ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

BIG DATA MANAGEMENT SYSTEMS:
PROJECT #1 MAPREDUCE/HADOOP

Big Data Management Systems

Supervisor: Prof. Damianos Chatziantoniou

Dimitrios Bouris (8190119)
Filippos Priovolos (8190147)

Contents

GitHub Repository

To avoid turning out repository Public we used a tool called Gitfront. GitFront is used to share private git repositories without making them public to people who do not necessarily have github accounts.

—--

The Gitfront link to view our repository can be found here.

—--

Note: *In the event of a non responding link please contact us.*

Project Description

The aim of this project is to implement the K-Means clustering algorithm using the Hadoop MapReduce framework in Python. Clustering algorithms are a common machine learning technique that involves grouping similar data points together. The K-Means algorithm specifically involves grouping data points into K clusters, where K is a predefined number. The algorithm does this by iteratively computing the mean of each cluster and reassigning data points to their nearest cluster.The MapReduce framework is a parallel computing model that is commonly used to process large datasets. It involves breaking down a large dataset into smaller chunks, processing those chunks in parallel, and then aggregating the results. This makes it well-suited for implementing clustering algorithms on large datasets.In this project, we will generate a synthetic dataset of 2D data points using the Scikit-learn library's make_blobs method. We will then use the MapReduce framework to implement the K-Means algorithm in Python and more specifically the Map - Combine - Reduce paradigm. Specifically, we will use Hadoop's MapReduce streaming API to perform the Map and Reduce phases of the K-Means algorithm. The Map phase will involve assigning each data point to its nearest cluster the Combine phase will make some sub-aggregations, while the Reduce phase will involve computing the mean of each cluster and reassigning data points to their nearest cluster.

To run the MapReduce job, we will use a Python script that acts as a Map-Combine-Reduce runner. The script will take care of setting up the Hadoop environment and running the MapReduce job using Hadoop's command line tools. Once the MapReduce job has completed, we will examine the output to determine the final coordinates of the centers for each cluster.

Overall, this project will provide a good introduction to both the K-Means clustering algorithm and the MapReduce framework, as well as practical experience implementing them using Python.

Hadoop Installation

The installation of Hadoop on our local machine (Mac M1) was done according to the following instructions:

- Article 1
- Article 2
- YouTube Video

Note: To successfully install and configure hadoop locally, ***python*** and ***java jdk*** need to be installed. Versions Used:

- `python 3.9.15`
- `java jdk1.8.0_361`

Data Generation

For our K-Means MapReduce implementation, we generated a synthetic dataset consisting of 1.2 million two-dimensional data points. The data points were generated using the scikit-learn library's make_blobs method. The make_blobs method generates isotropic Gaussian blobs for clustering. We generated data points around three centers located at (-100000, -100000), (1, 1), and (100000, 100000) with a standard deviation of 6.0.
The dataset was generated using the generateDataset.py script which takes three input parameters: dataset_size, centers, and export_path. We specified a dataset size of 1.2 million rows, three centers, and an export path of "data-points.csv". The data was exported to a CSV file with two columns representing the x and y coordinates of each data point.

MapReduce Implementation of KMeans

For implementing the KMeans Clustering algorithm in a Hadoop, the programming paradigm Map - Combine - Reduce was used. Map - Combine - Reduce is a variation of the MapReduce programming model that adds an additional "combine" phase between the "map" and "reduce" phases. The "combine" phase is similar to the "reduce" phase in that it aggregates key-value pairs.
The process consists of 4 parts in total.

Mapping Phase

The mapping phase of the operation is responsible for assigning each input point to the optimal cluster, which is the cluster whose center is the nearest (Manhattan distance). More specifically, the mapper reads each data point from the input file and emits the key-value pair for each data point. The key will be the closest center to the data point, and the value will be the data point itself.

- Output Shape: **“num_cluster   [x,y]”**

This operation occurs in the mapper.py file. The code is represented below.

```python
#!/usr/bin/env python
# The above shebang is used to specify the interpreter to be used when run in the #
# hadoop cluster
import sys
import numpy as np

def read_centers(centers_file):
   """Reads the centers from a CSV file."""
   with open(centers_file, "r") as f:
       centers = []
       # the old-centers file contains previous centers
       # each center is represented with the x and y coordinates
       # each line represents one center
       for line in f:
           # decode the x and y coordinates into 2 separate variables
           x, y = map(float, line.strip().split(","))
           centers.append([x, y])
       return centers

def calculate_distances(point, centers):
   """Calculates the Manhattan distance between a point and each center."""
   distances = []
   for center in centers:
       # calculate the Manhattan distance between the point and the center
       x_distance = abs(point[0] - center[0])
       y_distance = abs(point[1] - center[1])
       distance = x_distance + y_distance
       # save all the distances in a list
       distances.append(distance)
   return distances

def find_closest_center(distances):
   """Finds the index of the closest center."""
   # find the closest center from the distances matrix
   # the arg min function returns the index of the list -> the id of the center
   return np.argmin(distances)

centers = read_centers("files/current_centers.csv")

# iterate over each input line
for line in sys.stdin:
   # decode the line
   point = list(map(float, line.strip().split(",")))
```

```
    # get the distance the point has from each center
    distances = calculate_distances(point, centers)

    # find the closest center
    cluster = find_closest_center(distances)

    # print the result to std out
    print('%s\t%s' % (cluster, point))
```

Combining Phase

In the combiner phase, we will receive the key-value pairs from the mapper and calculate the partial sum of each dimension of each data point that belongs to each cluster. The combiner phase takes as input the output of the mapper phase.

- Input Shape: **“num_cluster [x,y]”**
- Output Shape: “**num_cluster ([subsum_x, subsum_y], num_points_participated)”**

This operation occurs in the combiner.py file. The code is represented below.

```
#!/usr/bin/env python

import sys

import ast

# variable to identify the cluster we are working on

working_cluster = None

# the cluster the line belongs

cluster = None

subsum = []



def parse_input(line):



    """ The function decodes the stdin which comes from the mapper """

    cluster, point = line.strip().split('\t', 1)

    point = ast.literal_eval(point)
```

```
    return cluster,point


# iterate over each line of the stdin

for line in sys.stdin:



    # decode the input and get the cluster and the coordinates of the point

    cluster, point = parse_input(line)



    # Note: The input is sorted by the clusterid so all the cluster 1 items will
come first,

    # then the ones with cluster 2 and lastly the ones in the 3rd cluster

    # we use the cluster and current cluster to identify when one cluster is done
and the

    # other cluster starts (cluster != current cluster)



    # if we still have data about the current cluster, alter the sums

    if working_cluster == cluster:

        # add the x and y coordinates to the equivalent sums

        subsum[0] += point[0]

        subsum[1] += point[1]

        num_points += 1

     # if a new cluster starts, print the partial sum and the number of points
participated in it

    else:

        if working_cluster:

            # Write cluster, partial sum and number

            print ('%s\t%s' % (working_cluster,
```

```
                (subsum, num_points)))



        # Update the sums with the values of the point of the new cluster

        subsum = point

        num_points = 1

        working_cluster = cluster



# output the sum and num of points of the last cluster as it does not happen above

if working_cluster == cluster:

   print ('%s\t%s' % (working_cluster, (subsum, num_points)))
```

Reducing Phase

The reducer is responsible for calculating the new centers of each cluster. The new cluster center is calculated as the average of x and y coordinates of the items that have been assigned to each cluster. This is done by taking the sum of all the points assigned to a particular cluster and dividing it by the total number of points assigned to that cluster. The reducer uses as input the output of the combination phase.

- Input Shape: **“num_cluster ([subsum_x, subsum_y], num_points_participated)”**
- Output Shape: **“[new_center_x, new_center_y]”**

This operation occurs in the reducer.py file. The code is represented below.

```
#!/usr/bin/env python
import sys
import ast

def parse_input(line):

   """This function decodes the input line from stdin"""

   cluster, partial = line.strip().split('\t', 1)
   sub_sum = ",".join(partial.split(",", 2)[:2]).replace("(", "")
   num_points = partial.split(",", 2)[2].replace(")", "")
   num_points = int(num_points)
   sub_sum = ast.literal_eval(sub_sum)
```

```python
    return cluster, sub_sum, num_points

def calculate_center(total_sub_sum, num_points):

    """This function decodes the input line from stdin"""

    # the new centroid is calculated as the mean value of the x and y coordinates
    # of the points which have been assigned to the cluster
    xCenter = round(total_sub_sum[0] / num_points, 1)
    yCenter = round(total_sub_sum[1] / num_points, 1)
    # the new centroid is returned as a list in the std out
    new_centroid = [xCenter, yCenter]
    print('%s\t' % (new_centroid))

# instantiate some counters
working_cluster = None
total_sub_sum = []
cluster = None
num_points = 0


# iterate over all the input lines
for line in sys.stdin:

    # decode the line and get the clusterid, sub_sum and number of points that the
sum is for
    cluster, sub_sum, num_points = parse_input(line)

    # Note: The input is sorted by the clusterid so all the cluster 1 items will
come first,
    # then the ones with cluster 2 and lastly the ones in the 3rd cluster
    # we use the cluster and current cluster to identify when one cluster is done
and the
    # other cluster starts (cluster != current cluster)

    # if we still have data about the current cluster, alter the sums
    if working_cluster == cluster:
        total_sub_sum[0] += sub_sum[0]
        total_sub_sum[1] += sub_sum[1]
        num_points += num_points
    # if a new cluster starts, get the new centroid of the ready cluster
    else:
        # if the current cluster has a value, meaning that it is not the first
iteration
        if working_cluster:
```

```
            calculate_center(total_sub_sum, num_points)
        # restart the sum variables to the first values of the new cluster
        total_sub_sum = sub_sum
        num_points = num_points
        working_cluster = cluster

# calculate the new center of the last cluster as it does not happen above
if working_cluster == cluster:
    calculate_center(total_sub_sum, num_points)
```

KMeans Master file

The kMeans.py code integrates the three MapReduce steps and runs the K-Means algorithm iteratively until convergence is achieved. The MapReduce runner coordinates the execution of the K-Means algorithm across multiple machines in a distributed computing environment. In each iteration, the current centers are fed to the mapper phase, which emits the key-value pairs for each data point. These pairs are then aggregated by the combiner and reducer phases to produce the new centers. The process is repeated until the centers converge.

The MapReduce Runner is responsible for configuring and initiating the Hadoop streaming job, which implements the K-Means algorithm in a distributed fashion. It specifies the input and output directories, the mapper, combiner, and reducer scripts, and any additional files that are required for the job. The runner then submits the Hadoop job to the cluster and monitors its progress.

In our implementation, the MapReduce Runner is written in Python and uses the subprocess module to execute shell commands. The Hadoop streaming jar file and its associated libraries are installed locally, and the runner interacts with the Hadoop cluster through the Hadoop command-line interface.

The runner first defines the input and output directories for the Hadoop job. It then specifies the mapper, combiner, and reducer scripts, as well as any additional files required by the scripts. In our implementation, the current centers file is passed to the reducer script as an additional file.

Once the Hadoop job has been configured, the runner uses the subprocess module to execute the hadoop jar command, passing in the necessary arguments. The output of the Hadoop job is then copied from HDFS to the local file system using the Hadoop fs command.

After the output file has been retrieved, the runner checks whether the algorithm has converged. If the algorithm has converged, the runner terminates the job and prints the final coordinates of the centers. If the algorithm has not converged, the runner updates the current centers file with the new centers and submits another Hadoop job.

The MapReduce Runner is an essential component of our implementation of the K-Means algorithm in the MapReduce framework. It enables the algorithm to be executed efficiently and scalably in a distributed computing environment.

```
#!/usr/bin/env python

"""
kMeans.py: Implements the K-Means algorithm using the Map - Combine - Reduce Hadoop
Operation
"""

import ast
import random
import subprocess

# set a random seed for reproductivity
random.seed(13)

def emptyFiles():
   files = [current_centers, allCenters]
   for file in files:
       with open(file, 'w') as file:
           file.write('')

def getPoints(file):
   """ Gets all the x and y coordinates of the points from the file specified """
   with open(file, "r") as data:
       data = data.readlines()
       dataList = []
       for d in data:
           d = d.strip().split(",")
           d = [float(d[0]), float(d[1])]
           dataList.append(d)
   return dataList

def storeCenters(centers):
   """ Saves each group of calculated centers in a file """
   with open(allCenters, "a") as file:
       # iterate over all the centers and write them in the file
       for center in centers:
           file.write("%s\n" % str([center]).strip('[]'))

def getCenters(file_path):
   """ Gets the centers from the file specified. Each center is stored as a tuple
of the
```

```
    x and y coordinates and each line represents a center.
    """
    with open(file_path, "r") as cfile:
        # read all the lines of the centers file
        cfile = cfile.readlines()
        centers_all = []
        # iterate over all the lines - centers and append them in a list
        for center in cfile:
            center = ast.literal_eval(center)
            centers_all.append(center)
    return centers_all

def checkConvergence(previous_centers, new_centers):
    """ Checks if the current and previous centers have converged meaning that they
have not changed
    much between the two iterations of the algorithm. Takes as an input the two
lists of the old and new centers """
    converged = False
    # sort them before as they might be in a different order
    # if they are the same, return True and stop the iterations
    if sorted(previous_centers) == sorted(new_centers):
        converged = True
    # else return False meaning that we need at least one more iteration
    return converged

def replaceOldCenters(centers):
    """ Replaces the old center with the newly calculated ones """
    # open the file with the w+ parameter which opens the file for writing and
truncates the file to zero length.
    with open(current_centers, "w+") as file:
        # iterate over each center and write it to the file
        for center in centers:
            file.write("%s\n" % str([center]).strip('[]'))

if __name__ == "__main__":

    # specify the path foreach file used
    current_centers = "files/current_centers.csv"
    dataPoints = "files/data-points.csv"
    LocalHadoopOutput = "files/LocalHadoopOutput/part-00000"
    allCenters = "files/all-centers.csv"


    emptyFiles()
```

```
    # Retrieve the initial data points
    dataPointsList = getPoints(dataPoints)
    # Randomly generate the initial centers foreach cluster
    num_clusters = 3
    initialCentroids = random.sample(dataPointsList, k=num_clusters)
    replaceOldCenters(initialCentroids)
    storeCenters(initialCentroids)

    converged = False
    # As long as the centers have not converged, repeat
    while (converged == False):

        # Connect with HDFS and run Map-Combine-Reduce process through the Hadoop
Streaming
        # we need to specify the mapper - combiner and reducer files and also all
the input files
        # as well as the hadoop path where the output will be stored
        # the process below implements the first iteration of the KMeans algorithm

        map_combine_reduce =
subprocess.run(["/Users/dimitrisbouris/hadoop-3.2.3/bin/hadoop", "jar",

"/Users/dimitrisbouris/hadoop-3.2.3/share/hadoop/tools/lib/hadoop-streaming-3.2.3.j
ar",
        "-file", "mapper.py", "-mapper","mapper.py", "-file","combiner.py"
,"-combiner" ,"combiner.py",
        "-file", "reducer.py", "-reducer", "reducer.py", "-file",
"files/current_centers.csv", "-input", "/kmeans/data-points.csv",
        "-output","kmeans_output/output"], stdout=subprocess.PIPE)

        # After the completion of the first iteration, we will copy the output file
from hdfs to the local folder
        # This output file will be used as input to the second iteration. This
process will repeat foreach iteration of the algo.
        # The output file is stored in hadoop by default with the name "part-00000"
        # This file is the output of the reducer and thus represents the new centers

        copy = subprocess.run(["/Users/dimitrisbouris/hadoop-3.2.3/bin/hadoop",
"fs", "-get",
        "/user/dimitrisbouris/kmeans_output/output/part-00000",
"files/LocalHadoopOutput/"])

        # delete the output directory created before as the hdfs throws an error as
it attempts to recreate it
        delete = subprocess.run(["hdfs", "dfs", "-rm", "-r" , "/user/"])
```

```python
        # get the old and new centers
        previous_centers = getCenters(current_centers)
        new_centers = getCenters(LocalHadoopOutput)
        previous_centers= [list(center) for center in previous_centers]
        storeCenters(new_centers)

        # chech if they have converged
        converged = checkConvergence(previous_centers, new_centers)
        # If the centers have changed run another iteration of the KMeans algo
        if converged == False:
            # Update the current centers file with the new ones
            replaceOldCenters(new_centers)
            # we also remove the part-00000 we got from the last iteration in order
to replace it with the next one
            remove_part_00000 = subprocess.run(["rm", "-r",
"files/LocalHadoopOutput/part-00000"])
        # If the centers have converged, end the algo and print the final
coordinates of the centers
        else:
            # delete the part_00000 from local directory
            remove_part_00000 = subprocess.run(["rm", "-r",
"files/LocalHadoopOutput/part-00000"])
            # Print a small report with the results of the operation
            print()
            print("The Map - Combine - Reduce process ended succesfully!")
            print("The final calculated coordinates of the centers are: ")
            for i in range(len(new_centers)):
                print("Cluster " + str(i) + ": " + str(new_centers[i]))
```

Execution & Results

To execute the K-Means algorithm on our test dataset, we first need to generate the dataset by running the generateDataset.py script located in the /files folder. This script contains the generateDataPoints() function that generates a dataset of 2D data points distributed around pre-specified centers. Once the dataset is generated, we need to start the Hadoop clusters and move the data points file to HDFS by running the following commands from the /hadoop directory of the project:

```
$ start-all.sh # start all the daemons (processes) required to run a Hadoop cluster

$ hdfs dfs -mkdir /kmeans #create a directory to save the data points file

$ hdfs dfs -put files/data-points.csv /kmeans # move the data points file to hdfs
```

After moving the dataset to HDFS, we can execute the KMeans.py script. This script runs the Map-Reduce operation and prints the final cluster centers. However, to run the project successfully, we need to change the Hadoop paths in the KMeans.py file to match our local installation.Once the Map-Reduce operation is completed, we can check the accuracy of the K-Means algorithm by examining the all-centers.csv file, which contains the different states of the centers for each iteration. The all-centers file generated contains the different states of the centers foreach iteration. Each line represents the x and y coordinates of each center. Since we have 3 clusters (thus 3 centers) the first 3 lines represent the centers of the first iteration and so on.
The 3 last coordinates which represent the centers in the last iteration of the KMeans algorithm match the pre-defined centers ([[-100000, -100000], [1, 1], [100000, 100000]]) which validates our implementation's accuracy.

```
# all-centers.csv

 -6.4, 5.9
 99996.0, 99998.7
 -3.0, -2.2
 0.0, 2.6
 61451.3, 61451.3
 -61450.7, -61452.3
 1.0, 1.0
 100000.0, 100000.0
 -100000.0, -100000.0
 1.0, 1.0
 100000.0, 100000.0
 -100000.0, -100000.0
```

During the execution of the KMeans.py script, logs are generated that provide information about the execution process. The logs contain information about the number of bytes read and written, number of input and output records, and other relevant metrics. The full logs can be found in the logs file of the project.The KMeans Map-Combine-Reduce operation successfully calculates the new centers after 3 iterations, and the returned centers match with the pre-defined ones, indicating that the KMeans algorithm ran successfully and that the results are accurate.

Here is a snippet of the logs generated from the first KMeans and Map - Combine - Reduce operation.
The full logs can be found here.

```
...

2023-04-21 16:42:29,197 INFO mapred.Task: Final Counters for
attempt_local868978880_0001_r_000000_0: Counters: 30
        File System Counters
                FILE: Number of bytes read=10885
                FILE: Number of bytes written=577409
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=18034485
                HDFS: Number of bytes written=54
                HDFS: Number of read operations=10
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=3
                HDFS: Number of bytes read erasure-coded=0
        Map-Reduce Framework
                Combine input records=0
                Combine output records=0
                Reduce input groups=3
                Reduce shuffle bytes=168
                Reduce input records=3
                Reduce output records=3
                Spilled Records=3
                Shuffled Maps =1
                Failed Shuffles=0
                Merged Map outputs=1
                GC time elapsed (ms)=0
                Total committed heap usage (bytes)=304611328
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Output Format Counters
                Bytes Written=54
    ...
```

References

- Medium: MapReduce with Python
- MapReduce Jobs in Python
- Mapreduce Python example
- GeeksForGeeks: MapReduce - Combiners