Evaluating Spark on Azure and AWS by K-Means Algorithm

Andrew Zhou
Department of Data Science
University of Washington
Seattle, WA 98195
ajz55@uw.edu

1 Introduction

K-Means is a unsupervised clustering machine learning algorithm that is widely used today. The goal is to partition the data set into k clusters by the similarity of the data points. The algorithm uses L^2 distance to find similar points and cluster the points that are close together. The algorithm uses the distance from every point to its centroid as its cost function, and the goal is to try and reduce the cost as close to 0 as possible. Although K-Means is a very popular algorithm, the running time of the algorithm is very slow. With the increasing competition in the cloud computing world, it would be beneficial to try out the competition and use the best cloud computing system.

In this paper, I will be running the K-Means algorithm on Azure and AWS spark systems. From my experiments, I found Azure's spark system to run faster than AWS across the board. The speed seems to be faster as the size of the data set gets larger. I will be analyzing the performance of the spark systems depending on both data size and number of workers. The K-Means algorithm will be modified in a way that will maximize fairness among the two systems and list out any differences among the systems.

2 Data set

To conduct the analysis for this project, I will be using large-scale astronomical imaging surveys found at Sloan Digital Sky Survey. The data is heavily numerical and ideal for clustering. The schema of the data is as follows:

Column	Value
ID	Unique Identifier for each source
X	Attribute for source (Float)
Y	Attribute for source (Float)
Z	Attribute for source (Float)
W	Attribute for source (Float)

This data set includes 15M sources of data which we will use the K-Means clustering algorithm to cluster the data points. This data set can be found as a CSV file in S3. The file can be loaded directly from S3 using Spark or loaded from a CSV file. For this data, we will be using k=7 centroids that are pre-selected for each run of K-Means.

3 Evaluated Systems

In this paper, I will be examining the running times of K-Means algorithm on Azure and AWS spark systems. Both systems are storage optimized and delta cache accelerated and running Spark 3.1.2. The following subsections will provide information on each system and the K-Means algorithm.

3.1 Azure

For Azure, the worker type chosen was "Standard_E4ds_v4" with 32 GB memory and 4 cores. The driver type chosen was the same as the worker type. This system has slightly higher memory than the AWS system.

3.2 AWS

For AWS, the worker type chosen was "i3.xlarge" with 30.5 GB memory and 4 cores. The driver type chosen was the same as the worker type. This system has slightly less memory than the Azure system chosen.

3.3 K-Means Algorithm

For this experiment, I will be running the K-Means clustering algorithm. The K-Means algorithm aims to partition data X to k clusters $P = \{P_1, ..., P_k\}$, by minimizing the sum of squared L^2 distances between every data point and the its centroid.

$$\min_{C,P} \Phi(C, P) = \min_{C,P} \sum_{i=1}^{k} \sum_{x \in P_i} ||x - c^{(i)}||_2^2$$

The cost function can be calculated below:

$$\phi(C) = min_P \Phi(C, P) = \sum_{x \in X} \min_{c \in C} ||x - c||_2^2$$

The algorithm for K-Means clustering can be found below:

Algorithm 1 k-means Algorithm

```
1: procedure K-MEANS
2:     Select k points as initial centroids of the k clusters.
3:     for iteration := 1 to MAX_ITER do
4:         for each point $x$ in the data set do
5:             Cluster of $x$ ← the cluster with the closest centroid to $x$
6:         end for
7:         for each cluster $P$ do
8:             Centroid of $P$ ← the mean of all the data points assigned to $P$
9:         end for
10:        Calculate the cost for this iteration.
11:    end for
12: end procedure
```

The code for the K-Means algorithm can be found in the Appendix 7. The code provided allows one to perform K-Means on a Spark RDD, which is unique to the K-Means function provided by PySpark. PySpark's implementation of K-Means must take a Spark DataFrame.

For this analysis, the K-Means algorithm will have a set of pre-chosen initial centroids to ensure that both systems will have the same initial centroids. By doing so, the calculations on both systems will be the same, and the clusters of the data set will be the same. k=7 will be run for all configurations. The stopping criterion used for this analysis is 20 iterations, to ensure that all K-Means must run for 20 iterations, no less or no more.

4 Problem Statement and Method

4.1 Problem Statement

The goal of this analysis was to determine if there were any difference in performance between Azure and AWS spark systems. K-Means algorithm is used to benchmark the two systems. Section 3 describes the systems and algorithm in detail and how it was implemented to maximize fairness among the systems. The main questions are listed below:

1. How does Azure and AWS spark compare when computing K-Means algorithm?
2. How does increasing the size of the data impact the running time?
3. How does increasing the number of workers impact the running time?

4.2 Method and Analysis Technique

The main metric I will be using for my analysis is time. By using K-Means to benchmark AWS and Azure, I will obtain the amount of time it ran in seconds. I will be testing Azure and AWS spark sessions with different sizes of data. The full data set has 15M data points. I plan on running the k-means algorithm with sizes 1M, 5M, 10M, 15M rows of data by sampling from the original 15M data points. I'll also be varying the number of worker clusters to see how that impacts Azure and AWS running times of the k-means cluster algorithm. The number of worker clusters that will be tested are: 2, 4, 8, and 16 workers.

5 Results

5.1 Running Times

AWS Running Time Results (seconds)				
Workers	Data Size (rows)			
	1 Million	5 Million	10 Million	15 Million
2	11.98s	48.97s	88.02s	121.85s
4	8.56s	22.09s	40.21s	56.67s
8	6.32s	14.28s	24.45s	42.42s
16	6.15s	10.57s	16.86s	18.97s

Table 1: AWS Running Time Table

AWS Running Time Results (seconds)				
Workers	Data Size (rows)			
	1 Million	5 Million	10 Million	15 Million
2	7.84s	27.03s	53.21s	98.53s
4	12.46s	16.43s	31.45s	41.99s
8	4.43s	11.02s	18.83s	31.44s
16	4.44s	8.27s	15.83s	15.99s

Table 2: Azure Running Time Table

The two tables above are the raw running time results for AWS and Azure. Each table displays the running times for the K-Means algorithm in seconds for 1M, 5M, 10M, 15M data points and 2, 4, 8, 16 workers. Table 1 are the results for AWS. Table 2 are the results for Azure.

5.2 Running Time by Data Size

Figure 1 displays the running times in seconds for all configurations. Figure 2 displays the running time separated by the number of workers.

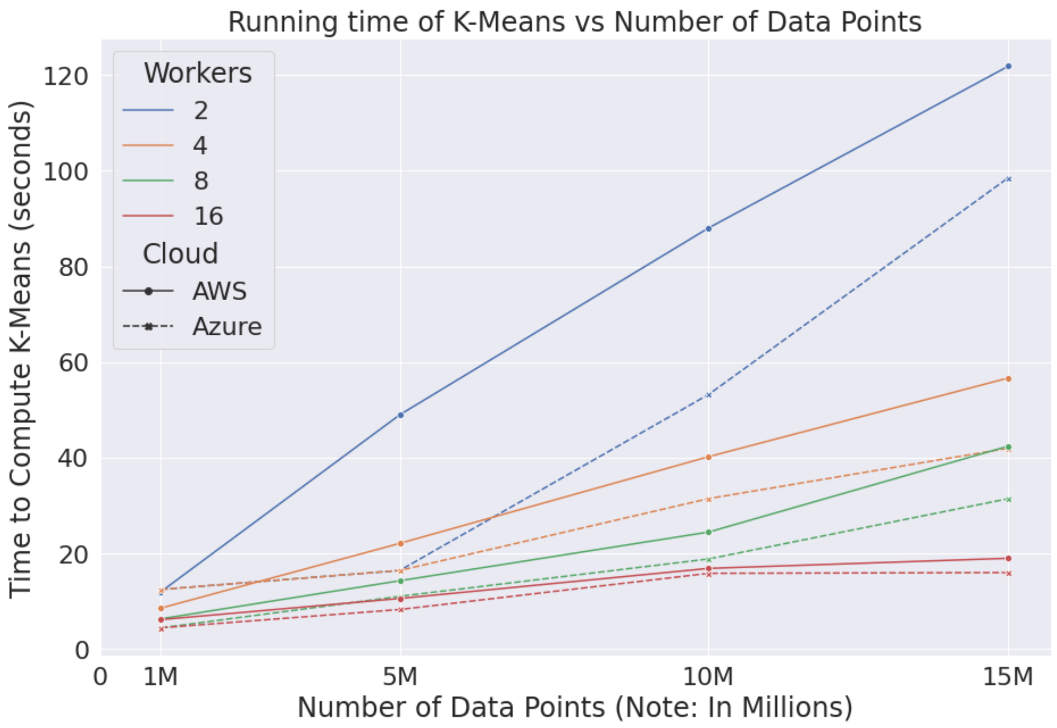
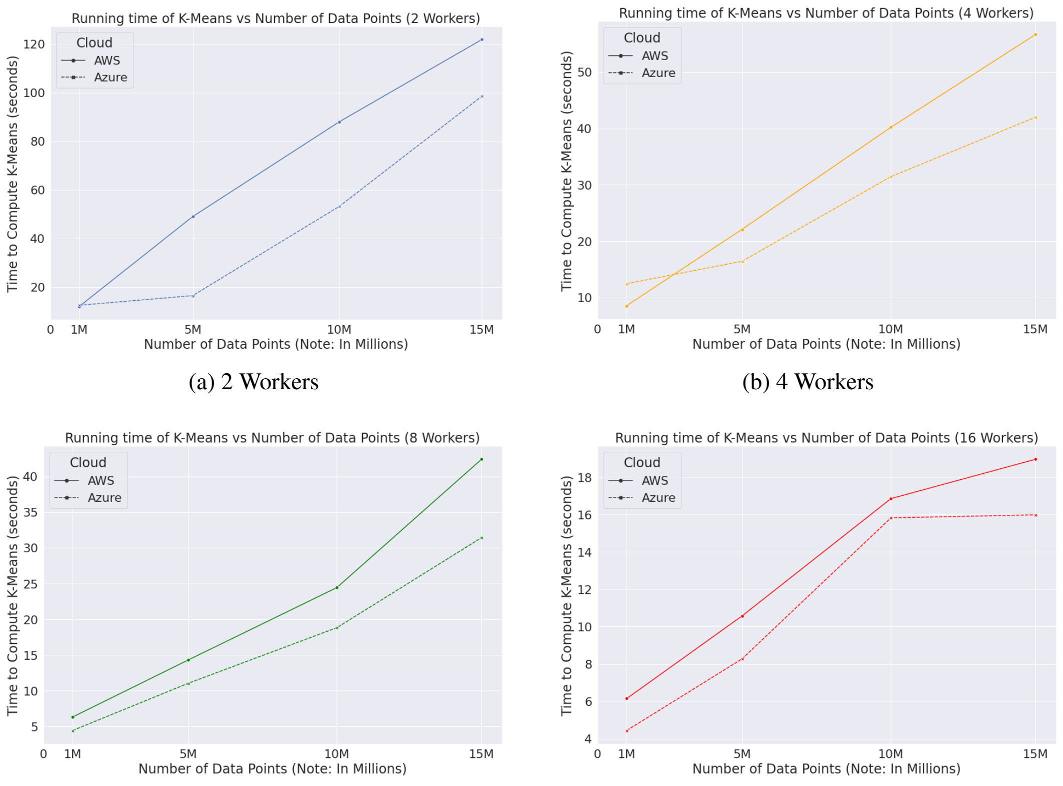


Figure 1: Running time of All Cloud systems and number of Workers



(a) 2 Workers

(b) 4 Workers

(c) 8 Workers

(d) 16 Workers

Figure 2: Individual running time by number of Workers

Through these plots we can see that time increases roughly linearly as the number of data points increases. The slope of each plot seems to be greater the less workers there are. With 16 workers. the slope is quite small in comparison with the slope of 2 workers. Azure almost always outperforms AWS in running time. The time difference between Azure and AWS seems to get bigger as the number of data points increases. Meaning, the more data points there are in the data set, Azure seems to perform even better in comparison with less data points.

5.3 Running Time by Number of Workers

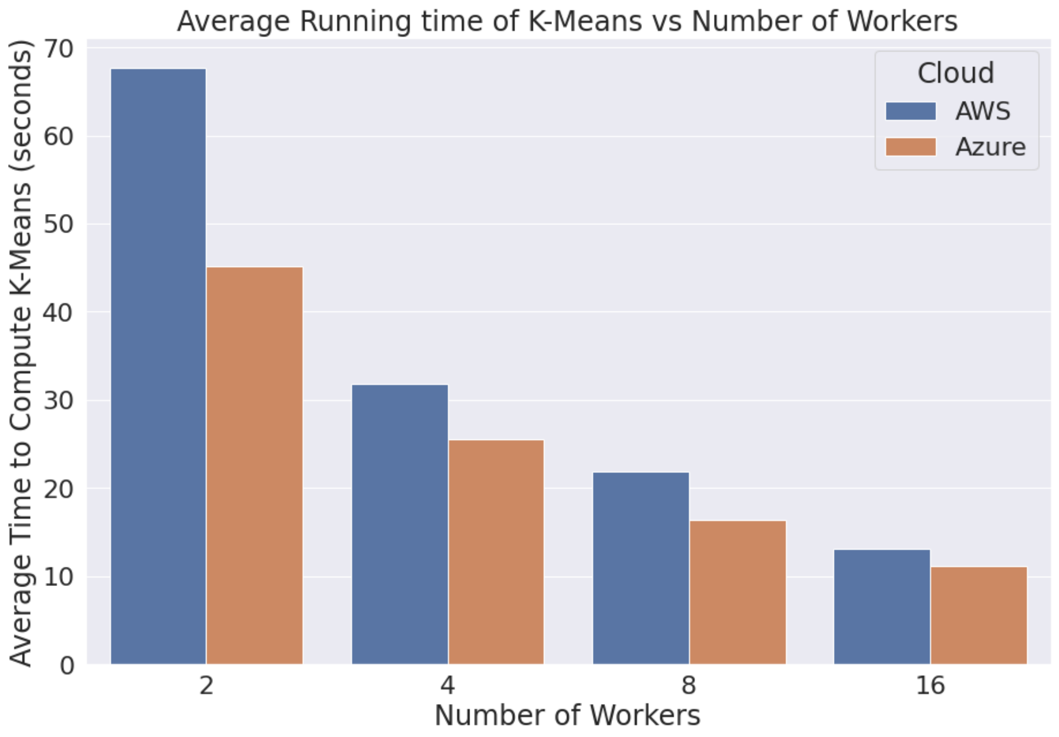


Figure 3: Average Running time of K-Means by Number of Workers

Figure 3 displays the average time to compute K-Means for 1M, 5M, 10M, 15M data points by the number of workers. Azure's average running time is shorter than AWS across all configurations of number of workers. However as the number of worker increases, AWS's running time performance approaches to very similar results produced by Azure. It can also be seen that as the number of workers increases, the running time seems to be decreasing at a exponential decay rate. That means every time we double the number of workers, the performance gain is less. There is more significant improvement in performance by changing number of workers from 2 to 4 than 8 to 16.

5.4 Summary

Through this analysis, it was found that Azure runs faster than AWS for almost all configurations. However, it is worthy to note that Azure's worker type had slightly more memory than AWS's worker type. Azure's edge in performance seems to drop off as the number of workers increases. That is AWS's performance approaches very similar performance to Azure as the number of workers increase. However, the general trend seems to indicate as the number of data increases, the difference in running time between Azure and AWS increases. This seems to indicate that Azure spark seems to scale better than AWS when it comes to bigger data.

From the results, the relationship between the number of data points and the running time seems linear. As the number of data points increases, running time also increases linearly. It is also worthy to note that the more workers there are, the smaller the slope. That means extra workers scale better in terms of large data and require less extra time for bigger data sets. The slope is extraordinarily big for 2 workers, we can see that the time increases roughly 40 seconds for each increased data size, while the slope for the 16 workers is only 7-10 seconds for each increased data size.

Increasing the number of workers decreases running time by a exponential decay. The trend seems to suggest that after a certain number of workers, the performance increase will be flattened out and not be worthwhile. Increasing the number of workers from a low number of workers seems to be very effective and can be very beneficial. The performance will likely reach an asymptotic value as the number of workers increase. The results also suggest that the larger the data set, the better a large number of workers scale. Which means a high number of workers is not likely to hit an asymptotic performance with a very large data set.

6 Conclusion

In conclusion, from this analysis it would seem beneficial to use Azure spark over AWS spark for machine learning. Azure won in performance for almost all configurations of K-Means. It is worthy to note that Azure had a slight edge with its worker type with 1.5 GB of extra memory compared to AWS. The relationship discovered between data size and computing time seems to be linear, with slope decreasing as number of workers increases. It is also found that computing time decreases at a exponential decay as number of workers increases and will likely hit an asymptotic value depending on the data size.

7 Appendix

```python
# Helper functions for the k-means algorithm
def l2_dist(pair):
    """
    Function used to calculate the l2 distance
    """
    d = (np.array(pair[0]) - np.array(pair[1]))
    return (pair, np.linalg.norm(d))

def filter_min(item):
    """
    Find the nearest cluster for a data point
    """
    key = item[0]
    value = item[1]
    min_val = value[0][1]
    new_val = []
    for i in value:
        if(i[1] == min_val):
            new_val.append(i)
    return (key, new_val)

def remap_for_centroid(pair):
    """
    Regroup data points into their clusters
    """
    key = pair[0]
    val = pair[1]
    new_key = val[0][0]
    new_val = []
    for i in val:
        if(i[0] == new_key):
            new_val.append((key, i[1]))
    return (new_key, new_val)

def get_mean_of_points(pair):
    """
    Calculate the new means for all the centroids
    """
    values = pair[1]
    val = []
    for i in values:
        val.append((i[0]))
    val = np.array(val)
    return (tuple(np.mean(val, axis=0)), values)

def get_total_cost(pair):
    """
    Calculates the cost function
    """
    cluster = pair[0]
    data_points = pair[1]
    total_cost = 0
    for x in data_points:
        point = x[0]
        d = np.linalg.norm(np.array(point) - np.array(cluster))**2
        total_cost += d
    return (cluster, total_cost)
```

```python
# K-means function
def k_means(data, initial, max_iters=20):
    """
    K-means algorithm for Spark RDD's. This function takes in an RDD and an initial
        centroid set.

    :param data: The datapoints as an RDD object.
    :param initial: The initial centroids to start with. Must also be an RDD object.
    :param max_iters: The stopping criterion for the k-means. By default it will
        stop in 20 iterations.
    :returns: The cost function at each iteration.
    """
    total_cost_by_iteration = []
    centroids = initial
    for i in range(max_iters):
        # find the cluster closest to x
        # Get the cartesian
        cartesian = data.cartesian(centroids)
        # Calculate the distance for every point in data to a centroid
        distance = cartesian.map(l2_dist).map(lambda p: (p[0][0], (p[0][1], p[1])))
        # Group the distances by the data point
        grouped = distance.groupByKey().mapValues(list).mapValues(lambda p:
            sorted(p, key=lambda tup: (tup[1])))
        # Get the shortest distance to cluster only
        grouped_reduced = grouped.map(filter_min)

        # Group into their clusters
        centroid_map = grouped_reduced.map(remap_for_centroid)
        centroid_group = centroid_map.groupByKey().mapValues(list).mapValues(lambda
            t: [item for sublist in t for item in sublist])

        # Set the new centroids
        new_centroid = centroid_group.map(get_mean_of_points)
        centroids = new_centroid.map(lambda p: p[0])

        # Calculate the cost
        centroid_cost = centroid_group.map(get_total_cost)
        total_cost = centroid_cost.values().sum()
        total_cost_by_iteration.append(total_cost)
    return total_cost_by_iteration
```