uc3m | Universidad Carlos III de Madrid

Master Degree in Big Data Analytics

2023-2024

Technological fundamentals in the big data world

k-means: serial, multiprocessing and threading

Gonzalo España-Heredia Llanza (100365421)

Fabio Scielzo Ortiz (100374708)

Teacher in charge: Jesús Carretero Perez

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Objectives

This project aims to implement the k-means algorithm in both serial and parallel computing paradigms using Python and subsequently measure the speedup achieved by the parallel implementation. In the context of parallelism, we will explore both multi-processing and multi-threading approaches.

The comparison between these different implementations will be conducted over the entire program's execution, which involves the following key steps:

- Execute the k-means algorithm (serial or parallel) on the original data for varying values of k within the range 2 to 15 and calculate their Within-Cluster Sum of Squares ($WCSS$).
- Utilize the aforementioned results to determine the optimal number of clusters that best fits the data.
- Construct the final model with the optimal value of k, as determined in the previous step.

For each implementation of the algorithm, we will provide a comprehensive explanation of the program's design approach, along with key metrics related to execution time.

1.2. Data

1.2.1. Data generation

For the generation of the dataset used in this project, we employed the *Python* script *computers-generator.py*, which was provided during our course. Our dataset consists of 1 million observations (rows) and encompasses nine variables that incorporate a combination of numerical and categorical data.

1.2.2. Data Cleaning

Upon data ingestion, we optimized the reading process by specifying data types to expedite the operation.

While the generated dataset exhibited overall cleanliness, we undertook several essential modifications to enhance its suitability for analysis:

- We designated the *id* variable as the index for the data frame, obviating the need for later removal.
- Standard encoding was applied to the binary variables (columns) *cd* and *laptop*.
- To facilitate further processing, the data frame was transformed into a 2D array using *Numpy*.

1.2.3. Python Implementation

To utilise this function effectively, make sure that the data set is named *computers.csv* and stored under a *data* folder located in the project's working directory.

The function returns a 2D *NumPy* array and the names of the columns for further analysis and processing.

```python
def fetch_data():
    """
    Function to read and process data into a numpy array in order to apply the KMEANS algorithm.
    """

    try:
        df = pd.read_csv('./data/computers.csv', header=0, index_col=0,
        engine='pyarrow', dtype={'price': np.int32, 'speed': np.int32,
        'hd': np.int32, 'ram': np.int32, 'screen': np.int32, 'cores': np.int32,
        'cd': str, 'laptop': str, 'trend': np.int32})

    except FileNotFoundError or OSError:

        print('ARCHIVO "./data/computers.csv" NO ENCONTRADO. Asegurate de tener los datos dentro de data/ y que
              el nombre sea computers.csv')

    else:

        df['cd'] = np.where(df['cd'] == 'no', 0, 1)
        df['laptop'] = np.where(df['laptop'] == 'no', 0, 1)

        return df.to_numpy(), df.columns.to_list()
```

2. K-MEANS ALGORITHM: PERFORMANCE ANALYSIS

The following Python class contains our implementation of the k-means algorithm.

```python
class KMEANS():
    """
    Class to compute k-means clustering with k clusters.
    """

    def __init__(self, data, k, tol=1e-4, seed=100365421, max_iterations=100):
        """
        Initializes the KMEANS object.

        Parameters:
            data: np.ndarray with numerical data (2D)
            k: number of clusters
            tol: tolerance to stop iterations
            max_iterations: maximum number of iterations
        """
        self.data = data
        self.k = k
        self.tol = tol
        self.max_iterations = max_iterations
        self.iter = 0
        self.seed = seed
        self.centroids = None
        self.labels = None
        self.wcss = []

    def _reallocate_centroids(self):
        """
        Reallocates centroids based on the current cluster assignments.
        """
        new_centroids = np.array([self.data[self.labels == i].mean(axis=0) for i in range(self.k)])
        self.centroids = new_centroids

    def _get_wcss(self):
        """
        Computes the Within-Cluster Sum of Squares (WCSS).
        """
        return np.sum(np.linalg.norm(self.data - self.centroids[self.labels], axis=1) ** 2)

    def fit(self):
        """
        Fits the KMEANS clustering model to the data.
        """
        # Initialize centroids
        np.random.seed(self.seed)
        self.centroids = self.data[np.random.choice(len(self.data), self.k, replace=False)]

        # Iterate until convergence or max_iterations
        for i in range(self.max_iterations):
            # Compute distances between data points and centroids
            distances = cdist(self.data, self.centroids)

            # Assign data points to the nearest centroid
            self.labels = np.argmin(distances, axis=1)

            # Compute the Within-Cluster Sum of Squares (WCSS)
            self.wcss.append(self._get_wcss())

            # Check for convergence
            if i > 0 and abs(self.wcss[-2] - self.wcss[-1]) / self.wcss[-2] < self.tol:
                break

            # Reallocate centroids
            self._reallocate_centroids()

            self.iter = i + 1

        return self
```

The `KMEANS` class is designed for performing k-means clustering. It is initialised with the following parameters:

- `data`: np.ndarray with numerical data (2D).
- `k`: the desired number of clusters.
- `tol`: the convergence tolerance.
- `seed`: a random seed for reproducibility.
- `max_iterations`: the maximum number of iterations.

`fit()` Method

The `fit()` method is the core of the `KMEANS` class and implements the K-means algorithm. It follows these steps:

1. Random Initialization of Centroids: Instead of randomly selecting centroids from the entire data space, we choose to initialize centroids using a random selection of data points, enhancing the robustness of initialization.
2. Compute L2 Distances: Utilizes Scipy's `cdist()` function to calculate L2 distances from each data point to each centroid.
3. Assign Data Points: Determines the closest centroid for each data point.
4. Calculate Within-Cluster Sum of Squares (WCSS): Computes the WCSS as a measure of the variance within clusters.
5. Convergence Check and Centroid Reallocation: Checks for convergence by comparing the change in WCSS to the tolerance (`tol`). If not converged, it reallocates centroids by computing the mean of each variable for all observations within a cluster.

`_reallocate_centroids()` Method

This is a private method that reallocates centroids based on the current cluster assignments. It computes the mean of each variable for all observations within a cluster, updating the centroids.

`_get_wcss()` Method

Another private method that calculates the Within-Cluster Sum of Squares (WCSS) as a measure of the variance within clusters.

2.1. Selecting the Optimal k

The previously described class provides the essential algorithm for computing k-means clustering for a given value of k. However, the most computationally intensive aspect of our program is the process of selecting the optimal value for k. This selection requires running the algorithm multiple times, which can be time-consuming. To mitigate this computational burden, we leverage parallelization techniques to expedite the execution of our program.

For each of the three approaches—serial, multi-processing, and multi-threading—we have tailored our method for determining the best k to align with the characteristics of each parallel computing paradigm. This optimization allows us to harness the power of parallelization and significantly reduce the execution time of our program.

2.1.1. Serial Approach

In the serial approach, we have developed a function that systematically iterates over a range of k values and fits the clustering model with each value of k. During each iteration, the resulting Within-Cluster Sum of Squares ($WCSS$), obtained either through convergence or reaching the maximum number of iterations, is computed and appended to a list. Once the loop concludes, we invoke the function *choose_best_k*(), which, based on the list of $WCSS$ values and the range of k values, selects the optimal value of k by analysing the second derivatives:

```
def choose_best_k(x, y, tol=10e-2):

    # Calculate the first derivative
    first_derivative = np.gradient(y)
    # Calculate the second derivative
    second_derivative = np.gradient(first_derivative)

    x_range = x

    res = np.argmax(second_derivative) + 2 # +2 because first arg is 0, and because K starts at 2

    return res, x_range, second_derivative

def compute_best_K(data, space=range(2, 16)):
    """
    Function to obtain the best K in a range of values for KMEANS clustering.

    Parameters:
        data: Input data for clustering.
        space: Range of K values to evaluate.

    Returns:
        The best K value.
    """
    wcss_k = []
    for i in space:
        res = KMEANS(data, k=i).fit()
        wcss_k.append(res.wcss[-1])
    res, x_range, values = choose_best_k(np.arange(2, 16), np.array(wcss_k))
    return res, x_range, values, wcss_k

if __name__ == '__main__':

    import time
    import numpy as np
    from utils.utils import *
    from scipy.spatial.distance import cdist
    from sklearn.decomposition import PCA
```

```
    from sklearn.preprocessing import StandardScaler

    output_path = 'data/output/'
    filename = 'Serial_'

    X, cols = fetch_data()

    # Init timer
    start_time_best_k = time.time()

    # Compute best K
    best_k, x_range, second_derivative_values, wcss_list = compute_best_K(X)
    best_k_time = time.time()
    # Compute KMEANS with best K from previous step
    final_clustering = KMEANS(X, k=best_k).fit()
```

2.1.2. Multiprocessing Approach Using `starmap()`

In this section, we describe the implementation and workings of the multiprocessing approach using the ‘starmap()‘ function to efficiently compute the optimal k in K-means clustering.

Code Explanation

The code snippet below demonstrates how the multiprocessing approach is used to execute K-means clustering with various values of k in parallel.

```

def execute_kmeans(data, k):

    kmeans = KMEANS(data, k).fit()
    return kmeans.wcss[-1]

if __name__ == '__main__':

    output_path = 'data/output/'
    filename = 'multiprocessing_'

    X, cols = fetch_data()

    p = mp.Pool(processes=int(mp.cpu_count()/2))

    start_time_best_k = time.time()
    wcss_results = p.starmap(execute_kmeans, [(X, k) for k in range(2,16)])
    p.close()
    p.join()
    best_k, x_range, second_derivative_values = choose_best_k(np.arange(2, 16), np.array(wcss_results))
    final_clustering = KMEANS(X, k=best_k).fit()
    end_time_best_k = time.time()

    elapsed_time_bestk = end_time_best_k - start_time_best_k
```

Here is a step-by-step explanation of how the code works:

1. The `execute_kmeans` function takes two arguments: the data to be clustered and the number of clusters (`k`). It initializes a K-means object with the specified `k` and fits the model to the data. The Within-Cluster Sum of Squares (`WCSS`) value after convergence is returned.

2. Within the `if __name__ == '__main__':` block, a multiprocessing pool (`p`) is created with a number of processes equal to half of the available CPU cores. This allows multiple K-means clustering tasks to be executed concurrently.

3. The `starmap()` method is used to parallelize the execution of the `execute_kmeans` function over a range of `k` values (from 2 to 15). The results (`WCSS` values) are collected in the `wcss_results` list.

4. After all parallel tasks are completed, the code calls the `choose_best_k` function to identify the best value of `k` based on the `WCSS` results.

5. The K-means clustering is performed again with the optimal `k`, and the final clustering results are stored in the `final_clustering` variable.

6. The elapsed time for finding the best `k` is calculated by measuring the time before and after the process. The result is stored in the `elapsed_time_bestk` variable.

2.1.3. Multithreading Approach

In this section, we describe the implementation and workings of the multithreading approach for computing the optimal k in K-means clustering.

Code Explanation

The following code demonstrates how multithreading is used to execute K-means clustering with various values of k in parallel:

```python
def execute_kmeans(k, output):

    kmeans = KMEANS(data=X, k=k).fit()
    output.append((k, kmeans.wcss[-1]))

def sort_values_by_keys(tuple_list):
    # Sort the list of tuples by the first element (keys)
    sorted_tuples = sorted(tuple_list, key=lambda x: x[0])

    # Extract the second element (values) from the sorted tuples
    sorted_values = [t[1] for t in sorted_tuples]

    return sorted_values


if __name__ == '__main__':

    output_path = 'data/output/'
    filename = 'multithreading_'

    output_list=[]
    jobs=[]
    ks = [i for i in range(2,16)]

    start_time_best_k = time.time()

    for k in ks:
        thread = threading.Thread(target=execute_kmeans, args=(k, output_list))
        jobs.append(thread)

    for j in jobs:
        j.start()

    for j in jobs:
        j.join()

    print(output_list)

    # Get WCSS results sorted by the key (number of clusters)
    wcss_results=sort_values_by_keys(output_list)

    # Choose best K
```

```
    best_k, x_range, second_derivative_values = choose_best_k(np.arange(2, 16), np.array(wcss_results))

    final_clustering = KMEANS(X, k=best_k).fit()

    end_time_best_k = time.time()

    elapsed_time_bestk = end_time_best_k - start_time_best_k
```

Here is a step-by-step explanation of how the code works:

1. The `execute_kmeans` function takes two arguments: the number of clusters (`k`) and an `output` list. It initializes a K-means object with the specified `k` and fits the model to the data (`X`). The resulting Within-Cluster Sum of Squares (`WCSS`) is appended to the `output` list as a tuple.
2. The `sort_values_by_keys` function is used to sort a list of tuples by the first element (keys), and then extract the second element (values) from the sorted tuples.
3. Within the `if __name__ == '__main__':` block, a loop is used to create a thread for each value of `k`. These threads are appended to the `jobs` list.
4. The `start()` method is called on each thread, initiating the execution of K-means clustering for each `k` value in parallel.
5. After all threads have completed their tasks using the `join()` method, the `output_list` contains tuples of `k` values and their corresponding `WCSS` values.
6. The `WCSS` values are sorted by the number of clusters (`k`) using the `sort_values_by_keys` function.
7. The best `k` is determined by calling the `choose_best_k` function, and the final K-means clustering is performed with the optimal `k`.
8. The elapsed time for the entire process is calculated.

This approach leverages multithreading to concurrently execute K-means clustering tasks for various values of `k`, optimizing the search for the optimal `k` in a more time-efficient manner.

2.1.4. Time comparison and metrics

When considering the execution time of each approach, it becomes evident that implementing parallelism yields significant performance improvements. The serial implementation boasts an average execution time of under 4 minutes for 1 million data points, while the multiprocessing and multithreading approaches achieve execution times of approximately 84.98 and 90.78 seconds, respectively.

In terms of performance and speedup, it is noteworthy that both parallel implementations outperform the serial approach by an order of magnitude in terms of computational efficiency, resulting in nearly a threefold reduction in execution time (speedup).

Below you will find 3 tables with detailed data.

Approach	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Serial	225.3	231.2	240.4	**239.6**	246.6	254.3
Multiprocessing	81.16	83.57	84.01	**84.98**	86.29	89.87
Threading	88.03	88.71	90.64	**90.78**	92.72	93.82

TABLE 2.1. TIME - COMPARING SERIAL, MULTIPROCESSING AND MULTITHREADING.

Approach	Performance
Serial	4.17e-3
Multi-Processing (starmap)	1.17e-2
Multi-Threading	1.11e-2

TABLE 2.2. PERFORMANCE COMPARISON

Approach	Speedup
Multi-Processing (starmap)	2.819
Multi-Threading	2.639

TABLE 2.3. SPEEDUP COMPARISON VS SERIAL APPROACH

2.1.5. Elbow plot and optimal k

With the three approaches (serial, multiprocessing and threading) we have obtained the same results in terms of Elbow plot and the optimal value of k. This was expected as we set a seed in the initialisation process in order to have reproducibility.

In our case, the optimal value of k is 3, and it has been obtained using the second derivative of the Elbow plot. Specifically, the optimal k has been define as the k for which the second derivative is maximum, and, as you can see in the plot, that k is 3.

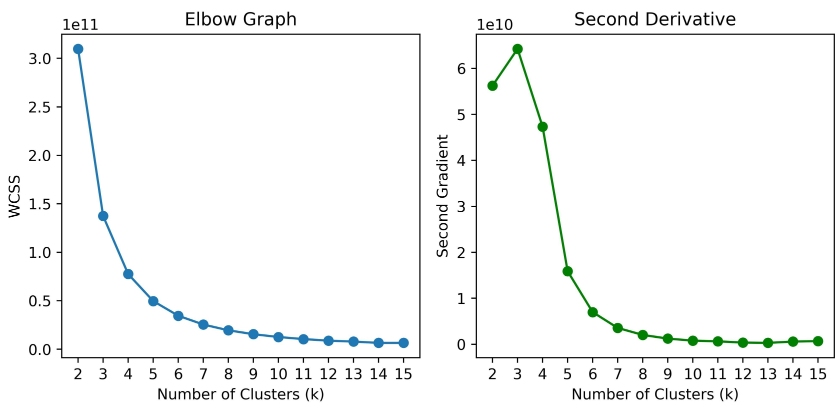

Fig. 2.1. Elbow plot and second derivative

2.2. Final clustering analysis

2.2.1. Cluster with the highest average price

At this point, we have computed the cluster with the highest average price, as well as that average price. The cluster with the highest average price is **Cluster 1** and has an average price of 3709.3€

2.2.2. PCA-Clusters plot

In this section, we have performed a Principal Component Analysis on our dataset.

The first two components have been selected and plotted, distinguishing the clusters defined by our k-means class with the optimal k, that is, $k = 3$.

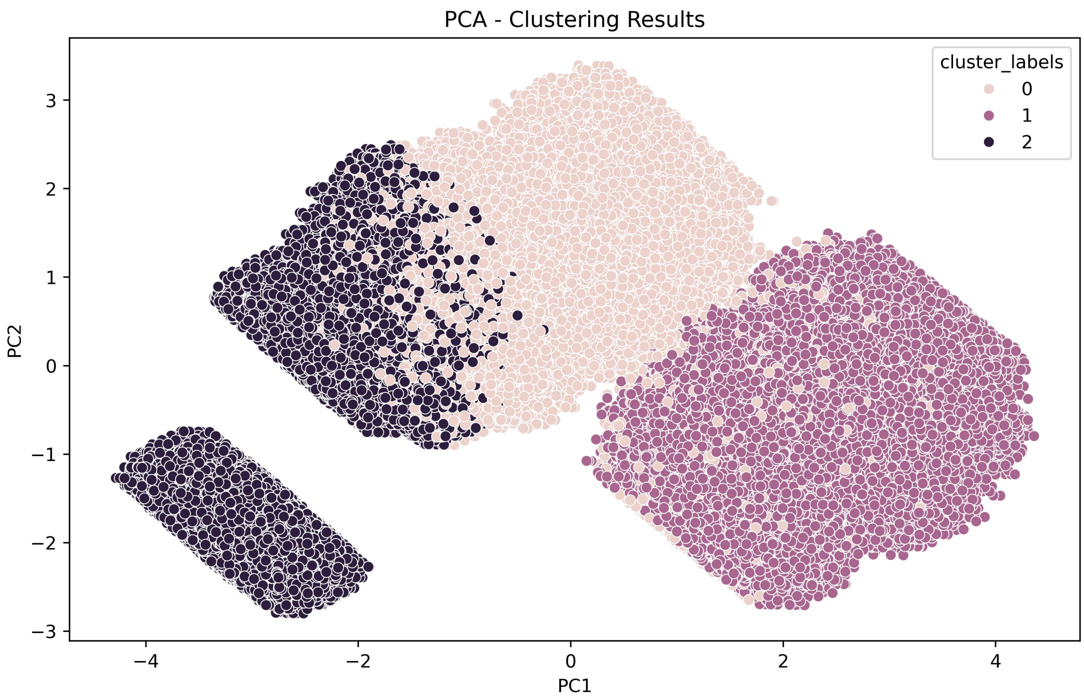

Fig. 2.2. PCA-Clusters plot

2.2.3. Heat-map plot

A heatmap has been generated to visualize the normalized centroids of the clusters obtained for $k = 3$. This heatmap aids in the interpretation of the clusters based on the dataset's variables.

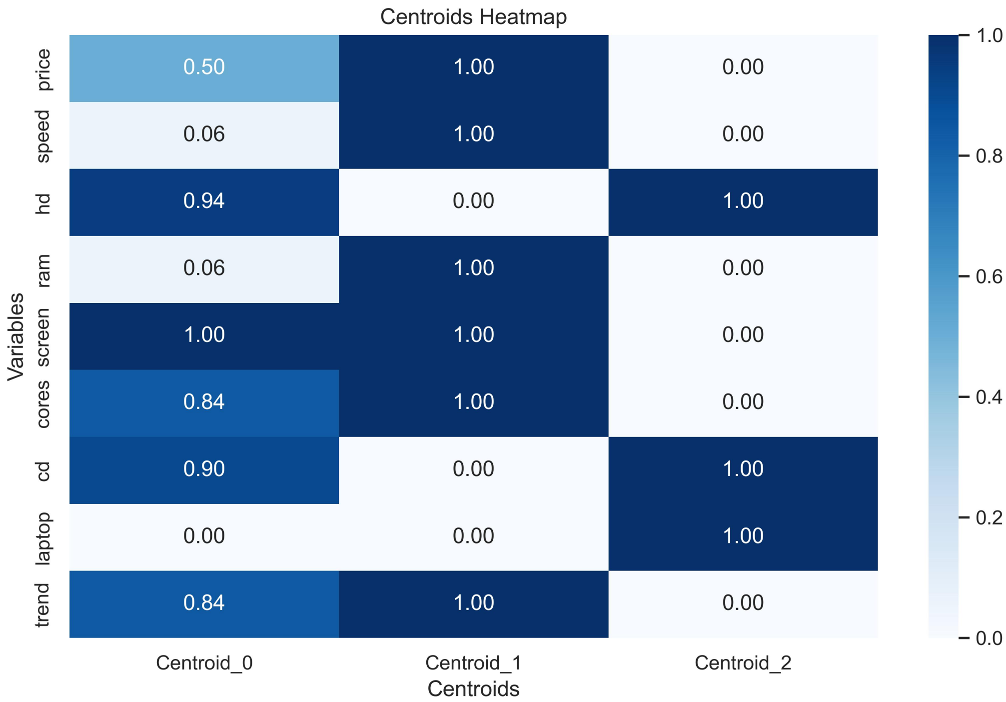

Fig. 2.3. Heat-maps plot

The heatmap provides insights into the clusters' characteristics. For example, it reveals that cluster 1 has the highest average price, while cluster 2 has the lowest. This analysis applies to any variable within the dataset. For instance, cluster 1 also exhibits the highest number of cores and RAM, while cluster 2 demonstrates the lowest values in these aspects.

3. CONCLUSIONS

In conclusion, the time comparison and performance metrics clearly indicate the advantages of implementing parallelism in the context of the k-means algorithm. The serial implementation, while feasible, exhibits a longer execution time, taking just under 4 minutes to process 1 million data points. In contrast, both the multiprocessing and multithreading approaches achieve substantially improved execution times, with approximately 84.98 and 90.78 seconds, respectively. This represents a remarkable reduction in processing time.

Furthermore, in terms of performance and speedup, the parallel implementations significantly outperform the serial approach. Both the multiprocessing and multithreading approaches offer an order of magnitude improvement in computational efficiency, resulting in a nearly threefold reduction in execution time, as reflected in the speedup metrics.