Task 1

For this task my plan was to split the input data into classes of the numbers rounded to n decimal places. Since the data set contained a roughly uniform distribution of numbers I could split the problem into 1 / 1000 th of its size by specifying a class to be the value rounded to one decimal point, i.e. the classes are 0.0, 0.1, 0.2 etc. After mapping the input into an RDD of pairs of (class, value), it is simple to just group them by their class which is the key in the RDD. After the elements are grouped, they can be roughly ordered by sorting them by the value of their keys.

After the grouped values are sorted we need some information to locate the median in the RDD. What we need to calculate is the total count to find the midpoint, which is the index of the median and then we will need the counts by each key to start counting how many elements have been passed as each key is processed. After this information has been gathered, we can collect the keys into an array from the RDD.

When the keys of the RDD are collected into an array we can iterate over all the keys. For each key we can get the number of elements from the map that we stored earlier from the counts of each key. Now for each key we can check if we are in the region where the median is located. This is done by summing up the elements that have been seen thus far.

After the threshold for the median's location has been reached we can evaluate firstly whether the median is a single value or the average of two values. If it is a single value, the index for the median in the current class can be easily calculated and the median can be found by sorting the array and accessing the element at that index.

If the median is the average of two values it can be more difficult to calculate, since the median can span two different classes. So if the lower value of the median is the last element of the current class we know that this represents one half of the median and the first element of the next class will be the other half. However, if both parts of the median are located in the current class it can be trivially calculated.

This solution can of course vary in terms of performance and depending on what kind of numbers are calculated, the parameter for what kind of rounding the classes will use might vary. If there are not enough distinct classes, we might want to consider splitting it more and if there are too many classes, we might want to reduce the number of classes. Furthermore, to completely function, there would have to be some precaution for the case, where some of the classes are too large, i.e. if the class we are interested can't be sorted because it is too big, there would have to be some way to further split that class up to be properly processed.

The median this algorithm found was 50.0068525415.

Task 2

For this task I first started by trying to implement the entire matrix multiplication by myself, by mapping each element of the matrix into an element of type ((row, column) value) in an RDD. Using this it is possible to easily multiply the original matrix by its transpose and after that multiply by the original again. But as expected, this solution does not even scale properly to the smaller sample size. Finding one row/column pair seemed to take about 0.8 seconds so the entire execution would take very long, but eventually it should reach the correct solution.

After this I found out about IndexedRowMatrices which apparently might be good for this kind of distributed matrix problems. Unfortunately it is not possible to multiply the IndexedRowMatrix by itself so I tried to create a two dimensional rdd that I tried to map by multiplying the IndexedRowMatrix with a DenseMatrix made up of a single row at a time. For this I also wrote a transpose function that was roughly based on the mapping I created in my original brute-force type solution. Using these Spark could do the matrix multiplications which I found quite weird since I couldn't access the results in any way since the RDD operations were nested so in the end I am not sure if the calculations were done really.

Finally I found the BlockMatrix class which is another distributed matrix in Spark that actually allows for multiplication with objects of the same class. For this I again constructed the normal matrix, this time I used first the IndexedRowMatrix and used its toBlockMatrix command to create the BlockMatrix. For the transpose I already had my function so I could get it also very easily. After I had created both of these matrices it was only a matter of doing the matrix multiplications using them.

Still even after using BlockMatrices the process seemed to be extremely slow for the larger data set so I am not sure if it would complete or not after running it for a long time. At least the nodes start receiving errors after the cluster moves onto the actual matrix multiplication phase but I have not let it run long enough to see if it reaches the solution or not. Maybe it would require the implementation of some of the distributed matrix multiplication algorithms or some linear algebra trick to actually do this calculation efficiently.

Also in general I think there should be some limit for the number of cores that can be used. I ended up using maybe 15 hours trying to do these exercises over 3 different days and in the end maybe 5 hours were productively used where I could iterate over my solutions and the rest of the time I was waiting for the one person who decided that they need 100 cores and ended up blocking the entire system for over an hour on multiple occasions.