Project 1
Distributed Data Infrastructure

Walter Genchi 014961054

November 27, 2018

1 Problem 1

The idea behind this exercise is to divide the data in bags (n_of_bags is the a-priori parameter to be chosen) and determine the bag where the median is located. The task will be thus to sort only that bag.

There is a trade-off in the choice of n_of_bags. From one hand, for large n_of_bags, the sorting task on the median bag (see step 8) will be simplified, but the Spark operations (see step 5 and 6) will be affected by an increased number of keys. On the other hand, for small n_of_bags, the sorting task will be slower and Spark operations faster.

Moreover, the bag-assigning strategy (see step 3) is effective for uniformly distributed data (i.e. all bags have a similar number of elements), but lacks generality for other data distributions (e.g. highly skewed distributions).

Operations can be summarized as:

1. Reading the file data-1.txt and converting the numbers to float (using the map function)
2. Make data persistent, in order to make further actions on data (i.e. max, min and count) more efficient in terms of time and memory consumption.
3. Every number n is mapped to his bag (using the map function) by 3 steps:
   - Normalization, i.e. $x = (n - min)/(max - min)$
   - Assigned bag = $int\,(x * \text{n\_of\_bags})$
   - FINAL RESULT: n -> (Assigned Bag, n)
4. Computing the median position (by definition of median the value in the middle)
5. Creating the bags, i.e. get something like

$$[(0, [\text{numbers in 0 - 0.0999}]), \ldots, (10, \text{numbers in 0.9 - 1}])]$$

by using groupByKey. Note that reduceByKey is usually preferred for grouping + aggregating tasks (e.g. word count) due to better performances on large dataset. However, since we require the grouped bags for future operations, we perform first the grouping (groupByKey) and then the size (mapValues(len)) of the bags.

6. Computing the bag size, i.e. get something like

   $$[(0, [\text{how many numbers in 0 - 0.0999}]), \ldots, (10, \text{how many numbers in 0.9 - 1}])]$$

   by using mapValues(len).

7. Determining the bag which contains the median, by computing cumulative sums of the size of each bag, starting from the first bag.

8. Sorting the median bag and writing the result.

2 Problem 2

The points I took into account while considering how to solve this problem were the following:

- A has much more rows (10^6) than columns (10^3).
- The product between three matrices has the associative property, i.e. $(AB)C = A(BC)$. Thus, I have preferred to perform first A^TA, since the result is a 10^3x10^3 matrix, which is easier to store in memory than the AA^T matrix, which has size 10^6x10^3.
- While using map function in Spark, I noticed that it is easy to iterate over RDD rows (by using a lambda function), but it is hard to iterate over RDD columns (unless you transpose the matrix, which is probably not feasible because of its dimension).

Compute $B = A^TA$

In order to use map (transformation) and reduce (action) in Spark, I exploited the property of outer product in matrices.

Given matrix

$$A = \begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix}$$

then

$$B = A^TA = \begin{pmatrix} a & c & e \\ b & d & f \end{pmatrix} \begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix} = \begin{pmatrix} a^2 + c^2 + e^2 & ab + cd + ef \\ ba + dc + fe & b^2 + d^2 + f^2 \end{pmatrix}.$$

This product can be computed as the sum of the outer products of the rows of A, i.e.

$$\begin{aligned}\begin{pmatrix} a & b \end{pmatrix} \otimes \begin{pmatrix} a & b \end{pmatrix} + \ldots + \begin{pmatrix} e & f \end{pmatrix} \otimes \begin{pmatrix} e & f \end{pmatrix} = \\ \begin{pmatrix} a^2 & ab \\ ba & b^2 \end{pmatrix} + \ldots + \begin{pmatrix} e^2 & ef \\ fe & f^2 \end{pmatrix} = \\ \begin{pmatrix} a^2 + c^2 + e^2 & ab + cd + ef \\ ba + dc + fe & b^2 + d^2 + f^2 \end{pmatrix} = A^T A\end{aligned}$$

In Spark this means that the operation (np.outer) is always computed on the same row and thus very efficient with map.

After that, the intermediate result is one RDD, where each element k is a np.array of size 10^3x10^3, obtained as np.outer(k,k). This is the input of the reduce function, which adds the 10^6 matrices (each with dimension 10^3x10^3) through the commutative operator add.

This solution works well with the sample dataset (10^3 rows: $\sim$ 1 second for map and $\sim$ 3 seconds for reduce) and with a bigger sample (10^5 rows: $\sim$ 2 seconds for map and $\sim$ 56 seconds for reduce), which I generated from the original dataset.

However, when testing on the whole matrix (10^6 rows), I got an error on the reduce operation. That is the reason why I treid to implement a different code, which first assigns an index to each element of the RDD, then maps each index to one key (the set of keys is "smaller" than the set of indices) and finally uses reduceByKey to add the matrices with the same key. This simplifies the computations in the reduce part and acts as a "combiner" in MapReduce. Finally, the results are again summed up using reduce(add), this time with less matrices. This second solution did not provide satisfactory results on the whole matrix (10^6 rows), due to errors in the reduce(add) function.

Compute $W = AB$

This product was implemented (using map in Spark) by computing the numpy dot product between each row of A and the W matrix, which results as a 1x1000 vector. The dot producrt between the first row of A and matrix B constitutes the first row of the final matrix W. The results are finally sorted by key, since RDD does not guarantee the order of the computations in the output.