Evaluation of Exact Parallel DBSCAN Algorithm Using Threading and Message Passing in Python

Luke M. Unterman
Virginia Commonwealth University, Richmond, Virginia, United States of America

Abstract:

While unsupervised learning methods such as clustering can be incredibly effective at pattern recognition, the sheer size of modern unlabeled datasets necessitates that new, faster clustering algorithms continue developing. Many researchers have focused on improving the speed of the DBSCAN clustering algorithm for use with large datasets with high-dimensionality. This research project in particular is focused on replicating Mochurad et al.'s [12] implementation of an exact Euclidean parallel DBSCAN algorithm and comparing it with another exact parallel DBSCAN algorithm. Furthermore, in addition to replicating Mochurad's parallel algorithm with threads as proposed in the original paper, this research project has developed a similar parallel DBSCAN algorithm which uses a Message-Passing Interface. The results of this research project indicate that 1) Mochurad et al.'s algorithm is not as fast as Wang et al.'s [16] algorithm and 2) despite varying runtime complexities, the MPI version surpasses the Threaded version due to Python's Global Interpreter Lock (GIL)

Introduction:

Machine learning is a branch of computer science which involves identifying hidden patterns in data and using that data to classify or predict events related to a problem [2]. There are two main approaches to machine learning: unsupervised machine learning and supervised machine learning. Supervised machine learning algorithms use prelabeled data to form models which can separate unseen data into classes based on the patterns they have observed from their input data. Unsupervised machine learning algorithms use unlabeled data and, based on varying distance metrics [11], form clusters based on similarity. While supervised learning algorithms tend to classify data more accurately than unsupervised learning algorithms [5], the creation of gold-standard prelabeled data is incredibly time-consuming [7]. Prelabeled datasets also often need to be preprocessed, as null values and nominal data is not typically useful for machine learning algorithms. In contrast, unlabeled data is much more prevalent than prelabeled data. For example, in their paper entitled Twitter Discussions and Emotions About the COVID-19 Pandemic, Xue et al. [17] analyzed 4 million unlabeled Twitter messages via unsupervised machine learning. For this reason, unsupervised learning datasets have a pertinent need for fast clustering algorithms which can handle data with high-dimensionality.

Upon initial examination, the Density-based spatial clustering of applications with noise (DBSCAN) clustering algorithm is well-suited to these constraints. DBSCAN, which was developed by Ester et al, has minimal requirements of domain knowledge to

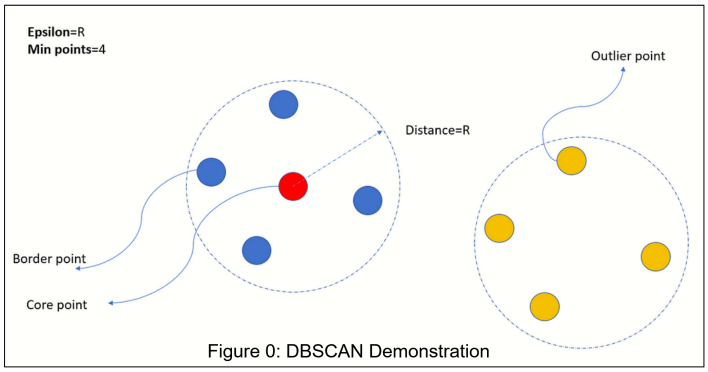

Figure 0: DBSCAN Demonstration

determine the input parameters, is resistant to noise, and has relatively good efficiency on large datasets [6]. DBSCAN's only parameters are *eps* and *minPts,* which determine 1) the range of a nearest-neighbor search and 2) the minimum points required for a cluster to be considered a core point. There are three types of classifiers for points in the DBSCAN algorithm; core points are within the nearest-neighbor search radius and minimum points values, border points have a core point within the radius but do not have a sufficient number of neighbors, and noise points do not have a core point within the radius. Core points extend the boundary of a cluster, border points define the boundary of a cluster, and noise points are ignored. A visualization of these types of points can be seen in *Figure 0* [4]**.** The classical DBSCAN algorithm has a serial time complexity under worst-case conditions of $O(n^2)$. This time-complexity was relatively fast for its time period (1996), but with the increasing size of modern datasets, researchers have since devised several ways to improve DBSCAN's time complexity.

Related Works:

Many researchers have leveraged different tree-like data structures to shorten the time required to perform range queries, which are the most computationally complex operation in the DBSCAN algorithm. In Ester et al.'s [6] original paper, they suggested the use of an R* tree [1], which they claimed could perform a range query in O(log n) time as a result of its utilization of spatial indexing. Gan and Tao [8] revisited this claim in 2015 and found that the complexity of the original DBSCAN algorithm with R* trees was still $O(n^2)$, although a genuine O(n log *n*) algorithm in a 2-dimensional space has since been found by Gunawan [9]. Gan and Tao [8] proved that for d >= 3, the exact DBSCAN problem requires $\Omega(n^{4/3})$ time [8]. However, approximate solutions can be solved in linear O(n) time [8], with the approximate DBSCAN problem being to find a set of approximate clusters of P where each of its core points has exactly one approximate-cluster [8]. Other examples of algorithms which find approximate DBSCAN solutions include LSH-DBSCAN (kd-tree) [15], BLOCK-DBSCAN [3] (cover tree), and DBSCAN with ball-trees [13].

Another avenue of increasing the speed of the traditional DBSCAN algorithm is parallelization. Parallelization allows for the concurrent execution of different parts of an algorithm at the same time for massive runtime improvements. The parallelization of DBSCAN allows for the concurrent clustering of different points in a dataset at the same time. In addition to papers on approximating sequential DBSCAN to improve runtime, there are a multitude of papers which parallelize both exact and approximate DBSCAN algorithms. Wang et al. [16] devised a number of approximate and exact parallel Euclidean DBSCAN algorithms in their paper entitled "Theoretically-Efficient and Practical Parallel DBSCAN*." In their [16] approximate n-dimensional parallel DBSCAN algorithm, Wang et al. construct a k-d tree and a quad-tree in parallel to find neighboring cells and answer range queries, respectively. In their [16] exact solution, they follow the same general structure except exact Range Queries instead of approximate Range Queries are issued at each core point. In their [10] paper entitled 'HPDBSCAN: highly parallel DBSCAN", Götz et al describe how their algorithm overcomes the sequential nature of DBSCAN through a divide-and-conquer approach while also utilizing a regular hypergrid to spatially index the data points.

However, the goal of this research paper is to evaluate the proposed solution of a different paper by researchers Mochurad et al. Their paper is much simpler, as it does not use any spatial indexing structures like kd-trees to optimize the nearest neighbor search [12]. Instead, they have proposed a nearest neighbor algorithm (Figure 1) which they claim has a worst-case time complexity of less than $O(n^2)$. This is supposedly an improvement upon the classical DBSCAN algorithm [6]. They [12] use this Range Query algorithm for their parallel algorithm to concurrently generate the nearest neighbors in the dataset and use these nearest neighbors to begin clustering points. They [12] have a parallel approach similar to pipelining, where they disproportionately allocate worker threads using OpenMp in C++ to the GetNeighbors() function while cluster threads wait until their point index has been updated by the worker threads.

Algorithm 1 Get Neighbors

```
1:  procedure GetNeighbors(start_value: int, step: int)
2:      n ← length of self.data
3:      for i ← start_value − 1 to n do
4:          start_value += step
5:          if ¬self.neighbors_found[i] then
6:              for j ← i + 1 to n do
7:                  distance ← self.dist(i, j)
8:                  if distance < self.eps then
9:                      self.neighbors[i].append(j)
10:                     self.neighbors[j].append(i)
11:                 end if
12:             end for
13:             self.neighbors_number[i] ← length of self.neighbors[i]
14:             self.neighbors_found[i] ← True
15:         end if
16:     end for
17:     return self.neighbors, self.neighbors_number
18: end procedure
```

Figure 1. Range Query Pseudocode

While Mochurad et al. [12] considered using a Message Passing Interface (MPI) in Python, they determined that the overhead would be too costly for sending messages. Instead, they used C++ threads which all have shared values, eliminating the need for any communication. Interestingly, while they provided comparisons in accuracy to other clustering algorithms, they did not compare other exact parallel algorithms. For this reason, my goals for this research paper are to **1)** replicate the algorithm with both MPI and threads in Python, **2)** determine the difference in runtime performance between my implementations and Mochurad et al.'s implementations, and **3)** compare runtime performance with other exact parallel DBSCAN algorithms.

Methods:

Just like in Mochurad et al.'s paper [12], I am using the "Top Streamers on Twitch" [14] dataset to test my clustering algorithms. This dataset contains 1000 rows of information regarding the most popular Twitch streamers, and each of the rows have 11 columns. The features extracted from Twitch include 1) *Channel*, 2) *Watch time* (minutes), 3) *Stream time* (minutes), 4) *Peak viewers*, 5) *Average viewers*, 6) *Followers*, 7) *Followers gained*, 8) *Views gained*, 9) *Partnered*, 10) *Mature*, and 11) *Language*. To prepare this data for clustering, the columns with nominal values (Partnered, Mature, and Language) are converted into categorical values with scikit-learn's LabelEncoder() function and then each of the columns are normalized via Min-Max normalization. Finally, the column containing the Twitch channel names is removed. This results in a preprocessed dataset of shape 1000 x 10 which is ready to be clustered.

<u>Threading:</u>

While Mochurad et al. did not mention the measures they took to prevent race conditions, I am fairly sure that my Python threading implementation is essentially the same as their [12] C++ threading implementation. A subset of the total threads is used to generate the neighbors for each point and save the neighbors to a shared list, while a smaller subset of the total threads is used to assign clusters to each of the points because it is less computationally complex than the generation of neighbors. Each of the threads are assigned “subsequences” of the total dataset via clever indexing; each thread has an assigned *threadIndex* and *step* value which determines how they traverse through the loops.

As mentioned by Mochurad et al. [12], the more proportional the distribution of workers is, the higher the chance of the clustering threads waiting for their next point. This is just a consequence of the load imbalance between the two tasks. A lock (although somewhat redundant due to the GIL) is used to prevent race conditions causing unpredictable neighbor counts. Still, there is very slight variance in the generated labels in some cases where a race condition occurs despite the usage of mutex locks.

MPI:

A few modifications had to be made to the threaded DBSCAN algorithm for a parallel Message Passing implementation of the DBSCAN algorithm to work. First of all, MPI processes, unlike threads, do not have shared memory. This means that some inter-process communication needs to be done for each of the MPI processes to have access to a unified *neighbors* list. This also means that pipelining is not feasible in this instance, so it makes more sense to assign all of the MPI processes to neighbor generation to improve speed. Also, instead of handling conflicting labels lists, one of the MPI processors is designated to complete the clustering section sequentially at a slight runtime speed loss. After each of the MPI processors works on their subsequence of neighbors, the MPI processor with rank 0 sequentially reconstructs the *neighbors* lists of each of the other workers so that it is in order. The *neighbors* and *neighbors_count* variables from rank 0 are broadcasted to every other worker, and then the labels are computed sequentially.

Specifications:

Each of the runtime durations will be determined by a computer with these specifications:

- CPU: AMD Ryzen 5 5600X 6-core Processor
- Memory: 16 GB 3600 MHz CL 16
- Python version: 3.11.4
- Threading library: built-in threading library from Python
- MPI library: mpi4py

Results:

Runtime complexity (threaded):

My threaded parallel DBSCAN implementation should theoretically have the exact same runtime complexity as the algorithm it is modeled after [12]. Let us assume that there are *p* threads and

that the dataset has *n* rows and *m* columns. Each thread is assigned a job, either neighborGenerator or clusterGenerator. Let us say that there are *j* numberGenerator jobs and *k* clusterGenerator jobs, where *j* + *k* = p. Each of the threads that are assigned the job of neighborGenerator are tasked with subsequences of the dataset. Taking these subsequences into account, each thread iterates through the first loop in O(n/j) time. Then, they iterate through the second loop in O(n) time.

This makes the total runtime complexity for each of the numberGenerator workers n*n/j = $O(n^2/j)$. Threads assigned with the clusterGenerator job are also equally distributed subsequences of the dataset to cluster. Taking these subsequences into account, each thread iterates through the first loop of the function in O(n/k) time. Then, each thread iterates through a while loop in *approximately* O(n) time. This makes the total runtime complexity for each of the clusterGenerator workers n/k * n = $O(n^2/k)$.

Overall, assuming that the ratio of *j* and *k* processors is optimal, the runtime complexity of the average thread worker could be generalized to approximately $T_p = O(n^2/p)$. The serial time complexity of this algorithm is still approximately $O(n^2)$. Just like the calculations from Mochurad et al.'s paper [12] , the speedup (S) can be approximated to $S = T_s / T_p = O(n^2/p / n^2) \sim= p$. The efficiency E = S / p, so (with an optimal working ratio) E = p / p = 1. Also, since $pT_p = n^2 = T_s$, the algorithm is cost-optimal. Because the efficiency of the algorithm is approximately 1, the isoefficiency cannot be calculated.

Runtime complexity (MPI):

Due to the communication requirements in my MPI implementation of parallel DBSCAN, the runtime complexity of this implementation will differ from the runtime complexity of the threaded DBSCAN implementation. In this case, each MPI processor is first assigned a neighborGenerator worker, meaning there are now *p* neighborGenerator workers. Each of the p workers is equally distributed a subsequence of the n rows in the dataset. The runtime complexity calculations have not changed for the neighborGenerator workers, so it is the same as the threaded implementation. Each MPI processor loops through the neighbor generation function in a runtime complexity of $O(n^2/p)$, since in this case *p* = *j*.

The main difference in the runtime complexity for the MPI implementation comes after the neighbors are generated. First, the MPI processor with rank 0 stitches the neighbor together in proper order with a runtime complexity of $O(n^2)$. Then, the properly stitched *neighbors* list is broadcasted alongside another list entitled *neighbors_number*, which just saves the number of neighbors per index. The time complexity for each of these MPI broadcast operations is $T = (t_s + t_w n) \log p$. Then, if the MPI processor is rank 0, it has also sequentially clusters each of the points in the *neighbors* list. The time complexity of this is $O(n^2)$. Finally, the labels are broadcast to each of the other MPI processors in a time complexity of $T= (t_s + t_w n) \log p$.

There is a somewhat substantial load imbalance in the MPI parallel DBSCAN algorithm. Using rank 0 as the “representative” MPI processor for the time complexity analysis results in a time

complexity of $T_p = O(n^2/p) + O(n^2) + O(n^2) + 3((t_s + t_w n) \log p)$. An approximation of this runtime complexity could be considered $T_p = O(n^2) + t_s \log p + t_w n * \log p$. T_s still approximately equals $O(n^2)$. This means that $pT_p \sim= O(n^2 p) + t_s p \log p + t_w np * \log p \sim= O(n^2 p) + t_w np * \log p$ when eliminating the smaller communication cost. $T_o = pT_p - T_s = O(n^2 p) - O(n^2) + t_w np * \log p$. The isoefficiency of this algorithm is equal to $O(((p \log p) / (2 - p))^2)$. The speedup is equal to T_s / T_p, which is $S \sim= n^2 / n^2 + n*\log p$. The efficiency then has to be $E = (p / n^2 / n^2 + n*\log p)$. This algorithm is not cost optimal.

Actual performance:

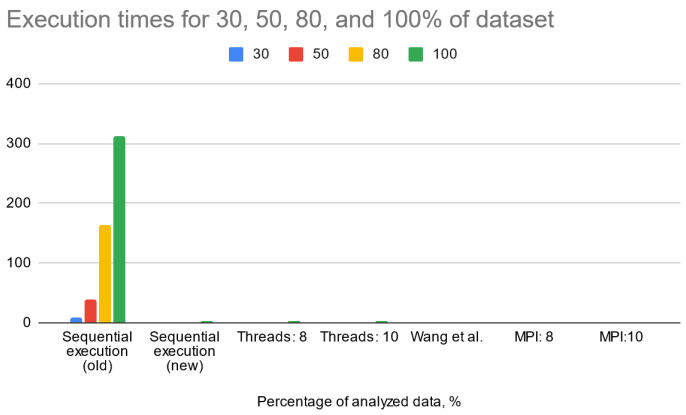

Figure 2:

What you will immediately notice from Figure 2 is that the execution times from the Mochurad et al. paper [12] are inordinately high in comparison to my Threaded and MPI implementations for 8 and 10 processing units and to the parallel algorithm that Wang et al. [16] used in their paper. This is most likely due to the sheer difference in computing power between our machines. Mochurad et al.'s computer took approximately 300 seconds to cluster the Twitch (1000x10) dataset, as they used an [12] Intel (R) Core (TM) i5-6200U CPU, 2,300 MHz, 2 cores, 4 logical processors. In contrast, (albeit while on a different machine), Wang et al.'s [16] implementation completed the clustering task in 0.06 seconds on my computer. Because of the difference in computing power in my implementation and Mochurad et al.'s implementation, the Threaded parallel DBSCAN algorithm coded in Python will serve as the stand-in for Mochurad et al.'s original implementation. Still, the Threaded implementation completed in approximately 2 seconds, while Wang et al.'s [16] implementation completed in 0.06 seconds.

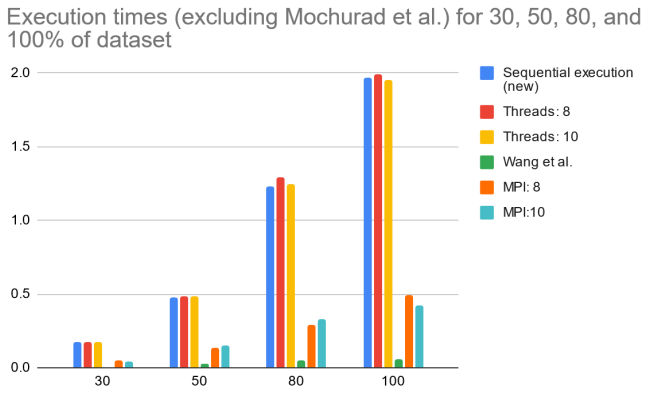

Figure 3:

After removing Mochurad et al.'s data from the plot, you can identify a few main differences between the execution times of the Threaded, MPI, and Wang et al. [16] parallel DBSCAN implementations. The first to note is that the execution time of the Threaded implementation does not substantially change between 8 and 10 threads. This is due to Python's Global Interpreter Lock which prevents more than one thread from taking control of the Python interpreter. This is also the reason why the sequential execution time is approximately equal to the Threaded execution times. Also, even though the Threaded implementation has a better runtime complexity than the MPI implementation, the MPI implementation is handedly faster than the Threaded implementation for both 8 and 10 processing units. Finally, the Wang et al. implementation is much faster than any of the code that I wrote, as it is in C++ and uses a spatial indexing structure to decrease the runtime complexity of range queries. Another reason why it is faster than the MPI implementation is that the code I wrote for MPI does not generate the labels for each of the clusters in parallel.

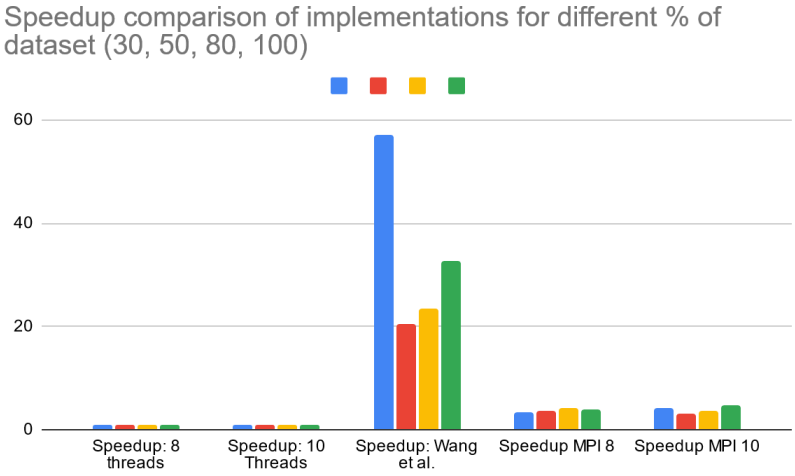

Figure 4:

The aforementioned differences in execution times are further validated by the charts in Figure 4. My Threaded implementations have somewhat equal speedup values despite the difference

in number of threads, Wang et al.'s [16] speedup values dwarf the competition, and the MPI implementation shows a slight difference in speedup for 8 and 10 processing units.

Conclusions:

In conclusion, this research paper delves into the optimization and parallelization of the DBSCAN algorithm, describing how improvements in runtime execution can be made with the use of spatial indexing structures like kd-trees and threading/message-passing. Specifically, this research paper is focused on the exact Euclidean parallel DBSCAN implementation from Mochurad et al. [12] which utilized a clever range query algorithm and pipelining techniques to parallelize DBSCAN in a cost-optimal manner. The purpose of this research project was to replicate the algorithm with both MPI and threads in Python, determine the difference in runtime performance between my implementations and Mochurad et al.'s implementations, and compare runtime performance with other exact parallel DBSCAN algorithms. After carefully considering the runtime complexities of my own parallel DBSCAN implementations, it was determined that my Threaded implementation was cost-optimal, just like that from Mochurad et al., while my MPI implementation was not cost-optimal as a result of required communication between processing units. The overall runtime complexity of the Threading implementation when given an optimal distribution of different workers was $O(n^2/p)$, while the runtime complexity of the MPI implementation was $O(n^2) + t_s \log p + t_w n * \log p$. Furthermore, because Mochurad et al. did not compare their parallel DBSCAN algorithm with other exact parallel DBSCAN algorithms, I also compared the performance of the Mochurad et al. implementation with Wang et al.'s implementation. Even while factoring in the difference in computing power when testing Mochurad's implementation, it seems like Wang et al.'s implementation was much faster as a result of spatial indexing structures. The two parallel algorithms I devised were also compared, and it was determined that despite the difference in runtime complexity between the Threaded version and the MPI version, the MPI version outperformed the Threaded version due to Python's GIL.

References

1. Beckmann, N., Kriegel, H.-P., Schneider, R., & Seeger, B. (1990). The R*-tree: An efficient and robust access method for points and rectangles. In Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data. SIGMOD 90: SIGMOD'90. ACM. https://doi.org/10.1145/93597.98741
2. Berry, M. W., Mohamed, A., Yap, B. W. Supervised and Unsupervised Learning for Data Science. (2020). In M. W. Berry, A. Mohamed, & B. W. Yap (Eds.), Unsupervised and Semi-Supervised Learning. Springer International Publishing. DOI
3. Chen, Y., Zhou, L., Bouguila, N., Wang, C., Chen, Y., & Du, J. (2021). BLOCK-DBSCAN: Fast clustering for large scale data. In Pattern Recognition (Vol. 109, p. 107624). Elsevier BV. https://doi.org/10.1016/j.patcog.2020.107624

4. Dasgupta, S. (2021, January 5). Understanding the epsilon parameter of DBSCAN clustering algorithm. Medium. Link
5. Delua, J. (2021, March 12). Supervised vs. Unsupervised Learning: What's the Difference? IBM Blog. Link
6. Ester, M., Kriegel, H.-P., Sander, J., & Xu, X. (1996). A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In International Conference on Knowledge Discovery and Data Mining (KDD). 226–231.
7. Fredriksson, T., Mattos, D.I., Bosch, J., Olsson, H.H. (2020). Data Labeling: An Empirical Investigation into Industrial Challenges and Mitigation Strategies. In: Morisio, M., Torchiano, M., Jedlitschka, A. (eds) Product-Focused Software Process Improvement. PROFES 2020. Lecture Notes in Computer Science(), vol 12562. Springer, Cham. DOI
8. Gan, J., & Tao, Y. (2015). DBSCAN Revisited. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. SIGMOD/PODS'15: International Conference on Management of Data. ACM. https://doi.org/10.1145/2723372.2737792
9. Gunawan, A. (2013). A faster algorithm for DBSCAN. Master's thesis, Technische University Eindhoven, March 2013.
10. Götz, M., Bodenstein, C., & Riedel, M. (2015). HPDBSCAN. In Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments. SC15: The International Conference for High Performance Computing, Networking, Storage and Analysis. ACM. DOI
11. Kumar, V., Chhabra, J. K., & Kumar, D. (2014). Performance Evaluation of Distance Metrics in the Clustering Algorithms. INFOCOMP Journal of Computer Science, 13(1), 38–52. Link
12. Mochurad L, Sydor A and Ratinskiy O (2023) A fast parallelized DBSCAN algorithm based on OpenMp for detection of criminals on streaming services. Front. Big Data 6:1292923. [DOI](https://doi.org/10.3389/fdata.2023.1292923)
13. Suchithra, M. S., and Pai, M. L. (2020). Data Mining based geospatial clustering for suitable recommendation system. In 2020 International Conference on Inventive Computation Technologies (ICICT) (Coimbatore), 132–139. DOI
14. Top Streamers on Twitch (n.d.). Available Online at: [Link](https://www.kaggle.com/datasets/aayushmishra1512/twitchdata) (accessed August 10, 2023).
15. Vijayalaksmi, S., & Punithavalli, M. (2012). A fast approach to clustering datasets using DBSCAN and pruning algorithms. International Journal of Computer Applications, 60(14).
16. Wang, Y., Gu, Y., & Shun, J. (2019). Theoretically-Efficient and Practical Parallel DBSCAN (Version 4). arXiv. DOI
17. Xue J, Chen J, Hu R, Chen C, Zheng C, Su Y, Zhu T. (2020). Twitter Discussions and Emotions About the COVID-19 Pandemic: Machine Learning Approach. J Med Internet 2020;22(11):e20550. DOI