Yinyang K-means with Scikit-Learn

Ben Seifert
beseifer@ncsu.edu
NC State University

Abstract

This paper presents an attempt to implement Yinyang K-means[5], a new algorithm for K-means clustering, as part of the Scikit-Learn package in Python3. By utilizing Cython and OpenMP multi-threading, this implementation significantly outperforms prior K-means algorithms implemented as part of Scikit-Learn. The new implementation consistently provides greater performance than prior K-means algorithms for all experimented datasets when the number of clusters is sufficiently high. This makes this implementation of Yinyang K-means an effective replacement for prior K-means algorithms for sufficiently large datasets and numbers of clusters, and potentially for smaller datasets and cluster numbers if the implementation can be improved.

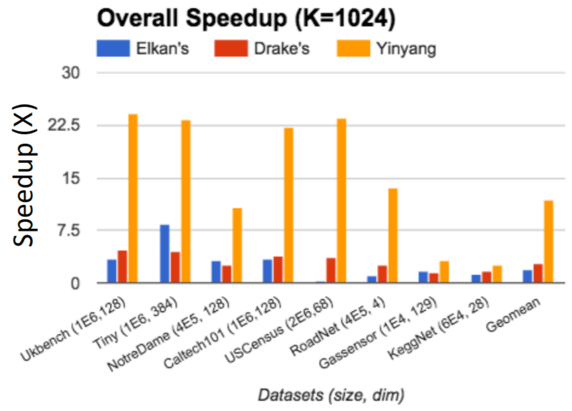

Figure 1. Experimental Speedup of Yinyang K-means vs. Elkan and Drake[5]

1 Introduction and Motivation

The goal of this project is to implement a manually optimized version of the K-means algorithm that leverages the triangle inequality to avoid a large number of unnecessary (and potentially expensive) distance calculations. The idea behind Yinyang K-means is that storing the results of distance calculations performed can allow for future distance calculations to be avoided by using the stored values to perform much cheaper operations to check whether a new distance need be computed at all.

The potential of Yinyang K-means is expressed most clearly by Figure 1, a side-by-side comparison of the speedup of Yinyang K-means over Lloyd's K-means against the speedup possible with Elkan's or Drake's algorithms. Yinyang K-means has the greatest potential for speedup among the three algorithms presented, proving that the triangle inequality is a useful property to explore for optimizing algorithms like K-means.

More generally, Yinyang K-means is a specific case of a manual optimization for one problem from a set of similar problems (K-means, K-nearest neighbors, etc.) which could benefit from triangle-inequality-based optimizations. Eventually, these optimizations could be built into compilers themselves, providing an automated means of optimizing problems like k-means using the triangle inequality. Exploring manual implementations of triangle-inequality-based optimizations could help give rise to a programmatic approach to triangle-inequality-based optimizations, which could be incorporated into compilers.

2 Objectives

As noted in the introduction, the goal of this project involves the implementation of the Yinyang K-means algorithm as part of Scikit-Learn, a Python library. This requires the addition of a new option for the "algorithm" argument, "yinyang," which sets the algorithm to be used as Yinyang K-means. This implementation must also include a new argument, "maxmem" for the KMeans class in sklearn.cluster. The "maxmem" argument may be given in units of bytes, kilobytes, megabytes, gigabytes, or terabytes, and specifies the amount of memory that Yinyang K-means is allowed to use. This argument is ignored for any algorithm type that is not Yinyang K-means.

The secondary objective for this project is to try and make the implementation of Yinyang K-means as fast as possible. Scikit-Learn itself already builds a number of its modules using Cython, which allows for code to be written in Python syntax which is then automatically converted or translated into C or C++ code. Thus, optimizing the implementation of Yinyang K-means for this project is possible using Cython to write C++ code that is as close to the code provided (written in C++ with Graphlab) as posisble. Using Cython and OpenMP also allows for the use of multithreading, which provides additional opportunities for optimization.

The last objective for this project was to test the newly implemented Yinyang K-means algorithm against the existing implementations (Lloyd's and Elkan's K-means algorithms), and compare the performance, as well as document the final product. Details concerning the three objectives given above are discussed in the remainder of the report.

3 Challenges

The first major challenge was understanding the source code provided. I have little experience working in C++ and no experience with Graphlab, which made reading through the provided code challenging. To solve this, my main approach was reading and re-reading both the research paper provided and course slides on Yinyang K-means to improve my understanding of the algorithm, so I can better understand what the provided code is attempting to achieve and why.

The second major challenge was finding a way to translate the source code provided from C++ and Graphlab into a solution that did not involve map reduce operations, because Python and Cython did not have native support for map reduce. The challenge here was that I have no experience with map reduce, so the process of translating distributed map reduce operations into single-threaded linear operations efficiently was difficult.

The third major challenge was code optimization. When I first tested the algorithm, on the Kegg dataset, it regularly required approximately 20-22 seconds when finding 4 clusters (from the results of 10 runs of the algorithm with different initial clusters), whereas both Lloyd's and Elkan's algorithms required closer to 3-4 seconds. As a result, significant work needed to be done to optimize my implementation. This was difficult, however, because this is the first time I have written anything in Cython, and I have very little experience with C++ or OpenMP, so my lack of experience presented a definite challenge.

The fourth major challenge was attempting to preserve efficiency of my code when converting back and forth between Python and C++ data structures. Cython allows for C/C++ functions and data structures to be defined in Python syntax, but Cython only exposes Python functions to outside modules, so the function where most of my logic resided was a Python function which called various C++ functions. These calls, however, required any Python data structures to be coerced into C++ data structures before the function could perform its computations. This contributed significantly to the overhead of my implementation.

The final major challenge was the size and availability of the testing datasets. Because of the time I had available, I was only able to test my implementation against two of the datasets used in the original research paper from Dr. Shen, the Gassensor and Kegg datasets. Due to the other challenges I'd encountered, my final implementation wasn't completed until very close to the end of the semester, and with testing/debugging I did not have time to do testing with the larger datasets. As a result, I only had the resources to test the algorithm on two sets of data, the Kegg and Gassensor datasets.

4 Solutions

The solution to the first major challenge I faced was to work through online tutorials on Graphlab and C++, which helped me understand the language of the original source code. After that, I was able to take a second look at the provided implementation of Yinyang K-means and understand what each instruction was meant to do. Once I understood what the original program did, this made it much easier to write a new version of the algorithm in Python (and Cython).

The solution to the second major challenge I faced was actually to avoid the problem, for the most part. Originally I was focused on trying to get rid of map reduce operations in favor of an efficient single-threaded algorithm. Instead, the solution I found to be easiest was to convert the original parallel map reduce operations (like cluster_center_reducer and cluster_center_reducer_redun) into multi-threaded functions, rather than pursue a single-threaded approach. Solving the problem in this way resulted in an efficient solution that still leveraged some of the advantages that Graphlab was able to achieve with distributed computing.

The solution to the third major challenge I faced was to read extensive documentation from Cython[2], C++[3], and OpenMP[4] to come up with ideas for how to optimize my code. The problem I initially faced was that it was taking >20 seconds to run my algorithm on one of the smallest datasets, the Kegg dataset, when attempting to find only 4 clusters. By contrast, the pre-existing implementations of K-means required only 3-4 seconds to complete their execution on the same dataset, producing equivalent clusters. In order to overcome this, I needed to familiarize myself with multithreading techniques using OpenMP, and devise strategies that would efficiently divide up work among a system's threads, without introducing race conditions or synchronization issues. Cython/OpenMP do not have a great deal of support for synchronization for the tasks I needed synchronized, so I had to use other means of dividing work than I would have otherwise. However, through some creative use of parallelism and a better understanding of Cython, I was able to reduce the runtime of the algorithm from >20 seconds to approximately 6-7 seconds during testing.

The solution to the fourth major challenge I faced was to predefine as many data structures as C++ data structures in my Python code as possible. Defining my data structures as C++ types helped reduce the burden of type coercion of many Python objects to C++ types when they were passed to the various C++ functions I wrote.

The last major challenge I faced did not have a clear solution. Because of time and resource constraints, I was forced to only do testing with the two smallest datasets, and report my results on those tests. With more time or opportunity, I could potentially acquire, prepare, and test the remaining datasets from Dr. Shen's original research paper, but as of now, there has been no opportunity to do so.

5 Lessons and Experiences

Through this project experience, I learned a number of lessons. First, I learned that the triangle inequality is a powerful tool that can be applied to a number of different algorithms, including K-means, K-nearest Neighbors, and others, when distance calculations can be expensive, and it is computationally cheaper to store and reuse information than to calculate it anew every time it is needed. As such, triangle-inequality-based optimizations could serve very well in a number of applications, and it would be extremely profitable to develop a compiler-based solution for applying this type of optimization when generating code.

Second, I learned that logical divisions in the work an algorithm must do are different from the actual experimental divisions. I learned this through my experience with multi-threading. When I implemented a multi-threaded approach to divide up the work of the "cluster_center_reducer" and the "cluster_center_reducer_redun" functions between multiple threads, one for each cluster, this did not always evenly divide the work that needed to be done among the threads created. Instead, due to the nature of the problem, some clusters would have significantly more points added/subtracted in one iteration than others, leading to inconsistent division of work between threads, and therefore inconsistent runtimes for different threads, which reduced the benefits of multi-threading. This taught me that even though a problem can be split up to multiple threads logically, it does not necessarily result in even division of work.

Last, I learned that triangle-inequality-based optimization is difficult, and I do not know how to efficiently implement the Yinyang K-means algorithm, let alone implement programmatic triangle inequality optimizations in a compiler. Working in Python and Cython and trying to use multi-threading to enhance my implementation, I realized that although theoretically a triangle-inequality-based optimization will perform better than the classic approach to a problem, implementing that triangle-inequality-based solution is a non-trivial task.

6 Results

All results were computed on a Lenovo t460p laptop with a 2.60 GHz Intel Core i7-6700HQ CPU with 8 GB of RAM. This was the machine used to test the implementation of Yinyang K-means, as well as the existing Standard and Elkan implementations of K-means in Scikit-Learn.

Due to time constraints, the only datasets from the original research paper from Dr. Shen which experiments were performed on are the Gassensor and Kegg datasets. For context, the Gassensor dataset contained 13,900 128-dimensional points[6], and the Kegg dataset contained 65,554 28-dimensional points[5]. Figures 2 and 3 show the speedup of Yinyang K-means over both the Standard and Elkan K-means algorithms for these two datasets, when k is set as 4, 16, 64, or 256. The heights of the bars represent speedup, computed as the total time required by the alternative algorithm (Standard or Elkan) divided by the total time required by the Yinyang K-means algorithm for that particular dataset and number of clusters. Below are given the average speedups for Yinyang K-means over the alternative algorithms for each dataset, computed as the simple average of the data presented in the two figures.

- Average Speedup over Elkan, Gassensor: 1.09
- Average Speedup over Standard, Gassensor: 1.49
- Average Speedup over Elkan, Kegg: 1.59
- Average Speedup over Standard, Kegg: 2.43

When collecting the data used to calculate the average speedup values above and the speedup values shown in Figures 2 and 3, I made sure to evaluate the clustering quality when running the three K-means algorithms, to ensure that the clustering results were similar enough to be considered equivalent. To this end, I used "Inertia" as a metric for cluster evaluation. Inertia as a metric is computed by Scikit-Learn after the completion of the K-means algorithm to give a means of comparing clustering results[1]. Inertia itself is computed as the sum of squared distances from each point to its corresponding cluster center. As such, inertia itself does not have units or meaning outside of its utility as a cluster evaluation metric. The best set of clusters is the set that minimizes inertia. When performing my experiments, before collecting data, I first compared the inertia of each algorithm's output clustering, and verified that they were all within approximately 1% of each other before collecting any data from that particular run. This threshold of 1% allowed me to be sure that the output of each algorithm was equivalent, so the time of execution could be compared fairly and speedup could be computed appropriately.

Considering the average speedups above, we can see that the implemented version of Yinyang K-means can indeed prove to be a very efficient replacement for either Elkan or Standard K-means. My implementation has an average speedup of greater than 1 over both of the algorithms for both of the datasets tested against, meaning that it has been proven that my implementation can operate at greater efficiency than either Standard or Elkan K-means.

It is also important to note the distinct shift in both Figure 2 and Figure 3 when going from k=16 to k=64. In both figures, for k < 16, the speedup of Yinyang K-means is less than 1, meaning that Yinyang K-means performs worse than either Standard or Elkan K-means in these situations. However, for k=64 and k=256, Yinyang K-means performs significantly better than either Elkan or Standard K-means. In some cases, Yinyang K-means performs over 3-4 times as well as Standard or Elkan K-means (Kegg dataset, k=64,256).

What we can gather from the distinct shift is this: my implementation of Yinyang K-means is well optimized for large values of k, but there is still significant overhead or

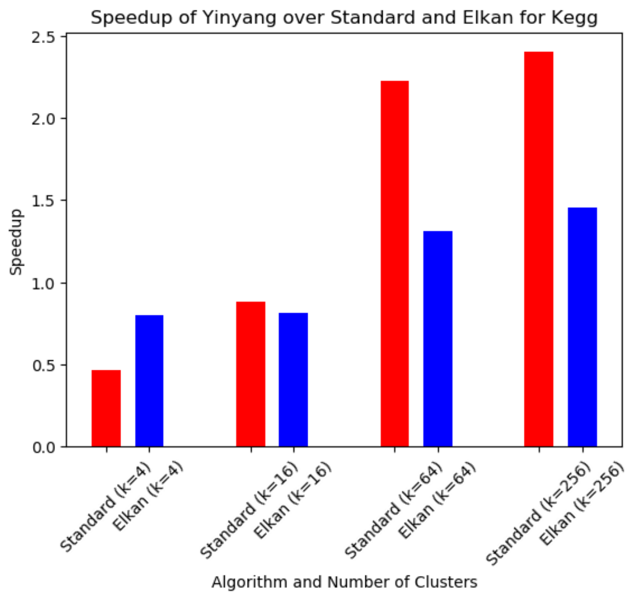

Figure 2. Speedup of Yinyang K-means over Elkan and Standard for Gassensor Dataset.

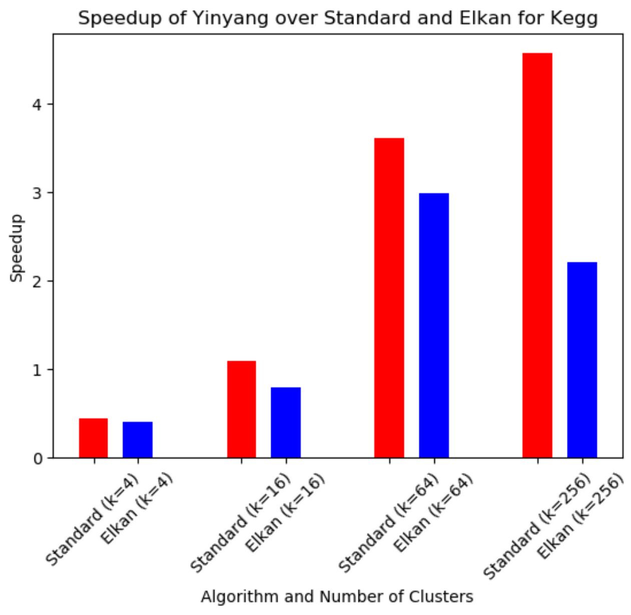

Figure 3. Speedup of Yinyang K-means over Elkan and Standard for Kegg Dataset.

other issues causing the implementation to perform poorly for small values of k. In the original implementation from Dr. Shen's research paper, Yinyang K-means outperformed other algorithms for all data sizes and values of k[5]. This must mean that my implementation is still lacking in some ways, resulting in its inefficient behavior for small values of k, but these inefficiencies could potentially be resolved with future work, resulting in an implementation closer to that presented by Dr. Shen's team.

7 Remaining Issues

One major issue remaining is the fact that my implementation is still less efficient for small cluster sizes than prior algorithms. The original research paper from Dr. Shen contended that Yinyang K-means should be a drop-in replacement for traditional K-means that produces the same (or equivalent) results, while providing speedup for any input data size, dimensionality, or number of clusters[5]. My implementation has, thus far, failed to deliver on this point. While my implementation of Yinyang K-means has proven faster (and significantly faster at that) for large numbers of clusters, it still struggles to match Lloyd's or Elkan's algorithms in terms of performance for smaller numbers of clusters.

Another major issue is that as of right now, my implementation of the algorithm does not support sparse data. Traditional K-means supports clustering for sparse data, and the implementation provided by Dr. Shen also supports sparse data, so it is intended that Yinyang K-means also supports sparse data, but as of this moment, my implementation does not have support for sparse data. This can be a problem, because it means there are some datasets for which Yinyang K-means cannot be used. However, because the implemented version of Elkan's K-means in Scikit-Learn does not support sparse data, and due to time constraints with this project, I elected not to add support for sparse data to my implementation.

A third issue is that the implementation has only been tested on a few datasets, and these datasets have been relatively small, as compared to the other datasets from Dr. Shen's original research paper (like the "Tiny" dataset with 1E6 384-dimensional points)[5]. This means that the findings in the Results section above may only apply to smaller datasets, and that for sufficiently large datasets, it is possible that my implementation of Yinyang K-means can actually prove more efficient than either the Standard or Elkan's K-means algorithms. Without more experiments, we cannot know at this point how the current implementation would perform for larger datasets.

The last remaining issue is that the use of parallelism in the current implementation is severely limited. As of right now, some naive approaches to parallelism have been exercised to attempt to improve performance to match prior implementations of other algorithms. However, these approaches are not particularly well-suited to the task, and instead are somewhat of an overly simplistic method of parallelizing some computations. In particular, the main issues are in the parallelization of the "cluster_center_reducer" and "cluster_center_reducer_redun" functions. The method I have employed to parallelize the computations done involves creating one thread for every cluster and assigning each thread the task of computing all the changes to each cluster based on the points which changed from one cluster to another in the last iteration of the algorithm. This approach does not take into account the fact that not all clusters will have the same number of points being added or subtracted each iteration, and as a result, the work is unevenly distributed between threads. A solution to this problem would require a total rewrite of these functions to more closely match the behavior of the map reduce operations performed in the original implementation provided by Dr. Shen.

8 Possible Solutions

The solution to the first remaining issue is to attempt to reduce the overhead associated with initializing the data structures needed for the Yinyang K-means algorithm, as well as speed up the implementation of Yinyang K-means globally through solving the other remaining issues. As of right now, my implementation of Yinyang K-means constructs a new "vertex_data" object for each point in the input dataset, which requires the copying of every input data point. This also wastes a great deal of memory because both the original data and a new copy of the original data needs to be stored. To solve this, a solution could be devised which does not require the copying of the input data, but rather would allow the direct referencing of data from the original input data structures. This would save the overhead of having to copy the original input data just to reuse it without any modifications.

The solution to the second remaining issue is to go back through every existing function in the current implementation and add checks whenever a point or center of a cluster is referenced to see if the point or center is sparse or dense, and use the data accordingly. The groundwork for this is laid already, with the "vertex_data" and "cluster" C++ structs already defined with the necessary field to store sparse data points. To refactor the code to support sparse data would likely be a simple process, as a result, and would just require branching in a few key points in the code to handle certain sensitive operations differently depending on whether input data was sparse or dense.

The solution to the third remaining issue is simply to test my implementation against more datasets, particularly those that are larger than the two I was able to test against thus far. Doing this should produce results that may not still exhibit the same distinct shift from k=16 to k=64 as is seen in Figures 2 and 3. It would require more experimentation and testing to see whether the size of the dataset can have an impact on the performance of Yinyang K-means over Standard or Elkan K-means in this regard.

The solution to the last remaining issue is to totally refactor the functions currently employing parallelism in the current implementation of Yinyang K-means. Most importantly, the functions which need refactoring are the "cluster_center_reducer" and "cluster_center_reducer_redun" functions mentioned earlier, that use a naive approach to parallelism to achieve minor speedup in their major functions. As described before, the method I have employed to parallelize the computations done involves creating one thread for every cluster and assigning each thread the task of computing all the changes to each cluster based on the points which changed from one cluster to another in the last iteration of the algorithm. Also noted earlier is the fact that this approach does not take into account the fact that not all clusters will have the same number of points being added or subtracted each iteration, and as a result, the work is unevenly distributed between threads. The solution, then, is to more evenly distribute the work between threads.

There are two possible ways to more evenly divide work between threads in the "cluster_center_reducer" and "cluster_center_reducer_redun" functions. The first way is to make use of some native Python (or Cython) map reduce libraries which could do the work of computing new cluster center changes without the need for manual implementation of parallelism in these functions. This would allow me to re-implement the same methods used in the original implementation from Dr. Shen, and thus provide the best speeds. However, this may not be feasible. The second way would be to attempt to divide the actual points whose cluster assignments changed between threads, starting one thread per data point whose cluster center changed (or one thread per n data points whose cluster center changed). This would guarantee that work is divided evenly between threads, which would allow for maximum speedup.

9 Conclusion

Yinyang K-means is a powerful algorithm improving on the existing K-means algorithm, and can provide significant improvements in the time it takes to compute cluster centers for a given dataset. When paired with multi-threading techniques, a simple implementation of Yinyang K-means can produce over 3-4 times speedup over either Standard or Elkan K-means. For my implementation in particular, this speedup grows with the number of clusters computed, which means that for large datasets and large numbers of clusters, my implementation of Yinyang K-means can provide significant improvements over other algorithms, without sacrificing cluster quality. Optimizing algorithms is a difficult task, and requires extensive knowledge of programming, multiprocessing concepts, and other important topics. As a result, it is highly desirable to find a way to programmatically optimize algorithms, rather than manually, so that the burden of optimization can be shifted away from the writer of the algorithm, requiring less human knowledge and investment in algorithm optimization to produce clean, efficient code.

References

[1] [n.d.]. 2.3. Clustering¶. https://scikit-learn.org/stable/modules/clustering.html

[2] [n.d.]. C-Extensions for Python. https://cython.org/

[3] [n.d.]. The C Resources Network. http://www.cplusplus.com/

[4] Blaise Barney. [n.d.]. https://computing.llnl.gov/tutorials/openMP/

[5] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Yinyang K-means: A Drop-in Replacement of the Classic K-means with Consistent Speedup. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, 579–587. http://dl.acm.org/citation.cfm?id=3045118.3045181

[6] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml