K-means Clustering Using Spark

Kyu Min Shim
kmshim@uwaterloo.ca
University of Waterloo

Figure 1: K-means RGB Clustering of a Raccoon.
From left to right, result with 2 clusters, 5 clusters, and the original image.

ABSTRACT

For the final project, I have decided to implement K-means clustering algorithm using PySpark. Specifically, this K-means implementation is for clustering different colors in images. In addition to the PySpark implementation of K-means, I implemented two other highly optimized variations of K-means for comparison in performance, measured by the runtimes. One uses the NumPy package, and the other uses the fully implemented K-means function from the PySpark.ml.clustering package. My own implementation of K-means was unable to outperform the other two implementations due to resource constraints. However, it showed improved performance against the NumPy implementation as the size of the clustering problem increased with a higher number of clusters and a higher number of iterations. Thus, my own implementation of K-means showed evidence of scalability with access to the right resources.

KEYWORDS

K-means Clustering, Color Clustering, Distributed Computing

1 INTRODUCTION

Clustering refers to the process of grouping together similar points. The points usually belong in a high-dimensional feature space, and the similarity between points is measured by a distance measure in the feature space, where two points are said to be similar if the distance between them is small. Clustering is a type of unsupervised learning algorithm as there is no pre-assignment of correct labels, but instead, such labels are iteratively discovered through the algorithm. A pseudocode for clustering can be found in Algorithm 1.

Suppose there are k clusters, and $S = (S_1, S_2, \ldots, S_k)$ where S is the set of all points and S_i is a subset of S of all points in cluster i. Similarly, $\mu = (\mu_1, \mu_2, \ldots, \mu_k)$ where μ_i is the cluster center of cluster i. For some distance measure d, clustering tries to minimize the overall error (1) by iteratively adjusting the cluster centers μ and the cluster assignments $S = (S_1, S_2, \ldots, S_k)$.

Algorithm 1 Clustering Pseudocode

```
Number of clusters k > 0
Number of iterations t > 0
i ← 0
Randomly choose k points as initial cluster centers
Assign each point to its closest center
while i < t do
    Compute new cluster centers as some statistic of all points in each cluster
    Re-assign each point to its new closest cluster center
    i = i + 1
end while
Return the latest cluster centers
```

$$E(S, \mu) = \sum_{i=1}^{K} \sum_{p \in S_i} d(f_p, \mu_i) \tag{1}$$

A common distance measure used in clustering is the Euclidean distance (L2 norm), and the corresponding optimal cluster centers μ that minimizes equation (1) is the mean of all points in each cluster. If a different distance measure is used, say the absolute deviation (L1 norm), then the optimal cluster centers μ that minimizes equation (1) is the median of all points in each cluster.

The clustering algorithm is simple, but the amount of computation required grows quickly with the number of data points, number of clusters, and number of iterations. Hence, a scalable solution is required to efficiently process large data sets and identify clusters that best categorize each data point. Distributed computing is a solution to this problem. Note that the bulk of the algorithm is calculating the distance between each point and k cluster centers, then identifying the cluster that is closest to the point. In most

cases, $k << N$ where N is the number of data points. Hence, this process can be parallelized by having multiple workers simultaneously calculate the distances between each of the N points and k cluster centers. Once the closest cluster is identified for each point, the points can be aggregated by their clusters and compute the new cluster centers. Ultimately, the runtime of the clustering algorithm can be maintained as the size of the problem grows by utilizing distributed computing.

This paper focuses on K-means clustering, which is one of the most common types of clustering algorithms where the Euclidean distance is used as the distance measure between points. Implementations of K-means clustering in the context of color clustering (RGB clustering) in images are used to demonstrate the results of the clustering algorithm, where data points are defined by the RGB values of each pixel of an image. Each cluster center is defined by the mean RGB values of all points in the cluster. I have implemented my own version of K-means clustering using PySpark, as well as two other implementations of K-means clustering to compare performances. One implementation is a straightforward implementation of the algorithm using NumPy, and it is optimized by minimizing nested loops through the use of NumPy functions. The other implementation is done using the PySpark.ml.clustering package. The details of each implementation are covered below. A sample image of a raccoon in Figure 1 is used to test each implementation. The dimensions of the sample image are 1024×768, hence there are approximately 800000 RGB values in the image.

2 IMPLEMENTATIONS

Python code for all implementations can be found in Appendix A. Note the indentations have been modified to accommodate the format of the report. Full implementations of all code used in this report along with detailed comments can be found at https://git.uwaterloo.ca/kmshim/bigdata2023w/-/blob/main/CS631_Final_clean.ipynb.

2.1 NumPy Implementation

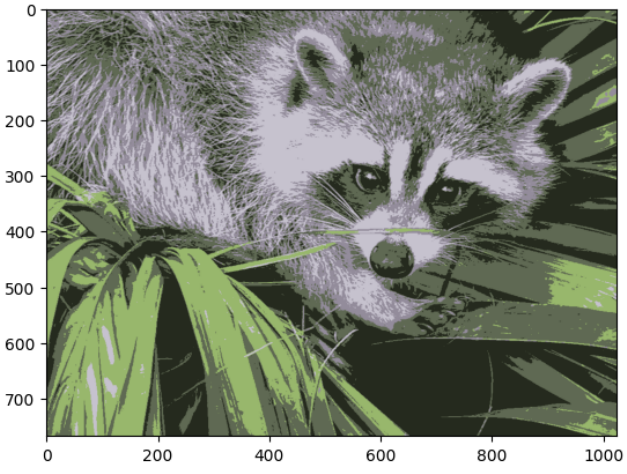

Figure 2: Result of NumPy implementation with 5 clusters and 10 iterations

The NumPy implementation is explained first as it is the easiest to follow. The NumPy implementation stores the image as a multi-dimensional array of RGB tuples. First, k pixels from the image are randomly chosen as the initial cluster centers. Then, the iterative part of the algorithm is implemented using two nested for-loops. A snippet of code from Appendix A.1 for the main iterative section can be seen below.

```
for i in range(num_iter):
shape = (numrows, numcols)
opt_labels = np.full(
    shape,
    fill_value=no_label
)

min_dist = np.full(
    shape,
    fill_value=np.inf
)

col_coord, row_coord = np.meshgrid(
    range(numcols),
    range(numrows)
)

for label in range(k):
    full_dist = np.sum(
        np.square(
            means[label,:3] - img.astype(float)),
        axis=2)
    min_dist = np.where(
        min_dist < full_dist,
        min_dist,
        full_dist
    )
    opt_labels = np.where(
        min_dist < full_dist,
        opt_labels,
        label
    )
```

The first layer of for-loop iterates over the inputted number of iterations. It initializes the *opt_labels* array which stores and updates the closest cluster for each pixel, and the *min_dist* array which stores and updates the distance to its closest cluster for each pixel. The second layer of for-loop iterates over the inputted number of clusters, and calculates the distance between every pixel to its nearest cluster center. If the distance to the current cluster center is closer than the distance to its nearest cluster center so far, then the *opt_labels* array and *min_dist* array are updated accordingly. After the second layer of for-loop completes, the new cluster centers are calculated using the *opt_labels* array and the next iteration for the first layer of for-loop begins.

Even though there are nested for-loops in this implementation, the code is well-optimized using the NumPy functions which allows

element-wise operations for arrays. However, there is no distributed computing involved in this implementation, and the computation time can increase rapidly as the size of the image, the number of clusters, or the number of iterations increases. For example, if the input image was a much larger, detailed image such as a map, the number of pixels would explode to tens of millions and the NumPy implementation would not be suitable for clustering such a large image in a reasonable amount of time. That is, this implementation does not scale well.

2.2 PySpark Implementation

The PySpark implementation stores the image as an RDD, where each row is a tuple of RGB value of a pixel. Unlike the NumPy implementation where data is stored in multidimensional arrays, the PySpark implementation loses spatial information of pixels by storing data in RDD and only retains the RGB values of pixels.

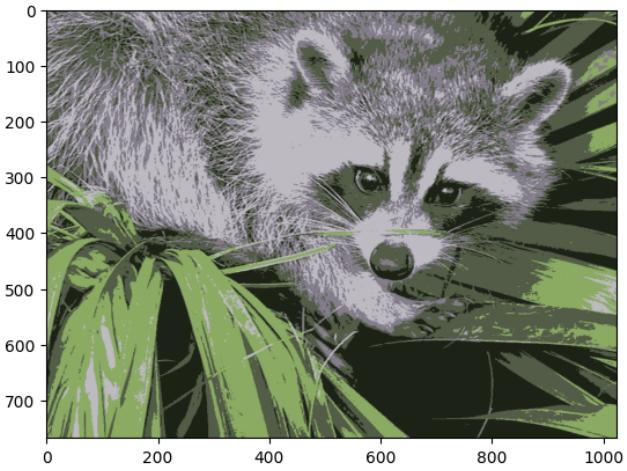

Figure 3: Result of PySpark implementation with 5 clusters and 10 iterations

The code is further optimized by grouping pixels with the same RGB values in order to reduce the size of the RDD inputted through the clustering algorithm. Pixels that are adjacent to each other in the image usually share the same colors. Hence, reducing the initial RDD of RGB values by the unique RGB values while tracking the number of pixels with the same RGB values can significantly reduce the size of the RDD passed through the clustering algorithm. For this specific image of a raccoon used throughout this paper, it was able to reduce the size of the problem to about 1/8, or around 100000 unique RBG values. A snippet of code from Appendix A.2 for the main iterative section can be seen below.

```
def map_pixels(pixel):
    p = pixel[0]
    cmeans = cluster_means.value
    closest = k.value
    min_diff = 256**2 * 3
    for i in range(k.value):
      norm = 0
      for j in range(len(p)):
        norm += (cmeans[i][j] - p[j])**2
      if norm < min_diff:
        closest = i
        min_diff = norm
    return (closest, pixel)

for i in range(num_iter):
iter_time = time.time()
old_rgb = avg_rgb
avg_rgb = pixels_grouped
    .map(map_pixels)
    .map(
        lambda x: (x[0],
            (tuple(
                [c * x[1][1] for c in x[1][0]]
            ), x[1][1]
            )
        )
    )
    .reduceByKey(
        lambda x,y: (
            [sum(z) for z in zip(x[0],y[0])],
            x[1] + y[1]
        )
    )
    .map(
        lambda x:
            tuple([z/x[1][1] for z in x[1][0]])
    )
    .cache()
cluster_means = sc
    .broadcast(avg_rgb.collect())
if old_rgb:
  old_rgb.unpersist()
```

The PySpark implementation replaces the second layer of for-loop of the NumPy implementation with Spark functions. After flattening the image into an RDD of RGB values, a helper function *map_pixels* is mapped to each pixel in the RDD to compute distances to each cluster center and then identify the closest cluster center. Note that there are no NumPy functions or arrays used here to avoid potential conflicts when NumPy and PySpark are used together. Then, the RDD is reduced by the closest cluster center for each pixel and average RGB values for each cluster are calculated. These new cluster centers are then broadcasted into the next iteration of the algorithm. Once the iterations are complete, the final cluster centers are collected. Finally, the original inputted image is stored as a matrix, and the closest cluster center for each pixel is calculated so as to not lose the spatial information of each pixel.

This solution attempts to apply distributed computing to the most computationally intensive part of the algorithm. As the size of the problem grows with the size of the image or the number of clusters, this implementation can also scale by adding more workers to

parallelize the computation. However, there exists the initial overhead cost of using the Spark framework such as parallelizing initial data to RDD, converting back RDD to arrays, and the interaction between Python and Spark.

2.3 PySpark ML Package Implementation

PySpark has ml.clustering package which has a fully implemented K-means function. After converting an image into a Spark dataframe, it can be used to fit a K-means model with a certain number of clusters to generate optimal cluster means.

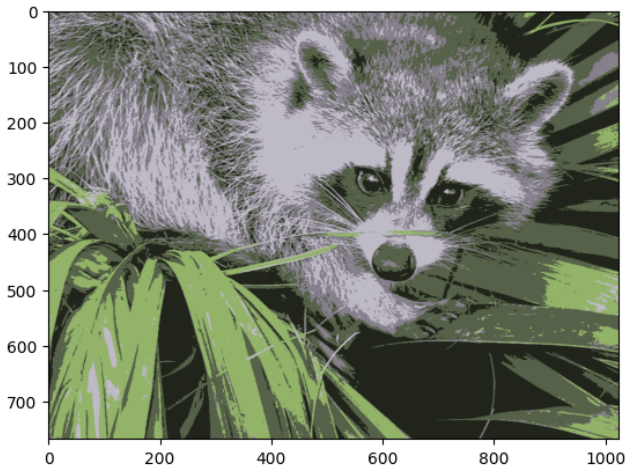

Figure 4: Result of PySpark ML implementation with 5 clusters

A snippet of code from Appendix A.3 for the main iterative section can be seen below.

```
img_rdd = sc.parallelize(img)
    .flatMap(
        lambda x: [px.tolist() for px in x]
    )
df = spark
    .createDataFrame(
        img_rdd,
        schema = ["r","g","b"]
    )
vecAssembler = VectorAssembler(
    inputCols=["r", "g", "b"],
    outputCol="features"
)
input_df = vecAssembler.transform(df)
kmeans = kmeans(k=num_clusters, seed=1)
model = kmeans
    .fit(input_df.select("features"))
transformed = model.transform(input_df)
cluster_means = transformed
    .select("r","g","b","prediction")
    .groupBy("prediction")
    .avg("r","g","b")
    .toPandas().to_NumPy()
```

Converting an image into a Spark dataframe loses the spatial information of pixels in the process, as each row of the dataframe is a pixel and each column represents the intensity levels of RGB. Similar to the previous PySpark implementation, once the final cluster centers are generated, the inputted image stored as a matrix is used to calculate the closest cluster center for each pixel without losing the spatial information of each pixel.

This solution uses distributed computing and does it in a much more efficient manner. Note that the built-in K-means function does not take in the number of iterations as a parameter. This means that the algorithm iterates until the cluster centers converge. It also has drastically reduced overhead costs due to the optimization done behind the scenes as a part of PySpark's ML package.

3 PERFORMANCE COMPARISON

The performance of each implementation is measured by the runtime of the functions for specified values of parameters such as the number of clusters and iterations. Different values for the number of clusters and iterations are used to identify trends.

Table 1 shows the performance of NumPy implementation. The runtimes of NumPy implementation can be seen to increase rapidly as both the number of iterations and the number of clusters increases, demonstrating its lack of scalability. However, it does very well with a low number of iterations and a low number of clusters as there is no overhead cost that results from using the Spark framework.

Table 2 shows the performance of PySpark implementation done by me. The runtimes of my own implementation appear much greater than the NumPy implementation which does not seem intuitive. There are a few explanations for this. First, the programming was done on Google Colab using the similar setup as the assignments, where the spark context is only setup locally without access to a remote distributed computing cluster. Even though the whole point of PySpark is distributed computing, there was no distribution of computing done due to the resources that are currently accessible to me. Hence, the performance of this implementation should be considered as being done on a single machine instead of multiple which would be the case in the ideal scenario. Second, there are overhead costs associated with using the Spark framework in Python that requires constant API calls back and forth. The fact that no distribution of computations could be done means that the PySpark implementation got the bad part of Spark with not many of the good parts. Lastly, the initialization of the algorithm is expensive. Randomly choosing k initial points means the entire RDD must be scanned. The function used here is PySpark's *takeSample* function, which needs to scan the entire RDD then randomly choose k different rows. The effect of this can be seen in Figure 5 which timed each iteration of the PySpark implementation.

We can observe in Figure 5 that the first iteration of the PySpark implementation takes almost 30 seconds to run, while the subsequent iterations take only 5 seconds on average. This is due to the initialization process being included in the first iteration, while the rest of the iterations purely deal with the main iterative part of the clustering algorithm.

Table 1: NumPy implementation runtimes (in seconds)

Number of clusters \ Number of iterations	1	5	10	20	50
2	0.3382	0.7623	1.6039	4.6291	7.6653
3	0.3047	1.4035	2.2722	4.3261	10.2454
5	0.3929	1.9001	3.6076	6.6482	14.6531
8	0.5585	2.6335	4.5622	8.7303	25.3359
10	0.7302	2.5217	5.4514	12.3100	26.0757

Table 2: PySpark implementation runtimes (in seconds)

Number of clusters \ Number of iterations	1	5	10	20	50
2	23.3564	30.8672	45.0538	72.2435	162.8648
3	20.8825	31.2942	46.0326	83.1631	177.8211
5	21.1502	38.8359	55.3694	94.4785	212.5548
8	32.1805	48.3324	71.7056	119.3261	263.1677
10	31.7725	54.0019	82.3575	135.0163	295.9264

```
iteration 0 begin
iteration 0 completed in:  28.772608518600464
iteration 1 begin
iteration 1 completed in:  4.553659915924072
iteration 2 begin
iteration 2 completed in:  4.601704120635986
iteration 3 begin
iteration 3 completed in:  6.174346685409546
iteration 4 begin
iteration 4 completed in:  4.573071718215942
iteration 5 begin
iteration 5 completed in:  5.756019115447998
iteration 6 begin
iteration 6 completed in:  6.698551177978516
iteration 7 begin
iteration 7 completed in:  4.487605094909668
iteration 8 begin
iteration 8 completed in:  5.482691049575806
```

Figure 5: Runtimes of first 8 iterations for a run of PySpark implementation with 10 clusters

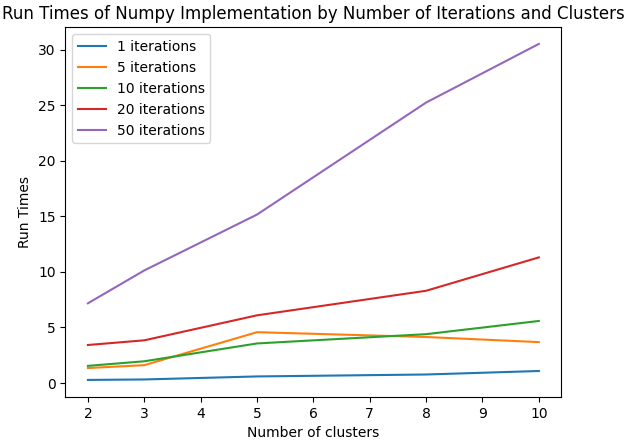

Figure 6: Runtimes of NumPy implementation

The long runtimes of my PySpark implementation can be reduced if there is access to additional workers for distributed computing and alternative methods to initialize cluster centers. The issue with initialization may also be solved with additional workers by having each worker randomly choose one RGB value as an initial cluster center so that the entire RDD does not have to be scanned by a single machine. Below are a few figures to better argue these points.

Figure 6 shows the runtimes of NumPy implementation and Figure 7 shows the runtimes of PySpark implementation. Comparing the graph of 50 iterations in the two figures, even though the magnitude of runtimes is significantly higher in the PySpark implementation across all number of clusters, the increase in runtimes appears steeper in the NumPy implementation as the number of clusters increases. We can further verify this trend by taking the ratio (NumPy runtime divided by PySpark runtime) between the runtimes of the two implementations shown in Figure 8.

Except for the graph corresponding to 5 iterations, Figure 8 shows a clear upward trend in the ratio of runtimes between NumPy and PySpark implementations with respect to both the number of iterations and the number of clusters. This means that, as the number of iterations or the number of clusters increases, the runtime of NumPy implementation approaches that of PySpark implementation. Extending this trend, it is reasonable to project that with a sufficiently large problem, the PySpark implementation will have similar runtime as the NumPy implementation. Considering that the

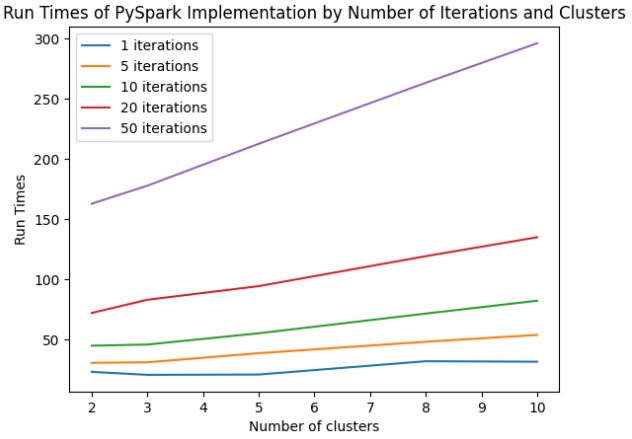

Figure 7: Runtimes of PySpark implementation

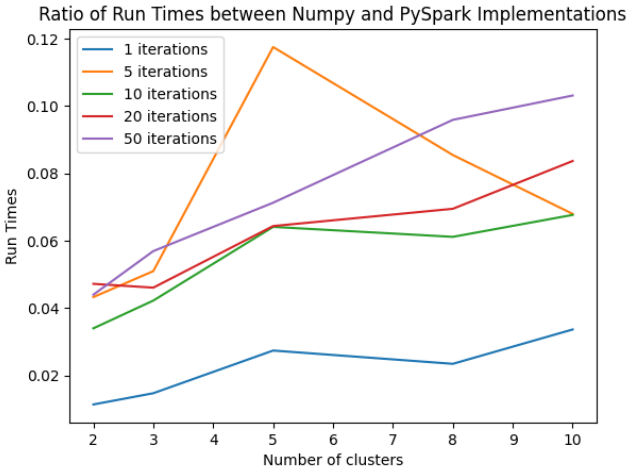

Figure 8: Ratio of runtimes between NumPy and PySpark implementations

current PySpark implementation does not have access to distributed systems to utilize the full benefits of distributed computing, it is very reasonable to project that PySpark implementation will easily outperform the NumPy implementation with a handful of workers and a sufficiently large number of clusters and iterations. That is, as the clustering problem grows in size, the PySpark implementation can scale where as the NumPy implementation cannot.

Table 3 shows the performance of the PySpark ML implementation. Note that this function does not take in the number of iterations as a parameter as it continues to iterate until the cluster means converge.

What is interesting to see here is that there is no obvious increase in runtime as the number of clusters increases. The ML package likely has very good optimization with low overhead cost due to less back-and-forth between Python and Spark. It also may be the case that, since the image is quite simple, there is no need for a large number of iterations. The cluster means of both NumPy and

Table 3: PySpark ML package implementation runtimes (in seconds)

Number of clusters	
2	20.2106
3	23.6867
5	19.4070
8	22.5226
10	21.4224

PySpark implementations may have converged well before their final iterations, but continued to iterate due to the use of for-loops. It should also be noted that the set up for the built-in K-means function uses a different spark session configuration which gives it access to a spark cluster, hence it is able to utilize distributed computing. Combining this fact with before, the lack of increase in runtimes as the number of clusters increase is likely due to the fact that the runtimes are mostly dominated by the initialization of the algorithm, and the iterative parts are well-distributed hence does not contribute meaningfully to the runtimes. Figure 9 shows the runtimes of all three implementations, where 10 iterations are used for both NumPy and PySpark implementations.

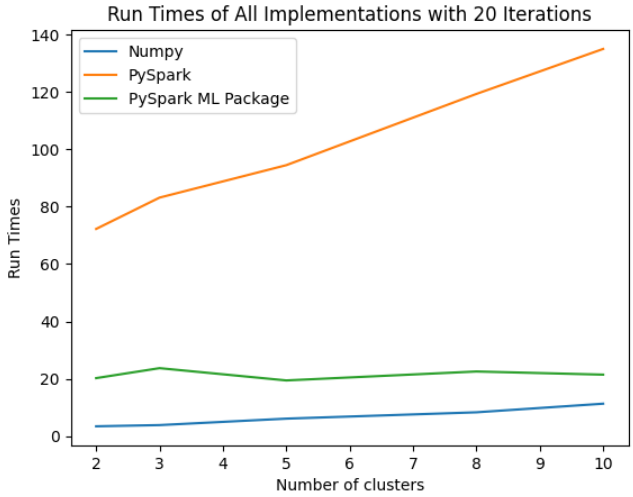

Figure 9: Runtimes of all 3 implementations with 20 iterations

We can see the effect of the overhead cost associated with Spark framework as even the ML package implementation has a slightly longer runtime than the NumPy implementation. However, the two runtimes do appear to converge as the number of clusters increases, and with a sufficiently large problem, the ML package implementation can scale whereas the NumPy implementation cannot.

4 CONCLUSION

In this paper, we studied the performance of 3 different implementations of the K-means clustering algorithm. First was the NumPy

implementation which utilized NumPy array functions to perform clustering. Second was the PySpark implementation written by me using materials learned in CS631. Third was the PySpark ML implementation using the fully implemented K-means function from a package.

The NumPy implementation showed good runtimes for small clustering problems with a low number of iterations and a low number of clusters, but its runtime quickly increased as those parameters increased. Hence, this solution is unscalable for a very large clustering problem that requires the processing of a large image with the identification of a large number of clusters over many iterations.

The PySpark implementation showed much greater runtimes than the NumPy implementation. This issue is attributable to three main reasons. One is the overhead costs associated with using Spark framework in Python. The other is the lack of distributed computing that can be performed on current spark context setup. The final reason is the inefficient initialization of cluster centers that results from using PySpark's *takeSample* function. However, even with the current complications, taking the ratio between the runtimes of the NumPy implementation and the PySpark implementation showed an increasing trend as the number of clusters and the number of iterations grew. That is, as the size of the clustering problem increases, the PySpark implementation scales better than the NumPy implementation. With access to additional workers for distributed computing, the PySpark implementation is projected to outperform the NumPy implementation for large scale clustering problems.

The PySpark ML implementation showed how good the PySpark implementation could be with extreme optimization. With this built-in K-means function, there are fewer overhead costs associated with using the Spark framework and access to distributed computing resources from a different spark session configuration. The effect of overhead cost is visible when comparing the PySpark ml implementation runtime on small clustering problems to that of the NumPy implementation, but the scalability of using the Spark framework can easily be observed as the number of clusters increases.

A PYTHON CODE

Python code for K-means RGB clustering used in this paper. Indentations have been modified to accommodate for the format of the report. Refer to https://git.uwaterloo.ca/kmshim/bigdata2023w/-/blob/main/CS631_Final_clean.ipynb for the full source code with detailed comments.

A.1 NumPy implemenetation

```
def kmeans_numpy(img, num_clusters, num_iter = 10):
  numrows = img.shape[0]
  numcols = img.shape[1]
  k = num_clusters
  no_label = num_clusters

  means = np.zeros((k, 3),'d')
  poolX = range(numcols)
  poolY = range(numrows)

  init_pixels = np.array(
    [np.random.choice(poolX, k),
     np.random.choice(poolY, k)]
  ).T

  for label in range(k):
        means[label,:3] = img[
            init_pixels[label,1],
            init_pixels[label,0],:3
        ]

  labels = np.full(
    (numrows, numcols),
    fill_value=no_label
  )

  for i in range(num_iter):
    shape = (numrows,numcols)
    opt_labels = np.full(
        shape,
        fill_value=no_label
    )

    min_dist = np.full(
        shape,
        fill_value=np.inf
    )

    col_coord, row_coord = np.meshgrid(
        range(numcols),
        range(numrows)
    )

    for label in range(k):
        full_dist = np.sum(
            np.square(
                means[label,:3]-img.astype(float)
            ),
            axis=2
        )
        min_dist = np.where(
            min_dist < full_dist,
            min_dist,
            full_dist
        )
        opt_labels = np.where(
            min_dist < full_dist,
            opt_labels,
            label
        )
```

```
    labels = opt_labels

    col_coord, row_coord = np.meshgrid(
        range(numcols),
        range(numrows)
    )

    for label in range(k):
        if label not in labels:
            means[label,:] = np.infty
        else:
            means[label, :3] = np.mean(
                img[np.where(labels==label)],
                axis=0
            )

  new_img = np.zeros(img.shape)
  for label in range(k):
    new_img[labels == label] = np.mean(
        img[np.where(labels == label)],
        axis=0
    )
  return new_img.astype(int)
```

A.2 PySpark implementation

```
def kmeans_spark(img, num_clusters, num_iter):
  k = sc.broadcast(num_clusters)
  pixels = sc
    .parallelize(img)
    .flatMap(
        lambda x: [tuple(px) for px in x]
    )
  cluster_means = sc
    .broadcast(
        np.array(
            pixels.takeSample(False, k.value, 0)
        )
    )

  pixels_grouped = pixels
    .map(lambda x: (x, 1))
    .reduceByKey(lambda x, y: x+y)
    .cache()

  def map_pixels(pixel):
    p = pixel[0]
    cmeans = cluster_means.value
    closest = k.value
    min_diff = 256**2 * 3
    for i in range(k.value):
      norm = 0
      for j in range(len(p)):
        norm += (cmeans[i][j] - p[j])**2
      if norm < min_diff:
        closest = i
        min_diff = norm
    return (closest, pixel)

  start_time = time.time()
  avg_rgb = None
  for i in range(num_iter):
    iter_time = time.time()
    old_rgb = avg_rgb
    avg_rgb = pixels_grouped
        .map(map_pixels)
        .map(
            lambda x: (x[0],
                (tuple(
                    [c * x[1][1] for c in x[1][0]]
                ), x[1][1]
                )
            )
        )
        .reduceByKey(
            lambda x,y: (
                [sum(z) for z in zip(x[0],y[0])],
                x[1] + y[1]
            )
        )
        .map(
            lambda x:
                tuple([z/x[1][1] for z in x[1][0]])
        )
        .cache()
    cluster_means = sc
        .broadcast(avg_rgb.collect())
    if old_rgb:
      old_rgb.unpersist()


  final_means = np.array(
    [list(x) for x in cluster_means.value]
  )
  opt_labels = np.full(
    (img.shape[0],img.shape[1]),
    fill_value=num_clusters
  )
  min_dist = np.full(
    (img.shape[0],img.shape[1]),
    fill_value=np.inf
  )
  for label in range(num_clusters):
```

```
      rgb_dist = np.sum(
        np.square(
            final_means[label,]-img.astype(float)
        ),
        axis=2
      )
      min_dist = np.where(
        min_dist < rgb_dist,
        min_dist,
        rgb_dist
      )
      opt_labels = np.where(
        min_dist < rgb_dist,
        opt_labels,
        label
      )

  new_img = np.zeros(img.shape)
  for label in range(num_clusters):
    new_img[opt_labels == label] = final_means[label,]
  return new_img.astype(int)
```

A.3 PySpark ML package implementation

```
def kmeans_spark_ml(img, num_clusters):
  img_rdd = sc.parallelize(img)
    .flatMap(
        lambda x: [px.tolist() for px in x]
    )
  df = spark
    .createDataFrame(
        img_rdd,
        schema = ["r","g","b"]
    )
  vecAssembler = VectorAssembler(
    inputCols=["r", "g", "b"],
    outputCol="features"
  )
  input_df = vecAssembler.transform(df)
  kmeans = kmeans(k=num_clusters, seed=1)
  model = kmeans
    .fit(input_df.select("features"))
  transformed = model.transform(input_df)
  cluster_means = transformed
    .select("r","g","b","prediction")
    .groupBy("prediction")
    .avg("r","g","b")
    .toPandas().to_NumPy()

  final_means = cluster_means[:,1:]
  opt_labels = np.full(
    (img.shape[0],img.shape[1]),
    fill_value=num_clusters
  )
  min_dist = np.full(
    (img.shape[0],img.shape[1]),
    fill_value=np.inf
  )
  for label in range(num_clusters):
      rgb_dist = np.sum(
        np.square(
            final_means[label,]-img.astype(float)
        ),
        axis=2
      )
      min_dist = np.where(
        min_dist < rgb_dist,
        min_dist,
        rgb_dist
      )
      opt_labels = np.where(
        min_dist < rgb_dist,
        opt_labels,
        label
      )

  new_img = np.zeros(img.shape)
  for label in range(num_clusters):
    new_img[opt_labels == label] = final_means[label,]
  return new_img.astype(int)
```