Project Spark Report

Alexandros Matakos

Student ID: 015538460

1

For task 1, pyspark's *statcounter* library was used. The *statistics()* function takes the data as input and calculates the count, min, max, mean and variance. This function is also used in the median finding algorithm later to calculate count, mean and variance in each iteration, and so a boolean variable *with_min_max* is included to avoid unnecessary computations of the min and max. Values for this task:

$$min = 1.01311593426e{-}06$$

$$max = 97.999993121$$

$$mean = 42.060411373468106$$

$$variance = 193.2777614391939$$

2

The median is calculated through an implementation of the binmedian algorithm. This algorithm works well with data that are both unique or have duplicates. The algorithm starts by splitting the data evenly into a user-defined number of bins. It then begins to count the number of elements in each bin, from left to right, until it has amassed more than half of the original data's elements. The last bin which exceeded that threshold is bound to contain the median, and the algorithm recurses on that bin. Mallows et. al. result that the median is at most one standard deviation away from the mean is used, and this is applied at each iteration before recursing on the new bin. If the data is odd sized, then the algorithm stops once the median is found. If the data is even sized, then the algorithm finds the "left" median, and then repeats the last step to find the "right" median. The algorithm then returns the average of the two medians. Documentation is provided as comments in the code which explain each step.

Pyspark's *rdd.histogram()* function is used to divide the data into bins at each step, and this turns out to work quite well in a MapReduce framework like

Spark. The data are never accessed directly during the runtime of the algorithm. Instead, they are manipulated/filtered implicitly via the bins' endpoints.

I had initially implemented the quickselect algorithm, and the runtime was over 6 hours. This runtime can be severally improved by using dataframes, but I felt this was outside the scope of this course and opted for the binmedian algorithm instead. My implementation ran in about 24 minutes with 10 cores using 20 bins for each iteration. It seemed to perform similar with 10 bins, but I did not optimize further in order to save resources for other students since 24 minutes seemed fine already.

$$median = 42.0233594627$$

3

The data we are given for the project seems to come from a uniform distribution without duplicates. Since the floating point precision practically guarantees that there will be no duplicates, it would not make sense to calculate the mode of this dataset. One case where it would make sense to calculate the mode would be if the data contained integer values, with at least some of them containing duplicates.

Since we are using a MapReduce framework, I would calculate the mode as follows: First partition the data row-wise, and apply a *map* step to each element of the array, $x \to (x, 1)$. Then apply a *reduce* step to count the 1's for each x. Finally combine the results from each partition and construct an array containing key-value pairs of (x, n) for each unique x in the data, with n = number of occurences. Sort that array descending on n, and choose the the first pair. The x of that first pair is the mode. This can be very efficiently implemented in pyspark, since up to the final point where we are choosing the first pair we can efficiently do everything with RDD manipulations. An example is shown in the code.