DBSCAN on spark

Yotam Lev (315870964), yotam.h.lev@gmail.com, Reichman University, Israel

February 18, 2023

Abstract

Unsupervised or semi-supervised clustering of data is an important application in modern large-scale services. For example, traffic GPS patterns can serve to detect hot spots, traffic jams, and unusual events as well as better allocate ride-sharing and ride-hailing services. The challenge is that popular clustering algorithms such as DBSCAN need a parallelization scheme to be usable for large datasets with 10's of Millions of records or more. We implemented two parallelizations: the first of DBSCAN itself, and the second of connected components. We calculate expected complexity and confirm them with empirical measurements on several datasets.

1 Introduction

In the convention PyData 2022 Tal Erez Hauer speaker talked about difficulties surrounding Data Science projects in industry. While there are many possible reasons, an interesting one for fellow data scientists is scale difficulties. Modern business applications need to process millions and billions of data points, in real time, while requiring costly resources. This can make manny academic innovations impractical for business use.

In Erez Hauer's case the company was PayPal and the need was huge: 400M PayPal transaction records. They wanted to cluster these using DBSCAN, a popular clustering algorithm.

DBSCAN is popular due to several advantages over other algorithms, such as k-means or Gaussian misture models:

1. It doesn't require pre-determination of the number of clusters in the data. Clustering is a form of unsupervised learning, and can be used as the first step in processing a new, unfamiliar dataset. The less of our preconditioned human biases we put into the model, the better.
2. DBSCAN is able to outline complex and concave clusters, because it doesn't have centroids. Instead, it follows around "density" wherever it may lead, and parts between clusters by defining "empty" and "outlier" areas. Thus DBSCAN doesn't impose assumed topoligies on the data.
3. DBSCAN is able to classify outliers as such, and doesn't let them impact the clusters formed.

So how does DBSCAN achieve this? The mathematical definition is quite simple:
We define a mathematical object in the data called a "core point". A "core point" is a point in space, without loss of generality we'll consider two-dimentional space, that satisfies the following condition:
$\forall (p_1, p_2) \in R^2 : \exists \{(x, y)\} : |\{(x, y)|D((x, y), (p_1, p_2)) < \epsilon)\}| > n$.
With (p_1, p_2) being the point considered as "core", ϵ being the minimal distance, D being a measure of distance, and n being the minimal number of points required to be considered "core".

As we can see, three hyper-parameters arise from this definition: n, ϵ, and D. Since D is commonly taken as the Euclidean distance, the former two are the main consideration.

Next, the algorithm searches for other reachable core points from the core point. Two core points are considered "reachable" if there exists a path, $p_1, p_2, ..., p_N$ such that for each p_n point, $D(p_{n-1}, p_n) < \epsilon$ and $D(p_n, p_{n+1}) < \epsilon$. There is a distinction between types of reachability and density connected-ness that are unimportant for this article.

The output of DBSCAN is to divide the data into clusters and anomaly points: A cluster contains all points that are reachable from a core point, and only those points. Anomaly points are unreachable from core points. Points that are part of a cluster but not core points themselves are labeled "border points".

There are many different algorithms who's output achieves these mathematical definitions: [......]

2 Parallelizing DBSCAN

As we've seen in the previous section, DBSCAN requires two different phases:

1. Determining which points are core points
2. Connecting all reachable core points to each other

In both phases points are useless by themselves: they cannot be processed without the greater context of points near them, or core points reachable by them.

We've mentioned in the introduction that this problem was encountered as part of a business project. The linear algorithm cannot run on the big datasets within reasonable time complexity constraints. Therefore the need to parallelize it. But these business requirements imply that we are willing to make some changes to the mathematical definitions in order to achieve realistic performence.

2.1 Using cells instead of points

The first to suggest this was Gunawan in "A faster algorithm for DBSCAN." (2013). The article proposes to divide the space into cells with size epsilon. Now the whole dataset can be divided into worker nodes: the point comes with its general neighborhood, allowing each worker node to determine whether it satisfies the conditions of a "core point". Since the cell is of size epsilon, we can determine weather the entire cell is "core" or not.

There is a downside: The cells are dividing the data arbitrarily. We illustrate an example of a "non-core" classification of points that in the original definition would have been classified as "core points":

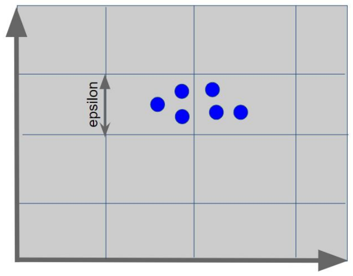

Figure 1: The cell sides are epsilon long.

If the number of points needed to be considerd "core" is 4, then the seperation of this cluster into two cells will lead to it being classified as non-core. However, working with massive real datasets, we believe such cases do not impact the final outcome.

2.2 Parallelizing cells

In the first stage of the algorithm, we perform a "classic" MapReduce: the mapping function assigns each data point to a cell. Then the reducer sums the number of points in each cell. If the number is above the minimal points needed, the cell is considered "core", and goes to the next stage.

article

```
def mapfunc(point):
    x_cell = math.floor((point[0]-x_min)/epsilon)
    y_cell = math.floor((point[1]-y_min)/epsilon)
    return [(x_cell,y_cell),1]
```

2.3 Connecting core cells

Now we have our core cells, but now need to determine which are "reachable" from each other, and unify them into clusters. This can be solved using the Connected Components algorithms, from graph theory. As in the case of DBSCAN there exist many single-threaded algorithms, like Breadth First

Search. We used the distributed algorithm based on the MapReduce approach: the one offered by Raimondas Kiveris et al. from Google in their paper "Connected Components in MapReduce and Beyond", published in 2014.

In it, the authors describe two operations which:

1. Perserve component connectivity, as proven in Lemmas 1 & 3 in the paper
2. Do not increase the number of graph edges, as proven in Lemmas 2 & 4 in the paper

These operations are the algorithms presented as "Small Star" and "Large Star". Their role in the grand algorithm is to skip "intermediary" nodes and connect all of the components directly to a single node. This relies on all nodes having labels that have a Strong Order Relation defined on them. In our implementation, the labels were the cell coordinates, and the relation was the lexicographic ordering.

The authors provide two algorithms which implement these "star" operations in a certain order, and guarantee convergence.

For our work, we chose the algorithm called "Alternating Algorithm", where successive Large-star and Small-star are performed in a loop until convergance.

For example, here are my small-star mapper and reducer:

```
def small_star_map(edge):
    # This function returns the edge aligned to hold the larger label first
    if edge[1] <= edge[0]:
        return edge
    elif edge[1] > edge[0]:
        return (edge[1],edge[0])

def small_reductor(key_val_tuple):
    nodes = [key_val_tuple[0]] + key_val_tuple[1]
    # the tuple is (key-node, list of connected nodes)
    minimal = min(nodes)
    nodes.remove(minimal)
    return [(i,minimal) for i in nodes]
```

3 Complexity

3.1 Cells

If n is the number of datapoints in the set, we first divide them into cells, with time complexity of $O(n)$. This operation is parallelized, so the time is divided by the number of worker nodes. From now on, all calculations are independant of n, and deal with the number of cells, k. For this reason our method is better suited for lower-dimensional data.

3.2 Connected components

If d is the dimensionality of our dataset, then each datapoint is a vector of d length. We ensure the data is normalized so that each of the d coordinates of the datapoint vector falls in the region $[0, 1]$. Hence, all the data falls within a hypercube with sides the length of 1. Hence, if each edge of our cells is of length epsilon, then we have in total $k = \frac{1}{\epsilon}^d$ The number of cells is independent of the size of the data, instead determined by the size of epsilon. In this work we limit ourselves to 2-dimensional datasets.

If k is the number of cells, then what is our worst-case scenarion for connected components? The important measure is the diameter of the largest component in the graph: If p is the node with lowest lexicographical value (meaning it's on the lowest-leftest side) in the component and s is the point in the component whose shortest-path from p is the longest, then the distance between p and s is the diameter. Each round of the Alternating Algorithm reduces the diameter by 1, connecting an intermediary node with the node "downstream" from it. Thus, We can only have $O(k)$ round of MapReduce operations. Let us first demonstrate with a cluster with diameter $\sqrt{k}$: The worse case scenario is of course a diameter k cluster of cells:

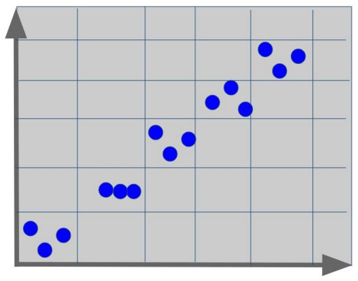

Figure 2: The core cells form a diagonal across the space, each cell is an intermediary.

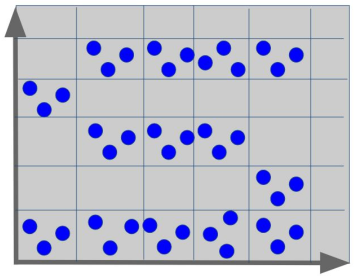

Figure 3

4 Validation

In order to check for algorithm correctness, we ran several tests:

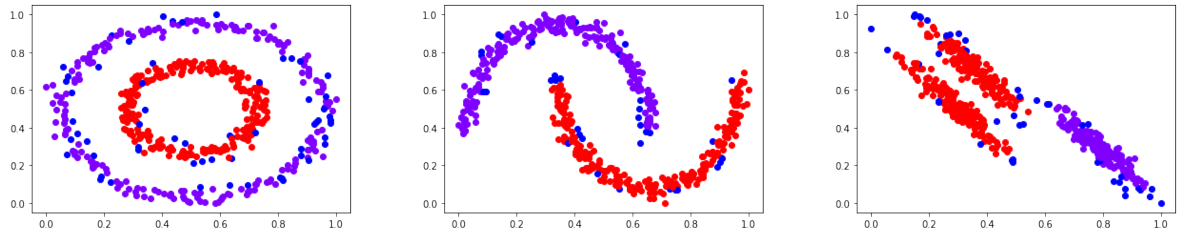

Figure 4: Three simple data spreads

We can see how (c) might benefit from lowering the epsilon value:

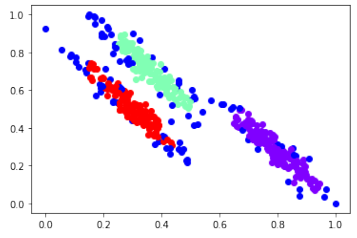

Figure 5

We should notice that not all datasets benefit equally from the methods described here. Data that has high dimensionality will most probably be weighted down by the number of cells and their potential neighbors, as well as the risk of sparsity. Datasets that have a high volume of anomalies at the edges of the spectrum, such as power-law distributions, will also not benefit, as the normalization might compress all the interesting data into one cluster.

5 References

Raimondas Kiveris, Silvio Lattanzi, Vahab Mirrokni, Vibhor Rastogi, and Sergei Vassilvitskii. 2014. Connected Components in MapReduce and Beyond. In Proceedings of the ACM Symposium on Cloud Computing (SOCC '14). Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2670979.2670997

Tal Erez Hauer, TEH. (2022, December 13). Unleash Big Data Clustering: Parallelize DBSCAN over 400M PayPal Records. PyData, Tel-Aviv, Israel.