UNIVERSIDAD POLITÉCNICA DE YUCATÁN

BIS Universities

HIGH-PERFORMANCE COMPUTING

MARCH 24, 2024

EDUARDO MENDOZA VARGAS
FÁTIMA MIRANDA PESTAÑA
CYNTHIA VIDAL OVANDO

Implementing and Comparing K Means Clustering Algorithms

Introduction

In this project, we aim to implement three versions of the K Means algorithm: Pure Python, Numpy Arrays, and Cython. Each version offers a different approach to optimizing the algorithm for performance. By employing profiling techniques, we will measure and analyze the execution time, memory usage, and other relevant metrics of each implementation. Additionally, we will explore the scalability of these implementations concerning the size and dimensionality of the dataset.

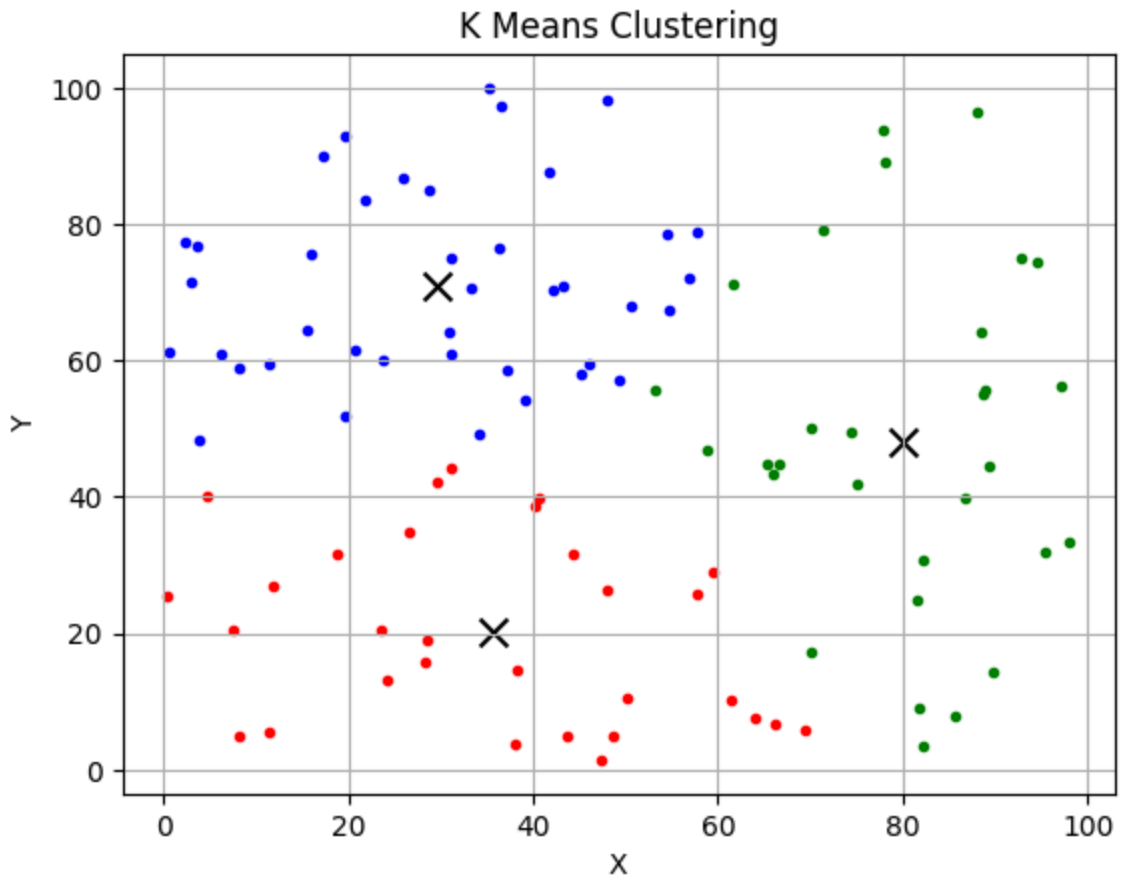
The K Means clustering algorithm is a fundamental unsupervised learning technique used for partitioning data points into distinct clusters based on similarity. It is widely employed in various fields such as image segmentation, customer segmentation, and anomaly detection. However, as datasets grow larger and more complex, the computational demands of the algorithm increase, needing efficient implementations.

Implementation Details

We used Google Colaboratory to implement our code and observe results. We also created a GitHub repository titled “K-MEANS_IMPLEMENTATIONS,” housing the source code named “Implementing_and_Comparing_K_Means_Clustering_Algorithms.ipynb.” Our focus within this notebook revolves around implementing and comparing various K-Means clustering algorithms.

The project has a variety of files, including datasets, Python scripts, and Cython files. This describes our attempt to use Cython to optimize K-Means clustering in order to improve performance.

We started by implementing the basic K Means algorithm using only Python standard library functions and made a plot to see how it behaved



Afterwards we did the same but now using NumPy Arrays by importing the library and using it in the source code and lastly we tested doing it with Cython.

Experimental Setup

For the experimental setup, we went for a certain number of clusters (K), some important Dimensionality of data points and Size of the dataset.

The number of clusters is defined by the *k* variable. To modify this value, we simply change the value of *k* in the line corresponding to the assignment of the number of clusters. For example, if we want to change the number of clusters to 5, we simply modify the line *k* = 3 to *k* = *5*

As dimensionality increases, the feature space becomes more sparse and the notion of distance between data points can become less meaningful, that's why in this context, we chose to use 2 dimensions and we generate datasets with varying dimensions and sizes for testing our K Means implementations using Numpy due to its efficiency in handling large arrays and matrices.

Profiling

We tested the environment with specific hardware to see how the parameters affect the performance. We can check it by using the "timeit" library.

The hardware used was the following:

- CPU: AMD Ryzen 5 3450U
- GPU : Radeon Vega Mobile Gfx
- RAM: 12.0 GB (9.89 GB usable)

This was tested using Windows 11 Home Single Language and our results were as follows:

```
PS C:\Users\Cynthia\Documents\HPC> python kmeans_pure_python.py
         8612 function calls in 0.006 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1500    0.002    0.000    0.004    0.000 C:\Users\Cynthia\Documents\HPC\kmeans_pure_python.py:6(euclidean_distance)
     1500    0.002    0.000    0.002    0.000 C:\Users\Cynthia\Documents\HPC\kmeans_pure_python.py:8(<listcomp>)
     1500    0.001    0.000    0.005    0.000 C:\Users\Cynthia\Documents\HPC\kmeans_pure_python.py:24(<lambda>)
      500    0.001    0.000    0.006    0.000 {built-in method builtins.min}
        1    0.001    0.001    0.006    0.006 C:\Users\Cynthia\Documents\HPC\kmeans_pure_python.py:16(kmeans)
     1530    0.000    0.000    0.000    0.000 {built-in method builtins.sum}
     1500    0.000    0.000    0.000    0.000 {built-in method math.sqrt}
      500    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        5    0.000    0.000    0.000    0.000 C:\Users\Cynthia\Documents\HPC\kmeans_pure_python.py:27(<listcomp>)
       15    0.000    0.000    0.000    0.000 C:\Users\Cynthia\Documents\HPC\kmeans_pure_python.py:28(<listcomp>)
        1    0.000    0.000    0.000    0.000 C:\Users\Cynthia\Documents\HPC\kmeans_pure_python.py:10(initialize_centroids)
        1    0.000    0.000    0.000    0.000 C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.11_3.11.2288.0_
        3    0.000    0.000    0.000    0.000 C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.11_3.11.2288.0_
andbits)
       32    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {built-in method _abc._abc_subclasscheck}
        5    0.000    0.000    0.000    0.000 C:\Users\Cynthia\Documents\HPC\kmeans_pure_python.py:20(<listcomp>)
        1    0.000    0.000    0.000    0.000 {built-in method _abc._abc_instancecheck}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
        1    0.000    0.000    0.000    0.000 <frozen abc>:117(__instancecheck__)
        1    0.000    0.000    0.000    0.000 <frozen abc>:121(__subclasscheck__)
        5    0.000    0.000    0.000    0.000 {method 'getrandbits' of '_random.Random' objects}
        1    0.000    0.000    0.000    0.000 C:\Users\Cynthia\Documents\HPC\kmeans_pure_python.py:13(<listcomp>)
        1    0.000    0.000    0.000    0.000 <frozen _collections_abc>:315(__subclasshook__)
        3    0.000    0.000    0.000    0.000 {method 'bit_length' of 'int' objects}
        3    0.000    0.000    0.000    0.000 {method 'add' of 'set' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

```
PS C:\Users\Cynthia\Documents\HPC> python -m cProfile -s time kmeans_pure_python.py
         10327 function calls in 0.009 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1800    0.002    0.000    0.005    0.000 kmeans_pure_python.py:6(euclidean_distance)
     1800    0.002    0.000    0.002    0.000 kmeans_pure_python.py:8(<listcomp>)
     1800    0.001    0.000    0.006    0.000 kmeans_pure_python.py:24(<lambda>)
        6    0.001    0.000    0.001    0.000 kmeans_pure_python.py:27(<listcomp>)
      600    0.001    0.000    0.007    0.000 {built-in method builtins.min}
        1    0.001    0.001    0.009    0.009 kmeans_pure_python.py:16(kmeans)
     1836    0.000    0.000    0.000    0.000 {built-in method builtins.sum}
     1800    0.000    0.000    0.000    0.000 {built-in method math.sqrt}
      600    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
       18    0.000    0.000    0.000    0.000 kmeans_pure_python.py:28(<listcomp>)
        1    0.000    0.000    0.000    0.000 kmeans_pure_python.py:10(initialize_centroids)
        1    0.000    0.000    0.000    0.000 C:\Program Files\WindowsApps\PythonSoftwareFoundation.Py
       38    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        3    0.000    0.000    0.000    0.000 C:\Program Files\WindowsApps\PythonSoftwareFoundation.Py
andbits)
        1    0.000    0.000    0.000    0.000 {built-in method _abc._abc_instancecheck}
        1    0.000    0.000    0.000    0.000 {built-in method _abc._abc_subclasscheck}
        6    0.000    0.000    0.000    0.000 kmeans_pure_python.py:20(<listcomp>)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
        1    0.000    0.000    0.000    0.000 <frozen abc>:117(__instancecheck__)
        1    0.000    0.000    0.000    0.000 <frozen abc>:121(__subclasscheck__)
        1    0.000    0.000    0.000    0.000 <frozen _collections_abc>:315(__subclasshook__)
        1    0.000    0.000    0.000    0.000 kmeans_pure_python.py:13(<listcomp>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        3    0.000    0.000    0.000    0.000 {method 'getrandbits' of '_random.Random' objects}
        3    0.000    0.000    0.000    0.000 {method 'bit_length' of 'int' objects}
        3    0.000    0.000    0.000    0.000 {method 'add' of 'set' objects}
```

These results were shown only using pure python, now we can check the results of using the numpy Arrays:

```
PS C:\Users\Cynthia\Documents\HPC> python kmeans_numpy.py
         20696 function calls in 0.033 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     2400    0.012    0.000    0.029    0.000 C:\Users\Cynthia\Documents\HPC\kmeans_numpy.py:7(euclidean_distance)
     2432    0.006    0.000    0.006    0.000 {method 'reduce' of 'numpy.ufunc' objects}
     2400    0.004    0.000    0.013    0.000 c:\Users\Cynthia\Documents\HPC\myenv\Lib\site-packages\numpy\core\fromnumeric.py:71(_wrapreduction)
     2400    0.003    0.000    0.017    0.000 c:\Users\Cynthia\Documents\HPC\myenv\Lib\site-packages\numpy\core\fromnumeric.py:2177(sum)
     2400    0.002    0.000    0.002    0.000 c:\Users\Cynthia\Documents\HPC\myenv\Lib\site-packages\numpy\core\fromnumeric.py:72(<dictcomp>)
        1    0.001    0.001    0.036    0.036 C:\Users\Cynthia\Documents\HPC\kmeans_numpy.py:17(kmeans2)
        8    0.001    0.000    0.033    0.004 C:\Users\Cynthia\Documents\HPC\kmeans_numpy.py:21(<listcomp>)
     2400    0.001    0.000    0.001    0.000 c:\Users\Cynthia\Documents\HPC\myenv\Lib\site-packages\numpy\core\fromnumeric.py:2172(_sum_dispatcher)
       24    0.000    0.000    0.000    0.000 {built-in method numpy.asanyarray}
     2449    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
       16    0.000    0.000    0.000    0.000 {built-in method numpy.array}
       24    0.000    0.000    0.001    0.000 c:\Users\Cynthia\Documents\HPC\myenv\Lib\site-packages\numpy\core\_methods.py:101(_mean)
     2400    0.000    0.000    0.000    0.000 {method 'items' of 'dict' objects}
      800    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
       24    0.000    0.000    0.002    0.000 c:\Users\Cynthia\Documents\HPC\myenv\Lib\site-packages\numpy\core\fromnumeric.py:3385(mean)
       24    0.000    0.000    0.000    0.000 c:\Users\Cynthia\Documents\HPC\myenv\Lib\site-packages\numpy\core\_methods.py:67(_count_reduce_items)
        8    0.000    0.000    0.002    0.000 C:\Users\Cynthia\Documents\HPC\kmeans_numpy.py:25(<listcomp>)
        1    0.000    0.000    0.000    0.000 C:\Users\Cynthia\Documents\HPC\kmeans_numpy.py:11(initialize_centroids)
        8    0.000    0.000    0.000    0.000 c:\Users\Cynthia\Documents\HPC\myenv\Lib\site-packages\numpy\core\numeric.py:2378(array_equal)
        8    0.000    0.000    0.000    0.000 {method 'argmin' of 'numpy.ndarray' objects}
       24    0.000    0.000    0.000    0.000 C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.11_3.11.2288.0_x64__qbz5n2kfra8p0\Lib\contextlib.py:104(__init__)
       48    0.000    0.000    0.000    0.000 c:\Users\Cynthia\Documents\HPC\myenv\Lib\site-packages\numpy\core\_ufunc_config.py:452(_no_nep50_warning)
       24    0.000    0.000    0.000    0.000 C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.11_3.11.2288.0_x64__qbz5n2kfra8p0\Lib\contextlib.py:141(__exit__)
       24    0.000    0.000    0.000    0.000 C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.11_3.11.2288.0_x64__qbz5n2kfra8p0\Lib\contextlib.py:299(helper)
       48    0.000    0.000    0.000    0.000 {built-in method builtins.next}
       24    0.000    0.000    0.000    0.000 C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.11_3.11.2288.0_x64__qbz5n2kfra8p0\Lib\contextlib.py:132(__enter__)
       24    0.000    0.000    0.000    0.000 {method 'reset' of '_contextvars.ContextVar' objects}
        8    0.000    0.000    0.000    0.000 c:\Users\Cynthia\Documents\HPC\myenv\Lib\site-packages\numpy\core\fromnumeric.py:1236(argmin)
        8    0.000    0.000    0.000    0.000 c:\Users\Cynthia\Documents\HPC\myenv\Lib\site-packages\numpy\core\fromnumeric.py:53(_wrapfunc)
```

And now using Cython:

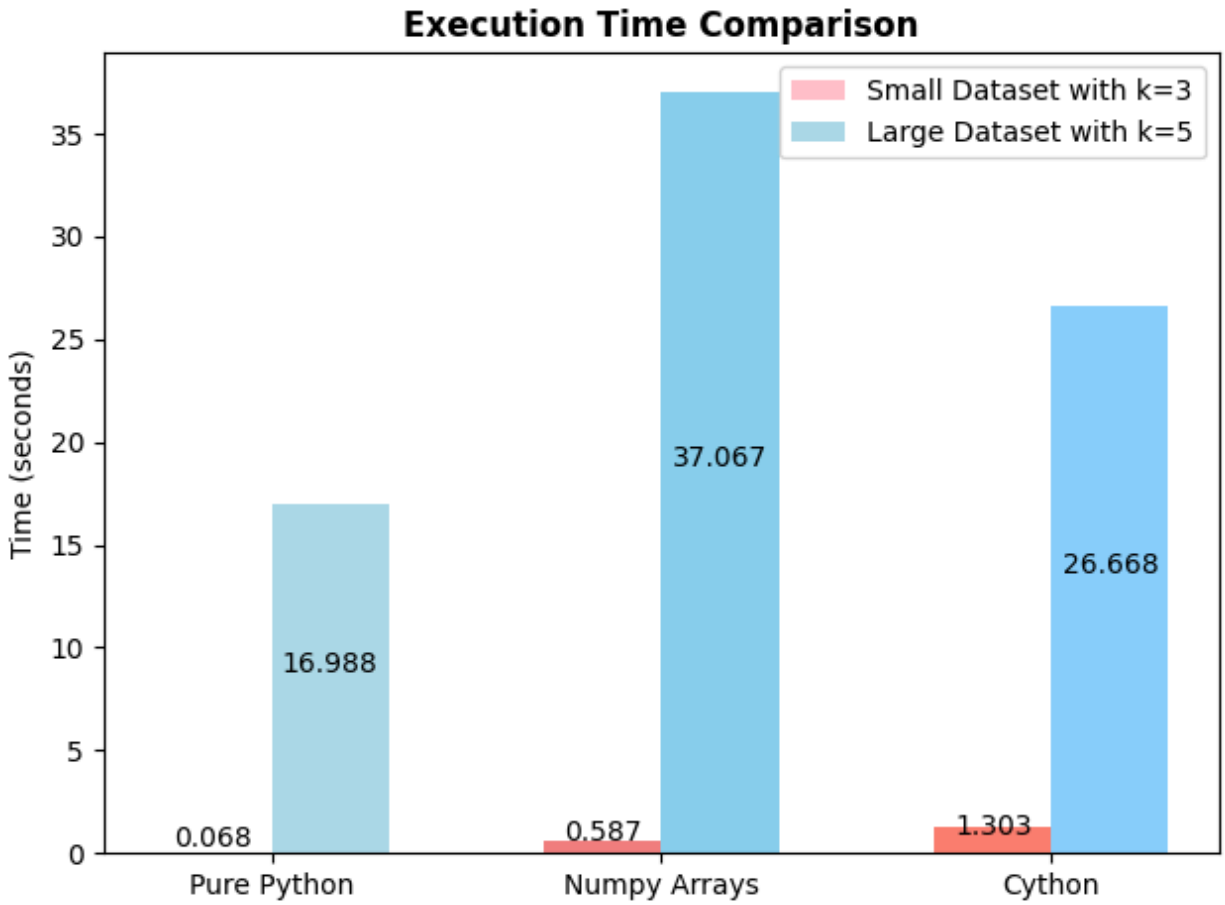
```
PS C:\Users\Cynthia\Documents\HPC> python kmeans_cython_runner.py
Centroids:
[15.11697058 64.06844866]
[49.85508062 17.65220266]
[76.3584618  66.32294168]

Clusters:
[[ 2.2313578  81.85977106]
 [24.33275361 82.77000747]
 [ 1.17874185 90.77058495]
 [15.59856839 40.88258231]
 [31.2015635  61.17289322]
 [11.44812982 61.40309485]
 [ 0.31446744 46.72736896]
 [22.51551538 47.32154367]
 [ 8.2400814  29.12221485]
 [ 2.50869434 51.67259028]
 [19.64869176 93.82782099]
 [27.79405011 91.28304379]
 [18.59877145 38.71499144]
 [ 0.24715976 76.18870525]
 [10.68842672 53.03401829]
 [12.72669418 61.71765828]
 [ 5.41927242 93.90312161]
 [31.479234   48.88072679]
 [18.41393038 75.18181676]
 [16.10839749 67.95283372]
 [31.38874262 45.80202254]
 [18.55284514 60.97043016]
 [12.65069024 45.01021826]
 [ 8.49976224 55.50757718]
```

```
 67/24    0.002    0.000    0.005    0.000 _parser.py:516(_parse)
   312    0.002    0.000    0.002    0.000 {method 'reduce' of 'numpy.ufunc' objects}
 10001    0.002    0.000    0.002    0.000 multiarray.py:153(concatenate)
   146    0.002    0.000    0.051    0.000 <frozen importlib._bootstrap>:1054(_find_spec)
   586    0.002    0.000    0.002    0.000 _inspect.py:65(getargs)
   136    0.002    0.000    0.002    0.000 {method '__exit__' of '_io._IOBase' objects}
   114    0.002    0.000    0.050    0.000 <frozen importlib._bootstrap_external>:1007(get_code)
 10000    0.002    0.000    0.002    0.000 shape_base.py:77(_atleast_2d_dispatcher)
 149/2    0.002    0.000    0.207    0.103 <frozen importlib._bootstrap>:1165(_find_and_load)
 312/6    0.001    0.000    0.003    0.001 arrayprint.py:789(recurser)
   331    0.001    0.000    0.002    0.000 functools.py:35(update_wrapper)
  4278    0.001    0.000    0.001    0.000 {method 'startswith' of 'str' objects}
  1097    0.001    0.000    0.002    0.000 <frozen importlib._bootstrap_external>:119(<listcomp>)
   527    0.001    0.000    0.002    0.000 <frozen importlib._bootstrap>:100(acquire)
 116/1    0.001    0.000    0.412    0.412 {built-in method builtins.exec}
    14    0.001    0.000    0.490    0.035 __init__.py:1(<module>)
  1245    0.001    0.000    0.002    0.000 typing.py:1304(__setattr__)
   228    0.001    0.000    0.006    0.000 <frozen importlib._bootstrap_external>:437(cache_from_source)
   527    0.001    0.000    0.002    0.000 <frozen importlib._bootstrap>:179(_get_module_lock)
    28    0.001    0.000    0.002    0.000 inspect.py:867(cleandoc)
   317    0.001    0.000    0.009    0.000 overrides.py:142(decorator)
  5088    0.001    0.000    0.001    0.000 {method 'endswith' of 'str' objects}
137/22    0.001    0.000    0.003    0.000 _compiler.py:37(_compile)
   527    0.001    0.000    0.001    0.000 <frozen importlib._bootstrap>:125(release)
   296    0.001    0.000    0.002    0.000 typing.py:168(_type_check)
   605    0.001    0.000    0.004    0.000 _inspect.py:96(getargspec)
   278    0.001    0.000    0.005    0.000 overrides.py:83(verify_matching_signatures)
   300    0.001    0.000    0.001    0.000 _methods.py:67(_count_reduce_items)
  1519    0.001    0.000    0.002    0.000 typing.py:1252(_is_dunder)
 149/2    0.001    0.000    0.206    0.103 <frozen importlib._bootstrap>:1120(_find_and_load_unlocked)
250/40    0.001    0.000    0.191    0.005 <frozen importlib._bootstrap>:1207(_handle_fromlist)
   159    0.001    0.000    0.002    0.000 typing.py:251(_collect_parameters)
```

Results & Analysis

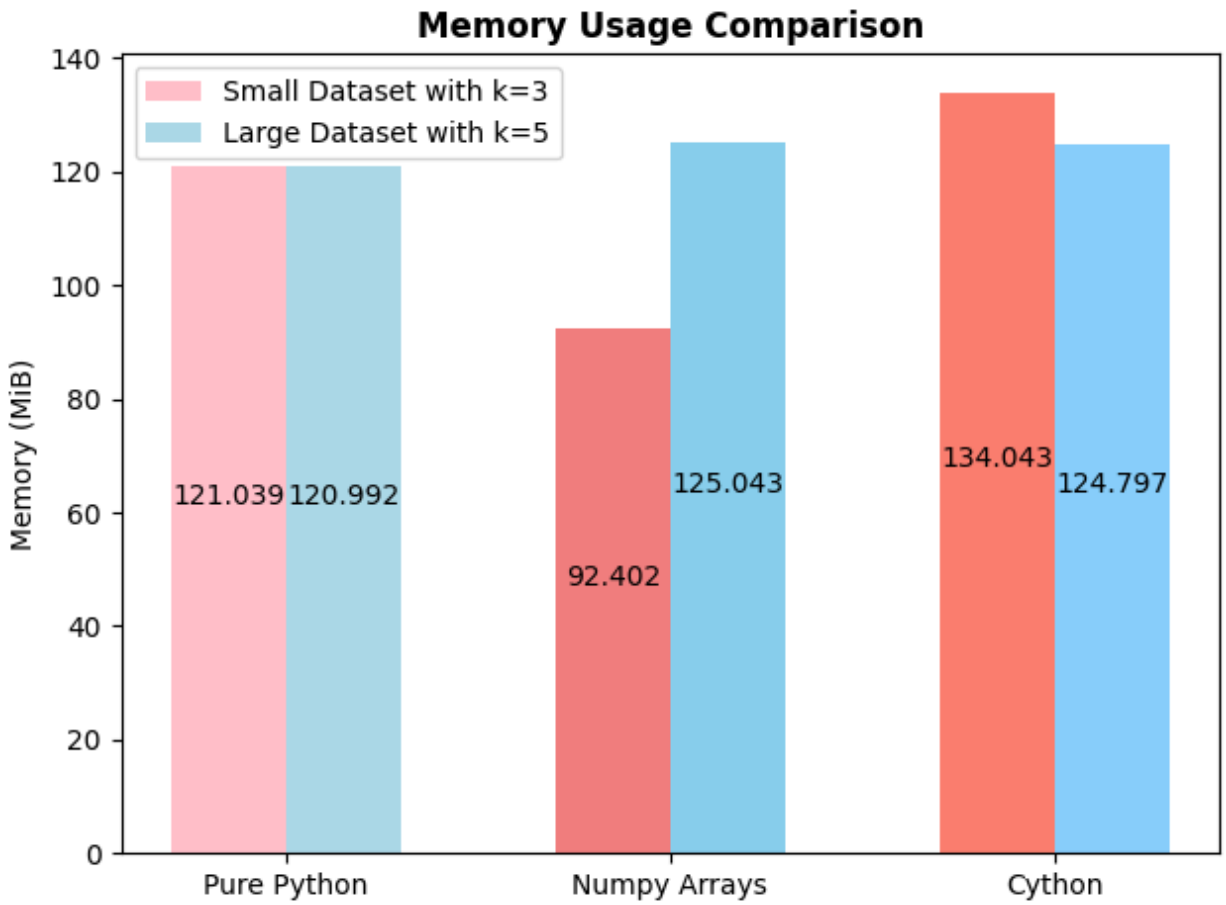
All our results and the source code can be found here.

Our results show that the execution time varies as well as the memory usage depending on the amount of data used.



As we can see, the pure python is faster in execution time in both small and large datasets, followed by Cython and lastly, the NumPy Arrays.

Now speaking in terms of memory usage, we can analyze the next graph.



As we can observe, the NumPy Arrays use less memory than its counterparts in small datasets, in large datasets however, they all have a similar memory usage, but Pure Python is better in this context.

Conclusions

“Each implementation taught us something important about performance optimization. We learned that while pure Python is invaluable for rapid development and prototyping, for applications that demand maximum performance, such as in HPC, it is imperative to turn to tools such as NumPy and Cython. These tools not only improve performance but also challenge us to think about how our code interacts with the underlying hardware, a crucial consideration for high performance computing. The experience of compiling and profiling with Cython, despite its challenges, showed us that low-level optimizations can have a dramatic impact on the performance of our applications, which helps us gain experience for our HPC subject matter.

We observed that from our implementations, Cython got the best results in terms of performance, showing the importance of compiler-level optimizations and the efficiency that can be achieved by moving closer to the hardware, an essential consideration in HPC. This underscores a key principle in high-performance computing: the need to balance development efficiency and code execution.

But, he has to say that the pure Python version, although intuitive and straightforward, turns out to be the least efficient in terms of speed and memory usage. This is mainly due to the interpreted nature of Python and its handling of high-level data types, which introduces significant overhead compared to compiled languages or approaches closer to hardware.

On the other hand, the implementation using NumPy shows considerable improvement in both aspects. NumPy, by operating closer to the hardware and leveraging optimized and compiled libraries, such as BLAS and LAPACK, for numerical operations, drastically reduces the computation time and memory required. This demonstrates how the use of specialized libraries can be crucial in HPC, where efficiency is key.”

References

Ramírez, L. (2023, 5 enero). Algoritmo k-means: ¿Qué es y cómo funciona? Thinking For Innovation. https://www.iebschool.com/blog/algoritmo-k-means-que-es-y-como-funciona-big-data/#:~:text=El%20algoritmo%20k-means%20es%20un%20m%C3%A9todo%20de%20agrupamiento,entre%20s%C3%AD%20que%20los%20puntos%20en%20otros%20clusters.

Voc. (2024, 15 abril). GitHub - VOC12/K-MEANS_IMPLEMENTATIONS. GitHub. https://github.com/VOC12/K-MEANS_IMPLEMENTATIONS