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

Today there are many competitors in the cloud computing world. One of the most popular analytical engines for big data is Spark. For my proposal, I would like to try the k-means algorithm on a predetermined set of centroids on Azure Spark and AWS Spark. My goal is to compare and contrast the running times of Spark on Azure and AWS. I will not be looking at k-means clustering performance, but the running time of the algorithm on different sizes of data and number of worker clusters.

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 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 classifier 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. If it fails to load, the csv file can also be imported locally and used directly.

3 Algorithms and Techniques

3.1 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 centroid of associated with the cluster.

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

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

```
procedure K-MEANS
    Select k points as initial centroids of the k clusters.
    for iteration := 1 to MAX_ITER do
        for each point x in the data set do
            Cluster of x ← the cluster with the closest centroid to x
        end for
        for each cluster P do
            Centroid of P ← the mean of all the data points assigned to P
        end for
        Calculate the cost for this iteration.
    end for
end procedure
```

3.2 Techniques

I will be testing Azure and AWS spark sessions with different sizes of data. The data set I will be using has 15M data points. I plan on running the k-means algorithm with sizes 1M, 5M, 10M, 15M. I'll also be varying the number of worker clusters to see how the number of clusters (perhaps of sizes: 2, 4, 8, 16) for Azure and AWS affect the running time of the k-means cluster algorithm.

4 Things I've Accomplished So Far

I wrote a k-means algorithm that takes in an RDD of the data set instead of a pyspark dataframe. The function also takes in an initial set of centroids to make sure that each k-means runs the exact same algorithm, as long as the same stopping criterion of 20 iterations. This is unique to the k-means provided by spark, as spark's k-means uses a pyspark dataframe. The code can be found below. I also finished running the k-means algorithm for all varying cluster sizes and data sizes. I have some initial plots generated from the running times below.

4.1 Things I need to do

I have much work left on the report. I need to restructure/reorganize the paper into a final report after this milestone. I need to add in an introduction and more details about the data set. Now that I have completed the running times, I will also include as much detail as possible of how I designed the k-means to be as equal as possible for both AWS and Azure for a fair comparison. I will also talk about the worker types that were selected in both systems. I also need to perform more analysis with the results to talk about and finish with a conclusion.

4.2 Results:

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

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

```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
```

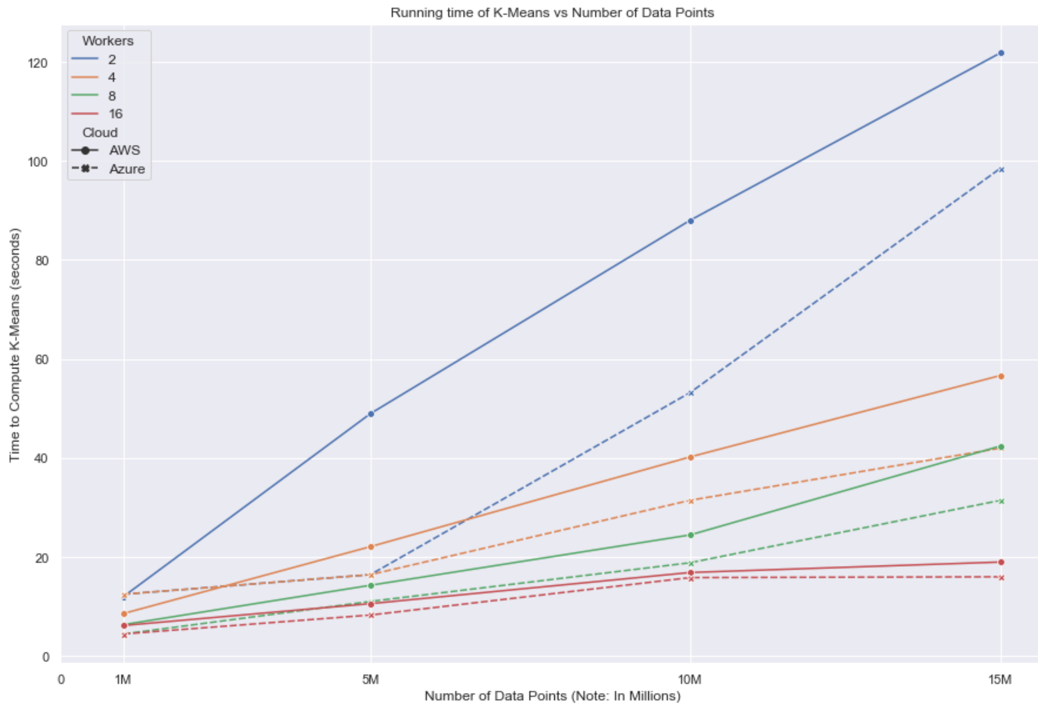
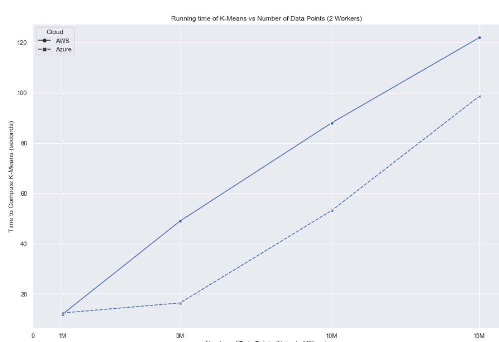
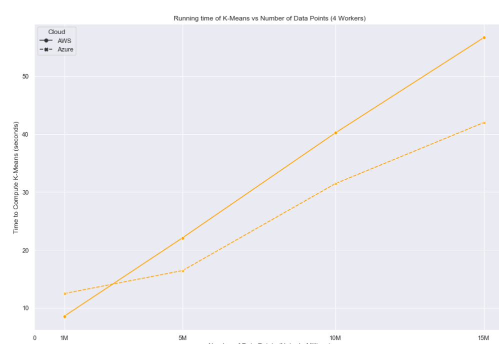


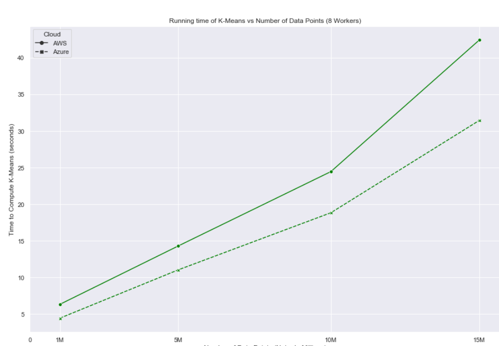
Figure 1: Running time of All Cloud systems and number of Workers



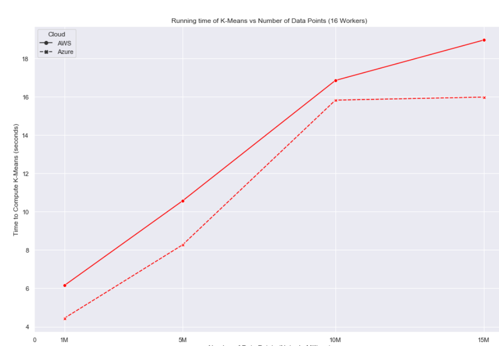
(a) 2 Workers



(b) 4 Workers



(c) 8 Workers



(d) 16 Workers

Figure 2: Individual running time by number of Workers