PC-2016/17 Course Project Template

Kai Schewina
kai.schewina@fu-berlin.de

Abstract

K-means is a simple clustering technique, that partitions the data-set in k clusters. This is achieved by iterative assignment to the closest centroid with regard to squared euclidean distance. We investigate a sequential and a parallel implementation of the algorithm in Python and compare the respective speedup. RESULTS

1. Introduction

K-means is a simple clustering technique, that partitions the data-set in k clusters. This finds multiple applications, such as image

The following displays the process in pseudocode:

1. Set number of clusters k
2. Initialize k random centroids in the feature space
3. Iterate until stopping criterion (i.e., no change in cluster assignments)
   (a) For each data point calculate squared euclidean distance to all centroids
   (b) Assign data points to the closest centroid
   (c) Reposition centroids to the mean of all assigned data points

Note that the k-means algorithm is highly dependent on the initial positioning of the centroids. There is a stream of literature on improving on the random positioning to achieve more consistent results [1]. As this is not central to the question of parallelization, we opted for using random initialization.

1.1. Distance criterion

The k-means algorithm can be applied in an m-dimensional space. We test multiple setups to evaluate the difference. The measurement for point 3.a) in the pseudocode is the squared euclidean distance, that is defined as follows for two points p and q in the m-dimensional space:

$$d(p, q) = \sqrt{(p_1 + q_1)^2 + ... + (p_m + q_m)^2}$$

To test the impact of the dimensionality, we try out multiple setups. See the experimental section for details.

1.2. Selection of k

The k-means algorithm profits from a low amount of parameters. The amount of clusters k needs to be provided initially. For choosing k there exist a number of methods. [2] review six techniques, namely rule of thumb, elbow method, information criteria, information theory, silhouette and cross-validation. As, this is not central to the question of k-means parallelization, we set the parameter as fixed.

1.3. Stopping criterion

Second, there is a stopping criterion to be defined. In the original version, that is used here, the algorithm stops after no observations changed their cluster assignment after an iteration. There are multiple extensions, for example early stop in case of convergence [3]. As with the cluster

amount k and the initialization issue, we apply the standard stopping criterion.

2. Design

In the following we discuss the aspects of the design. As a general approach, Pythons library *multiprocessing* is used for parallelization of the process. We use a data-parallelization approach, by chunking the data in c chunks, with c being the amount of cores that are used for parallelization.

The data generation process was implemented by using the sklearn library and its make_blobs function, that is specifically designed to create blobs of data in m-dimensional space. The centroids are initialized by using the below function. To ensure reproducibility a random seed is set and we draw from a uniform distribution from the lower and upper values of each dimension. It returns a list of length k that contain m coordinates.

```
def initialize_centroids(lower, upper,
    n_centroids, n_features):
    np.random.seed(42)
    centroids = np.random.uniform(lower,
        upper, [n_centroids, n_features])
    return centroids
```

The two functions that are iterated through are the ones assigning the clusters and repositioning the centroids. The assign cluster function takes chunks of data (or the full data in case of the sequential version) and calculates the euclidean distance in m-dimensional space between all n observations and all k centroids. It then returns a list of length n that contains the number of the cluster each observation has been assigned to.

```
def assign_clusters(chunk, centroids):
    clusters = []
    for index, e in chunk.iterrows():
        dist_list = []
        for i, c in enumerate(centroids):
            dist = [(a - b) ** 2 for a, b
                in zip(e, c)]
            dist = math.sqrt(sum(dist))
            dist_list.append(dist)
        clusters.append(dist_list.index(min
            (dist_list)))
    return clusters
```

The function to reposition the centroids takes the data, centroids, number of centroids and number of features and returns the new coordinates of each centroid. For all centroids, it takes the average of all dimensions of all observations that belong to the respective cluster. It then returns the updated position for each centroid.

```
def reposition_centroids(data, centroids,
    n_centroids, n_features):
    new_centroids = centroids
    for i in range(n_centroids):
        for j in range(n_features):
            new_centroids[i, j] = data.loc[
                data['cluster'] == i, [j]].
                sum() / data[data['cluster']
                 == i].shape[0]
    return new_centroids
```

Finally, we check the stopping criterion by applying a function that checks for changes in the cluster assignments and allows for stopping the iterative part of the algorithm.

```
def check_stop(data, new_clusters):
    new_clusters = pd.DataFrame(
        new_clusters, columns=['cluster'])
    if data["cluster"].equals(new_clusters[
        "cluster"]):
        return True
    else:
        return False
```

The functions are brought together in the main part of the program, that is displayed below for the sequential version of the script. The data is generated, following we initialize the centroids and first cluster assignment. Following, we enter a while loop to iterate until the stopping criterion is met. In each iteration the centroids are repositioned, the clusters reassigned and the stopping criterion checked.

```
def main(n_centroids, n_features, n_samples
    ):

    data, lower, upper = make_data(
        n_samples=n_samples, n_centroids=
        n_centroids, n_features=n_features)

    start_time = time.time()

    centroids = initialize_centroids(lower,
         upper, n_centroids, n_features)
    clusters = assign_clusters(data,
        centroids)
    data["cluster"] = clusters
```

```
    counter = 0
    while True:
        centroids = reposition_centroids(
            data, centroids, n_centroids,
            n_features)
        clusters = assign_clusters(data,
            centroids)
        stop = check_stop(data, clusters)
        if stop:
            break
        else:
            data["cluster"] = clusters
            counter += 1
    end_time = time.time()
    return end_time - start_time
```

Profiling the function reveals that most time is spent on assigning the clusters, especially for a higher n. Therefore the parallel version of the code chunks the data into c chunks and calls the assign_cluster function using pool.map of multiprocessing.

```
...
chunk_size = int(data.shape[0]/
    num_processes)
chunks = [data.iloc[data.index[i:i +
    chunk_size]] for i in range(0, data.
    shape[0], chunk_size)]


with mp.Pool(processes=num_processes) as
    pool:
    result = pool.map(partial(
        assign_clusters, centroids=centroids
        ), chunks)
    ...
```

3. Experiments

We test multiple setups to evaluate the performance. For the amount of cores 2, 4 and 6 cores were tested on a Ryzen 5500U with 16 GB of DDR4 RAM. For the data 1,000, 100,000 and 1,000,000 of data points in 3, 10 and 30-dimensional space were tested out. The detailed runtimes can be found in the appendices.

Parameter	Values
m	[3, 10, 30]
n	[1,000, 100,000, 1,000,000]
c	[2, 4, 6]

Table 1. Parameter Grid

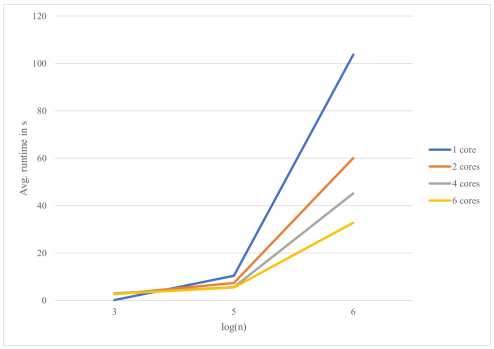

Figure 1. Runtime depending on samples and cores

Method	Frobnability
Theirs	Frumpy
Yours	Frobbly
Ours	Makes one's heart Frob

Table 2. Results. Ours is better.

3.1. References

List and number all bibliographical references in 9-point Times, single-spaced, at the end of your paper. When referenced in the text, enclose the citation number in square brackets, for example [**?**]. Where appropriate, include the name(s) of editors of referenced books.

3.2. Illustrations, graphs, and photographs

All graphics should be centered. Please ensure that any point you wish to make is resolvable in a printed copy of the paper. Resize fonts in figures to match the font in the body text, and choose line widths which render effectively in print. Many readers (and reviewers), even of an electronic copy, will choose to print your paper in order to read it. You cannot insist that they do otherwise, and therefore must not assume that they can zoom in to see tiny details on a graphic.

When placing figures in LaTeX, it's almost always best to use `\includegraphics`, and to specify the figure width as a multiple of the line width as in the example below

```
\usepackage[dvips]{graphicx} ...
\includegraphics[width=0.8\linewidth]
                {myfile.eps}
```

3.3. Color

Color is valuable, and will be visible to readers of the electronic copy. However ensure that, when printed on a monochrome printer, no important information is lost by the conversion to grayscale.

References

[1] P. Fränti and S. Sieranoja. How much can k-means be improved by using better initialization and repeats? *Pattern Recognition*, 93:95–112, 2019.

[2] T. M. Kodinariya and P. R. Makwana. Review on determining number of cluster in k-means clustering. *International Journal*, 1(6):90–95, 2013.

[3] A. Mexicano, R. Rodríguez, S. Cervantes, P. Montes, M. Jiménez, N. Almanza, and A. Abrego. The early stop heuristic: a new convergence criterion for k-means. In *AIP conference proceedings*, volume 1738, page 310003. AIP Publishing LLC, 2016.

4. Appendix

log(n)	cores			
	1	2	4	6
3	0,15	2,31	2,76	3,36
5	16,8	12,02	7,55	7,08
6	209,12	113,04	83,22	67,35

Table 3. Feature dimension m=3

log(n)	cores			
	1	2	4	6
3	0,14	2,61	3,01	2,76
5	10,39	7,42	5,62	5,47
6	103,74	60,04	45,23	32,76

Table 4. Feature dimension m=10

log(n)	cores			
	1	2	4	6
3	0,25	2,31	2,69	3,09
5	16,75	13,37	8,34	7,75
6	167,42	99,17	73,35	57,21

Table 5. Feature dimension m=30