Optimization of LOF Anomoly Detection Algorithm For High Performance Scientific Computing

Dylan Slack

BS Computer Science Candidate, May 2019
Department of Computer Science, Haverford College

E-mail: dslack@haverford.edu.

April 2017

This paper, written for my final project for CS 287 at Haverford College, presents a comparative evaluation of implementations of the local outlier factor unsupervised anomaly detection algorithm. The goal at hand is to find the solution with the largest speedup from a naive implementation of the algorithm while maintaining correctness and ease of adaptability to new applications. I find that a Python implementation using scikit-learn's KDtree class to perform nearest neighbor calculations while using numpy to optimize LOF calculations performs the best for the given criteria.

Introduction

This paper proposes a more efficient implementation of the Local Outlier Factor (LOF) unsupervised anomaly detection algorithm. The main innovation is using a KD-tree to perform nearest neighbors search instead of computing a distance matrix and the optimized and flexible

implementation itself included the repository of this paper.

Related the CS 287 in particular, the prompt for this final project directs us to use the methods of parallelization discussed in class to speedup a known algorithm. This type of discussion applied to the LOF algorithm is useful and is accomplished in the project, but also the larger question remains, how can we implement the most efficient version of the algorithm? In class we discussed the limits of parallelization, meaning that converting a code to parallel may not always yield the best results. The programmer should not forget the theoretical framework to her problem and should not get swept up in applying this one method to encourage speedup. This project proves to be an exercise in just this. So while I deliver a parallelized euclidean search solution, this proves not nearly as quick as switching data types from a distance matrix to a KD tree. So while we can easily parallelize an implementation of LOF, this may not yield the most speedup.

High Level Algorithm

The LOF anomaly detection algorithm uses the concept of local density where locality is calculated by k nearest neighbors. The algorithm computes local density of all the points. Through comparison, the algorithm determines regions of high density. Those points with lower densities are considered outliers. This algorithm was first proposed by Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jrg Sander in 2000. I suggest looking through the paper for a comprehensive treatment of the material at hand.

Formal Definiton

Define $k\_distance(A)$ as the distance from point A to its k^{th} nearest neighbor. The set of k nearest neighbors is denoted by $N_k(A)$, where we can vary k to denote the number of neighbors

being examined. We then define reachability distance as:

$$reachability\_distance_k(A, B) = max\{k\_distance(B), d(A, B)\} \tag{1}$$

Which means the reachability distance of A from B is the distance from B to A but at least the distance to B. The local reachability distance is defined as:

$$lrd(A) := 1/\Big(\frac{\sum_{B \in N_k(A)} reachability\_distance_k(A, B)}{|N_k(A)|}\Big) \tag{2}$$

This is equivalent to the average reachability distance of A from its k nearest neighbors, taken as $\forall B \in N_k(A)$. Lastly we compare the reachability densities of points to their neighbors:

$$LOF_k(A) := \frac{\sum_{B \in N_k(A)} lrd(B)}{|N_k(A)|}/lrd(A) \tag{3}$$

This is the average local reachability density of the neighbors of a point divided by the local reachability density of that point. If the point has a similar local reachability density to the average of its neighbors, the value will produced will be 1. If the value is below 1, the point is in a high density area. If the value is somewhere above 1, the point can be identified as an outlier (the degree to which the outlier is defined will depend on the user's needs).

Discussion Of Parallelization

We can define a parallelization of LOF in two major stages. The main dependency in the algorithm is that we need to know the k nearest neighbors before we go on to compute the local outlier factor of each point. We can create a distance matrix to determine k nearest neighbors in parallel. It makes sense that in order to calculate the distance between two points we don't need to know the distance between another arbitrary pair of points. We can save time and space by refraining redundant computations, meaning if the euclidean distance from A to B is already done there is not point in computing B to A. Instead of performing this computation, we could

alternatively build a KD-tree out of the data. By building this tree, we automatically get K nearest neighbors. Discussion of how the KD-tree works is included later. We can't easily build the tree in parallel however. After we generate k nearest neighbors, we can compute the local outlier factor of each individual point. Lastly we simply define outliers based on the local outlier score.

Implementations

I compared four different implementations of LOF anomaly detect. I looked at an all python implementation (naive-LOF), parallelized all python implementation (parallel-naive-LO), numpy optimized implementation using euclidean distance calculations to perform k nearest neighbor search (euclidean-optimized-lof), and lastly a numpy optimized implementation but with a KD tree to replace euclidean distance search (KD-Tree Optimized). Each iteration served a different purpose to understanding the problem at hand. Naive-LOF gives a comprehensive treatment of each operation necessary to calculate local outlier factor scores. It is quite far from optimized, given that every computation is performed in python and it executes many redundant euclidean distance calculations. The parallel-naive-lof implementation splits the work of calculating lof scores between cores using multiprocessing in python in a manner similar to the method described in the discussion of parallelization section of the paper. Euclidean-optimized-lof operates in the same way as parallel-naive-lof yet performs all its computations using operations available in scikit learn and numpy. The most notable speedup is in this iteration as the algorithm begins to execute orders of magnitude faster than the previous versions. Yet we can still get more speedup. KD-Tree Optimized uses a KD-Tree to perform nearest neighbors search extremely quickly. Here we get significant speedup from the last iteration.

KD-Tree

The KD Tree data type delivers significant speedup in a k nearest neighbors search. This is because the data type is simply a binary search tree whose levels are split by dimension of the data, meaning that branches will include values who have small euclidean distances from each other. The trade off here is that the k nearest neighbors values the tree returns are estimates and may not be truly the k nearest neighbors. Observing the results returned by the KD-tree throughout test runs, the data the tree gives back seems to deliver correct enough results. In my implementation, I used scikit-learn's KD-tree class.

Speedup and Efficiency Results

I tested each of the three implementations of LOF on a fixed data set with a k nearest neighbors value of 5. I took the average of 10 runs to determine an average time for each problem size. The graph of each timing measurements and speedup/efficiency result is included in the appendix. The KD Optimized LOF implementation performed the best by orders of magnitude. Comparing KD Optimized LOF to the Naive implementation, KD Optimized LOF performed about 70,000 times quicker when identifying anomalies in a 500 point data set with two features. KD Optimized LOF when compared to Euclidean Optimized LOF performed about 15 times better on the same data set.

Conclusion

Overall I found that using a KD-tree to optimize nearest neighbors search proved to be quicker than computing a distance matrix in parallel when both functions were optimized using numpy and scikit-learn. I found that using numpy and scikit-learn to perform array computations led to the biggest increase in speedup overall. While parallel computing may lead to some speedup,

this speedup is limited to the number of cores available. So while parallel computing is an important tool, it is still only one tool. In optimizing the implementation of an algorithm, it is important to use parallel computing to attain some speedup, but it is also crucial to use an a fast language and to introduce new data types to make certain computations easier.

Appendix

Please note that all the x values of each of the graphs refer to total number of data points N while all time values are in seconds. Speedup and efficiency are measured relative to the naive implementation of lof.

Figure 1:

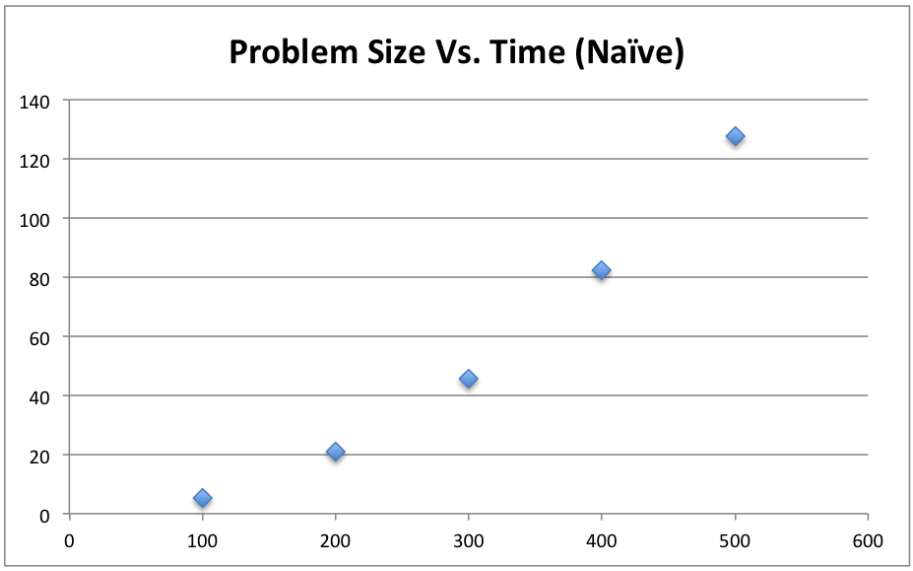


Figure 2:

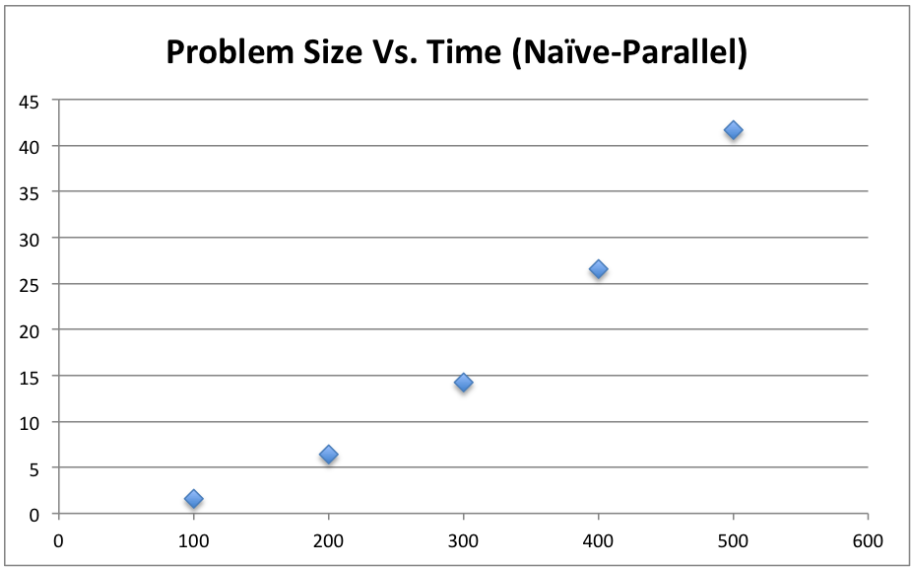


Figure 3:

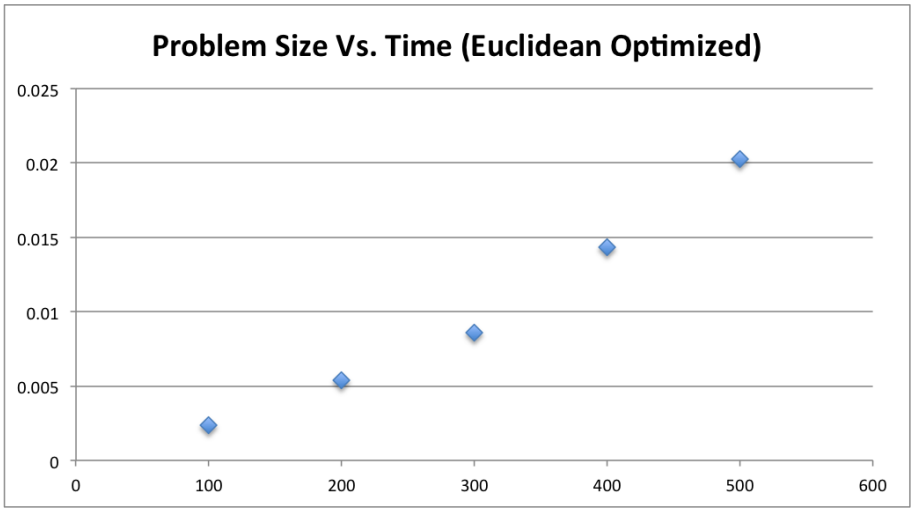


Figure 4:

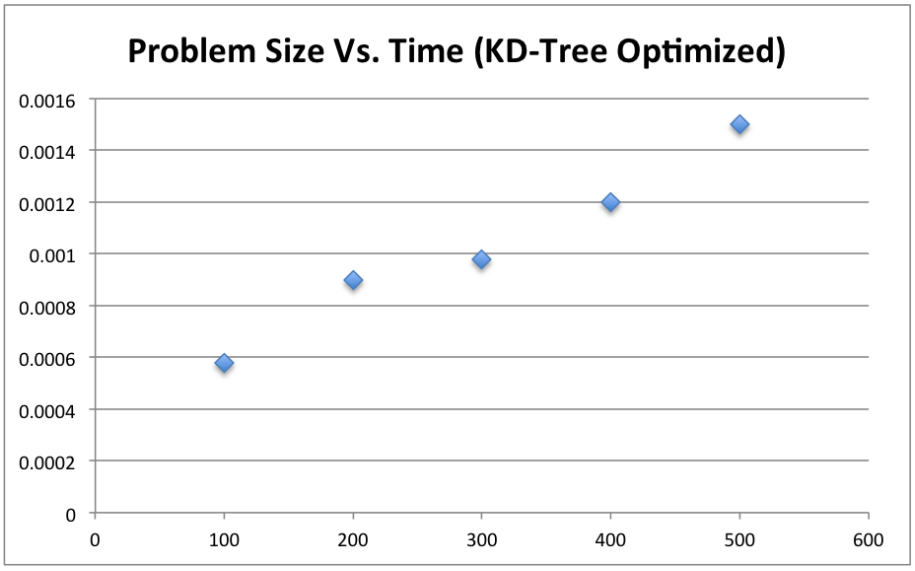


Figure 5:

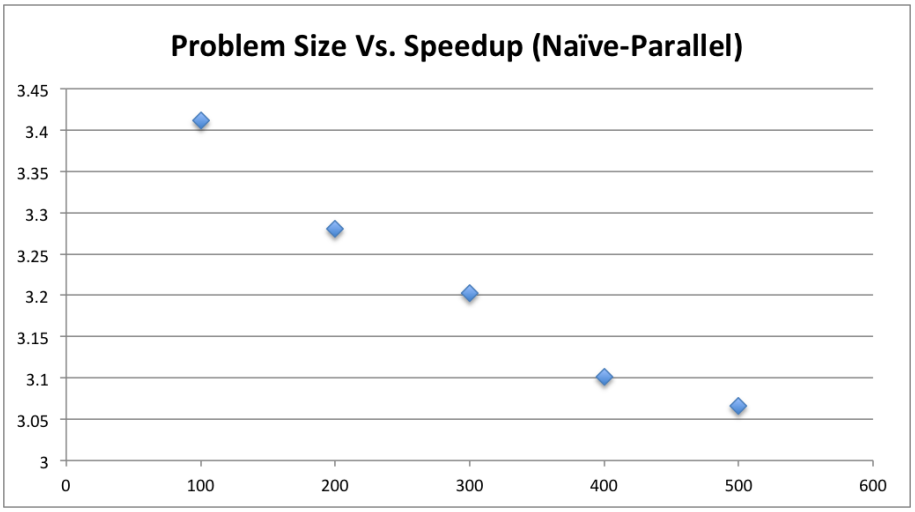


Figure 6:

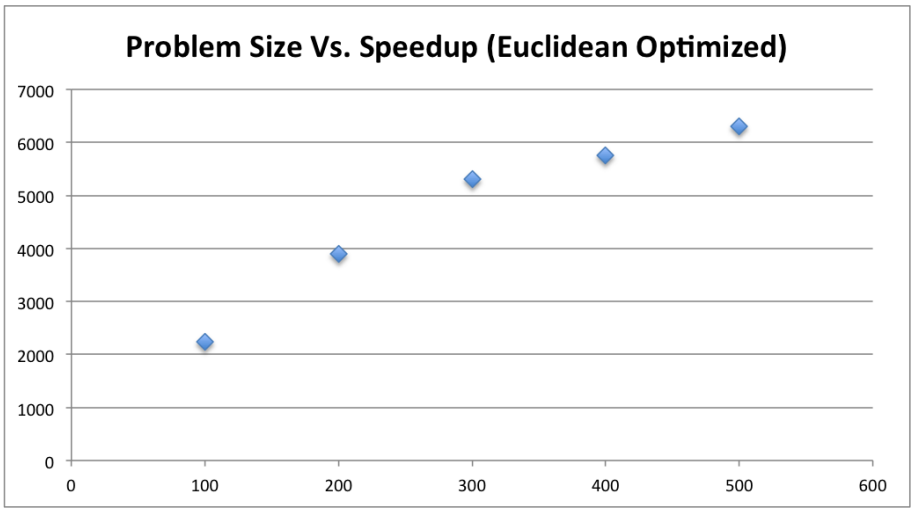


Figure 7:

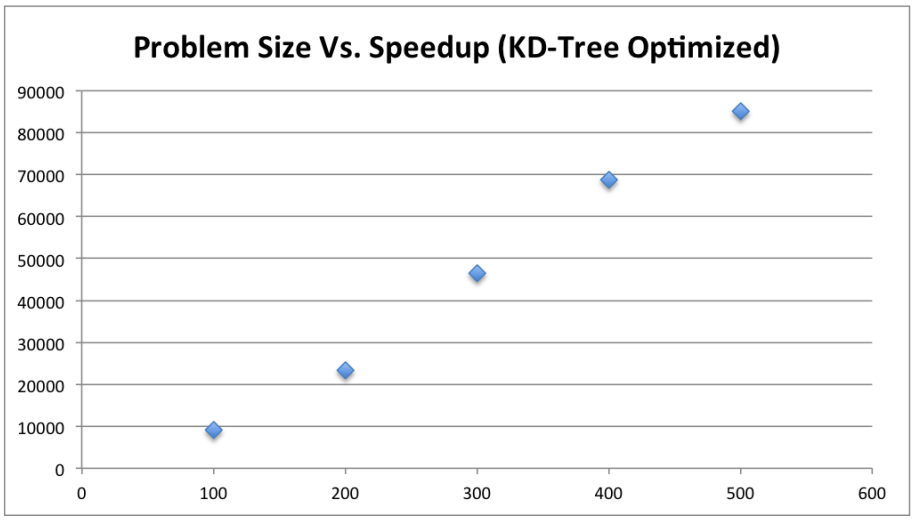


Figure 8:

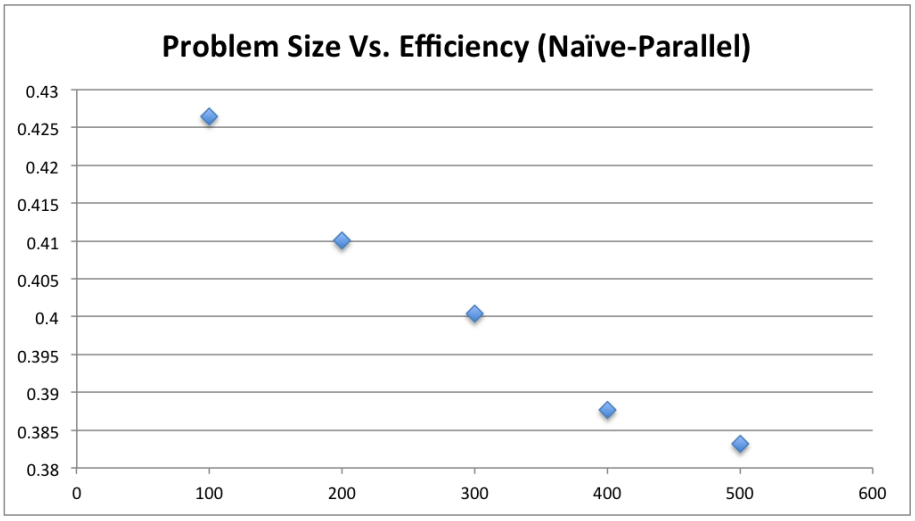


Figure 9:

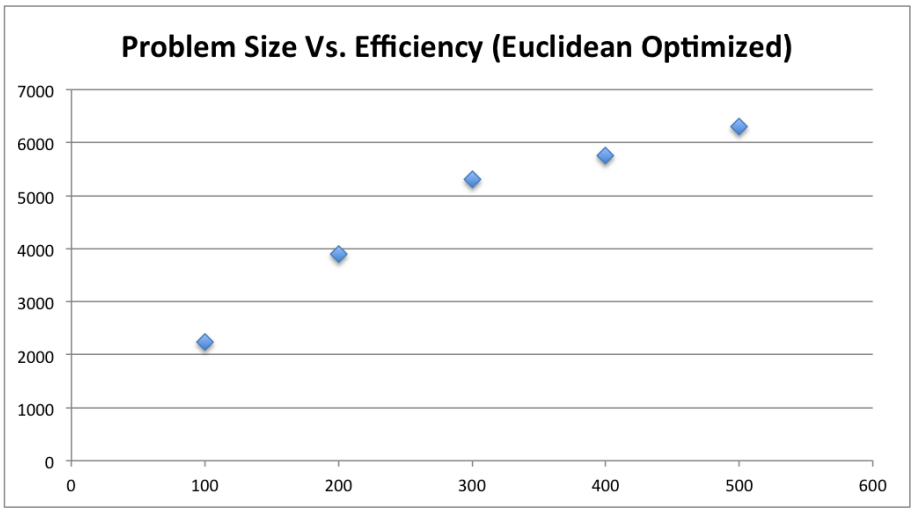
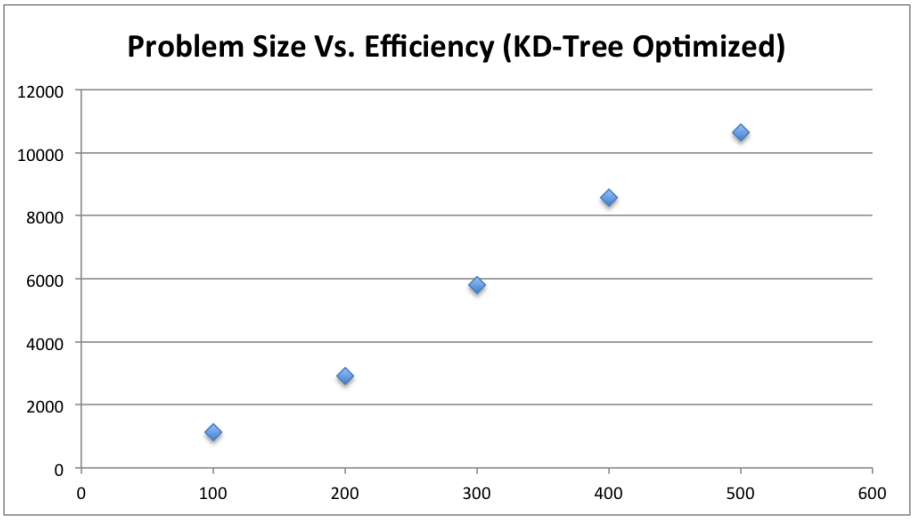


Figure 10:


Problem Size Vs. Efficiency (KD-Tree Optimized)
12000
10000
8000
6000
4000
2000
0
0
100
200
300
400
500
600

References

[1] Bista, Umanga. *Improving performance of Local outlier factor with KD-Trees*, 2014, Blog Post, http://www.bistaumanga.com.np/blog/lof/.

[2] Breunig, M. M., Kriegel, H.-P., Ng, R. T., and Sander, J. *Lof: Identifying Density-Based Local Outliers*. ACM Press, 93-104, 2000.

[3] Kunar, Damjan. *pylof*, 2016, GitHub Repository, https://github.com/damjankuznar/pylof, 2016.

[4] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.