K-Means Clustering Algorithm on a Multi-Node Hadoop Cluster

Wajahat Waheed
ECE Dept.
Purdue University Northwest
Hammond, IN

Bruno Hnatusko III
ECE Dept.
Purdue University Northwest
Hammond, IN

David Olson
ECE Dept.
Purdue University Northwest
Hammond, IN

I. INTRODUCTION

The purpose of this project is to implement a K-Means Clustering algorithm on a cluster of virtual machines to compare the performance and results with that of a single machine. The other purpose is to allow for the showcasing of skills learned throughout the course, from the "Big Data" concepts of Hadoop to the more complex machine learning algorithms discussed later in the course.

II. BACKGROUND

A. *Amazon Web Services (AWS)*

It was decided early on that the best course of action was to host the cluster on Amazon's Web Services. This provides numerous advantages: collaboration was much easier than an individual hosting the cluster. Also, as students, AWS provides $100 of credit per student for free, making the project ultimately free. There are other options, such as Google's Cloud Platform (GCP), but Amazon was chosen because it was already familiar. Ultimately, this did not work, as will be discussed in the results.

B. *Hadoop*

Hadoop is a software package that allows for data storage across servers. Programs and calculations can be performed on the data stored using other programs, such as MapReduce. For this project, both the mapper and reducer were written in Python, as it works nicely with Hadoop.

C. *K-Means Clustering Algorithm*

III. METHODOLOGY

A. *Mapper*

Python code was written in Python in the form of a Mapper and Reducer. Firstly, we have mapper_wdb.py which simply gathers data for the K-means data points.

```
import sys
for line in sys.stdin:
    line=line.strip()
    coords=line.split(',')
    print('%s\t%s'%(coords[0],coords[1]))
```

B. *Reducer*

Secondly, we have reducer_wdb.py, which contains our k-means clustering algorithm. Data first is gathered and put into centroids and data points. Data points come from the mapper.

```
points = []
oldCent = []
newCent = []
clustCount = 0
clusters = []
iterCount = 0
threshold = 1

for line in sys.stdin:
   line = line.strip()
   (x,y) = line.split('\t')
   points.append((float(x),float(y)))
```

Centroids come from a text file containing random coordinates. All code was written with the intent of being scalable, so one can have any number of centroids and clusters.

```
dir_path=os.path.dirname(os.path.realpath(_
_file__))
centFile = os.path.join(dir_path,
'centroids.txt')
f = open(centFile)

for line in f:
    line = line.strip()
    (x,y) = line.split(',')
    newCent.append((float(x),float(y)))
clustCount = len(newCent)
```

Next is the algorithm, which follows a simple pattern: form clusters out of data points, update centroids to make them the means of the new clusters' data points, check if centroids have reached convergence, and repeat the process if the centroids have not converged.
The first part of the algorithm is forming clusters out of data points. Each cluster coincides with a centroid. Each point is assigned to the cluster that corresponds to the centroid closest to that point.

```
def UpdateClusters():
    global points
```

```
        global oldCent
        global clusters
        clusters = [[]
for c in range(clustCount)]
            for point in points:
                minDist = float("inf")
                chosenCluster = -1
                for i in range(clustCount):
--The following code is further indented--
centroid = oldCent[i]
xDif = point[0] - centroid[0]
yDif = point[1] - centroid[1]
thisDist = math.sqrt(math.pow(xDif,2) +
math.pow(yDif,2))
if thisDist < minDist:
chosenCluster = i
minDist = thisDist
---------Further indentation ends---------
        clusters[chosenCluster].append
(point)
```

Second is updating each centroid. The sums of the x and y coordinates of all points in a cluster are found, then divided by size (or length in this case) of the cluster to give the new mean the centroid shall become.

```
def UpdateCentroids():
    global clusters
    global newCent
    for i in range(clustCount):
        (sumX,sumY) = (0,0)
        cluster = clusters[i]
        count = len(cluster)
        for point in cluster:
            (sumX,sumY) = (sumX +
point[0],sumY + point[1])
            if count > 0:
                newCent[i] =
(sumX/count,sumY/count)
```

Third is comparing the centroids with what they were before the current iteration of the k-means algorithm.

```
def centroidsConverged():
    global oldCent
    global newCent
    for thisOld, thisNew in
zip(oldCent,newCent):
        difX = abs(thisNew[0] - thisOld[0])
        difY = abs(thisNew[1] - thisOld[1])
        if (difX > threshold) or (difY >
threshold):
            return False
    return True
```

Finally, we have the main k_means function that uses the previous three functions and eventually produces output in the form of the finalized centroids.

```
def K_means():
    global oldCent
    global newCent
    oldCent = list(newCent)
    UpdateClusters()
    UpdateCentroids()
    converged = centroidsConverged()
    if converged:
        for centroid in newCent:
            print('%s\t%s'
%(centroid[0],centroid[1]))
    else:
        K_means()
K_means()
```

C. Altered reducer for graphing

In addition, an alternate version of this program was written to print the centroids and clusters before the first cluster update and at the end of each iteration. This was written so we could form graphs plotting the clusters and centroids, which is done in yet another program.

First we have reducer_for_graphing_wdb.py. For brevity, only the new and changed code will be shown here. First, we have an output folder created at the start.

```
outputFolderName = "k_means_output"
output_path =
os.path.join(dir_path,outputFolderName)

if os.path.exists(output_path):
    shutil.rmtree(output_path)
os.mkdir(output_path)
```

Next, we have the output function that creates a txt file containing centroids, clusters, and delimiters separating them.

```
def writeOutput(iterCount):
    fileName = 'output_%s.txt'%iterCount
    centroids = oldCent
    if iterCount == 0:
        fileName = 'beginning.txt'
        centroids = newCent
    filePath = os.path.join(output_path,
fileName)
    newFile = open(filePath,"a")

    for centroid in centroids:
        newFile.write('%s\t%s\n'
%(centroid[0],centroid[1]))
    for cluster in clusters:
        newFile.write('<Cluster Follows>\n')
        for point in cluster:
            newFile.write('%s\t%s\n'
%(point[0],point[1]))
```

Lastly, function calls for writeOutput function were put in the k_means function.

```
def K_means(iterCount):
    global oldCent
    global newCent
    oldCent = list(newCent)
    UpdateClusters()
    if iterCount == 0:
        writeOutput(iterCount)
    iterCount = iterCount + 1
    UpdateCentroids()
    writeOutput(iterCount)
    converged = centroidsConverged()
    if converged:
        for centroid in newCent:
            print('%s\t%s'
%(centroid[0],centroid[1]))
    else:
        K_means(iterCount)
K_means(0)
```

D. Grapher

Finally, we have grapher_wdb.py. This program is meant to take one of the txt files produced by the previous program and create a plot of the data.

First we have the variables declared in the beginning:

```
centroids = []
clusters = []
currentCluster = -1
clusterColors = []
centroidColors = []
```

Then we gather data from the input file:

```
import sys
for line in sys.stdin:
    line = line.strip()
    if line == '<Cluster Follows>':
        currentCluster = currentCluster + 1
        clusters.append([])
        else:
            (x,y) = line.split('\t')
            point = ((float(x),float(y)))
            if currentCluster == -1:
                centroids.append(point)
            else:
                clusters[currentCluster]
.append(point)
                clustCount = len(clusters)
```

Next are functions used at the end. First is a color generation function that is later used to create a unique color for each cluster and centroid. Used for cluster i in the cluster list of length n, it finds a color along the spectrum.

```
def makeColor(i,n):
    c = 3*float(i)/n
    prim = int(c)
    sec = (prim+1)%3
    u = math.floor(c/0.5)
    v = c - (0.5 * u)
    primV = 1
    secV = 2*v
    if (c % 1 >= 0.5):
        primV = 1-secV
        secV = 1
    color = [0,0,0]
    color[prim] = primV
    color[sec] = secV
    return color
```

In order to keep centroids the same hue as their clusters while also standing out, the following functions were written. The given factor parameters make the plot points easy to distinguish.

```
def getLighterColor(color,factor):
    newColor = list(color)
    for i in range(3):
        newColor[i] += factor *
        float(1-newColor[i])/2
    return newColor

def getDarkerColor(color,factor):
    newColor = list(color)
    for i in range(3):
        newColor[i] -= factor *
        float(newColor[i])/2
     return newColor
```

Colors are generated for each cluster and centroid.

```
for i in range(clustCount):
    currentColor = makeColor(i,clustCount)
    clusterColors.append
    (getLighterColor(currentColor,0.5))  `
    centroidColors.append
    (getDarkerColor(currentColor,1))
```

Finally, a scatter plot is rendered.

```
increment = 100
for i in range(clustCount):
    pointCount = len(clusters[i])
    currentPoint = 0
    for point in clusters[i][0::increment]:
        (pointX,pointY) = point
        plt.scatter(pointX,pointY,
        color=clusterColors[i])
        currentPoint = currentPoint +
        increment
        #print('Point %s/%s has been
plotted.'%(currentPoint,pointCount))
        #Uncomment the above line if you
want the program to show you progress.

for i in range(clustCount):
    (centX,centY) = centroids[i]
    plt.scatter(centX,centY,
    color=centroidColors[i])

plt.show()
```

IV. Results

A. Running in the cluster

We were unable to successfully run a Hadoop cluster on AWS. It is believed that the issues revolved around networking. In the end, a cluster was created using a series of 4 virtual machines on a single (local) host, not in AWS. A basic tutorial was followed, with the software setup consisting of Hadoop and YARN [1]. These were Ubuntu based, each with 2 vCPUs and 8GB RAM. The host had an i7-10700K, 32GB RAM, and NVMe flash storage. One virtual machine was dedicated to being the namenode, with the three others acting as data nodes.

The “screenshots” folder we provided contains screenshots showing proof of our local cluster and its usage.

B. K-means and Graphing

Initial centroids used in this experiment:

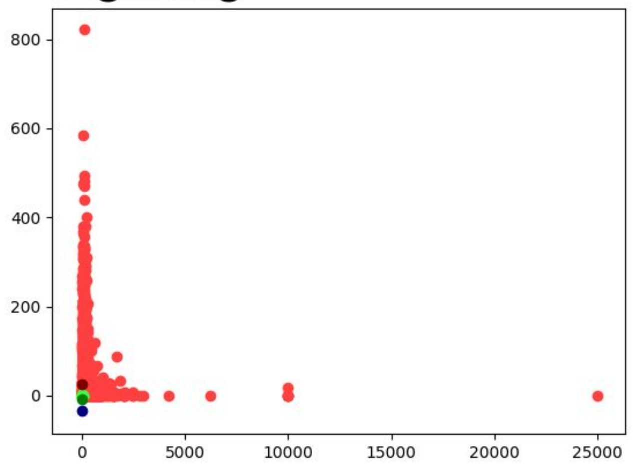
X	Y
9.99997035758	24.9999516687
1.9999175124	-9.00020832026
-15.000500407	-34.9999749295

Final centroids after iteration 25:

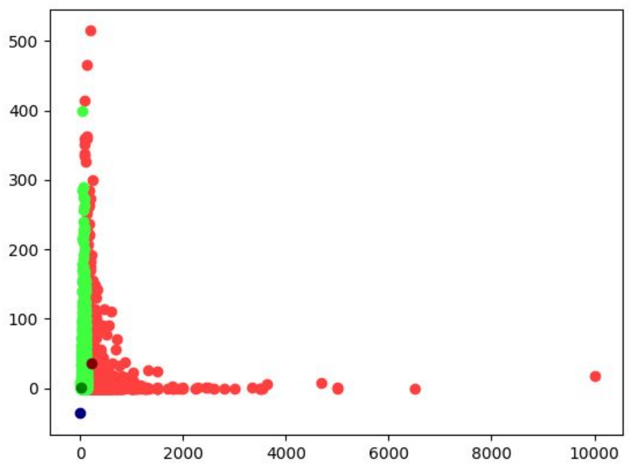
X	Y
10674.9773869	8.77135678392
1693.92626087	6.74330434783
162.240701831	35.2791315342

We have plots of the beginning, every 2^n iteration, and the final (25th) iteration, for when we used airbnb_edited.txt and centroids.txt with a convergence threshold of 1.
Note that given the sheer volume of data, we had the grapher plot every hundredth point in each cluster. Thus the plots are not identical. Secondly, the ratio between the highest x value and highest y value makes the points appear much horizontally closer than they truly are. Finally, each cluster’s points are put over the previous clusters’ points when they overlap (blue on green on red).

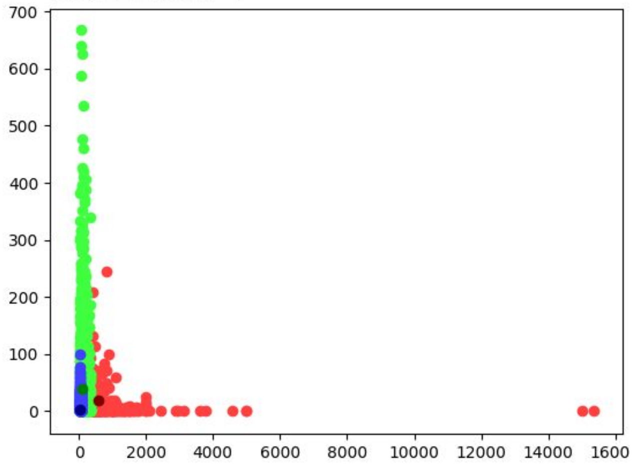
Beginning



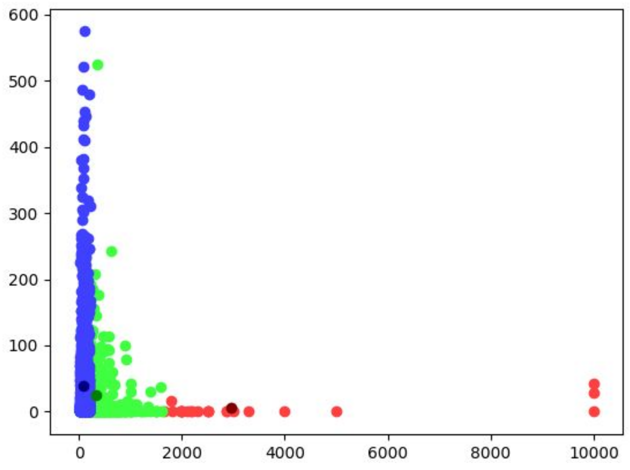
Iteration 2

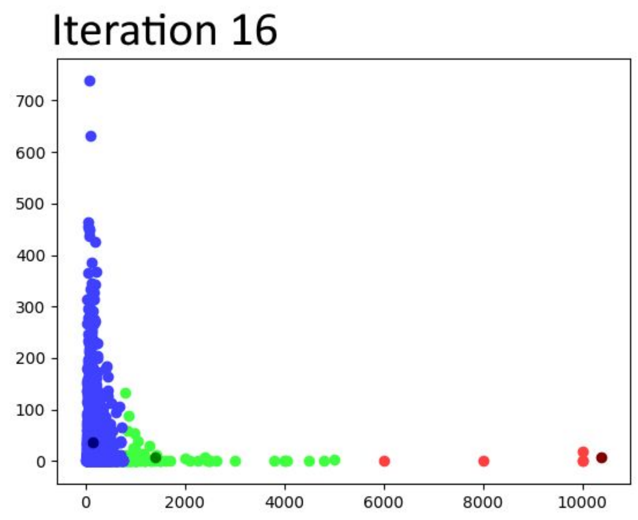


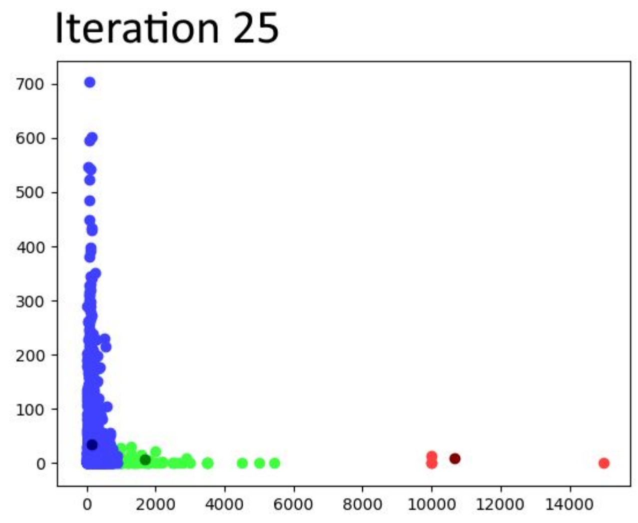
Iteration 4



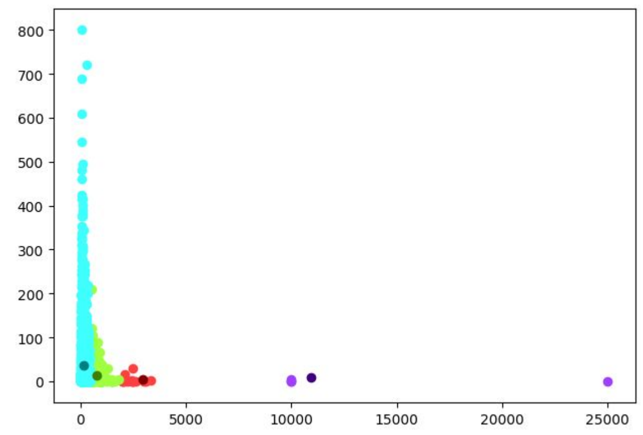
Iteration 8







Lastly, the code's scalability was put to the test. A fourth centroid was added to centroids.txt, and the code handled it perfectly, as seen in the following plot.



C. *Threshold and iterations*

We tried increasing the threshold for centroid convergence to see how it impacted code performance. We learned that by doing so, we can trade accuracy for speed.

Threshold	Iterations
1	25
50	18
100	16
200	2
300	1

Interestingly, the iterations taken jumped from 2 at threshold 138 to 16 at threshold 137. Closer inspection revealed that the first centroid was the cause.
Convergence checks for the first centroid:

Iteration	xDif	yDif
1	213.038725062	10.0949617088
2	137.107695363	2.1652361858
3	235.659453939	17.976071129

Two conclusions are drawn from this. Firstly, by increasing the convergence threshold, we can increase speed at the cost of accuracy. Secondly, speed also depends on the data itself, so performance of the k-means algorithm ultimately is not entirely controllable.

V. Conclusion

The costs that come with using an online web service for a cluster combined with the lack of a replicable set of instructions for setting up a multi-node cluster using a webservice posed problems for our project. Even the tutorial we followed implies in Step 35 that the process is not replicable, or at least, that the tutorial itself does not outline the process sufficiently[1].
Addressing these problems is vital when the Big Data course is revised for future students. Perhaps outlining every single step one takes to set up a multi-node cluster on AWS could be a project option, as that needs to be replicable by students before one even thinks about using such a cluster for any purpose.
Beyond that, setting up a local cluster went better and finding data and writing a K-means algorithm to work with that data was successful.

Contributions: **Wajahat** – wrote 1st version of code, helped in debugging both code and cluster, worked on report. **Bruno** – wrote final code, worked on AWS cluster, worked on report. **David** – found/processed data sets, created local cluster, worked on report.

VI. References

[1] https://medium.com/@jootorres_11979/how-to-set-up-a-hadoop-3-2-1-multi-node-cluster-on-ubuntu-18-04-2-nodes-567ca44a3b12