CSE530 Distributed Systems Winter 2024

Assignment-3

Problem Statement

Implementing MapReduce framework from scratch to perform K-means clustering on a given dataset in a distributed manner.

Implementing K-Means using MapReduce from scratch

0. Learning Resources

For this part of this assignment, you would need to be familiar with

- K-Means clustering algorithm: K-means Clustering: Algorithm, Applications, Evaluation Methods, and Drawbacks
- MapReduce Model
- K-Means algorithm using MapReduce Framework (Section: 3.1)

Setup:

1. For the sake of simplicity, we will deploy the entire MapReduce framework on one machine. But note that each mapper and each reducer, as well as the master, will be a SEPARATE PROCESS (having an IP address (localhost) and a distinct port no.) on the same machine. Additionally, each mapper and each reducer will store their intermediate/output data in a separate file directory in

the local file system (format specified in the “Sample Input & Output” section).

1. Mapper persists intermediate data
2. Reducer persists final output data

2. The use of gRPC for RPCs is mandatory for this assignment.
3. You may use any programming language you wish to use. We recommend using Python.
4. Google Cloud is not required for this assignment.

The K-Means algorithm

The K-means algorithm is an iterative algorithm that partitions a dataset into K clusters. The algorithm proceeds as follows:

1. Randomly initialize K cluster centroids.
2. Assign each data point to the nearest cluster centroid.
3. Recompute the cluster centroids based on the mean of the data points assigned to each cluster.
4. Repeat steps 2 and 3 for a fixed number of iterations or until convergence (i.e., until the cluster centroids no longer change significantly).

Refer to Section 3.1 of this paper for an explanation of how the K-Means clustering algorithm can be performed in a distributed manner. Note that you must implement the k-means algorithm from scratch (without using any library providing K-means algorithm function).

Implementation Details:

Your implementation should include the following components:

1. **Master**: The master program/process is responsible for running and communicating with the other components in the system. When running the master program, the following parameters should be provided as input:

- number of mappers (M)
- number of reducers (R)
- number of centroids (K)
- number of iterations for K-Means (Note: program should stop if the algorithm converges before)
- Other necessary information (This should be reasonable. Please check with us - if you are not sure!)

2. **Input Split (invoked by master):** You will need to write code to divide the input data (single file) into smaller chunks that can be processed in parallel by multiple mappers. Your code does not need to produce separate input files for each mapper, but each mapper should process a different chunk of the file input data. For partitioning the input data across different mappers, there are two possibilities:

- Scenario 1: Input data contains only one big file. Each mapper reads the entire input file and then processes only the indices allocated to it by the master.
- Scenario 2: Input data contains multiple files. Each mapper is responsible for a subset of these files (exact subset of files is allocated by Master).

In both the above scenarios, the master does not distribute the actual data to the mappers as that is going to be unnecessary network traffic.

For the purpose of this assignment, you are free to choose either one or both of the above scenarios.

Master should try to split the data equally (or almost equally) based on the number of mappers.

3. **Map (invoked by mapper):**

- You will need to write code to apply the Map function to each input split to generate intermediate key-value pairs.
- Mapper should read the input split by itself (based on the information provided by the master). Master should not send the input data points to the mapper.
- **Inputs to the Map function:**
    - Input split assigned by the master:
        - Range of Indices (Scenario 1) or List of file names (Scenario 2)
    - List of Centroids from the previous iteration
        - Other necessary information (This should be reasonable. Please check with us - if you are not sure!)
- **Output from the Map function**: For each data point processed by the map function, the function outputs:
    - Key: **index of the nearest centroid to which the data point belongs**
    - Value: **value of the data point itself**
- The output of each Map function should be passed to the partition function which will then write the output in a partition file inside the mapper's directory on the local file system. (Look at the directory structure given below)
- Note that each mapper needs to run as a separate process.
- Please ensure that mappers run in parallel, not sequentially. This is obvious but I have added it explicitly as some students seem to be running mappers sequentially using synchronous RPC calls in a loop.

4. **Partition (invoked by mapper):**

- The output of the Map function (as mentioned in the mapper above) needs to be partitioned into a set of smaller partitions.
- In this step, you will write a function that takes the list of key-value pairs generated by the Map function and partitions them into smaller partitions.
- The partitioning function should ensure that
    - all key-value pairs belonging to the same key are sent to the same partition file.
    - distribute the different keys equally (or almost equally) among each of the partitions. This can be done using very simple and reasonable partition functions (such as **key % num_reducers**)
- Each partition file is picked up by a specific reducer during shuffling and sorting.
- If there are M mappers and R reducers, each mapper should have R file partitions. This means that there will be **M*R partitions** in total.
- This step is performed by the mapper.

5. **Shuffle and sort (invoked by reducer):**

- You must write code to **sort the intermediate key-value pairs by key and group the values that belong to the same key.**
- This is typically done by sending the intermediate key-value pairs to the reducers based on the key.
- This step is performed by the reducer.

6. **Reduce (invoked by reducer):**

- The reducer **will receive the intermediate key-value pairs from the mapper, perform the shuffle & sort function** as

mentioned, and **produce a set of final key-value pairs as output.**

- You will need to write code to apply the Reduce function to each group of values that belong to the same key to generate the final output.
- Input to the reduce function:
    - Key: **Centroid id**
    - Value: **List of all the data points which belong to this centroid id** (this information is available after shuffle and sorting)
    - Other necessary information (This should be reasonable. Please check with us - if you are not sure!)
- Output of the reduce function:
    - Key: **Centroid Id**
    - Value: **Updated Centroid**
- The output of each Reduce function should be written to a file in the reducer's directory on the local file system.
- Note that each reducer needs to run as a separate process.
- [Added on April 14th, 2024] Please ensure that once the mappers have finished, all the reducers run in parallel, not sequentially. This is obvious but I have added it explicitly as some students seem to be running reducers sequentially using synchronous RPC calls in a loop.

7. **Centroid Compilation (invoked by master):** The master needs to parse the output generated by all the reducers to compile the final list of (K) centroids and store them in a single file. This list of centroids is considered as input for the next iteration. Before the first iteration, the centroids should be randomly selected from the input data points.

- Master must use gRPC calls to contact the Reducer for reading the output files (since in practice, master and reducer are not running on the same machines)
- Alternatively, Reducer can send the output file data as part of the response of the initial RPC request sent by the master (for running the reduce task). If you do this, then the master does not need to do a separate RPC for reading the output files from reducers.

Important Notes:

1. **Reducers should not read their input key-value pairs from the generated intermediate files**; instead, they should perform gRPC calls to the mappers to get the input. (Mapper however should read the input data points from the input files itself).
2. **Fault Tolerance**: Your implementation should take care of any failures associated with mapper or reducer. If a mapper or reducer fails, then the master should re-run that task to ensure that the computations finish. We will test two failure scenarios:
    1. **Scenario 1:** Mapper/reducer may encounter an error due to which it is not able to complete the allocated task. In that case, it will return "unsuccessful or failed" message to the master.
        - This is very easy to test. You can add a “probabilistic” flag just before returning to the master. The flag can be randomly set to false or true (probability for false can be 0.5 or higher so that some failure happens for sure). If the flag is false, return fail. else, return success. This is very easy to implement and test how the master deals with failures.

    Scenario 2: We may force stop a mapper or reducer process. [This seems difficult to test if you spawn mappers

and reducers from the master itself. But if they are spawned from a separate terminal, then it is easy to test as we can just kill the mapper process. You may have to add a sleep statement in mapper code to ensure you get sufficient time to kill the mapper.]
If you are running mappers and reducers from the master itself, then for your own testing, you can add sleep statements and kill the mapper/reducer using pid from the command line.

In both cases, master should ensure the failed task gets reassigned to the same or different mapper/reducer.

gRPC communication:

The gRPC communication between the three processes for each iteration looks something like this:

1. **Master ⇔ Mapper** (master sends necessary parameters to mapper, mapper performs map & partition)
2. Master ⇔ Reducer (after all mappers have returned to master successfully, master invokes reducers with the necessary parameters)
3. **Reducer ⇔ Mapper** (after invocation, reducers communicates with the mapper to get the input before the shuffle, sort, and reduce step)
4. **Master ⇔ Reducer** (after all reducers have returned to master successfully, master contacts the reducers to read the output datafiles)

1. Alternatively, Reducer can send the output file data as part of the response of the RPC request in point 2 above. If you do this, then the master does not need to do a separate RPC for reading the output files from reducers.

Sample input and output files:

We will provide sample input and output files for each application to have uniformity in evaluation. You should ensure that your code runs successfully in the provided format of sample input files and generates the final output files in the same format as the sample output files.

Data/

├─ Input/

│ ├─ points.txt (initial points)

├─ Mappers/

│ ├─ M1/

│ │ ├─ partition_1.txt

│ │ ├─ partition_2.txt

...

│ │ ├─ partition_R.txt (R based on the number of reducers)

│ ├─ M2/ ...

│ ├─ M3/ ...

...

├─ Reducers/

│ ├─ R1.txt

│ ├─ R2.txt

├─ centroids.txt (final list of centroids)

Please have a look at two more test cases for your reference here.

Print Statements:

Please log/print everything in a dump.txt file (just like we did in assignment 2) for easy debugging/monitoring.

Print/Display the following data while executing the master program for each iteration:

1. Iteration number
2. Execution of gRPC calls to Mappers or Reducers
3. gRPC responses for each Mapper and Reducer function (SUCCESS/FAILURE)
4. Centroids generated after each iteration (including the randomly initialized centroids)

Deliverables

1. Submit a zipped file containing the entire code with the given directory structure.

Evaluation

Your assignment will be evaluated on the following criteria:

1. You must test the master with different numbers of mappers, reducers, centroids, and iterations.
2. You can either set up the mapper and reducer processes before running the master process, or you can allow the master process to spawn the mapper and reducer processes as and when required.

3. The outputs of
    1. Mappers
    2. Reducers

will be verified for different configurations.