7 replaces, but serves the same purpose as Equation 5 in PKNN.

Next, we compute the typicality of the incoming point with respect to each class i by taking a normalized dot product of the typicality of the incoming point in its K-nearest prototypes with the membership of those prototypes in their respective classes, given by

$$\bar{t}_i(x_t) = \frac{1}{K} \sum_{k=1}^{K} \mu^i(p_{ik}) t_{ik}(x_t). \tag{8}$$

This step can be viewed as taking the average of an element-wise product between the class memberships of the K-nearest prototypes to x_t

[4]This equation was originally presented in the context of PCM as Equation 8 in Krishnapuram and Keller [62]

and the prototype typicalities of x_t in its K-nearest neighbors. To artificially boost typicality values, a scaling function is applied to the result of Equation 9,

$$T_i(x_t) = \begin{cases} 0 & \bar{t_i} \leq 0 \\ 2\bar{t_i} - \bar{t_i}^2 & 0 < \bar{t_i} \leq 1 \\ 1 & \bar{t_i} > 1 \end{cases} \cdot \tag{9}$$

The scaling function is plotted in Figure 1 of the supplementary files, along with the identity function $f(\bar{t_i}) = \bar{t_i}$. The scaling function accomplishes two things: typicalities not in the range $[0, 1]$ are clipped into that range and typicalities already in $[0, 1]$ are increased. Our stance is that a good choice of the η parameters render the scaling function unneeded, as η plays a much larger role in the scale of resulting typicalities. The scaling function in Equation 9 is plotted in Figure 4.

Lastly, a classification decision is made by choosing the class with the highest value of $T_i(x_t)$. At this stage, x_t is marked either as an inlier or outlier based on whether $\max_i T_i(x_t)$ is above or below, respectively, a user-provided threshold we denote $t_{outlier}$.

Outliers are added to an outlier list and then clustered using a limiting case of the possibilistic one-means (P1M) algorithm (P1M as $m \to 1$) to search for emerging structures. If any are found, neural gas is run on points in the structure and the new prototypes are added

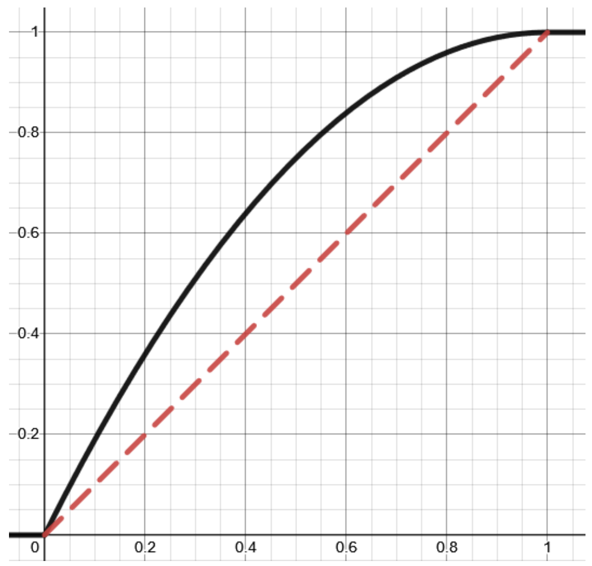


Figure 4: Plot of the scaling function used in StreamSoNG. The x-axis is $\bar{t_i}$ and the y-axis are values of $T_i(\bar{t_i})$ from Equation 9 (solid black line) and the identity function $f(\bar{t_i}) = \bar{t_i}$ (dotted red line).

to the existing prototypes. All outliers considered part of the new structure are removed from the outlier list.

For inliers, all existing prototypes in the same class as x_t are updated according to

$$p_{ik}^{t+1} = p_{ik}^{t} + \alpha T_i(x_t) e^{-k/\lambda}(x_t - p_{ik}^{t}). \tag{10}$$

We observe this to be identical to the update equation for neural gas (Equation 2 in Martinetz and Schulten [69]) with corresponding learning rate taken to be StreamSoNG's learning rate multiplied by the scaled typicality of x_t in the neuron's class.

Examples of how StreamSoNG performs on a synthetic two-dimensional dataset, as well as descriptions of key parameters and their effect on the algorithm, are shown in Appendix A.

D. StreamSoNGv2

At its core, StreamSoNGv2 operates in the same way as StreamSoNGv1. In this section, we present the adjustments that were made to StreamSoNGv1 to form StreamSoNGv2. The different initialization methods for StreamSoNGv2 are shown in Algorithm 3 and the update procedure is shown in Algorithm 4 Each component of the algorithm described in further detail later in this section.

1. *Growing Neural Gas*

Replacing the neural gas in StreamSoNGv1 with a growing neural gas is desireable for a number of reasons, including, but not limited to removing the need to specify the number of prototypes *a priori*, the ability for different classes to use only as many prototypes as needed, and GNG's propensity to make use of connections when moving prototypes. Modifying GNG to fit within the StreamSoNGv1 framework is straightforward and only requires scaling the learning rate of each prototype update by the typicality of a point in the class that the GNG is modeling. Algorithm 5 explicitly illustrates this adjustment, which, as the reader will notice, is almost identical to the standard GNG in Algorithm 2.

Algorithm 3 StreamSoNGv2 Initialization

```
procedure EmptyInitialization( )
    // Minimal initialization with no data
    Let SSv2.Structures = {}
    Let SSv2.Outliers = {}
    Let SSv2.$\eta$ = {}

procedure SupervisedInitialization($X$, $Y$)
    // $X = \{x_j \in \mathbb{R}^d, j = 1, 2, \ldots, N\}$, $N$ points with
        $d$ dimensions
    // $Y = \{y_j \in [0, 1]^C, j = 1, 2, \ldots, N\}$, soft label $y_j$ in
        $C$ classes for each $x_j$
    // If $Y$ is given as hard labels, they can first be softened
        by Equation 6
    Call EmptyInitialization( )
    for $i = 1, 2, \ldots, C$ do
        Let $U_i = \{y_{ji}, j = 1, 2, \ldots, N\}$
        Call TrainGNG in Algorithm 5 on $X$ and $U_i$
        Store the trained GNG in SSv2.Structures[$i$]
        Estimate $\eta_i$ for this class by minimizing the
            objective function in Algorithm 6
        Store $\eta_i$ in SSv2.$\eta$[$i$]

procedure UnsupervisedInitialization($X$)
    // $X = \{x_j \in \mathbb{R}^d, j = 1, 2, \ldots, N\}$, $N$ points with
        $d$ dimensions
    Let $Y$ be the cluster labels obtained by running a
        clustering algorithm on $X$
    Call SupervisedInitialization($X$, $Y$)
```

2. *DBSCAN*

Each class in StreamSoNGv1 is modeled using a neural gas, and thus arbitrarily-shaped classes can be modeled in StreamSoNGv1. However, the outlier list is clustered using P1M, which only finds hyper-spherical clusters. Arbitrarily-shaped classes can only be successfully discovered by StreamSoNGv1 if they are labeled and in the initialization set, or

Algorithm 4 StreamSoNGv2 Update

```
// $x \in \mathbb{R}^d$, a new point to process
// min_pts, the minimum number of points required to form a new class
// $t_{outlier}$, typicality that, below which, a new point is considered an outlier
// $\varepsilon_{bmu}$, $\varepsilon_{nbhd}$, $\lambda$, $\alpha_1$, $\alpha_2$, and $T$ as in Algorithm 2
// $\varepsilon_{DBSCAN}$ and min_samples, parameters for DBSCAN

procedure UpdateStreamSoNGv2($x$)
    Compute the soft labels $\mu^i(p)$ of each prototype $p$ in each $i \in$ SSv2.Structures
        according to Equation 6
    Set typicalities = {}
    for each $i \in$ SSv2.Structures do
        Compute $\bar{t_i}(x)$ in Equation 7 using SSv2.$\eta[i]$ and store it in typicalities[$i$]
    if max(typicalities) $< t_{outlier}$ then
        Set Label = $-1$
        SSv2.Outliers = SSv2.Outliers $\cup$ x
        if |SSv2.Outliers| > min_pts then
            Cluster SSv2.Outliers using DBSCAN with parameters $\varepsilon_{DBSCAN}$
                and min_samples
            if there are at least min_pts points in the majority cluster then
                Call TrainGNG($\varepsilon_{bmu}, \varepsilon_{nbhd}, \lambda, \alpha_1, \alpha_2, T$) in Algorithm 5 on all
                    outliers in the majority cluster
                Append the trained GNG to SSv2.Structures
                Estimate $\eta_y$ for this class by minimizing the objective function
                    in Algorithm 6
                Append $\eta_y$ to SSv2.$\eta$
                Remove all outliers in the majority cluster from SSv2.Outliers
                Set Label = |SSv2.Outliers|
    else
        Set $\hat{i} = \underset{i}{\text{argmax}}\ \bar{t_i}(x)$
        Set Label = $\hat{i}$
        for each $i \in$ SSv2.Structures do
            Call UpdateGNG($x$, $\bar{t_i}(x) \cdot \varepsilon_{bmu}$, $\bar{t_i}(x) \cdot \varepsilon_{nbhd}$, $\lambda$, $\alpha_1$, $\alpha_2$, $T$) in Algorithm 5
            // Check if best fit GNG should be expanded
            if $i = \hat{i}$ and enough time has elapsed since growth of the $i$-th GNG then
                Call ExpandGNG() in Algorithm 2 on the $i$-th GNG
    return Label, typicalities
```

if they start out as spherical so P1M can detect them and then evolve over time into a more complex shape. Using P1M to discover new

Algorithm 5 Modified Growing Neural Gas

```
// $U = \{u_j \in [0,1], j = 1, 2, \ldots, N\}$, typicality of $x_j$
    in this GNG
// Other parameters and methods as in Algorithm 2

procedure TrainGNG($X$, $U$, $\varepsilon_{bmu}$, $\varepsilon_{nbhd}$, $\lambda$, $\alpha_1$, $\alpha_2$, $T$)
    Initialize $W$ as two weights in $\mathbb{R}^d$
    repeat
        Sample a point $x_j \in X$
        UpdateGNG($x_j$, $u_j\varepsilon_{bmu}$, $u_j\varepsilon_{nbhd}$, $\lambda$, $\alpha_1$, $\alpha_2$, $T$)
    until convergence
```

classes belies the claim that StreamSoNGv1 can find arbitrarily-shaped clusters, and thus we find it important to replace P1M with a clustering algorithm that is aligned with the arbitrary shape claim.

We chose to use DBSCAN [37] as a drop-in replacement for P1M due both to its high regard within the unsupervised learning community and its support for arbitrary cluster shape. In terms of parameter selection, we now specify ε and `min_samples` parameters instead of the m and η required for P1M. The neighborhood size parameter ε determines the radius of an ε-ball around each point, and, if this ε-ball contains more than `min_samples` points, it is considered a core point. The union of overlapping core points then form clusters.

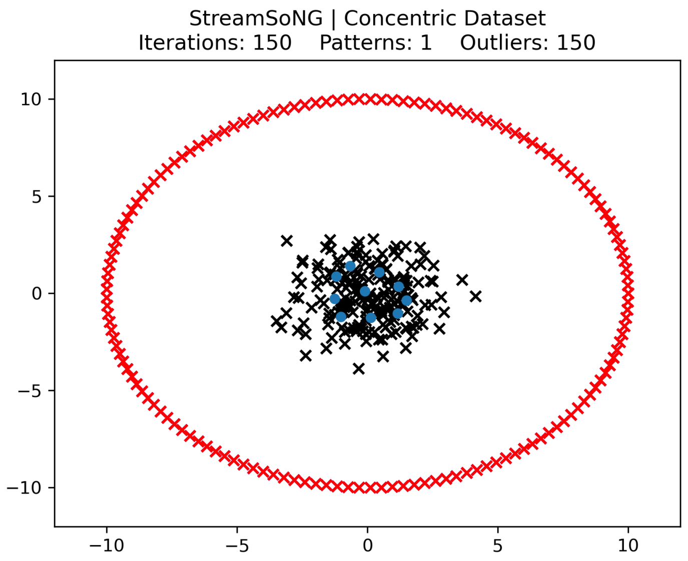
Both P1M (in the way that it is used in StreamSoNGv1) and DBSCAN make the assumption that clusters have similar density; P1M in the sense of finding `min_pts` points within radius η of a centroid and DBSCAN in the sense of finding a chain of ε-balls with `min_samples` in them. The ε and η parameters of DBSCAN and P1M, respectively, are related. Since we employ a procedure to infer η in PKNN (Section 6), which effectively serves the same purpose, we believe that good values for ε for DBSCAN can be discovered automatically as well.

Figure 2 in the supplementary material demonstrates a simple example of why the switch from P1M to DBSCAN is needed.

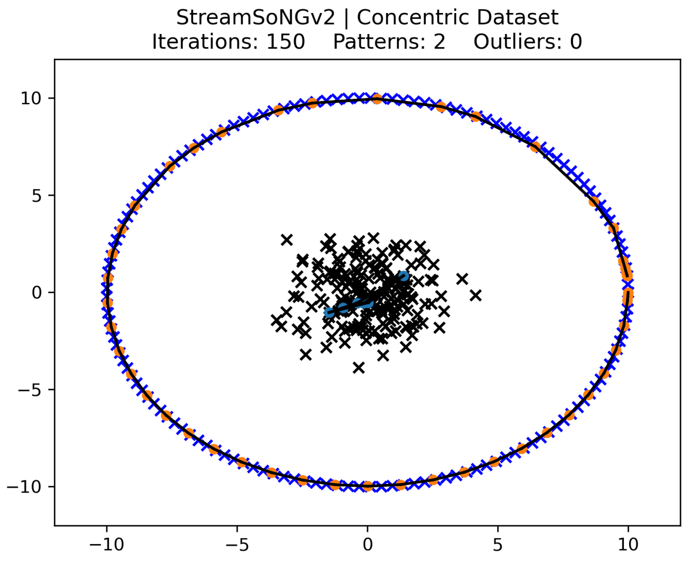
3. *Reprocessing Outliers*

In StreamSoNGv1, the only way for outliers to leave the outlier list is if they become part of a new class. However, definitions for known classes are always changing or expanding, meaning that what was thought to be an outlier early in the stream may in fact fit within an existing class once the class is further modeled by future points in the stream. Such a phenomenon can cause a point that was labeled an outlier in the past to remain an outlier forever because existing class definitions have encompassed that region of feature space to the degree that new points in that area will be marked as inliers, and hence not be added to the outlier list, preventing this outlier from being part of a cluster in the outlier list.

Taking inspiration from the work of Ghesmoune et al. in G-Stream [45], we periodically return the outlier list to the front of the stream in order to allow StreamSoNGv2 to determine whether points previously thought to be outliers now conform to a known structure. Our imple-



(a) StreamSoNGv1 is unable to detect new structures that are non-spherical, marking all points in the outer circle as outliers. The only class StreamSoNGv1 found is the center cluster, modeled with blue NG prototypes.



(b) Replacing P1M with DBSCAN allows StreamSoNGv2 to detect and model non-spherical structures without issue, indicated by the orange GNG prototypes in the annulus surrounding the middle cluster.

Figure 5: Synthetic dataset with a spherical initialization set centered at $(0, 0)$ and an outer circle that arises during the stream. The “x” marks are points in the dataset with blue indicating inliers and red indicating outliers. The purpose of this dataset is to demonstrate a simple example where the modifications we have made to StreamSoNGv1 allow it to detect structures it previously could not.

mentation of the "reservoir" idea of Ghesmoune et al. differs slightly from theirs in that they wait for the reservoir to become full before returning it to the front of the stream, whereas we perform this maintenance step every fixed number of iterations to keep the outlier list as clean as possible, easing the task of DBSCAN in clustering the outlier list.

Anecdotally, we have observed in synthetic datasets that periodic outlier resubstitution drastically improves the quality of the GNG prototypes when the data stream is randomly permuted. As the order of the stream commonly plays a large role in whether an SDA algorithm performs well, relative invariance to stream ordering is a very desirable property in an SDA algorithm.

4. *Updating All Classes*

After computing $T_i(x_t)$ for each class i in StreamSoNGv1, all except the maximum value are immediately discarded. This crisp decision is against the spirit of the principle of least commitment [59]. Ideally, fuzzy or possibilistic systems refrain from making crisp decisions in this way, which we implement here.

This could be accomplished in StreamSoNGv1 by simply applying Equation 10 over all classes i instead of just the one that best fits the sample point. In StreamSoNGv2, since we use GNG instead of NG, we instead run one iteration of GNG with

$$\varepsilon_{bmu} \leftarrow T_i(x_t)\varepsilon_{bmu} \quad \text{and} \tag{11}$$

$$\varepsilon_{nbhd} \leftarrow T_i(x_t)\varepsilon_{nbhd}. \tag{12}$$

This way, classes that are deemed highly dissimilar to the input will only move infinitesimally (or not move at all, given floating point precision), and classes that lay a non-negligible claim to the area of feature space containing the input point will move some prototypes to better accommodate the point.

Performing an update for all classes, instead of just the nearest, is desireable when class labels are not assumed to be mutually exclusive. If we had a "dog" class and a "brown" class, it would make sense that a brown dog should induce an update in both the "dog" prototypes and the "brown" prototypes. In fact, it would be reasonable to want two coincident prototypes in this situation that both model brown dogs: one in the "dog" class and one in the "brown" class. Future work in the domain of StreamSoNGv1 might explore the idea of shared prototypes between classes.

5. *Bounded Outlier List*

The outlier list in StreamSoNGv1 is unbounded and can potentially store the entire stream. For example, if η for P1M and $t_{outlier}$ are both very small, all points would be outliers and no outliers would cluster, hence remaining outliers forever. As a result, the entire stream would be permanently stored in the outlier list. Instead, the outlier list should be finite.

Even though modern computers have memory capacity larger than most people twenty years ago could dream of, there are still important reasons to bound the outlier list. First, the SDA model is intended to run *ad infinitum.* Repeatedly adding even a single bit of information that never gets deleted can be thought of as a memory leak and will eventually exhaust the memory of any device if run long enough, though, practically speaking, most use cases would not encounter this problem.

More importantly, a continuously growing outlier list will cause the time it takes to process new points to grow, as P1M is linear in the size of the outlier list n and DBSCAN is $O(n \log n)$. If the outlier list is clustered each time a new point arrives, the system would eventually take far too long to process incoming points. Alternatively, the outlier list could be clustered asynchronously to the rest of the algorithm, in which case excessively large outlier lists would lead to very long delays between when new structures are encountered in the data and when they are found by StreamSoNG.

A number of windowing approaches exist in the literature for limiting the amount of data that is maintained in SDA algorithms, such as damped windows [65], pyramidal windows [15], and sliding windows [90]. The latter, which entails only maintaining the o_{max} most recent outliers, is the most straightforward to employ and is implemented in StreamSoNGv2.

The choice of o_{max} is not especially critical in the convergence of StreamSoNGv2, and our rule of thumb is to make it as large as one feels comfortable. Setting o_{max} too small is the biggest issue that could arise from this change in StreamSoNGv1 as it would cause outliers to be discarded before enough context has arrived for them to be consider a new structure. Setting o_max too high could lead to the time and memory problems discussed earlier in this section, which we believe to be a comparatively smaller problem. Thus, for realtime applications where classification speed or embedded systems where memory is precious, one may see benefit from tuning o_{max} as low as possible. In general, and for the experiments in this research, we fixed $o_{max} = 10,000$ so that outliers are never discarded.

6. *Estimating the η Parameters*

In Krishnapuram and Keller [62], the η parameters in Equation 7 are simply referred to as "suitable positive numbers" and control the region of influence around each prototype. Choosing a good value of the

η parameters is crucial to achieving meaningful decision regions and, unfortunately, there is no known optimal way to choose them. In the limiting case where $\eta \to \infty$, all PCM centroids will converge to the grand mean of the data. On the other end of the spectrum, where $\eta \to 0$, the update equations in PCM will not move the centroids; they will remain where they started indefinitely. It is unclear and likely situation-dependent whether there should be one η parameter for the whole algorithm or one per class, whether the η parameters should be constant and, if not, of what their variability should be a function. Choosing η manually in a streaming setting is even more difficult, as making decisions about structures in the data that don't exist yet is problematic. As the choice of η is the biggest drawback in PCM and related algorithms, with the possible exception of initialization, many heuristics have been devised in the literature to select values for η [63, 75, 84, 88].

Perhaps the largest improvement of StreamSoNGv2 over its predecessor is our approach to inferring a good value of η for each class in an unsupervised setting. When choosing η parameters manually, we found the most effective approach was to vary η until the typicalities were spread out "nicely" over the interval $[0, 1]$. That is, we want the meaning of a typicality of, say, 0.5, to be the same, regardless of the model. This would translate to typicality values that can be thresh-

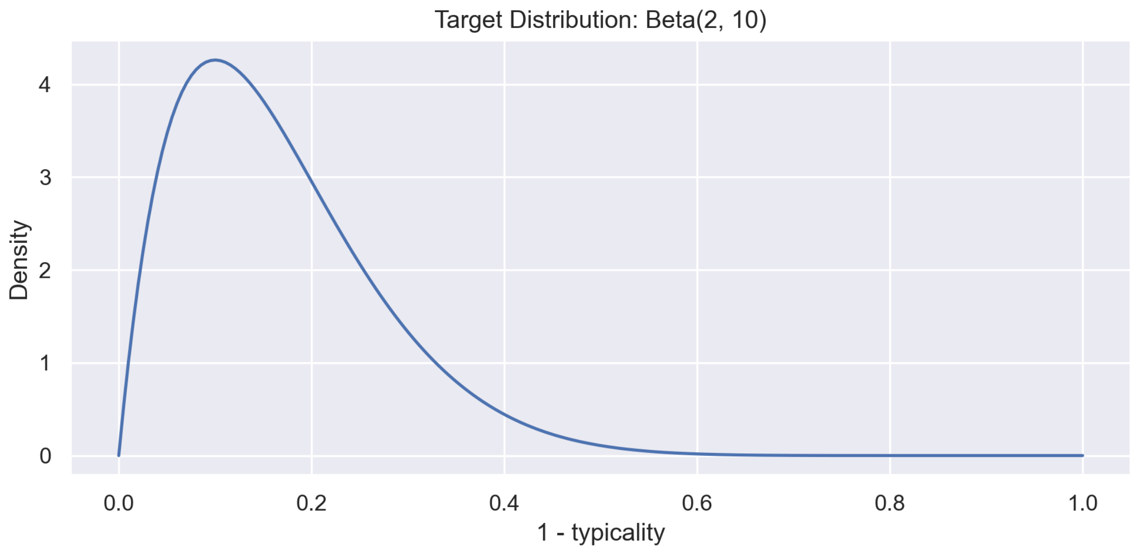


Figure 6: Target distribution used for η estimation in StreamSoNGv2, Beta$(2, 10)$. Since we want typicality of points in the currently examined class to be large, we want $1 - t$ for some typicality t to follow this distribution, which is equivalent to flipping this PDF over the line $x = 0.5$. Any distribution with the desired shape can be used here.

olded in the same was as p-values in statistics, independent of the model that produced the measurement.

First, we identify a statistical distribution φ whose PDF has the shape we want typicalities to take on. This can be any distribution, but we chose $\varphi = \text{Beta}(2, 10)$ reflected over the line $x = 0.5$ for our experiments, shown in Figure 6. We then define an objective function to minimize of the form

$$f(\eta; X, \alpha, \beta, \gamma, \varphi) = \alpha T_w(X_\eta; \varphi) + \beta U(X_\eta) + G(\eta; \gamma) \tag{13}$$

where T_w is a weighted Cramér-von Mises score, U is a measurement of the how much typicality is allocated to prototypes in other classes, and

G is a regularization function. Each of these three functions are described in the following paragraphs. We use the notation X_η to denote typicalities generated from Equation 4 using a set of points X with a particular choice of η. In addition, α and β are hyperparameters that balance the contribution of the T_w and U terms in the objective function.

The Cramér-von Mises criterion [35] measures the goodness of fit between a set of samples and a statistical distribution and is given by

$$T(X;\varphi) = \frac{1}{12n} + \sum_{i=1}^{n} \left[\frac{2i-1}{2n} - \Phi\left(x^{(i)}\right) \right]^2, \tag{14}$$

where x_i is the i-th sample in X, n is the number of samples, $x^{(i)}$ is the i-th smallest sample, and Φ is the CDF of the distribution being tested. We generalize this measure for use in our objective function by introducing weights that allow us to control how much of an impact each sample has on the overall function value. The weighted extension of the Cramér-von Mises criterion is

$$T_w(X;\varphi) = \frac{1}{12\sum_{j=1}^{n} w_j} + \sum_{i=1}^{n} w_i \left[\frac{w_i + 2\sum_{j=1}^{i-1} w_j}{2\sum_{j=1}^{n} w_j} - \Phi\left(x^{(i)}\right) \right]^2 \tag{15}$$

where w_i is the weight given to $x^{(i)}$, the i-th smallest sample in X. Simply put, we have only made the natural generalization that n be replaced with the total weight, the sum index i be replaced with the total weight “so far”, and each term of the summation be scaled by the weight of the sample under consideration. Additionally, we define the first term of the summation in the numerator of Equation 15 to be zero when $i = 1$. Notice that when all $w_i = 1$ for $i = 1, \ldots, n$, we obtain the original Cramér-von Mises criterion.

A low value of the Cramér-von Mises criterion is a strong indicator that the samples are from the distribution under examination. The samples X, in this case, are typicalities produced by PKNN, and are thus a function of m, K, and, most importantly, η. We fix $m = 2$ and $K = 5$ for our experiments here and perform a numerical minimization procedure on T_w over η.

The second term of the objective function is a measure of how much typicality is allocated to other classes for a particular choice of η. We wish to apply some penalty to the optimizer for picking very large values of η as a means of improving the consistency of convergence. One way to do this, which we employ here, is to penalize high typicality in other classes besides the one we are examining, denoted $\hat{c}$. We use the average to aggregate typicality of all points in all classes other than $\hat{c}$,

$$U(X_\eta) = \frac{1}{N(C-1)} \sum_{j=1}^{N} \sum_{\substack{i=1 \\ i \neq \hat{c}}}^{C} \mu^i(X_\eta(j)). \tag{16}$$

The last term of the objective function is simply to ensure that there always is a gradient guiding the optimizer toward reasonable magnitudes of η, given by

$$G(\eta; \gamma) = e^{\gamma\eta}, \tag{17}$$

where in our experiments, $\gamma = 0.01$. We have noticed anecdotally that sometimes, especially when using very small sample sizes, the optimizer sometimes diverged to very large η due to the numerical instability of very small gradients. This term has little to no effect on the optimization procedure until η gets quite large, allowing the other terms of the objective function to easily overpower it, but when the gradient of the other two terms is effectively zero or when η is extremely large, this regularization term keeps η from diverging toward infinity.

The pseudocode for evaluating the full objective function on an η proposed by the optimizer is shown in Algorithm 6. We use the L-BFGS-B algorithm for optimization, which is a bounded [28] limited memory [66] variant of the BFGS [27, 40, 47, 78] algorithm. In particular, we employ the SciPy [82] implementation of L-BFGS-B.

7. *Miscellaneous Changes*

Remove Scaling Function. The scaling function in Equation 9 is not necessary with good parameter choices. Well-chosen η and m parameters are capable of yielding typicalities on the order of magnitude that one might desire. As future work, we may be able to employ a strat-

Algorithm 6 Objective Function for η

```
// $\hat{\eta}$, candidate value for $\eta$ proposed by the optimizer
// $X = \{x_j \in \mathbb{R}^d, j = 1, 2, \ldots, N\}$, $N$ points with $d$ dimensions
// $W = \{W_i \in \mathbb{R}^{n_i \times d}\}, i = 1, 2, \ldots, C$, the set of all GNG prototypes in each
//     of $C$ classes
// $\hat{c}$, the label of the class for which we want to find $\eta$
// $K$, the number of neighbors to use in PKNN
// $m$, the fuzzifier parameter to use in PKNN
// $\alpha$, balance terms in objective function
// $\varphi$, the target distribution to use

procedure EtaObjective($\hat{\eta}$; $X$, $W$, $\hat{c}$, $K$, $m$, $\alpha$, $\beta$, $\varphi$)
    Use Equation 3 to compute soft labels $\tilde{\mu}^i(y)$ for all GNG prototypes $y$
        in all classes $i$.
    Use Equation 4 to compute typicality $\vec{\mu}_i = \{\mu^i(x)\}$ for each $x \in X$ in all
        classes $i$ using $m$ and $\hat{\eta}$.
    Compute the weighted Cramér-von Mises criterion $t = T_w(\vec{\mu}_{\hat{c}}, \varphi)$ using
        Equation 15 with $w = \mathrm{sort}(\vec{\mu}_{\hat{c}})$.
    Compute $u$, the amount of typicality allocated to other classes according
        to Equation 16.
    Compute $g$, the penalty term for large $\eta$ according to Equation 17.
    return $\alpha t + \beta u + g$
```

egy similar to batch normalization in neural networks [55] to coerce typicalities into better filling the interval $[0, 1]$ without necessitating a hard-coded scaling function.

Start in Streaming Phase. As is, StreamSoNGv1 requires an initialization set that, ideally, has labels associated with it. The algorithm would have more utility if it could also start directly in the streaming mode. To accomplish this, we label all stream points as outliers until a structure is found, after which time StreamSoNGv2 operates in the usual way. StreamSoNGv2 can still utilize an initialization set, just as StreamSoNGv1, but it is no longer mandatory.

Soft Label Initialization. Initialization for StreamSoNGv1 is done using hard labels, and thus requires the acquisition of hard labels for each initialization point, either by having them given *a priori* or by computing them using a clustering algorithm. Even when soft labels, *i.e.*, fuzzy or possibilistic labels, are known, they must be hardened into a crisp partition so that NG can be run on each class.

In StreamSoNGv2, we train a GNG for each class, but train each GNG on the whole dataset and scale the learning rate by the typicality of the currently considered point in the class associated with the GNG. Specifically, the update in the GNG for class i, when considering the j-th sample x_j whose soft label in class i is given by μ_{ij}, is carried out with a learning rate of

$$\varepsilon_{bmu} \leftarrow \mu_{ij}\varepsilon_{bmu} \quad \text{and} \tag{18}$$

$$\varepsilon_{nbhd} \leftarrow \mu_{ij}\varepsilon_{nbhd}, \tag{19}$$

where ε_{bmu} and ε_{nbhd} are hyperparameters for the base learning rates for GNG.

When soft labels are not known, or only hard labels are given for initialization, we soften the labels using Equation 6 before carrying out the previously described procedure.

E. Experiments

Based on the Carnein and Trautmann survey [31] described in Section II-A, we have identified five prime candidates for comparison with StreamSoNGv1 [85] and StreamSoNGv2. We chose two classical SDA algorithms (CluStream [15] and DenStream [29]), one competitive learning algorithm called evoStream [30]), and two SDAs identified by Carnein and Trautmann to be state-of-the-art (D-Stream [32] and DBSTREAM [50]). All comparison algorithms are run using implementations from the `stream` [49] and `streamMOA` [48] packages in the R language. Algorithms are evaluated based on the purity [61] and normalized mutual information (NMI) [64] measures.

It should be noted that the purity and NMI measures evaluate these algorithms based only upon crisp labels, and were chosen over soft metrics so as not to handicap the methods that StreamSoNGv2 is compared against. The reader should be aware that, in addition to crisp labels, StreamSoNGv2 also produces valuable soft labels.

G-Stream [45], a related algorithm to StreamSoNGv1 and StreamSoNGv2 that also uses growing neural gas in the domain of SDA, is not considered for comparison here because its purpose is only to summarize the dataset in a single pass, rather than classify each incoming point. However, G-Stream brings some very useful ideas to the SDA

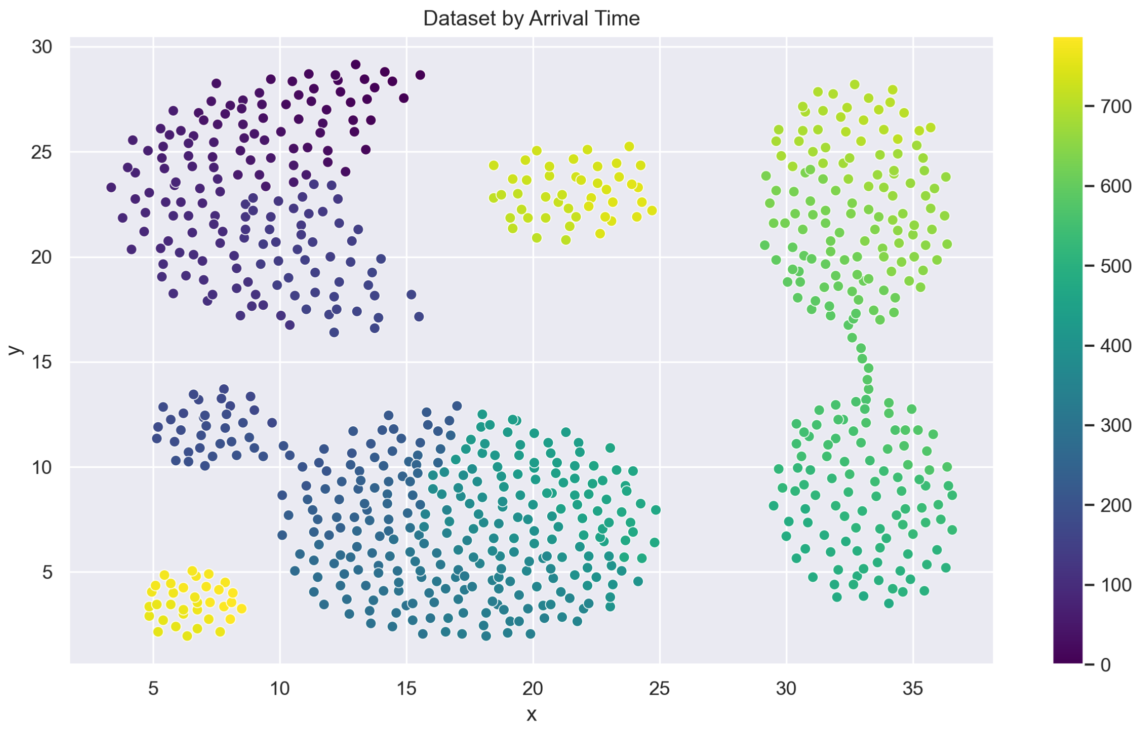


Figure 7: Synthetic dataset published by Gionis et al. [46] and presented sequentially as a stream. Points in the dataset are colored by arrival time.

domain, namely its concept of a reservoir which was the inspiration behind StreamSoNGv2's outlier resubstitution procedure.

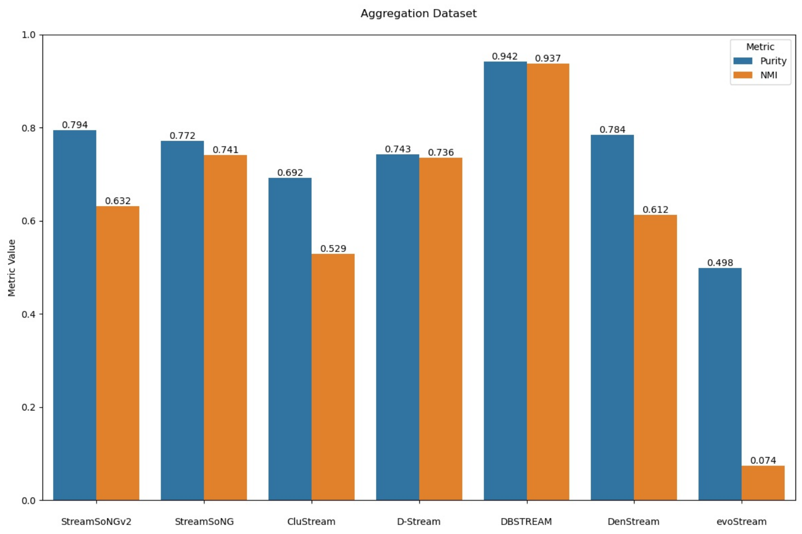
1. *Synthetic Dataset*

We begin with an evaluation of StreamSoNGv2 on synthetic dataset originally published by Gionis et al. [46]. A visualization of this dataset can be seen in Figure 7. We performed three experiments on this dataset, varying the order of the stream in each but keeping all parameters to each algorithm fixed.

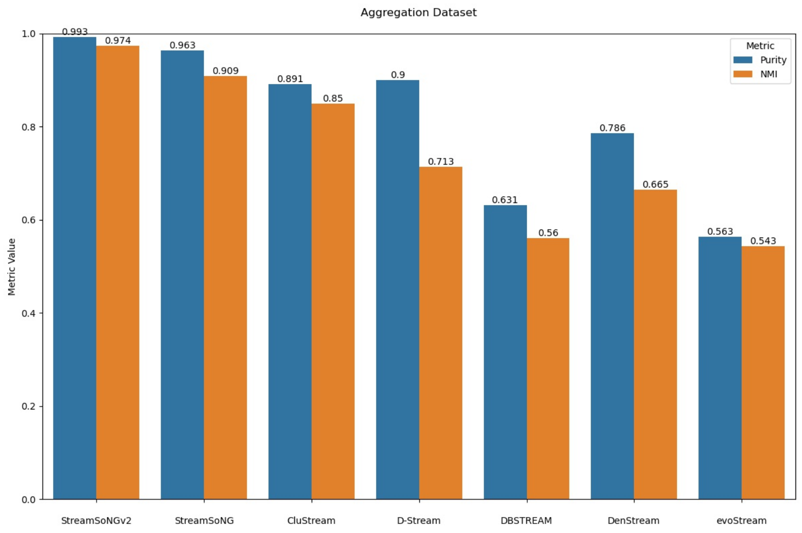
In all experiments, the first 200 samples in the stream are used as initialization with labels in all algorithms and the rest of the stream is presented one at a time. The first experiment presents the stream in the order it was originally published, with each cluster arriving in order. The initial state and final state of StreamSoNGv2 is visualized in Figure 8a along with performance metrics of all algorithms considered. We observe fairly similar performance among most algorithms, with DBSTREAM achieving the highest score among these metrics and evoStream the lowest (evoStream only found two clusters). The reason StreamSoNGv1 and StreamSoNGv2 do not perform as well in this experiment as they do in the others is because the highly non-stationary nature of this stream ordering causes the algorithms to be very sensitive to both learning rate and η. In the interest of fair comparison, we did not do significant parameter tuning on any of the algorithms here, and thus our choice of the key parameters in StreamSoNGv1 and StreamSoNGv2 were not optimal for this dataset.

In Figure 8b, we show the result of completely shuffling the stream. This leads to all classes being present in initialization, but the classes becoming more dense as the stream progresses. Other algorithms in the domain of SDA usually degrade severely in performance when the order of the stream is permuted, but seeing all classes in initialization puts StreamSoNGv1 and StreamSoNGv2 at a large advantage.

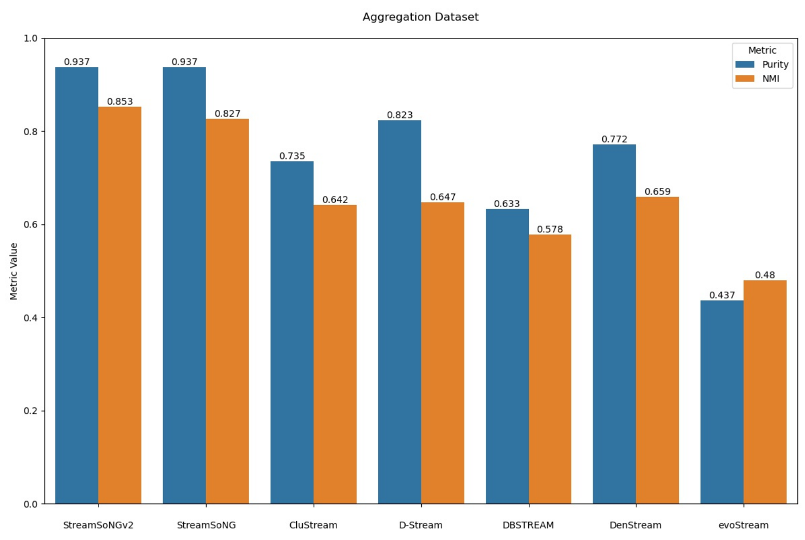
Finally, in Figure 8c, we ensure that two classes are not in the initialization set in order to evaluate whether each algorithm detects the ar-



(a) Original stream order



(b) Shuffled stream



(c) Shuffled stream, ensuring that two classes do not appear in initialization

Figure 8: Performance metrics and comparison with other algorithms on the synthetic dataset of Gionis et al. [46] with varied stream permutation methods.

rival of the new classes. We observe similar trends as in the previous two experiments on this dataset with StreamSoNGv1 and StreamSoNGv2 outperforming the other methods. Beyond achieving higher scores relative to the crisp purity and NMI metrics, the soft classification labels produced by StreamSoNGv1 and StreamSoNGv2 are valuable, especially in regions of feature space where points in close proximity may be have discrepancy in their ground truth labels.

2. *KTH-TIPS2B*

We also consider is a real texture dataset, the KTH-TIPS2b database [68] of eleven texture classes at varying illumination, pose, and scale. KTH-TIPS2b is valuable in that some classes are quite homogeneous, such as the “aluminum foil” class, and other are not, such as the “wool” class. This provides the opportunity to evaluate SDA algorithms on their ability to model highly disjoint classes.

We prepare this dataset for use in SDA as follows. First, we convert all images to grayscale, considering only the 4,319 images that are 200×200 pixels in resolution (some are a few pixels smaller in one or both dimensions). Then, we quantize each image to have only four distinct pixel values based on whether original grayscale intensities fall into the range $[0, 64)$, $[64, 128)$, $[128, 192)$, or $[192, 256)$. Lastly, we extract six features based on the gray-level co-occurrence matrix of the quantized image (contrast, dissimilarity, homogeneity, energy, correlation, and angular second moment) at four angles, $\frac{k\pi}{2}$ for $k = 0, 1, 2, 3$, as described by Hall-Beyer [51] using `scikit-image` [81].

The images are sequenced by taking a random walk over the classes in the dataset according to the transition matrix $P = \{p_{ij}\}$ with entries

$$p_{ii} = 0.99, \quad p_{ij} = \frac{0.01}{C - 1}, \tag{20}$$

where p_{ii} is the probability of sampling (without repetition) an image from the same class i as before, and p_{ij} is the probability of switching from class i to class j. With $p_{ii} = 0.99$, we have, on average, 100 consecutive samples from each class before switching.

A t-SNE plot of these features along with their labels is shown in

Figure 9. We can clearly see that the two-dimensional embedding produced by t-SNE indicates highly overlapping classes in the data. This observation is supported by much lower performance metrics across the board in Figure 10.

While StreamSoNGv2 does exhibit slightly higher performance metrics than other algorithms on this dataset, we do not believe that is the most important take-away from this experiment. These performance metrics rely on the assumption that the production of classification decisions that agree with the crisp labels of this dataset is the only objective. Figure 11 demonstrates the first time StreamSoNGv2 encounters a sample from the "cotton" class. Having no context to classify this sample as "cotton", it instead assigns this sample a typicality of around 0.6 in the known "linen" class. Examination of the initialization set for this dataset shows that there are indeed "linen" samples that closely resemble the "cotton" sample. Cases like this count against the algorithm in terms of performance metrics, but from a qualitative point of view, it is reassurance that StreamSoNGv2 has both learned and maintained useful knowledge from earlier in the stream despite no longer being able to examine older data points.

F. Conclusion

Based on StreamSoNG (v1) algorithm, we created a new verse that significantly extends it in a number of ways. Most significantly, automating the selection of η through an optimization procedure has made the

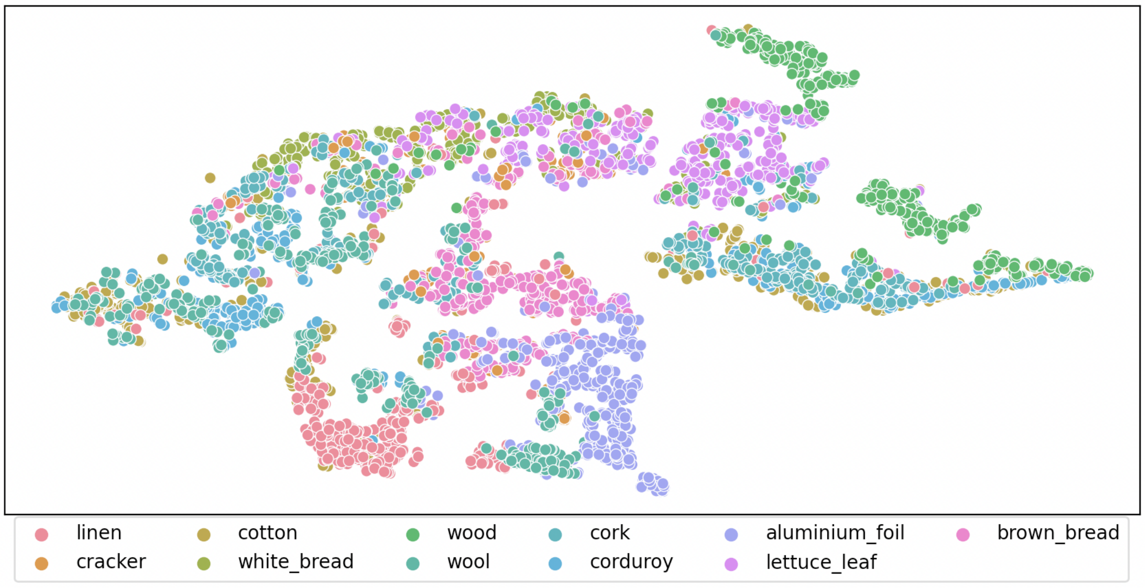


Figure 9: t-SNE embedding of 24-dimensional features extracted from the KTH-TIPS2b dataset.

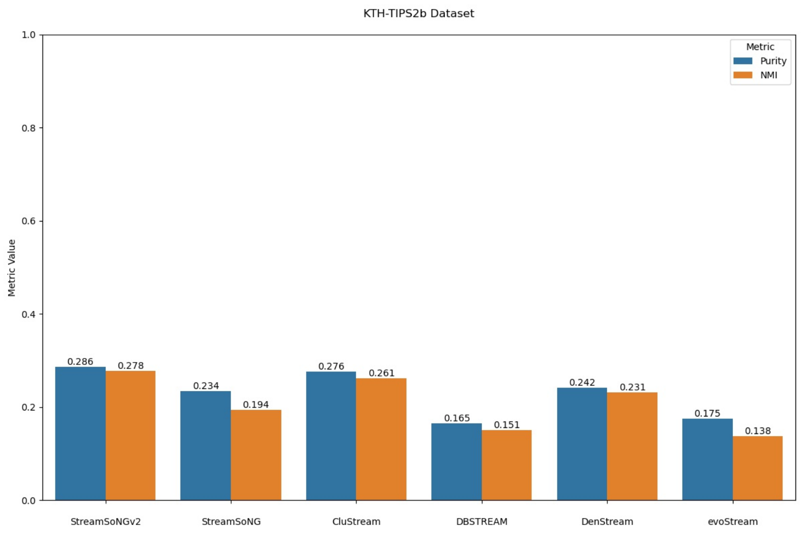
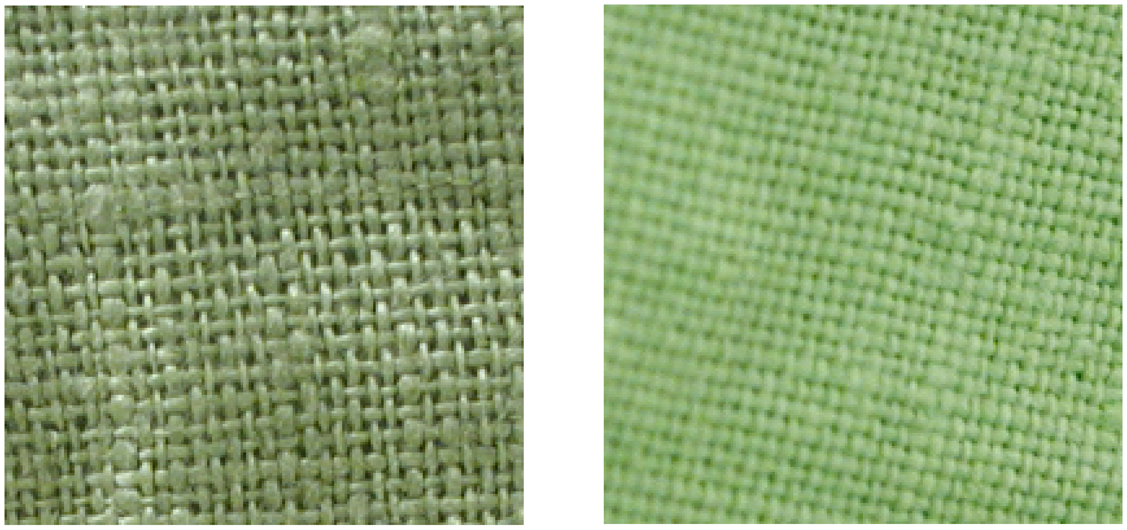


Figure 10: Performance metrics of each algorithm on the KTH-TIPS2b dataset.



(a) Sample from the "linen" class in the initialization set for StreamSoNGv2.

(b) First appearance of "cotton" class during the streaming phase (not seen in initialization).

Figure 11: Example of how StreamSoNGv2 makes explainable decisions, even if they don't agree with ground truth labels. StreamSoNGv2 assigned this cotton sample a typicality of 0.6 in the known "linen" class and a typicality of zero in other classes, and, even though this is not the correct crisp label, most humans would agree is a reasonable behavior.

most difficult part of applying StreamSoNGv1 to new datasets, parameter selection, substantially easier. Beyond that, using DBSCAN instead of P1M to cluster the outlier list has enabled the detection of arbitrarily shaped classes arising in the data stream, and the upgrade from neural gas to growing neural gas has ameliorated the need to specify the number of neurons used to model each class. We have also made a number of smaller modifications to StreamSoNGv1 to address what we believe to be shortcomings in the original design.

On synthetic data, we demonstrated that StreamSoNGv2 exhibits a higher invariance to permuting the stream than other related algorithms. Furthermore, StreamSoNGv2 performs marginally better than the competition on the KTH-TIPS2b dataset, though not to such a degree that one would be justified in saying that it is a generally superior algorithm for this task. Instead, we draw the following conclusions based on our experiments: (1) StreamSoNGv2 clearly shows improvement to StreamSoNGv1, (2) StreamSoNGv2 is competitive with state-of-the-art SDA algorithms, and (3) StreamSoNGv2 provides more detailed information about classification decisions than the other algorithms.

IV. Single-pass Possibilistic Clustering

Abstract — Streaming clustering is a domain that has become extremely relevant in the age of big data, such as in network traffic analysis or in processing continuously-running sensor data. Furthermore, possibilistic models offer unique benefits over approaches from the literature, especially with the introduction of a "fuzzifier" parameter that controls how quickly typicality degrades as one gets further from cluster centers. We propose a single-pass possibilistic clustering (SPC) algorithm that is effective and easy to apply to new datasets. Key contributions of SPC include the ability to model non-spherical clusters, closed-form footprint updates over arbitrarily sized damped windows, and the employment of covariance union from the multiple hypothesis tracking literature to merge two cluster mean and covariance estimates. SPC is validated against five other streaming clustering algorithm on the basis of cluster purity and normalized mutual information.

A. Introduction

The formulation of the streaming clustering problem varies depending on the source. In general, there is agreement that points from a data stream cannot be retained, they must be processed and then discarded. While this requirement may seem arbitrary in a time where buying more hardware is cheaper than improving methodology, certain data sources (*e.g.*, network traffic) simply provide so much information at such a high rate that the time and memory resources needed to iterate over the data are astronomical. Thus, there is high demand for streaming data analysis (SDA) algorithms, that is, algorithms that only make a single pass over the data.

We have observed a lack of possibilistic approaches to streaming clustering in the literature, which we believe (and will show) to have numerous desireable properties to offer. In this paper, we propose a single-pass possibilistic clustering (SPC) algorithm for the task of streaming clustering. As the name suggests, SPC adopts a possibilistic model, but with modifications that enable it to detect arbitrarily-shaped clusters.

SPC maintains a set of n structures that are intelligently placed in feature space so as to best describe the data stream. When a new point arrives, it is given its own structure and then, to keep the number of structures constant, the two most compatible structures are merged with respect to a Mahalanobis distance-based typicality measure. The priority given to recent points can be adjusted with decay factor parameters γ and β. To obtain a clustering of the stream, DBSCAN [38] is employed using a distance measure derived from SPC's typicality measure.

We evaluate the proposed SPC on stationary and non-stationary datasets of varying dimensionality and show that SPC achieves very high performance metrics in all cases, consistently either outperforming or staying competitive with related algorithms. As a qualitative evaluation, we also show that the decision region induced by SPC structures on two dimensional data is accurate to what human intuition would suggest.

The rest of the paper is organized as follows. Section B identifies related algorithms to SPC in the literature and details their similarities to and differences from SPC. Next, the SPC algorithm is described in detail in Section C, with the full algorithm presented in Algorithm 7. SPC is then evaluated against five state of the art streaming clustering algorithms on several datasets in Section D and conclusions about SPC are drawn in Section E.

B. Related Work

Few approaches in the past have sought to obtain a meaningful possibilistic clustering from data streams, though notable methods include Hu et al. with an evolutionary approach [52] and Wu et al. with a neural gas-based approach [86]. Additionally, Škrjanc et al. study the use of various norm-inducing matrices for evolving possibilistic clustering in data streams [80], including the inverse covariance matrix (*i.e.*, Mahalanobis distance) which we employ in this work.

It is common for algorithms that operate on streaming data to employ some kind of windowing model to allow the algorithm to focus on newer and potentially more relevant data points. The most used windowing models in the literature are sliding windows, damped windows, and pyramidal windows. The general idea of sliding windows is to only consider a fixed number of recent data points, whereas all points in the stream have an impact in damped and pyramidal window approaches. Pyramidal windows attempt to summarize the entire stream with a finite amount of information by maintaining coarser detail based on how far back points are in the stream. As damped windowing is employed in this work, we describe it in greater detail in this section.

The idea of damping has been studied for centuries in the form of damped motion in physics. The first use of damped windows in streaming clustering seems to be with DenStream [29] as a mechanism to determine when micro-clusters should be discarded. Since then, as Zubaroğlu et al. [90] point out, a number of algorithms have used damped windowing [19, 20, 22, 50, 53] in some capacity.

Many streaming clustering algorithms (*e.g.*, [19, 20, 22, 29, 50, 53]), including SPC, use the idea of damped windows to assign a weight to previously seen points in the stream, usually as a means to determine when to discard information about these points. Less commonly, damped window mean estimates are used, and applying a damped window model to covariance estimation is novel to streaming clustering.

From the crisp streaming clustering literature, DBSTREAM [50] has the most overlap with SPC, primarily due to both algorithms' dependence on DBSCAN [38] and damped windows. Key differences are that DBSTREAM bases its streaming updates on ideas from DBSCAN, whereas SPC uses DBSCAN directly for offline clustering, albeit with a specialized distance function. As for damped windows, DBSTREAM is one of many algorithms to use them in determining cluster weights.

Bechini et al. have proposed TSF-DBSCAN [22] which also shares some ideas with SPC. Both algorithms use DBSCAN, however TSF-DBSCAN uses a fuzzy extension of DBSCAN as an offline clustering procedure and SPC uses DBSCAN with a typicality-aware distance function. Again, both algorithms, like many others, use damped windows for measuring the weight of a structure. The most notable, and fundamental, difference is that TSF-DBSCAN is built on a fuzzy framework and SPC is built on a possibilistic framework.

Finally, SPC bears some resemblance to Gaussian mixture models (GMMs) in the sense that both models use estimates of the mean and covariance to find hyperellipsoidal clusters in data. Some key distinctions, however, are that GMMs require repeated iteration over the data, a fixed and pre-specified number of clusters (mixture components), and cluster memberships that sum to one. SPC, respectively, summarizes the dataset over a single pass, uses a fixed number of structures to model an arbitrary number of clusters, and uses a possibilistic measure of cluster belonging.

C. Methodology

We have divided the methodology of SPC into three sections. In Section 1, we motivate the need for a possibilistic model as opposed to probabilistic model for streaming clustering, and then describe the proposed possibilistic model. In Section 2, we explain how the structures used in SPC can be represented with fixed-size footprints and how to perform relevant operations using these footprints. Then, in Section 3, we outline the method of covariance union [57] from the domain of multiple hypothesis tracking and how it is valuable in streaming clustering. Finally, Section 4 contains an intuitive description of the SPC algorithm with the full pseudocode provided in Algorithm 7.

1. *Typicality with Mahalanobis Distance*

When using a mean and covariance matrix to model a structure, the natural method of determining the degree to which an arbitrary point belongs to the structure is to treat it as a Gaussian distribution and use the PDF to compute the probability that the point may have come from it. However, a possibilistic approach introduces a valuable "fuzzifier" parameter that controls how quickly typicality degrades when getting further from a structure. To compute the typicality of a point x within a structure s, the general equation, originally presented by Krishnapuram and Keller [62], is given by

$$u_m(s,x) = \frac{1}{1 + \left(\frac{d(s,x)^2}{\eta}\right)^{\frac{1}{m-1}}} \tag{21}$$

where $m > 1$ is a fuzzifier parameter, $d(s,x)$ is a distance measure between the structure s and a point x, and η is a "suitable positive number" that acts as a scale parameter governing the (squared) distance at which typicality reaches 1/2.

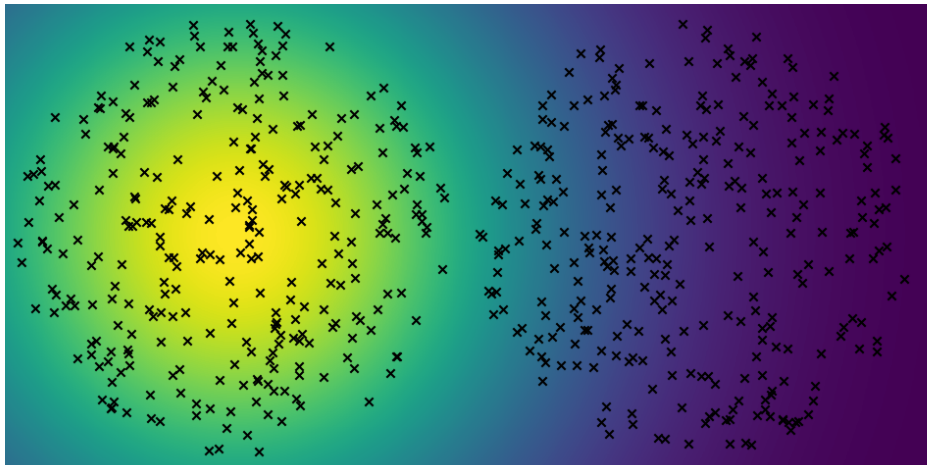
In the possibilistic C-means algorithm (PCM) [62] where Equation 21 was presented, the distance was defined to be Euclidean, thus constraining PCM to only find hyperspherical clusters. Replacing Euclidean distance with Mahalanobis distance leads to a class of typicality measures on m given by

$$u_m(x, \mu, \Sigma) = \frac{1}{1 + [(x-\mu)^T \Sigma^{-1} (x-\mu)]^{\frac{1}{m-1}}}. \tag{22}$$

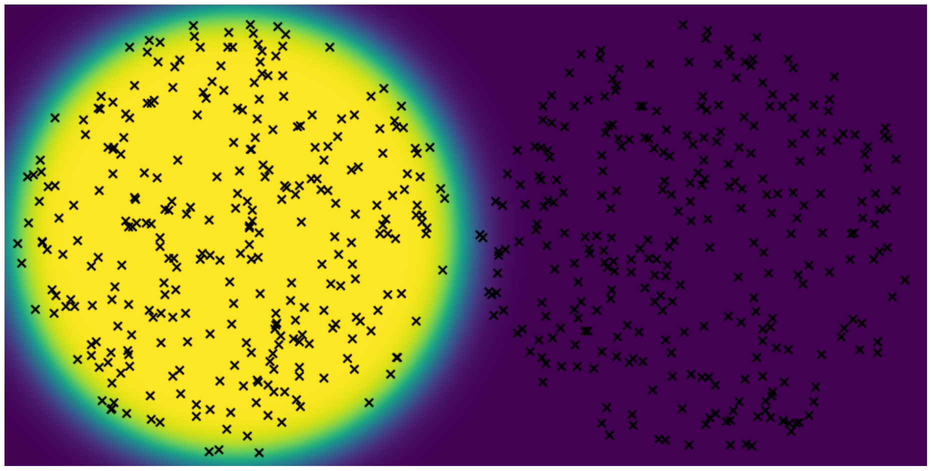
where μ is a structure mean, Σ is a measure of structure covariance, and m is the fuzzifier parameter as before. When $\Sigma = \eta\mathbb{I}$, Equation 22 reduces to Equation 21. Generalizing η to be a covariance matrix allows for the detection of hyperellipsoidal clusters as in the possibilistic Gustafson-Kessel (PGK) algorithm [62].

It should be noted that Equation 22 no longer satisfies the necessary conditions for convergence of the optimization procedure presented by Krishnapuram and Keller [62], but, in this work, we are using an entirely different method for finding structure in the data stream.

The introduction of the fuzzifier parameter in possibilistic models is especially useful when clusters in the data are very close together, but not overlapping. A synthetic dataset that illustrates this scenario is shown in Figure 12. Choosing the fuzzifier parameter m to be small allows the possibilistic model to tightly cover the left circle without allowing points in the right circle to have high typicality in it. A Gaussian model, for example, cannot cover the left circle without assigning high probability to points in the right circle as well.



(a) A Gaussian model assigns high probability in some points in the right circle.



(b) The fuzzifier parameter m in the possibilistic model allows fine control over typicality falloff, effectively separating the two circles. Here, we set $m = 1.1$.

Figure 12: Motivation for using a possibilistic model over a probabilistic model in SPC demonstrated on a synthetic dataset of two near-overlapping circles.

A key measure that is utilized in SPC is the distance between two structures s_1 and s_2. This distance must take into account the mean and covariance of both structures, and we would be discarding important information to simply compute the Euclidean distance between their means. We instead utilize the typicality measure of Equation 22 and transform it into a distance measure according to

$$D(s_1, s_2) = 1 - u_m(\mu_2, \mu_1, \Sigma_1)u_m(\mu_1, \mu_2, \Sigma_2). \tag{23}$$

The product of the typicality of μ_2 in s_1 and μ_1 in s_2 is used to make the measure symmetric, and the subtraction from one turns it from a measure of similarity to dissimilarity. The utilization of typicalities in the distance measure of Equation 23 is crucial to the performance of SPC, as it determines when structures are merged, deleted, and which structure are clustered together with DBSCAN. Thus, even though outputs of SPC are crisp, the algorithm relies heavily on concepts from possibility theory, especially those proposed in PCM.

It is also valuable to transform typicalities from Equation 22 into a logarithmic scale for the purpose of specifying parameter values in more intuitive ranges. To this end, we define the negative log typicality (NLT) as

$$NLT(x, \mu, \Sigma) = -\log u_m(x, \mu, \Sigma), \tag{24}$$

where, in this work, log refers to the natural logarithm. Like the negative log likelihood (NLL) in statistics, NLT takes on the range $[0, \infty)$ with higher values reflecting lower typicalities. For our experiments, we use an NLT threshold of 3 (see Table 3), corresponding to a typicality of around 0.05.

2. *Structure Footprints*

Each structure at time T is represented by its mean $\mu^{(T)}$ and a symmetric positive-definite (SPD) measure of spread $\Sigma^{(T)}$ that behaves like a covariance matrix. These measures are computed according to a damped window approach with decay factor $\gamma \geq 0$ that assigns exponentially decaying weight to older observations.

In general, damped window approaches apply an exponential coefficient such as $2^{-\gamma(T-t)}$ to the "importance" at the current time T of a point that arrived at time t. The parameter γ here serves as a damping factor that models how long it takes for points to be discounted. While, in theory, every point in an infinitely long stream would have some contribution in this model, this contribution very quickly becomes zero when working with fixed-precision numerical representations. For this reason, the damping factor γ is usually picked to be very small, *e.g.*, 10^{-3}.

In addition to tracking a mean and covariance for each structure, the structure footprint includes a measure of the average weight $w^{(T)}$ in this structure which determines how much typicality recent points in the stream have had in this structure, much like the idea of micro-cluster weights in DenStream and DBSTREAM. Since it is sometimes useful to apply a larger window to structure weights than for mean and covariance updates, the structure weight uses a separate decay factor $\beta \geq 0$ the behaves exactly as γ.

The footprint for a structure, consisting of mean $\mu^{(T)}$, spread $\Sigma^{(T)}$, and weight $w^{(T)}$ at time T, is computed according to

$$\mu^{(T)} = \frac{1}{\Gamma^{(T)}} \sum_{t=1}^{T} e^{-\gamma(T-t)} x^{(t)} \tag{25}$$

$$\Sigma^{(T)} = \frac{1}{\Gamma^{(T)}} \sum_{t=1}^{T} e^{-\gamma(T-t)} (x^{(t)} - \mu^{(t)})(x^{(t)} - \mu^{(t)})^T \tag{26}$$

$$w^{(T)} = \frac{1}{B^{(T)}} \sum_{t=1}^{T} e^{-\beta(T-t)} u_m(x^{(t)}, \mu^{(T)}, \Sigma^{(T)}) \tag{27}$$

$$\Gamma^{(T)} = \sum_{t=1}^{T} e^{-\gamma(T-t)} = \begin{cases} \frac{e^{\gamma} - e^{\gamma(1-T)}}{e^{\gamma}-1}, & \gamma > 0 \\ T, & \gamma = 0 \end{cases} \tag{28}$$

$$B^{(T)} = \sum_{t=1}^{T} e^{-\beta(T-t)} = \begin{cases} \frac{e^{\beta} - e^{\beta(1-T)}}{e^{\beta}-1}, & \beta > 0 \\ T, & \beta = 0 \end{cases} \tag{29}$$

where $x^{(t)}$ represents the t-th observation in the stream, u_m is as in Equation 22, and $\Gamma^{(T)}$ and $B^{(T)}$ are normalizing coefficients, differing only in their decay factor. We notice that when $\gamma = 0$, $\mu^{(T)}$ reduces to the mean of the first T elements of the stream and $\Gamma^{(T)} = T$. Thus, the damped window mean is a generalization of the usual arithmetic mean. Similarly, when $\gamma = 0$, the damped window covariance reduces to (biased) sample covariance.

We can merge two structures with footprints $(\mu_1^{(T_1)}, \Sigma_1^{(T_1)}, w^{(T_1)})$ and $(\mu_2^{(T_2)}, \Sigma_2^{(T_2)}, w^{(T_2)})$ quite easily with the following equations:

$$\mu^{(T)} = \frac{1}{\Gamma^{(T_1+T_2)}} \left(e^{-\gamma T_2} \Gamma^{(T_1)} \mu_1^{(T_1)} + \Gamma^{(T_2)} \mu_2^{(T_2)} \right) \tag{30}$$

$$\Sigma^{(T)} = \frac{1}{\Gamma^{(T_1+T_2)}} \left(e^{-\gamma T_2} \Gamma^{(T_1)} \Sigma_1^{(T_1)} + \Gamma^{(T_2)} \Sigma_2^{(T_2)} \right) \tag{31}$$

$$w^{(T)} = \frac{1}{B^{(T_1+T_2)}} \left(e^{-\beta T_2} B^{(T_1)} w_1^{(T_1)} + B^{(T_2)} w_2^{(T_2)} \right) \tag{32}$$

where $T = T_1 + T_2$. The multiplication of $\mu^{(T_1)}$ by $\Gamma^{(T_1)}$ in Equation 30 inverts the normalization done in the computation of $\mu^{(T_1)}$ and the multiplication by $e^{-\gamma T_2}$ effectively shifts the weight of each $x^{(t)}$ involved in computing $\mu^{(T_1)}$ back T_2 units of time. By the same logic, inverting the normalization done in computing $\mu_2^{(T_2)}$ is done by multiplication with $\Gamma^{(T_2)}$. Then, the two terms can be added together and re-normalized by $\Gamma^{(T)}$, which one will find is equivalent to Equation 25. The intuition behind the incremental updates for $\Sigma^{(T)}$ and $w^{(T)}$ is identical. Note that, while $\Gamma^{(T)}$ and $B^{(T)}$ can also be updated incrementally, they can more easily be computed in closed form as shown in Equations 28 and 29.

We also make use of the ability to update structure weight incrementally given a new point x according to

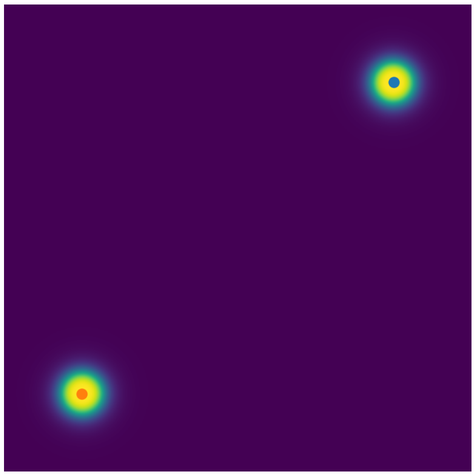
$$w^{(T+1)} = \frac{1}{B^{(T+1)}} \left(e^{-\beta} B^{(T)} w^{(T)} + u_m(x, \mu^{(T)}, \Sigma^{(T)}) \right). \tag{33}$$

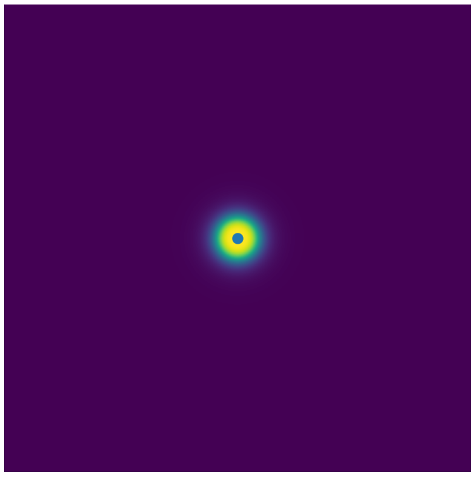
We observe Equation 33 to again follow the same general form as the other damped window updates.

All methods of computing structure footprints $(\mu^{(T)}, \Sigma^{(T)})$ are equivalent in theory, however for numerical stability, one may be concerned that repeated multiplication by $\Gamma^{(T-1)}$ and division by $\Gamma^{(T)}$ will lead to floating point errors that propagates indefinitely over time. If one was so inclined, they could instead maintain un-normalized estimates of $\mu^{(T)}$ and $\Sigma^{(T)}$ and only normalize by $\Gamma^{(T)}$ when needed. This is much akin to the summary statistics of the sum of data points and the sum of squared data points that are commonly used in incremental estimates of mean and covariance, but generalized to conform to the damped window model.

3. *Covariance Union*

The method of fusing the covariance of two structures presented in Equation 31 is only valid when we are considering two structures with the same mean. When the means of two structures s_1 and s_2 are different, we want a new covariance matrix that is large enough to encompass the region of feature space influenced by both constituent structures, which, in the event that the means of s_1 and s_2 are far apart, can be much larger than either of the individual covariances. Figure 13 demonstrates why this is the case.


(a) Two example structures with small covariances that we want to merge.


(b) Result of merging two structures using Equation 31.

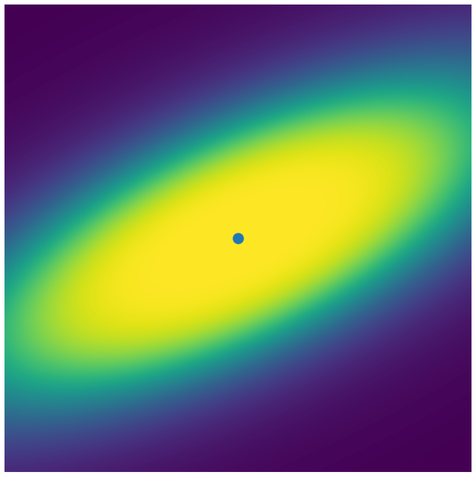

(c) Result of merging two structures using covariance union.

Figure 13: Illustration of why covariance union is needed to combined covariance matrices of two structures with unequal means.

The problem of fusing two or more unique mean and covariance pairs has been studied extensively in the filtering literature, specifically in the domain of multiple hypothesis tracking. In this work, we employ covariance union (CU) [57] for obtaining the fused covariance of two merged structures. It should be noted that CU was developed in the context of ensuring fused covariances are conservative estimators of the true state of the object(s) being tracked, but here, as we are not tracking anything per se, the method is used due simply to its observed effectiveness at producing high quality fused covariance estimates in our experiments.

Suppose again that we have two structures s_1 and s_2 with footprints $(\mu_1^{(T_1)}, \Sigma_1^{(T_1)}, w^{(T_1)})$ and $(\mu_2^{(T_2)}, \Sigma_2^{(T_2)}, w^{(T_2)})$ that we wish to fuse. The first step of CU is to determine a candidate mean $\mu^{(T)}$, where $T = T_1 + T_2$, of the fused state of the system, which we have established can be done with Equation 30. We then define two matrices U_1 and U_2 to be covariances Σ_1 and Σ_2 of s_1 and s_2 padded by the outer product of the difference between the structure mean and the candidate mean according to

$$U_1 = \Sigma_1 + \left(\mu^{(T)} - \mu^{(T_1)}\right)\left(\mu^{(T)} - \mu^{(T_1)}\right)^T \tag{34}$$

$$U_2 = \Sigma_2 + \left(\mu^{(T)} - \mu^{(T_2)}\right)\left(\mu^{(T)} - \mu^{(T_2)}\right)^T. \tag{35}$$

Then, let $U_2 = LL^T$ be the Cholesky decomposition of U_2, where L is lower triangular, and compute the eigendecomposition $L^{-T}U_1L^{-1} = Q\Lambda Q^T$, where the columns of Q are orthonormal eigenvectors and Λ is a diagonal matrix of eigenvalues of $L^{-T}U_1L^{-1}$. The fused covariance estimate $\Sigma^{(T)}$ is thus given by

$$\Sigma^{(T)} = LQ \max\{\Lambda, \mathbb{I}\}\, Q^T L^T \tag{36}$$

where max is taken elementwise between its arguments. Proof that the covariance matrix resulting from Equation 36 is conservative is provided in the original manuscript [57], though, to reiterate, our inputs to Equation 36 do not, in general, satisfy the assumptions required for that result to hold.

4. *Algorithm Description*

There is no initialization for SPC, the algorithm starts directly processing points from the stream. There is, however, a burn-in phase during the first n points, where n is the maximum number of structures parameter, during which no structure merging occurs. Each incoming point x_i, $i \leq n$, during this phase is modeled by its own structure with mean $\mu_i = x$ and identity covariance (*i.e.*, Euclidean distance used in typicality computation). One can still cluster the structures using DBSCAN as specified in Algorithm 7, but examining the structures directly would not yet yield any useful information about the stream.

Even after n structures have been created, the first step of processing a new point x_j, $j > n$, from the stream is to create a new structure with mean x_j and identity covariance. At this point, there are $n + 1$ structures, so one must be removed. This can happen in one of two ways. If the weight of this structure is too small, the structure will either be simply deleted or merged into the best fit existing structure,

Algorithm 7 Single-pass Possibilistic Clustering

```
// Let $\mu_i$ be the mean of the $i$-th structure
// Let $\Sigma_i$ be the covariance of the $i$-th structure
// Let $w_i$ be the weight of the $i$-th structure
// Let $T_i$ be the number of timesteps elapsed since the $i$-th structure was created
// Let $S = \{(\mu_i, \Sigma_i, w_i, T_i)\}$ be the set of $n$ structures, initially empty
// Let $N$ be the maximum number of structures to track

procedure UpdateSPC($x$, $m$, $\gamma$, $\beta$)

    // Create a new structure to accommodate this point.
    $S \leftarrow S \cup \{(x, \mathbb{I}, 1, 1)\}$

    // If there are too many structures, prune and merge.
    if $|S| > N$ then

        for $i = 1, 2, \ldots, |S|$ do
            Update $w_i$ using Equation 33.
            if $w_i < w_{min}$ then
                Let $\hat{j} = \underset{j=1,\ldots,|S|,\ j\neq i}{argmin} \ NLT(\mu_i, \mu_j \Sigma_j)$.
                if $NLT(\mu_i, \mu_{\hat{j}} \Sigma_{\hat{j}}) < NLT_{max}$ then
                    Call Merge($\hat{j}$, $i$).
                Delete $s_i$ from $S$.

        // Merge two closest structures using Equation 23.
        Let $(\hat{i}, \hat{j}) = \underset{i,j\in 1,\ldots,|S|,\ i\neq j}{argmin} \ D(i, j)$.
        Call Merge($\hat{i}$, $\hat{j}$).

procedure Merge($i$, $j$)
    Let $s_k$ be a new structure with:
        $\mu_k$ from $\mu_i$ and $\mu_j$ using Equation 30
        $\Sigma_k$ from $\Sigma_i$ and $\Sigma_j$ using Equation 36
        $w_k$ from $w_i$ and $w_j$ using Equation 33
        $T_k = T_i + T_j$
    Replace $s_i$ and $s_j$ in $S$ with $s_k$

procedure GetClustering($\varepsilon$, $min\_pts$)
    Run DBSCAN($\varepsilon$, $min\_pts$) on $S$ using the distance function in Equation 23.
    return Cluster labels from DBSCAN.
```

depending on whether there exists another structure in which the NLT of the to-be-removed cluster is high enough. If there are too many structures and all structures have too much weight to be deleted, the two structures that are deemed most similar according to Equation 23 are merged.

D. Experiments

In this section, we evaluate SPC both qualitatively on synthetic datasets and quantitatively on real datasets against other state-of-the-art streaming clustering algorithms (CluStream [16], DenStream [29], D-Stream [33], DBSTREAM [50], and StreamSoNG [86].). All parameters used in all experiments in this manuscript are documented in Table 3. For plotting decision regions, we use the nearest neighbors algorithm and the distance function

$$d(s_i, x_j) = 1 - \left(\frac{1}{1 + \left[(x_j - \mu_i)^T \Sigma_i^{-1} (x_j - \mu_i) \right]^{\frac{1}{m-1}}} \right)^2 \qquad (37)$$

where s_i is the i-th structure tracked by SPC with mean μ_i and covariance Σ_i, and x_j is an arbitrary point.

1. *Synthetic Dataset*

We first evaluate SPC on a two-dimensional synthetic dataset with seven clusters, originally published by Gionis et al. [46]. Key challenges

Figure 14 - Aggregation Dataset							
Parameter	n	γ	β	m	ε	w_{min}	NLT_{max}
Value	30	0.0	0.0	1.5	0.95	0.01	3

Figure 15 - Sine Dataset							
Parameter	n	γ	β	m	ε	w_{min}	NLT_{max}
Value	30	0.1	0.05	1.4	0.95	0.01	3

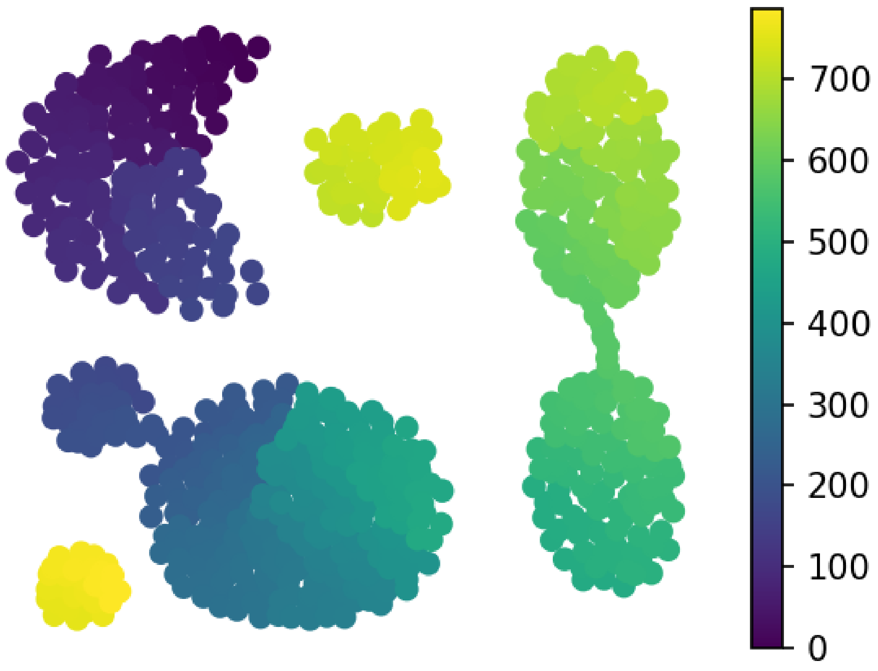
Figure 16 - High Dimensionality Gaussians							
Parameter	n	γ	β	m	ε	w_{min}	NLT_{max}
Value	50	0.0	0.00	1.5	0.95	0.01	3

Figure 17 - Overlapping Dataset							
Parameter	n	γ	β	m	ε	w_{min}	NLT_{max}
Value	30	0.0	0.0	1.5	0.95	0.01	3

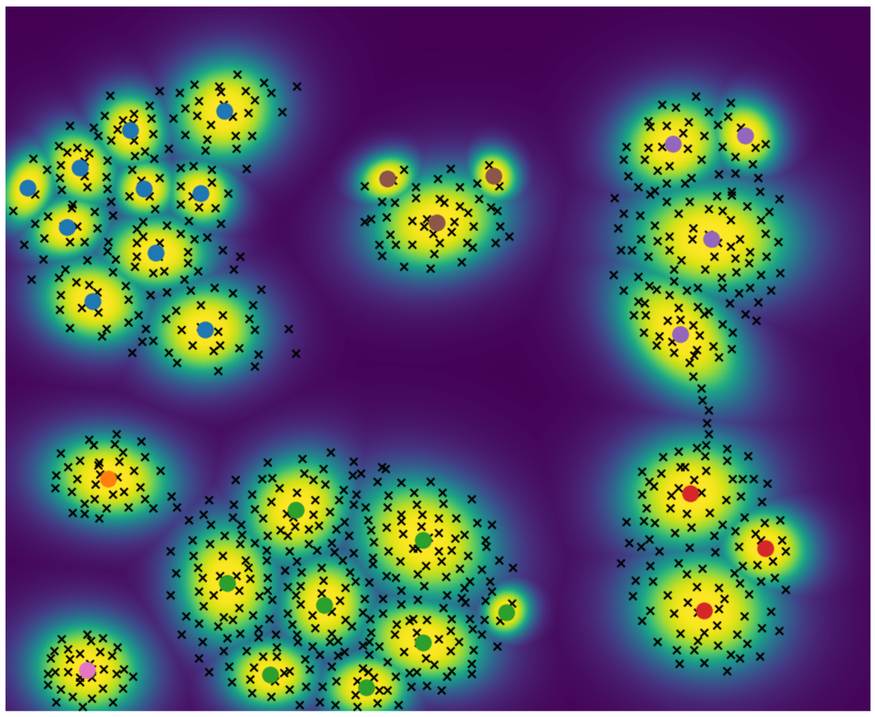
Table 3: Parameter values used for SPC experiments in this manuscript.

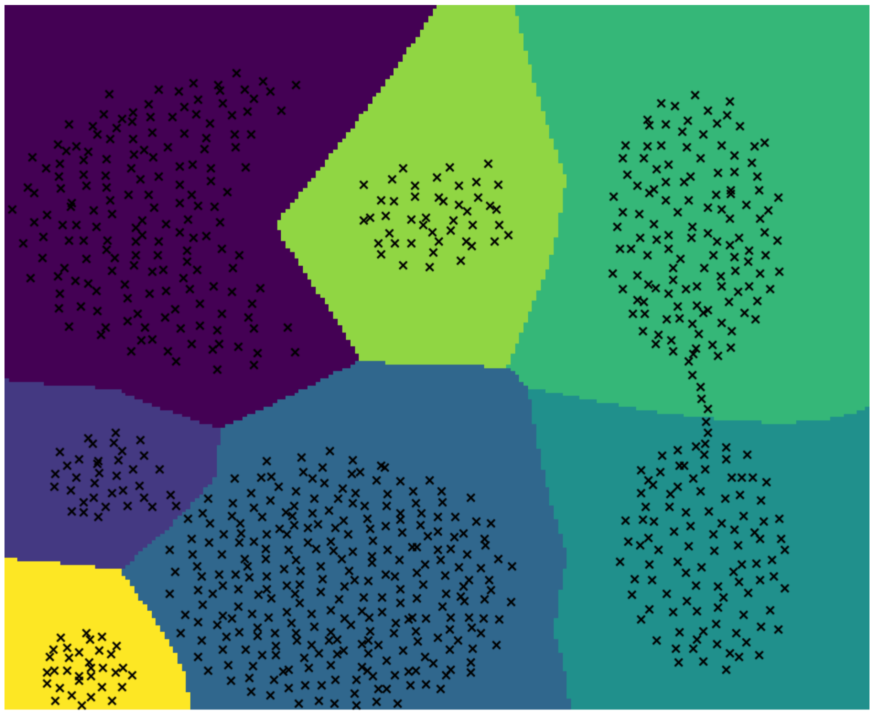
with this dataset include the non-Gaussian cluster in the upper left and the two sets of two clusters that bleed into each other. SPC handled both challenges well and produced a very good clustering. The decision region, shown in Figure 14c, leaves nothing to be desired and arguably looks as if it was drawn by a human.

For this dataset, the forgetting factors γ and β were both set to zero so the algorithm weights the whole dataset equally, rather than placing more weight on recently observed points. From the final position of the structures tracked by SPC (Figure 14b), it is clear that the entire dataset was effectively summarized. Despite being computed over a single pass of the dataset, the final clustering is indistinguishable from iterative methods that are allowed to make as many passes over the dataset as required.

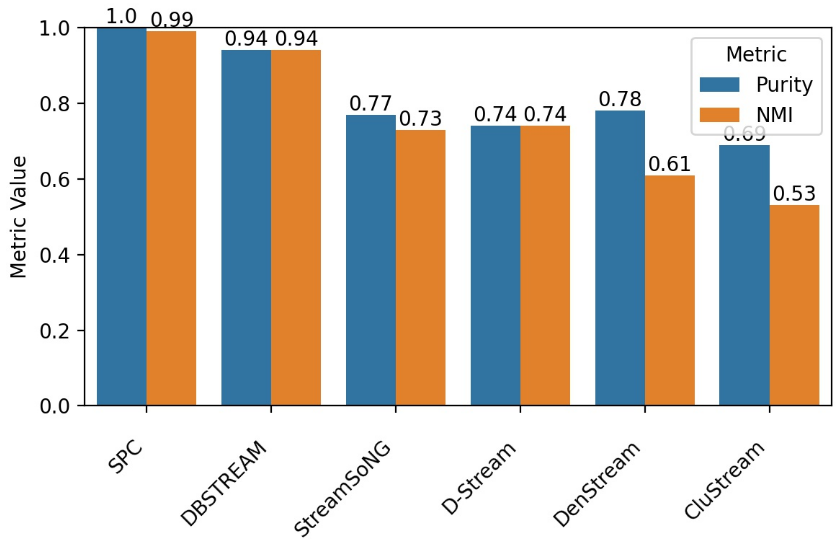


(a) Dataset of seven clusters in two dimensions, colored by arrival time from blue (upper left cluster) to yellow (small lower left cluster).


(b) Final configuration of SPC with 30 structures. The final mean $\mu^{(T)}$ of each structure is plotted as a filled in circle along with the typicality induced by its covariance matrix. The color of each circle represents the cluster label assigned with DBSCAN, which we observe to be consistent with the true labels.



(c) Decision region induced by running the nearest neighbor algorithm on a fine grid over the plot area.



(d) Comparison of streaming clustering algorithms on the synthetic dataset of Gionis et al. [46] with respect to cluster purity and normalized mutual information (NMI).

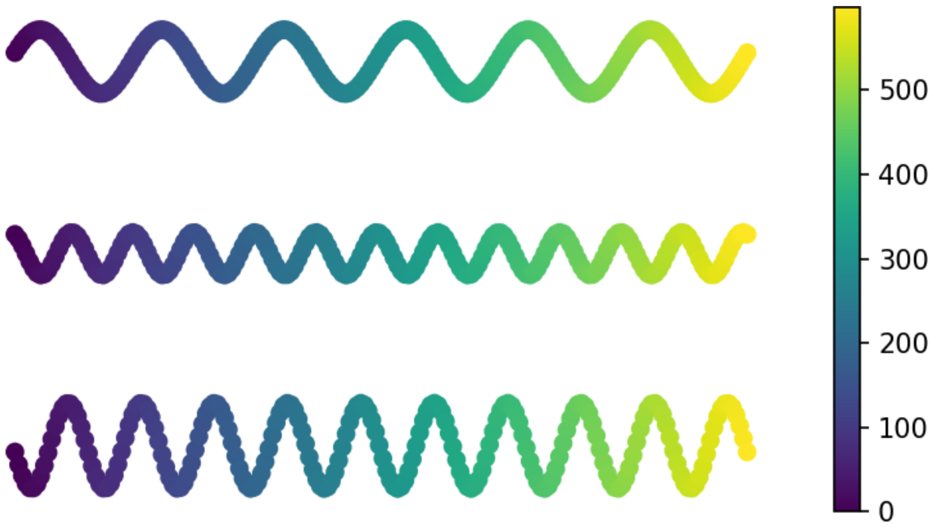
Figure 14: SPC performance on the synthetic clustering dataset of Gionis et al. [46].

As expected from its high qualitative performance, SPC achieve a near-optimal value in both purity and NMI on this dataset, shown in Figure 14d. DBSTREAM also performed very well on this dataset, followed by StreamSoNG, D-Stream, and DenStream.

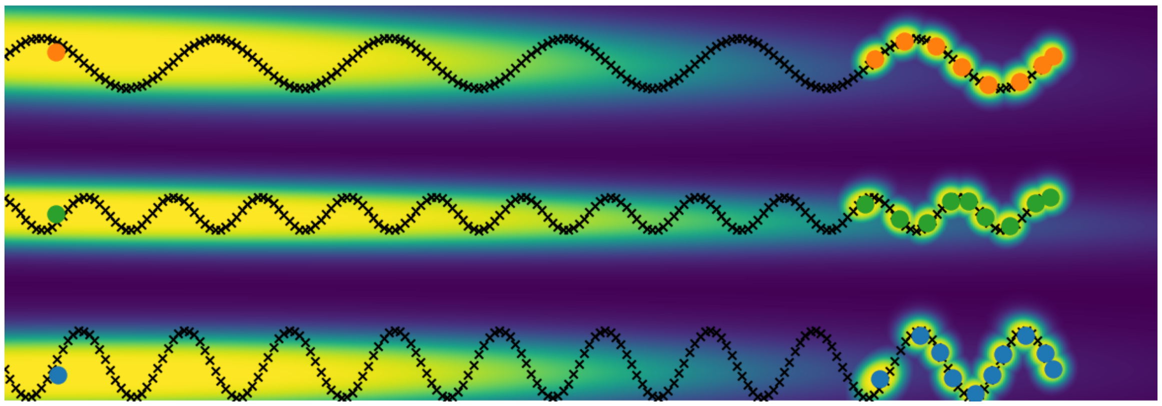
2. *Nonstationary Dataset*

Datasets that have nonstationary clusters that continually evolve over time inevitably require clustering algorithms to either forget older points, create an unbounded number of clusters, or increase the number of points modeled by a single cluster. SPC chooses the latter, as the first option is undesirable and the second violates the "finite storage to model infinite data" philosophy of streaming clustering.

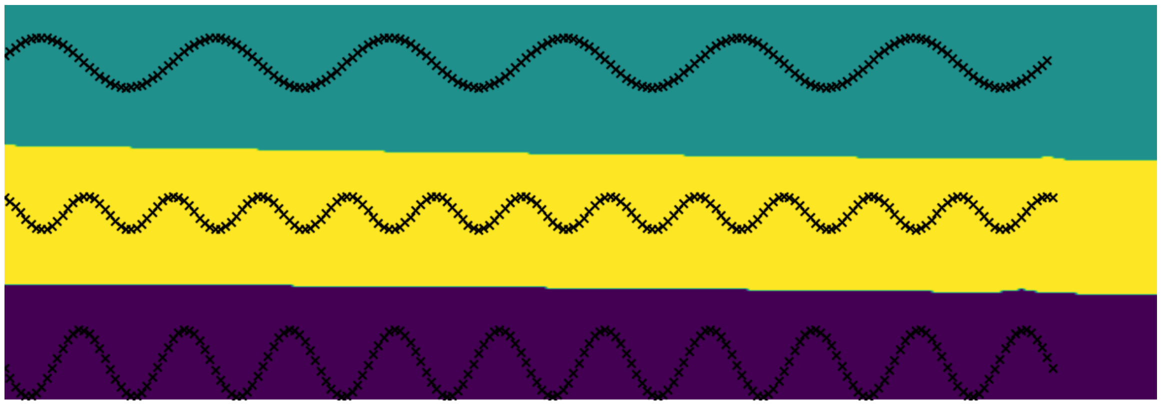
In Figure 15, we demonstrate a case where running SPC with a high forgetting factor is valuable. This dataset has three highly non-Gaussian clusters, each modeled by a sine wave with different frequency and amplitude. SPC was given a maximum of 30 structures to use in modeling this dataset, and allocated most of them to finely detail the most recently encountered points, appealing to the principle of spatiotemporal locality (new points in a stream are lie in close proximity to recently seen points). One structure with a very large region of influence is used to model the early points in the stream from each sine wave.



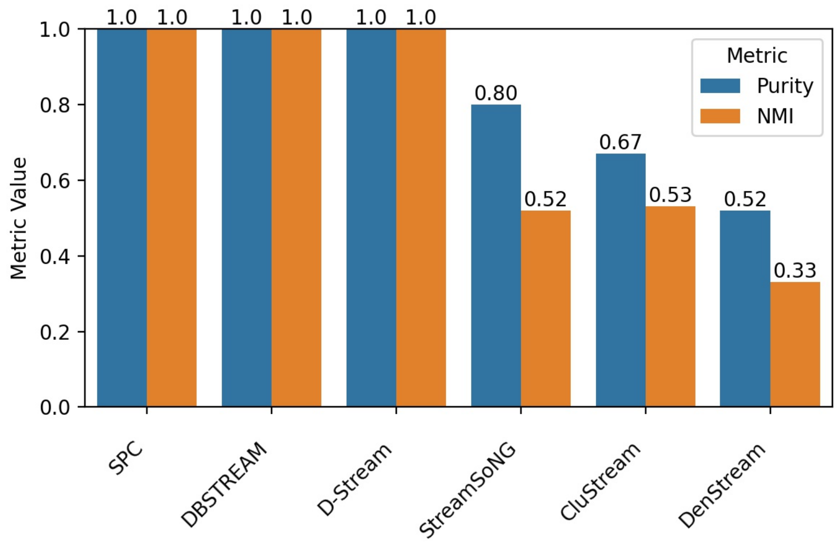
(a) Dataset of three sine waves arriving simultaneously and moving unilaterally toward $+\infty$ on the x-axis, colored by arrival time from blue to yellow. Points in the stream are sampled in a round-robin pattern from each class.



(b) Final configuration of SPC on the sine wave dataset with 30 structures. The final mean $\mu^{(T)}$ of each structure is plotted as a filled in circle along with the typicality induced by its covariance matrix. The color of each circle represents the cluster label assigned with DBSCAN, which we observe to be consistent with the true labels.



(c) Decision region induced by running the nearest neighbors algorithm on a fine grid over the plot area.



(d) Comparison of streaming clustering algorithms on nonstationary sine dataset with respect to cluster purity and normalized mutual information (NMI).

Figure 15: Nonstationary synthetic dataset of three sine waves illustrating the utility of SPC's forgetting factor in modeling newer data with finer detail. Old points are still modeled, but with less granularity. Here, SPC was run with a high forgetting factor of $\gamma = 0.1$.

We compare SPC against several other streaming clustering algorithms in Figure 14d. Clearly, the nonstationary, non-Gaussian nature of this dataset was problematic for some algorithms. SPC, D-Stream, and DBSTREAM achieved a perfect clustering, whereas CluStream, DenStream, and StreamSoNG struggled to either separate the three clusters or remember the whole stream.

When using DBSCAN to assemble structures into clusters, we use a specialized distance metric that leverages the covariance matrix of each structure. We see that the very large structures modeling older points in the stream are given the same cluster label as the small structures modeling newer points in the stream, but that no two structures from different sine waves are given the same cluster label. This is reflected in the perfect purity and NMI scores of Figure 15d.

3. *High Dimensionality Dataset*

Lastly, we evaluate SPC on a very high dimensional dataset consisting of 1024 points from 16 well-separated Gaussians in 1024-dimensional space [41]. The results of running SPC and other comparison algorithms on this dataset are shown in Figure 16. We see that SPC, DBSTREAM, and StreamSoNG produce good sets of clusters on this dataset, with DenStream and CluStream attaining slightly lower performance. We were not able to find a parameter set for D-Stream that produced a non-trivial clustering result. It is well known that D-Stream does not scale to high dimensionality due to its need to produce a grid over the entire feature space, which, for 1024-dimensional data, is highly intractable.

Notable drawbacks of covariance-based methods, like SPC, on high dimensional data are the difficulty in capturing the correlations between all pairs of variables and the quadratic (in dimensionality) storage requirement to retain full covariance matrices. When covariance is not constrained in some way (*i.e.*, spherical, diagonal, tied) to favor sparsity, there are $O(d^2)$ pairs of correlations in d-dimensional space, which can easily and overwhelmingly exceed the number of data points when d is large. The 1024-dimensional Gaussian data, for example, requires SPC to estimate over 1 million entries in each structure's covariance matrix, despite the stream only containing 1024 points.

In general, a stream in d dimensions would need to have $O(n(1+d))$ elements for the memory consumption of SPC to be less than that of storing the entire stream. With $n = 50$ structures in 1024-dimensional space, this would correspond to the stream containing on the order of 50,000 points. In cases like this dataset, where it it doesn't hold that $n \gg d$, it is hard to motivate using SPC, or essentially any streaming clustering algorithm, over static clustering algorithms that simply retain the stream and iterate over it repeatedly.

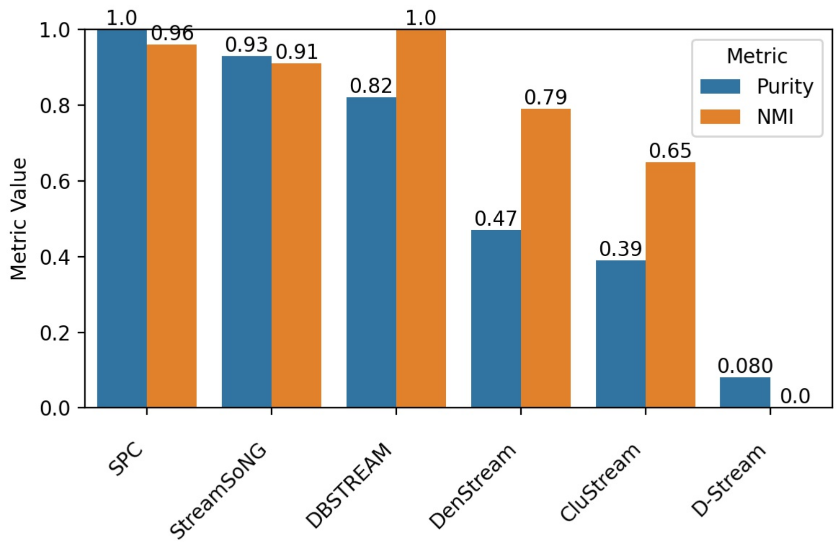
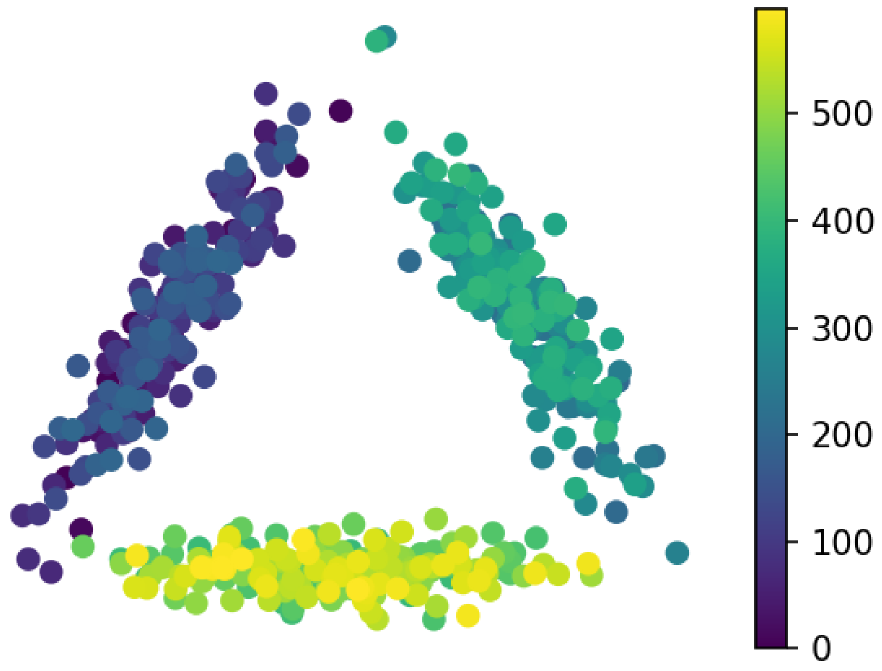


Figure 16: Comparison of streaming clustering algorithms on a high dimensional dataset [41] of 1024 points from 16 Gaussian clusters in $\mathbb{R}^{1024}$ with respect to cluster purity and normalized mutual information (NMI).

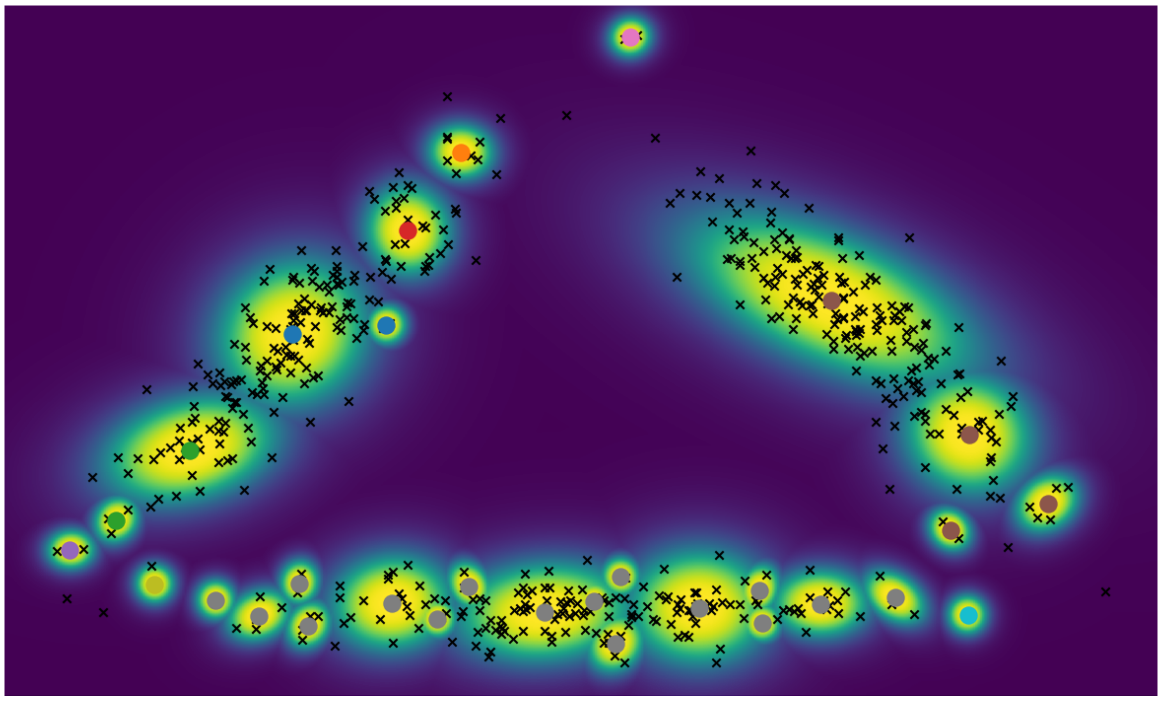
The purpose of this experiment is to show that, despite SPC's inherent incompatibility with high dimensional data, it can still perform well when clusters are sufficiently compact and well-separated. As future work, we wish to extend SPC to maintain sparse, constrained estimates of the covariance in each structure so as to make it more practical in high dimensional use cases. Generally, we would recommend employing some form of dimensionality reduction when applying SPC to such high dimensional streams.

4. *Overlapping Dataset*

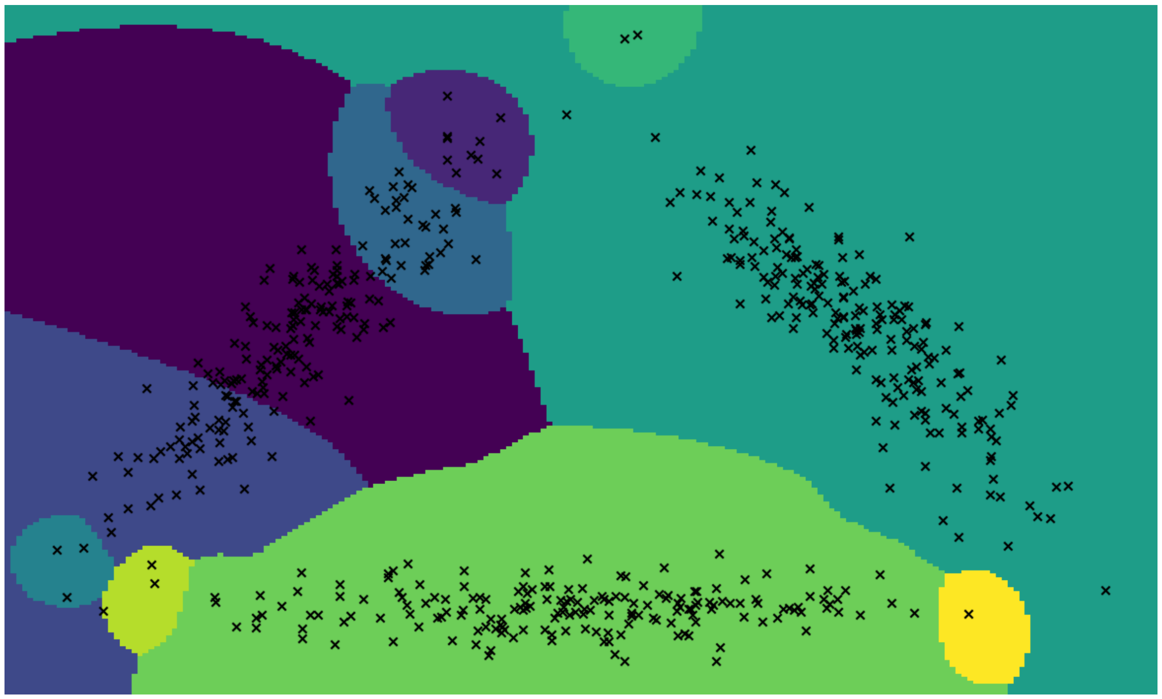
Datasets with overlapping clusters present a challenge for most streaming clustering algorithms, SPC included. To evaluate SPC and related



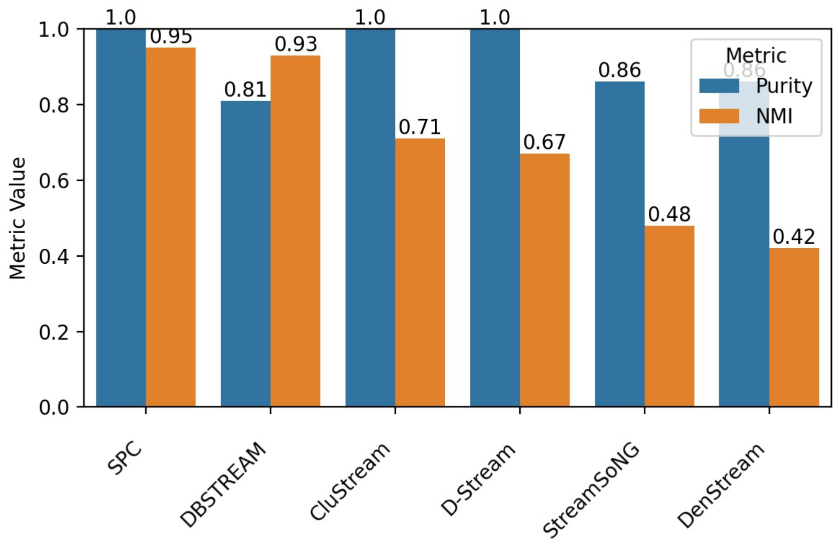
(a) Dataset of three clusters in two dimensions, colored by arrival time. Clusters arrive sequentially, but within each cluster, points arrive in a random order.



(b) Final configuration of SPC with 30 structures. The final mean $\mu^{(T)}$ of each structure is plotted as a filled in circle along with the typicality induced by its covariance matrix. The color of each circle represents the cluster label assigned with DBSCAN, which we observe to be mostly consistent with the true labels.



(c) Decision region induced by running the nearest neighbor algorithm on a fine grid over the plot area.



(d) Comparison of streaming clustering algorithms on an overlapping cluster dataset with respect to cluster purity and normalized mutual information (NMI).

Figure 17: SPC performance on an overlapping cluster dataset.

algorithms on overlapping data, we created a synthetic dataset of three highly correlated Gaussians in the shape of a triangle. This dataset, shown in Figure 17, tests the ability of algorithms to separate clusters that are connected by low-density regions of points. The most common pitfall when running streaming clustering algorithms on this dataset is grouping the entire dataset into a single cluster, making many algorithms sensitive to parameter choice.

As observed in Table 3, we use the same SPC parameters for this dataset as we do for the dataset in Figure 14, though we did have to adjust the parameters for the other algorithms. Clearly, this dataset was challenging across the board, evidenced by universally lower metrics in Figure 17d. SPC still had the highest performance[5], but in looking at the decision region shown in Figure 17c, we see that the leftmost cluster was split into three sizeable clusters and that there are several remaining structures placed on outlier points.

This dataset is also valuable in illustrating the complex decision regions that can be induced by the typicality function in Equation 22 (Figure 17c). As we get far away all of the points in the dataset, we see that the nearest neighbor structure is actually the structure with largest covariance rather than the structure whose mean is closest in

[5]SPC achieved a perfect purity of 1.0 in Figure 17d despite clearly having a cluster partition that is not identical to the ground truth partition because the clusters in SPC are all homogeneous with respect to ground truth labels, which is why we also evaluate based on NMI.

terms of Euclidean distance, although typicality is extremely low in all structures.

E. Conclusion

We have proposed a single-pass possibilistic clustering algorithm, coined SPC, that demonstrates strong performance when applied to a wide array of clustering datasets. Key contributions of SPC include the use Mahalanobis distance to compute typicality, merging of structures over a damped window, and the transfer of covariance union from the domain of multiple hypothesis tracking to that of streaming clustering.

A key design choice in SPC is its ease of application to new datasets; the only data-dependent parameter is the fuzzifier m, which, in most cases, can remain around $m = 1.5$. The decay factors γ and β allow the user to balance long term memory of the stream with prioritization of recently seen points, enabling SPC to function effectively on both stationary streams ($\gamma = \beta = 0$) and nonstationary streams ($\gamma, \beta > 0$), all while maintaining a constant memory footprint.

SPC either exceeds the performance of, or is competitive with, five other streaming clustering algorithms from the literature. Qualitatively speaking, SPC produces extremely high quality decision regions on two dimensional data, and, quantitatively speaking, achieves high purity and NMI on a compact and well-separated dataset in 1024 dimensions.

V. Simulated Texture Dataset

Abstract — With the recent announcement of Unreal Engine 5 (UE5) (see Figure 18), machine learning research communities have been abuzz with talk of creating realistic datasets in simulation. The image in Figure 18, undoubtedly created by a team of industry experts, shows the absolute pinnacle of what simulation can achieve. This example render is so sophisticated, we are confident that it could be passed off as an actual photograph. Although we, as computer science researchers, can't reasonably expect (yet) to create full datasets to such a high degree of realism, we show in this work that we can create effective datasets for use in machine learning applications, specifically for use in streaming data analysis (SDA) problems. We show that the created texture dataset readily lends itself to SDA, evidenced by the performance of several SDA algorithms from the literature, but still presents a significant challenge.

Image source: https://www.unrealengine.com/en-US/unreal-engine-5

Figure 18: Demonstration of photorealism in Unreal Engine 5.

A. Literature Review

The idea of using simulated datasets for computer vision tasks is still very new, and the state of the art is rapidly evolving as we discover new ways to construct and leverage these datasets. The barrier to entry is still relatively high, there are no standard simulated datasets for benchmarking, and it takes substantial effort to produce usable datasets for machine learning applications. In this section, we briefly survey two relevant works that leverage simulated data.

Simulation has been used recently in the domain of explosive hazard detection to augment the training set when real ground truth is sparse. Alvey et al. show that a large dataset of simulated imagery modeled

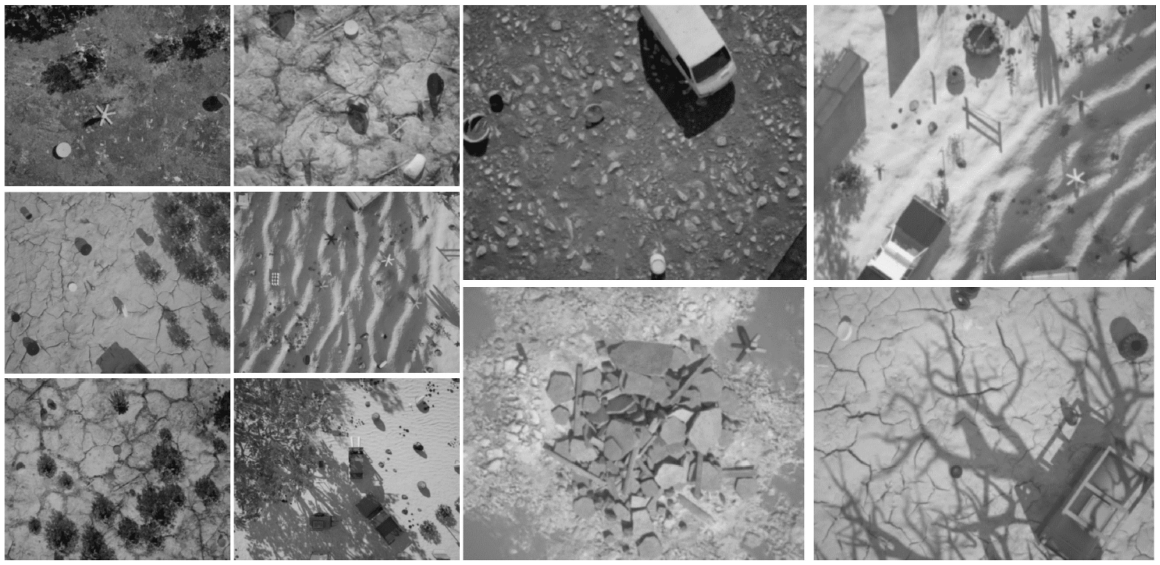


Figure 19: Simulated scenes for explosive hazard detection produced by Alvey et al. [17].

in Unreal Engine can outperform a small dataset of real imagery, and provided evidence that training on the union of real and simulated data can yield higher accuracy than either by itself [17]. A few example scenes from their work are shown in Figure 19 (taken from Figure 4 in their paper).

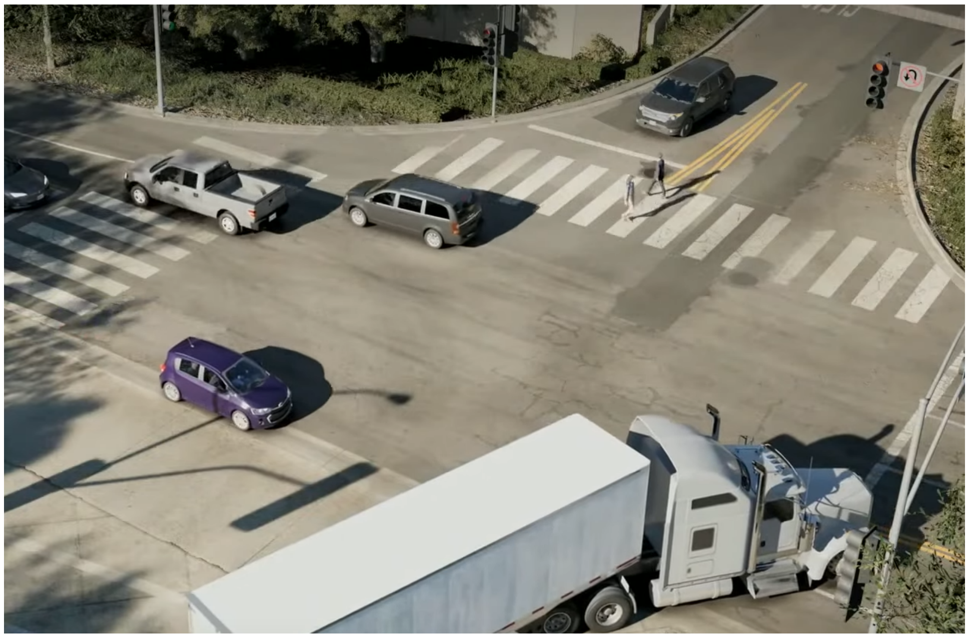
The state of the art in simulation, however, is nearly a perfect representation of reality. Tesla, Inc., armed with millions of dollars in computing resources and thousands of full-time employees, has produced an enormous simulated dataset for the purposes of training the neural networks behind their self-driving cars. A few screenshots from Tesla's 2021 AI Day presentation [54] are provided in Figure 20 to demonstrate the quality of their simulated dataset. Unfortunately, neither their methods nor implementation appear to be public, making reproducibility impossible.

B. Materials

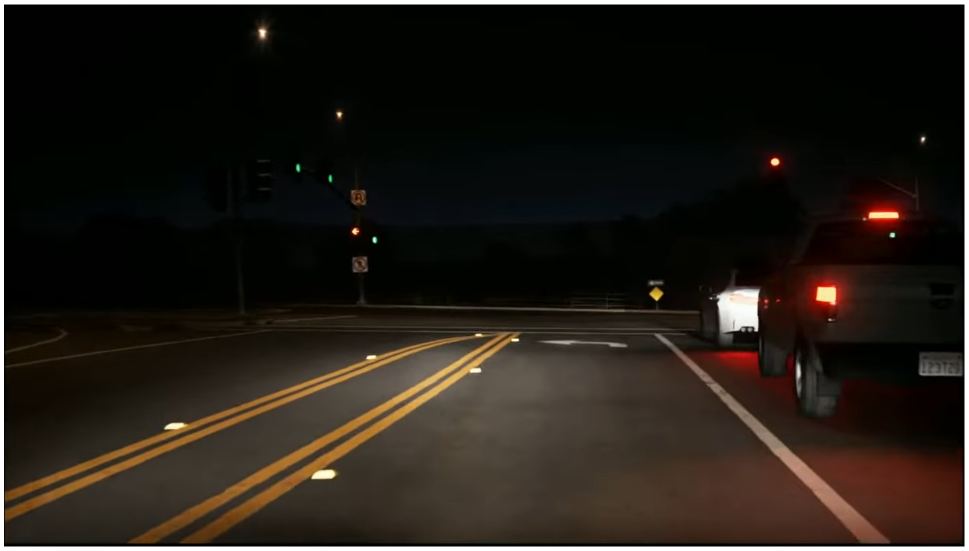
The first step to creating environments in Unreal Engine is to acquire textures for each type of terrain we want to create. In the past, textures were created by artists in graphic design software, however technology now allows for extremely high resolution 3D scanning of real objects for use in game engines. Most of the textures we use in this work are from the Megascans database by Quixel [72], which are freely available for use in Unreal Engine.

Next, a material must be made for each environment. Textures from the Megascans database generally have albedo (base color), roughness, and normal maps associated with them. Albedo provides the color, independent of lighting, at each pixel of the texture and the roughness and normal maps are both used for the engine to compute lighting and shadows. Thanks to the roughness and normal maps, two dimensional textures can be given the illusion of depth in simulation, which is a strong argument for why rendering the textures through a game engine like UE5 is desireable over simply writing a script to sequence these textures together.

As these textures are much smaller than the landscape on which they are painted, texture repetition is a significant issue hampering the



(a) Daytime scene, retrieved from `https://youtu.be/j0z4FweCy4M?t=5736`



(b) Nighttime scene, retrieved from `https://youtu.be/j0z4FweCy4M?t=5914`

Figure 20: Selection from Tesla's 2021 AI Day demonstration, found at `https://youtu.be/j0z4FweCy4M`.

Figure 21: Example of texture repetition, a behavior we wish to suppress.

realism of the environment and potentially opening an avenue for machine learning algorithms to memorize the training data rather than generalizing. An example is shown in Figure 21. To reduce texture repetition, two common techniques in video game level design are employed: macrovariation, that is, blending a texture with a scaled version of itself, and blending multiple unique texture samples of the same environment type. Employing these techniques has led to realistic looking pure textures, which we call "primitives", shown in Figure 22. The material diagram in Unreal Engine that produced the sand ripple example in Figure 22 is shown in Figure 23, with other environment types created in a similar way.

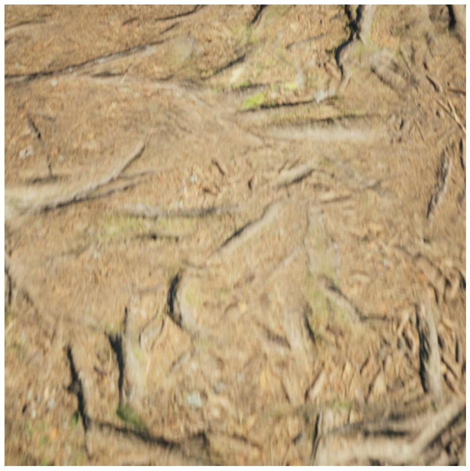
To aid with object detection tasks, we also randomly place a number of freely available 3D objects found online [1–14] in the scene, shown in

(a) Flat sand texture

(b) Sand ripple texture

(c) Leaves texture

(d) Plant texture


(e) Roots texture

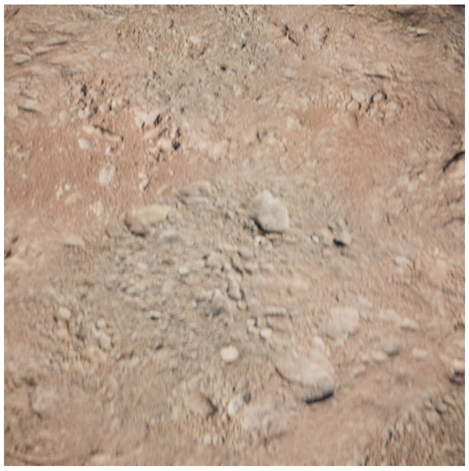

(f) Stone texture

Figure 22: Each pure texture, or “primitive”, used in this dataset, rendered as 720×720 image chips. The dataset contains 6,297 images of each primitive, totaling 37,782 pure texture images.

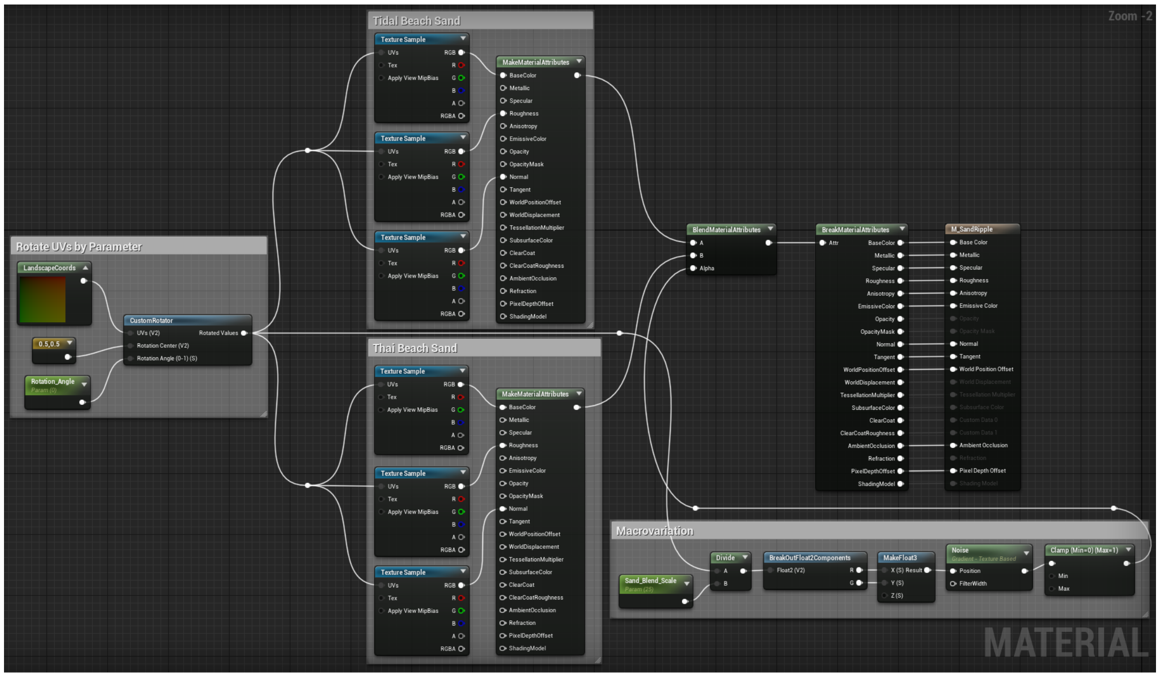


Figure 23: Material diagram of sand ripple texture in Unreal Engine.

Figure 24. The method we use to place these objects is quite simple, shown in Algorithm 8.

C. Ground Truth

As previously mentioned, the ability to produce pixel-perfect ground truth in this simulated data is extremely lucrative. For objects, we accomplish this using postprocess materials and custom depth stencils. Custom depth stenciling is an engine feature that allows a user-specified integer to be associated with objects in the scene, which can then be queried in the postprocess. By assigning class labels to the custom stencil of objects we wish to see in the segmentation view, we can overwrite the color of pixels on the object based on their class label. This process

Algorithm 8 Target placement in UE5

```
procedure PlaceTargets(object_array, $n$)

    for $i = 1, 2, \ldots, n$ do
        Choose a random position $(x, y)$ on the landscape
        Cast a ray from $(x, y, z_{max})$ to $(x, y, z_{min})$ to find the height $z$ of the
            landscape at $(x, y)$
        Choose a random object from object_array
        Apply a random rotation to object
        Instantiate the object at $(x, y, z)$
        Assign a custom depth stencil value to the object for use in
            producing segmentation labels

    return
```

is shown in Figure 25 and a before/after example is shown in Figure 26. In Figure 26, note that each target type has a different grayscale intensity, making some targets difficult to see in the segmentation as their label is represented by an intensity near, but not equal to zero.

For the sake of providing as much information about the dataset as possible, we can also export ground truth lighting information. Knowing the amount of lighting applied at each pixel can be useful for a number of applications, but currently, we have only used it to know where shadows are. Lighting can be computed by dividing scene color by albedo, which is easily done in a postprocess material as demonstrated in Figure 27.

Figure 24: Objects placed in the scene are selected from these models found online under Creative Commons licenses [1–14].

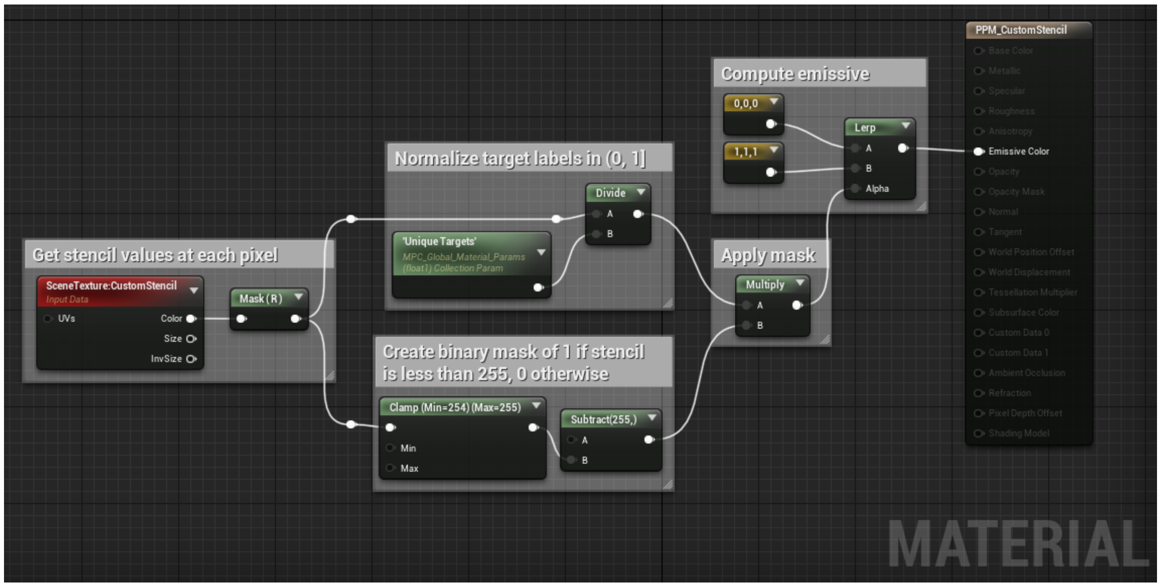
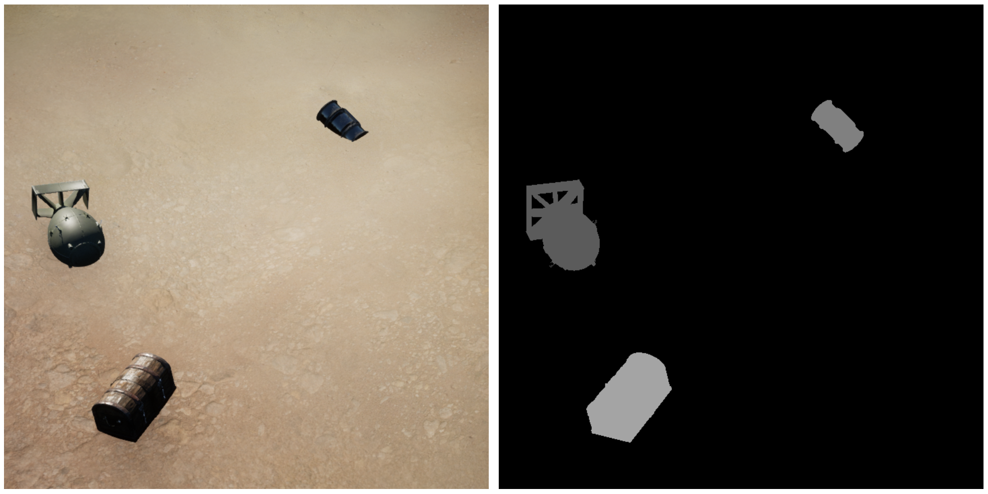


Figure 25: Postprocess material diagram demonstrating how to color pixels on screen based on object stencil values.



(a) Scene (b) Segmentation

Figure 26: Result of applying the custom stencil postprocess material to scene. Each class of target is mapped to a unique grayscale pixel intensity and the background is mapped to a grayscale value of zero.

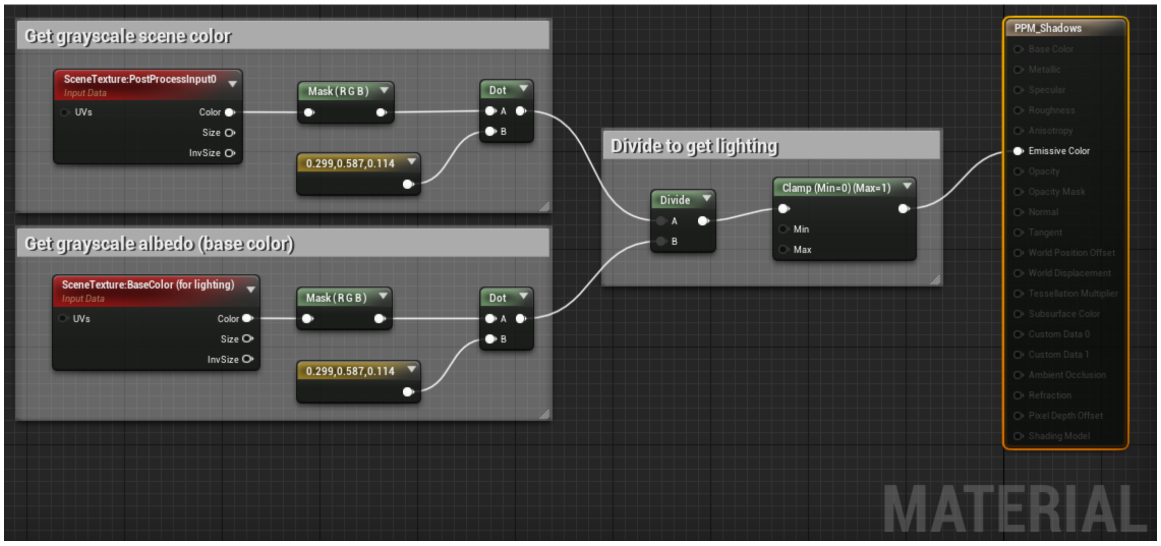


Figure 27: Postprocess material to extract lighting information at each pixel.

D. Data Collection

Throughout this research, we have used two approaches for collecting imagery from the simulated texture dataset. The first uses a drone simulation software suite developed by Microsoft called AirSim [77], available as a plugin for Unreal Engine 4. AirSim provides drone classes with complex functionality like physics simulation, noise, pathing, and accepting commands from a physical controller. For our purposes, we use AirSim in computer vision mode, meaning drones essentially amount to a scriptable floating camera.

AirSim drones can be controlled from Python scripts that communicate with Unreal Engine through the remote procedure call (RPC) protocol. To collect the dataset, we wrote a Python script that moves the drone along a series of strips across the landscape, illustrated in Figure 28. The drone makes nine strips of length 400 meters with 50 meters between each strip.

We have modified the AirSim drone to capture five types of imagery from five co-located cameras. First, the original scene is captured as seen in Unreal Engine. Then, the scene is captured by a second camera that has all objects and foliage hidden, giving us a view of only the terrain. The remaining three cameras have postprocess materials applied to capture various scene segmentations: one capturing target

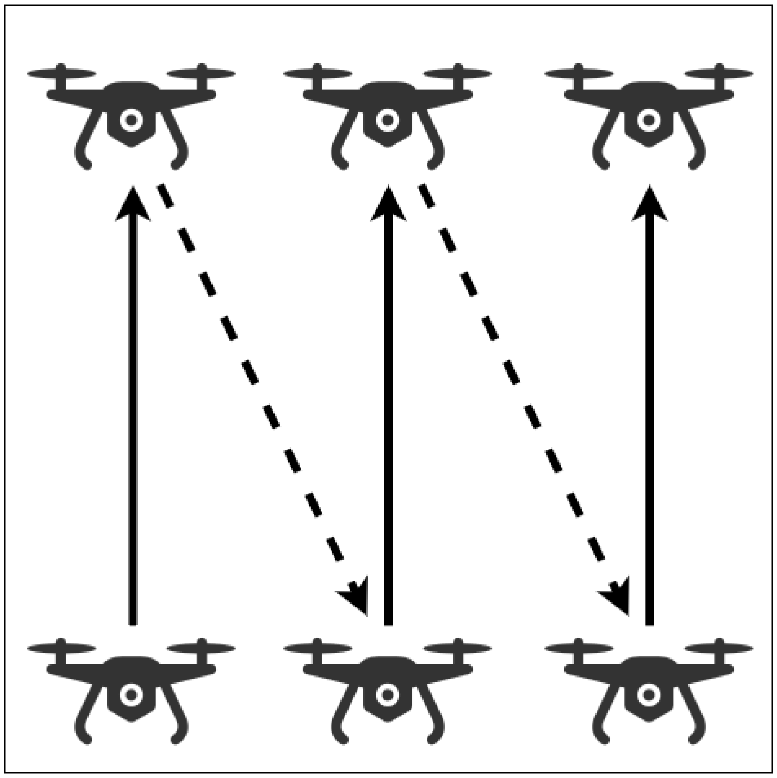


Figure 28: Scripted pattern of the drone over a top-down view of the landscape. Solid lines indicate smooth movement along a path, dotted lines indicate instantaneous movement between two points.

segmentation, one capturing the texture blend weights at each pixel, and the last computing the lighting information.

As we have migrated to Unreal Engine 5, we removed our dependency on AirSim in favor of a native waypoint-based approach to drone navigation, primarily because, as of writing, AirSim is not UE5 compatible. Our "drone" in this case is a camera whose position is updated each game tick by the level blueprint. For brevity, we do not include all of the code involved with moving the camera, but the bulk of the logic in the level blueprint is shown in Figure 29.

There are pros and cons to both data collection methods. Perhaps the most important factor in choosing AirSim or our blueprint-based approach is the engine version the user wishes to use. As of writing, AirSim does not appear to have made public any plans to upgrade to UE5, which means that using AirSim will lock the user into UE4 for the foreseeable future. AirSim is also much slower to capture imagery than the blueprint-based approach, which uses UE5's movie render queue. While this may not be a huge factor when exporting a one-and-done dataset, the difference in speed is drastic; AirSim captured at around 2 frames per second and the movie render queue is faster than real-time, upward of 30 frames per second.

On the other hand, AirSim is far more feature rich than what we wrote in blueprint. With AirSim, one can simulate the effect of wind

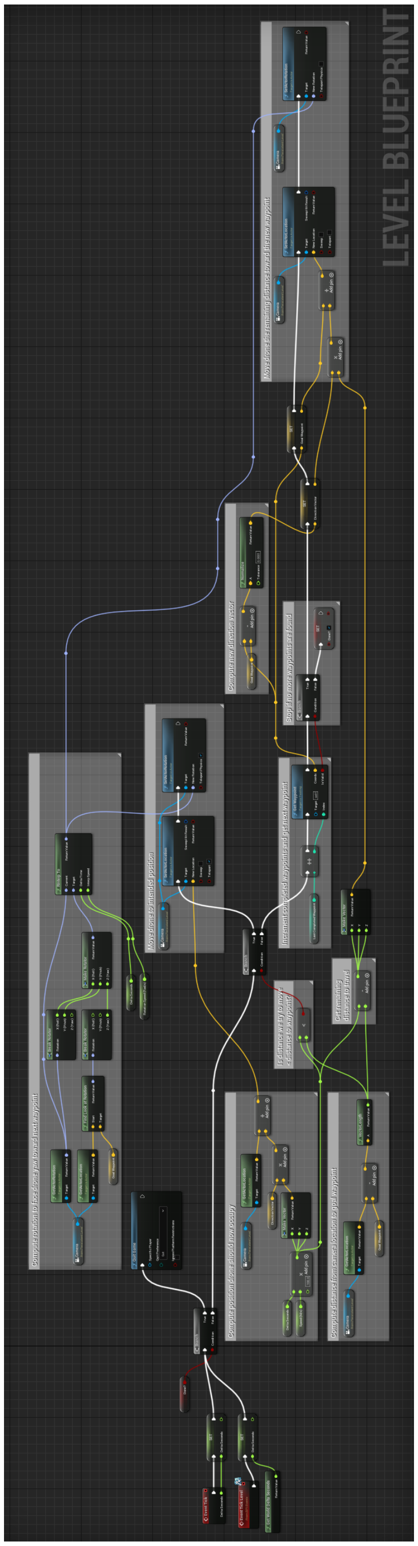


Figure 29: Blueprint for moving a camera between waypoints.

and turbulence on the drone, sensor noise, have multiple sensing modalities (e.g., lidar and infrared), control the drone in real time with a controller, and much more. For collecting this texture dataset, however, we did not need any of these features and thus opted for the blueprint method so we could use UE5.

E. Collected Dataset

In total, we have collected 6,297 images collected at 720×720 resolution for each of the six texture primitives shown in Figure 22. To create a single stream with samples from each of the primitives, we employ a Markov chain-like model for choosing which primitive should appear in each from of the combined video. We start by selecting a random primitive for the first frame of the dataset. Then, we take a frame from that primitive and, with probability p, switch to a different primitive for the next frame, chosen uniformly at random. This process is repeated until we have reached the end of the collected imagery. For our choice of $p = 0.01$, we have, on average, runs of 100 examples from the same primitive before switching. The resulting labels for the stream are shown in Figure 30. One can think of this collection process as having six camera angles for a scene and stochastically cutting between them.

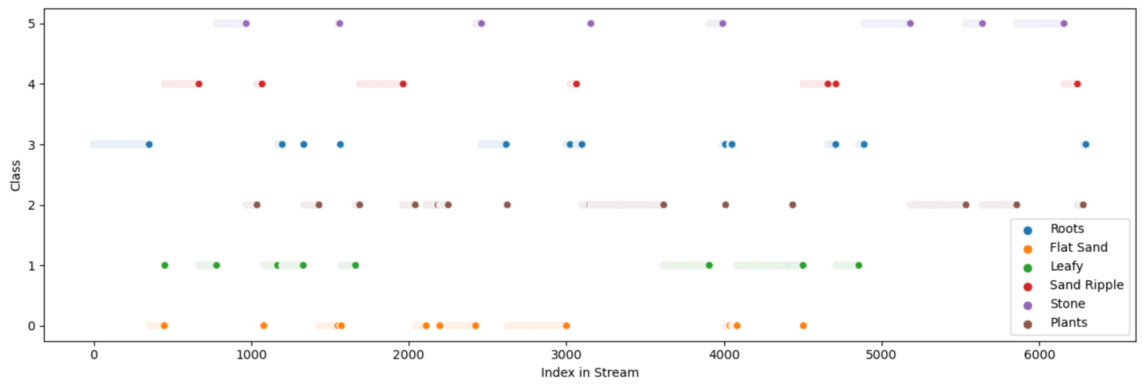


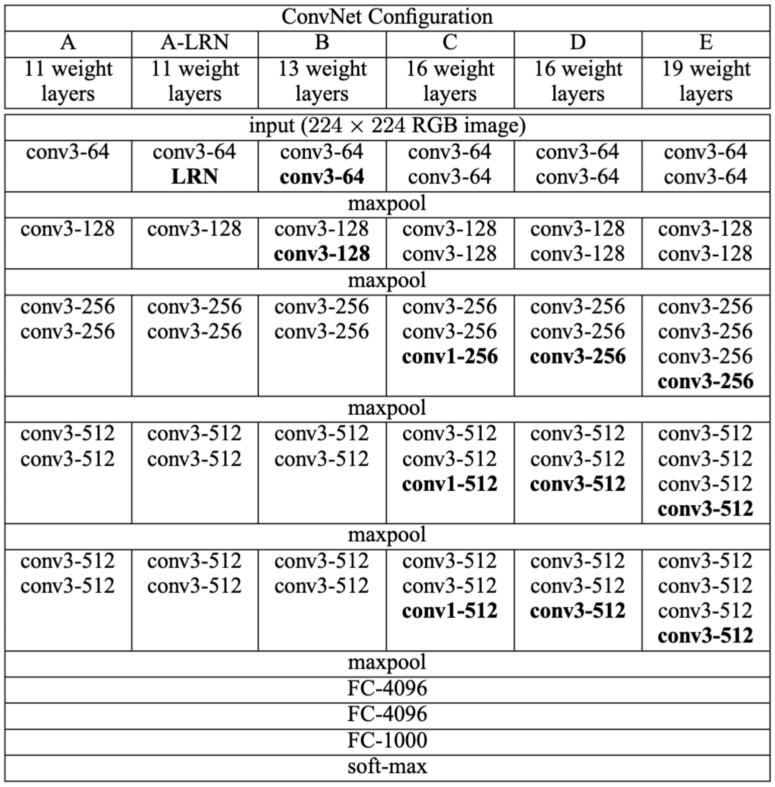
Figure 30: Plot showing the label of each image in the stream.

F. Experiments

The natural follow-up to creating a new dataset for SDA is to evaluate the performance of prominent SDA algorithms on it. We would like to emphasize that the goal of these experiments is less so to rank the algorithms that are run on the dataset and more to gain understanding about how how well the dataset lends itself to the problem of SDA. For detailed analysis on relative performance of SDA algorithms, including the two proposed in this overarching work, we refer the reader to the sections dedicated to this matter (Section III-E and Section IV-D).

1. *Feature Extraction*

None of the SDA algorithms we use in this work can operate directly on images and instead rely on a feature extractor to reduce the dimensionality from 720×720 to a more manageable size. We have found pre-trained deep convolutional neural networks are excellent at extract-



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 4: VGG network architecture, reproduced from Table 1 in Simonyan and Zisserman [79]. We use VGG-16, corresponding to column D of the table.

ing features from texture images, and in this experiment, we use the VGG-16 architecture [79] (Table 4), pre-trained on ImageNet [36]. By discarding the FC-1000 and subsequent layers, we are able to extract very descriptive and discriminative 4096-dimensional features from images. When applied to images from the UE5 dataset, we observe in Figure 31 that each class in the dataset appears to be grouped together in feature space and are reasonably well-separated.

Though the 4096-dimensional VGG-16 features are indicated by *t*-SNE to be separable, the dimensionality is still too high for most SDA algorithms to handle. For further dimensionality reduction, we apply

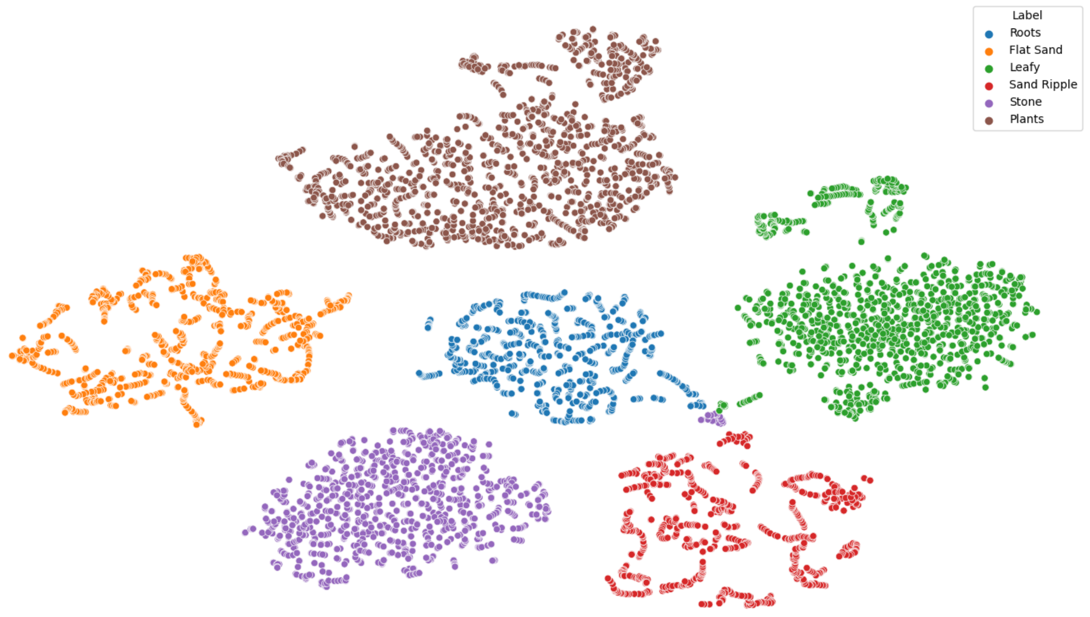
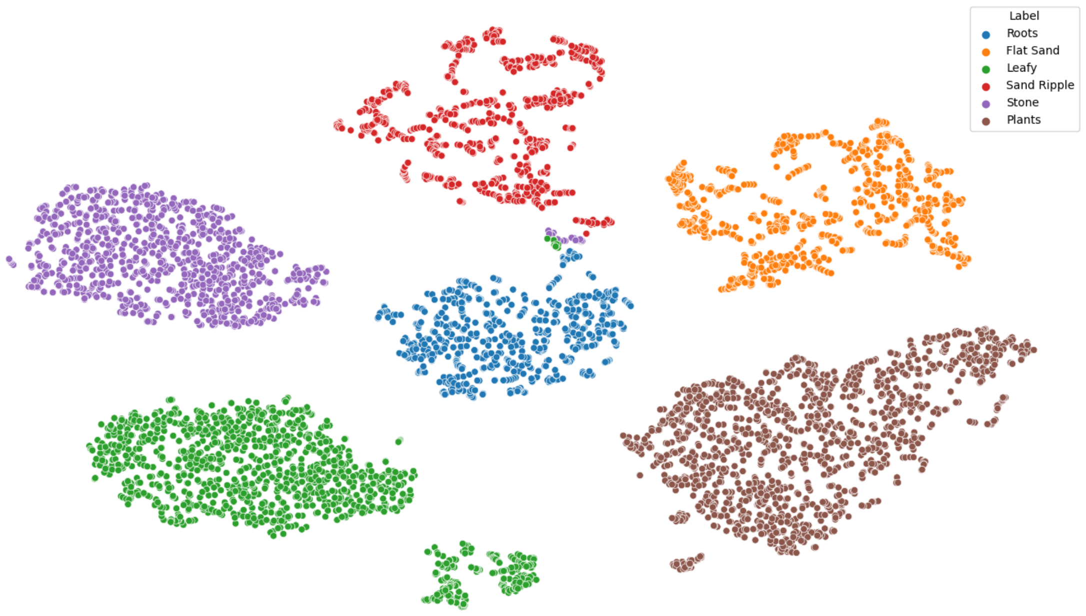


Figure 31: t-SNE plot of 4096-dimensional output of the VGG-16 feature extractor when run on the UE5 dataset.

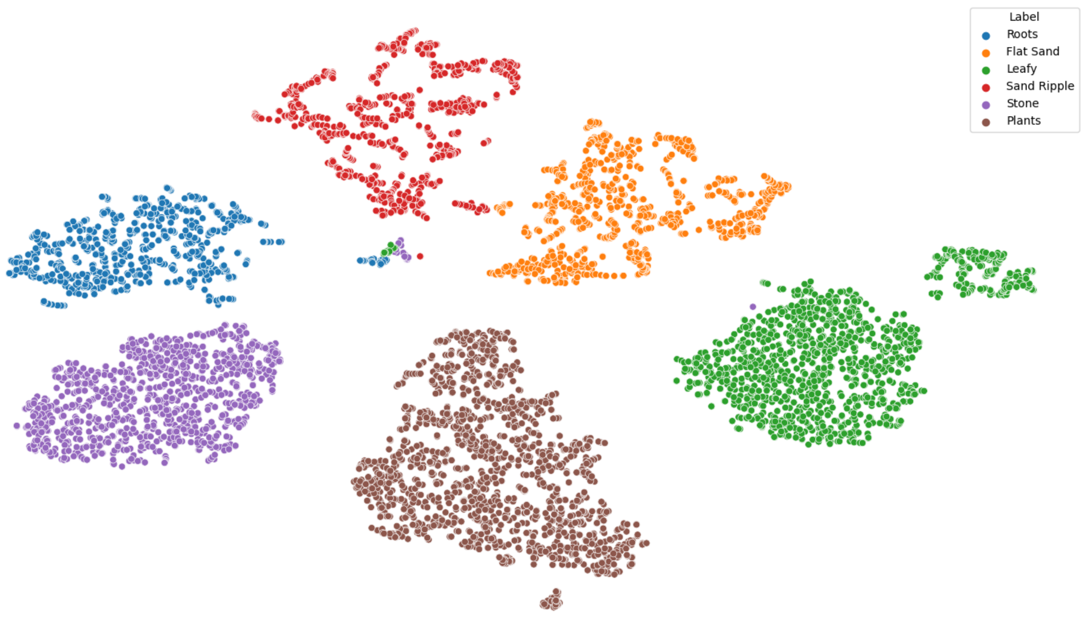
PCA to the dataset to produce features of varying dimension. However, PCA requires the whole dataset to be present at runtime, and thus we cannot justify using the entire streaming dataset to compute the PCA projection. Instead, we use the KTH-TIPS2b dataset to define the PCA projection and then apply this projection to the UE5 dataset. t-SNE embeddings for some lower dimensional PCA projections using this strategy are shown in Figure 32, and appear to maintain a large amount of the separability between the classes in the dataset.

2. *Dataset Evaluation*

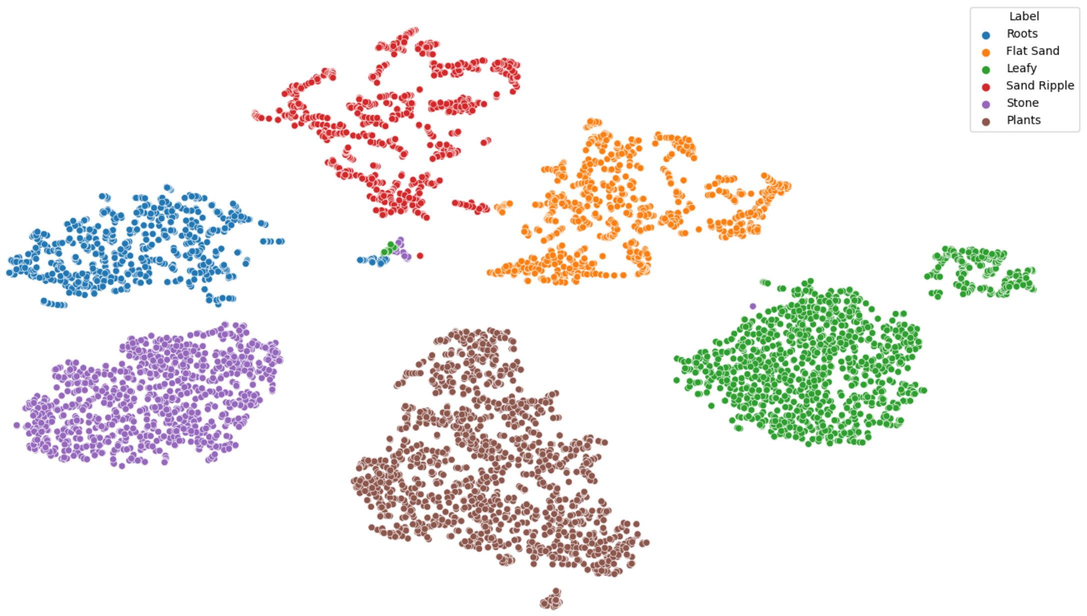
In Figure 33, we run several SDA algorithms on the UE5 dataset after reducing its dimensionality to 36, 24, 16, and 12 with PCA, respectively.



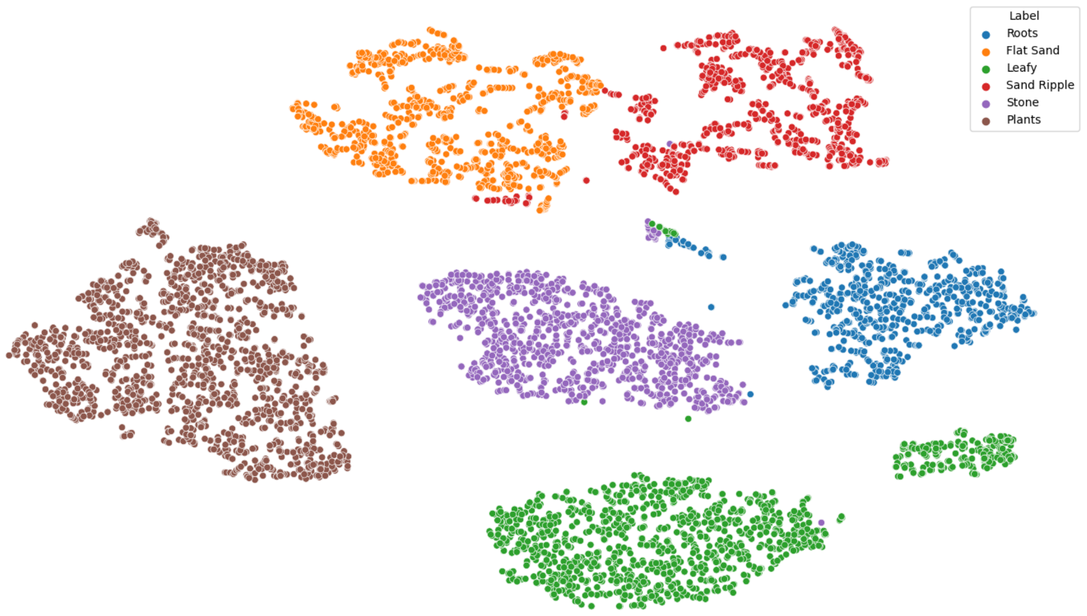
(a) t-SNE plot of 36-dimensional PCA projection.



(b) t-SNE plot of 24-dimensional PCA projection.

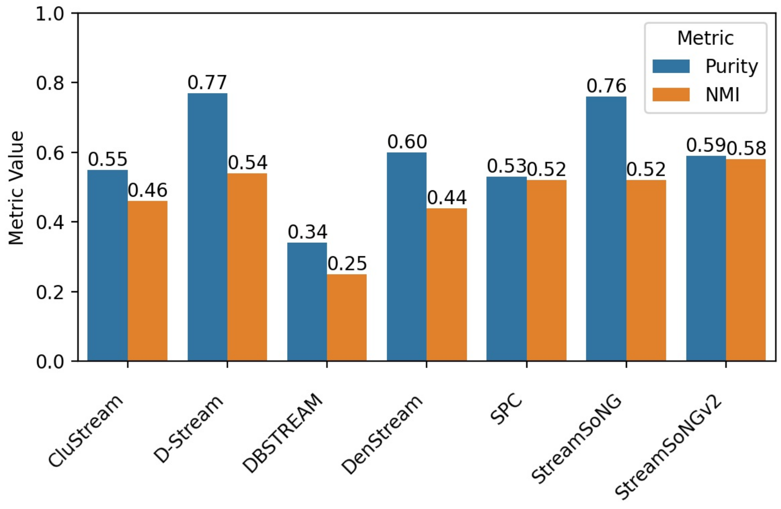


(c) t-SNE plot of 16-dimensional PCA projection.

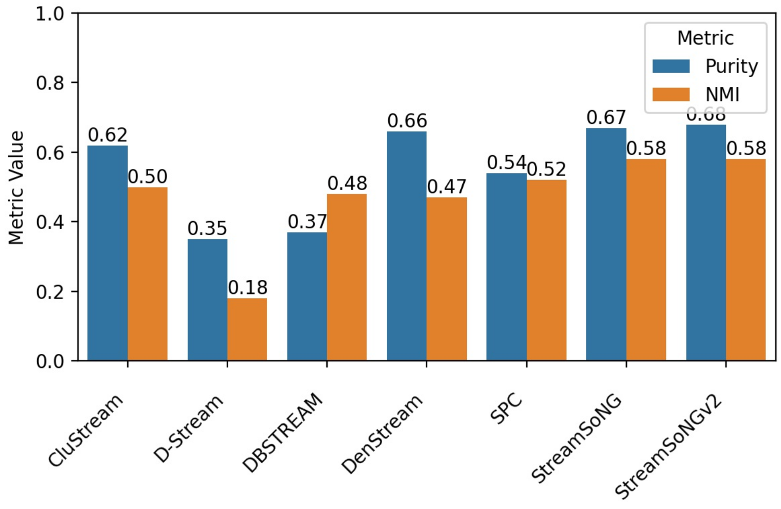


(d) t-SNE plot of 12-dimensional PCA projection.

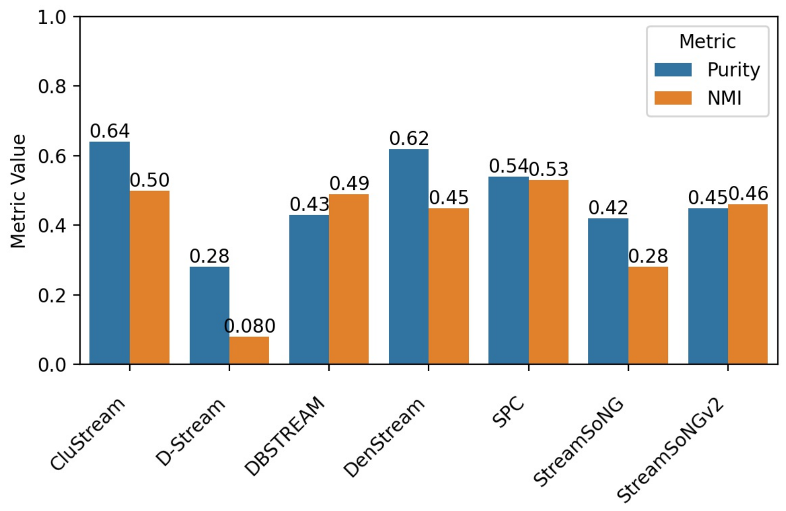
Figure 32: Projection from 4096-dimensional VGG-16 features on UE5 dataset to various lower dimensions using PCA, visualized in two-dimensions using t-SNE.



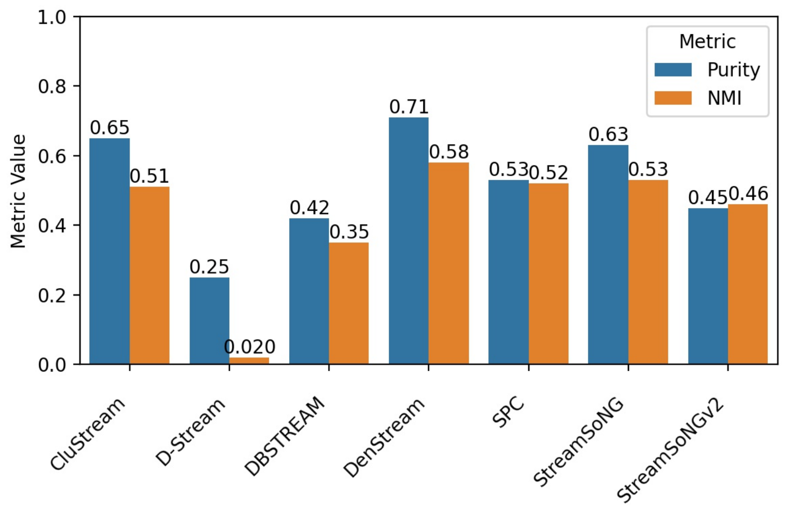
(a) SDA algorithm performance on 36-dimensional PCA projection of 4096-dimensional VGG-16 features.



(b) SDA algorithm performance on 24-dimensional PCA projection of 4096-dimensional VGG-16 features.



(c) SDA algorithm performance on 16-dimensional PCA projection of 4096-dimensional VGG-16 features.



(d) SDA algorithm performance on 12-dimensional PCA projection of 4096-dimensional VGG-16 features.

Figure 33: Evaluation of UE5 dataset based on SDA algorithm performance.

We observe that the performance of each algorithm is relatively consistent across dimensionality, and, as a whole, much higher than that of the KTH-TIPS2b dataset experiment performed in Section III-E.

The consistency across dimensionality suggests that the important features for texture recognition are either strongly correlated or lie on a low-dimensional manifold, and hence, the same information that is powerful for texture classification in the full 4096-dimensional VGG-16 features can mostly be conveyed in much lower dimensional spaces. This property of separability being largely independent of dimensionality is advantageous in using the UE5 dataset for benchmarking SDA algorithms in how well they scale with dimensionality.

G. Conclusion

Unreal Engine 5 opens new doors to creating realistic simulated datasets with an unprecedented degree of control over factors that, when collecting real-world data, must simply be accepted as uncontrollable. Namely, for simulated aerial image datasets like the one proposed here, the time of day, shadow intensity, camera properties, drone speed, drone height, and much more are at the full discretion of the user. Furthermore, since all of the collected imagery comes from a world of our own design, we truly have a dataset that satisfies the closed-world assumption.

Beyond producing a dataset of six unique texture primitives suited for SDA, we explored the use of deep convolutional neural networks in tandem with PCA for low-dimensional feature extraction from these images. We found, based both on perceived separability in t-SNE plots and on SDA algorithm performance, that the imagery in this dataset remains separable with respect to these features across varying degrees of dimensionality reduction. These properties make the UE5 texture dataset valuable as a benchmark dataset for unsupervised learning problems, both in the context of SDA and otherwise. Given the ease of generating more additional imagery from existing texture classes and the ability to add more classes to the dataset, one could generate orders of magnitude more labeled data and use this dataset in supervised learning as well, perhaps in pre-training a deep learning model prior to training on a smaller, real dataset.

VI. Conclusion

This work has focused on the problem of streaming data analysis (SDA) with an emphasis on application to the domain of streaming texture recognition. We have proposed two algorithms: StreamSoNGv2, an improvement to StreamSoNG, and single-pass possibilistic clustering (SPC), a novel approach to streaming clustering that leverages possibilistic memberships. These two algorithms are both part of the general category of SDA, but are designed for different tasks.

StreamSoNGv2's specialization is in online classification, that is, for problems where the goal is to produce a soft classification decision after seeing each sample that measures how much typicality the sample has in each known class. Paired with the ability to detect and model new classes as they emerge, StreamSoNGv2 is most useful when one is more interested in obtaining information about specific samples in the stream than in obtaining information about the stream as a whole.

Conversely, the goal of SPC is to produce a high quality summary of the stream using a fixed amount of memory. The structures in SPC are dynamically created, merged, and deleted to reflect the information gained from each point in the stream, resulting in a configuration of structures that, at any given time, summarize points from the stream. Older points in the stream can be given an exponentially decaying weight by SPC in the form of decay factor parameters that govern the length of a damped window used in updating the mean and covariance of each structure. By setting the decay factor parameters to zero, the entire stream is summarized with equal weight, independent of recency, whereas higher decay factor parameters cause older points in the stream to be forgotten more quickly.

Of course, both algorithms can perform both tasks. StreamSoNGv2 produces summary information about the entire stream in the form of its growing neural gas prototypes and SPC can perform online classification of points in the stream by considering nearby structures and their cluster label. By bringing to light the diverse use cases of the two algorithms, we hope that users can choose the algorithm best suited to the task at hand to obtain high quality information about the data stream under consideration.

Furthermore, we have developed a synthetic texture dataset through simulation in Unreal Engine 5 (UE5) to provide a medium for constructing streams of texture imagery. Using engine features as well as external scripts, we have created a stream of over 6000 images paired with pixel-perfect ground truth texture memberships. Though not directly used in this work, objects can also be randomly placed in the scene and associated with pixelwise segmentation labels for object detection tasks. The automated nature of this “data collect” provides us with the potential of creating even more data in the same way, limited only by disk space.

VII. Publications

To date, I have been involved in eighteen publications, nine of which I am primary author. The table below categorizes each publication based on my role on the paper and the venue. Following the table, citations for each publication are provided.

	Workshop	**Conference**	**Journal**
Primary	1	6	2*
Contributing	1	7	1

* Both pending review.

A. Primary Author

1. Dale, J., Nishimoto, A., & Obafemi-Ajayi, T. (2018). Performance Evaluation and Enhancement of Biclustering Algorithms. In *ICPRAM* (pp. 202-213).
2. Dale, J., Matta, J., Howard, S., Ercal, G., Qiu, W., & Obafemi-Ajayi, T. (2018, May). Analysis of grapevine gene expression data using node-based resilience clustering. In *2018 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)* (pp. 1-8). IEEE.
3. Dale, J., & Clark, A. (2018, December). An Ensemble of Face Recognition Algorithms for Unsupervised Expansion of Training Data. In *2018 International Conference on Computational Science and Computational Intelligence (CSCI)* (pp. 342-347). IEEE.
4. Dale, J., Zhao, J., & Obafemi-Ajayi, T. (2019, July). Multi-objective Optimization Approach to find Biclusters in Gene Expression Data. In *2019 IEEE Conference on Computational In-*

telligence in Bioinformatics and Computational Biology (CIBCB) (pp. 1-8). IEEE.

5. Dale, J., Galusha, A., Keller, J., & Zare, A. (2019, May). Evaluation of image features for discriminating targets from false positives in synthetic aperture sonar imagery. In *Detection and Sensing of Mines, Explosive Objects, and Obscured Targets XXIV* (Vol. 11012, p. 110120A). International Society for Optics and Photonics.
6. Dale, J. J., Huangal, D., Hurt, J. A., Bajkowski, T. M., Keller, J. M., Scott, G. J., & Price, S. R. (2020, October). Detection of unknown maneuverability hazards in low-altitude uas color imagery using linear features. In *2020 IEEE Applied Imagery Pattern Recognition Workshop (AIPR)* (pp. 1-9). IEEE.
7. Dale, J., Bajkowski, T., Hurt, J. A., Huangal, D., Earle, N., Keller, J., ... & Price, S. (2021, April). Towards an explainable AI adjunct to deep network obstacle detection for multisensor vehicle maneuverability assessment. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications III* (Vol. 11746, p. 117462H). International Society for Optics and Photonics.

B. Contributing Author

1. Galusha, A., Dale, J., Keller, J. M., & Zare, A. (2019, May). Deep convolutional neural network target classification for underwater synthetic aperture sonar imagery. In *Detection and Sensing of Mines, Explosive Objects, and Obscured Targets XXIV* (Vol. 11012, p. 1101205). International Society for Optics and Photonics.
2. Hurt, J. A., Huangal, D., Dale, J., Bajkowski, T. M., Keller, J. M., Scott, G. J., & Price, S. R. (2020, April). Maneuverability hazard detection and localization in low-altitude uas imagery. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications II* (Vol. 11413, p. 114131K). International Society

for Optics and Photonics.

3. Huangal, D., Dale, J., Hurt, J. A., Bajkowski, T. M., Keller, J. M., Scott, G. J., & Price, S. R. (2020, April). Evaluating deep road segmentation techniques for low-altitude uas imagery. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications II* (Vol. 11413, p. 114131L). International Society for Optics and Photonics.

4. Hurt, J. A., Scott, G. J., Huangal, D., Dale, J., Bajkowski, T. M., Keller, J. M., & Price, S. R. (2021, April). Differential morphological profile neural network for maneuverability hazard detection in unmanned aerial system imagery. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications III* (Vol. 11746, p. 1174629). International Society for Optics and Photonics.

5. Bajkowski, T., Hurt, J. A., Huangal, D., Dale, J., Keller, J., Scott, G., & Price, S. (2021, April). Accumulating confidence for deep neural network object detections and semantic segmentations in sequential UAS imagery through spatiotemporal feature correspondences generated from SfM techniques. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications III* (Vol. 11746, p. 117462V). International Society for Optics and Photonics.

6. Wu, W., Keller, J. M., Dale, J., & Bezdek, J. C. (2021). Stream-SoNG: A Soft Streaming Classification Approach. In *IEEE Transactions on Emerging Topics in Computational Intelligence.*

7. Bajkowski, T. M., Hurt, J. A., Dale, J., Huangal, D., Keller, J. M., Scott, G. J., & Price, S. R. (2021, October). Evaluating Visuospatial Features for Tracking Hazards in Overhead UAS Imagery. In *2021 IEEE Applied Imagery Pattern Recognition Workshop (AIPR)* (pp. 1-6). IEEE.

8. Stewart, D., Hampton, A., Zare, A., Dale, J., & Keller, J. (2021, July). The weakly-labeled Rand index. In *2021 IEEE International Geoscience and Remote Sensing Symposium IGARSS* (pp. 2313-2316). IEEE.

9. Walker, S., Peeples, J., Dale, J., Keller, J., & Zare, A. (2021, July). Explainable Systematic Analysis for Synthetic Aperture Sonar Imagery. In *2021 IEEE International Geoscience and Remote Sensing Symposium IGARSS* (pp. 2835-2838). IEEE.

C. Submitted Publications

1. Dale, J., Galusha, A., & Keller, J. (Pending). Single-pass possibilistic clustering with damped window footprints. Submitted to *IEEE Transactions on Fuzzy Systems*.
2. Dale, J., Galusha, A., & Keller, J. (Pending). StreamSoNGv2: Online classification of data streams using growing neural gas. Submitted to *IEEE Transactions on Emerging Topics in Computational Intelligence*.

References

[1] "BMX" (https://skfb.ly/6Uxuo) by Warkarma is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/). (cit. on pp. 102, 106).

[2] "Garbage bags" (https://skfb.ly/orDMJ) by DJMaesen is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/). (cit. on pp. 102, 106).

[3] "Medieval Chest" (https://skfb.ly/MzHZ) by AIV is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/). (cit. on pp. 102, 106).

[4] "Old Rusty Car 2020 (17K Followers!)" (https://skfb.ly/6SCUS) by Renafox is licensed under Creative Commons Attribution-NonCommercial (http://creativecommons.org/licenses/by-nc/4.0/). (cit. on pp. 102, 106).

[5] "The Wrench - Pipe Wrench" (https://skfb.ly/o8Zxo) by Miguel Adão is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/). (cit. on pp. 102, 106).

[6] "Texturing Suitcase Challenge" (https://skfb.ly/6SHVH) by Warkarma is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/). (cit. on pp. 102, 106).

[7] "Trash Shelter""(https://skfb.ly/o8X89) by jcasetta is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/). (cit. on pp. 102, 106).

[8] "Mine (Anti Tank)" (https://skfb.ly/6WQSu) by Warkarma is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/). (cit. on pp. 102, 106).

[9] "Claymore" (`https://skfb.ly/69GsI`) by Warkarma is licensed under Creative Commons Attribution (`http://creativecommons.org/licenses/by/4.0/`). (cit. on pp. 102, 106).

[10] "Fat Man - Nuclear bomb" (`https://skfb.ly/6SHVO`) by Warkarma is licensed under Creative Commons Attribution (`http://creativecommons.org/licenses/by/4.0/`). (cit. on pp. 102, 106).

[11] "Doomsday Device" (`https://skfb.ly/69GsP`) by Warkarma is licensed under Creative Commons Attribution (`http://creativecommons.org/licenses/by/4.0/`). (cit. on pp. 102, 106).

[12] "Grenade Edgy" (`https://skfb.ly/6WQTr`) by Warkarma is licensed under Creative Commons Attribution (`http://creativecommons.org/licenses/by/4.0/`). (cit. on pp. 102, 106).

[13] "Little Boy - Nuclear bomb" (`https://skfb.ly/68pxq`) by Warkarma is licensed under Creative Commons Attribution (`http://creativecommons.org/licenses/by/4.0/`). (cit. on pp. 102, 106).

[14] "TNT Sack" (`https://skfb.ly/6WOGJ`) by Warkarma is licensed under Creative Commons Attribution (`http://creativecommons.org/licenses/by/4.0/`). (cit. on pp. 102, 106).

[15] Charu C Aggarwal et al. "A framework for clustering evolving data streams". In: *Proceedings 2003 VLDB conference*. Elsevier. 2003, pp. 81–92 (cit. on pp. 7, 10, 13, 14, 46, 54).

[16] Charu C Aggarwal et al. "A framework for clustering evolving data streams". In: *Proceedings 2003 VLDB conference*. Elsevier. 2003, pp. 81–92 (cit. on p. 82).

[17] Brendan Alvey et al. "Improving explosive hazard detection with simulated and augmented data for an unmanned aerial system". In: *Detection and Sensing of*

Mines, Explosive Objects, and Obscured Targets XXVI. Vol. 11750. International Society for Optics and Photonics. 2021, 117500B (cit. on p. 99).

[18] Amineh Amini, Teh Ying Wah, and Hadi Saboohi. “On density-based data streams clustering algorithms: A survey”. In: *Journal of Computer Science and Technology* 29.1 (2014), pp. 116–141 (cit. on p. 19).

[19] Amineh Amini et al. “MuDi-Stream: A multi density clustering algorithm for evolving data stream”. In: *Journal of Network and Computer Applications* 59 (2016), pp. 370–385 (cit. on p. 68).

[20] Jonathan de Andrade Silva, Eduardo Raul Hruschka, and João Gama. “An evolutionary algorithm for clustering data streams with a variable number of clusters”. In: *Expert Systems with Applications* 67 (2017), pp. 228–238 (cit. on p. 68).

[21] Fabrizio Angiulli and Fabio Fassetti. “Distance-based outlier queries in data streams: the novel task and algorithms”. In: *Data Mining and Knowledge Discovery* 20.2 (2010), pp. 290–324 (cit. on p. 21).

[22] Alessio Bechini, Francesco Marcelloni, and Alessandro Renda. “TSF-DBSCAN: a novel fuzzy density-based approach for clustering unbounded data streams”. In: *IEEE Transactions on Fuzzy Systems* (2020) (cit. on p. 68).

[23] James C Bezdek, Robert Ehrlich, and William Full. “FCM: The fuzzy c-means clustering algorithm”. In: *Computers & geosciences* 10.2-3 (1984), pp. 191–203 (cit. on p. 5).

[24] James C Bezdek and James M Keller. “Streaming Data Analysis: Clustering or Classification?” In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 51.1 (2020), pp. 91–102 (cit. on p. 4).

[25] Markus M Breunig et al. "LOF: identifying density-based local outliers". In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 2000, pp. 93–104 (cit. on p. 22).

[26] Jason Brownlee. *Clever algorithms: nature-inspired programming recipes*. Jason Brownlee, 2011, pp. 108–113 (cit. on p. 17).

[27] Charles George Broyden. "The convergence of a class of double-rank minimization algorithms 1. general considerations". In: *IMA Journal of Applied Mathematics* 6.1 (1970), pp. 76–90 (cit. on p. 51).

[28] Richard H Byrd et al. "A limited memory algorithm for bound constrained optimization". In: *SIAM Journal on scientific computing* 16.5 (1995), pp. 1190–1208 (cit. on p. 51).

[29] Feng Cao et al. "Density-based clustering over an evolving data stream with noise". In: *Proceedings of the 2006 SIAM international conference on data mining*. SIAM. 2006, pp. 328–339 (cit. on pp. 10, 13, 15, 19, 54, 68, 82).

[30] Matthias Carnein and Heike Trautmann. "evostream–evolutionary stream clustering utilizing idle times". In: *Big data research* 14 (2018), pp. 101–111 (cit. on pp. 10, 16, 18, 54).

[31] Matthias Carnein and Heike Trautmann. "Optimizing data stream representation: An extensive survey on stream clustering algorithms". In: *Business & Information Systems Engineering* 61.3 (2019), pp. 277–297 (cit. on pp. 10, 12, 13, 18, 54).

[32] Yixin Chen and Li Tu. "Density-based clustering for real-time stream data". In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2007, pp. 133–142 (cit. on pp. 7, 12, 13, 18, 22, 54).

[33] Yixin Chen and Li Tu. “Density-based clustering for real-time stream data”. In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2007, pp. 133–142 (cit. on p. 82).

[34] Jeffrey Dale et al. “Analysis of grapevine gene expression data using node-based resilience clustering”. In: *2018 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*. IEEE. 2018, pp. 1–8 (cit. on p. 22).

[35] Donald A Darling. “The kolmogorov-smirnov, cramer-von mises tests”. In: *The Annals of Mathematical Statistics* 28.4 (1957), pp. 823–838 (cit. on p. 49).

[36] Jia Deng et al. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255 (cit. on p. 114).

[37] Martin Ester et al. “A density-based algorithm for discovering clusters in large spatial databases with noise.” In: *kdd*. Vol. 96. 34. 1996, pp. 226–231 (cit. on pp. xii, 15, 19, 40).

[38] Martin Ester et al. “A density-based algorithm for discovering clusters in large spatial databases with noise.” In: *kdd*. Vol. 96. 34. 1996, pp. 226–231 (cit. on pp. 66, 68).

[39] Evelyn Fix and Joseph Lawson Hodges. “Discriminatory analysis. Nonparametric discrimination: Consistency properties”. In: *International Statistical Review/Revue Internationale de Statistique* 57.3 (1989), pp. 238–247 (cit. on p. 29).

[40] Roger Fletcher. “A new approach to variable metric algorithms”. In: *The computer journal* 13.3 (1970), pp. 317–322 (cit. on p. 51).

[41] P. Fränti, O. Virmajoki, and V. Hautamäki. “Fast agglomerative clustering using a k-nearest neighbor graph”. In: *IEEE Trans. on Pattern Analysis and Machine Intelligence* 28.11 (2006), pp. 1875–1881 (cit. on pp. 89, 91).

[42] Hichem Frigui and Paul Gader. “Detection and discrimination of land mines in ground-penetrating radar based on edge histogram descriptors and a possibilistic k-nearest neighbor classifier”. In: *IEEE Transactions on Fuzzy Systems* 17.1 (2008), pp. 185–199 (cit. on pp. 5, 29, 32).

[43] Bernd Fritzke et al. “A growing neural gas network learns topologies”. In: *Advances in neural information processing systems* 7 (1995), pp. 625–632 (cit. on pp. xii, 16, 26, 28).

[44] A Galusha et al. “Deep convolutional neural network target classification for underwater synthetic aperture sonar imagery”. In: *Detection and Sensing of Mines, Explosive Objects, and Obscured Targets XXIV*. Vol. 11012. International Society for Optics and Photonics. 2019, p. 1101205 (cit. on p. xi).

[45] Mohammed Ghesmoune, Mustapha Lebbah, and Hanene Azzag. “A new growing neural gas for clustering data streams”. In: *Neural Networks* 78 (2016), pp. 36–50 (cit. on pp. 10, 13, 16, 41, 54).

[46] Aristides Gionis, Heikki Mannila, and Panayiotis Tsaparas. “Clustering aggregation”. In: *Acm transactions on knowledge discovery from data (tkdd)* 1.1 (2007), 4–es (cit. on pp. 55, 58, 82, 85, 140).

[47] Donald Goldfarb. “A family of variable-metric methods derived by variational means”. In: *Mathematics of computation* 24.109 (1970), pp. 23–26 (cit. on p. 51).

[48] Michael Hahsler and Matthew Bolanos. *streamMOA: Interface for MOA Stream Clustering Algorithms*. `https://CRAN.R-project.org/package=streamMOA`. 2015 (cit. on p. 54).

[49] Michael Hahsler, Matthew Bolanos, and John Forrest. “Introduction to stream: An extensible framework for data stream clustering research with r”. In: *Journal of Statistical Software* 76 (2017), pp. 1–50 (cit. on p. 54).

[50] Michael Hahsler and Matthew Bolaños. “Clustering data streams based on shared density between micro-clusters”. In: *IEEE Transactions on Knowledge and Data Engineering* 28.6 (2016), pp. 1449–1461 (cit. on pp. 7, 12, 13, 18–21, 54, 68, 82).

[51] Mryka Hall-Beyer. “GLCM texture: a tutorial”. In: *National Council on Geographic Information and Analysis Remote Sensing Core Curriculum* 3.1 (2000), p. 75 (cit. on p. 59).

[52] Zhengbing Hu et al. “A neuro-fuzzy Kohonen network for data stream possibilistic clustering and its online self-learning procedure”. In: *Applied soft computing* 68 (2018), pp. 710–718 (cit. on p. 67).

[53] Richard Hyde, Plamen Angelov, and Angus Robert MacKenzie. “Fully online clustering of evolving data streams into arbitrarily shaped clusters”. In: *Information Sciences* 382 (2017), pp. 96–114 (cit. on p. 68).

[54] Tesla Inc. *Tesla AI Day.* `https://youtu.be/j0z4FweCy4M`. Accessed 2021-08-23 (cit. on p. 99).

[55] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning.* PMLR. 2015, pp. 448–456 (cit. on p. 52).

[56] Lu Jiang et al. “Beyond synthetic noise: Deep learning on controlled noisy labels”. In: *International Conference on Machine Learning.* PMLR. 2020, pp. 4804–4815 (cit. on p. 5).

[57] Simon J Julier, Jeffrey K Uhlmann, and David Nicholson. “A method for dealing with assignment ambiguity”. In: *Proceedings of the 2004 American Control Conference.* Vol. 5. IEEE. 2004, pp. 4102–4107 (cit. on pp. 69, 78, 80).

[58] Karbhari V Kale et al. “A research review on hyperspectral data processing and analysis algorithms”. In: *Proceedings of the national academy of sciences, India section a: physical sciences* 87.4 (2017), pp. 541–555 (cit. on p. 33).

[59] James M Keller and Paul Gader. “Fuzzy logic and the principle of least commitment in computer vision”. In: *1995 IEEE International Conference on Systems, Man and Cybernetics. Intelligent Systems for the 21st Century.* Vol. 5. IEEE. 1995, pp. 4621–4625 (cit. on pp. 5, 43).

[60] James M Keller, Michael R Gray, and James A Givens. “A fuzzy k-nearest neighbor algorithm”. In: *IEEE transactions on systems, man, and cybernetics* 4 (1985), pp. 580–585 (cit. on pp. 29, 33).

[61] Minho Kim and RS Ramakrishna. “New indices for cluster validity assessment”. In: *Pattern Recognition Letters* 26.15 (2005), pp. 2353–2363 (cit. on pp. 54, 172).

[62] Raghuram Krishnapuram and James M Keller. “A possibilistic approach to clustering”. In: *IEEE transactions on fuzzy systems* 1.2 (1993), pp. 98–110 (cit. on pp. xii, 32, 34, 46, 70, 71).

[63] Raghuram Krishnapuram and James M Keller. “The possibilistic c-means algorithm: insights and recommendations”. In: *IEEE transactions on Fuzzy Systems* 4.3 (1996), pp. 385–393 (cit. on p. 47).

[64] Andrea Lancichinetti, Santo Fortunato, and János Kertész. “Detecting the overlapping and hierarchical community structure in complex networks”. In: *New journal of physics* 11.3 (2009), p. 033015 (cit. on pp. 54, 172, 173).

[65] Carson Kai-Sang Leung and Fan Jiang. “Frequent itemset mining of uncertain data streams using the damped window model”. In: *Proceedings of the 2011 ACM Symposium on Applied Computing.* 2011, pp. 950–955 (cit. on p. 46).

[66] Dong C Liu and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". In: *Mathematical programming* 45.1 (1989), pp. 503–528 (cit. on p. 51).

[67] Xingjun Ma et al. "Dimensionality-driven learning with noisy labels". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 3355–3364 (cit. on p. 5).

[68] P Mallikarjuna et al. "The kth-tips2 database". In: *Computational Vision and Active Perception Laboratory, Stockholm, Sweden* (2006), pp. 1–10 (cit. on pp. xii, 58, 168).

[69] Thomas Martinetz, Klaus Schulten, et al. "A "neural-gas" network learns topologies". In: (1991) (cit. on pp. xii, 25, 27, 32, 36).

[70] John Matta et al. "Robust graph-theoretic clustering approaches using node-based resilience measures". In: *2016 IEEE 16th international conference on data mining (ICDM)*. IEEE. 2016, pp. 320–329 (cit. on p. 22).

[71] Ken Perlin. "An image synthesizer". In: *ACM Siggraph Computer Graphics* 19.3 (1985), pp. 287–296 (cit. on p. 162).

[72] *Quixel Megascans - The World's Largest Scanned 3D Asset Library*. `https://quixel.com/megascans`. Accessed 2021-08-10 (cit. on p. 100).

[73] Ingo Rechenberg. "Evolutionsstrategien". In: *Simulationsmethoden in der Medizin und Biologie*. Springer, 1978, pp. 83–114 (cit. on p. 17).

[74] Brian Reinhardt et al. "Measuring Human Assessed Complexity in Synthetic Aperture Sonar Imagery Using the Elo Rating System". In: *arXiv preprint arXiv:1808.05279* (2018) (cit. on p. xi).

[75] T Runkler and J Keller. "Sequential Possibilistic One–Means Clustering with Variable Eta". In: *Proceedings, 27th Workshop on Computational Intelligence*. 2017, pp. 103–116 (cit. on p. 47).

[76] Thomas A Runkler and James M Keller. “Sequential possibilistic one-means clustering”. In: *2017 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. IEEE. 2017, pp. 1–6 (cit. on p. 32).

[77] Shital Shah et al. “Airsim: High-fidelity visual and physical simulation for autonomous vehicles”. In: *Field and service robotics*. Springer. 2018, pp. 621–635 (cit. on p. 108).

[78] David F Shanno. “Conditioning of quasi-Newton methods for function minimization”. In: *Mathematics of computation* 24.111 (1970), pp. 647–656 (cit. on p. 51).

[79] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014) (cit. on p. 114).

[80] Igor Škrjanc et al. “Inner matrix norms in evolving cauchy possibilistic clustering for classification and regression from data streams”. In: *Information Sciences* 478 (2019), pp. 540–563 (cit. on p. 67).

[81] Stefan Van der Walt et al. “scikit-image: image processing in Python”. In: *PeerJ* 2 (2014), e453 (cit. on p. 59).

[82] Pauli Virtanen et al. “SciPy 1.0: fundamental algorithms for scientific computing in Python”. In: *Nature methods* 17.3 (2020), pp. 261–272 (cit. on p. 51).

[83] Hongzhi Wang, Mohamed Jaward Bah, and Mohamed Hammad. “Progress in outlier detection techniques: A survey”. In: *Ieee Access* 7 (2019), pp. 107964–108000 (cit. on pp. 21, 22).

[84] Wenlong Wu, James M Keller, and Thomas A Runkler. “Sequential possibilistic one-means clustering with dynamic eta”. In: *2018 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. IEEE. 2018, pp. 1–8 (cit. on p. 47).

[85] Wenlong Wu et al. “StreamSoNG: A Soft Streaming Classification Approach”. In: *arXiv preprint arXiv:2010.00635* (2020) (cit. on pp. xii, 3, 7, 12, 13, 25, 32, 54).

[86] Wenlong Wu et al. “StreamSoNG: A Soft Streaming Classification Approach”. In: *IEEE Transactions on Emerging Topics in Computational Intelligence* (2021) (cit. on pp. 67, 82).

[87] Di Yang, Elke A Rundensteiner, and Matthew O Ward. “Neighbor-based pattern detection for windows over streaming data”. In: *Proceedings of the 12th international conference on extending database technology: advances in database technology*. 2009, pp. 529–540 (cit. on p. 22).

[88] Miin-Shen Yang, Shou-Jen Chang-Chien, and Yessica Nataliani. “A fully-unsupervised possibilistic c-means clustering algorithm”. In: *IEEE Access* 6 (2018), pp. 78308–78320 (cit. on p. 47).

[89] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. “BIRCH: an efficient data clustering method for very large databases”. In: *ACM sigmod record* 25.2 (1996), pp. 103–114 (cit. on pp. 8, 10, 13).

[90] Alaettin Zubaroğlu and Volkan Atalay. “Data stream clustering: a review”. In: *Artificial Intelligence Review* 54.2 (2021), pp. 1201–1236 (cit. on pp. 46, 68).

Appendix

We include extra results that are either not the primary focus of this work, too lengthy, or that may not be of interest to most readers in this section.

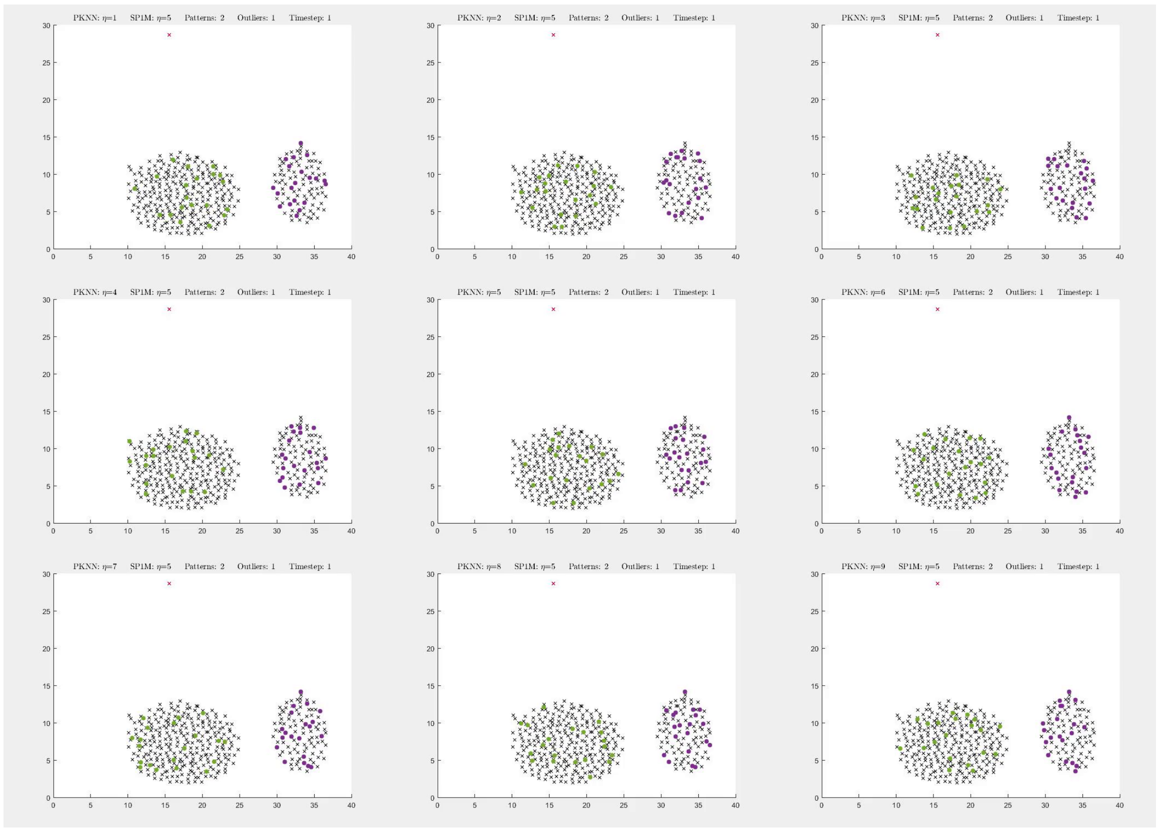
A. StreamSoNG Parameters

In this series of experiments, we use a two dimensional toy dataset originally published by Gionis et al. [46] and retrieved from `http://cs.joensuu.fi/sipu/datasets/` to illustrate the effect of key parameters on the ability of StreamSoNG to function. Points from the dataset are plotted as an $\times$ and are colored blue if StreamSoNG considered them an inlier at the current timestep or red if considered an outlier. The filled in circles in each plot represent neural gas prototypes and are color-coded according to which class they belong. Only the neural gas prototypes and outlier points are maintained by StreamSoNG at any given time, the rest are only kept for the purposes of visualization in these experiments.

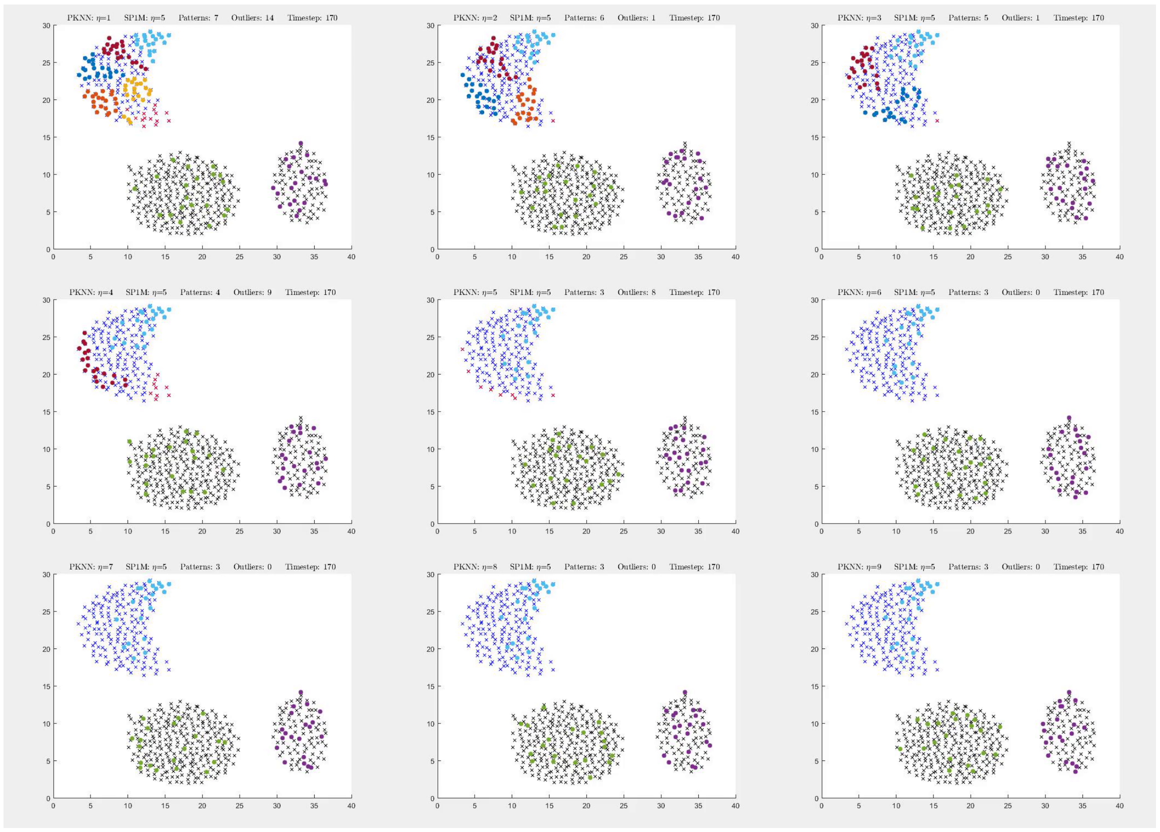
1. *Varying η in PKNN*

Comparison of nine choices of η in the PKNN component of StreamSoNG. For each of the 3×3 grid of plots that follow, in row-major order, the parameter values are $\eta = 1, 2, \ldots, 9$. We observe that when η is too small, e.g., $\eta = 1$, incoming points must be very close to a neural gas prototype to be considered in inlier. As a result, most points from the stream are marked as outliers and are then clustered by SP1M in the outlier list to form new classes in the model. Inversely, when η is too large, e.g., $\eta = 9$, nearly all points in the stream are marked as inliers and it is very difficult to form a new class. For this dataset, somewhere in between $\eta = 5$ and $\eta = 6$ seems to be a good choice.

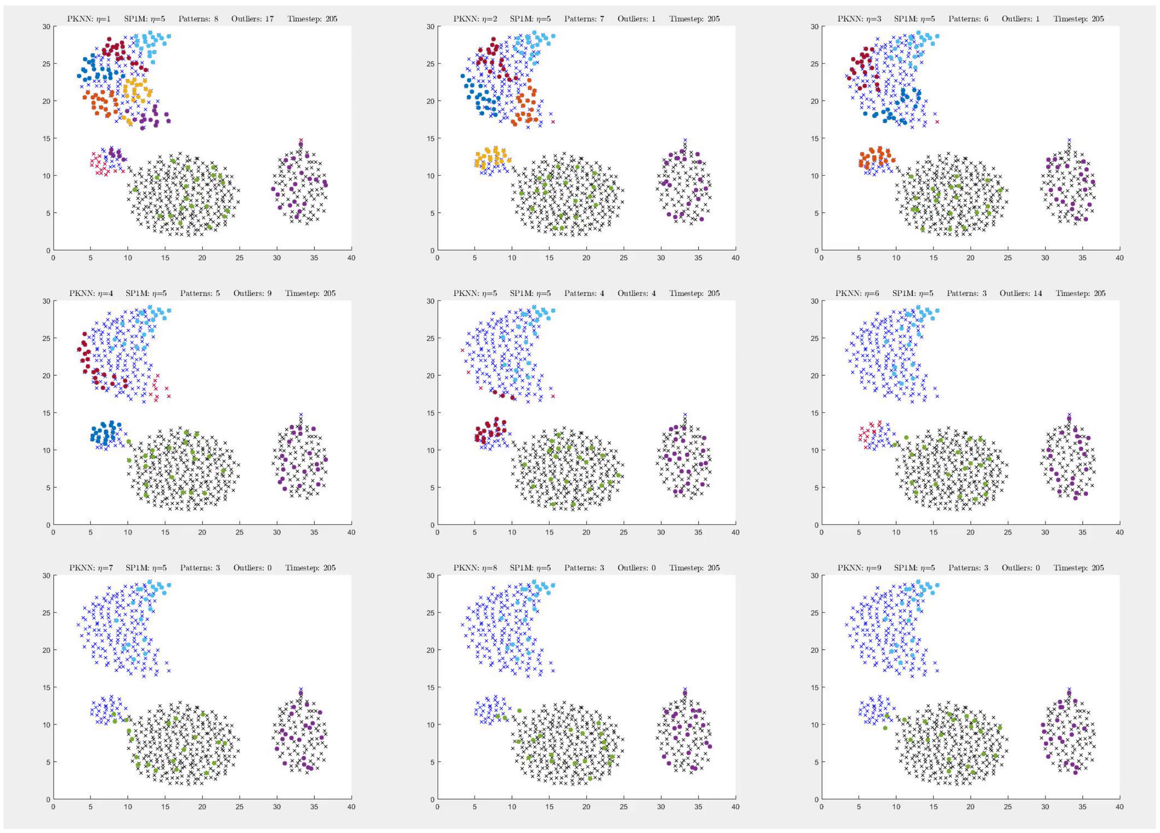
All other parameters are fixed at predetermined reasonable values for this dataset. An animated version of this figure can be viewed at `https://youtu.be/BuGs-9aUcYk`.



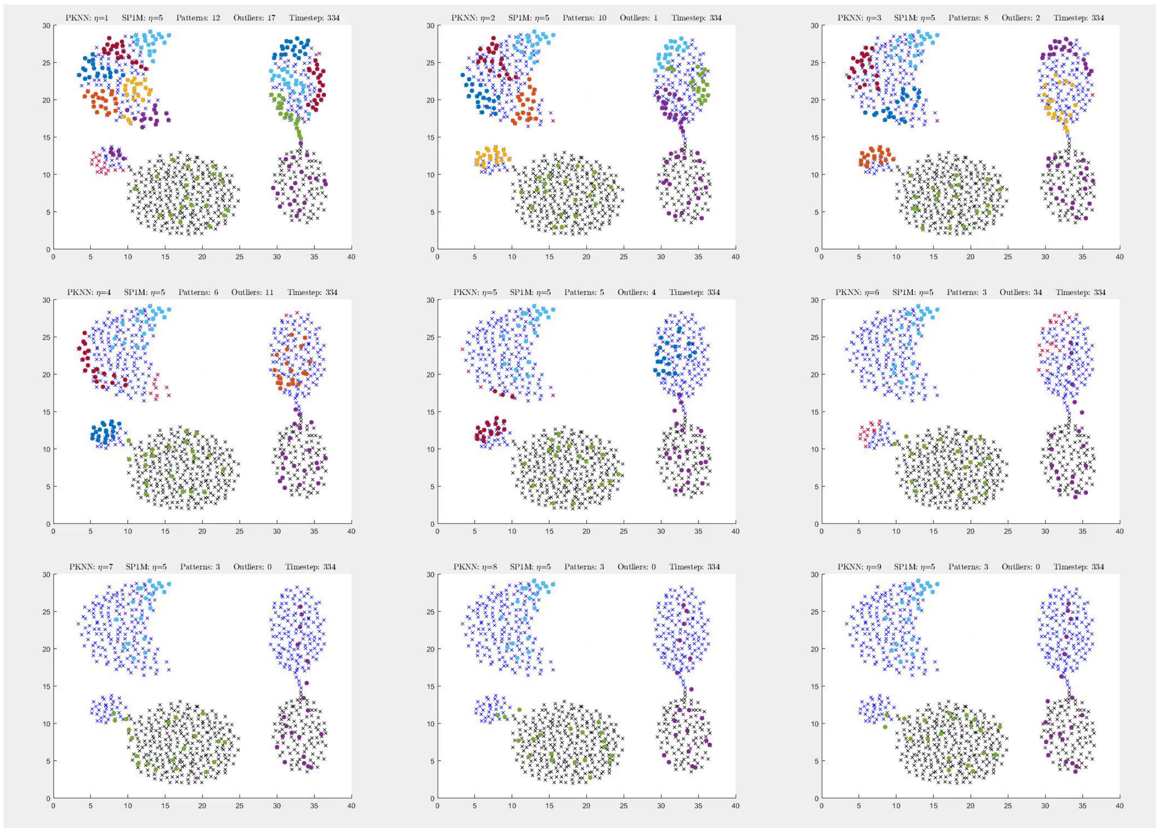
(a) Initialization set



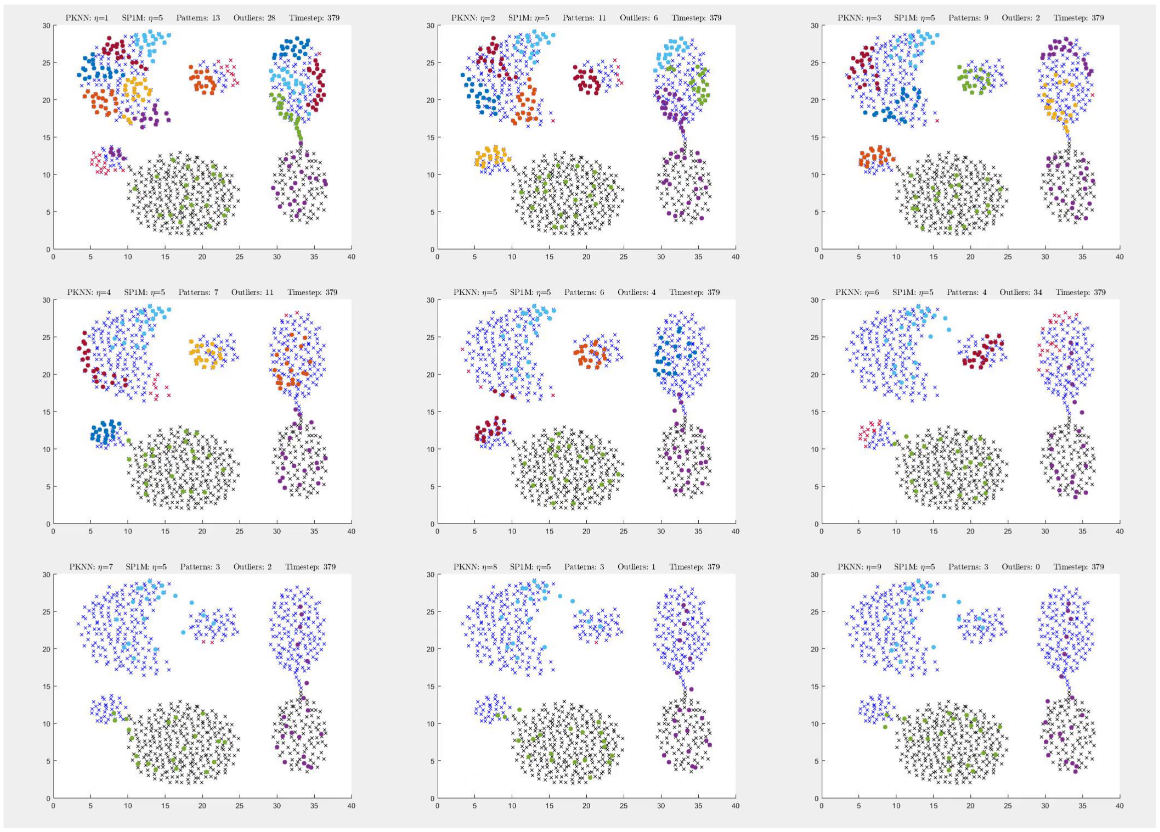
(b) After first cluster



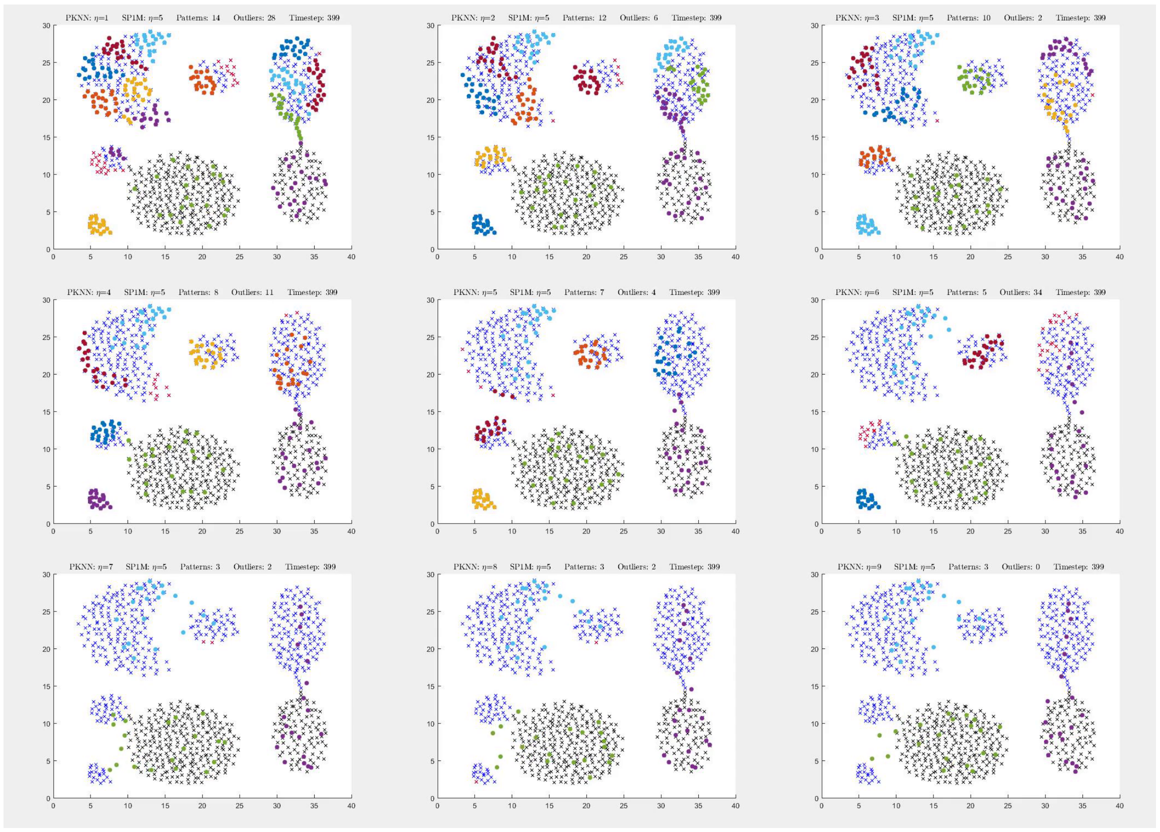
(c) After second cluster



(d) After third cluster



(e) After fourth cluster



(f) After fifth cluster

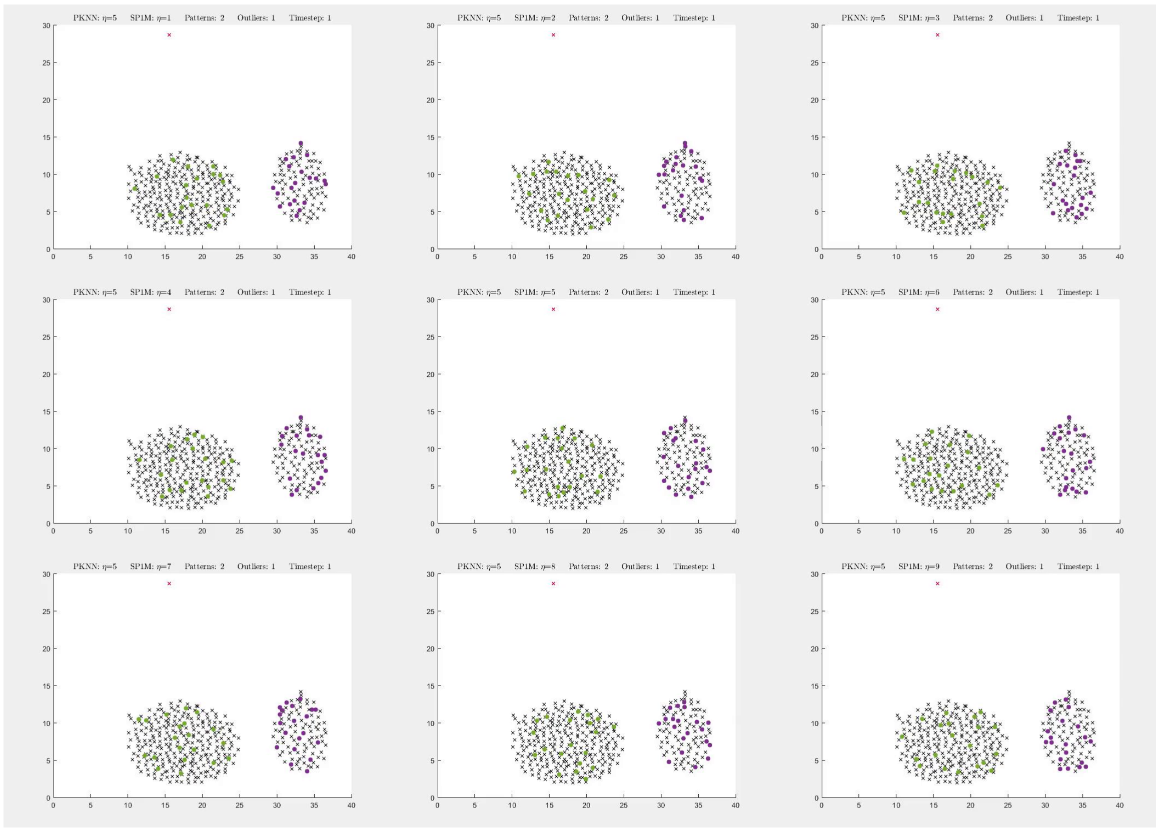
2. *Varying η in SP1M*

Similarly to the previous experiment, we are now varying the η used in SP1M to cluster the outlier list in StreamSoNG. There is an important distinction between these two η parameters (that of PKNN and that of SP1M), and it is not always the case that they should be equal. In this experiment, we vary the SP1M η from $\eta = 1, 2, \ldots, 9$ as we did before with the PKNN eta.

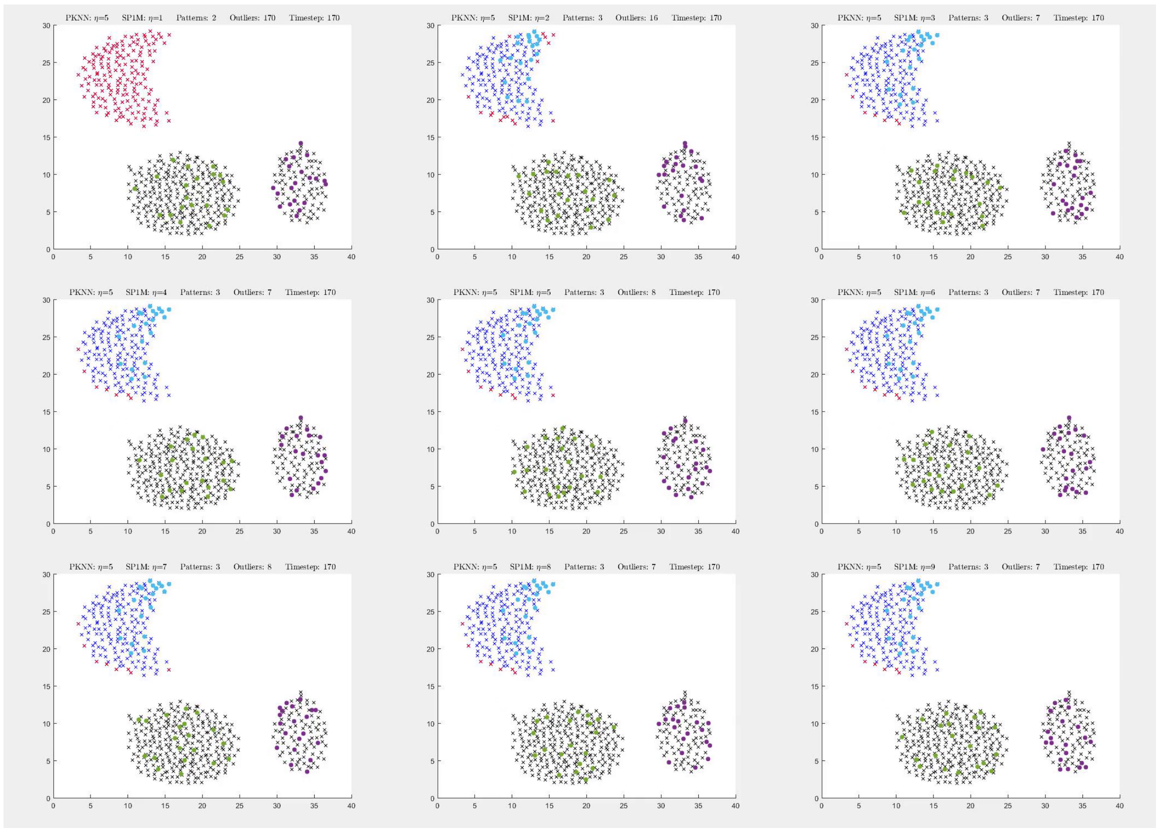
When the SP1M η is too small, no clusters will be found in the outlier list, and thus points that are initially flagged as outliers will stay outliers indefinitely. This also highlights a problem with StreamSoNG – choosing SP1M η sufficiently small will cause it to retain the entire data stream, which goes against the spirit of streaming data analysis.

For this dataset, any $\eta > 5$ produces almost identical final prototypes, though it should be noted that this phenomenon generally occurs only in datasets where the clusters are presented sequentially. In other datasets, having SP1M η too high would cause the entire outlier list to form a new class every time SP1M is run, regardless of the spatial proximity between the points in the outlier list.

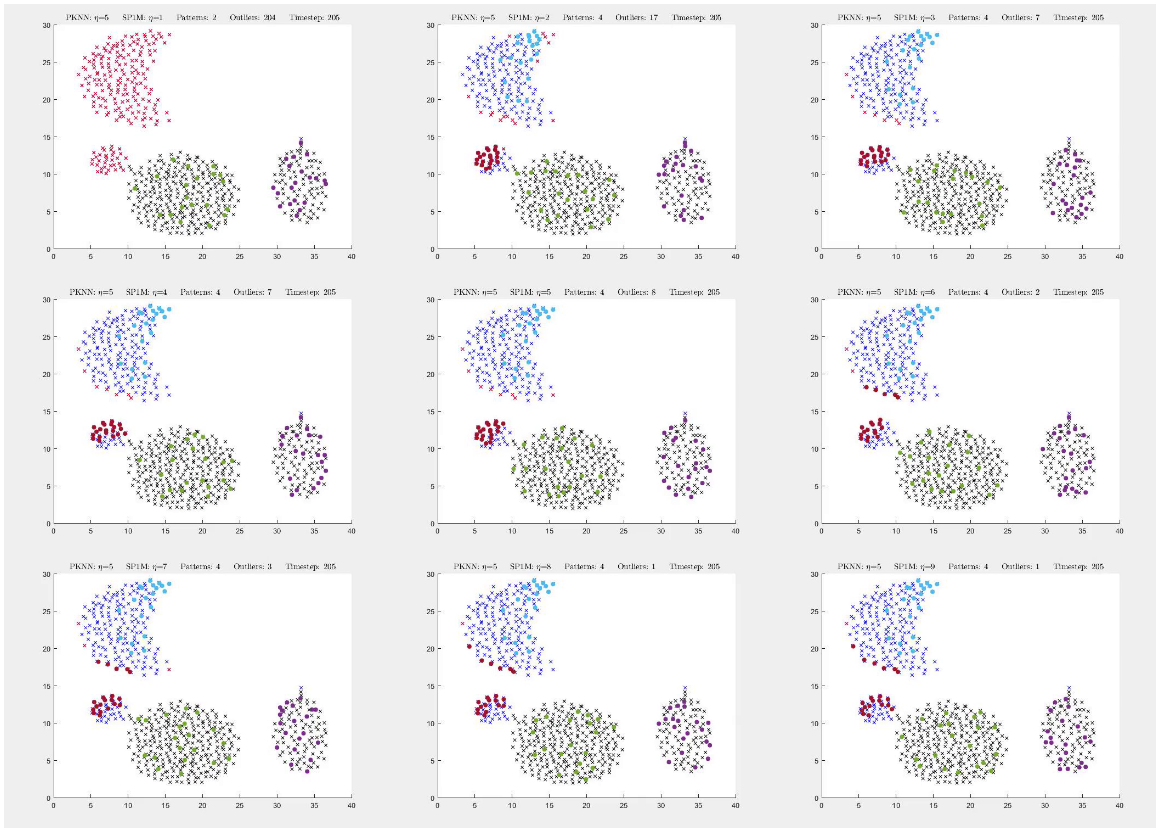
All other parameters are fixed at predetermined reasonable values for this dataset. An animated version of this figure can be viewed at `https://youtu.be/MUhITo0up_s`.



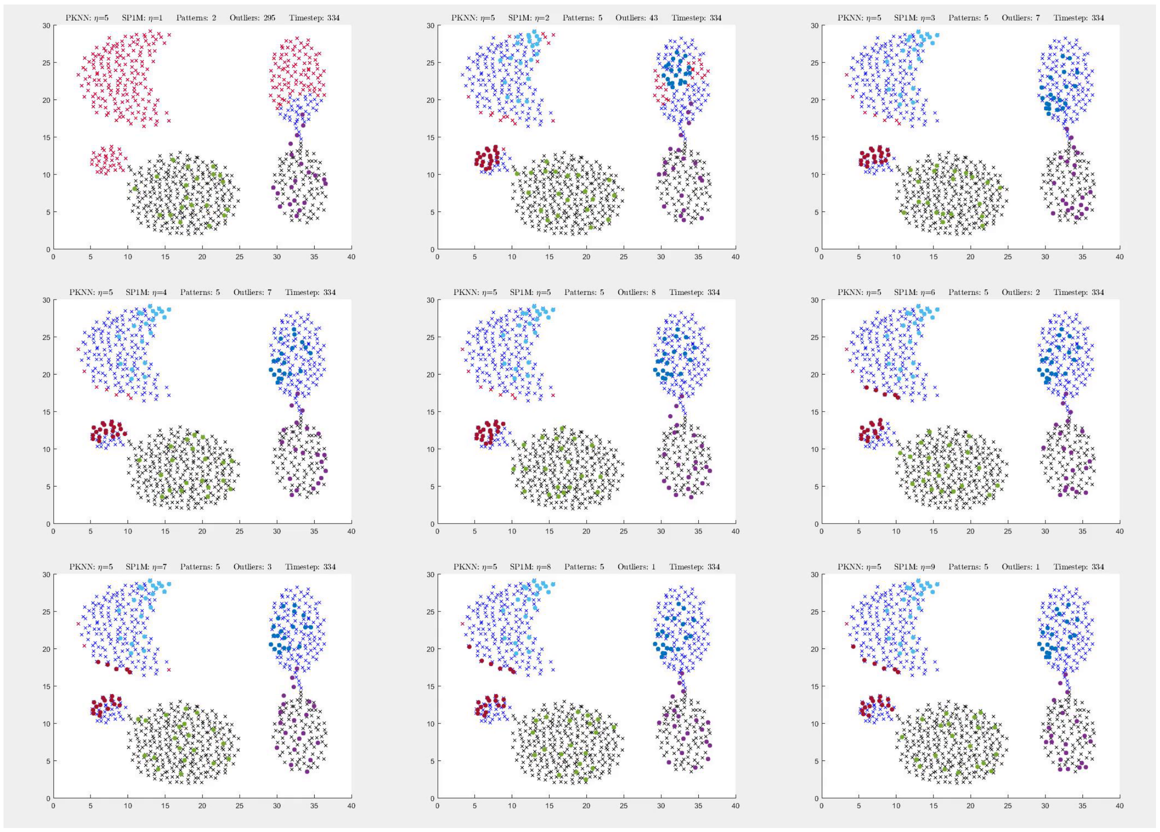
(a) Initialization set



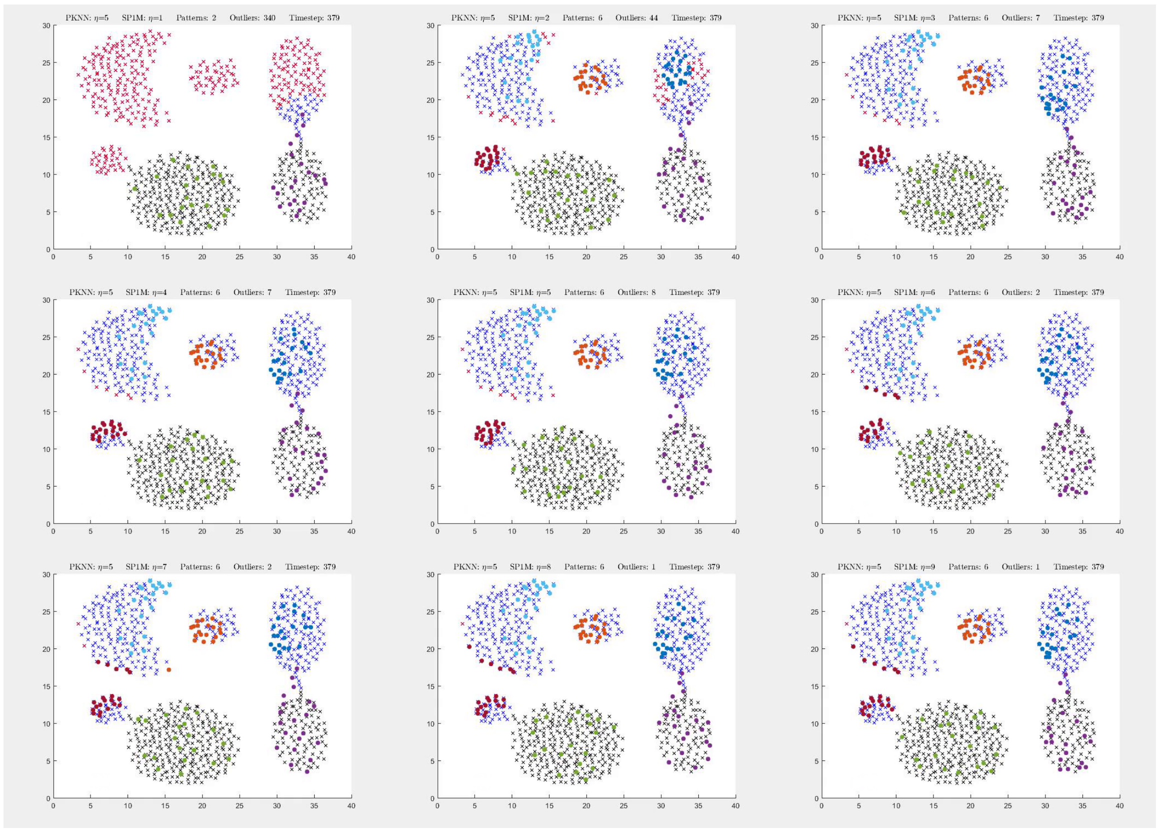
(b) After first cluster



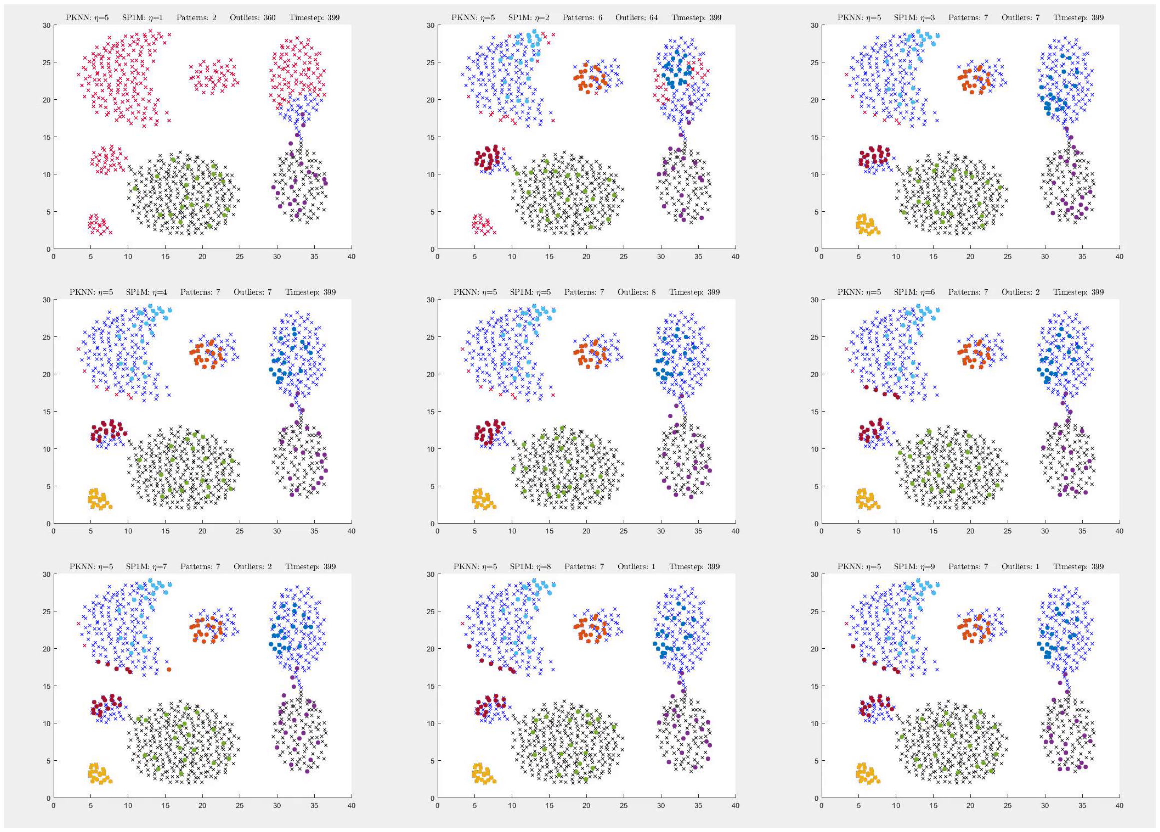
(c) After second cluster



(d) After third cluster



(e) After fourth cluster



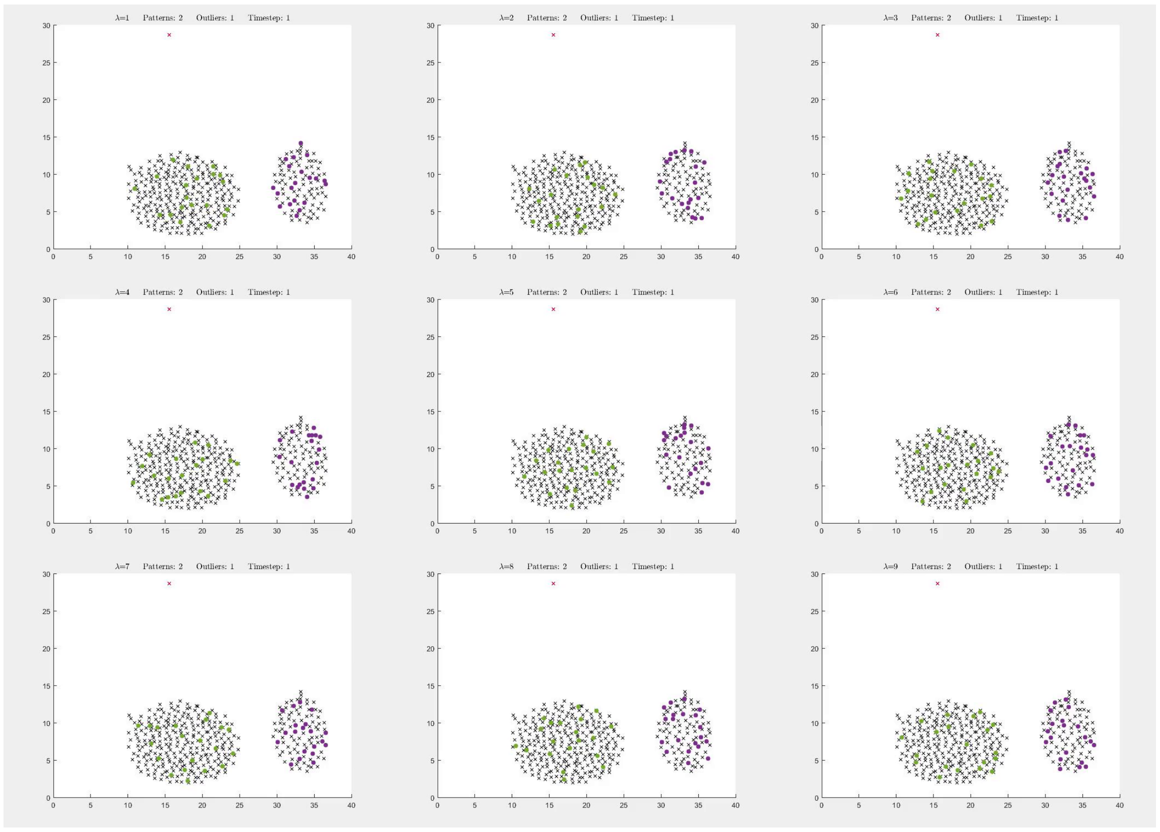
(f) After fifth cluster

3. *Varying λ in StreamSoNG*

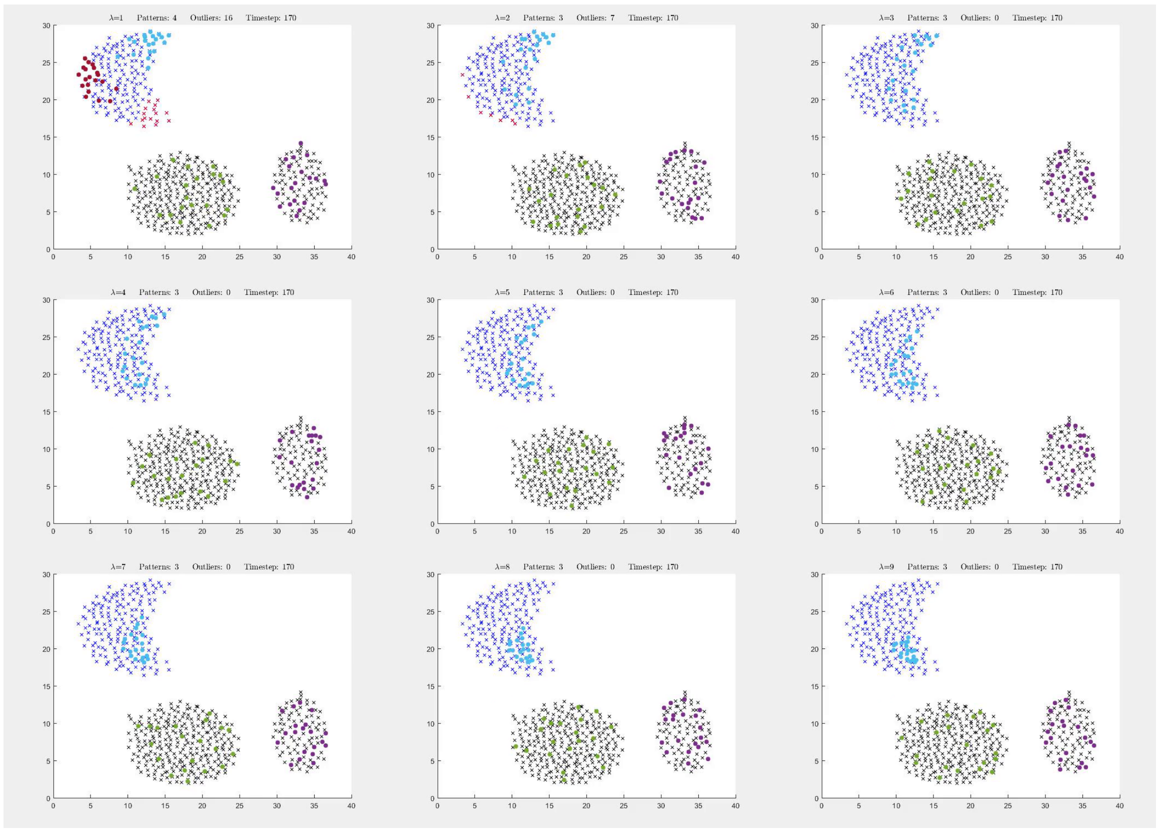
The λ parameter in StreamSoNG determines how many of its neural gas prototypes are updated, and by how much. In neural gas, the amount the k-th closest prototype to an input moves toward it is scaled by the term $\exp\{-k/\lambda\}$. Thus, large values of λ make this term non-negligible for higher values of k as well as making the term larger for all values of k.

We vary λ from $\lambda = 1, 2, \ldots, 9$ as we did the other parameters, and observe that $\lambda = 2$ seems to produce the best final prototypes. When λ is too small, the prototypes don't move fast enough to keep up with the data and new classes are created when the are not necessary. When λ is too large, all of the prototypes move toward new inputs, causing older points in the stream to be forgotten very quickly.

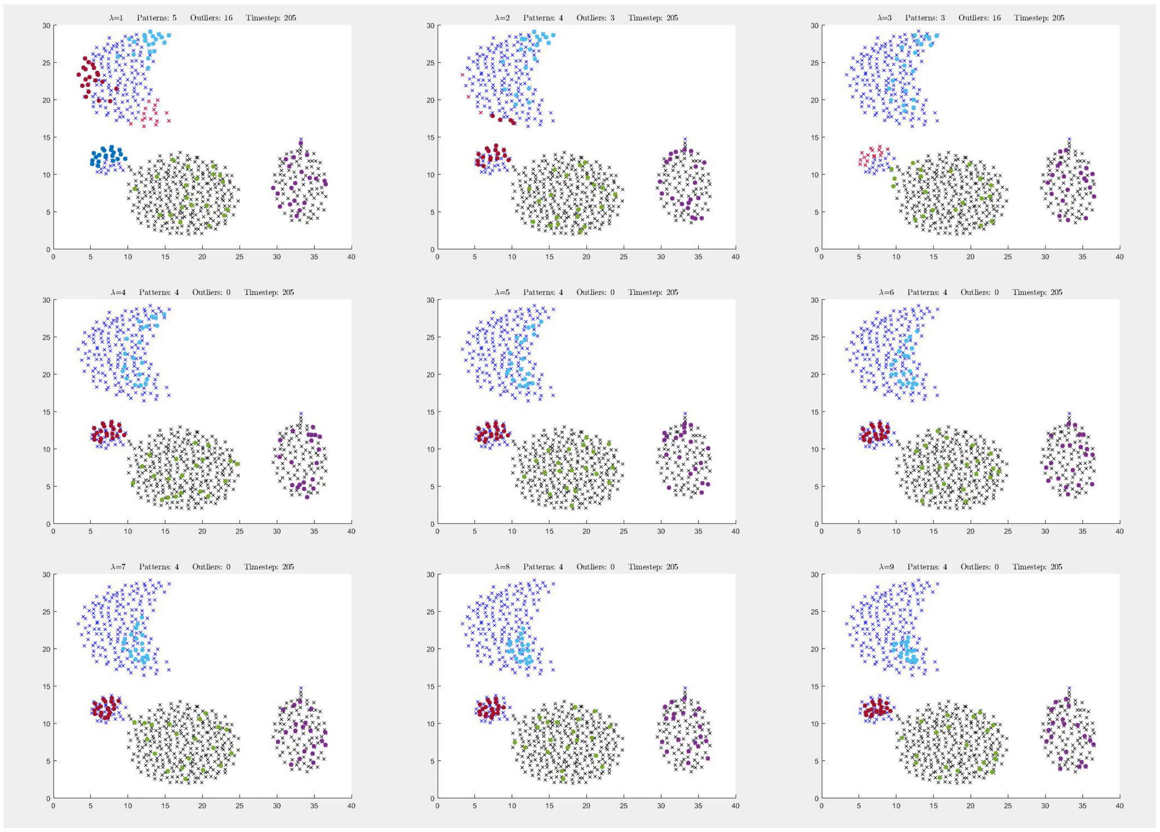
All other parameters are fixed at predetermined reasonable values for this dataset. An animated version of this figure can be viewed at `https://youtu.be/HlkScpt5YgQ`.



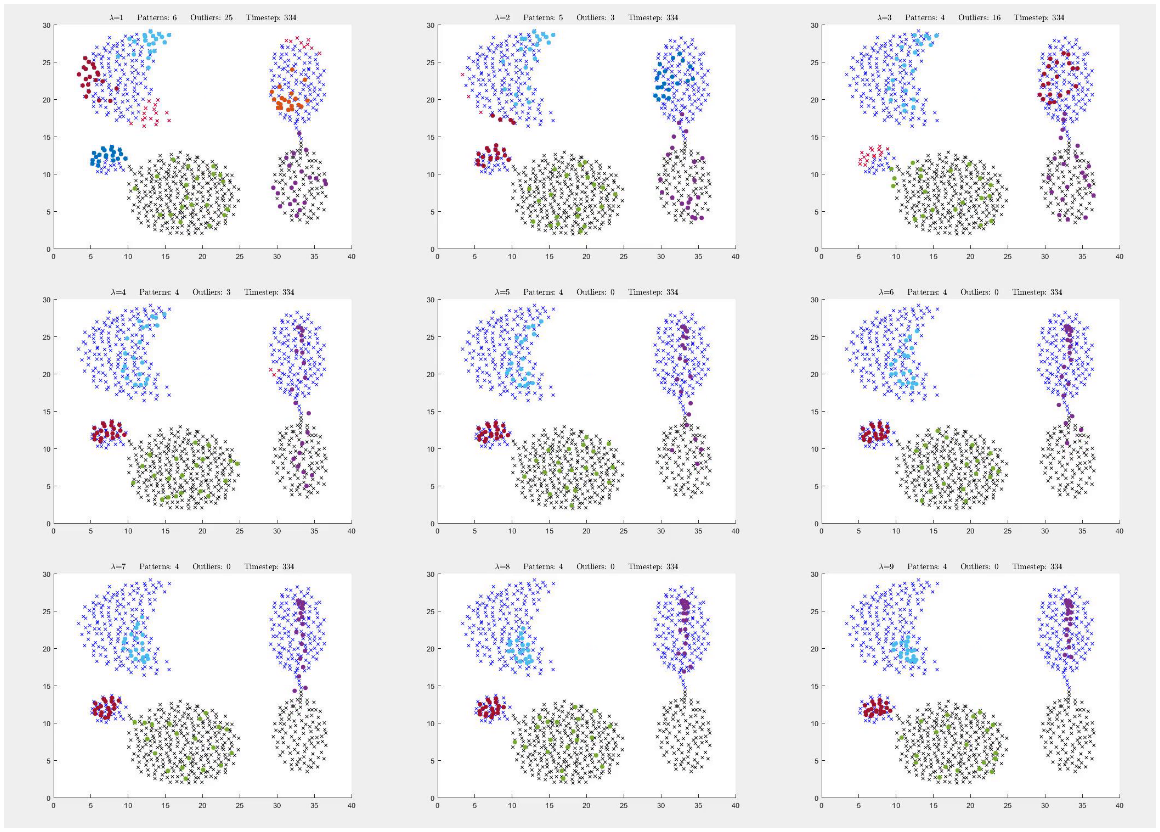
(a) Initialization set



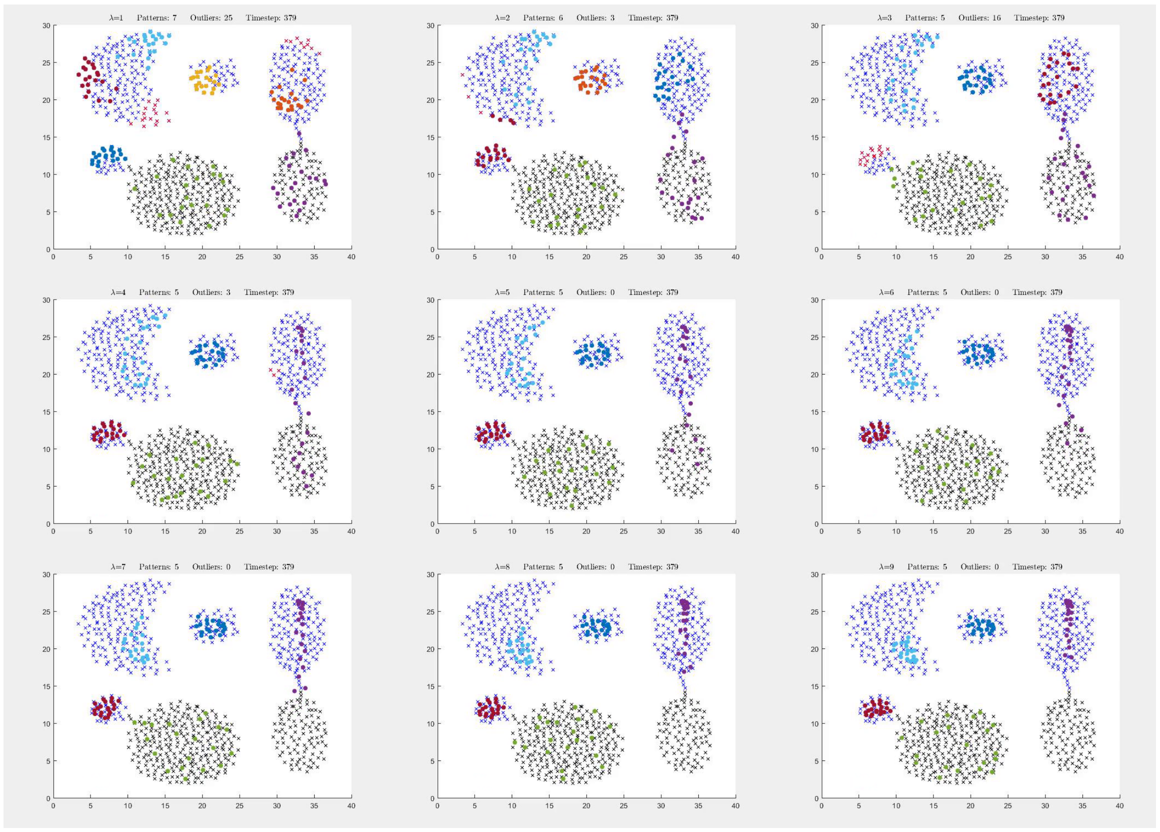
(b) After first cluster



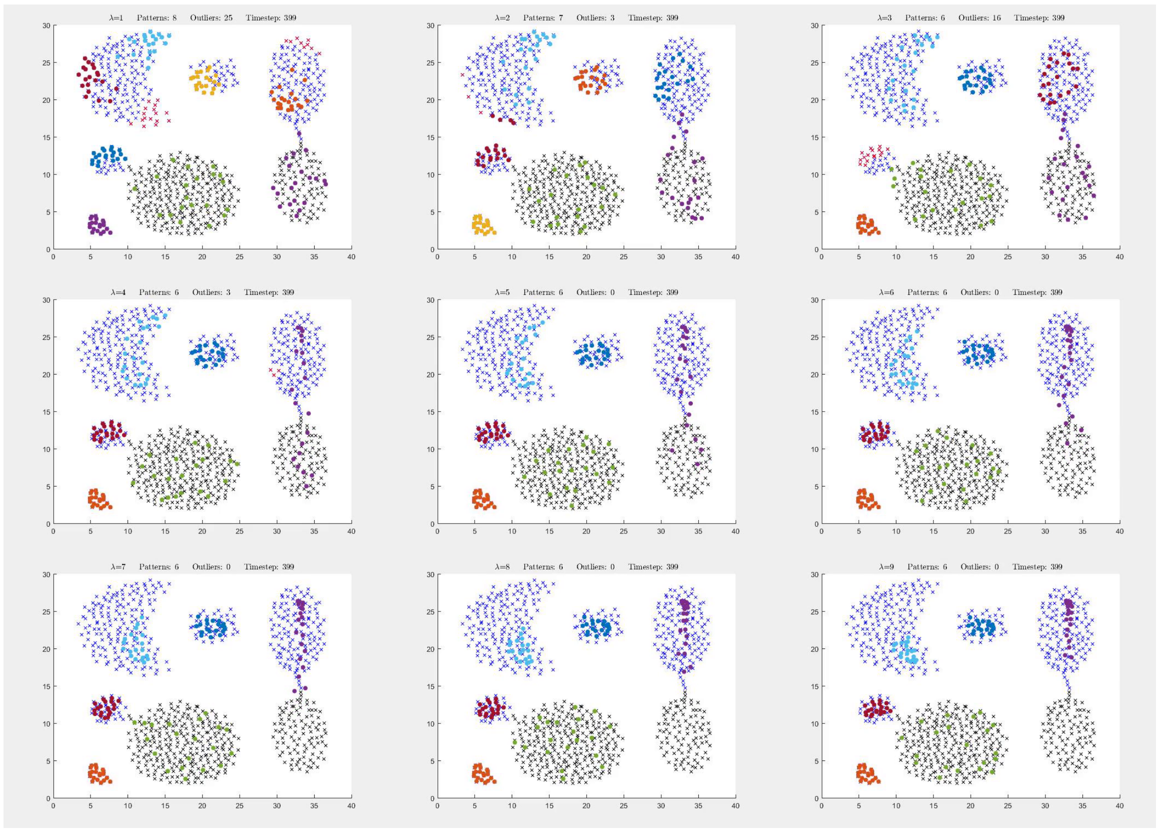
(c) After second cluster



(d) After third cluster



(e) After fourth cluster



(f) After fifth cluster

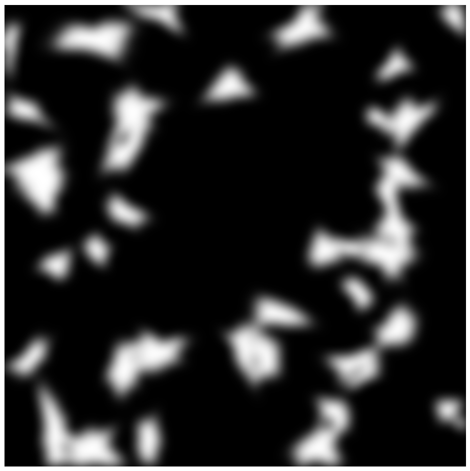
B. Realistic Texture Blending

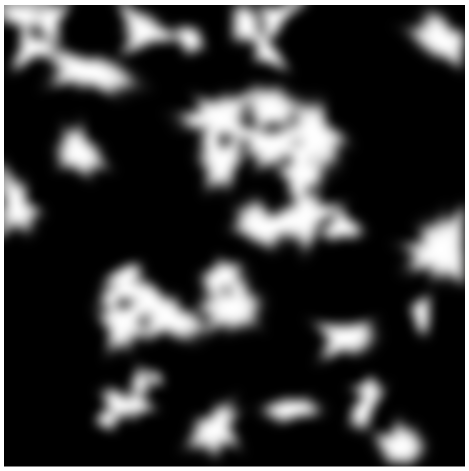
Ongoing work by Galusha et al. seeks to blend these texture primitives of Section V-B in a realistic way using unique masks for each primitive. Preliminary results of their process, which we hope to use for future iterations of the UE5 dataset, are demonstrated in Figure 37.

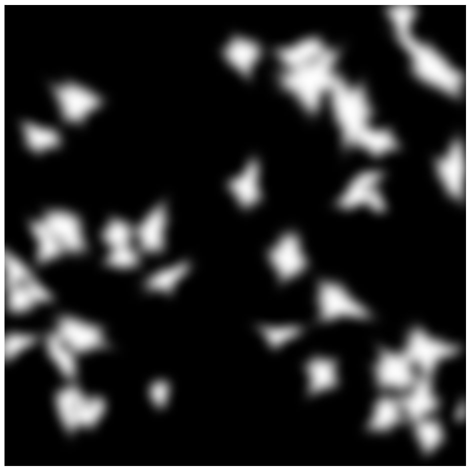
We want to be able to segment the environments produced by Galusha's blending method for use in texture recognition experiments, which is much easier said than done in UE5. Unlike when creating target segmentation labels, postprocess materials do not immediately lend themselves to the task of material segmentation. We have identified three methods for extracting pixelwise texture labels at runtime, all of which have strengths and weaknesses depending on the situation. These methods are described in the following sections.

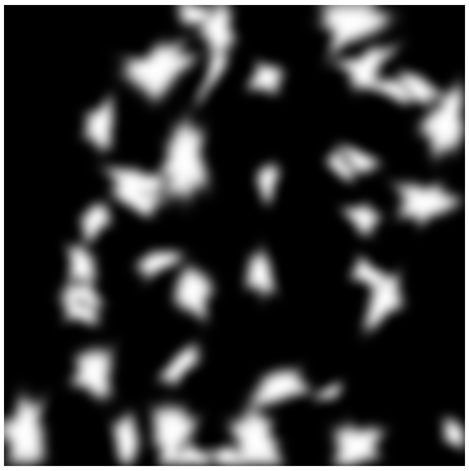
1. *Perlin Noise*

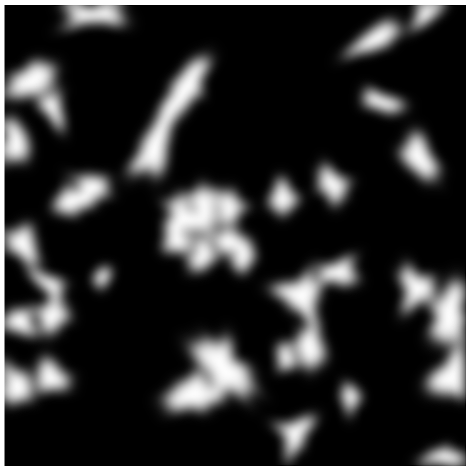
Our first method for blending textures is done using procedurally generated Perlin noise [71] inside of the material. As there doesn't seem to be an intentional mechanism in Unreal Engine for sending this information to the postprocess, we instead commandeer the unused metallic channel of the material. The output of the noise generation is multiplied by 0.1 before being sent to the "metallic" channel to decrease its effect on the landscape, and then recovered in the postprocess by di-


(a) Stone mask


(b) Plant mask


(c) Sand ripple mask


(d) Leaves mask


(e) Roots mask

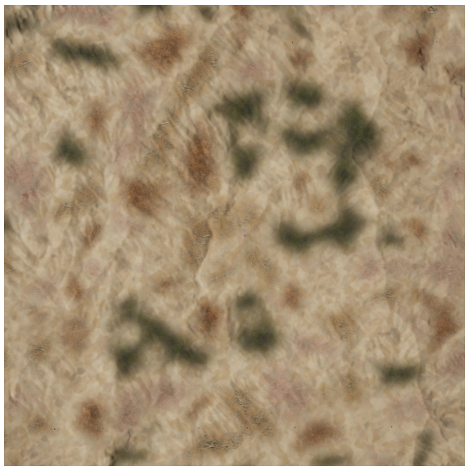
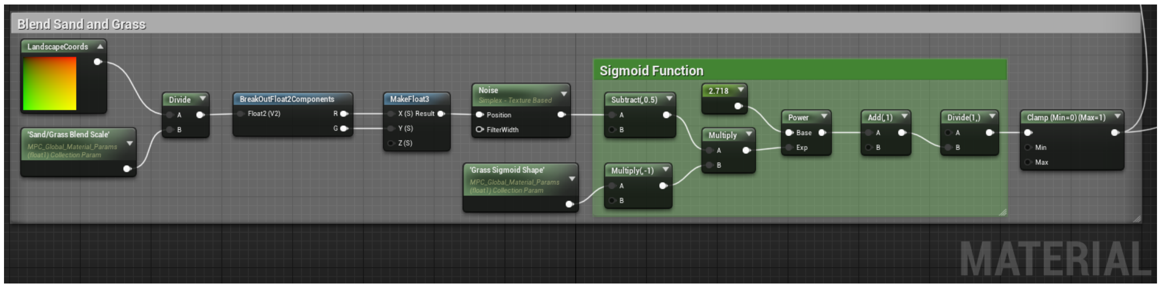
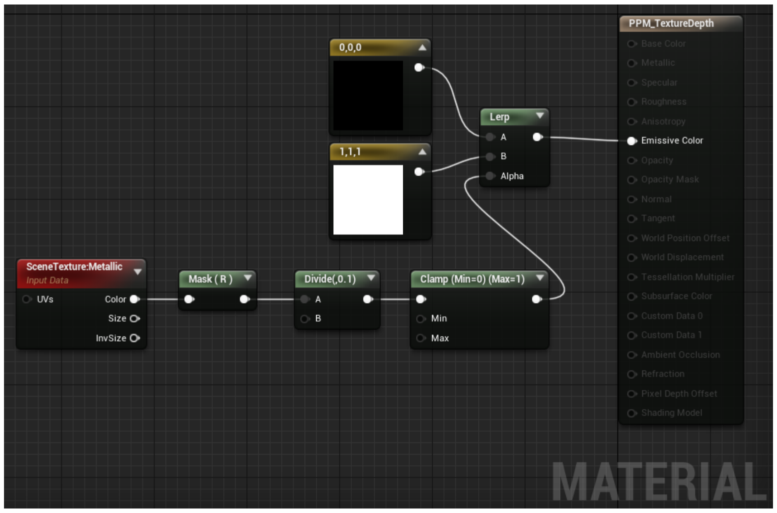

(f) Rendered scene

Figure 37: (a)-(e): The weight of each texture primitive in Figure 22, with the exception of flat sand. The weight of flat sand at a pixel is given by one minus the sum of the other texture weights, set equal to zero if negative. (f): The scene with texture painted according to the masks.



(a) Noise generation process for blending textures. Output of clamp node is multiplied by a small number (0.1 in this case) and sent to "Metallic" output node of material.



(b) Extracting blending information in the postprocess.

Figure 38: Method to send texture blending information to the postprocess in order to create segmentation maps.

viding the input metallic channel by the same value of 0.1 before using the information. While the act of multiplying and dividing by 0.1 are mathematically inverses of one another, this doesn't necessary hold in floating point arithmetic. As future work, we wish to find a cleaner and more numerically stable way to acquire the texture segmentation in the postprocess.

2. *Changing Landscape Materials*

A big limitation of the Perlin noise is its two-class nature, which makes it unusable for our purposes. There is no straightforward way to blend more than two textures together with Perlin noise, which is why great effort was put forth by Galusha et al. to create soft, realistic partition labels. Galusha's method overcomes the limitations of Perlin noise, but at a cost of being very difficult to use as a postprocess material for ground truth labeling. To obtain texture labels with their method directly in UE5, the most effective approach we found was to simply replace the material on each landscape with the corresponding primitive mask, disable all lighting in UE5, and re-run th data collect. This is not perfect, however, as a number of routines in UE5 such as static lighting, dynamic lighting, ambient lighting, camera exposure, and more can play a role in distorting segmentation masks. Unfortunately, due to an engine limitation in UE5, we cannot cast shadows onto these landscapes because they use translucency.

3. *Manual Blending*

The third method we have used for producing texture labels, which has been the most useful to our research, is performing the texture blending entirely outside of UE5. In UE5, we run the exact same data collection once for each texture primitive, producing a large sequence of images containing only one unique texture in them. Doing so, we ensure that the simulated drone is in the exact same position on the n-th frame of each primitive's sequence of N images, $n = 1, 2, \ldots, N$.

Then, in a script outside of UE5, we read the sequence of exported images from each texture primitive simultaneously and manually determine where the camera's field of view would land within the high resolution texture blending masks (Algorithm 9). After extracting the correct region of the masks, we create a blended image by simply multiplying the pixel intensities in each texture primitive by the weights of that primitive from the mask.

4. *Collected Dataset*

A preliminary render of this blended dataset has yielded 6,297 images collected at 720×720 resolution. The size of this dataset on disk is around 4.77 GB. Examining the total membership in each texture class among all pixels in all images, we find that roughly 15% of the total membership in the dataset belongs to each non-background texture (plants, sand ripple, leafy, roots, stone) and the remaining 25% belongs to the background texture (flat sand).

A video showing each frame from the full dataset, with pixelwise segmentation labels, can be viewed on YouTube[6]. Selected frames from

[6]`https://www.youtube.com/watch?v=5jC9C2DQcbc`

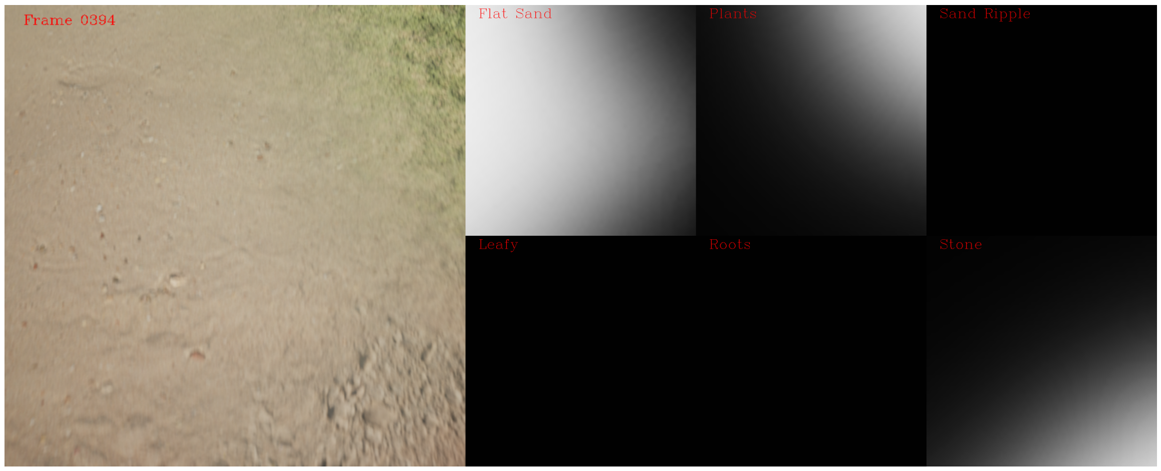
Algorithm 9 Locating the camera's field of view within a texture blending mask.

```
procedure GetMask(index, L, W, stride)
    // index, number of frames since start of collect
    // (L, W), number of frames in length/width of route
    // stride, number of pixels moved by drone per frame

    // See how many loops were completed
    loop_ix = floor(index / (2 * (L + W)))

    // See how far we are into the current loop
    loop_offset = mod(index, (2 * (L + W)))

    // Forward phase
    if loop_offset < L then
        rotate_amount = 0°
        bottom_left_x = 2 * loop_ix * W
        bottom_left_y = loop_offset

    // First right phase
    else if loop_offset < L + W then
        rotate_amount = 90°
        bottom_left_x = 2 * loop_ix * W + (loop_offset - L)
        bottom_left_y = L

    // Backward phase
    else if loop_offset < 2*L + W then
        rotate_amount = 180°
        bottom_left_x = (2 * loop_ix + 1) * W
        bottom_left_y = L - (loop_offset - L - W)

    // Second right phase
    else
        rotate_amount = 270°
        bottom_left_x = (2*loop_ix + 1) * W + (loop_offset - (2*L + W))
        bottom_left_y = 0

    bottom_left_x = bottom_left_x * stride
    bottom_left_y = bottom_left_y * stride

    return (bottom_left_x, bottom_left_y, rotation_amount)
```

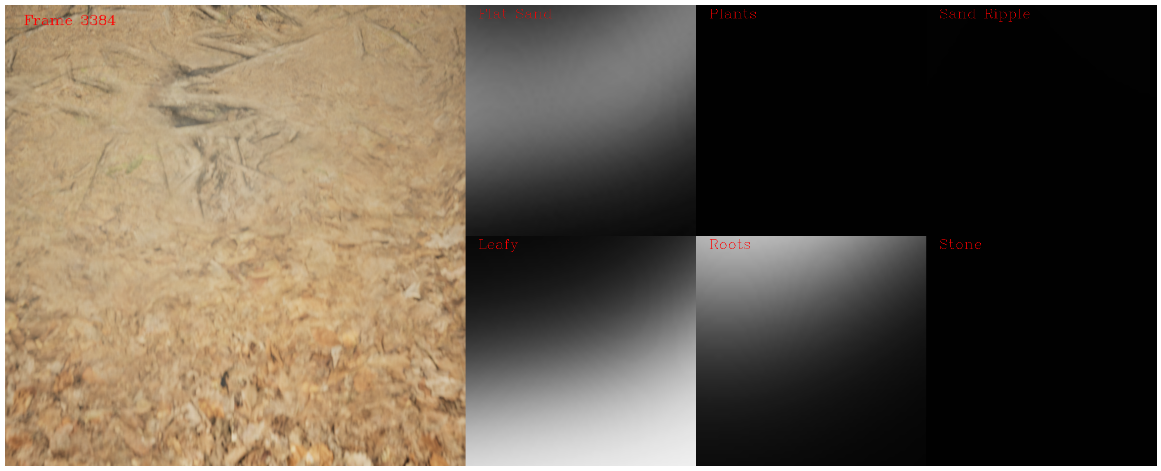
this video are shown in Figure 39. The large leftmost image is the image from the dataset and the 2×3 grid of grayscale images on the right are the memberships of each pixel of the image in each of the six texture primitives. For the purpose of visualization, the frame number in the video is drawn to the upper left corner of the image and the name of the primitive is drawn on the upper left of each segmentation image. To be clear, this text is not present in the actual dataset.

5. *Analysis*

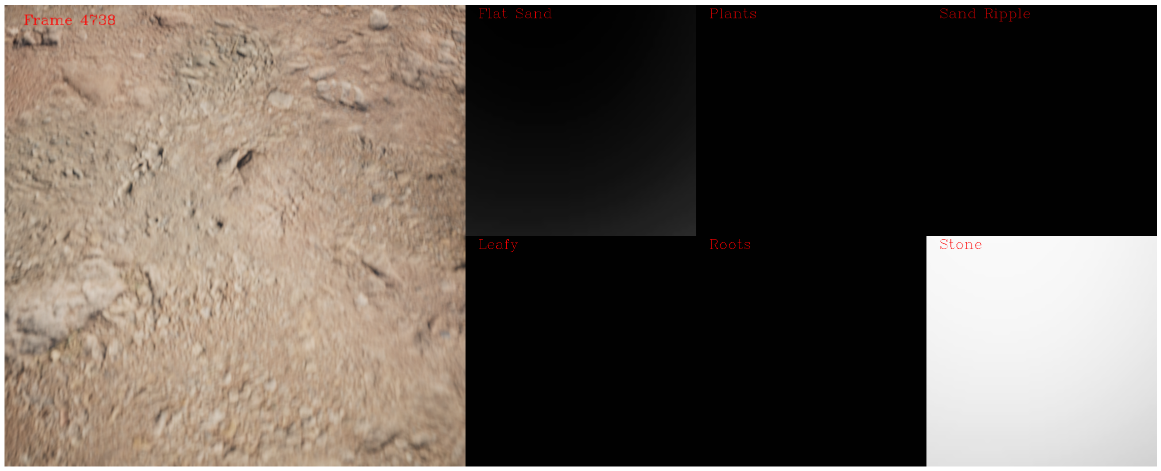
After computing features on the blended UE5 dataset using VGG-16 as discussed in Section V-F, we have produced the t-SNE plot in Figure 40. We observe that each class in the dataset appears to be grouped together in feature space, though classes tend to bleed together through a series of points in the background class ("Flat Sand"). For comparison, the VGG-16 features of the KTH-TIPS2b dataset [68] shown in Figure 41 appear to be very separable. This makes sense, as one of the primary objectives in constructing the UE5 dataset was to model realistic and gradual changes from one texture to another. As a result, this dataset presents a challenge for algorithms that rely on density-based clustering.



(a) Transition between stone and plant primitives



(b) Transition between leafy and roots primitives



(c) Mostly pure example of stone primitive

Figure 39: Example frames from dataset with segmentation.

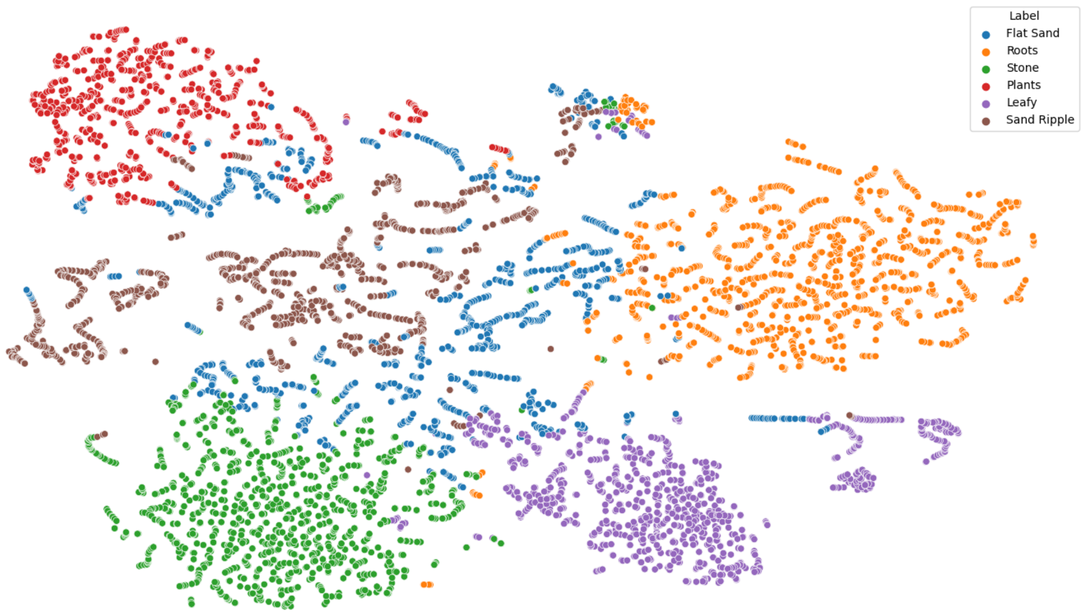


Figure 40: *t*-SNE plot of 4096-dimensional output of the VGG-16 feature extractor when run on the UE5 dataset.

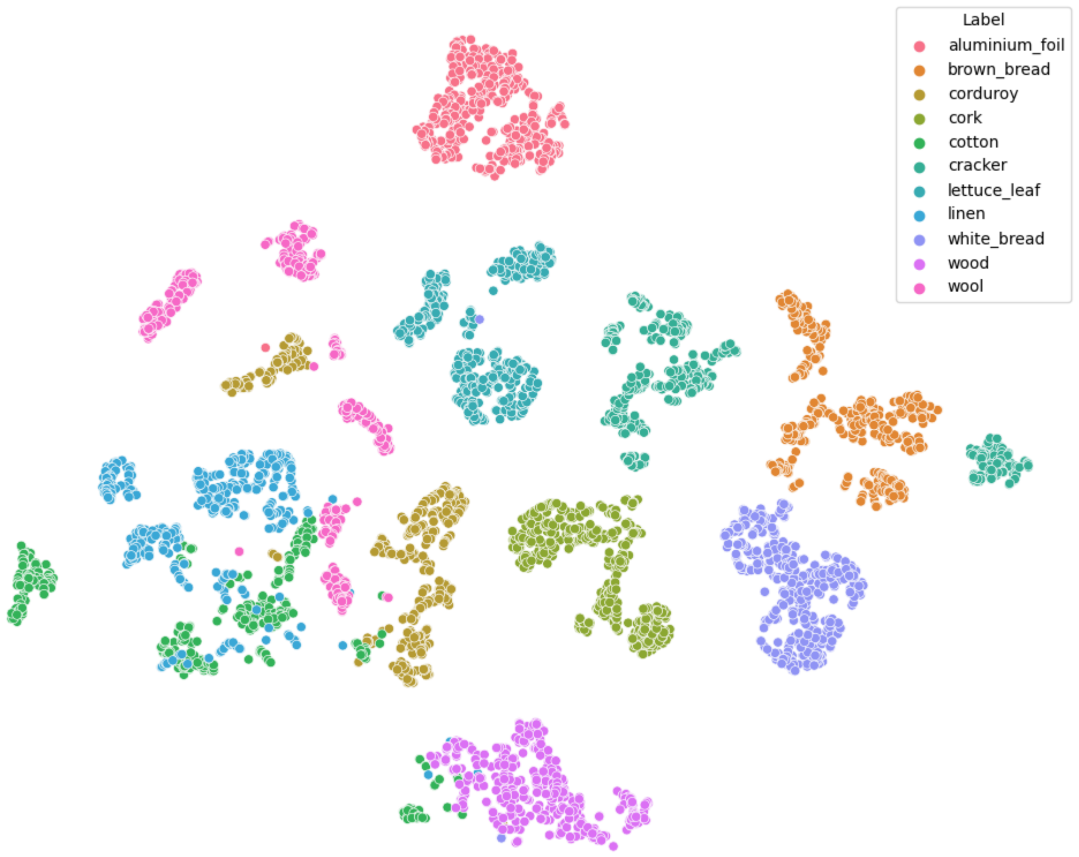


Figure 41: t-SNE plot of 4096-dimensional features from VGG-16 on the KTH-TIPS2b dataset.

C. Clustering Evaluation Metrics

Throughout these experiments, we use two external cluster validation indices to evaluate the performance of SDA algorithms: purity [61] and normalized mutual information (NMI) [64]. In this section, we will describe these metrics in more detail.

1. *Cluster Purity*

Purity [61], as the name suggests, reflects the degree to which each discovered cluster in the data is "pure" with respect to a ground truth partition. When each discovered cluster contains only points from a single ground truth cluster, purity attains its optimal value of one. To compute purity, one uses the following equation:

$$\text{Purity}(C, Y) = \frac{1}{N} \sum_{k=1}^{|C|} \max_{j=1,\dots,|Y|} |C_k \cap Y_j| \tag{38}$$

where C is the set of discovered clusters and Y is the set of ground truth clusters. We observe that purity can be optimal in some cases where the partitions C and Y are different, for example, when every point is assigned to its own cluster. For this reason, we include additional metrics on which to base our performance evaluation.

2. *Normalized Mutual Information*

Normalized mutual information (NMI) [64] is another external cluster validity index used in experiments throughout this work. Unlike purity, NMI is only optimized when the two partitions C and Y are identical. NMI is computed according to

$$\mathrm{NMI}(C,Y) = \frac{2 \cdot I(C,Y)}{H(C)+H(Y)} \tag{39}$$
$$= \frac{2(H(C)-H(C|Y))}{H(C)+H(Y)} \tag{40}$$

where H is the entropy of the clustering, given by

$$H(X) = -\sum_{k=1}^{|X|} P(X_k)\log P(X_k). \tag{41}$$

To compute $P(X_k)$, one simply divides the number of points in cluster X_k by the number of points in the dataset. From Equation 40, we see that when $C = Y$, the value of NMI is one since $H(C|Y) = 0$ in this case. Similarly, when the mutual information of C and Y is zero, NMI is zero.

Vita

Jeffrey Dale was born in Missouri and attended the Ozark school district until he graduated high school in 2014. He then went to Missouri State University from 2014-2018 where he received B.S. degrees in Computer Science and Mathematics. After that, he began graduate school at the University of Missouri in 2018, obtaining M.S. and Ph.D degrees in Computer Science in 2019 and 2022, respectively.