Big Data Analysis of Chicago Crime Records

Maksyk Vladyslav, Hafiz Rayaan Shahid
vladmaksyk@gmail.com, rayaan.shahid.rs@gmail.com
University of Stavanger, Stavanger, Norway

Abstract. The purpose of this project is to understand and implement MapReduce programming model on a Hadoop cluster. MapReduce is a framework that performs resource intensive tasks in a parallel manner by distributing the computations on large number of nodes. In this study, we examine the applicability of MapReduce in clustering. The goal of this study is to perform k-means clustering using Hadoop MapReduce.

1 Introduction

The dataset reflects reported incidents of crime that occurred in the City of Chicago from 2001 to 2017. Data is extracted from the Chicago Police Department's CLEAR (Citizen Law Enforcement Analysis and Reporting) system. In order to protect the privacy of crime victims, addresses are shown at the block level only and specific locations are not identified. [1]

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster. [2] The MapReduce algorithm contains two important tasks, namely Map and Reduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job. [3] So MapReduce is considered as one of the fastest ways of processing huge data in the Data intensive approach. This approach can be performed in a Cluster environment of Hadoop and get desired results out of it.

1.1 Overview

The Chicago Crimes dataset consists of about 8 million records. In our project we use this dataset to implement three MapReduce algorithms for analysis:

- K-means clustering
- Finding periods with the highest and the lowest amount of crimes for every year.

- Finding particular type of crime which annually influences the drop of crimes the most. Finding the month of the year which was the most successful in reducing the amount of drug crimes.

2 Dataset Overview

For our project we used dataset that is available on: https://www.kaggle.com/currie32/crimes-in-chicago. The csv file consists of twenty-two columns.

- **ID**: Unique identifier for the record.
- **Case Number**: The Chicago Police Department RD Number (Records Division Number), which is unique to the incident.
- **Date:** Date when the incident occurred. this is sometimes a best estimate.
- **Block:** The partially redacted address where the incident occurred, placing it on the same block as the actual address.
- **IUCR:** The Illinois Unifrom Crime Reporting code. This is directly linked to the Primary Type and Description.
- **Primary Type:** The primary description of recorded crime.
- **Description:** The secondary description of the recorded crime, a subcategory of the primary description.
- **Location Description:** Description of the location where the incident occurred.
- **Arrest:** Indicates whether an arrest was made.
- **Domestic:** Indicates whether the incident was domestic-related as defined by the Illinois Domestic Violence Act.
- **Beat:** Indicates the beat where the incident occurred. A beat is the smallest police geographic area – each beat has a dedicated police beat car. Three to five beats make up a police sector, and three sectors make up a police district. The Chicago Police Department has 22 police districts.
- **District:** Indicates the police district where the incident occurred.
- **Ward:** The ward (City Council district) where the incident occurred.
- **Community Area:** Indicates the community area where the incident occurred. Chicago has 77 community areas.
- **FBI Code:** Indicates the crime classification as outlined in the FBI's National Incident-Based Reporting System (NIBRS).
- **X Coordinate:** The x coordinate of the location where the incident occurred in State Plane Illinois East NAD 1983 projection.
- **Y Coordinate:** The y coordinate of the location where the incident occurred in State Plane Illinois East NAD 1983 projection.
- **Year:** Year the incident occurred.
- **Updated On:** Date and time the record was last updated.
- **Latitude:** The latitude of the location where the incident occurred.
- **Longitude:** The longitude of the location where the incident occurred.

- **Location:** The location where the incident occurred in a format that allows for creation of maps and other geographic operations on this data portal.

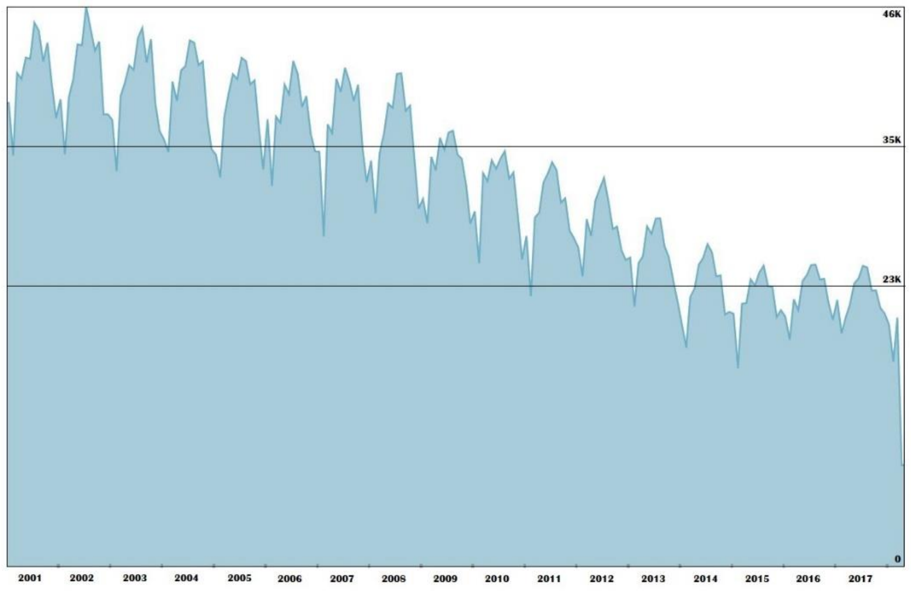
3 Data Analysis

One of goals for our project was to determine how has crime changed over the years. With statistical analysis applied on the dataset we discovered that the amount of committed crimes was annually reducing. That encouraged us to build several algorithms to determine what actually caused those changes.

3.1 Statistics

This graph shows how the amount of crimes was changing with every month during the period 2001 - 2017

Figure 1.1: Number of Reported Crimes by Date



3.2 Algorithm for data analysis.

This algorithm allows us to find which crime in particular influenced the drop of the crimes between two consecutive years the most. First, for every line in the dataset we take the Year and the PrimaryType of the crime record and pass it to the reducer as a key with a value equal to 1. The reducer sums up every common pair of keys which contains Year and PrimaryType. Next, every YearType/occurrences pair is saved in a dictionary where YearType is the key and occurrences is the value. It keeps saving the key/value pair to the dictionary until the year of the YearType value is changed for the first time. When that happens, we continue to save the YearType, occurrences value pair, except now we compare the amount of CrimeType occurrences of the new year with the previous year. We keep comparing each key/value pair of the previous year with the current year and keep saving maximum difference and type of crime with the maximum difference until the year is changed again. When that happens, we print the most reduced PrimaryType of a crime for a one-year period. This process continues until the last year.

Algorithm 1 Finding particular type of crime which annually influenced the drop of crime the most.

```
class MapReduce
    method Mapper (key, values)
        for all Year & PrimaryType ∈ values do
            yield (Year, PrimaryType), count 1

    method Reducer (YearType, occurrences [c1, c2, ...])
        sum ← 0
        for all occurrence ∈ occurrences [c1, c2, ...]) do
            sum ← sum + c
        yield (Year, Type), occurrence sum

    method Mapper2(YearType, occurrences)
        Year, Type ← YearType
        CurentYear ← Year
        if PreviousYear & CurentYear ≠ PreviousYear then
            if YearChangedFirstTime then
                yield (PreviousYear -1, PreviousYear), MostReducedCrime
                maxdif ← 0
                mostReducedCrime ← None
            else
                yearChangedFirstTime ← True
        if YearChangedFirstTime
            CrimeRecords [(Year, Type)] ← sum(occurrences)
            if (Year -1, Type) in CrimeRecords
                maxdif ← CrimeRecords [Year - 1] – CrimeRecords[Year]
                mostReducedCrime ← PrimaryType
        if YearChangedFirstTime == False
            CrimeRecords[YearType] ← sum(occurrences)
        PreviousYear ← CurentYear
```

4 K-means Clustering

A cluster is a group of objects that are similar amongst themselves but dissimilar to the objects in other clusters. Identifying meaningful clusters and thereby a structure in a large unlabeled dataset is an important unsupervised data mining task. Technological progress leads to enlarging volumes of data that require clustering. Clustering large datasets is a challenging resource-intensive task and the key to scalability and performance benefits is to use parallel or concurrent clustering algorithms.
We implement the k-means algorithm in two parts. One is MapReduce part that will calculate the k-means for the incoming clusters. Second is a runner that runs the MapReduce job multiple times until convergence.

4.1 k-means

K-means is a common and well-known clustering algorithm. It partitions a set of n objects into k clusters based on a similarity measure of the objects in the dataset. As the number of objects in the cluster varies, the centers of gravity of the cluster shifts. This algorithm starts off with the selection of the k-initial random cluster centers from the n objects. Each remaining object is assigned to one of the initial chosen centers based on similarity measure. When all the n objects are assigned, the new mean is calculated for each cluster. These two steps of assigning objects and calculating new cluster centers are repeated iteratively until the convergence criterion is met. Comparing the similarity measure is the most intensive calculation in k-means clustering. While the distance calculation between any object and a cluster center can be performed in parallel, each iteration will have to be performed serially as the center changes will have to be computed each time.

4.2 Runner algorithm

Algorithm 2 Algorithm for Runner

Require:

- A set of d-dimensional objects $X = \{x_1, x_2, \dots, x_n\}$
- k-number of clusters where $k < n$
- Initial set of centroids $C = \{c_1, c_2, \dots, c_k\}$
- δ convergence delta

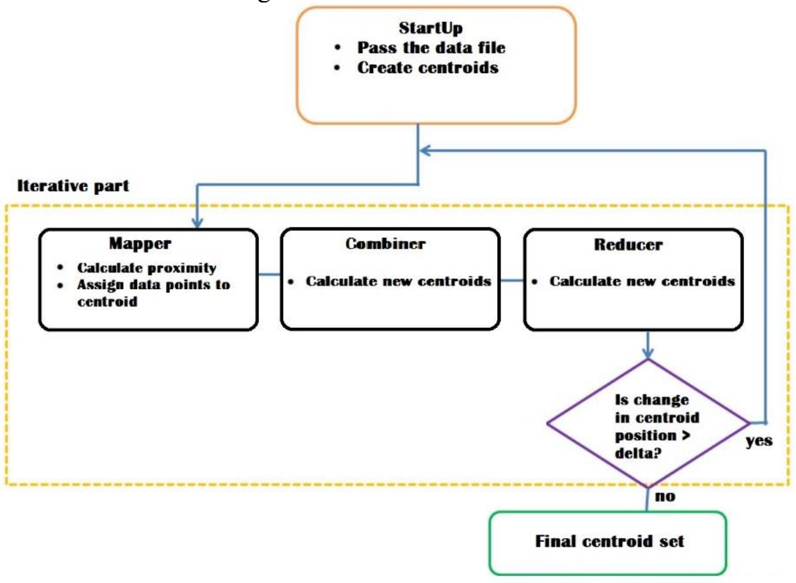
Output: a new set of centroids, number of iterations, final clusters, time taken to converge

```
1: initialize timetoConverge, numClusters, numIterations, convergenceDelta
2: startTime ← currentTime ()
3: generateInitialCentroids (C)

4: oldCentroids ← C
5: numIter ← 1
6: while True do
7:      newCentroids ← perform MapReduce ($X$, oldCentroids)
9:      newCentroids ← check(newCentroids)
9:      difference ← difference (newCentroids, oldCentroids)
10:     if difference < δ do:
11:           finalClusters ← newCentroids
12:           endTime ← currentTime ()
13:           timetoConverge ← (endTime − startTime)
14:           print numIter, finalClusters, timetoConverge
15:           break
16:     else do
17:           oldCentroids ← newCentroids
18:           numIter ← numIter + 1
```

We use a runner program to initiate the process by defining number of clusters, convergence delta and generate random initial set of centroids. Then, we call the MapReduce program. The Mapper assigns each object to cluster by calculating the distance between the centroid and the objects. The mapper algorithm calls the Reducer to recalculate new centroids by calculating the mean of all assigned objects to the centroid. The check function generates new coordinates for the centroids that were not assigned to any points. The runner program, then, evaluates the change in the centroid positions and compares it with the convergence delta that we have defined. If the change is more than δ, the start-up program iterates through MapReduce job again. When the change in centroid position is less than δ, we assume convergence. At this point, the algorithm generates the output file with final clusters and calculates the elapsed time.

The structure the algorithm is given below:

Figure 1.2: Structure of the Algorithm



4.3 Mapper algorithm

Each input object in the subset is assigned to its closest centroid by the mapper. We have used Euclidean distance to measure proximity of points. The distance between each point and each centroid is measured and the shortest distance is calculated. Then, the object is assigned to its closest centroid. When all the objects are assigned to the centroids, the mapper sends all the input objects and the centroids they are assigned to, to the combiner and reducer.

Algorithm 3 Algorithm for Mapper

Require:

- A subset of d-dimensional objects of $(x_1, x_2, \dots, x_n)$ in each mapper
- initial set of centroids $C = \{c_1, c_2, \dots, c_k\}$

Output: A list of centroids and objects assigned to each centroid

```
1: $M_1 \leftarrow (x_1, x_2, \dots, x_m)$
2: Centroids $\leftarrow$ C
3: euclDist $(p, c)$ = $\sqrt{\sum_{i=1}^{d}(p_i - c_i)^2}$ where $p_i$ (or $c_i$) is the coordinate of $p$ (or $c$) in dimension.
4: method Mapper
5:     for all $x_i \in M_1$ such that $1 \leq i \leq m$ do
```

```
6:          point ← $x_i$
7:          minDist ← ∞
8:          for centroid ∈ Centroids do
9:              dist ← euclDist (point, centroid)
10:             if dist < minDist then
11:                 minDist ← dist
12:                 centroid ← i
13:         yield centroid, point
```

4.4 Reducer algorithm

The reducer accepts the key/value pair output from the mappers, loops through the sorted output of the mappers and calculates new centroid values. For each centroid, the reducer calculates a new value based on the objects assigned to it in that iteration. This new centroid list is emitted as the reducer value output and sent back to the runner program.

Algorithm 4 Algorithm for Reducer

Require:

Input: (cluster, points) where cluster is a centroid and the points are points assigned to the centroids by the mapper.

Output: (_, value) where value is the cluster id and the new best centroid

```
1: method Reducer (cluster, points)
2:      count ← null
3:      sumX ← null
4:      sumY ← null
5:      for all point ∈ points do:
6:          count += 1
7:          sumX += point.x
8:          sumY += point.y
9:      yield None, (cluster, sumX ÷ count, sumY ÷ count)
```

4.5 Details of Runner implementation

Here we describe the most challenging and interesting details of our clustering MapReduce implementation.

4.5.1 Passing files to MRJob

One of the challenges that were faced was how to pass an additional file to the MRJob. We not only have to pass a clusters file to the MRJob every time we run it, but also specify the path to the data file on HDFS.

To pass the clusters file over and over to the MRJob for every iteration we used a configure option in the MRJob class itself. To pass files through to the mapper/reducer we used add_file_option() where the value represents the path to the file.
The file will then be available to each task local directory and the value of the option will be changed to its path. The file can be accessed with the self.options.centroids as a file path. After we have done that we can easily pass any file to the MRJob, from the Runner, using the add_file_option() value with an equal sign and the path of the file.

```python
def configure_options(self):
    super(MRJobKMeans, self).configure_options()
    self.add_file_option('--centroids')

def loadfile(self):
    centroids = []
    with open(self.options.centroids, 'r') as f:
        for line in f:
            lat, long = line.split(',')
            centroids.append([float(lat), float(long)])
    return centroids


mr_job = MRJobKMeans(args=args + ['--centroids=' + TempFile])
```

To pass the data file from HDFS we save the HDFS path of the file, using sys.argv. It is a list in Python, which contains the command-line arguments passed to the script. Then using that list we can pass the data file every time we want to create an MRJob.

```python
args = sys.argv[1:]
mr_job = MRJobKMeans(args=args + ['--centroids=' + TempFile])
with mr_job.make_runner() as runner:
runner.run()
```

4.6 Hadoop Setup

We have tested this code on Sandbox and Hadoop cluster with one master and ten slave nodes.

4.6.1 Running on Sandbox

```
python Runner.py --hadoop-streaming-jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming.jar -r hadoop hdfs:///Project/Crimes2001-2017.csv
```

Here we run the Runner.py with python, specify the path for hadoop-streaming.jar file and the dataset on HDFS. We don't have to include the initial cluster file here, it is generated and passed to the MRjob with Runner.

4.6.2 Running on the cluster

```
python Runner.py -r hadoop hdfs:///Project/Crimes2001-2017.csv
```

4.7 Performance tuning

In order to tune the performance of the clustering algorithm we decided to modify the implementation of the algorithm. First, we decided to add a combiner to it, run it several times on the whole dataset and see whether the performance increased or not. Second, we changed the implementation of the mapper.

4.7.1 Combiner

A Combiner in Hadoop is a mini reducer that performs the local reduce task. Many MapReduce jobs are limited by the network bandwidth available on the cluster, so the combiner minimizes the data transferred between map and reduce tasks. Combiner function will run on the Map output and combiner's output is given to Reducers as input. In one-word, combiner function is used for network optimization. [5]
If the map generates more number of outputs as per requirement, then we need to use combiner but:

- Unlike a Reducer, input/output key and value types of combiner must match the output types of your Mapper.
- Combiners can only be used on the functions that are commutative $\{a.b = b.a\}$ and associative $\{a.(b.c) = (a.b).c\}$. From this, we can say that combiner may operate only on a subset of your keys and values. Or may does not execute at all, still, you want the output of the program to remain same.
- From multiple Mappers, Reducer get its input data as part of the partitioning process. Combiners can only get its input from one Mapper.
- Combiner function is used as per requirements, it has not replaced Reducer. The execution of combiner is not guaranteed, it may be called 0, 1 or more times

In our case combiner is similar to the reducer.

4.7.2 Modification of the mapper

For further performance improvements we did a modification to the mapper algorithm. We noticed that getting the nearest cluster id with a function results in a slightly better performance.

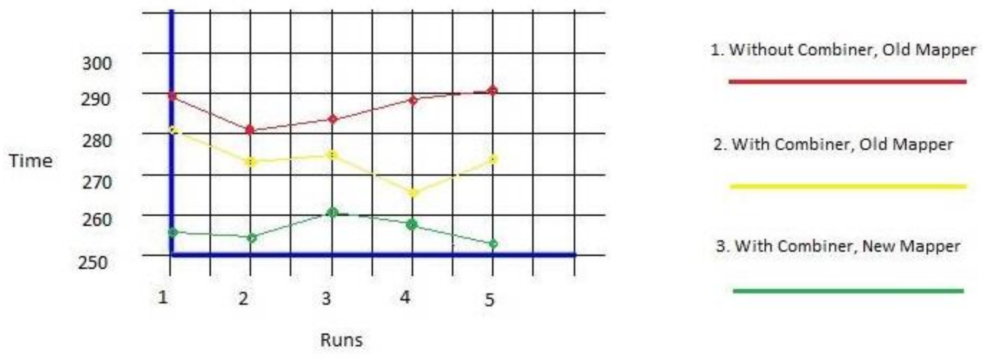
```
def get_nearest_cluster(self, point, centroids):
    nearest_cluster_id = None
    nearest_distance = 10000000
    for i in range(len(centroids)):
        dist = self.eucl_dist(point, centroids[i])
        if dist < nearest_distance:
            nearest_cluster_id = i
            nearest_distance = dist
    return nearest_cluster_id

def mapper(self, _, value):
    centroids = self.loadfile()
    try:
        NR, ID, CaseNumber, Date, Block, IUCR, Prima-
ryType, Description, LocationDescription, Arrest, Domes-
tic, Beat, District, Ward, CommunityArea, FBICode, XCoor-
dinate, YCoordinate, Year, UpdatedOn, Latitude, Longi-
tude, Location = words

        point = [float(Latitude), float(Longitude)]
        centroid = self.get_nearest_cluster(point, cen-
troids)

        yield centroid, point

    except ValueError:
        pass
```

This is a simplified version of the MapReduce python code. Full code can be found at the end of the Report.

4.7.3 Graph illustration of algorithms performance improvements

We made a total of 15 complete runs on the whole 8 million record dataset. Five for each algorithm. The time of one run is the average time taken by 1 iteration. Each run required a different amount of iterations, the maximum amount of iterations per run for a convergence delta of 0.01 was 10 and the minimum 4.

Figure 1.3: Performance illustration



1. Without Combiner, Initial Mapper implementation

Run no.	Num of Itera-tions	Time (sec)	Avg per Iteration
1	7	2027	289.5
2	6	1684	280.6
3	6	1699	283.1
4	5	1444	288.8
5	8	2323	290.3
Total			286.4

2. With Combiner, Initial Mapper implementation

Run no.	Num of Itera-tions	Time (sec)	Avg per Iteration
1	4	1123	280.7
2	10	2733	273.3
3	7	1929	275.5
4	6	1599	266.5
5	4	1092	273
Total			273.8

3. With Combiner, Tuned Mapper

Run no.	Num of Itera-tions	Time (sec)	Avg per Iteration
1	7	1791	255.8
2	8	2034	254.3
3	5	1304	260.8
4	6	1552	258.7

5	10	2524	252.4
Total			256.4

4.8 Discussion and Future improvements

Unless our solution shows good performance and results, we think that it is a bit complicated. We think that a spark implementation would be much easier.
Hadoop is great for dealing once a very large number of data with RAM-less computer. If you have RAM and you want to do more complicated work on the data, use Spark. If you want to develop low-cost algorithm for data processing, use Spark. Otherwise, you will lose so many time programming MapReduce job for not a good performance. So, do Hadoop ETL or adopt Spark.

5 Code

5.1 Algorithm for drop in crimes

```
class MRRatingCounter(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper,
                   reducer=self.reducer),
            MRStep(mapper_init=self.mapper_init,
                   mapper=self.mapper_avg)
        ]

    def mapper(self, key, value):

        line = value.replace(', ', '').strip()
        words = line.split(",")

        try:
            NR, ID, CaseNumber, Date, Block, IUCR, Prima-
ryType, Description, LocationDescription, Arrest, Domes-
tic, Beat, District, Ward, CommunityArea, FBICode, XCoor-
dinate, YCoordinate, Year, UpdatedOn, Latitude, Longi-
tude, Location = words
            Date = Date[:2]
            Date = Year + "/" + Date
            yield Date, 1

        except ValueError:
            pass
    def reducer(self, key, values):
        yield key[0:4], (key[5:], sum(values))

    def mapper_init(self):
        self.oldkey = None
        self.min = 10000000
        self.max = 0
        self.monthmin = None
        self.monthmax = None
        self.currentkey = None

    def mapper_avg(self, key, value):

        self.currentkey = key

        if self.oldkey and self.oldkey != self.cur-
rentkey:
            yield (self.oldkey, self.monthmax,
self.max),(self.oldkey, self.monthmin, self.min)
            self.min = 1000
            self.max = 0
            self.monthmin = None
            self.monthmax = None

        if value[1] < self.min:
            self.min = value[1]
            self.monthmin = value[0]

        if value[1] > self.max:
            self.max = value[1]
            self.monthmax = value[0]

        self.oldkey = self.currentkey

if __name__ == '__main__':
    MRRatingCounter.run()
```

5.2 MapReduce Clustering Python Code

```
from mrjob.job import MRJob
import mrjob
from math import sqrt

class MRJobKMeans(MRJob):
    SORT_VALUES = True
    def configure_options(self):
        super(MRJobKMeans, self).configure_options()
        self.add_file_option('--centroids')

    def loadfile(self):
        centroids = []
        with open(self.options.centroids, 'r') as f:
            for line in f:
                lat, long = line.split(',')
                centroids.append([float(lat),
float(long)])
        return centroids

    def eucl_dist(self, point, centroid):
        return sqrt(pow(centroid[0] - point[0], 2) +
pow(centroid[1] - point[1], 2))

    def mapper(self, _, value):

        centroids = self.loadfile()

        line = value.replace(', ', '').strip()
        words = line.split(",")

        try:
            NR, ID, CaseNumber, Date, Block, IUCR,
PrimaryType, Description, LocationDescription, Arrest,
Domestic, Beat, District, Ward, CommunityArea,
FBICode, XCoordinate, YCoordinate, Year, UpdatedOn,
Latitude, Longitude, Location = words

            point = [float(Latitude), float(Longi-
tude)]
            min_dist = 100000000
            centroid = 0

            for i in range(len(centroids)):
                dist = self.eucl_dist(point, cen-
troids[i])
                if dist < min_dist:
                    min_dist = dist
                    centroid = i

            yield centroid, point

        except ValueError:
            pass

    def combiner(self, cluster, points):
        count = 0
        sum_x = 0.0
        sum_y = 0.0
        for point in points:
            count += 1
            sum_x += point[0]
            sum_y += point[1]
        yield cluster, (sum_x / count, sum_y / count)


    def reducer(self, cluster, points):
        count = 0
        sum_x = 0.0
        sum_y = 0.0
        for point in points:
            count += 1
            sum_x += point[0]
            sum_y += point[1]
        yield None, (str(cluster) + "," + str(sum_x /
count) + "," + str(sum_y / count))


if __name__ == '__main__':
    MRJobKMeans.run()
```

5.3 Runner Code

```
from mrjob.job import MRJob
from Clustering import MRJobKMeans
import sys, os, random
import os.path
import shutil
from math import sqrt
import time
from operator import itemgetter, attrgetter

InitialCentroids = "centroids.txt"
TempFile = "centroids_temp.txt"
FinalClusters = "final_clusters.txt"
convergence_delta = 0.01
k_delta = 10
```

```
def generate_init_centroids(num):
    centroids = []
    for i in range(0, num):
        centroid = (str(i) + "," + str(random.uni-
form(41.6, 42.05)) + "," + str(random.uniform(-87.5, -
87.95)))
        centroids.append(centroid)
    return centroids

def write_to_file(centroids, file):
    with open(file, 'w') as f:
        centroids = sorted(centroids, key=itemget-
ter(0))
        for centroid in centroids:
            k, cx, cy = centroid.split(",")
            f.write("%s,%s\n" % (cx, cy))

def read_from_file(file):
    centroids = []
    with open(file, 'r') as f:
        for line in f:
            if line:
                cords = line.split(",")
                x, y = cords
                centroids.append([float(x), float(y)])
    return centroids


def get_random_coords_in_region(id):
    centroid = (str(id) + "," + str(random.uni-
form(41.6, 42.05)) + "," + str(random.uniform(-87.5, -
87.95)))
    return centroid


def missing_elements(L, k_delta):
    start, end = 0, k_delta
    return sorted(set(range(start, end)).differ-
ence(L))


def kmeans_check(centroids, k_delta):
    centroids = sorted(centroids, key=itemgetter(0))
    exist = []
    missing_centroids = []
    for centroid in centroids:
        k, cx, cy = centroid.split(",")
        exist.append(int(k))
    missing_centroids = missing_elements(exist,
k_delta)

    if missing_centroids == []:
        return centroids
    else:
        for id in missing_centroids:
            centroid = get_random_coords_in_region(id)
            print("Regenerated centroid:", centroid)
            centroids.append(centroid)
            centroids = sorted(centroids, key=itemget-
ter(0))
        return centroids

def eucl_dist(point, centroid):
    return sqrt(pow(centroid[0] - point[0], 2) +
pow(centroid[1] - point[1], 2))

def get_job_centroids(job, runner):
    centroids = []
    for line in runner.stream_output():
        key, value = job.parse_output_line(line)
        centroids.append(value)
    return centroids

def difference(cs1, cs2):
    max_difrnc = 0.0
    for i in range(len(cs1)):
        dist = eucl_dist(cs1[i], cs2[i])
        if dist > max_difrnc:
            max_difrnc = dist
    return max_difrnc


if __name__ == '__main__':
    start_time = time.time()

    args = sys.argv[1:]
    print(args)

    generated_centroids = generate_init_cen-
troids(k_delta)
```

```
    write_to_file(generated_centroids, InitialCen-
troids)


    shutil.copy(InitialCentroids, TempFile)

    old_centroids = read_from_file(InitialCentroids)
    i = 1
    while True:
        print ("Step #%i" % i)
        mr_job = MRJobKMeans(args=args + ['--cen-
troids=' + TempFile])
        #mr_job.set_up_logging(stream=sys.stdout)
        with mr_job.make_runner() as runner:
            runner.run()
            centroids = get_job_centroids(mr_job, run-
ner)
            centroids2 = kmeans_check(centroids,
k_delta)
            write_to_file(centroids2, TempFile)

        new_centroids = read_from_file(TempFile)
        max_dif = difference(new_centroids, old_cen-
troids)
        print("Maximum difference: ", max_dif)
        if max_dif < convergence_delta:
            print("Final Clusters:")
            j = 1
            for c in new_centroids:
                print(j, c[0], c[1])
                j += 1
            elapsed_time = time.time() - start_time
            write_to_file(centroids2, FinalClusters)
            os.remove(TempFile)
            os.remove(InitialCentroids)
            print("Running time: %i - seconds" %
elapsed_time)
            print("Amount of iterations: %i" % i)
            print ("Average time per iteration: %f" %
float(elapsed_time / i))
            break
        else:
            old_centroids = new_centroids
            i += 1
```

5.3 Sample output

```
ubuntu@master:~/Toy_data$ python Runner.py -r hadoop
hdfs:///Toy_data/Chicago_Crimes_2001_to_2017.csv
['-r', 'hadoop', 'hdfs:///Toy_data/Chi-
cago_Crimes_2001_to_2017.csv']
Step #1
('Regenerated centroid:', '8,41.8607150, -87.9067431')
('Maximum difference: ', 5.917641607197041)
Step #2
('Maximum difference: ', 0.4247055835879136)
Step #3
('Maximum difference: ', 0.23029309450545374)
Step #4
('Maximum difference: ', 0.02107006470200763)
Step #5
('Maximum difference: ', 0.016160529406906408)
Step #6
('Maximum difference: ', 0.009595897475536538)
Final Clusters:
(1, 41.7790780092, -87.6571851681)
(2, 41.9595701941, -87.7992396574)
(3, 41.8793011066, -87.6461079852)
(4, 41.7601668102, -87.5874181628)
(5, 36.619446395, -91.686565684)
(6, 41.9006012597, -87.7346682736)
(7, 41.825285236, -87.7176595917)
(8, 41.9683452455, -87.6781799743)
(9, 41.7323280255, -87.6930629698)
(10, 41.6955993204, -87.625832783)
Running time: 1791 - seconds
Amount of iterations: 6
Average time per iteration: 298.659068
```

6 References

1. https://www.kaggle.com/currie32/crimes-in-chicago
2. https://en.wikipedia.org/wiki/MapReduce
3. https://www.tutorialspoint.com/hadoop/hadoop_mapreduce.htm
4. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. 3rd edn. Springer-Verlag, Berlin Heidelberg New York (1996)
5. http://data-flair.training/forums/topic/what-is-difference-between-reducer-and-combiner