2017 IEEE International Conference on Data Mining Workshops

Accelerated Hierarchical Density Based Clustering

Leland McInnes and John Healy

***Abstract*—We present an accelerated algorithm for hierarchical density based clustering. Our new algorithm improves upon HDBSCAN*, which itself provided a significant qualitative improvement over the popular DBSCAN algorithm. The accelerated HDBSCAN* algorithm provides comparable performance to DBSCAN, while supporting variable density clusters, and eliminating the need for the difficult to tune distance scale parameter ϵ. This makes accelerated HDBSCAN* the default choice for density based clustering.**

I. Introduction

Clustering is the attempt to group data in a way that meets with human intuition. Unfortunately, our intuitive ideas of what makes a 'cluster' are poorly defined and highly context sensitive [1]. This results in a plethora of clustering algorithms each of which matches a slightly different intuitive notion of what a natural grouping is.

Despite the uncertainty underlying the clustering process it continues to be used in a multitude of scientific domains. The fundamental problem of finding groupings is pervasive and results, however poor, are still important and informative. It is used in diverse fields such as molecular dynamics [2], airplane flight path analysis [3], crystallography [4], and social analytics [5], among many others.

While clustering has many uses to many people, our particular focus is on clustering for the purpose of exploratory data analysis. By exploratory data analysis we mean the process of looking for "interesting patterns" in a data set, primarily with the goal of generating new hypotheses or research questions about the data set in question. This necessitates minimal parameter selection and few apriori assumptions about the data. In this use case, it is highly desirable that solutions have informative failure modes. Specifically, when data is poorly clustered or does not contain clusters, it is necessary to have some indication of this from the clustering algorithm itself.

Many traditional clustering algorithms are poorly suited to exploratory data analysis tasks. In particular, most clustering algorithms suffer from the problems of difficult parameter selection, insufficient robustness to noise in the data, and distributional assumptions about the clusters themselves.

Many algorithms require the selection of the number of clusters, either explicitly, or implicitly through proxy parameters. In the majority of use cases we have encountered, selecting the number of clusters is very difficult apriori. Methods to determine the number of clusters such as the elbow method and silhouette method are often subjective and can be hard to apply in practice. Ultimately these methods all hinge on the clustering quality measure chosen; these are diverse and often highly related with particular clustering algorithms [1].

Many practitioners fail to distinguish between partitioning and clustering to the point where the terms are now often used interchangeably. By clustering we specifically mean finding subsets of the data which group "naturally", without necessarily assigning a cluster for all points. Partitioning, on the other hand, requires that every data point be associated with a particular cluster. In the presence of noise the partitioning approach can be problematic. Even without noise, if clear clusters are not present, partitioning will simply return a poor solution.

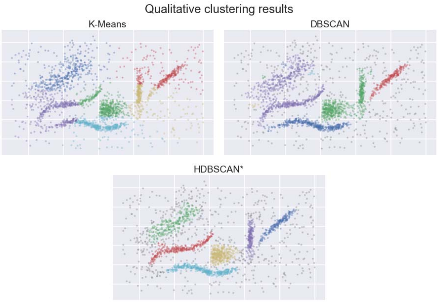


Fig. 1: A qualitative comparison of some candidate clustering algorithms on synthetic data. Colors indicate cluster membership (grey denotes noise). We advocate density based clustering methods when performing exploratory data analysis as they require fewer assumptions about the data distribution, and can refuse to cluster points. Unclustered "noise" points for both DBSCAN and HDBSCAN* are depicted in gray. The above clustering results represent the result of a qualitative hand tuned search for optimal parameters.[1]

Distributional assumptions on the data are difficult to make in exploratory data analysis. As a result we examine density based clustering since it has few implicit assumptions about the distribution of clusters within the data. Among density based clustering techniques DBSCAN [6] is attractive in that it is efficient and is robust to the presence of noise within data. Its primary difficulties include parameter selection and the handling of variable density clusters. In [7] and [8] Campello, et al. propose the HDBSCAN* algorithm which addresses both of these problems, but its major difficulty is that it sacrifices performance to do so.

In Figure 1 we compare three candidate clustering algorithms: K-Means, DBSCAN, and HDBSCAN*. The archetypal clustering algorithm, K-Means, suffers from all three of the problems mentioned previously: requiring the selection of the number of clusters; partitioning the data, and hence assigning noise to clusters; and the implicit assumption that clusters have Gaussian distributions. In comparison, being a density based approach, DBSCAN only suffers from the difficulty of parameter selection. Finally HDBSCAN* resolves many of the difficulties in parameter selection by requiring only a small set of intuitive and fairly robust parameters.

Section II introduces the HDBSCAN* algorithm. We provide two different descriptions of the algorithm: the first is a new description of HDBSCAN* that builds upon Chauduri et al. [9], [10] and Stuetzle et al. [11], [12], viewing the algorithm as a statistically motivated extension of Single Linkage clustering; the second description follows Campello et al. [7], [8], viewing the algorithm as a natural hierarchical extension of the popular DBSCAN algorithm. Both the statistical and computational descriptions of HDBSCAN* have been published before, but the equivalence has rarely been made explicit.

The major contribution of this paper is section III, which describes a new algorithm for computing HDBSCAN* clustering results. This new algorithm, making use of data structures and techniques from March et al. [13] and Curtin et al. [14], [15], offers significant improvements in average case asymptotic performance.

In section IV we compare the performance of our new HDBSCAN* algorithm against other clustering algorithms. In particular, we demonstrate the asymptotic performance improvement over the reference HDBSCAN* algorithm, and show our new algorithm provides HDBSCAN* with comparable asymptotic performance to DBSCAN, one of the fastest extant clustering algorithms.

[1] See https://github.com/lmcinnes/hdbscan_paper/Qualitative%20clustering%20results.ipynb for code used to generate these plots

2375-9259/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDMW.2017.12

II. HDBSCAN* Explained Two Ways

Algorithms like HDBSCAN* lie at the convergence of several lines of research from different fields. To highlight this convergence we will describe the HDBSCAN* algorithm from two different perspectives: from a statistically motivated point of view; and with a computationally motivated mindset. Through this repetition we hope to both provide a sound introduction to how the algorithm works, and to place it in a richer context of ideas. We also hope that the explanation that is less familiar will become easier to follow by analogy.

A. Statistically Motivated HDBSCAN*

A statistically oriented view of density clustering begins with the assumption that there exists some unknown density function from which the observed data is drawn. From the density function f, defined on a metric space $(\mathcal{X}, d)$, one can construct a hierarchical cluster structure, where a cluster is a connected subset of an f-level set $\{x \in (\mathcal{X}, d) \mid f(x) \geq \lambda\}$. As $\lambda \geq 0$ varies these f-level sets nest in such a way as to construct an infinite tree, which is referred to as the *cluster tree* (see figure 2 for an example). Each cluster is a branch of this tree, extending over the range of λ values for which it is distinct. The goal of a clustering algorithm is to suitably approximate the cluster tree, converging to it in the limit of infinite observed data points.

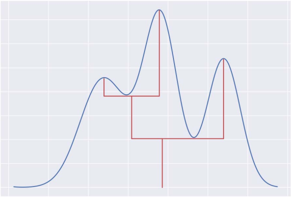


Fig. 2: The cluster tree (red) induced by a density function (blue).

This idea dates back at least to Hartigan [16], and has become an increasingly popular way to frame the clustering problem; see [17], [18], [11], [12] and [19] for examples. Our description of HDBSCAN* in these terms follows Chaudhuri et al. [9], [10] and their description of Robust Single Linkage.

The motivation for the approach is based on Hartigan's work on consistency results for single linkage clustering [16]. Hartigan's results while impressive, only apply to one dimensional data. The commonly cited drawback of single linkage clustering is that it is not robust to noise and suffers from chaining effects (spurious points merging clusters prematurely) [20], [21]. Wishart proposed a heuristic algorithm as a potential solution to this in [21]. The Robust Single Linkage algorithm [9], [10] extends Wishart's basic approach, and provides suitable theoretical underpinnings.

The Robust Single Linkage algorithm assumes that the data set

$$X = \{X_1, X_2, \ldots, X_N\}$$

is sampled from an unknown density f on some metric space $(\mathcal{X}, d)$. We then define $B(X_i, \varepsilon)$ to be the open ball centered at X_i of radius ε in $(\mathcal{X}, d)$. The algorithm takes two inputs, k and α. For each $X_i \in X$ define

$$r_k(X_i) = \inf\{\varepsilon \mid B(X_i, \varepsilon) \text{ contains } k \text{ points}\}.$$

For each $\varepsilon \geq 0$ define a graph G_ε with vertices $\{X_i \in X \mid r_k(X_i) \leq \varepsilon\}$ and an edge (X_i, X_j) if $d(X_i, X_j) \leq \alpha\varepsilon$. Define the clusters at level ε of the tree to be the connected components of G_ε.

In [9] and [10] Chaudhuri et al. provide a number of results on the consistency and convergence of this algorithm in Euclidean space ($\mathbb{R}^d$) for $k \sim d \log N$ with $\sqrt{2} \leq \alpha \leq 2$. Eldridge et al. [22] provide even stronger consistency results by introducing stricter notions of consistency. This provides a sound statistical basis for the approach.

A remaining issue with this algorithm is that the resulting cluster tree with N leaves is highly complex, making analysis difficult for large data set sizes. This is, of course, an issue faced by many hierarchical clustering algorithms. Several authors, including Stuetzle et al. [11], [12], and Chaudhuri et al. [10] have proposed approaches to pruning the cluster tree to simplify presentation and analysis. While Chaudhuri et al. provide consistency guarantees for their approach, we find the required parameters to be less intuitive, and harder to tune. We therefore will follow the "runt pruning" algorithm of Stuetzle [11].

Tree simplification begins with the introduction of a new parameter m, the minimum cluster size. Any branch of the cluster tree that represents a cluster of less than m points is pruned out of the tree, and we record the ε value of the split, defining it as the ε value when the points of the pruned branch left the parent branch. That is, for each branch C_i of the cluster tree there is an associated set of points $\{X_{i_1}, X_{i_2}, \ldots, X_{i_t}\} \subseteq X$, and for each point X_{i_ℓ} in $\{X_{i_1}, X_{i_2}, \ldots, X_{i_t}\}$ there exists a value ε_ℓ for which the point X_{i_ℓ} is deemed to have left the cluster (including because the cluster C_i split, or because it was removed).

The resulting pruned tree has many fewer branches, and hence fewer leaves. Furthermore, each remaining branch has a record of the points remaining in the branch at each ε value for which the branch exists. The result is a far simpler tree of clusters, amenable to further analysis, but still containing rich information about the actual cluster structures at a point-wise level.

Finally, it is often desirable to extract a flat clustering – selecting a set of non-overlapping clusters from the tree. For hierarchical cluster schemes this often takes the form of choosing a "cut level" (in our case a choice of ε) and using the clustering at that level of the tree. When we wish to consider variable density clusters, the cut level varies through the tree, and thus we must choose a different approach to selecting a flat clustering.

Notionally our goal is to determine the clusters that persist over the largest ranges of distance scales. To do this we require a measure of the persistence of a cluster. To make this concrete we refer again to Hartigan [23], and also to Müller and Sawitzki [24], for the notion of excess of mass. Given a density function f, let C be a subset of the domain of f, and define the *excess of mass* of C at a level λ to be

$$E(C, \lambda) = \int_{C_\lambda} (f(x) - \lambda) dx,$$

where $C_\lambda = \{x \in C \mid f(x) \geq \lambda\}$. Given a cluster tree for f, we can define the excess of mass of a cluster C_i that exists at level λ_{C_i} of the cluster tree as follows: Let $\lambda_{\text{min}}(C_i)$ be the minimal λ value for the branch associated to C_i in the cluster tree. Then define the excess of mass of C_i to be

$$E(C_i) = \int_{C_i} (f(x) - \lambda_{\text{min}}(C_i)) dx.$$

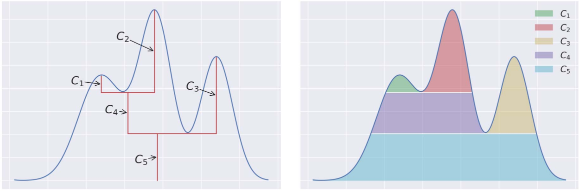
Next we follow [7] in defining the *relative excess of mass* for a cluster C_i. First we define $\lambda_{\text{max}}(C_i)$ to be the maximal lambda value for which C_i exists as a distinct cluster (i.e. before it splits into sub-clusters in the cluster tree). Then the relative excess of mass is

$$E_R(C_i) = \int_{C_i} (\min(f(x), \lambda_{\text{max}}(C_i)) - \lambda_{\text{min}}(C_i) dx.$$

Alternatively, if $C_{i_1}, C_{i_2}, \ldots, C_{i_k}$ are the children of C_i in the cluster tree then

$$E_R(C_i) = E(C_i) - \sum_{j=1}^{k} E(C_{i_j}).$$

That is, the relative excess of mass of a cluster is the total mass of the cluster *not including* the mass of any descendant clusters in the cluster tree. We see this demonstrated in figure 3 with the shaded areas indicating the excess of mass of the each for clusters from the cluster tree.



(a)

(b)

Fig. 3: Relative excess of mass for the cluster tree from figure 2. In (a) we label the clusters, and (b) depicts the relative excess of mass as shaded areas, using different colours for each clusters relative excess of mass.

We can translate these notions to the empirical pruned tree described above. The pruned tree can be used to construct a discrete density function ranging over data points. In order to do this we require two things. Firstly, a density associated to each data point. This is simply the inverse of the ε value at which the point left the tree (this was the data we recorded in addition to pruning branches of the tree). Secondly, we need an ordering on the data points such that the cluster tree of the density function is isomorphic to the pruned tree. This is simply a matter of sorting the points via a depth first search of the pruned tree (making use of the per point ε values to order data points within a branch of the cluster tree). Explicitly we have an empirical density

$$\hat{f}(X_j) = \frac{1}{\varepsilon_{X_j}}$$

where ε_{X_j} is the ε value at which we recorded the point X_j leaving the tree. Further we have a cluster tree associated with $\hat{f}$ isomorphic to the pruned tree and we can define $\lambda_{\text{min}}(C_i)$ for any cluster C_i in the pruned tree accordingly.

This allows us to compute excess of mass for $\hat{f}$ and any cluster C_i from the pruned tree as

$$E(C_i) = \sum_{X_j \in C_i} \left(\hat{f}(X_j) - \lambda_{\text{min}}(C_i)\right)$$

and consequently we have a persistence score provided by the relative excess of mass. That is, given cluster C_i having children $C_{i_1}, C_{i_2}, \ldots, C_{i_k}$, we define the persistence score

$$\sigma(C_i) = E(C_i) - \sum_{j=1}^{k} E(C_{i_j}).$$

The optimal flat clustering can then be described as the solution to a constrained optimization problem. If the set of clusters is $\{C_1, C_2, \ldots, C_n\}$ then we wish to select $I \subseteq \{1, 2, \ldots, n\}$ to maximize

$$\sum_{i \in I} \sigma(C_i)$$

subject to the constraint that, for all $i, j \in I$ with $i \neq j$, we have

$$C_i \cap C_j = \emptyset.$$

That is, we wish to maximize the total persistence score over chosen clusters, subject to the constraint that clusters must not overlap. This constrained optimization problem is can be solved in a straightforward manner [8].

B. Computationally Motivated HDBSCAN*

HDBSCAN* can be thought of as a natural extension of the popular DBSCAN algorithm. We begin, following [7] and [8], by describing a modified version of DBSCAN, denoted DBSCAN*, that will make the relationship clearer. This algorithm is an adaptation of standard DBSCAN which removes the notion of border points (see [7] for more details). Removing border points provides clarity and improves consistency with the statistical interpretation of clustering in section II-A. DBSCAN* takes two parameters, ε and k, where ε is a distance scale, and k is a density threshold expressed in terms of a minimum number of points. Extending to HDBSCAN* can be conceptually considered as searching over all ε values for DBSCAN* to find the clusters that persist for many values of ε. This selection of clusters, which persist over many distance scales, provides the benefits of not only eliminating the need to select the ε parameter but also of dealing with the problem of variable density clustering, something which classical DBSCAN struggles with.

We will describe DBSCAN* in the same terms used by Campello, et al. [7]. Again we will be working with a set of $X = \{X_1, X_2, \ldots, X_N\}$ of data points in a metric space $(\mathcal{X}, d)$.

A point X_i is called a *core point* with respect to ε and k if its ε-neighbourhood contains at least k many points, i.e. if $|B(X_i, \varepsilon) \cap X| \geq k$. That is, the open ball of radius ε contains at at least k many points from X.

Two core points X_i and X_j are *ε-reachable* with respect to ε and k if $X_i \in B(X_j, \varepsilon)$ and $X_j \in B(X_i, \varepsilon)$. That is, they are both core points with respect to k, and are both contained within each others ε-neighbourhood. Two core points X_i and X_j are *density-connected* with respect to ε and k if they are directly or transitively ε-reachable.

A *cluster* C, with respect to ε and k, is a non-empty maximal subset of X such that every pair of points in C is density-connected. This definition of cluster results in the DBSCAN* algorithm.

To extend the algorithm to get HDBSCAN* we need to build a hierarchy of DBSCAN* clusterings for varying ε values. The key to doing this is to redefine how we measure distance between points in X. For a given fixed value k, we define a new distance metric derived from the metric d, called the *mutual reachability distance*, as follows. For any point X_i we define the *core-distance* of X_i, denoted $\kappa(X_i)$ to be the distance to the k^{th} nearest neighbor of X_i; then, given points X_i and X_j we define

$$d_{\text{mreach}}(X_i, X_j) = \begin{cases} \max\{\kappa(X_i), \kappa(X_j), d(X_j, X_j)\} & X_i \neq X_j \\ 0 & X_i = X_j \end{cases}.$$

It is straightforward to show that this is indeed a metric on X. We can then apply standard Single Linkage Clustering [25] to the discrete metric space (X, d_{mreach}) to obtain a hierarchical clustering of X. The clusters at level ε of this hierarchical clustering are precisely the clusters obtained by DBSCAN* for the parameter choices k and ε; in this sense we have derived a hierarchical DBSCAN* clustering.

The goal of a density based algorithm, such as DBSCAN, is to find areas of the greatest density. To do this we need to shift from the notion of distance to a notion of density. An efficient estimate of the local density at a point can be provided by the reciprocal of the distance to its k^{th} nearest neighbor. This is simply the inverse of the core-distance of that point. With this in mind we will work in terms of varying density instead of varying distance by constructing our cluster tree with respect to $\lambda = \frac{1}{\varepsilon}$ and consider the λ value at which cluster splits occur.

The next step in the algorithm is to produce a condensed tree that simplifies the hierarchy. For this we introduce a new parameter m which will denote the minimum cluster size that will be accepted. We process the tree from the root downward. At each cluster split we consider the child clusters. Any child cluster containing fewer than m points is considered a spurious split, and we denote those points as "falling out of the parent cluster" at the given λ value. If only one child cluster contains more than m points we consider it the continuation of the parent, persisting the parent cluster's label/identity to it. If more than a single child cluster contains more than m points then we consider the split to be a "true" split. In this fashion we arrive at a tree with a much smaller number of clusters which "shrink" in size as they persist over increasing λ values of the tree. One can consider this a form of smoothing of the tree.

We can now define the stability of a cluster to be the sum of the range of λ values for points in a cluster. Explicitly, we define $\lambda_{\max,C_i}(X_j)$ to be the λ value at which the point X_j falls out of the cluster C_i (either as an individual point, or as a cluster split in the condensed tree). Similarly we define $\lambda_{\min,C_i}(X_j)$ as the minimum lambda value for which X_j is present in C_i. Then the stability of the cluster C_i is defined as

$$\sigma(C_i) = \sum_{X_j \in C_i} \left(\lambda_{\max,C_i}(X_j) - \lambda_{\min,C_i}(X_j)\right).$$

The optimal flat clustering can then be described as the solution to a constrained optimization problem. If the set of clusters is $\{C_1, C_2, \ldots, C_n\}$ then we wish to select $I \subseteq \{1, 2, \ldots, n\}$ to maximize

$$\sum_{i \in I} \sigma(C_i)$$

subject to the constraint that, for all $i, j \in I$ with $i \neq j$, we have

$$C_i \cap C_j = \emptyset$$

That is, we wish to maximize the total persistence score over chosen clusters, subject to the constraint that clusters must not overlap.

We should note that while this explanation is the more compact of the two, it is also the least formal, and most heuristically motivated.

III. Accelerating HDBSCAN*

As described in [7] and [8] the HDBSCAN* algorithm on N data points has $O(N^2)$ run-time. To be competitive with other high performance clustering algorithms a sub-quadratic run-time is required, with an $O(N \log N)$ run-time strongly preferred. The run-time analysis of HDBSCAN* in [8] identified three steps having $O(N^2)$ time complexity: the computation of core-distances (and mutual reachability distances); the computation of a minimum spanning tree (MST) used for single linkage computation; and the tree condensing. We propose to improve each of these steps, and in so doing, approach an average case complexity that grows approximately proportionally to $N \log N$.

One of the most common techniques for asymptotic performance improvement in the face of pairwise statistical problems (in our case pairwise distance computations) are space tree algorithms [26]. Indeed, these techniques are the basis for the impressive asymptotic performance of the DBSCAN and Mean Shift clustering algorithms, and are even used to accelerate some versions of K-Means. These techniques can also be applied to HDBSCAN* whenever the input data is provided as points in some metric space.

The computation of core-distances is a query for the k^{th} nearest neighbor of each point in the input data set. The use of space tree algorithms for efficient nearest neighbor computations is well established. In particular kd-trees [27] in euclidean space, and ball-trees [28] or cover trees [29] for generic metric spaces, provide fast asymptotic performance for nearest neighbor computation. Strict asymptotic run-time bounds for such algorithms are often complicated by properties of the data set. For example, cover tree nearest neighbor computation is dependent upon the expansion constant of the data, and the performance of kd-trees and ball-trees are similarly dependent upon the data distribution. However, an all points nearest neighbor query algorithm for cover trees with "linear" run-time complexity $O(c^{16}N)$, where c is the expansion constant for the cover tree, is presented by Ram et al. [26]. Claims of empirical run-time complexity of approximately $O(N \log N)$ for kd-trees and ball-trees are also common. While explicitly stating a run-time complexity for the core-distance computation is difficult, we feel confident in stating that, except for carefully constructed pathological examples, we can achieve sub-quadratic complexity.

With core-distance computation improved, the next challenge is the efficient computation of single linkage clustering using mutual reachability distance. In [8], Campello et al. use Prim's algorithm [30] to compute a minimum spanning tree of the complete graph with edges weighted by the mutual reachability distance. Campello et al. then sort the edges, and use that data to construct the single linkage tree. Such an approach is similar to the SLINK algorithm [31], [25] which essentially uses a modified version of Prim's algorithm (that does not explicitly compute an MST). For the purposes of computing a MST, Prim's is among the fastest available algorithms, however it is targeted toward graphs where the number of edges is some small multiple of the number of vertices, rather than complete graphs with $O(|V|^2)$ edges. In particular, if we have extra information about the vertices of the graph, other algorithms such as Borůvka's algorithm [32] become more appealing. This is because if vertices are points in some metric space and edge weights are distances, Borůvka's algorithm resembles a series of repeated all points nearest neighbor queries.

In [13] March et al. make use of this observation and describe the Dual-Tree Borůvka algorithm for computing minimum spanning trees of points in a metric space. Given points X in $(\mathcal{X}, d)$, they provide an algorithm to compute a minimum spanning tree of the weighted complete graph with vertices X and edges (X_i, X_j) with weight $d(X_i, X_j)$, where $X_i, X_j \in X$. The algorithm makes explicit use of space trees to provide impressive asymptotic performance. In particular, if cover trees are used, March et al. prove a run-time complexity of $O(\max\{c^6, c_p^2, c_l^2\}c^{10} N \log N \alpha(N))$, where c, c_p, and c_l are data dependent constants and α is the inverse Ackermann function [33]. Here we provide a (minor) adaptation of the algorithm to compute a MST of the mutual reachability distances, resulting in a computation with sub-quadratic complexity.

In describing the algorithm we follow the approach of Curtin et al. in [15] where they provide a version of March's algorithm adapted to a generic space partitioning tree framework. We begin with the introduction of notation to allow for easier statements of required algorithms.

For our purposes, a *space tree* on a data set $X \subset (\mathcal{X}, d)$ is a rooted tree with the following properties:

- Each node holds a number of points (possibly zero), has a single parent and has some number of children (possibly zero);
- each $X_i \in X$ is contained in at least one node of the tree;
- each node of the tree has an associated convex subset of $(\mathcal{X}, d)$ that contains all the points in the node, and the convex subsets associated with all of its children.

Notationally we will use a number of short form conventions to make discussions of points, children, descendants, and distances between nodes more convenient. Again, following Curtin et al. we will use the following notation:

- The set of child nodes of a node $\mathscr{N}_i$ will be denoted $\mathscr{C}(\mathscr{N}_i)$ or simply $\mathscr{C}_i$ if the context allows.
- The parent node of a node $\mathscr{N}_i$ will be denoted $\mathscr{U}(\mathscr{N}_i)$.
- The set of points held in a node $\mathscr{N}_i$ will be denoted $\mathscr{P}(\mathscr{N}_i)$ or simply $\mathscr{P}_i$ if the context allows.
- The convex subset of $(\mathcal{X}, d)$ associated to a node $\mathscr{N}_i$ will be denoted $\mathscr{S}(\mathscr{N})$ or simply $\mathscr{S}_i$ if the context allows.
- The set of *descendant nodes* of a node $\mathscr{N}_i$, denoted by $\mathscr{D}^n(\mathscr{N}_i)$ or $\mathscr{D}^n_i$, is the set of nodes $\mathscr{C}(\mathscr{N}_i) \cup \mathscr{C}(\mathscr{C}(\mathscr{N}_i)) \cup \dots$.
- The set of *descendant points* of a node $\mathscr{N}_i$, denoted $\mathscr{D}^p(\mathscr{N}_i)$ or $\mathscr{D}^p_i$, is the set of points $\{p \mid p \in \mathscr{D}^n(\mathscr{N}_i) \cup \mathscr{P}(\mathscr{N}_i)\}$.
- The *minimum distance* between two nodes $\mathscr{N}_i$ and $\mathscr{N}_j$, denoted $d_{\min}(\mathscr{N}_i, \mathscr{N}_j)$ is defined as $\min\{d(p_i, p_j) \mid p_i \in \mathscr{D}^p_i, p_j \in \mathscr{D}^p_j\}$.
- The *maximum child distance* of a node $\mathscr{N}_i$, denoted $\rho(\mathscr{N}_i)$ is maximum distance from the centroid of $\mathscr{S}(\mathscr{N}_i)$ to any point in $\mathscr{N}_i$.
- The *maximum descendant distance* of a node $\mathscr{N}_i$, denoted $\lambda(\mathscr{N}_i)$ is the maximum distance from the centroid of $\mathscr{S}(\mathscr{N}_i)$ to any *descendant point* of $\mathscr{N}_i$.

In general, the minimum distance between nodes can be bounded below statically without having to compute all the point to point distances. For example, in kd-trees we have $d_{\min}(\mathscr{N}_i, \mathscr{N}_j)$ bounded below by the minimum distance between $\mathscr{S}_i$ and $\mathscr{S}_j$ which can be computed at the time of tree construction without computing any point to point distances. Other types of space trees offer similar methods to bound node distances.

In [15] Curtin et al. provide a generic algorithm from which specific dual tree algorithms can be constructed. This provides a simple breakdown of a dual tree algorithm into core constituent parts, which the authors of this paper found particularly helpful in understanding March's algorithm. We therefore work within the same general framework here.

Dual tree algorithms make use of two different space trees, a *query tree* $\mathscr{T}_q$ and a *reference tree* $\mathscr{T}_r$. Curtin et al. breaks dual tree algorithms into three components. The first component is a *pruning dual tree traversal*. This is a method of traversing a query and reference tree pair, pruning branches along the way. At each stage of such a pruning traversal we apply two procedures: the first, called Score, determines whether a branch is to be pruned (and potentially prioritises child branches); the second, called BaseCase, performs some algorithm specific operation on the pair of nodes at that stage of the traversal.

A simple approach to a dual tree traversal is a depth first traversal with no prioritisation of child nodes to explore. Algorithm 1 describes such an approach. In practice, one may want a more finely tailored traversal algorithm, with concomitant complexity of description, but for our explanatory purposes, this simple traversal is sufficient.

Algorithm 1 Depth First Dual Tree Traversal

```
procedure DepthFirstTraversal($\mathscr{N}_q, \mathscr{N}_r$)
    if Score($\mathscr{N}_q, \mathscr{N}_r$) = $\infty$ then
        return
    for all $p_q \in \mathscr{P}_q, p_r \in \mathscr{P}_r$ do
        BaseCase($p_q, p_r$)
    for all $\mathscr{N}_{qc} \in \mathscr{C}_q, \mathscr{N}_{qr} \in \mathscr{C}_r$ do
        DepthFirstTraversal($\mathscr{N}_{qc}, \mathscr{N}_{rc}$)
```

Given a traversal algorithm, the specifics of March's Dual Tree Borůvka algorithm now falls to the BaseCase and Score procedures. To explicate these we begin by describing Borůvka's original algorithm, and then explain how we reconstruct it within a dual tree framework.

The general idea for Borůvka's algorithm (Algorithm 2) is to build a forest, adding minimum weight edges to connect trees in iterative rounds. Borůvka's algorithm starts with a weighted graph G, and initializes a forest T to have the vertices of G, and no edges. Each pass of Borůvka's algorithm finds minimum weight edges that span distinct connected components of T, and then adds those edges to T. As the algorithm proceeds, T has larger but fewer connected components. The algorithm terminates when the forest T is a single connected component, and thus a tree.

Algorithm 2 Classical Borůvka's algorithm

```
procedure MST($G = (V, E)$)
    $T \leftarrow (V, \emptyset)$   ▷ Initialize a graph $T$ with vertices from $G$ and no edges
    while $T$ has more than one connected component do
        for all components $C$ of $T$ do
            $S \leftarrow \emptyset$
            for all vertices $v$ in $C$ do
                $D \leftarrow \{a \in E \mid a$ meets $v$ and is not wholly contained in $C\}$
                $e \leftarrow$ minimum weight edge in $D$
                $S \leftarrow S \cup \{e\}$
            $e \leftarrow$ minimum weight edge in $S$
            Add $e$ to the graph $T$
```

To convert Borůvka's algorithm to a dual tree algorithm employing the spatial nature of the data, we make use of the space trees to find the nearest neighbors in a different component of the current forest for each point in the dataset. We then compile this information together to update the forest, and then reapply the nearest neighbor search.

Notationally we are building a forest F with connected components F_i. At initialization F has no edges, and there are N connected components. At each pass of the algorithm we will add edges to F and update the list of connected components accordingly.

To keep track of state during processing, a number of associative arrays are required. First we require a mapping from points to the connected component of F in which they currently reside. We denote this $\mathcal{F}$ and define $\mathcal{F}(p)$ to be the component F_i which contains the point p. During the tree traversal we keep track of the nearest candidate point for each component with an associative array $\mathcal{N}$ such that $\mathcal{N}(F_i)$ is the candidate point (not in component F_i) nearest to component F_i found so far. To keep track of which point in the component F_i is closest to the candidate point we use an associative array $\mathcal{P}$ such that $\mathcal{P}(F_i)$ is the point in component F_i nearest to $\mathcal{N}(F_i)$. Finally we keep track of the distance to a nearest neighbor for each component through an associative array $\mathcal{D}$ such that $\mathcal{D}(F_i)$ is the distance between $\mathcal{N}(F_i)$ and $\mathcal{P}(F_i)$.

To perform passes of the algorithm we need to use a modified nearest neighbor approach that looks for the nearest neighbor in a different component. Since we are searching for the "nearest neighbors" of the reference points each time, the query tree and reference tree are the same. After such an all-points nearest neighbor style tree search we can collate the results found for $\mathcal{N}$, $\mathcal{P}$ and $\mathcal{D}$ and use that to update the forest, and the associative array $\mathcal{F}$. This allows us to reset $\mathcal{N}, \mathcal{P}$ and $\mathcal{D}$ and make another pass with the same nearest neighbor style search. Each pass reduces the number of connected components in F until we have a minimal spanning tree.

With this is mind, the BASECASE (algorithm 3) needs to find points in different components that have a shorter distance separating them than the current value stored for the component under consideration. If such a pair is found we update $\mathcal{N}$, $\mathcal{P}$ and $\mathcal{D}$ accordingly.

Algorithm 3 Borůvka's algorithm base case

```
procedure BASECASE($p_q, p_r$)
    if $p_q = p_r$ then
        return
    if $\mathcal{F}(p_q) \neq \mathcal{F}(p_r)$ and $d(p_q, p_r) < \mathcal{D}(\mathcal{F}(p_q))$ then
        $\mathcal{D}(\mathcal{F}(p_q)) \leftarrow d(p_q, p_r)$
        $\mathcal{N}(\mathcal{F}(p_q)) \leftarrow p_r$
        $\mathcal{P}(\mathcal{F}(p_q)) \leftarrow p_q$
```

The benefit of the tree based approach is that we are able to prune branches from our tree search which we know will not yield useful results. Since our queries are closely related to nearest neighbor queries we can make use of similar bounding approaches. The simplest such bound will prune the node pair $(\mathscr{N}_q, \mathscr{N}_r)$ if and only if the minimal distance between the nodes $d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$ is greater than the maximum of the nearest neighbor distances found so far for any point in $\mathscr{D}_q^p$. That is, if the closest any point in the query node (or its descendants) can be to any point in the reference node (or its descendants) is greater than all the current query node nearest neighbors found, clearly we do not need to descend any further.

In practice, with care and use of the triangle inequality, better bounds can be derived. We refer the reader to [15] for the derivation, but note that we can define a bound

$$\begin{aligned} B(\mathscr{N}_q) = \min\Big\{ & \max\{\max_{p \in \mathscr{P}_q} D_p, \max_{\mathscr{N}_c \in \mathscr{C}_q} B(\mathscr{N}_c)\}, \\ & \min_{p \in \mathscr{P}_q} \big(D_p + \rho(\mathscr{N}_q) + \lambda(\mathscr{N}_q)\big), \\ & \min_{\mathscr{N}_c \in \mathscr{C}_q} \Big(B(\mathscr{N}_c) + 2\big(\lambda(\mathscr{N}_q) - \lambda(\mathscr{N}_c)\big)\Big), \\ & B(\mathscr{U}(\mathscr{N}_q))\Big\}, \end{aligned}$$

where D_p is the distance to the nearest neighbor of p found so far. Given this bound, if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) \geq B(\mathscr{N}_q)$ then we can safely prune the pair $(\mathscr{N}_q, \mathscr{N}_r)$. In practice pruning will be done based on the pre-computed lower bound estimate for $d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$. Furthermore, as the bound is expressed recursively, we can cache previous computations and calculate $B(\mathscr{N}_q)$ efficiently.

We can improve our pruning further by using component membership to prune: if all the descendant points in the query and reference nodes are in the same component then we do not need to descend and check any of those points. Again, this is a computation that can be done recursively and cached. With that in mind we can define a SCORE function (Algorithm 4) that prunes away unnecessary branches, resulting in far fewer distance computations being required.

Algorithm 4 Borůvka's algorithm scoring

```
procedure SCORE($\mathscr{N}_q, \mathscr{N}_r$)
    if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) < B(\mathscr{N}_q)$ then
        if $\forall(p_q \in \mathscr{D}_q^p, p_r \in \mathscr{D}_r^p) : \mathcal{F}(p_q) = \mathcal{F}(p_r)$ then
            return $\infty$
        return $d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$
    return $\infty$
```

Combining all these pieces together provides us with an algorithm to compute a minimum spanning tree of the distance weighted complete graph of points in a metric space. In practice, we wish to compute a MST using mutual reachability distance, and want to compute as few distances as possible. We can do this by using precomputed core-distances as a filter on the pairs of points passed to BASECASE, and only compute distances (and mutual reachability distances) for a subset of points. Furthermore, by prioritising the order in which we perform our dual tree traversal we can construct tighter bounds B sooner, and thus perform more tree pruning, resulting in even fewer distances being computed.

We can express this in a more detailed tree traversal algorithm. The traversal algorithm is assumed to have access to the associative arrays $\mathcal{F}$ and $\mathcal{D}$. We also introduce a new associative array $\mathcal{C}$ such that $\mathcal{C}(p)$ is the core-distance (i.e. distance to the k^{th} nearest neighbor) for the point p. We can then expand out the loop over pairs of algorithm 1 into a pair of nested for loops over points in the query node, and points in the reference node. This allows us to check if the core-distance of a point exceeds the current best distance for the component the point lies in. If the core-distance is larger then the mutual reachability distance is necessarily also larger, and hence this point can be eliminated from consideration.

Algorithm 5 Tailored dual tree traversal

```
procedure DUALTREETRAVERSAL($\mathscr{N}_q, \mathscr{N}_r$)
    if SCORE($\mathscr{N}_q, \mathscr{N}_r$) = $\infty$ then
        return
    for all $p_q \in \mathscr{P}_q$ do
        if $\mathcal{C}(p_q) < \mathcal{D}(\mathcal{F}(p_q))$ then
            for all $p_r \in \mathscr{P}_r$ do
                if $\mathcal{C}(p_r) < \mathcal{D}(F(p_q))$ then
                    BASECASE($p_q, p_r$)
    $L \leftarrow$ SORT($[(\mathscr{N}_{qc}, \mathscr{N}_{rc}) \mid \mathscr{N}_{qc} \in \mathscr{C}_q, \mathscr{N}_{qr} \in \mathscr{C}_r],\ d_{\min}(\cdot,\cdot)$)
    for $(\mathscr{N}_{qc}, \mathscr{N}_{qr})$ in $L$ do
        DUALTREETRAVERSAL($\mathscr{N}_{qc}, \mathscr{N}_{rc}$)
```

We can also prioritise the tree descent based on, for example, the distance between nodes, descending to nodes that are closer together first such that bounds get updated earlier. Algorithm 5 gives an example of such a tailored algorithm that takes advantage of core-distances and prioritises descent down the tree. In practice the exact traversal and descent strategy can be more carefully tuned according to the exact space tree used.

It only remains to adapt the BASECASE procedure presented in Algorithm 3 to use core-distances to compute d_{mreach} and use it instead of d (as seen in Algorithm 6) and we have a Dual Tree Borůvka algorithm adapted to perform HDBSCAN*. Such an algorithm allows us to compute a minimum spanning tree of the mutual reachability distance weighted complete graph without having to compute all pairwise distances. This results in an asymptotically sub-quadratic MST computation. While the data dependent nature of complexity analysis for tree based algorithms makes it difficult to place an explicit bound on the run-time complexity, analyses such as March et al. [13] suggest we can certainly approach $O(N \log N)$ asymptotic performance for many data sets.

One notable feature of mutual reachability distance is that it can result in many equal distances. We can exploit this fact within our modified Dual Tree Borůvka algorithm. After a tree traversal the algorithm updates the forest F and then resets the bounds B. Since there are many equal distances we can run the tree search again with the same bounds B and find new potential edges to add to F. Such a run is extremely efficient as it has very tight bounds and thus rapidly prunes branches. We can repeat these runs until no new edges are found, and only then reset the values for B, forcing the algorithm to make fast progress in the face of ties, and near ties. Unfortunately this breaks the guarantee that the algorithm will also find a minimal

Algorithm 6 HDBSCAN* tailored Borůvka's algorithm base case

```
procedure BaseCase($p_q, p_r$)
    if $p_q = p_r$ then
        return
    if $F(p_q) \neq F(p_r)$ then
        dist $= \max\{d(p_q, p_r), \mathcal{C}(p_q), \mathcal{C}(p_r)\}$
        if dist $< \mathcal{D}(\mathcal{F}(p_q))$ then
            $\mathcal{D}(\mathcal{F}(p_q)) \leftarrow$ dist
            $\mathcal{N}(\mathcal{F}(p_q)) \leftarrow p_r$
            $\mathcal{P}(\mathcal{F}(p_q)) \leftarrow p_q$
```

spanning tree. However, in practice the result is a close approximation of a minimal spanning tree. We can trade off a small loss in accuracy for a significantly faster algorithm. This heuristic does not alter the asymptotics of the algorithm, but improve the constants. Furthermore, the minor differences in MST get smoothed out in tree condensing and flat cluster extraction process, resulting in very small deviations in final cluster results. On real world data sets (see [8]) this results in essentially no difference in cluster quality scores. This optional trade-off of performance for accuracy is only relevant for higher dimensional data sets when using kd-trees or ball-trees.

Given a minimal spanning tree it is possible to generate a single linkage cluster tree. This proceeds in two stages. The first stage is to sort the edges of the MST by weight (in this case, the mutual reachability distance between the pair of points the edge spans). Such an operation can be performed in $O(N \log N)$ run-time. In the second stage we process the edges in order using a union-find data structure [34]. This allows us to build the single linkage tree, providing the cluster merges and weights at which they occur, by progressively merging points and clusters by increasing weight. Since an MST has $O(N)$ edges we can complete this in $O(N\alpha(N))$ (using union-rank and path compression in our union-find algorithm).

The next step is to process the single linkage tree into a condensed tree. We can do this in a single pass working from the root in a breadth first traversal, building an associative array mapping single linkage cluster identifiers to new condensed tree cluster identifiers. At each node we need only check on the sizes of the child nodes, update the associative array accordingly, and record any data points falling out of the cluster. Since the single linkage tree has $N \log N$ nodes, the condensed tree processing can be completed in $O(N \log N)$ run-time.

In summary, the overall asymptotic run-time performance of the algorithm is bounded by the core-distance and minimum spanning tree computation stages, both of which now have sub-quadratic performance, and can be expected to approach $O(N \log N)$ performance for many data sets. This represents a significant improvement in potential scaling performance for HDBSCAN* clustering.

To test these algorthmic improvements we have implemented our accelerated HDBSCAN* algorithm in Python [35].Our Python implementation builds from, and conforms to, the scikit-learn [36] software, making use of the kd-tree and ball tree data structures provided. Making use of scikit-learn has enabled our implementation to support a wide variety of distance metrics, as well as the ability to fall back to fast $O(N^2)$ algorithms when provided with a (sparse) distance matrix rather than vector space data. In the following section we will make use of our accelerated HDBSCAN* implementation to compare scaling of run-time performance with data set size with classic HDBSCAN*, and with other popular clustering algorithms.

IV. Performance Comparisons

In this section we will analyse the performance of our accelerated HDBSCAN*. For the purpose of this paper we will not be considering the quality of clustering results as that has been adequately covered in [7] and [8]. Instead we will demonstrate the computational competitiveness of our accelerated HDBSCAN* against other existing high performance clustering algorithms. We are mindful of the difficulties of run-time analyses [37]. We therefore focus on scaling trends with data set size (and dimension), and speak to the comparability of algorithms rather than making claims of strict superiority.

All our run-time benchmarking was performed on a Macbook Pro with a 3.1 GHz Intel Core i7 processor and 8GB of RAM. Furthermore the benchmarking was performed in Jupyter notebooks which we have made available at https://github.com/lmcinnes/hdbscan_paper.We encourage others to verify and extend these benchmarks.

A. Comparisons with HDBSCAN* reference implementation

As a baseline we compare the performance of our Python HDBSCAN* implementation against the reference implementation in Java from the original authors. Given two very different implementations in different languages our focus is on demonstrating that overall scalability and asymptotic performance can be improved through the space tree based acceleration techniques described.

We compare the performance on data sets of varying size for both 2-dimensional and 50-dimensional data. The results can be seen in Figure 4. The left hand column demonstrates raw performance times for both 2-dimensional and 50-dimensional data, while the right hand column provides a log-log plot that makes clear the different asymptotic performance of the algorithms. The accelerated Python version shows significantly improved performance, both in absolute terms, and asymptotically (having significantly lower linear slope in the log-log plot), clearly demonstrating sub $O(N^2)$ performance. Furthermore, in both the 2-dimensional and 50-dimensional cases, the accelerated Python version demonstrates roughly two orders of magnitude better absolute run-time performance on data set sizes of 200,000 points.

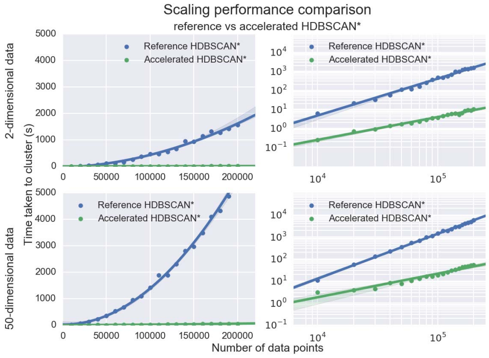


Fig. 4: We compare the reference implementation in Java with the accelerated version implemented in Python. [2]

B. Comparisons among clustering algorithms

In order to gain an overview of the performance landscape of clustering algorithms in general, we compare a number of the more popular clustering algorithms found in scikit-learn[3] [36] [38]. Since we recognise that implementation can have a significant effect

[2] See https://github.com/lmcinnes/hdbscan_paper/blob/master/Performance%20data%20generation.ipynb for the code used to generate this plot

[3] Benchmarking was performed using scikit-learn v0.18.1.

on run-time performance, our goal here is merely to provide a sample of the performance space rather than direct comparisons to specific algorithms. We chose scikit-learn as it provides a number of techniques that all rest on a common implementation foundation (including our scikit-learn compatible HDBSCAN* implementation).

For the initial comparison we consider the following algorithms as implemented in scikit-learn: Affinity Propagation [39], Birch [40], Complete Linkage [41], DBSCAN [6], KMeans [42], Mean Shift[43], Spectral Clustering [44], and Ward Clustering [45]. We compare these with our HDBSCAN* implementation. At this time scikit-learn has no implementation of OPTICS [46]. In order to maintain a consistent implementation base for runtime comparison we do not compare with OPTICS[4].

Since this is a broad comparison of overall performance characteristics, each algorithm will be initialized with default scikit-learn parameters. In the next section, we do a more detailed comparison; carefully considering the impact of clustering algorithm parameters on performance.

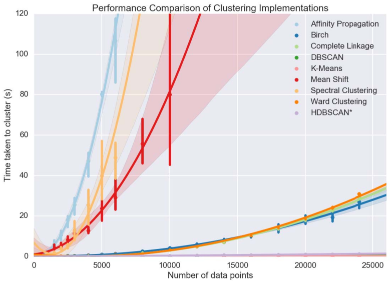


Fig. 5: Comparison of scaling performance for scikit-learn implementations of a number of different clustering algorithms. Vertical bars present the range of run-times obtained over several runs at a given data set size.[5]

As demonstrated in Figure 5, there are three classes of implementation. The first is Affinity Propagation, Spectral Clustering, and Mean Shift, which all had poor performance beyond a few thousand data points. Some of this is undoubtedly implementation specific (particularly in the case of Spectral Clustering and Mean Shift). The next class of implementations are Ward, Complete Linkage and Birch, which performed better, but still scaled poorly for larger data set sizes. Finally, there was the group of DBSCAN, K-Means and HDBSCAN*, which are difficult to tell apart from one another in Figure 5.

If we consider a log-log plot of the same data (Figure 6) in order to better see and understand the asymptotic scaling, we see algorithms ranging from K-Means impressive approximately $O(N)$ performance, through to the traditional $O(N^2)$ algorithms. DBSCAN and HDBSCAN* demonstrate similar asymptotics to each other, and are the closest in performance to K-Means. Also worth noting is that Mean Shift, while having poorer performance in general, has similar asymptotic performance to DBSCAN and HDBSCAN*.

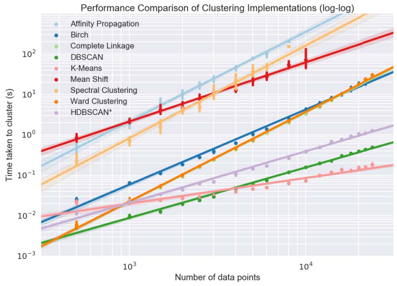


Fig. 6: Comparison of scaling performance for scikit-learn implementations of a number of different clustering algorithms plotted on a log-log scale to demonstrate asymptotic performance more clearly.[6]

K-Means, while being the fastest and most scalable algorithm (Figures 5 and 6) explicitly fails to meet our desiderata. Although K-Means has only a single parameter, the selection of that parameter is difficult. K-Means also has implicit apriori assumptions about the data distribution – specifically that clusters are Gaussian. Finally, K-Means is explicitly a partitioning algorithm and does not cope well with noise or outliers.

This leaves DBSCAN as the main competitor to our accelerated HDBSCAN*. We therefore seek a more detailed comparison of performance between DBSCAN and HDBSCAN*, specifically considering how parameter selection can affect performance.

C. Comparisons with DBSCAN

A detailed comparison of the quality of clusterings generated by DBSCAN and HDBSCAN* is beyond the scope of this paper, for this we refer readers to [8]. Thus, we limit our experiments in this section to a run time comparison of these two algorithms.

The difficulty with DBSCAN run-time comparisons using default parameters is that very small values of ε will return few or no core points. This results in a very fast run with virtually all the data being relegated to background noise. Conversely, for large values of ε DBSCAN will have very poor performance. Our desire is to not misrepresent DBSCAN's run-times for real world use cases. To circumvent this problem we will perform a search over the parameter space of DBSCAN in order to find the parameters which best match our HDBSCAN* results on a particular data set. This is reasonable because, as described in section II-B, HDBSCAN* can be viewed as a natural extension to DBSCAN. Once suitable parameters have been discovered we will benchmark the run-time of DBSCAN using those specific parameters against our HDBSCAN* run-time. Of course, in practice a user may not, apriori, know the optimal parameter values for DBSCAN; in these experiments we do not penalize DBSCAN for this weakness.

As is the case for all tree based algorithms, run-time and run-time complexity are data dependent. As indicated in [37] this raises significant difficulties when benchmarking algorithms or implementations. Our interest is in demonstrating the comparability of the scaling performance of these algorithms. Under the assumption that both algorithms are tree based, they should have similar performance changes under different data distributions. As such, we will examine the run-time behaviour of both algorithms with respect to a fairly simple data set.

For this simple scaling experiment our data will consist of mixtures of Gaussian distributions laid down within a fixed diameter hypercube. We use variable numbers of constant variance Gaussian

[4] comparison with a not yet merged scikit-learn OPTICS implementation shows equivalent performance difference to that demonstrated in the prior section comparing with the reference HDBSCAN* implementation

[5] See https://github.com/lmcinnes/hdbscan_paper/blob/master/Perfomance%20comparisons%20among%20clustering%20algorithms.ipynb for the code used to generate this plot

[6] See https://github.com/lmcinnes/hdbscan_paper/blob/master/Perfomance%20comparisons%20among%20clustering%20algorithms.ipynb for the code used to generate this plot

balls for simplicity and to not unfairly penalize DBSCAN. DBSCAN, as has been previously mentioned, does not support variable density clusters and thus could not match the output of HDBSCAN* in such cases. We vary dimension, number of clusters and number of data points to determine their effect on run-time.

Although we are building a generative model on which to compare the performance of DBSCAN and HDBSCAN*, it should be noted that we are comparing the clustering results of the algorithms directly against each other and not against the underlying generative model. This is intentional, since for any given instantiation of our generative model the generative model is not necessarily the most likely model (see [1]). We avoid this issue entirely by ignoring the generative model used to create the data for these experiments.

For the purposes of this experiment we chose to use scikit-learn's implementation of DBSCAN[7]. We did this for two reasons. First, scikit-learn's DBSCAN implementation is among the fastest of available DBSCAN implementations (see [37] for DBSCAN implementation comparisons). Second, our Python HDBSCAN* implementation was built using scikit-learn[8]. This means that both the DBSCAN and HDBSCAN* implementations will be using the same underlying library implementations and, in particular, the same implementation of kd-trees which account for a significant part of any performance gains. A common implementation base aids in extension of results from implementations to algorithms.

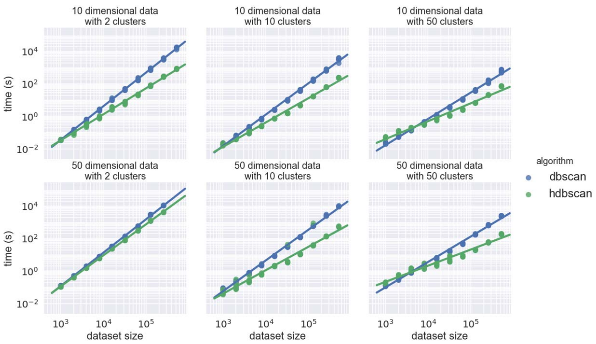


Fig. 7: Comparison of scaling performance for scikit-learn's implementation of DBSCAN and our accelerated HDBSCAN*. Axes are on a $\log_{10}$ scale. Each individual plot provides a log-log plot of runtime against data set size, with individual plots for each combination of data set dimension and number of clusters. For each parameter combination ten random data sets where generated in order to assess the variation from data distribution. The plot shows that accelerated HDBSCAN* and DBSCAN exhibit comparable performance. [9]

In order to find the DBSCAN parameters of best fit to our HDBSCAN* clustering, we make use of the Gaussian process optimization framework within scikit-optimize [47]. We treat the background noise identified by both DBSCAN and HDBSCAN* as a single extra "cluster" which yields a partition, allowing us to use the adjusted Rand-index, as proposed by [48], to compute a similarity between the partitionings generated by each algorithm. We then perform Gaussian process optimization to find the ε and k for DBSCAN which optimize this partition similarity score. Due to the expense of this parameter search this optimization was distributed across multiple nodes of a large memory cluster[10].

The run-time comparison can be found in Figure 7. Each individual figure provides a log-log plot of run-time against data set size, with individual figures for each combination of data set dimension, and number of clusters.

Figure 7 clearly indicates that our accelerated algorithm has comparable asymptotic performance to DBSCAN. Furthermore, our implementation has comparable absolute performance. This is a significant achievement considering that HDBSCAN* can be thought of as computing DBSCAN for all values of ε. In fact, a single HDBSCAN* run allows a user to easily extract the DBSCAN clustering for any given ε. More importantly, parameter selection and variable density clusters are DBSCANs challenges; our accelerated HDBSCAN* algorithm has overcome both of these challenges without sacrificing performance.

V. Future work

A number of avenues for significant future work exist. First there are several ways that our current Python implementation could be improved. The effects of approximate nearest neighbor search via spill trees [49], bounding adjustments [14], or RP-trees with local neighborhood exploration [50], both on core-distance computation, and within March's algorithm remains unexplored. Approximate nearest neighbor computations may offer significant performance improvements for a small trade-off in the accuracy of results.

A significant weakness of our accelerated HDBSCAN* algorithm as described is that it is inherently serial. The inability to parallelise the algorithm is an obstacle for its use on large distributed data sets. We believe that partitioning the space via spill trees [49], and building MSTs on the partitioned data in parallel, then using the techniques of Karger, Klein and Tarjan [51] to reconcile the overlapping trees may result in such a parallel algorithm. This is a topic of continued research.

VI. Conclusions

The HDBSCAN* clustering algorithm lies at the confluence of several threads of research from diverse fields. As a density based algorithm with a small number of intuitive parameters and few assumptions about data distribution, it is ideally suited to exploratory data analysis. ChauduriIn this paper we have presented a new statistically motivated presentation of HDBSCAN*, demonstrating its relationship to Robust Single Linkage clustering, and described an accelerated HDBSCAN* algorithm that can provide comparable performance to the popular DBSCAN clustering algorithm. Since it has more intuitive parameters and can find variable density clusters, HDBSCAN* is clearly superior to DBSCAN from a qualitative clustering perspective [8]. As the improvements of our accelerated HDBSCAN* make its computational scalability comparable in performance to DBSCAN, HDBSCAN* should be the default choice for clustering.

References

[1] C. Hennig, "What are the true clusters?" *Pattern Recognition Letters*, vol. 64, pp. 53 – 62, 2015, philosophical Aspects of Pattern Recognition. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167865515001269

[2] R. L. Melvin, R. C. Godwin, J. Xiao, W. G. Thompson, K. S. Berenhaut, and F. R. Salsbury Jr, "Uncovering large-scale conformational change in molecular dynamics without prior knowledge," *Journal of Chemical Theory and Computation*, vol. 12, no. 12, pp. 6130–6146, 2016.

[7] Benchmarks were run using scikit-learn v0.18.1

[8] Our HDBSCAN* implementation has a similar level of genericity to DBSCAN, supporting the same distance metrics etc.

[9] A more detailed supplemental notebook can be found at https://github.com/lmcinnes/hdbscan_paper/blob/master/Benchmark_vs_DBSCAN.ipynb

[10] The results of this optimization can be found at https://github.com/lmcinnes/hdbscan_paper/blob/master/optimizationResults.csv. Due to the fact that we are only concerned with relative timings between dbscan and hdbscan* we omit the exact specifications of this cluster

[3] A. T. Wilson, M. D. Rintoul, and C. G. Valicka, “Exploratory trajectory clustering with distance geometry,” in *International Conference on Augmented Cognition*. Springer, 2016, pp. 263–274.

[4] P. R. Spackman, S. P. Thomas, and D. Jayatilaka, “High throughput profiling of molecular shapes in crystals,” *Scientific reports*, vol. 6, 2016.

[5] M. Korakakis, P. Mylonas, and E. Spyrou, “Xenia: A context aware tour recommendation system based on social network metadata information,” in *Semantic and Social Media Adaptation and Personalization (SMAP), 2016 11th International Workshop on*. IEEE, 2016, pp. 59–64.

[6] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, “A density-based algorithm for discovering clusters in large spatial databases with noise.” in *Kdd*, vol. 96, no. 34, 1996, pp. 226–231.

[7] R. J. Campello, D. Moulavi, and J. Sander, “Density-based clustering based on hierarchical density estimates,” in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2013, pp. 160–172.

[8] R. J. Campello, D. Moulavi, A. Zimek, and J. Sander, “Hierarchical density estimates for data clustering, visualization, and outlier detection,” *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 10, no. 1, p. 5, 2015.

[9] K. Chaudhuri and S. Dasgupta, “Rates of convergence for the cluster tree,” in *Proceedings of the 23rd International Conference on Neural Information Processing Systems*, ser. NIPS’10. USA: Curran Associates Inc., 2010, pp. 343–351. [Online]. Available: http://dl.acm.org/citation.cfm?id=2997189.2997228

[10] K. Chaudhuri, S. Dasgupta, S. Kpotufe, and U. von Luxburg, “Consistent procedures for cluster tree estimation and pruning,” *IEEE Transactions on Information Theory*, vol. 60, no. 12, pp. 7900–7912, 2014.

[11] W. Stuetzle, “Estimating the cluster tree of a density by analyzing the minimal spanning tree of a sample,” *Journal of classification*, vol. 20, no. 1, pp. 025–047, 2003.

[12] W. Stuetzle and R. Nugent, “A generalized single linkage method for estimating the cluster tree of a density,” *Journal of Computational and Graphical Statistics*, vol. 19, no. 2, pp. 397–418, 2010.

[13] W. B. March, P. Ram, and A. G. Gray, “Fast euclidean minimum spanning tree: algorithm, analysis, and applications,” in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2010, pp. 603–612.

[14] R. R. Curtin, “Faster dual-tree traversal for nearest neighbor search,” in *International Conference on Similarity Search and Applications*. Springer, 2015, pp. 77–89.

[15] R. R. Curtin, W. B. March, P. Ram, D. V. Anderson, A. G. Gray, and C. L. Isbell Jr, “Tree-independent dual-tree algorithms,” *arXiv preprint arXiv:1304.4327*, 2013.

[16] J. A. Hartigan, “Consistency of single linkage for high-density clusters,” *Journal of the American Statistical Association*, vol. 76, no. 374, pp. 388–394, 1981.

[17] A. Rinaldo, A. Singh, R. Nugent, and L. Wasserman, “Stability of density-based clustering,” *Journal of Machine Learning Research*, vol. 13, no. Apr, pp. 905–948, 2012.

[18] A. Rinaldo and L. Wasserman, “Generalized density clustering,” *The Annals of Statistics*, pp. 2678–2722, 2010.

[19] U. Von Luxburg and S. Ben-David, “Towards a statistical theory of clustering,” in *Pascal workshop on statistics and optimization of clustering*. Citeseer, 2005, pp. 20–26.

[20] A. Martínez-Pérez, “On the properties of α-unchaining single linkage hierarchical clustering,” *Journal of Classification*, vol. 33, no. 1, pp. 118–140, 2016.

[21] D. Wishart, “Mode analysis: A generalization of nearest neighbor which reduces chaining effects,” *Numerical taxonomy*, vol. 76, no. 282-311, p. 17, 1969.

[22] J. Eldridge, M. Belkin, and Y. Wang, “Beyond hartigan consistency: Merge distortion metric for hierarchical clustering.” in *COLT*, 2015, pp. 588–606.

[23] J. Hartigan, “Estimation of a convex density contour in two dimensions,” *Journal of the American Statistical Association*, vol. 82, no. 397, pp. 267–270, 1987.

[24] D. W. Müller and G. Sawitzki, “Excess mass estimates and tests for multimodality,” *Journal of the American Statistical Association*, vol. 86, no. 415, pp. 738–746, 1991.

[25] R. Sibson, “Slink: an optimally efficient algorithm for the single-link cluster method,” *The computer journal*, vol. 16, no. 1, pp. 30–34, 1973.

[26] P. Ram, D. Lee, W. March, and A. G. Gray, “Linear-time algorithms for pairwise statistical problems,” in *Advances in Neural Information Processing Systems*, 2009, pp. 1527–1535.

[27] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[28] S. M. Omohundro, *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[29] A. Beygelzimer, S. Kakade, and J. Langford, “Cover trees for nearest neighbor,” in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 97–104.

[30] R. C. Prim, “Shortest connection networks and some generalizations,” *Bell Labs Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[31] D. Müllner, “Modern hierarchical, agglomerative clustering algorithms,” *arXiv preprint arXiv:1109.2378*, 2011.

[32] O. Borůvka, “O jistém problému minimálním,” 1926.

[33] W. Ackermann, “Zum hilbertschen aufbau der reellen zahlen,” *Mathematische Annalen*, vol. 99, no. 1, pp. 118–133, 1928.

[34] R. E. Tarjan, “Efficiency of a good but not linear set union algorithm,” *Journal of the ACM (JACM)*, vol. 22, no. 2, pp. 215–225, 1975.

[35] L. McInnes, J. Healy, and S. Astels, “hdbscan: Hierarchical density based clustering,” *The Journal of Open Source Software*, vol. 2, no. 11, mar 2017. [Online]. Available: https://doi.org/10.21105%2Fjoss.00205

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[37] H.-P. Kriegel, E. Schubert, and A. Zimek, “The (black) art of runtime evaluation: Are we comparing algorithms or implementations?” *Knowledge and Information Systems*, pp. 1–38, 2016.

[38] A. (Release Manager) Müller and Contributors, “Scikit-learn,” https://github.com/scikit-learn/scikit-learn, 2017.

[39] B. J. Frey and D. Dueck, “Clustering by passing messages between data points,” *science*, vol. 315, no. 5814, pp. 972–976, 2007.

[40] T. Zhang, R. Ramakrishnan, and M. Livny, “Birch: an efficient data clustering method for very large databases,” in *ACM Sigmod Record*, vol. 25, no. 2. ACM, 1996, pp. 103–114.

[41] D. Defays, “An efficient algorithm for a complete link method,” *The Computer Journal*, vol. 20, no. 4, pp. 364–366, 1977.

[42] J. MacQueen *et al.*, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA., 1967, pp. 281–297.

[43] K. Fukunaga and L. Hostetler, “The estimation of the gradient of a density function, with applications in pattern recognition,” *IEEE Transactions on information theory*, vol. 21, no. 1, pp. 32–40, 1975.

[44] A. Y. Ng, M. I. Jordan, Y. Weiss *et al.*, “On spectral clustering: Analysis and an algorithm,” in *NIPS*, vol. 14, no. 2, 2001, pp. 849–856.

[45] J. H. Ward Jr, “Hierarchical grouping to optimize an objective function,” *Journal of the American statistical association*, vol. 58, no. 301, pp. 236–244, 1963.

[46] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, “Optics: ordering points to identify the clustering structure,” in *ACM Sigmod record*, vol. 28, no. 2. ACM, 1999, pp. 49–60.

[47] G. L. Manoj Kumar, Tim Head and contributors, “Scikit-optimize,” https://github.com/scikit-optimize/scikit-optimize, 2017.

[48] L. Hubert and P. Arabie, “Comparing partitions,” *Journal of Classification*, vol. 2, no. 1, pp. 193–218, 1985. [Online]. Available: http://dx.doi.org/10.1007/BF01908075

[49] T. Liu, A. W. Moore, A. G. Gray, and K. Yang, “An investigation of practical approximate nearest neighbor algorithms.” in *NIPS*, vol. 12, 2004, p. 2004.

[50] J. Tang, J. Liu, M. Zhang, and Q. Mei, “Visualizing large-scale and high-dimensional data,” in *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2016, pp. 287–297.

[51] D. R. Karger, P. N. Klein, and R. E. Tarjan, “A randomized linear-time algorithm to find minimum spanning trees,” *Journal of the ACM (JACM)*, vol. 42, no. 2, pp. 321–328, 1995.