Ivan Capin – Shishir Dubey – Master Big Data – Ensai - 2016

Privacy-Preserving Data Publishing

Practical Work : Implementing

Mondrian and the Laplace Mechanism

Part one: Mondrian

Program structure

The java project *MondrianPartition* contains the following classes:

- The Tuple.java class which contains 2 qids as integers and one sensitive data as a string;
- The Partition.java class which represent equivalence classes;
- The LaunchMe.java class which contains the `public static void main(String args[])` method and the Mondrian method `private static List<Partition> Mondrian(Partition partition)` which is the center piece of part one.

Equivalence class

The Partition.java class represents the equivalence class and is created with:

```
public static Partition CreateFormTupleList(List<Tuple> tuples)
```

A partition contains the tuple list it was created with. It also contains the list of the values taken by each QID, without duplicates.

Program Work-flow

1. Random tuples are created and stored in a partition class;
2. This initial partition is persisted in a file called *Initial_data.csv*;
3. Mondrian, run on the initial partition and returns a list of partitions;
4. This list of partition is persisted in a file called *sanitized data with N-anonymity.csv* where N is the privacy parameter. This file contains the sanitized data.

Mondrian

The method used in Mondrian to determine which dimension to use to split a partition in two is a greedy one: we take the dimension that contains the largest number of elements.

See in Partition.java: `public List<Partition> GreedySplit()`

Then, if the cardinality of this dimension is greater or equals to twice the privacy level, we perform the split and recursively call Mondrian on the two issuing partitions.

Partition persistence

Partition.java has two methods to persist itself in csv files:

- `public void Write(BufferedWriter bf)` merely writes all the tuples it contains. It is called at the beginning of the program to write the initial values;
- `public void WriteSanitizedPartition(BufferedWriter bf)` can writes a partition as an equivalence class. That means that a given QID Q has the same value for all sensitive data in this partition. This value is actually a list of values: all the values taken by Q in the tuples of the partition.

How to run the program?

If launched from the command line with arguments, the argument should be: *datasetSize*, *minQid1*, *maxQid1*, *minQid2*, *maxQid2*, *privacyParam*.

If the program is run without arguments, default values will be set (see `private static void DefaultInitialisation()`).

Example 1: java LaunchMe

Example 2: java LaunchMe 1000 100 200 1000 2000 2

Then, open the two csv files: *Initial_data.csv* and *sanitized data with N-anonymity.csv* to see the result.

Part 2: Differential Privacy

Programme structure

The java project *DiffPrivacy* contains the following classes:

- The Laplace.java class which is able to generate a random noise based on the Laplace distribution. As required in the assignment, this class throw an exception when it is running out of epsilon unless the test mode is turned on;
- The LaunchMe.java class which contains the `public static void main(String args[])` method.

Question4

The following command: *java LaunchMe question4* creates the *question4.csv* file that contains 1000 numbers generated with: `laplace.genNoise(10, 5.0)` (meaning: sensitivity = 10 and epsilon = 5,0).

Loading this file in R Studio and comparing it with a Lapace distribution with coefficient 2.0 gives the graph called *compared Laplace Density* (see below). It clearly shows that Laplace.genNoise generates the good probability density.

Question 5

The following command: *java LaunchMe question5* creates the *question5.csv* file which contains the data required in question 5. These data loaded in R studio are plotted with an horizontal lines that represents the real value in red +- 10 % in blue. The results is the graph called *Average of COUNT request* (see below).

It shows it takes more than 1000 perturbations to be able to guess the real value with a 10 % margin. We could say it a pretty good result as far as sanitisation is concerned. However, we also notice the huge variability of the request SUM. In order to be used by a statistician, size of the dataset should be increased.

Question 6

In this question we are going to perform SUM queries.

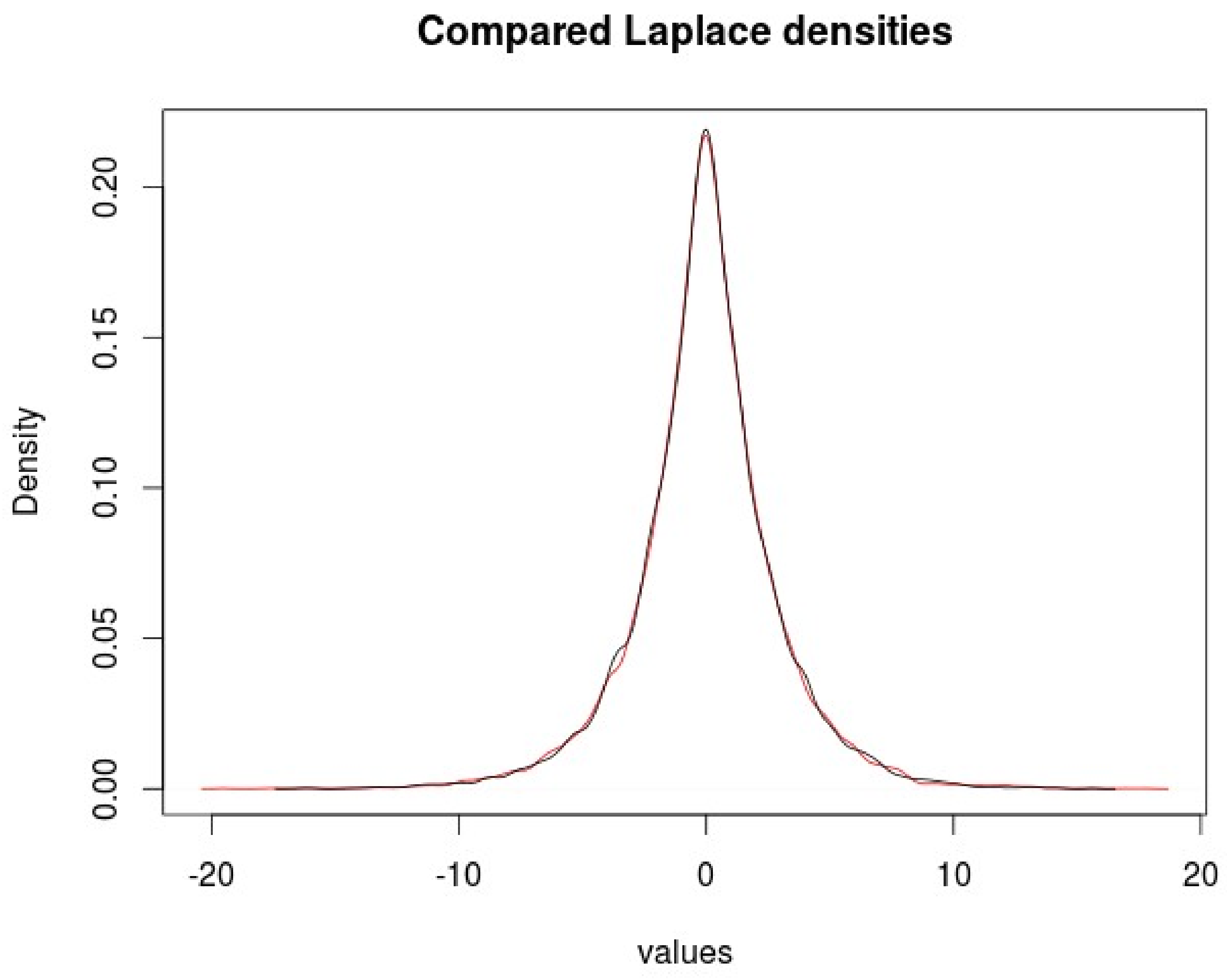
The following command: *java LaunchMe question6*

1. prints out the average error for 1000 SUM requests. This error only depends on *epsilon* and *m* that are used to generate perturbations. Therefore it does not depend on the dataset size. However it depends on the data set values because the sensibility of a SUM request is the maximum spread between two values. In our case it is *m* – 0 = 1000;
2. prints the ratio between the average error and the Laplace scale factor. In our case,

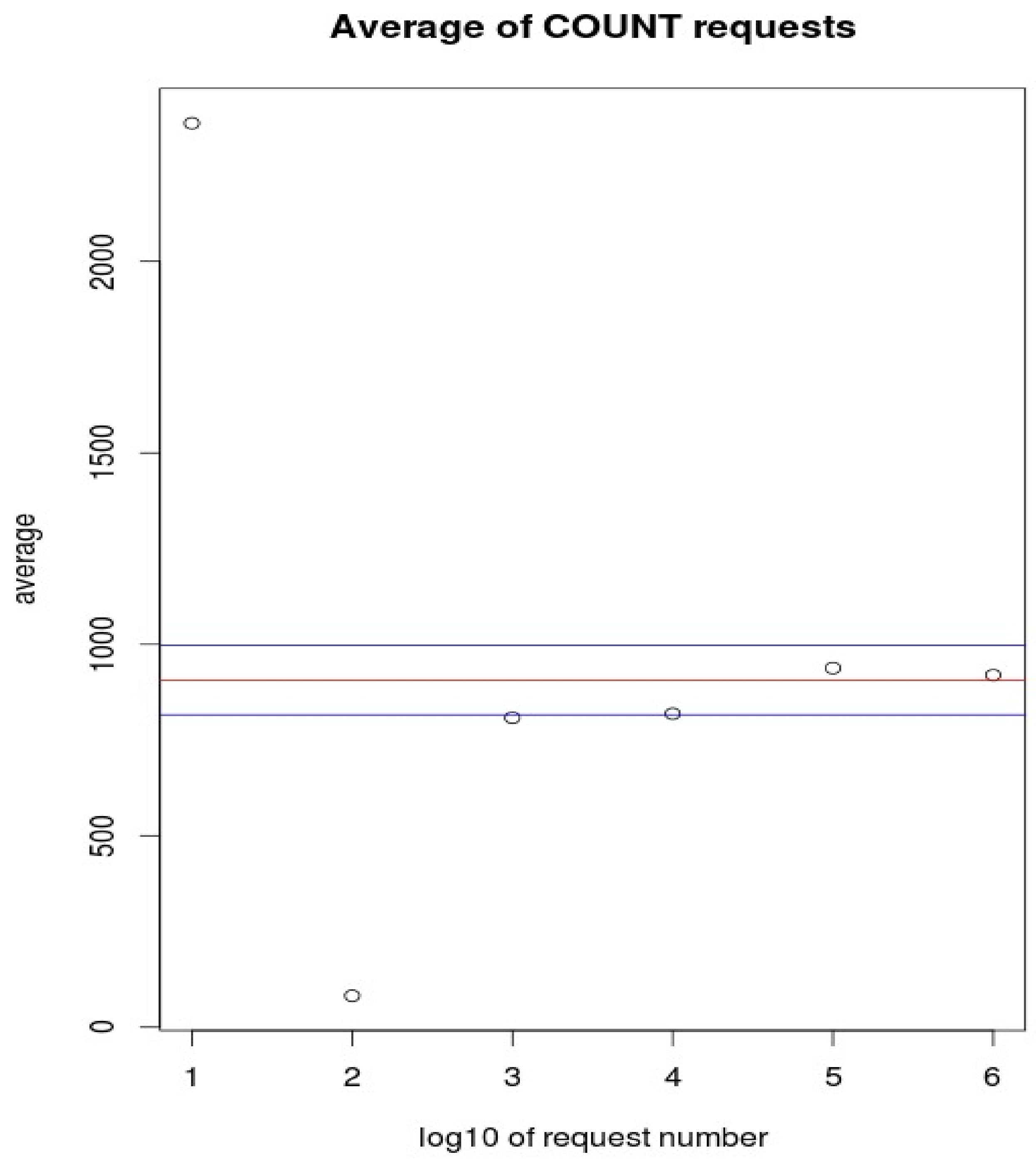
the Laplace scale factor is: 1000 / 0,01 = 100000;

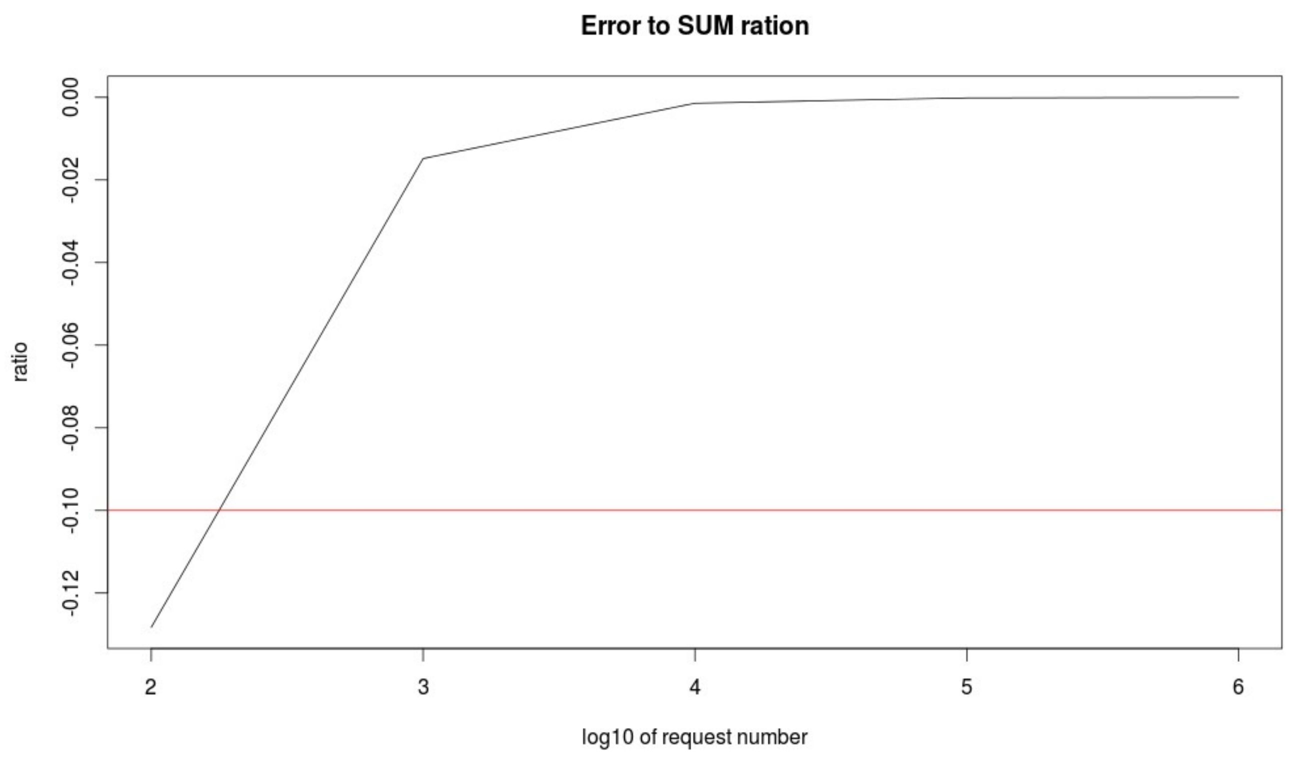
3. write the requested values in in a file called *question6.csv*. The values are displayed in the graph called *Error to sum ratio* (see below).

After running this example several times, we notice that the error to sum ratio becomes very rapidly smaller that 10 % for the SUM request. It clearly shows that we must limit the number of requests. Otherwise, a malicious person could be able to draw information from requests which would differ by just one element, thereby guessing with a very good approximation the value of that element.



Graphs


Average of COUNT requests
average
0
500
1000
1500
2000
1
2
3
4
5
6
log10 of request number


Error to SUM ration
ratio
0.00
-0.02
-0.04
-0.06
-0.08
-0.10
-0.12
2
3
4
5
6
log10 of request number