Parallel `stable k-means`*

Emin Arakelian† Fadi Kfoury‡ Jason Poulos§

Abstract

We explore parallelization of the ***`stable k-means`*** *algorithm, proposed by Ben-Hur et al. [2001] to determine the optimal number of clusters in unlabeled data. We focus on using Python's native multiprocessing module, which accommodates multithreading and multiprocessing, to parallelize the serial algorithm. In experiments on simulated data, we demonstrate a 1.3 times speed-up for the multithread algorithm and a 3.3 times speed-up for the multiprocess algorithm. A profiling of parallel* ***`stable k-means`*** *shows that significant time is spent in waiting and acquiring the lock to write to the array. We discuss attempts at optimizing the parallel version, challenges faced during parallelization, and suggest opportunities for speed-up and scalability.*

1. Introduction

Unsupervised learning is a branch of machine learning that infers patterns from data that has no labels. k-means is a popular unsupervised learning algorithm for finding clusters and cluster centers in data. The goal is to choose k cluster centers to minimize the total squared distance between each data point and its closest center. In other words, its objective is to find

$$\arg\min_{s} \sum_{i=1}^{k} \sum_{x \in S_i} \| x - \mu_i \|^2 .$$

Given k initial centers chosen uniformly at random from the data points, k-means alternates between two steps until convergence: (*assignment step*) each point is assigned to the nearest cluster center and; (*update step*) each center is recomputed as the center of mass of all points assigned to it.

There are are several methods to determine the optimal number of clusters in unlabeled data. One way is to use different clustering metics such as the silhouette coefficient [1]. Another approach — which is the subject of this project — is stability [4][5]. As we will later discuss, assessing stability requires multiple runs of k-means, which will make it computationally very expensive. Algorithms that speed up the selection of k initial centers have been proposed, such as `k-means++`, which chooses only the first cluster center uniformly at random, and selects subsequent centers from the data points, weighed by a probability proportional to its contribution to the overall error [2]. Bahmani et al. [3] propose a method of parallelizing the initialization that reduces the number of passes by sampling $O(k)$ points and for $O(\log n)$ rounds, instead of sampling a single point in each pass.

Given that Python is one of the most popular coding languages for doing data science, we explore the options of parallelization of the stability-based method, `stable k-means`, in Python. We will take a look at our options in using parallelization in Python and analyze them briefly. As we will discover Python is not the ideal platform for parallel programming, so why bother? The reason is that Python is a very high level language with many good data types, databases, text parsing and many other capabilities that allow for complicated algorithms. One of the simplest functions we have in our `stable k-means` algorithm is generating an array of random numbers. In the snippet in Section A we see a comparison of this same task in C++ and Python. Thus it is a waste not to use Python's Parallel capabilities for speed ups, solely for the reason that they are not ideal. While we will mainly focus on Python in this project, it is entirely possible to use `Ctypes` in Python and get even higher performances using C extensions.

We describe the serial version of `stable k-means` in

†`emin@berkeley.edu`
‡`fadi.kfoury@berkeley.edu`
§`poulos@berkeley.edu`

*The code used for this project is available on Github: `https://github.com/plumSemPy/parallel_kmeans`.

Section 2 and its parallel implementation using Python in Section 3 . We then describe the implementation on simulated data in Section 4. We profile the parallel algorithm in Section 5 and discuss attempts at optimization in Section 6. We discuss challenges and future directions in Section 7 and draw conclusions in Section 8.

2. Serial `stable k-means`

Ben-Hur et al. [4] propose an algorithm for any clustering mechanism that will assess the stability of the points and thus let us infer the optimal number of clusters. The algorithm begins with subsampling the data twice with a ration more than half. It is believed that the general structure of the dataset is captured when subsampling with a ration bigger than a half. Then, for a given number of clusters, the two subsamples are clustered. The points in the intersection of the two subsamples are analyzed to see how many of them are clustered together in the same cluster in both subsamples. Then, depending on the number of the points clustered together in the intersection of both of the subsamples, a similarity score is calculated. We will use the correlation score for our application.

Let labeling $\mathcal{L}$ be a partition of X into k subsets $S_1, S_2, ..., S_k$. In that case the correlation score between the clustering of intersection of two subsamples is given by:

$\text{Corr}(\mathcal{L}_1, \mathcal{L}_2) = \frac{\langle \mathcal{L}_1, \mathcal{L}_2 \rangle}{\sqrt{\langle \mathcal{L}_1, \mathcal{L}_1 \rangle \langle \mathcal{L}_2, \mathcal{L}_2 \rangle}}$. The algorithm for `stable k-means` is displayed in Algorithm 1.

Algorithm 1

Input: X {a dataset}, k_{max} {maximum number of clusters}, num_subsamples {number of subsamples}
Output: $S(i,k)$ {list of similarities for each k and each pair of sub-samples}
Require: A clustering algorithm: cluster(X,k) ; a similarity measure between labels: $s(\mathcal{L}_1, \mathcal{L}_2)$
1: $f = 0.7$
2: for $k = 2$ to k_{max} **do**
3: for $i = 1$ to $num_subsamples$ **do**
4: sub_1 =subsamp(X, f) {a sub-sample with a fraction of f the data}
5: sub_2 =subsamp(X, f)
6: $\mathcal{L}_1$ =cluster(sub_1, k)
7: $\mathcal{L}_2$ =cluster(sub_2, k)
8: Intersect $= sub_1 \cap sub_2$
9: $S(i,k) = s(\mathcal{L}_1(Intersect), \mathcal{L}_2(Intersect))$ {Compute the similarity on the points common to both subsamples}
10: **end for**
11: **end for**

Given this algorithm, for each k, a distribution of scores will be attained. The distribution with the highest score and least variance is the best; distributions that are unimodal are also highly preferred. These criteria means that the consistently intersection points were clustered together and thus that certain formation of clusters was stable.

3. Parallelization

In this project we will be focusing on using Python's native multiprocessing module, which accommodates multithreading and multiprocessing. In the following sections we will take a closer look on how each of the function.

3.1. Threading and the GIL

Python utilizes a multithreading module letting the user use multiple threads. Python threads are real system threads, i.e. Posix threads or Windows threads and are fully operated by the host operating system. The caveat however is that for CPU-bound functions, the threaded version will perform worse than the serial version. The reason behind that is that parallel execution is forbidden in Python and there is a GIL, Global Interpreter Lock, that ensures only one thread is ran at a time in the interpreter which simplifies many of the low level tasks like memory management and call out to C extensions. With the existence of GIL we get a cooperative multitasking as shown in Figure 1. When a thread is running it is holding the GIL and the GIL is released upon I/O commands such as read, write, send, receive etc. Thus the merit of multithreading will be apparent when the functions are I/O bound.

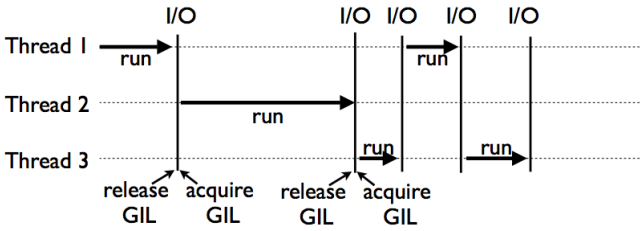

Figure 1. Cooperative multitasking with GIL

CPU-bound threads that never perform I/O are handled differently where there are special check for every 100 "ticks", see Figure 2. These ticks should not be confused with the clock ticks. These ticks are roughly equivalent to one interpreter instruction. The check is as follows, the current thread will reset the tick counter, reset the signal handler if it is the main thread, release the GIL and reacquire it.

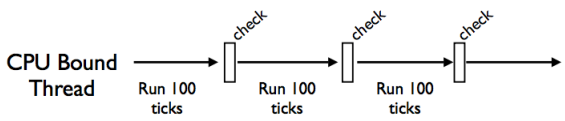

Figure 2. No I/O function checks

In thread switching on a single core two cases can happen. Assume we have two threads, with the first thread running. If at some point the first thread has an I/O instruction, it will release the GIL and the second thread will acquire it. However, if the first thread has no I/O instructions, then the GIL release will happen on the check. In which case the lock is free and two threads are ready to run. Here the priority will be handled by the OS, and the next thread may or may not be thread 2, however experimental result has shown that this thread switching happens rarely, and there would be thousands of checks before thread two will run. A more interesting problem arises when multiple cores have runnable threads. Here the threads get scheduled simultaneously on multiple cores and thus they will battle over the GIL. This happens because when thread one releases the GIL and signals universally, thread two takes some time to wake up to acquire the lock, by which time thread one has acquired the lock and thread two, failing at it's endeavors decides to sleep again. An illustration is provided in Figure 3.

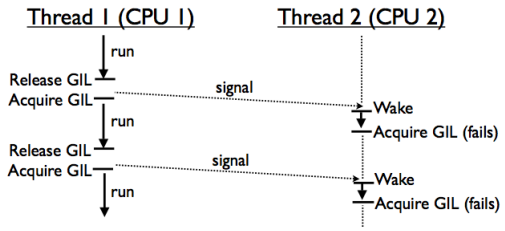

Figure 3. GIL wars on multicored threads

To remedy this problem, in Python 3 is a new GIL implementation that allows thread two to signal a timeout in which case thread one will drop the lock after its current instruction and thread two can take over. This seems to address the problem a little, but still there are starvation scenarios where the most deserving threads do not get to run first due to scheduling. A suggested work around for this is to separate CPU-bound, low priority threads and I/O, high priority threads.

In conclusion threading in Python has many caveats and edge cases and must be dealt with cautiously. However it does provide improvements that will be wasteful not to utilize.

3.2. Multiprocessing

A good alternative to multithreading is multiprocessing which came about around Python 2.6. The syntax is exactly the same as multithreading but here python uses processes instead of threads and thus eliminating shared anything. Here we will have multiple instances of the interpreter running each having their own GIL and the way they communicate is message passing. The messages are being passed using "pickle"-ing. The pickle module in Python can serialize and deserialize python objects into byte-streams. Since we are using message passing here, we can deploy our code on multiple machines as well.

3.3. Other Python Parallel Libraries

- **PyCUDA :** A Python wrapper for the CUDA API. `https://documen.tician.de/pycuda/`
- **PyMPI :** An underdevelopment message passing interface for Python being developed by Lawrence Livermore National Laboratory. `http://pympi.sourceforge.net/`
- **Asyncio :** Asynchronous I/O, event loop, coroutines and tasks. `https://docs.python.org/3/library/asyncio.html`
- **Twisted :** An event driven networking engine for Python. `https://twistedmatrix.com/trac/`

The latter two are trying to convert all the I/O handling into event, and lean towards event driven programming and move away from threads and processes.

4. Experiments

We run our algorithm and compare its results on a toy dataset of 500 samples and 2 features. The dataset is plotted in Figure 4. The dataset is obviously split into 5 clusters at most. Running the algorithm on this dataset will yield the following similarity scoring distribution (Figure 5).

As seen the best results are given for 2 to 5 clusters.* It will be evident why we are getting such high scores on those cluster formation, in other words why are those formations stable.

We will now assess the runtime of our serial and parallel implementations. In the parallel implementation we have parallelized the inner loop. In that each process runs a subsampling, clustering and a similarity scoring. They all then write their outputs to

*The plots for 2 to 5 clustering formation is given in Section B.

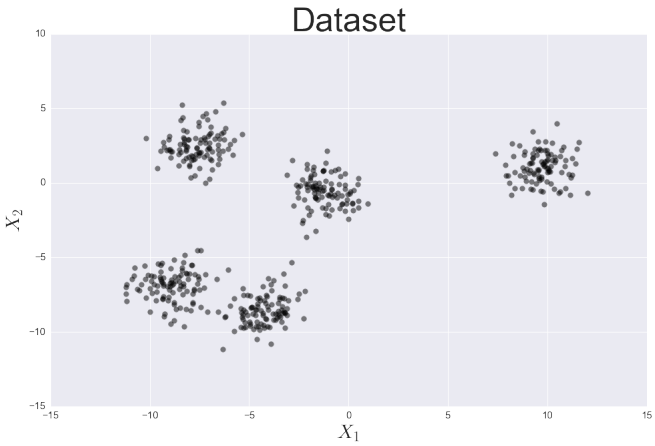

Figure 4. Dataset that will be used to assess `stable k-means`

one shared array. Thus each processes or thread will run number_of_samples/(number_of_processes or number_of_threads) times. Figure 6 shows the baseline serial run time. The time complexity of k-means is $O(n^{dk+1}) \log n$. Since we are doing a loop over $k = 2$ to$k = K$ then our complexity will go to $O(n^{dk_1} + n^{dk_2} + ... + n^{dK})n \log n$. Note that the fact that we run in num_subsamples does not affect the time complexity.

Figure 7 shows the strong and weak scaling of the multithread and multiprocess algorithms. As evident the optimal number of threads and processes are attained at 3 threads and 3 processes. After that the code starts getting slow most likely because the machine that this code was run on had 4 cores. As expected, since the code is I/O bound, threading improved the run time, but not as much as multiprocessing, where concurrency of the threading is replaced by parallelization. Sadly since we do net yet have access to more cores we aren't able to tune the size up to attain a good weak scaling. It looks as though the OS is scheduling one process or thread per core and hence the speedups and speed downs.

Figure 8 shows a comparison of the run time on the serial, concurrent (multithread) and parallel (multiprocess) code. The sample size varies between 100 to 500 samples and 3 threads and processes are taken to do the comparison as they were the optimal on the previous part. From the serial implementation and 500 samples we have a 1.3 times speed up for the multithread algorithm and a 3.3 times speed up for the multiprocess algorithm.

5. Profiling

We profiled our algorithm to see where the time was wasted. The result is shown in Figure 9. As seen most of the time is spent on either acquiring the lock or waiting on the lock. The full profiling output can be reviewed in Appendix Section C.

6. Optimization

To counter the effect of the shared array that we saw while profiling our code, we did the following. In the implementation mentioned above, all the processes and threads write their outputs after every operation to a global shared array. That was because every process was running the inner most loop of a three-nested-loop system. To remedy this we rearranged the loops such that each process will run $\frac{1}{P}$ of the workload in *all* the processes and threads. In multithreading this resulted in a worse performance. However in multiprocessing this improved the results a bit. This can be seen in Figure 10, where "Sep" means the optimized algorithm.

7. Challenges and Future Directions

One of the first challenges faced was the lack of proper documentation on many of the Python's parallel libraries. We were initially decided to include GPUs by using CUDA as a comparison here. But due to the huge difficulty of installing CUDA on the only Nvidia machine we had, and given that we want to make this package available for public use, we revised and decided to hold off on CUDA for now until a much more user friendly installation pipeline is provisioned. The pyMPI package seems promising but it is still under development.

As mentioned earlier the multiprocessing module uses pickles to send messages. Our implementation parallelizes a method in a class in Python and unfortunately pickle cannot pickle a class instance method. For that we had to use a fork of the multiprocessing called `pathos.multiprocessing` which uses "dill" instead of "pickle".

There might be some speedups, if we collect the results from each process separately and then put them in the global array, instead of asking them to write their result at the end of each k^{th} iteration. A profiling of the method shows that significant time is spent in waiting and acquiring the lock to write to the array.

In terms of scalability of the memory, one improvement might be possible if instead of giving each process an entire copy of the data, we put the jobs in a queue and ask each process to take its job from there, or alternatively we can send them their jobs. But that might

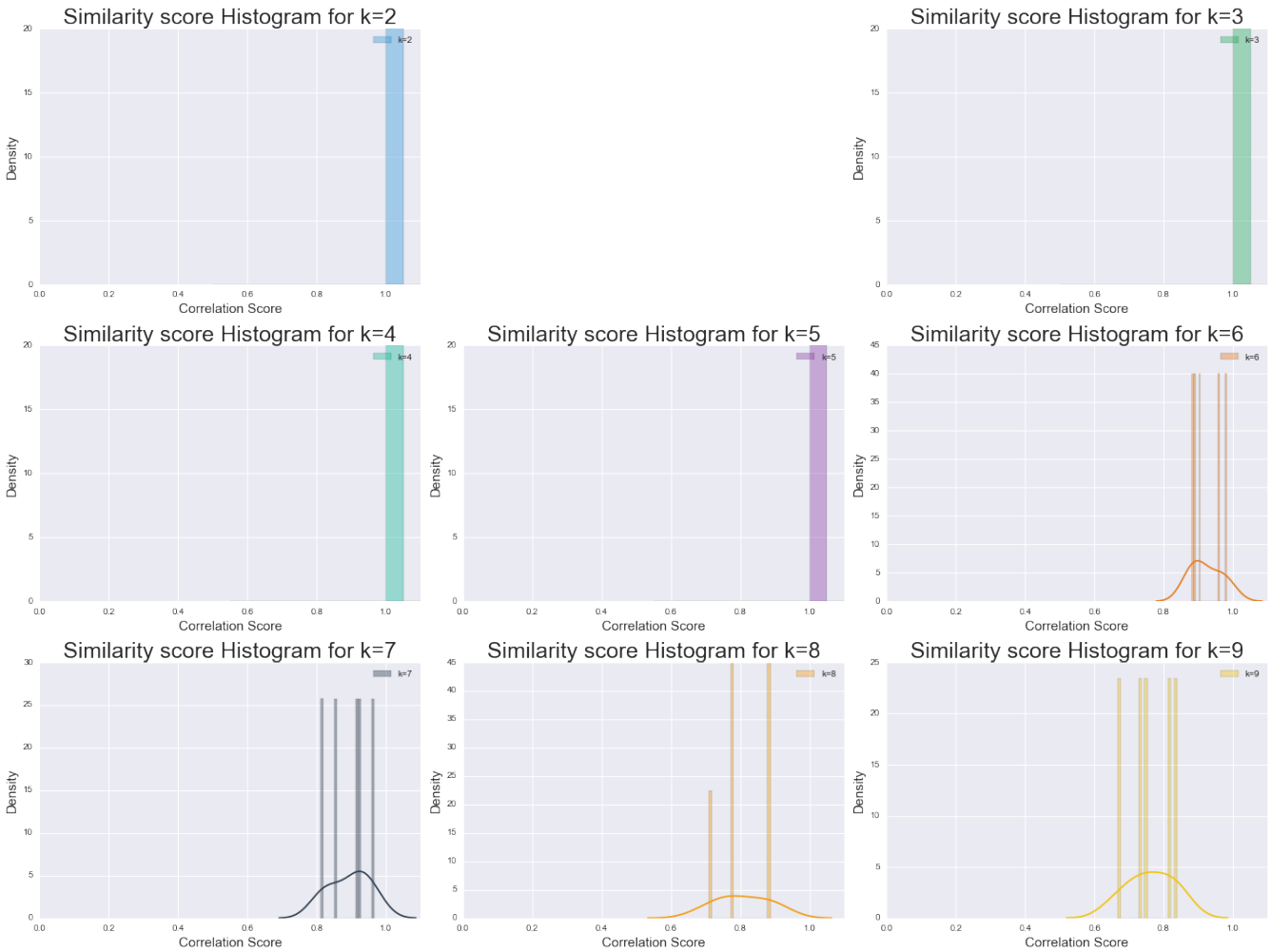

Figure 5. Histograms of Similarity scores for different values of k.

result in increased waiting times on the lock again to get the job from the queue.

8. Conclusion

Python is a hugely popular platform for performing very high level complex algorithms. The merit being the fact that the programmer needs to spend less time worrying about optimization and tweaking the little knobs and spend more time on creating something more complex. Python is infamous for its GIL and threading problems. The creators of Python wanted to simply their low level managements by avoiding shared anything. That should not however prevent us from trying to get the best out of the language, especially with the advent of the new multiprocessing library. PyCUDA and pyMPI show a lot of promise, however CUDA really needs to revamp its use pipeline. We attempted parallelizing a very useful algorithm that returns the optimal number of clusters in a given dataset. The results show that we can attain very decent speedups using multiprocessing on an average machine.

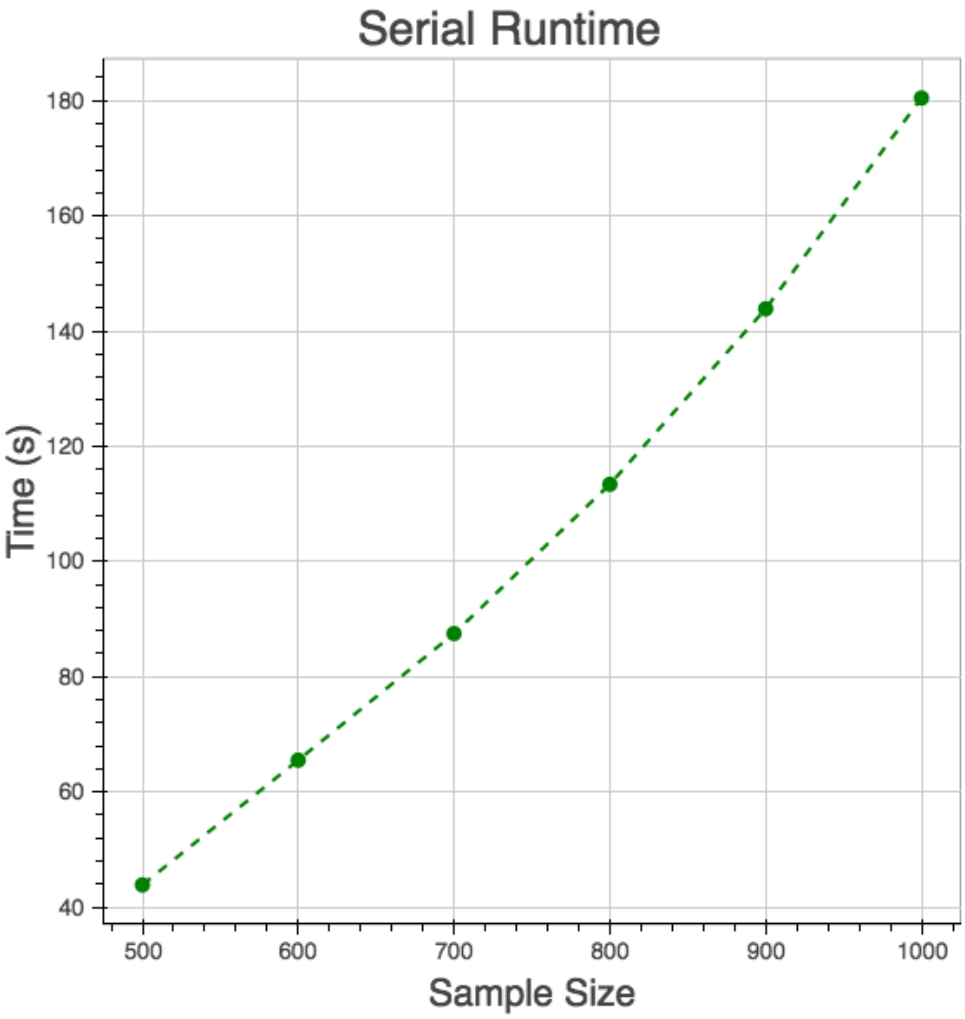

Figure 6. Serial Run Time

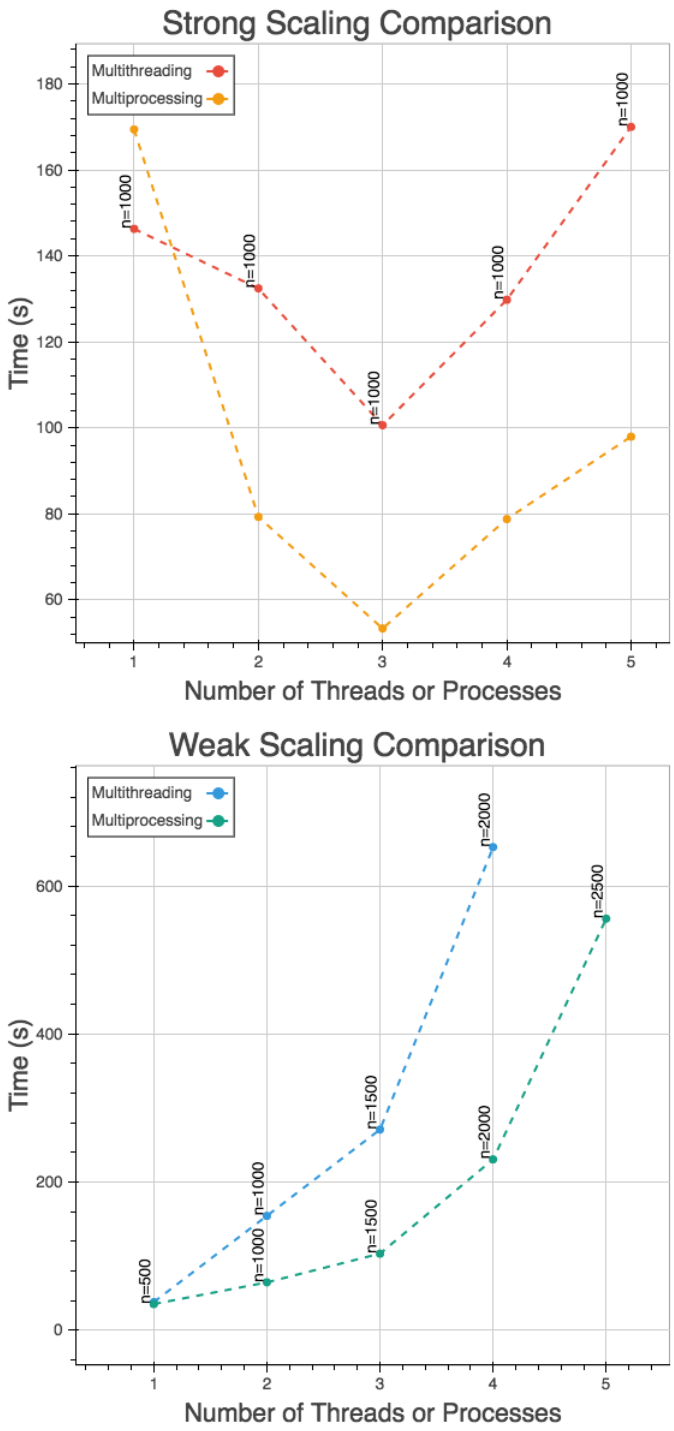

Figure 7. Strong and Weak Scaling Comparison for Multithreading and Multiprocessing

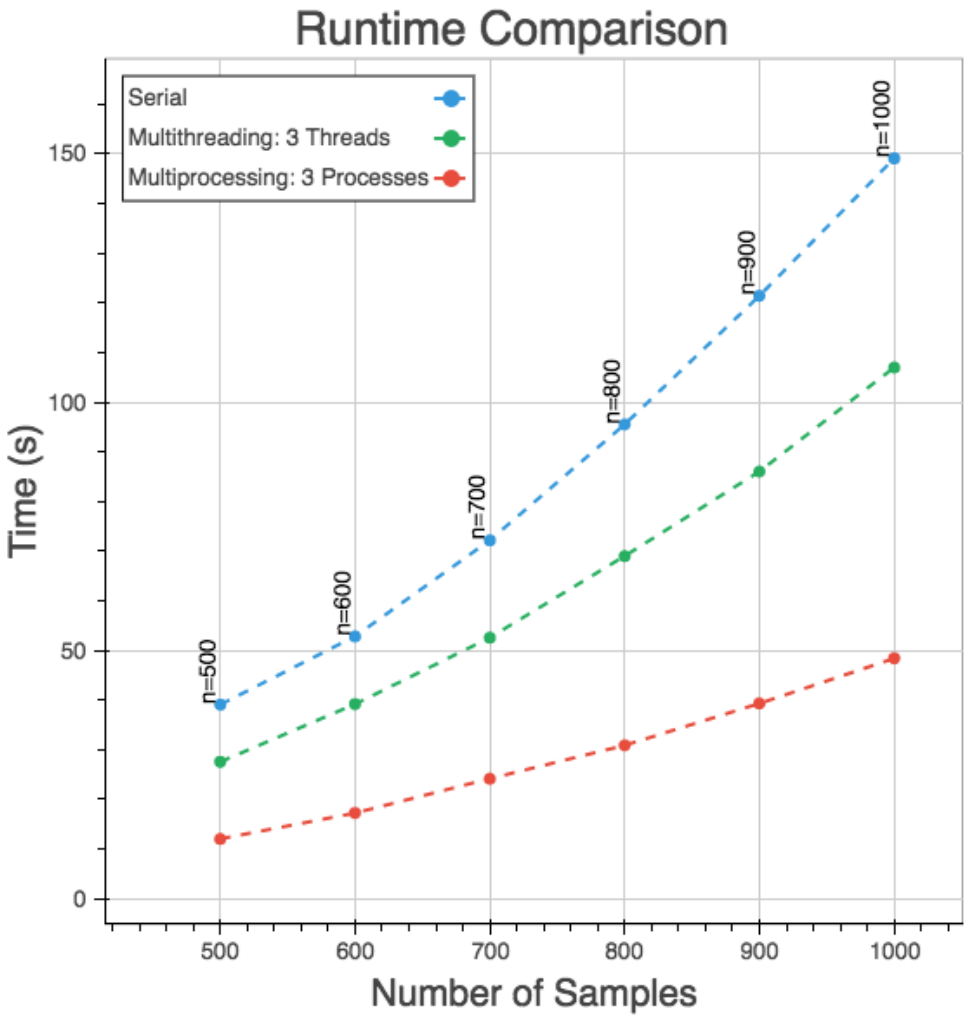

Figure 8. Speed up Comparison

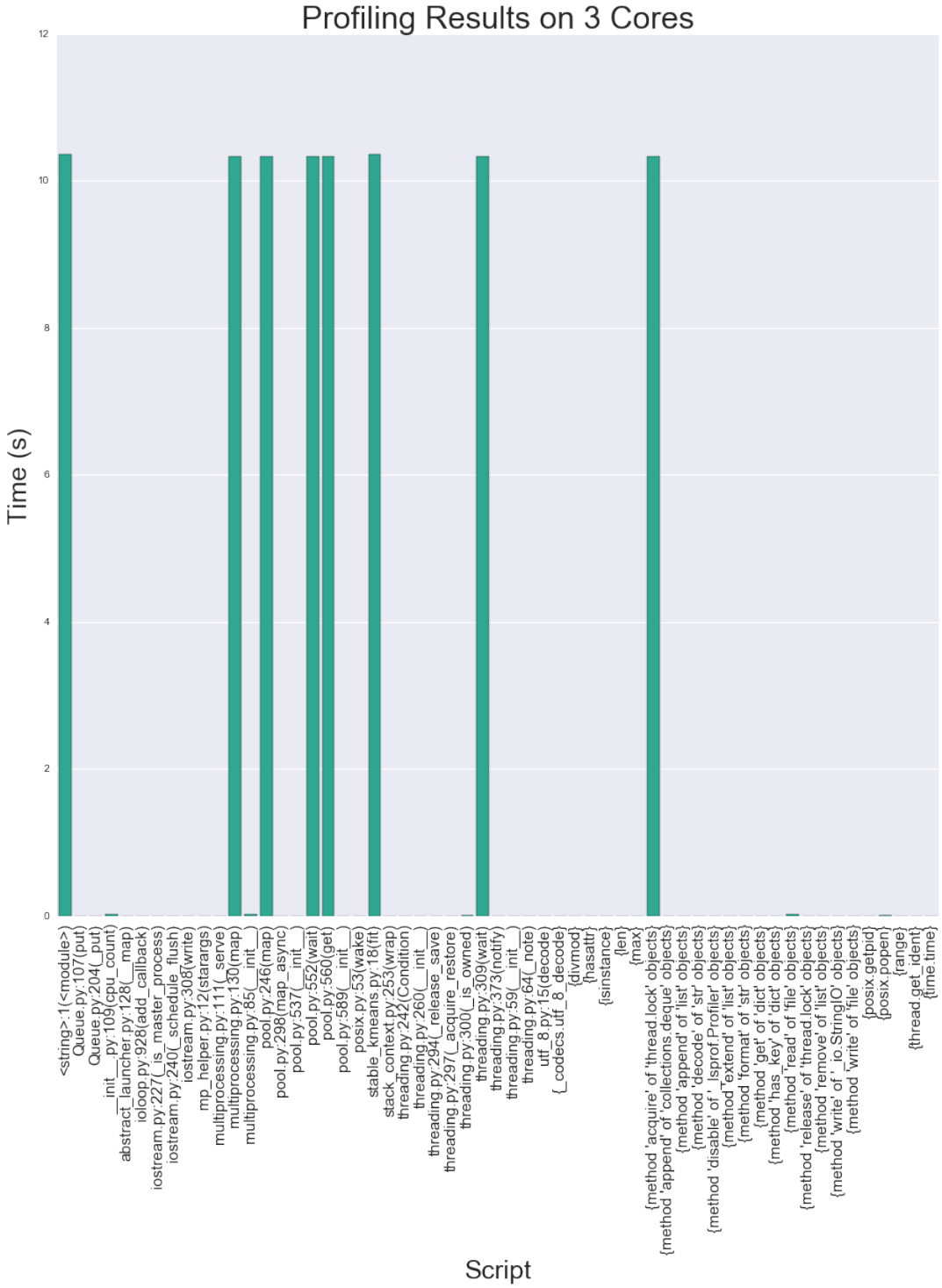

Figure 9. Profiling of the code

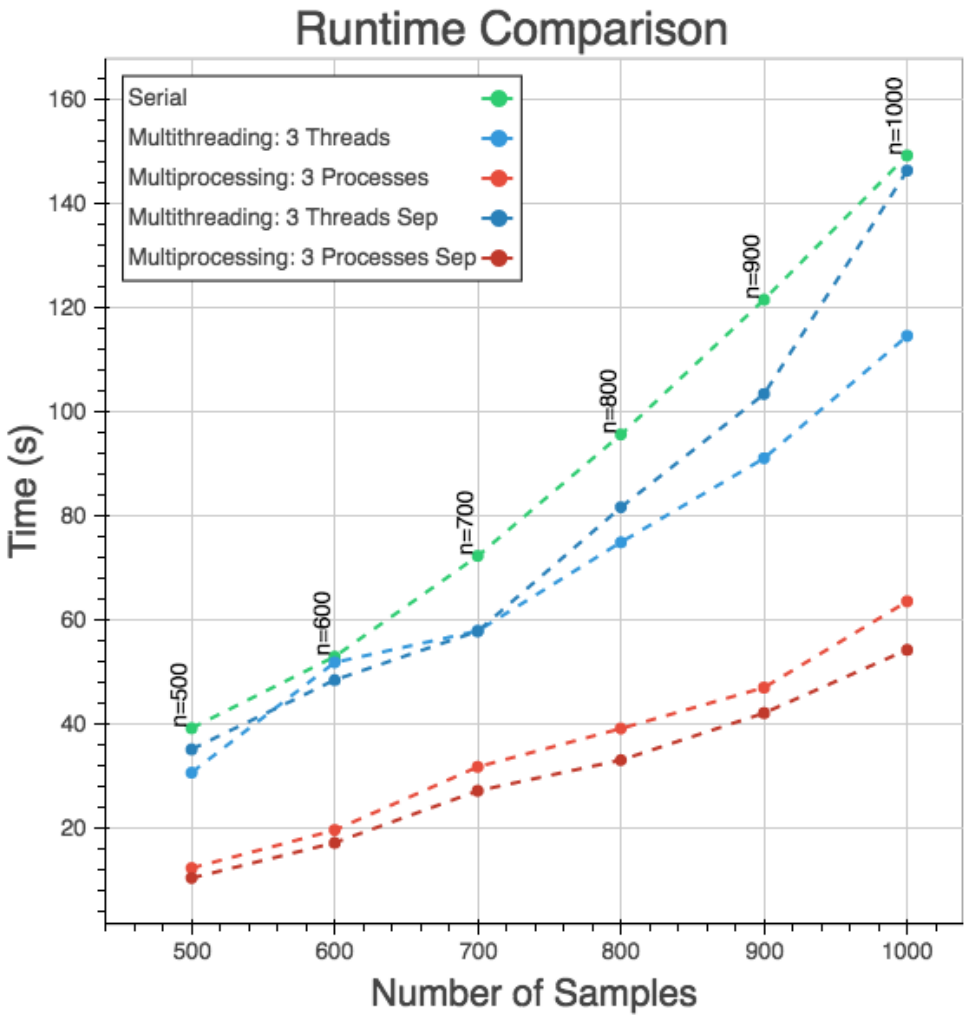

Figure 10. Improvement runtime comparisons

References

[1] S. Aranganayagi and K. Thangavel. Clustering categorical data using silhouette coefficient as a relocating measure. In *Conference on Computational Intelligence and Multimedia Applications, 2007. International Conference on*, volume 2, pages 13–17. IEEE, 2007.

[2] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[3] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii. Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7):622–633, 2012.

[4] A. Ben-Hur, A. Elisseeff, and I. Guyon. A stability based method for discovering structure in clustered data. In *Pacific symposium on biocomputing*, volume 7, pages 6–17, 2001.

[5] S. P. Smith and R. Dubes. Stability of a hierarchical clustering. *Pattern Recognition*, 12(3):177–187, 1980.

[6] B. Vaux and S. Golder. The harvard dialect survey. *Cambridge, MA: Harvard University Linguistics Department*, 2003.

Appendices

A. Python vs. C++ comparison

```
C++
#include <algorithm>
#include <cstdlib>
#include <iostream>scale=0.35

int main() {
    int a[8] = {0, 1, 2, 3, 4, 5, 6, 7};
std::srand(unsigned(std::time(0)));
    std::random_shuffle(a, a + 8);
    for (int i=0; i<8; i++)
std::cout << a[i] << " "; std::cout << std::endl; }

Python
import random
import numpy as np

random_array = np.array(random.sample(range(8),8))
```

B. Cluster formation

Figure 11, 12 , 13 , 14 , 15 show why 2 to 5 clusters are stable for our toy dataset and 6 clusters is not. We can see in the figures that there is a clear grouping between 2, 3, 4 and 5 clusters. Thus certain points always cluster together. However for 6 clusters, one of the obvious clusters needs to split into 2, and that is an arbitrary choice and hence the non-unimodal histograms.

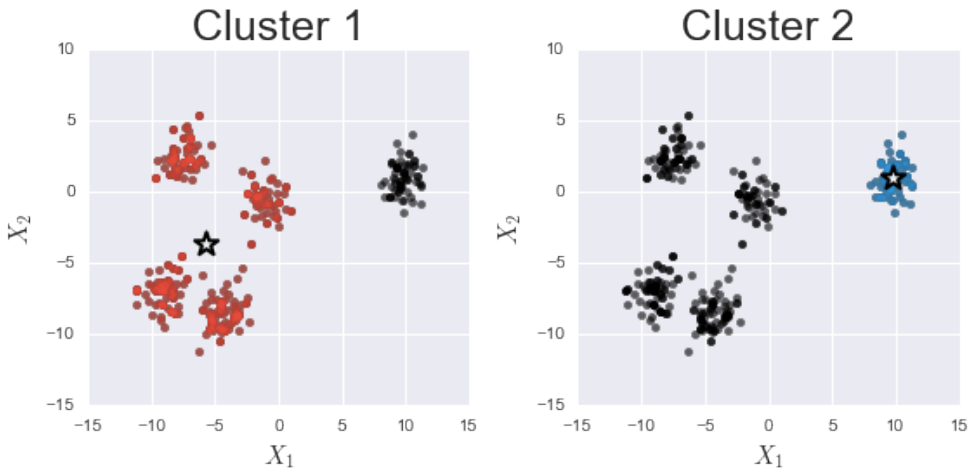

Figure 11. 2 Clusters Formation

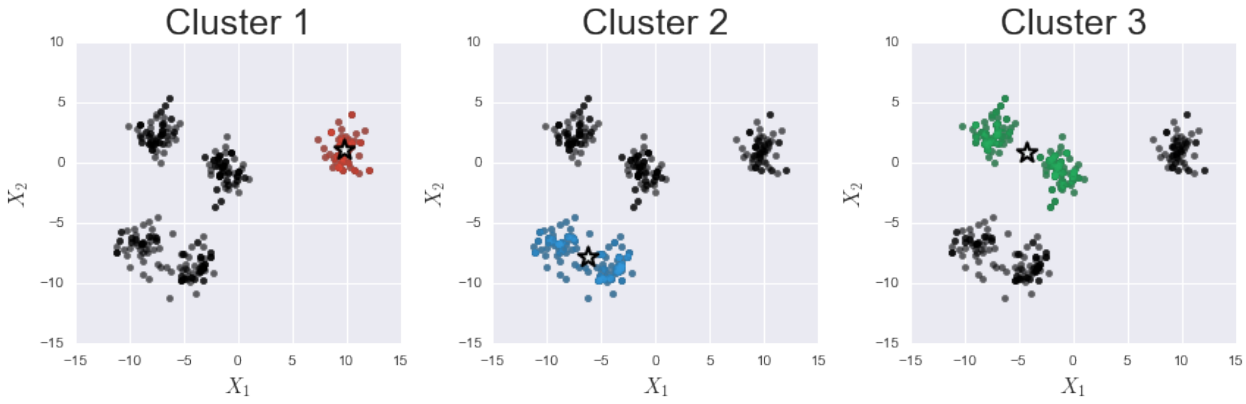

Figure 12. 3 Clusters Formation

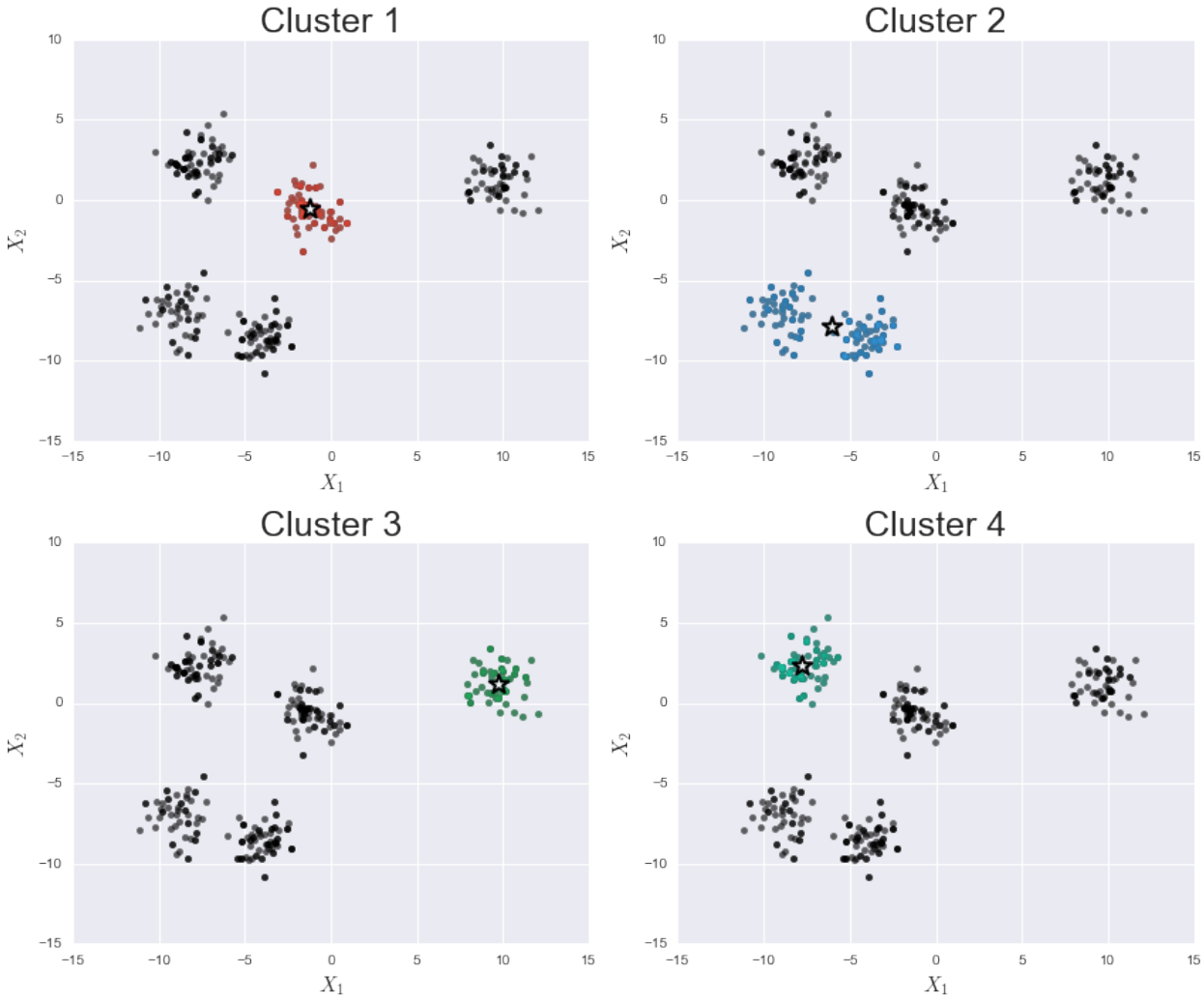

Figure 13. 4 Cluster Formation

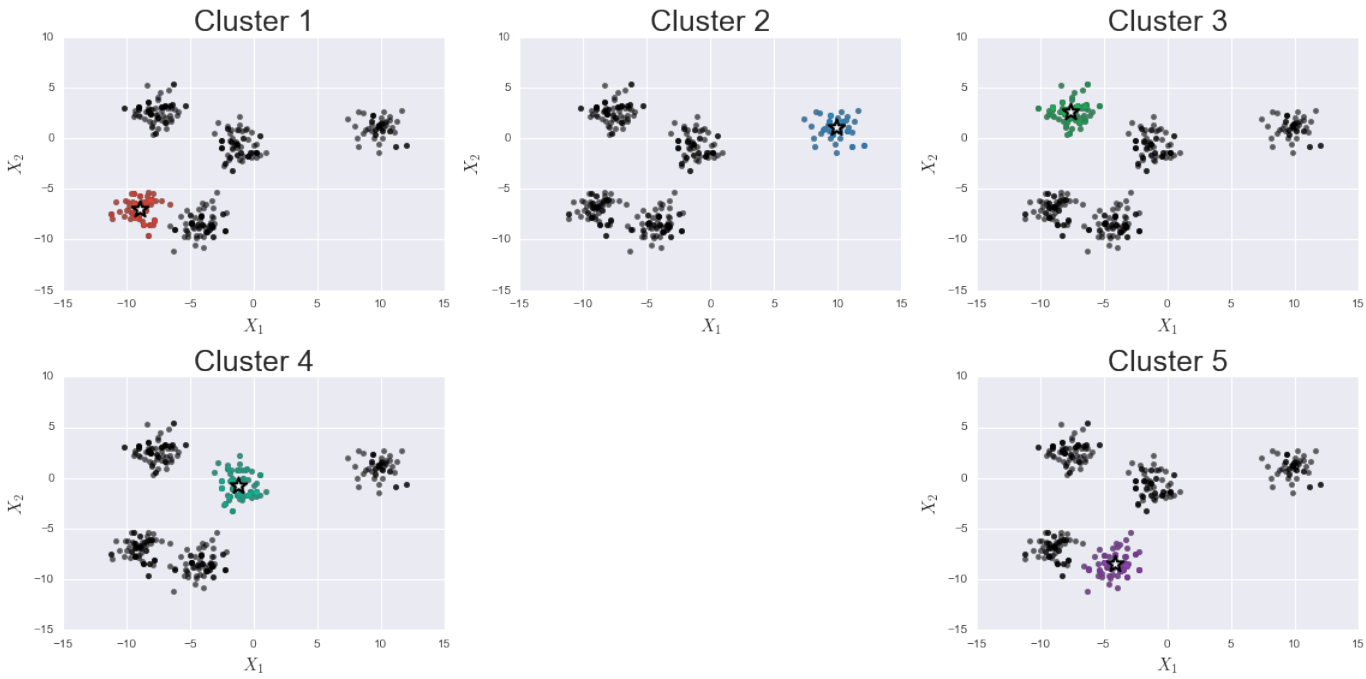

Figure 14. 5 Cluster Formation

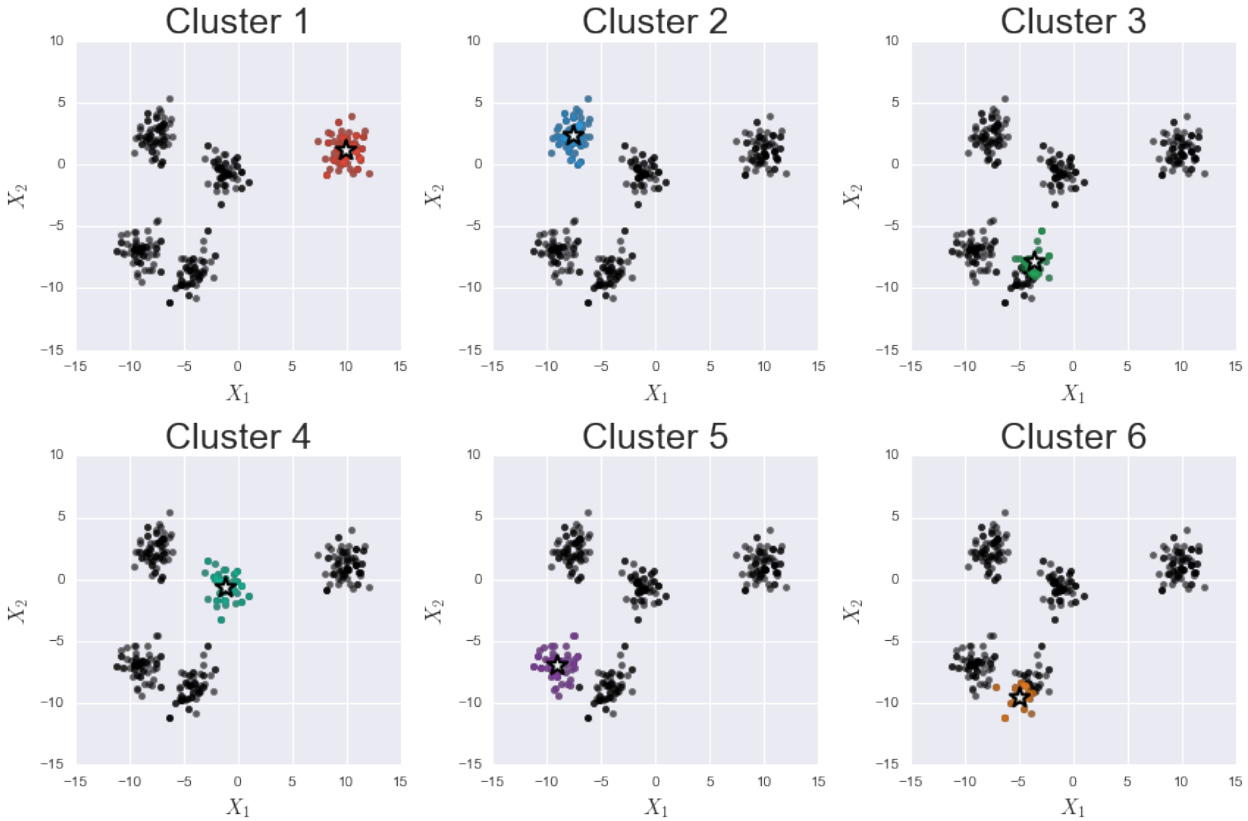

Figure 15. 6 Cluster Formation

C. Full Profiling Output

```
ncalls   tottime   percall   cumtime   percall   filename:lineno(function)
1    0.000    0.000   10.353   10.353   <string>:1(<module>)
6    0.000    0.000    0.000    0.000   Queue.py:107(put)
6    0.000    0.000    0.000    0.000   Queue.py:204(_put)
1    0.000    0.000    0.020    0.020   __init__.py:109(cpu_count)
6    0.000    0.000    0.000    0.000   abstract_launcher.py:128(__map)
6    0.000    0.000    0.000    0.000   ioloop.py:928(add_callback)
12    0.000    0.000    0.000    0.000   iostream.py:227(_is_master_process)
12    0.000    0.000    0.000    0.000   iostream.py:240(_schedule_flush)
12    0.000    0.000    0.000    0.000   iostream.py:308(write)
6    0.000    0.000    0.000    0.000   mp_helper.py:12(starargs)
7    0.000    0.000    0.000    0.000   multiprocessing.py:111(_serve)
6    0.000    0.000   10.332    1.722   multiprocessing.py:130(map)
1    0.000    0.000    0.020    0.020   multiprocessing.py:85(__init__)
6    0.000    0.000   10.332    1.722   pool.py:246(map)
6    0.000    0.000    0.000    0.000   pool.py:298(map_async)
6    0.000    0.000    0.000    0.000   pool.py:537(__init__)
6    0.000    0.000   10.332    1.722   pool.py:552(wait)
6    0.000    0.000   10.332    1.722   pool.py:560(get)
6    0.000    0.000    0.000    0.000   pool.py:589(__init__)
6    0.000    0.000    0.000    0.000   posix.py:53(wake)
1    0.000    0.000   10.353   10.353   stable_kmeans.py:18(fit)
6    0.000    0.000    0.000    0.000   stack_context.py:253(wrap)
6    0.000    0.000    0.000    0.000   threading.py:242(Condition)
6    0.000    0.000    0.000    0.000   threading.py:260(__init__)
6    0.000    0.000    0.000    0.000   threading.py:294(_release_save)
6    0.000    0.000    0.000    0.000   threading.py:297(_acquire_restore)
12    0.000    0.000    0.010    0.001   threading.py:300(_is_owned)
6    0.000    0.000   10.332    1.722   threading.py:309(wait)
6    0.000    0.000    0.000    0.000   threading.py:373(notify)
6    0.000    0.000    0.000    0.000   threading.py:59(__init__)
12    0.000    0.000    0.000    0.000   threading.py:64(_note)
12    0.000    0.000    0.000    0.000   utf_8.py:15(decode)
12    0.000    0.000    0.000    0.000   {_codecs.utf_8_decode}
6    0.000    0.000    0.000    0.000   {divmod}
12    0.000    0.000    0.000    0.000   {hasattr}
12    0.000    0.000    0.000    0.000   {isinstance}
25    0.000    0.000    0.000    0.000   {len}
6    0.000    0.000    0.000    0.000   {max}
42   10.332    0.246   10.332    0.246   {method 'acquire' of 'thread.lock' objects}
6    0.000    0.000    0.000    0.000   {method 'append' of 'collections.deque' objects}
12    0.000    0.000    0.000    0.000   {method 'append' of 'list' objects}
12    0.000    0.000    0.000    0.000   {method 'decode' of 'str' objects}
1    0.000    0.000    0.000    0.000   {method 'disable' of '_lsprof.Profiler' objects}
6    0.000    0.000    0.000    0.000   {method 'extend' of 'list' objects}
6    0.000    0.000    0.000    0.000   {method 'format' of 'str' objects}
8    0.000    0.000    0.000    0.000   {method 'get' of 'dict' objects}
2    0.000    0.000    0.000    0.000   {method 'has_key' of 'dict' objects}
1    0.017    0.017    0.017    0.017   {method 'read' of 'file' objects}
24    0.000    0.000    0.000    0.000   {method 'release' of 'thread.lock' objects}
6    0.000    0.000    0.000    0.000   {method 'remove' of 'list' objects}
12    0.000    0.000    0.000    0.000   {method 'write' of '_io.StringIO' objects}
```

```
6    0.000    0.000    0.000    0.000   {method 'write' of 'file' objects}
12    0.000    0.000    0.000    0.000   {posix.getpid}
1    0.003    0.003    0.003    0.003   {posix.popen}
13    0.000    0.000    0.000    0.000   {range}
12    0.000    0.000    0.000    0.000  {thread.allocate_lock}
6    0.000    0.000    0.000    0.000   {thread.get_ident}
12    0.000    0.000    0.000    0.000   {time.time}
```