Finding Similar Images to an Input Image in a Large Dataset with MapReduce

Musab Erayman
Computer Science
Ihsan Dogramacı Bilkent University
Ankara, Turkey
musab.erayman@ug.bilkent.edu.tr

Serhan Gürsoy
Computer Science
Ihsan Dogramacı Bilkent University
Ankara, Turkey
serhan.gursoy@ug.bilkent.edu.tr

Ömer Faruk Karakaya
Computer Science
Ihsan Dogramacı Bilkent University
Ankara, Turkey
faruk.karakaya@ug.bilkent.edu.tr

Ege Yosunkaya
Computer Science
Ihsan Dogramacı Bilkent University
Ankara, Turkey
ege.yosunkaya@ug.bilkent.edu.tr

I. Introduction

Throughout the years, improvements in technology reduced the cost of storage and increased the information gathering speed for everyone. These improvements now resulted highly big volume of data, gathered from billions of people at the same time. Hence, big data become something real for industry, which raised a flag for some people in the industry because even though computational power also increased in time, it is still not efficient for big data processing[1].

Big data contains different data types from all over technological fields. However, in this study, we only focused on graphical images and processing those images in order to find a meaningful result. Our aim was finding the similarity between one input image with the whole dataset which contains one million unique images. We used publicly available dataset as our similarity dataset. For similarity finding, we used GIST feature vectors. For general approach, we used MapReduce Framework and Hadoop which built on our AWS servers. We will discuss these in detail in following parts of our paper.

II. Approach

While creating our system, we developed several approaches and run some test with them to find most efficient among all of them for our metrics. However, even after our results we can't claim that *one* algorithm/approach projects the best result among them because there are several parameters we should think of and these parameters can become important from project to project and computer to computer.

In all approaches we used MapReduce as base framework [2]. Regardless, MapReduce can give different results according to their mapper and reducer implementations. In all approaches, we fixed the mapper count. We used 1000 mappers, each gets 1000 unique image GIST as batch input. We detailed our all approaches below.

A. MapReduce without Grouping

This approach was the first idea comes up on our minds because it relies on intuition without thinking problems beforehand. It was using MapReduce framework right away without doing any optimization. In this approach, each mapper outputs single dataset image and input image as value and mapper's ID as key. Therefore, it creates reducers as same count as mappers. And each reducer gets list as parameter and finds similarity value between value images using GIST feature vector. As trivial as it sounds, it is not efficient because creating reducer count is directly related to mapper count whereas in an optimized structure, we would like to have less reducer than mappers. Without diving into implementing this approach, we come up with a better solution in theory, hence we skipped implementing this and jumped on implementing that approach which we will introduce below.

B. MapReduce with Grouping

This approach is the improved version of part A. In this approach, rather than simply creating reducers with the same count as mappers, we are controlling the reducer count. Each mapper outputs pair image GISTs as value, one from input image and other one from dataset. As key value, we grouped results with an group parameter. i.e. if group

parameter is set to 100 then each mapper would result 100 groups with 10 image value in it, furthermore, this also results 100 reducer. Unlike part A, now we have total control over reducer size. In order to find best reducer size, we run some tests by changing the reducer count. Results of these tests can be seen in Tests section.

C. MapReduce with Content Type Based Grouping

This approach is more improved version of part A and slightly improved version of part B. In theory, general logic is the same as part B. We are again grouping images in our mapper phase. Yet, in this approach, rather than blindly grouping images, we decide to group them according to their content. If you are familiar with Image Processing, you would know that extracting the content of image is a challenging process. After some research, we came across with IBM Watson [3]. Which offers an API for Image content detection. Approach works as follows, your input image is used as input image for Watson as well. Returned content label used for grouping like we did in part B. i.e, input image label returned as *animal* from Watson's API. Then each mapper only groups input image pair with that label only. Therefore, redundant images will not considered by algorithm. For *animal* case, there is no need for checking similarity of *plants* since they share no similarity. For mapping the input keys, we decide to use an upper bound for reducers as well. In our local tests, we successfully integrated this Watson support in our code. However, this requires a labelled dataset which we didn't have. IBM Watson has a API query limit which was less than our dataset image count. Hence, we dropped this approach for this research. Regardless, in theory, this approach seems like the best, if you have labelled dataset and good Deep Learning machine which successfully identifies the image content.

III. DATASET

Our dataset is The MIRFLICKR-1M open evaluation project consists of 25000 images downloaded from the social photography site Flickr[4] . MIRFLICKR-1M is Offered by the LIACS Medialab at Leiden University, The Netherlands Introduced by the ACM MIR Committee in 2008 as an ACM sponsored image retrieval evaluation [5]. MIRFLICKR-1M imageset does not contain any duplicate images. Imageset also contain various visual descriptors of all images such as histogram descriptors, texture descriptors, and feature descriptors. We used GIST descriptors for this project which commonly used for detecting similar scenes. For given image, a GIST descriptor is computed by[6]:

1. Convolve the image with 32 Gabor filters at 4 scales, 8 orientations, producing 32 feature maps of the same size of the input image.
2. Divide each feature map into 16 regions (by a 4x4 grid), and then average the feature values within each region.
3. Concatenate the 16 averaged values of all 32 feature maps, resulting in a 16x32=512 GIST descriptor.

IV. ENVIRONMENT

As environment we used Amazon Web Services. There were three services we used for the project.

a) AWS EC2 Service

AWS Elastic Compute Cloud or EC2 for short provides virtual computing environments known as instances[7].

We used EC2 instances for data transfer. The instances were linux machines with ssh access and internet connection. We developed a script to automate creating EC2 instances, setting up Python and the environment, uploading another script and running the uploaded script. The aim of the uploaded script was downloading a part of the data, unzipping the zip file and start uploading the unzipped data to the AWS S3 Service. We used boto library in Python for programatically access and run commands in the instances.

b) AWS S3 Service

S3 Service was our data storage in the project. We uploaded our gist feature vectors and images to S3 Buckets and our MapReduce cluster accessed to the data from the S3 buckets. Moreover, we used S3 buckets to upload our jar files to make them accessible to our MapReduce cluster.

c) AWS Elastic MapReduce Service

Elastic MapReduce Service provides a cluster with Apache Zookeeper for cluster management and Hadoop MapReduce installed. The cluster accepts job as a jar file from our S3 service. After getting the jar file from the bucket it starts its process and gives log files to track the process. As workflow, after we uploaded all our data to the S3 buckets, we write the map and reduce functions in the form that MapReduce wants, then compiled to a jar file. Then, uploaded to S3 and start the job. We used 20 Machines for the project and the specifications of the machines were:

- ➔ 4x 2.4 Ghz Intel Xeon Processor
- ➔ 16GB RAM

V. Final Algorithm

A. Similarity Metric

As a metric for image similarity we use euclidean distance between input image and target image GIST feature vectors. Using euclidean distance on GIST feature vectors is not an uncommon technique for web scale image search [8].

B. MapReduce Algorithm

MapReduce Algorithm - Mapper

```
Mapper:
  counter = 0
  for each GIST vector in database:
       reducerID = counter % ReducerSize
       counter += 1
       Generate <key= reducerID,
value=(query GIST, target GIST) >
```

MapReduce Algorithm - Reducer

```
Reducer:
<key= reducerID , valuelist=[(query gist,
target gist)] >:
  In the value list:
  For each (query gist, target gist) pair:
       calculate similarity S between query
gist and target gist
        if S > Threshold Similarity:
             Generate <key= target gist,
value = S>
```

C. Algorithm Analysis

Communication cost analysis:

N: Number of Images
g: Number of groups
Replication Rate: 1
Reducer Size: N/g
# of Reducers: g

Communication Cost: 2N

VI. Tests

A. Application Tests

Same scene with different contrast euclidean distance between GIST descriptors are 0.06. (Figure 1 and Figure 2)

250964.jpg

Figure 1

250023.jpg

Figure 2

Similar scenes with similar angles. Euclidean distance between GIST descriptors are 0.28. (Figure 3 and Figure 4)

398661.jpg

Figure 3

235003.jpg

Figure 4

Less similar scenes with different color sets.Euclidean distance between GIST descriptors are 0.32 (Figure 5 and Figure 6)

235000.jpg

Figure 5

384424.jpg

Figure 6

B. Time Tests

We ran our application with different parameters and measured elapsed time for many times.(Figure 7) The obtained results showed us that MapReduce paradigm results a better performance for time as we expected thanks to the parallelism and distribution of the data and task. It is seen that classical serial computing takes much more time than the MapReduce computing. Relevant measured time metrics are given below:

Total time spent by all map tasks(TTAM): The total time taken in running all the map tasks, including speculative tasks which are record reader, map, combiner, and partitioner. This is a cumulative number of wall clock time of all map tasks.

Total time spent by all reduce tasks(TTAR): The total time taken in running all the reduce tasks, including speculative tasks which are shuffle, sort, reduce, and output. This is a cumulative number of wall clock time of all reduce tasks.

Elapsed wall clock time with MapReduce(TMR): The total wall clock time, real time, of MapReduce computing, including all tasks of framework.

Elapsed wall clock time with serial computing(TSC): The total wall clock time, real time, of serial computing.

VII. RESULTS

A. Single Process vs MapReduce

We tested the speed up by running a similar process in a local computer with no parallelism. In that case, total running time (TSC) was approximately 37.5 hours. To compare the MapReduce version with linear run, we run our cluster with same task and it (TMR) took approximately 14 minutes. It was a significant speedup for our case. To investigate further, in the MapReduce total time spent by all map tasks (TTAM) were approximately 20.02 hours and all time spent by the reducer tasks (TTAR) were approximately 6.07 hours, with a combined 26.1 hours. However TMR was only 14 minutes. Thus, it shows that map tasks and reduce tasks ran in parallel.

B. MapReduce with Different Groupings

We used different key groupings in the tests, In the figure below X axis represents the number of times the reduce function have been called, and of course the number of groups. Because of the map task times were almost the same,

we focused on total reduce task times in grouping analysis (Figure 7).

Note that, in the tests, framework decides how many tasks there should be to call reduce functions. In our case, there were 75 reducer tasks. For instance, in the 1000 group case the valuelist for each reduce function was 1 million / 1000 = 1000 feature vector pairs. Thus, there were 75 processes which were calling 1000 reduce functions and each call processed 1000 feature vector pairs.

The least time in the reduce tasks were in the 50 group case. In that case there were 1 million / 50 = 20.000 feature vector pairs have to be processed in a single reduce function calls. However, because of there are only 50 calls to make, every reduce task would run only 1 reduce function. This case was the optimal and had the least wallclock time to finish as well. After the 50 group point, for less group numbers, the parallelism level were same, every reduce task run only 1 reduce function but the valuelists grows. Thus, in every call there were more feature vector pairs to process.

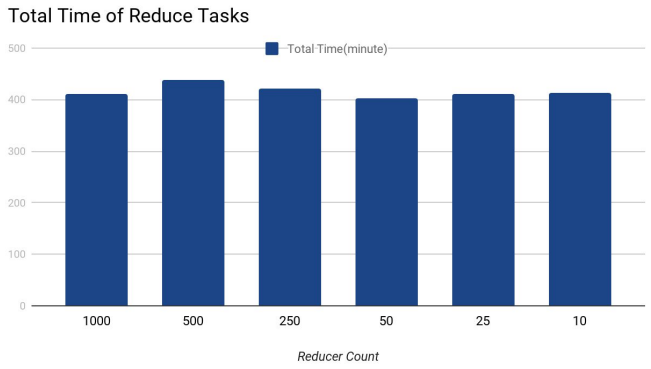

Figure 7: Reducer Group vs Total Time

References

[1] J. Yan, Y. Meng, L. Lu and L. Li, "Industrial Big Data in an Industry 4.0 Environment: Challenges, Schemes, and Applications for Predictive Maintenance," in *IEEE Access*, vol. 5, pp. 23484-23491, 2017. doi: 10.1109/ACCESS.2017.2765544

[2] Lämmel, R. (2008). "Google's MapReduce programming model — Revisited". Science of Computer Programming. 70: 1–30.doi:10.1016/j.scico.2007.07.001.

[3] IBM Watson [Online] Available: https://www.ibm.com/watson/services/visual-recognition/ [Accessed: 08-May-2018].

[4] MIRFLICKR Download. [Online]. Available: http://press.liacs.nl/mirflickr/mirdownload.html. [Accessed: 08-May-2018].

[5] "The MIRFLICKR Retrieval Evaluation," MIRFLICKR Download. [Online]. Available: http://press.liacs.nl/mirflickr/. [Accessed: 08-May-2018].

[6] "Modeling the shape of the scene: a holistic representation of the spatial envelope," Daniel Jackson. [Online]. Available: http://people.csail.mit.edu/torralba/code/spatialenvelope/. [Accessed: 08-May-2018].

[7] What Is Amazon EC2? - Amazon Elastic Compute Cloud. *Docsawsamazoncom*. 2018. Available at: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html. Accessed April 3, 2018.

[8] M. Douze, H. Jégou, H. Sandhawalia, L. Amsaleg, and C. Schmid, "Evaluation of GIST descriptors for web-scale image search," Proceeding of the ACM International Conference on Image and Video Retrieval - CIVR 09, 200