Distributed LOF: Density-Sensitive Anomaly Detection With MapReduce

Group Members: Russell Davis and Marie Solman

Abstract

Density based outlier detection is best for regions with clusters of varying densities. Local Outlier Factor or LOF is one implementation of density based outlier detection that provides a numeric value to indicate how much of an outlier a point is compared to other outliers. Implementing LOF within a distributed system adds an additional layer of complexity since LOF requires all points to find its nearest neighbors from among all other points. In addition, LOF requires points to find out information from its neighbors, which is more difficult in a distributed environment than a local one. In the end, we implemented a performant distributed LOF in a series of six MapReduce functions. Our performance metrics indicated that distributed processing achieved some improvements in stages that required point comparisons, but full-scale testing was inhibited by limitations to our computing power. Some potential improvements are presented.

Introduction and Background

Outlier detection as a data mining technique sees use in a variety of contexts, including finance, weather prediction, and network security [2]. In these contexts, it serves to identify contexts that are quantifiably abnormal based on a certain metric. In many cases, this metric is Euclidean distance in the case that the data can be construed as existing as a geometric point, but other metrics exist as well.

Distance-based approaches represent a hugely impactful portion of challenges that require use of anomaly or outlier detection. In big data contexts, the ability to vectorize data and interpret them as points on a Euclidean plane renders distance-based approaches highly versatile. As a result, this field has received notable attention and a wide variety of approaches exist for classifying outliers in a geometric context. We present below a brief discussion of application contexts, other approaches, and a specific technique and its strengths in the field of outlier detection.

One often-mentioned context that makes use of outlier detection is fraud detection. One such implementation of this, designed by Foster and Provost [4], uses a series of neural networks each trained on a specific purchasing pattern in a given customer's behavior. These neural networks work in parallel to determine if a purchase fits into any given pattern of behavior; if it does not, it is flagged as being potentially suspicious.

Outlier detection has also been identified as having relevance in such specific fields as sports analysis, as discussed by Knorr et al. [5]. It is also related to the approach famously described by Michael Lewis in Moneyball, which relied on identifying baseball players that were outliers in when assessing the ratio of productivity to cost.

In a simple form, outlier detection can be seen as clustering data and simply observing which points are not deemed to belong to any cluster. These points can be indicative of noise in the data, in which case identifying them accurately is a prerequisite for removal. Alternately, outliers may not be noise, and rather represent important data points that are unique and therefore merit special attention.

Therefore, outlier detection shares similar goals with cluster analysis, which groups points together based on shared characteristics. Again, as data can often be construed as existing as points on a Euclidean plane, this is comparable to seeing which points are proximal

to each other compared to other points. In implementation, this means that clusters are formed to minimize internal distances and maximize distance to other clusters.

In order to classify an outlier, an algorithm must first understand what constitutes expected or normal behavior, in order to assess which data deviates from this and classifying those data as outliers. In the above case for clustering, this means finding which data do not belong to a cluster. In other contexts, this can involve such approaches as performing statistical tests and finding the z-score of data points or using a Bayesian likelihood-driven approach.

Simple distance-based approaches include use of the k-Nearest Neighbors concept, with an outlier being any object with a distance to its k-nearest neighbor that exceeds a certain threshold. This can be modified in simple ways by changing parameters such as the value of k or by using average distance to the k-nearest neighbors. This approach struggles in cases that have clusters of varying density, as a single threshold can fail to capture the complexity of the data.

Another such approach is a density-based one, which in its simplest implementation determines the density of points around the object and terms any value that exceeds a certain threshold to be an outlier. However, this simplistic approach again fails to properly parse and interpret clusters of varying density, with corresponding shortcomings in the classification of outliers.

In both the sports-related performance assessment methodology and the fraud detection methodology discussed above, an approach can be from the perspective of density-based outlier detection. In the context of sports performance, a variety of metrics can be quantified and interpreted as being vectors in multi-dimensional space. Subsequently, individuals can be clustered based on the manner in which they perform based on the density of those points, and outliers can be detected representing their deviation from those clusters.

In the context of fraud detection, the neural networking approach can be adapted to, again, quantifying a variety of characteristics pertaining to the purchases that are made and representing them as vectors. Here, purchases that share similar attributes will again be seen as clusters in a similar space, facilitating the identification of outliers.

However, in both cases, standard approaches struggle at assessing full datasets due to the variation in density of different clusters when represented in Euclidean space. This provides some insight as to why Fawcett and Provost relied on multiple networks each trained on a specific pattern -- different patterns will yield clusters of varying density. Normal density-based approaches cannot account for this variation, and will subsequently mis-classify points that

should properly be interpreted as similar to other purchases and therefore belonging to a cluster with them. This can be seen as a shortcoming of having a fixed density threshold that is applied to each cluster, despite the variation in density.

In order to properly address data sets where anomaly detection must be performed in the presence of clusters of varying densities, the Local Outlier Factor, or LOF, was proposed by Breunig et al. [1]. In Breuing's approach, data points are assessed explicitly in the context of their neighbors. By integrating this flexibility, full datasets such as those discussed above can be assessed in their entirety without having to break them down into disparate segments or patterns.

LOF has already been employed in contexts like fault diagnosis in mechanical failure by Zeqi et al. [6]. Zeqi used LOF in tandem with back-propagation neural networks focused on analyzing failure in steel plates. Back-propagation neural networks are particularly vulnerable to data anomalies, and Zeqi's work showed that using LOF as a mitigating technique achieved significant results in performance. Back-propagation is a key part of neural networks in a wide array of applications, meaning that this approach may see further use as a result.

LOF is also seeing active use in more traditional fields making use of outlier detection, such as network intrusion detection. Lazarevic et al. showed that LOF meaningfully outperformed a variety of more standard approaches. Cybersecurity and network protection have become points of increasing concern as interconnectedness and online presence become omnipresent, meaning that LOF's usefulness in this field is likely to cause it to see continued use.

Given the demonstrated utility of LOF, and the idea that it is of special use when trying to assess a full dataset rather than only portions of one, there is value in finding ways to implement LOF on tools that are core parts of standard "big data" approaches. In contexts such as sports analysis, which does not involve real-time decision-making and can make reliable use of batch processing, the MapReduce programming model employed by Apache Hadoop can be usefully employed.

Naive Solution

Local Outlier Factor is an algorithm that is usually not implemented in a distributed environment. The steps for the algorithm are vague enough that it is easy to make a non-performant implementation, even in a local environment. The original Local Outlier Factor implementation was conducted in a local, non-distributed environment. Local Outlier Factor, or LOF, is a density based outlier detection algorithm. Each point first finds its k-nearest neighbors. After that, each point finds a k-dist to its furthest nearest neighbor. Each point then finds the largest k-dist from all of its neighbors and compares that to its own k-dist. The larger of the two values is determined to be the reachability distance between the two points. Next, each point takes the inverse of the averages of the reachability distances between it and each k nearest neighbor to find its own local reachability distance or LRD. Finally, the LRD of the k nearest neighbors are averaged and divided by the LRD of the point to find the LOF value. If a point has an LOF value close to 1, it is in a similarly dense region to its neighbors. If the LOF value is greater than 1, it is an outlier, with larger values indicating a point is more an outlier.

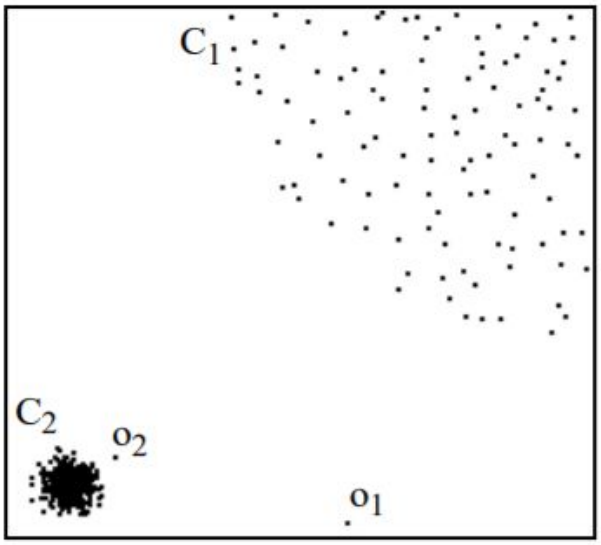


Figure 1: 2-*d* dataset DS1

[1] LOF is optimal for regions of varying densities

Another paper, distributed LOF implemented the LOF algorithm in Hadoop. In their case, because they had a large number of points to consider over a large area, they proposed two optimizations. First, they proposed a skewed partitioning to handle the skewed dispersion of points within clustered regions. This would allow them to properly load balance and ensure that one mapper or reducer was not unduly burdened.

A second optimization was a supporting area. This required there to be two MapReduce jobs that analyzed the points. The first MapReduce job took all the points and distributed them

among the mappers. Within each mapper, they found first the k-nearest neighbors per point. Second, they determined the distance between each point and its furthest k-nearest neighbor. Lastly, they determined among all the points, the maximum distance to find those k-nearest neighbors. Each mapper sent that value to the reducer to find the minimum required distance to ensure that any point could find its k-nearest neighbors. While each mapper compared every point to every other point, the fact that mappers are distributed ensures that the load is shared among many mappers. In essence, the traditionally terrible N^2 performance now became $X(N/X^2)$, a much more reasonable value.

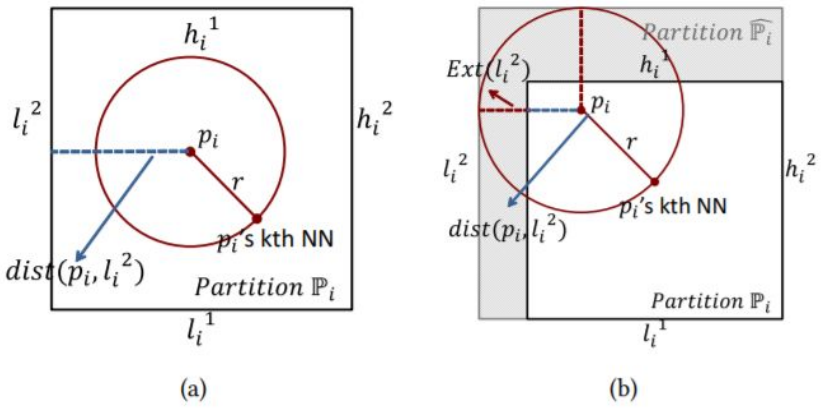


Figure 3: DDLOF: Supporting Area.

[2] Supporting area implemented by DLOF

The second MapReduce job then took this distance and used it as a "supporting area" around each point. Each point would be assigned to a cell and this supporting area would be used to send points to their neighboring cells reducers. Each reducer would then receive a list of points as well as a list of points within that supporting distance, drastically reducing the number of points to compare to each point.

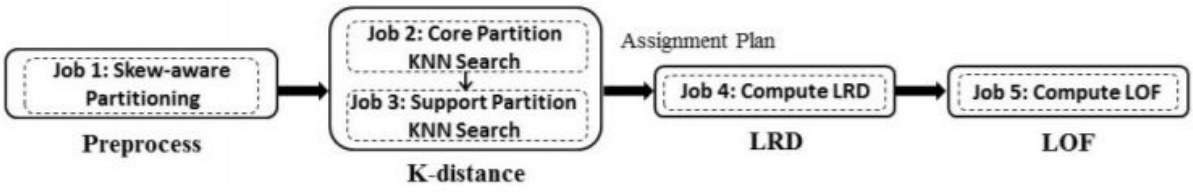


Figure 4: Overall Process of Data-driven DLOF

[2] Flow of MapReduce Jobs in DLOF paper

As we began planning out the sequence of steps for the algorithm, we identified several areas where we could simplify the implementation. This would allow quicker, easier code at the expense of performance. First, we considered one of the optimizations proposed in the DLOF paper. This optimization took all the points and computed the farthest distance required for any point to find its k-nearest neighbors. The DLOF paper then took this distance and labelled it as a supporting distance. In the next step of their implementation, they separated the points into cells and for each cell, compared its points to all points within the radius of the supporting distance.

One of the most obvious naive solutions is to skip this step and instead consider each point when looking for the k-nearest neighbors for a particular point. This might not be problematic for local implementations but it would be disastrous in a distributed environment with the scale of data often computed. The performance in this case would be terrible as it would become a N^2 problem. To note, while the DLOF paper's proposed first step does check for the k-nearest neighbors for each point in its first step, this occurs in a distributed set of partitions. If, for example, the points are sent to four partitions, the performance is now a $(N/4)^2$ problem, a significant improvement.

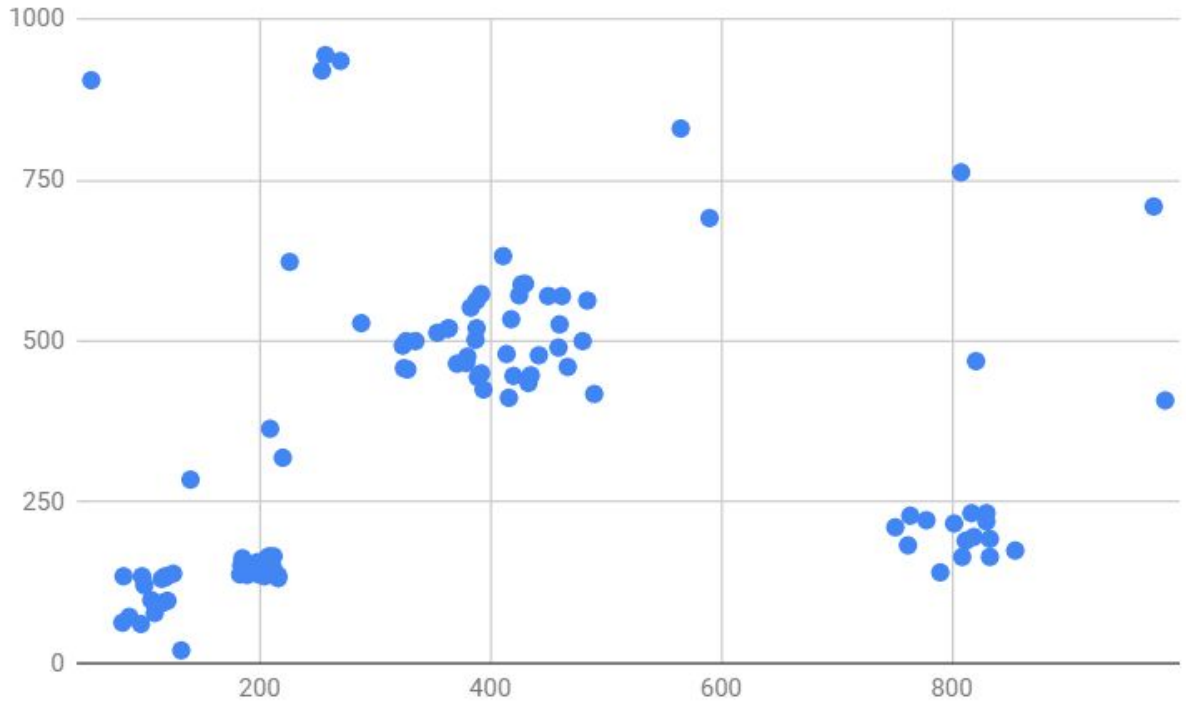
In the second mapreduce step of our solution, we planned to implement the DLOF paper's optimization. We planned to divide the points into cells and only compare the points from one cell to the points from cells within the support distance. We considered skipping a step in our code where we would check if the neighboring cells were valid cells before sending points to the reducer. We would easily be able to tell when no core points reached those reducers that the cell was invalid and not output any point-neighbor rows. We decided against this shortcut since this would cause unnecessary traffic between mappers and reducers in the second MapReduce function.

Before deciding to use the DLOF's proposed supporting distance, we considered only checking the points from neighboring cells, as we did in the density index homework problem. We were aware that with this solution, we would have some sparse points that would not find enough neighbors. Those points would need to be set aside and repeat the same step by pulling in neighboring points from a larger area, say two cells in each direction. This step would need to repeat until every point found the necessary number of neighbors. In the end, skipping the calculation for the supporting distance would force us to do more work and repeat this step multiple times to cover all sparse points.

Proposed Solution

The code for our project is available online at "https://github.com/MarieSolman/DLOF". There is an attached file to help run each of the MapReduce functions as well as a java file to generate clustered points.

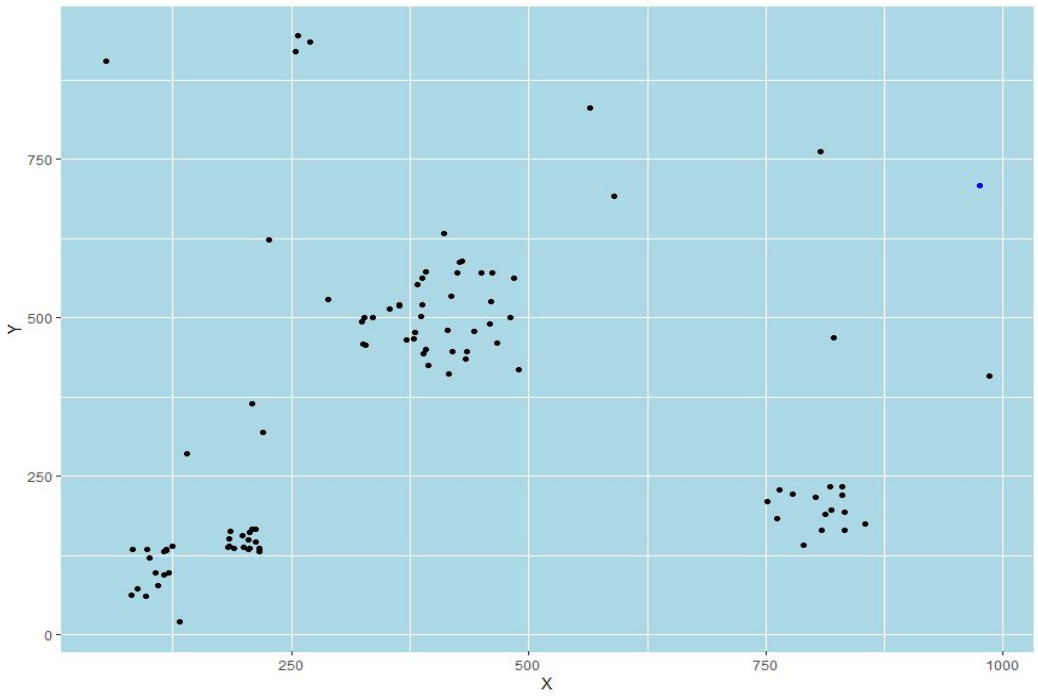
Our implementation of distributed Local Outlier Factor is separated into six MapReduce steps. Before beginning the actual MapReduce functions, it was necessary to have a dataset with decent clusters. Since LOF looks at clusters and outliers, a pure random data sample would not be helpful. Instead, we constructed a Java program that allowed a defined number of random points to generate within a defined region. In the code, we simulated clusters by defining a number of centroids and assigning each centroid a width. As points are randomly assigned to each cluster, a random x and y offset was chosen. Lastly, to ensure outliers in the dataset, some points were randomly distributed across the whole region. The resulting random points are visualized in a scatter plot below.



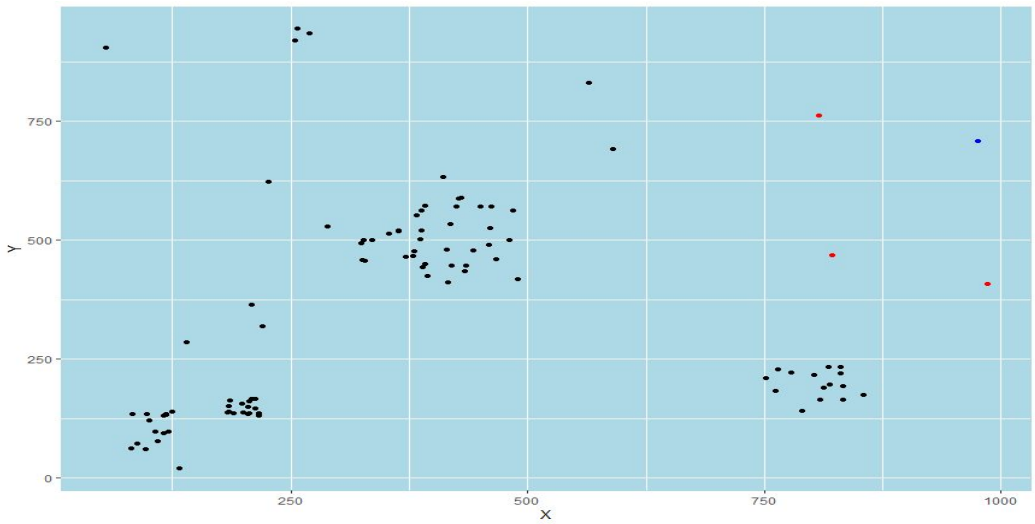
To find a reliable value for max Distance, we used MapReduce's default hash partitioning to create mappers with a random subset of the points in the dataset. We iterated through each of these to find the maximum k-Distance in each mapper, which represented a

conservative assessment. We then used a single reducer to compare the values obtained from each mapper to find the highest value of k-Distance, which guarantees that accurate results are obtained for the k nearest neighbors of each point and allows the second MapReduce job to consistently obtain the full set of nearest neighbors.

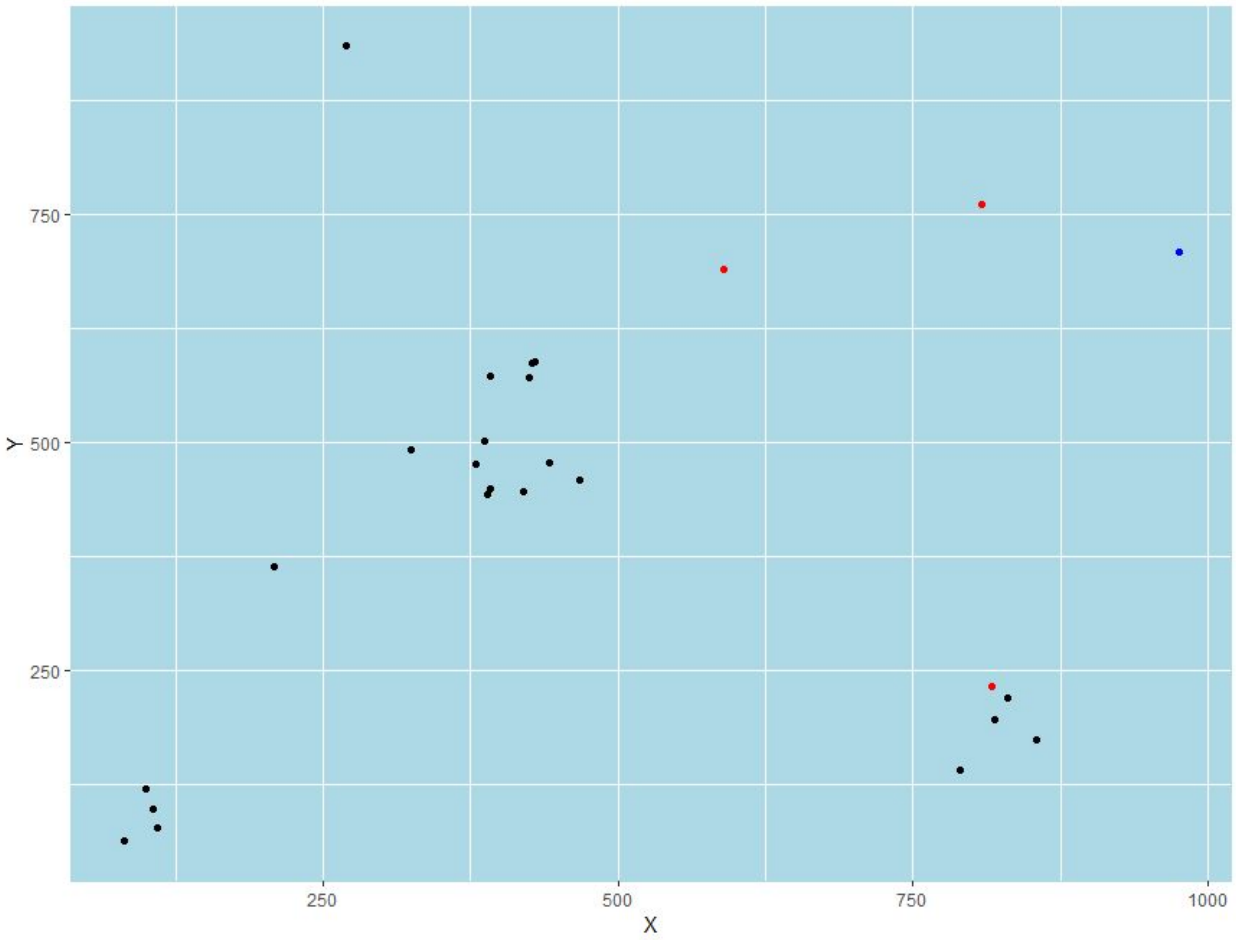
This may be more easily understood with the aid of the following figures. The first, shown below, considers a single point in the top right hand corner, colored blue.



When assessing this point's 3 nearest neighbors, we would consider the following points, now colored red.



However, in order to achieve a distributed solution, we cannot guarantee that all three of these points will be seen. Instead, we will perhaps see a dataset like the following. Again, the 3 nearest neighbors are colored red. Now, by assessing the k-distance based on these three points in a sparser data set, we are guaranteed to achieve a result that is at least as large as the true k-distance.



After taking in the dataset along with the conservative value for k-Distance, we can partition the geometric space that the dataset exists in into equally-sized grid squares. Every point will be passed to the grid square it belongs to as a core point, as well as to every grid square that contains a point for which it may be one of the k-nearest neighbors as a 'supporting point'. This supporting distance is based off of the amount of grid squares covered by the maximum distance obtained in the first MapReduce job. Subsequently, the reducer iterated through each core point for every grid cell and produced a list of the *k* nearest neighbors, taken to be 6 based on recommendation by Yan et al.

At this point in the algorithm, each point needed to find the k-dist from each of its neighbors. Unfortunately, while each point had a list of its k-neighbors, each point did not know the points that considered it a neighbor. An important distinction here is that just because a point considers another point a neighbor, that other point might not consider it a neighbor, especially when an outlier point looks at its neighbors that might be within a cluster. The third MapReduce function sets up the code to resolve this issue. Within the mapper, each point knows it must send its own k-dist to a reducer. In addition, each point will send a request for each neighbor to send their k-dist back to the requesting point. The key-value pair would look as follows: {($point_x$,$point_y$),"d"+[k-dist]} for each point and {($neighbor_x$,$neighbor_y$),"r"+($point_x$,$point_y$)} for each neighbor of those points. Each reducer would receive values based upon a point. Each point would have sent its own k-dist to the reducer but other points would have also sent a request to the reducer for this k-dist. The reducer would parse the list of values to determine if each value was the k-dist or a point. For each point, the reducer would write that point, the neighbor, and the k-dist to a file. The resulting file would therefore have a list of points and the point value and k-dist of a neighbor.

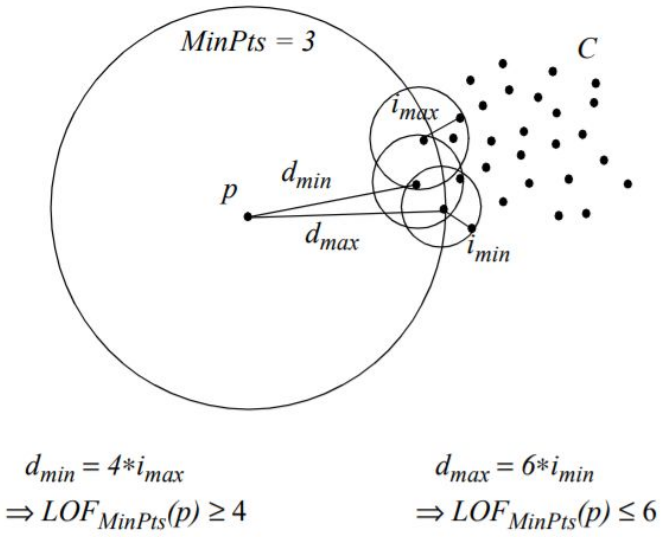


Figure 3: Illustration of theorem 1

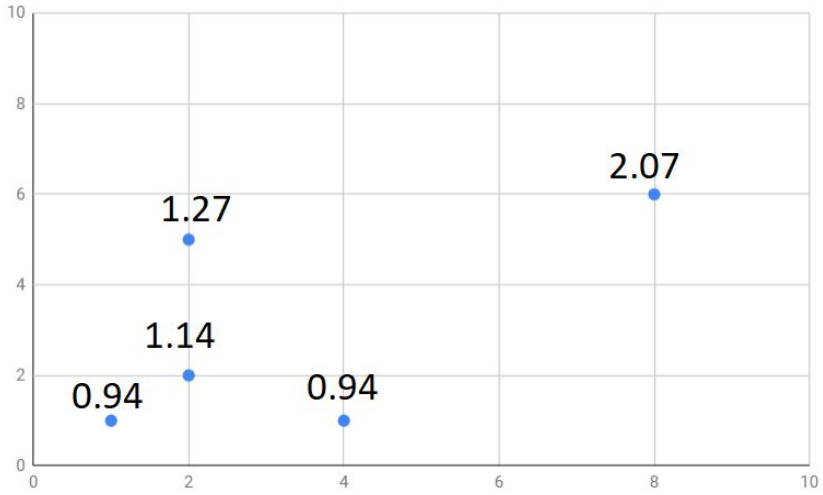
[2] LRD used to find LOF

MapReduce 4 would take the output from MapReduce 3 to compute the LRD. Each Mapper receives kdist, point, and requests for kdist for each point, outputs {p,(n,kdist)} for each requesting point. The output of the reducer is virtually unchanged from the input. The reducer takes the kdist of each neighbor and compares that value to the distance between neighbor and point. The larger of these two values is set aside for each neighbor. That value for each neighbor is averaged to find the local reachability distance or LRD. Finally, the reducer writes the point and its LRD to a file in the format "point LRD"

MapReduce functions 5 and 6 have a similar issue to 3 and 4 in that each point wants data from its neighbors. MapReduce 5 intends to output the point with LRD of each neighbor.

The mapper imports two lists. First, it imports the list of points and neighbors from MapReduce 2. Secondly, it imports the list of points and LRDs from the previous step. The two mappers map the neighbor and requesting point as well as the point and LRD to send to the reducer. The reducer receives per neighbor its own LRD and a list of neighbors requesting that LRD. After parsing the LRD and list of points requesting it, the reducer outputs each requesting point with the LRD of its neighbor.

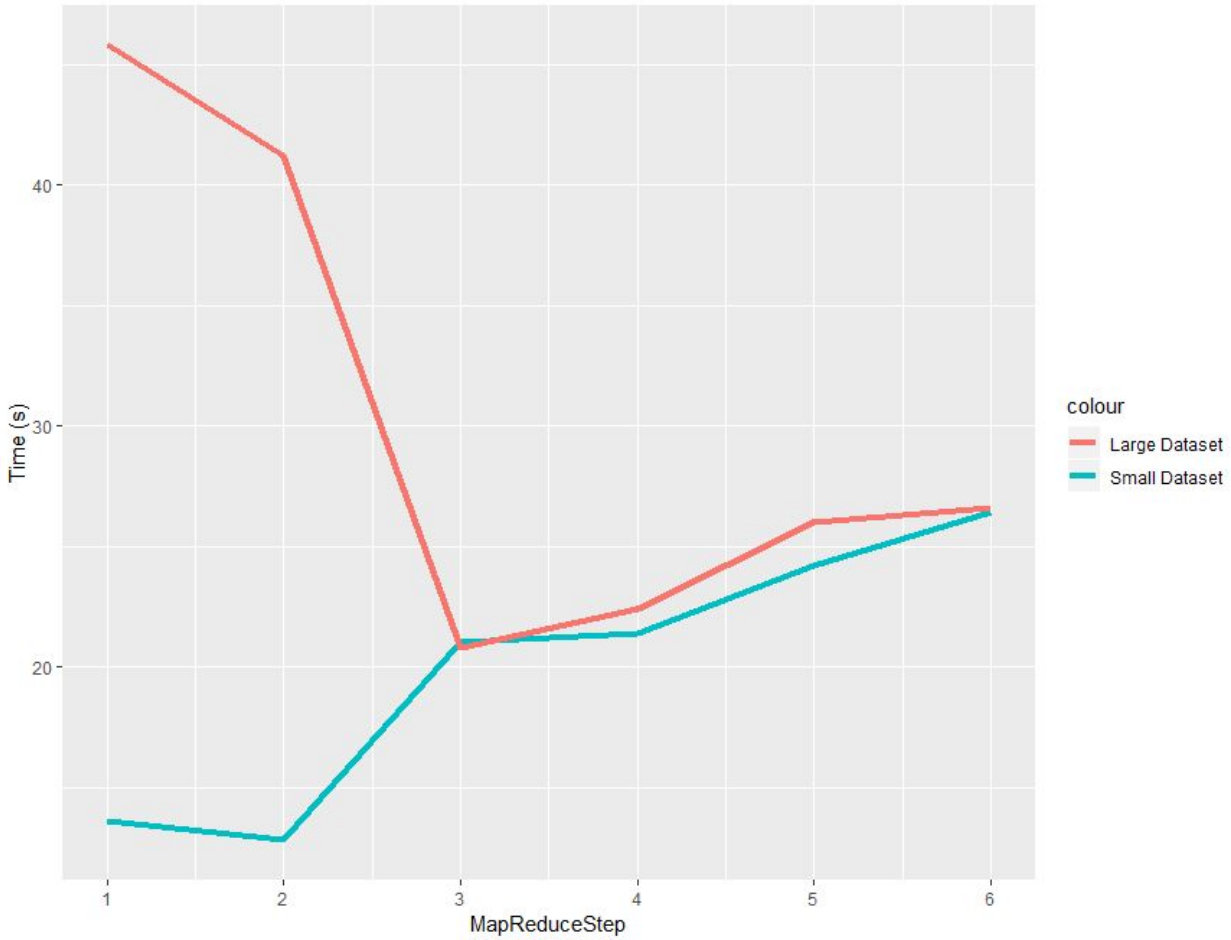
MapReduce 6 finally computes the LOF. The mapper imports the points and LRD of itself from MapReduce 4 as well as the point and LRD of neighbor from the previous MapReduce 5. This data is sent to the reducer as is. The reducer receives the point, its own LRD, and the LRDs of all neighbors. It computes the average of neighbor LRDs and divides that value by its own LRD to get the final LOF value. The final output is a list of points and their respective LOF value.



Toy set of points with LOF values

Performance Evaluation

A series of trials were taken in the distributed environment to assess the variation in performance based on dataset size. A small dataset was used consisting of 100 points randomly distributed in a 1000 x 1000 area, as well as a large dataset consisting of 1000 points in a space of the same size. Datasets of larger size were attempted as well, such as one of 10,000 points, but they took a prohibitively long time to finish given our computing resources. The average performance metrics for the two successful trials are shown below.



The initial steps, which require comparison of each point to each other point, take a significantly longer time when the dataset is increased by an order of magnitude, although the raw value is still less than a minute. In contrast, the jobs that only rely on the list of points and their k-nearest neighbors have minimal change in cost. Because those first steps are $O(n^2)$ and the latter steps are O(n), this discrepancy is expected.

Our experimentation was done using personal computers, meaning that the ability to actually achieve distributed processing was limited. We conjecture that this, along with the $O(n^2)$ nature of the first two steps, prevented the larger datasets from completing. In the event that they had finished, we would have experimented with larger areas, which we conjecture would have primarily affected the second MapReduce job that relies on grid-based partitioning. In this case, it would have required more reduce jobs, as one was instantiated for each grid cell.

Aside from boosting our computing power and subsequently our ability to parallel process, a few algorithmic improvements are readily achievable and have already been achieved by others. We use a highly conservative value for max Distance obtained from the initial job, which requires significantly more work when finding the k-Nearest Neighbors. The DLOF paper contains significant improvements that were beyond the scope of this project; however, implementing them would provide meaningful speed adjustments for the second job.

Additionally, we were limited in our ability to test a truly large dataset by our personal computers. As a result, we were not able to take full advantage of the distributed processing enabled by Hadoop. Doing so would go further to highlight the impact of our distributed approach.

From a MapReduce perspective, we also note that the DLOF paper written by Yan et al. stated that they were able to conduct LRD and LOF calculation in a single job each, as opposed to the two jobs each that we required to perform the same calculation. Condensing these calculations into half the number of jobs would also yield meaningful improvements in overall speed and efficiency, although it is currently beyond our technical acumen.

References

[1] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and JÃűrg Sander. 2000. "LOF: Identifying Density-based Local Outliers". In SIGMOD. ACM, 93–104.

[2] Y. Yan, L. Cao, C. Kuhlman, and E. A. Rundensteiner, “Distributed local outlier detection in big data,” in SIGKDD, 2017, pp. 1225–1234.

[3] P. Tan, M et al."Introduction to Data Mining (2nd Edition)". 2019. ISBN-13: 978-0133128901 ISBN-10: 0133128903.

[4] T. Fawcett. and F. Provost. "Adaptive fraud detection." *Data mining and knowledge discovery* 1.3 (1997): 291-316.

[5] E. Knorr, R. Ng, and V. Tucakov. "Distance-based outliers: algorithms and applications." in The VLDB Journal—The International Journal on Very Large Data Bases, 8.3-4, 2000, pp. 237-253.

[6] Zhao, Zeqi, et al. 2015. "Application of local outlier factor method and back-propagation neural network for steel plates fault diagnosis." In Control and Decision Conference (CCDC), 27th Chinese. IEEE, 2015.

[7] Lazarevic, Aleksandar, et al. "A comparative study of anomaly detection schemes in network intrusion detection." *Proceedings of the 2003 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, 2003