ECE795 Final Project Report
Edge Detection Using Spark and Comparing the Computing Times of Different Approaches

Ruijie Yin
Division of Biostatistics
Department of Public Health Sciences
Miller School of Medicine
University of Miami
rxy114@miami.edu

Abstract

Important information can be extracted from the edges of images and it can be further used by higher-level algorithms such as those in computer vision. This project aimed at using Spark to perform the famous and widely used edge detection technique, Canny edge detector, to perform edge detection on selected images downloaded from the internet and count the number of pixels that could be considered as edges. The computing time was compared between saving and not saving the intermediate results and results shows that saving results cost extra time on Spark.

Keywords: edge detection, spark, computing time

1. Introduction

Edges are significant local changes of intensity in an image, they typically occur on the boundary between two different regions in an image. Edge detection is an image processing technique for finding the boundaries of objects within images. It works by detecting discontinuities in brightness; Also, important features can be extracted from the edges of an image (e.g., corners,lines,curves) and these features are used by higher-level algorithms in areas such as computer vision and machine vision. Common edge detection algorithms include Sobel, Canny, Prewitt, Roberts, and fuzzy logic methods.

The Canny edge detector was used in this project; it is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. It was first developed by John F. Canny in 1986, and the method has been ameliorated throughout the year. Like many other methods mentioned above, it is a technique to extract useful structural information from different vision objects but at the same time dramatically reduce the amount of data to be processed. Among the edge detection methods developed so far, Canny edge detection algorithm is one of the most strictly defined methods that provides good and reliable detection. Owing to its optimality to meet with the criteria for edge detection and the simplicity of process for implementation, it became one of the most popular and widely used algorithms for edge detection.

The process of Canny edge detection algorithm can be broken down to 5 different steps:1.Apply Gaussian filter to smooth the image in order to remove the noise. 2.Find the intensity gradients of the image.3.Apply non-maximum suppression to get rid of spurious response to edge detection.4.Apply double threshold to determine potential edges.5.Track edge by hysteresis: finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.[1]

The goal of this project is utilizing Spark, which is a widely used open-source distributed general-purpose cluster-computing framework to perform edge detection on selected images, and compare the average time consumed when counting how many pixels can be treated as edges between saving and without saving the edge images. The report is organized as follows: detailed descriptions of the approaches, methods, and implementation of the methods are in Section 2; instructions on how to compile and run the program on Google Cloud Platform are in Section 3; Results of the computing time using different methods are shown in Section 4; a

conclusion and discussion of what has been found and discovered is presented in Section 5; a list of Python programs needed to implement the methods is in Section 6.

2. Methods and Implementation

There are a total of 125,436 pictures to be performed edge detection on; They can be downloaded from the database using 'wget.download' function in Python and upload them to Google Cloud Storage using blob:

```
def upload_blob(bucket_name, source_file_name,
destination_blob_name):
    filename = wget.download(source_file_name)
    bucket = storage_client.get_bucket(
    bucket_name)
    blob = bucket.blob(destination_blob_name)
    blob.upload_from_filename(filename,
    content_type='image/jpg')
    os.remove(filename)
```

As an intuitive example to show the effect of edge detection, the original image (256aef4239228950.jpg) is displayed as in Figure 1. After edges had been detected, the corresponding edge image is shown in Figure 2.

Figure 1. Original Image

To detect edges, save the edge image and count pixels as edges on Spark, we will first need to configure Spark using:

```
sconf=SparkConf().setMaster("yarn").setAppName(
"Edge_Detect")
sc = SparkContext(conf = sconf)
```

We use cv2.canny function from opencv-python library to detect the edges, by convention the original picture is read in as a numpy array type object with each entry a pixel representing a color; for example, the numpy array of image (256aef4239228950.jpg) is:

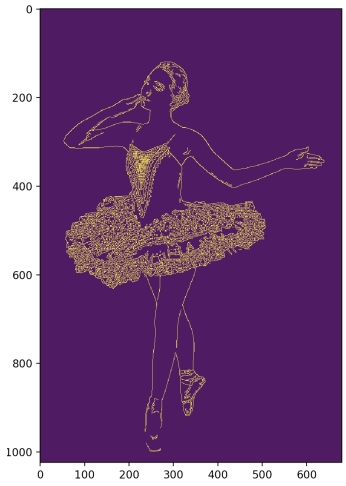

Figure 2. Edge Image

$$\begin{bmatrix} 233 & 247 & 253 \\ 231 & 247 & 253 \\ 229 & 248 & 253 \\ & \dots & \\ 239 & 251 & 253 \\ 239 & 251 & 253 \\ 239 & 251 & 253 \end{bmatrix}$$

After edge detection on this very same image, the entries in the numpy array of the edge image becomes binary, with only values of 0 and 255, and '255' represents the edge in the image:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 255 & 0 \\ 0 & 0 & 0 \\ & \dots & \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 255 \end{bmatrix}$$

Hence, the number of edges in the image can be obtained by simply counting the total amount of '255' in the array; before counting, we would better transform the edge image (call it "edges") from numpy array type to RDD type (call it "edge_rdd"); RDD stands for Resilient Distributed Dataset where each of the terms signifies its features; Resilient means it is fault tolerant by using RDD lineage graph (DAG), hence, it makes it possible to do recomputation in case of node failure; Distributed indicates that datasets for Spark RDD resides in multiple nodes; Dataset is the records of data that you will work with; it is the fundamental unit of data in Spark and makes data

processing faster; most Spark programming consists of performing operations on RDDs:

```
edges = cv2.Canny(img, 100, 200)
edge_rdd = sc.parallelize(edges)
```

After the transformation, the entries in the RDD are either 'array([0], dtype=uint8)' or 'array([255], dtype=uint8)'.

We proceed to save the RDD to the bucket as text files; Notice that each text file will be automatically partitioned into a few text files by default if the following saveAsTextFile function is used:

```
edge_rdd.saveAsTextFile('gs://<bucket_name>/
file_name)
```

For instance, the text file of RDD of our example image are partitioned into four parts, and are stored under the same folder (Figure 3):

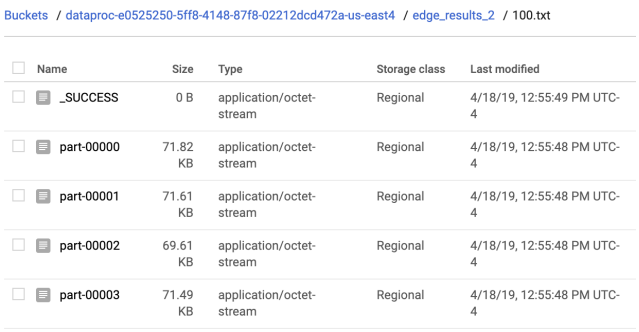

Figure 3. Partitioned text files

We can, in another way, save the numpy array of edge image as image files (see Figure 4), and the computing time will also be compared with saving them as text files, as our secondary goal; in this case, we would first save the edge image as a temporary file and then upload them to the bucket using blob, just like how we upload the downloaded image to the bucket:

```
bucket = storage_client.get_bucket(bucket_name)
    destination_blob_name = "edge" +
    image_name[i]
    with TemporaryFile() as gcs_image:
        edges.tofile(gcs_image)
        gcs_image.seek(0)
        blob = bucket.blob("edge_results/" +
        destination_blob_name)
        blob.upload_from_file(gcs_image)
```

Number of edges in the images are computed by iterating through the RDD or numpy array of edge image. If the edge images are stored as images, we can then read them back (called 'img') and turn into numpy array directly by using:

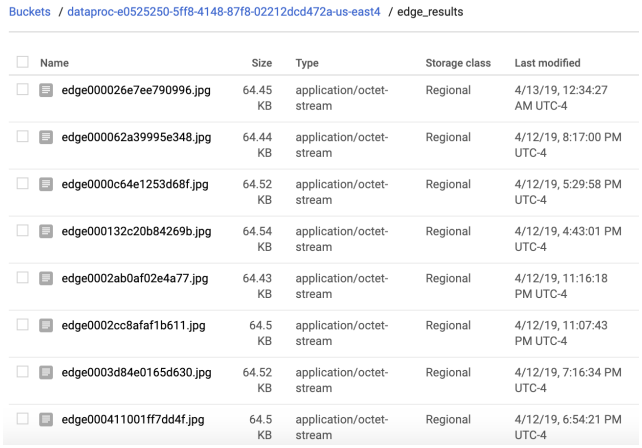

Figure 4. Saving edge images as image files

```
gcs_url = 'https://storage.cloud.google.com/ \
 <bucket_name>/edge_image_name'
resp = urllib.urlopen(gcs_url)
img = np.asarray(bytearray(resp.read()),
dtype="uint8")
```

If, however, the edge images are stored as text file, they are still in RDD type when reading them back:

```
edge_rdd=sc.textFile('gs://<bucket_name>/
text_file)
```

When obtaining the computing time for each action while saving the edge image, we use the "time" function from the time library; a pesudo code is shown as below:

total_computing_time = 0;
total_detect_time = 0;
count = 0;
while *not all images have been processed* **do**
    start_time=current time;
    read the image;
    detect edges;
    end_time_detect=current time;
    total_detect_time +=
     end_time_detect-start_time;
    count edges;
    save results;
    end_time = current time;
    total_computing_time += end_time - start_time
end

Hence, the average computing time for detecting edges on each image is total_detect_time/# of images and the average computing time for detecting edges on each image and at the same time saving the results to the bucket is total_computing_time/# of images; When not saving the results:

```
total_computing_time = 0;
total_detect_time = 0;
count = 0;
while not all images have been processed do
    start_time=current time;
    read the image;
    detect edges;
    end_time_detect=current time;
    total_detect_time +=
      end_time_detect-start_time;
    count edges;
    end_time = current time;
    total_computing_time += end_time - start_time
end
```

And the average computing time for detecting edges on each image can be obtained in the above same way.

3. Instructions

Google Cloud Platform and pyspark are used in this project; in the Dataproc section, create a cluster (Figure 5) using 4 vCPUs and 15 BG memory for both the master node and the worker nodes, set the number of worker nodes to be 2 as default; Check the "component gateway" checkbox so that we will be able to monitor the status of the cluster using YARN Resource Manager.

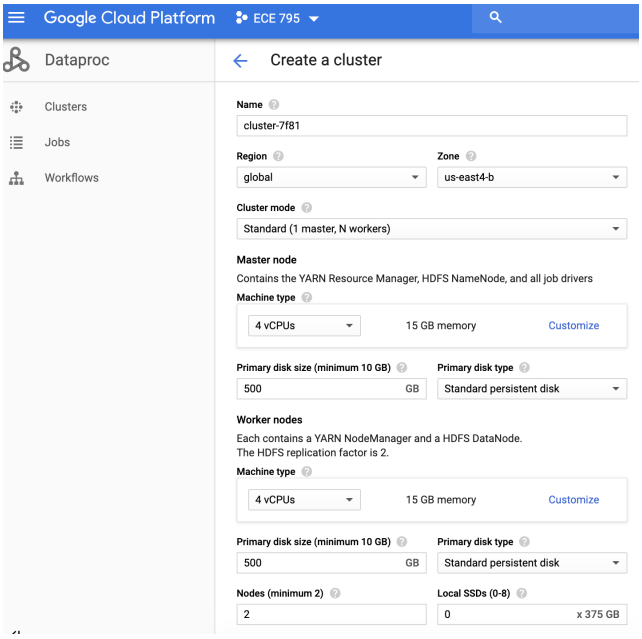

Figure 5. Create a cluster.

After the cluster is created, a few required Python libraries has to be installed on the master node and both of the two worker nodes, by SSH into the master and worker nodes in Compute Engine-VM instances. Especially, given that user do not have permission to do 'pip install', 'sudo' install of all libraries are performed;Pip for Python should be first installed using:

```
curl https://bootstrap.pypa.io/ \
get-pip.py -o get-pip.py
sudo python get-pip.py
```

Then modules: wget, pandas, gcsfs, opencv-python, pyspark and numpy will also be pip installed (Figure 6) using the following code, and in :

```
sudo pip install <module_name> --user
```

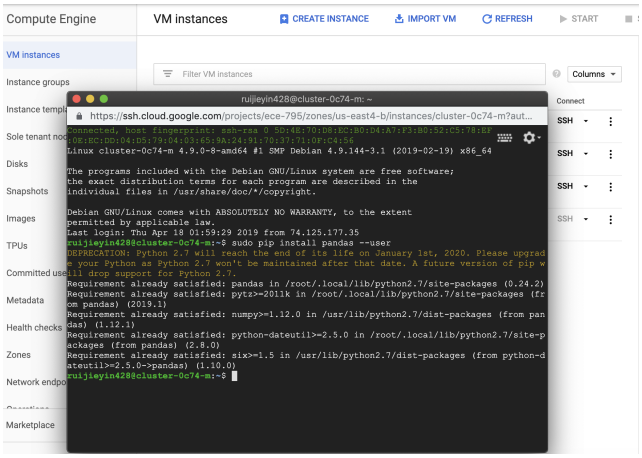

Figure 6. Install required packages.

Since library google-cloud is deprecated, it can be installed using:

```
sudo pip install --upgrade \
google-cloud-storage --user
```

Once installed all the required modules, manually upload the listing of urls of images ('urls.csv') to the Google Cloud Storage bucket that was linked to the cluster. Submit the Python file 'download_image.py' as a pyspark job in Dataproc (Figure 7), this will download all the 125,436 pictures and upload them into bucket. All pictures are stored in their original form (JPG) and can be read by GCP's default setting; for instance, a picture's relative path is:

```
gs://<bucket_name>/pic.jpg
```

and the absolute path is:

```
https://storage.cloud.google.com/<bucket_name>/
pic.jpg
```

Next, submit the Python file 'read_save_count.py' as a pyspark job in Dataproc, it will read each and every

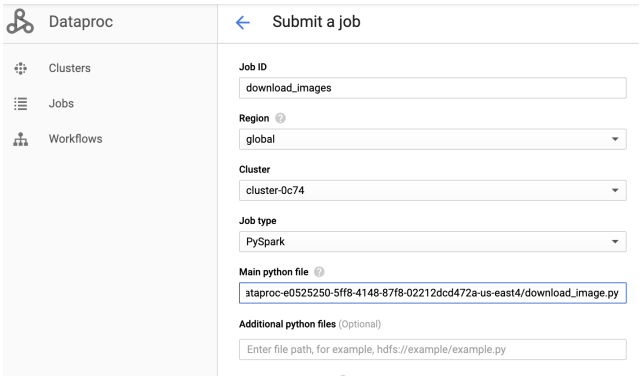

Figure 7. Submit a pyspark job.

original pictures in the bucket, perform edge detection on them, count how many pixels are considered as edges, save the edge image back to the bucket under a new folder and will print the average time consumed for processing each image when all processing all the images.

Finally, to compare the average time consumed for only read in the original picture, perform edge detection and count how many pixels are considered as edges directly without saving the edge image, submit the Python file 'read_count.py' as a pyspark job in Dataproc, as it will perform the above actions and print the average time consumed accordingly.

Until this point, one should be able to get the average time for the two methods. Detailed explanation of Python code and configuration of Spark are listed as comments in the above mentioned Python files.

4. Results

After detecting the edges, when saving the results as text files to the bucket, the average computing time is around 4.1476 seconds per image, and it is approximately 0.1342 seconds without saving the results; it takes nearly 4 more seconds to save each result (see Table 1).

Comparison of Computing Time	
Actions	Avg. Computing Time
Detecting edges only	0.1342 seconds
Detecting edges and saving results as text	4.1476 seconds

Table 1. Comparison of Avg. Computing Time w. and w.o. saving the results as text file.

When saving the edge images as images (JPG), the average computing time rounded up to 0.5201 seconds per image, and is approximately 0.1362 seconds without saving the results; it only takes around 0.38 seconds more to save each result (see Table 2).

Comparison of Computing Time	
Actions	Avg. Computing Time
Detecting edges only	0.1362 seconds
Detecting edges and saving results as image	0.5201 seconds

Table 2. Comparison of Avg. Computing Time w. and w.o. saving the results as image file.

Comparing the computing time of reading in the original image, counting the number of pixels as edges and with and without saving the corresponding result as either image or text file, we get the following computing time for each method, as shown in Table 3.

Comparison of Computing Time	
Actions	Avg. Computing Time
Detecting edges, Counting # of edges	0.3117 seconds
Detecting edges, Counting # of edges, save results as image	0.4282 seconds
Detecting edges, Counting # of edges, save results as text	4.2613 seconds

Table 3. Comparison of Avg. Computing Time w. and w.o. saving the results.

5. Discussion and Conclusion

Results show that reading in the original image and detecting edges on the images takes about 0.13 seconds per image, and counting edges takes around 0.18 seconds per image on Spark. When saving the edge image back to the Google Cloud Storage as image, it takes another 0.38 seconds per image; however, saving the edge image as text files can take, on average, 4 seconds per image, the reason why saving the edge images as text files is much slower than saving as images using "saveAsTextFile" function, and in fact saveAsTextFile will always be slower than any other memory operation is because by definition it is splitting data onto disk, and more frequently via HDFS with all the constraints linked to HDFS (block size, replication, etc.), this means RDD file is first partitioned and then write one file per partition to the bucket and by

default, the computation is distributed; we can, however, coalesce them to 1 partition but is not recommended to be used; users should be careful when calling coalesce with a real small number of partitions, for it can cause upstream partitions to inherit this number of partitions, and the parallelism of upstream stages will be lost to be performed on a single node; Besides, coalescing the RDD to force Spark to produce a single file also limits the number of Spark tasks that can work on the RDD in parallel.

It is expected that simply performing reading in the image, doing edge detection and counting how many pixels can be considered as edges along takes less time than performing all of these actions but also at the same time saving the results, no matter how we choose the form of the saved results, per image; It is clear that saving the results back to the storage takes some time from the results, as it is shown in the results.

6. List of Programs

A list of mentioned programs and their corresponding short description is shown in Table 4.

Listing of Programs	
Program Names	Descriptions
download_image.py	download pictures, upload pictures to bucket
read_save_count.py	read image, edge detection, count edges, save results
read_count.py	read image, edge detection, count edges

Table 4. List of python programs

7. Reference

[1]Wikipedia contributors, "Canny edge detector," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Canny_edge_detectoroldid=892496671 (accessed April 16, 2019).

[2]Marr, David, and Ellen Hildreth. "Theory of edge detection." Proceedings of the Royal Society of London. Series B. Biological Sciences 207, no. 1167 (1980): 187-217.

[3]Davis, Larry S. "A survey of edge detection techniques." Computer graphics and image processing 4, no. 3 (1975): 248-270.