OPTICS CLUSTERING IN JULIA

TOM GALLIGANI*

Abstract. With the ability to automatically identify clusters in data without the user specifying the number of clusters, OPTICS (Ordering Points to Identify the Clustering Structure) is a valuable tool for automated data science applications. For example, it can be used with word frequencies in news articles to identify topics with words that often appear at the same time. It would be beneficial to be able to parallelize this algorithm to allow faster analysis of more data. Unfortunately, because OPTICS relies on sequential processing and ordering of points to cluster them, it is not directly amenable to parallelization. We propose a method to identify similar clusters as the traditional serial OPTICS by partitioning the data into subsets based on their distance from each other and clustering these subset on multiple threads before merging these separate clusters. We succeeds in finding similar clusters with a small number of parallel threads, but becomes somewhat less stable with higher thread counts. We do, however, find significant speedups with increased thread counts.

Key words. Clustering, Julia, Distributed Computing

1. Introduction.

OPTICS[1], besides being a well named algorithm with an incredible acronym, is a density-based clustering method which has the helpful property of being agnostic to the number of clusters. Traditional hierarchical or k-means clustering algorithms have no simple way to identify the number of clusters which actually exist in the data and in practice choosing the number of partitions for the data can be more art than science. This is practical when the clustering method is used to conduct exploratory analysis; however, to use clustering as a part of predictive analysis, it is helpful to have an algorithm which can automatically identify the number of clusters in a dataset.

For example, a RAND study last year which focused on identifying malicious actors in international news publishing used OPTICS to identify the clusters of words which showed up most often each day. These word clusters basically identified particular "narratives" and the resulting time series enabled the authors to compare timing and relationships comparing when different news sources focused on different narratives. The study focused on coverage of the COVID pandemic and was able to identify and characterize tactics that Russian state-controlled media outlets (eg RT, Sputnik News, TASS) used to strategically focus on particular narratives (especially conspiracy theory related content.

While OPTICS is well suited for this application since it can be used to identify clusters automatically, with large, high dimensional data to cluster, like the daily frequency of every word used in a Russian news articles over a two month period, it can take a very long time. Therefore, it would be helpful to be able to speed up the algorithm by introducing parallelism.

This paper is organized as follows. First, we provide a brief overview of the relevent features of the OPTICS algorithm in section 2, then describe our methodology and implementation in section 3, and describe the testing and verification we conducted in section 4.

2. OPTICS Algorithm.

OPTICS is a density based clustering algorithm which generates an ordering and a "reachability plot" for the points in a dataset given parameters ε and `minpts`. This reachability plot, an example of which is shown in Figure 1, effectively represents the distance to the next closest point in the dataset. In Figure 1, this shows that OPTICS has identified a single dense cluster, corresponding

*MIT, Institute for Data Systems and Society (tfg@mit.edu).

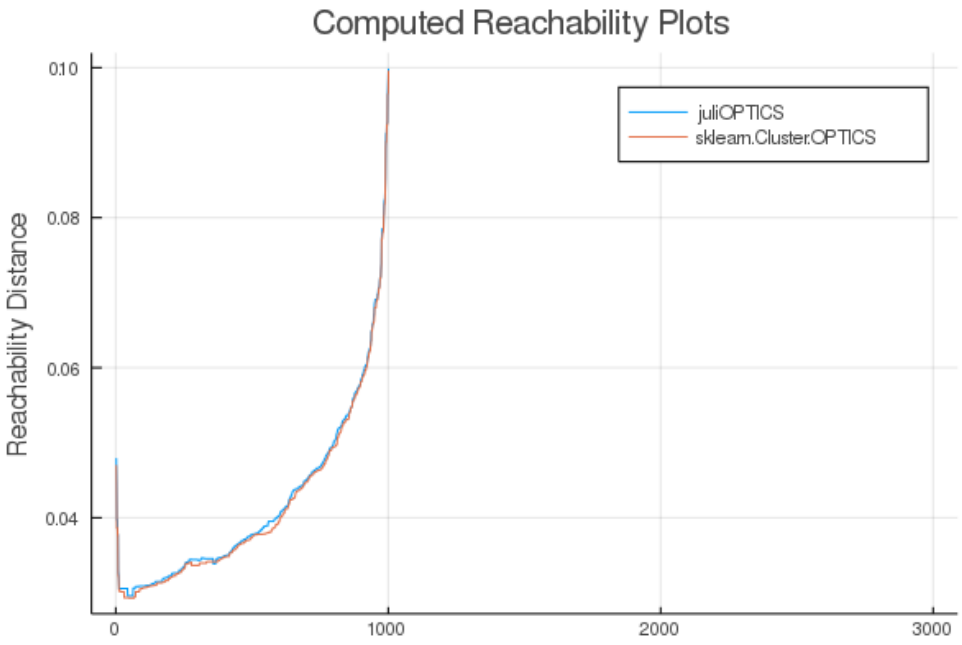


Fig. 1: Note that blank spaces indicate undefined reachability distance where points are not dense enough to be reachable

to the the valley of low reachability distance. Points which are not located in an ε-neighborhood with `minpts` other points are assigned infinite reachability distance and appear as blanks in the reachability plot.

To identify clusters from this reachability plot automatically, we use a third parameter ξ, which specifies the "steepness" of the reachability plot which defines the boundaries of a cluster. In the case of Figure 1, virtually any value of ξ would identify only one cluster - from point zero to 1,000. Points outside identified clusters, such as those with infinite reachability distances, are classified as noise.

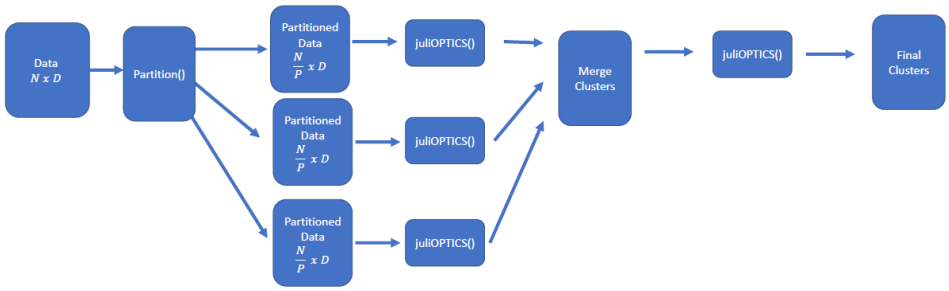


Fig. 2: Overview of faSTICS Algorithm

3. Julia Implementations. While a serial version of OPTICS already exists in python through the Sci-Kit Learn Clustering package[2], none exists in Julia's Clustering.jl. Therefore, we began by implementing a version of the algorithm in Julia, juliOPTICS. We compare the reachability plot for artificial data computed by juliOPTICS and by sklearn.Clustering.OPTICS in Figure 1. We see that these two methods produce almost indistinguishable reachability plots as they should.

OPTICS relies on the order points are processed to identify the structure of clusters in data and so direct parallelization across the the points in the dataset is unfortunately not possible; the main loop of the algorithm must run in serial. To get around this issue, we take advantage of the property of the OPTICS algorithm that it can not only place objects in clusters, but also identifies noise. We use this property to make the following assumption.

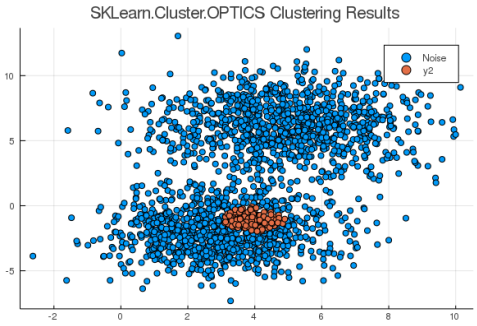
Objects in a cluster, by definition, are close to each other. We make the (admittedly strong) assumption that partitioning our dataset into P parts and using OPTICS to cluster these partitions separately, we can identify similar clusters to those identified in the original algorithm. Of course, cases on the boundary between partitions may be overlooked by this method. Consider the case where a partition bisects a cluster. This cluster, a news "topic" in our case, may not have sufficient density to register as a cluster in either of the subsets which bisect it. Clearly, it is not preferable to arbitrarily miss-identify some patterns in the data. Traditional hierarchical clustering algorithms would force these data on the edge case into one of the clusters which is more strongly represented within the partition, but since OPTICS can also identify noise - points which do not belong in a cluster - we can trust the algorithm to ignore points like this. It is, however, also not ideal to miss these clusters altogether. To ensure that clusters interrupted by partitioning are not missed altogether, we collect all points labeled as noise from OPTICS clustering each partitioned subset of the data and apply OPTICS once more to detect any such clusters which might exist in the data.

Algorithm 3.1 faSTICS Partitioning

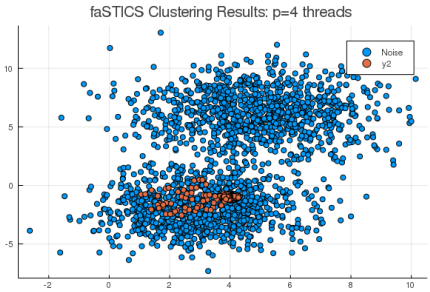
[!b]

```
function partition!(data, pworkers)
    n = size(data)[1]
    batchsize = ceil(Int, n/pworkers)
    available = fill(true, n)
    workerassignment = fill(0, n)
    idx = collect(1:n)
    tree = BallTree(data')
     for worker in 1:pworkers
        nextCenter = idx[available][1]
        neighbors= knn(tree, data[nextCenter, :], batchsize)
        workerassignment[neighbors] .= worker
        available[neighbors] .= false
    end
    return workerassignment
end
```

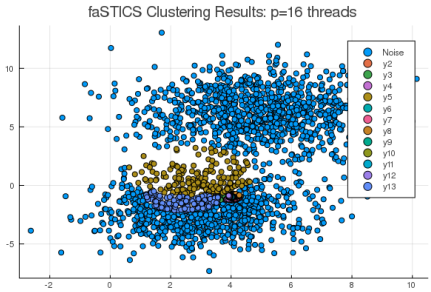
This partitioning scheme can allow us to distribute the clustering computations to multiple processes. An overview of this implementation is shown in Algorithm 3.1. Based on the number of points in the dataset `N` and number of available threads



(a) Original serial clustering results



(b) Parallel with four threads



(c) Parallel with sixteen threads

Fig. 3: More threads yielded clusters less similar to serial clusters

P, we calculate the number of points per subset required to evenly partition the points (`batchsize`).We then select the next unassigned point, calculate the `batchsize` closest neighbors and assign those to the next subset.

We use this partition to distribute each subset to a different thread and simultaneously cluster each using the regular serial OPTICS implementation. We then merge these clustering results into a single set of clusters. Finally, we once again apply OPTICS in serial to the remaining unclustered points. This algorithm is outlined in Figure 2

4. Performance Results. In this section, we discuss performance testing conducted on our implementation, first verifying that our parallelized version does not generate wildly different clusters compared to tradition serial implementations. Next, we analyze the speed of both Julia implementations relative to the python implementation in Sci-Kit Learn. For each of these analyses, we use the same process to generate artificial data: three normally distributed clusters in $\mathbb{R}^2$, center on the points $(4,-1),(3,-2)$, and $(5,6)$ with variance of $0.1, 1.6$, and 2.0 respectively.

4.1. Clustering Accuracy. To test the clustering accuracy, we relied primarily on qualitative analysis of the resultant clusters for faSTICS compared to the baseline of Sci-Kit Learn's python implementation. We performed clustering on a sample of 3,000 points with the serial python implementation and our parallel faSTICS version with 2 and 16 threads, all with OPTICS parameters minpts=50, $\varepsilon = 0.5$, and $\xi = 0.1$.

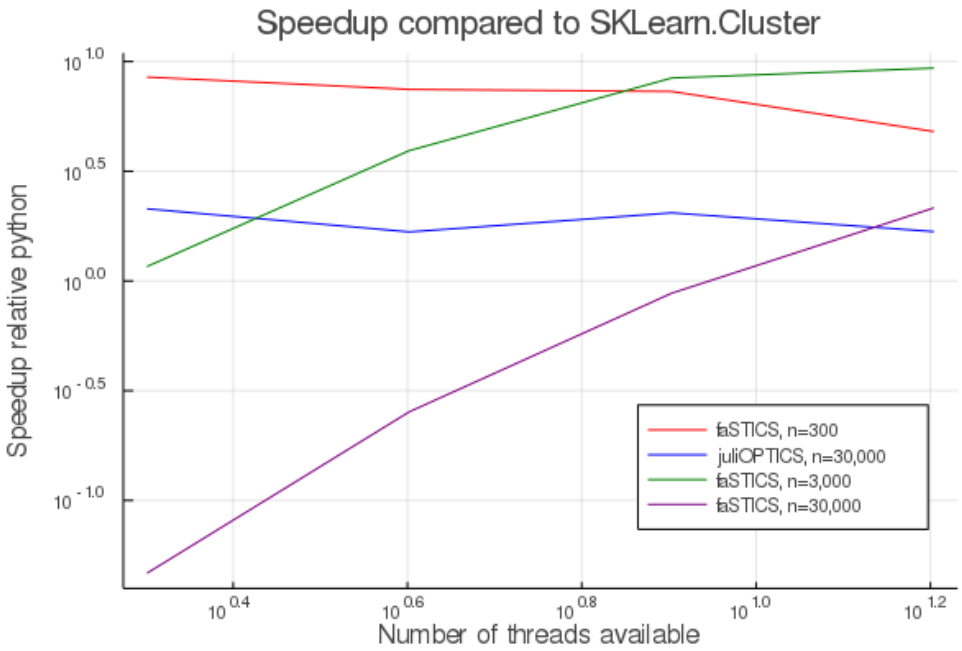


Fig. 4: As input size increased, faSTICS required more threads to improve the baseline

The results of this experiment are shown in 3, where blue dots correspond to noise and other colors to labeled clusters. We see that while our parallel version with four threads identifies basically the same cluster as identified by the serial version. We see that, unfortunately, as we increase the number of parallel threads, the clustering is less stable.

4.2. Speed. Next, we conducted experiments to determine the performance of our parallel implementation of OPTICS relative serial implementations. We compared performance based on both input size (the number of points in the dataset) and the number of threads available in parallel. For thread numbers of $2, 4, 8$, and 16, we clustered a simulated dataset generated as described above with 300, $3,000$, and $30,000$ points with the python Sci-Kit Learn serial implementation, our serial Julia implementation juliOPTICS, and our parallel implementation, faSTICS, in Julia. We measured the speed for each of these tests using Julia's BenchmarkTools package and computed the speedup gained relative to the Sci-Kit Learn version for both serial and parallel Julia versions. The results are shown in 4.

First, we see that we achieve a modest improvement over the Sci-Kit Learn implementation even with our serial implementation in Julia (shown in blue). This is in part at the cost of less memory efficiency, but we did not run into any memory issues. We can also see that with larger input sizes, faSTICS requires more and more threads to achieve a positive speedup. This is clearly a result of the overhead required both to partition the dataset and to communicate between threads. We see that using faSTICS actually results in a large slowdown relative to our python baseline. However, at 16 threads (the maximum on our machine) it showed improvement over both serial juliOPTICS and sklearn.Clusters.OPTICS, indicating that it could scale to larger

systems and acheive performance improvement.

REFERENCES

[1] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, *Optics: Ordering points to identify the clustering structure*, SIGMOD Rec., 28 (1999), p. 49–60, https://doi.org/10.1145/304181.304187, https://doi.org/10.1145/304181.304187.

[2] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, *Optics: Ordering points to identify the clustering structure*, in Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, SIGMOD '99, New York, NY, USA, 1999, Association for Computing Machinery, p. 49–60, https://doi.org/10.1145/304182.304187, https://doi.org/10.1145/304182.304187.