Utrecht University

Graduate School of Natural Sciences

Estimating Local Sensitivity for Differential Privacy using Machine Learning

Craig Leek (5689279)

MSc Artificial Intelligence

Supervisors:

dr. D.P. Nguyen
Utrecht University
dr. M.R. Spruit
Utrecht University
ir. S.W.R. Niesink
Info Support
J. Snijder
Info Support

Friday 10th July, 2020

Abstract

People value their privacy, and preserving it is important. However, due to privacy concerns a lot of data is currently not usable. Differential privacy promises to protect the privacy of individuals in a dataset, while still allowing statistics to be taken from it. Through adding noise to the output the accuracy of the answer gets reduced in such a way there is nothing to be said about the individuals in the dataset.

The noise that is currently added to a dataset has to protect all possible datasets. This is due to the fact that by protecting only this specific dataset, the noise tells us something about the dataset. Using this local approach thus does not allow us to guarantee privacy, but using a global approach the utility we get from the data is lower than possible.

We propose a method to allow less noise to be added, while still preserving differential privacy. To do this we calculate the required noise on a synthetic dataset generated by a GAN, rather than the real dataset. This approach allows us to produce an amount of noise based on the real dataset, without the noise disclosing information about the real dataset. Our method uses the Laplace mechanism to provide privacy to numerical aggregate queries.

We show that by using our method the accuracy of the result of queries increases, while preserving differential privacy in practice. We test our method on 3 different datasets on a total of 26 queries. We show that our system can be used in real world applications, keeping in mind the risks of differential privacy in general.

Contents

1 Introduction

1.1 Context

There is a massive trove of data that is currently not usable for research. There are many possible reasons why this data cannot be used, with one of the main ones being privacy. Laws have been created to protect the privacy of personal data, for instance the GDPR in the European Union and the CCPA in California[1][2]. Certain personal data, such as medical data, gets a specific position in the GDPR as sensitive data, which comes with even stricter processing standards. Preserving privacy on datasets is very important both for the people whose information is contained in the dataset, and for the owners of the dataset, as they can face hefty fines when there is a privacy leak.

Transforming raw data into privacy preserving data is currently a manual process, as someone has to go over each file by hand and redact all personal data. Due to the inefficiency of this process it is barely done, leading to a data shortage in research. This shortage is becoming more prevalent with the emergence of machine learning, as a lot of data is needed to properly train and test a model. Models based on sensitive data could for instance be used to predict whether a person will get a disease, and using more data to train such a model would lead to more accuracy in prediction. Therefore, it is important to find a way to use this data without having to manually edit out all personal information, while still preserving the privacy of everyone involved.

The basis of this issue lies with a dataset containing private information. This dataset can be anything from a collection of documents to a relational database. We want to be able to query our dataset to extract information from it. If we were to naively do queries, this can disclose private information. Take for instance a query `SELECT balance FROM customers WHERE firstName = 'Craig' AND lastName = 'Leek'` which we execute on a dataset provided by a bank. This query will disclose exactly how much money I have, which is not something I want other people to know. This means that we need to design a mechanism that allows us to still extract meaningful data from a dataset, without infringing on the privacy of anyone by being too precise. While directly querying a dataset like we do in this example can be easily avoided, there are more complex ways to get the same result. For instance we could do two `SUM()` queries where one is on the full dataset and the other is on the full dataset spare me. The difference between those two results will still disclose my exact wealth.

Due to the massive amounts of data a person produces nowadays, and how readily accessible it is, maintaining privacy has become increasingly hard. Research into privacy in datasets has been happening since at least the 1970s[3]. At the end of the 80s a first overview paper was written, which categorised the techniques to maintain privacy in datasets in three groups: query restriction, data perturbation and output perturbation[4]. Query restrictions limits the possible queries, data perturbation changes the data we execute our query on, and output perturbation changes the output of the query.

After its introduction in 2007, differential privacy has become the “gold standard” of privacy, as it allows us to specify the exact privacy risk posed by answering a query[5]. The main idea behind differential privacy is to add noise, scaled to the sensitivity of the query, to the result. The sensitivity of a query dictates how much the result can change by changing the data of one person. By adding noise scaled to the sensitivity, no information will be disclosed when changing the data of one person, and thus the privacy of this person is ensured.

1.2 Problem

Differential privacy relies on the **sensitivity of a function** [6]. The sensitivity of a function will be different depending on both the dataset and the function. While calculating the sensitivity of a function is trivial, doing so discloses information about the dataset. Take for instance a `SUM()` query. The sensitivity of a `SUM()` query is the maximum value in the dataset, as removing the maximum value will have a greater impact on the result than removing any other value. This means that when someone knows the sensitivity, they know the maximum value of the dataset. Therefore, using the sensitivity in this way leads to issues with privacy.

Currently a lot of differential privacy application rely on **global sensitivity**. The global sensitivity is the same across all datasets, and thus does not reveal anything about the dataset itself. However, due to the global sensitivity having to cover all datasets, it can be a lot higher than is needed for the dataset we are interested in. As global sensitivity has to protect all datasets, the value has to be the maximum possible for that datatype, for instance 2^{32} for a 32-bit integer. Using the global sensitivity can lead to a lot more noise being applied to the data than needed, reducing the accuracy of the final answer. If our final answer is less accurate the utility of our dataset is also decreased.

To combat this we want to find a way to use the **local sensitivity**, the sensitivity of the dataset itself. We need to construct a mechanism that allows us to use the local sensitivity, or a value close to it, without this sensitivity value disclosing anything about the dataset. This will allow us to add just enough noise to protect the privacy of everyone in our dataset, while keeping the utility as high as possible.

1.3 Research

We want to find a method that allows us to use a value near the local sensitivity value, while preserving the privacy of everyone in our dataset. To do so we propose using machine learning, specifically **generative adversarial networks** (GANs). We will use GANs to generate synthetic data based on the real dataset. Using this synthetic data we will estimate a sensitivity value which will allow us to preserve privacy.

Focus of research

For our research we will focus on the **Laplace mechanism** as proposed by Dwork et al[6]. The Laplace mechanism was designed to preserve differential privacy in numerical queries. To do so it adds noise to the result of the query, scaled to the sensitivity of the function. We will not change anything in the definition of the Laplace mechanism, but rather only look at the sensitivity which to scale of the noise. Dwork et al have proven the Laplace mechanism to be ($\epsilon.\ \delta$)-differentially private [5]. This means that the requirements for preserving differential privacy are the same for our method as they are for the Laplace mechanism. Firstly, we need a correct implementation of the Laplace mechanism and secondly, we need to use a sufficiently high sensitivity value.

There are many different possible queries, and providing privacy for them all would be ideal, but also quite challenging. Therefore, we will focus on a specific subset of queries, namely **numerical aggregate queries**. As is shown in the paper by Johnson et al, the most common aggregate queries are `COUNT()`, `SUM()` and `AVG()`, accounting for 88% of aggregate functions [7]. Therefore, we will focus on these 3 types of queries. However, we will have no specific queries for `COUNT()`, as `SUM()` and `COUNT()` are quite similar. `COUNT()` returns the count of rows of a dataset, where `SUM()`

returns the total sum of all values in the column. This means that if we do `SUM()` on a column containing only ones it will return the same result as `COUNT()`. We can also say that the sensitivity of a `COUNT()` query is always 1, as adding or removing one person from the dataset only changes the result bu a maximum of 1.

An example of the type of query we will look into is "`SELECT AVG(dose) FROM drugs WHERE drug='paracetamol' AND patientGender='F'`", which will return the average dosage of paracetamol taken by females in the dataset. These type of queries will all have non-trivial local sensitivity values, allowing us to properly test our method.

For the GANs we will use the **CTGAN** Python library[1][8]. This library allows us to generate tabular data without having to create the GAN ourselves. It was created specifically for generating tabular data, thus fits our needs nicely. Most other GANs are used on image data, for example to increase the quality of images[9].

We demonstrate our mechanism on three different datasets: the **Wisconsin breast cancer dataset**, the **kidney disease dataset** and the **diabetes dataset**, all from the UCI Machine learning repository[10]. We chose those datasets for two reasons. Firstly, the Wisconsin breast cancer and the kidney disease datasets are being used in the anomiGAN paper, which is one of the methods we will compare ours with[11]. Secondly, these datasets all have different properties which allow us to test our system more accurately.

Research question

Our main research question is:

Can GANs be used to estimate the local sensitivity of a function for use in differential privacy?

This leads us to three important subquestions.

Can GANs estimate a sensitivity value which satisfies the requirements of differential privacy? For an application to return a differentially private result there are two important considerations, when looking at this proof in isolation. Firstly the implementation of the Laplace distribution has to be correct. If this implementation is not correct privacy leaks can happen. An issue that could exist is the random number generator used not being secure for differential privacy. We will be using the diffprivlib Python library[2] created by IBM for differential privacy mechanisms[12]. As we use a preexisting solution for our Laplace mechanism, we will assume the implementation to be correct.

Secondly, the estimated sensitivity used has to be at least as high as the real sensitivity. If the estimated sensitivity is lower than the real sensitivity the Laplace mechanism will no longer return a differentially private result. Therefore, we need to verify whether the estimated sensitivity is at least as high as the real sensitivity.

Does our estimated sensitivity value disclose anything about the dataset? As we already briefly mentioned, by using the local sensitivity value we disclose information about the dataset.

[1] `https://pypi.org/project/ctgan/`

[2] `https://diffprivlib.readthedocs.io/en/latest/`

In the case of a `SUM()` query we disclosed the maximum value in the dataset. As this goes against the principles of differential privacy we will have to ensure that our estimated sensitivity does not disclose anything about the dataset.

Does our estimated sensitivity value give us a higher accuracy on the result of the query than when using global sensitivity? Our estimated sensitivity value is only useful if it can increase the accuracy over using global sensitivity. As noise is added to the result based on the sensitivity a higher sensitivity will lead to more noise being added. To demonstrate our estimated sensitivity gives an increase in accuracy we will compare results when using global sensitivity versus our estimated sensitivity.

1.4 Structure of this thesis

We start by exploring different methods that have been proposed to preserve privacy, and introduce differential privacy in Section 2. After this we give a short introduction into GANs in Section 2.3, before delving into our datasets and the GAN models we train on our datasets in Section 3. Once we have explored these, we introduce our method in Section 4. In Section 5 we provide a proof that our method can provide differential privacy in practice. We then show our experiment and results in Sections 6 and 7. We end with a conclusion in Section 8 and a discussion in Section 9.

2 Related work

2.1 Privacy mechanisms

In this section we will discuss some ways to preserve privacy. We will be looking at three different groups of mechanisms, query restriction, data perturbation and output perturbation. All information in this section is from Adam and Wortmann (1989) [4].

2.1.1 Query Restriction

Query restriction aims to prevent users from firing certain types of queries. This can be based on either the results of the query itself or it can be mechanisms that look at previous queries as well to prevent combinations of them from disclosing private data. We will look at a couple of mechanisms that try to achieve this.

Set size restriction This method looks to prevent queries that return too few results. While this method prevents the disclosure of one record naively, it is still relatively easy to retrieve the exact values of it with a bit of maths. If we set the minimum size of a return set to 10, a person looking to retrieve the value of one specific person can retrieve the aggregated values of 11 people first, and after that adjust the query to not include the one person he wants to know the value of. This will return a new result of the 10 remaining people and with some simple algebra the value of the one person can be constructed. Therefore this method is not sufficient to guarantee privacy on a dataset.

Set overlap control Result sets that have a lot in common are also quite likely to release information about each other. The example in the previous section is also applicable here, as the only reason this privacy infringement is possible is because of the overlap in the sets. Set overlap control looks to mitigate this risk, by preventing the result set from queries from overlapping too much. This entails setting a maximum amount of results that can be the same between two queries. However, there are some important flaws here as well. The most obvious being that if multiple users coordinate the overlap control will not work. Another big disadvantage is that once a person gets one statistic on a set, they cannot get another. This could for instance mean that after retrieving the count of people with disease A, they can no longer find the count of people with disease A undergoing treatment B. This could be a big hindrance to doing useful research on the set.

Auditing This method relies on keeping logs of all queries, and once a possible privacy infringement is detected preventing that from happening. This mechanism can look for both global and local privacy infringements. Global privacy infringements occur when multiple user do queries that can reveal private data, while local infringements happen when one user does all the queries required for revealing private data. While this looks promising, a log has to be kept going back forever, because if logs get deleted after a certain amount of time a person will still be able to do compromising queries if they just wait for the logs to be deleted. There also has to be a quite intricate algorithm to prevent all forms of privacy infringement, which might cause a large overhead in query processing time, especially if global infringements are checked against all past queries.

Partitioning Partitioning looks to group entities into mutually exclusive subsets. This creates statistics on those sets and as long as there are no subsets with exactly one entity, this should ensure privacy quite well. However, to prevent subsets with one entity they might need to be merged with a neighbouring set, which leads to information loss.

Cell suppression Where partitioning combines a cell that might leak private information with another, cell suppression plainly does not release that cell. The advantage of this is that with cells that are composed of more than one individual do not have their data tainted by being combined with another cell. However, the data of cells that would be combined in the partitioning approach are completely lost. Both partitioning and cell suppression only work with static databases, as with dynamic databases data can be found by changing the composition of the database and looking at the counts.

2.1.2 Data perturbation

Data perturbation changes the data itself rather than the output. There are two general categories these methods can be put into, statistical methods that looks at the distribution in the data and uses that to create a new sample, and permanent methods that change the original data once and for all. Quite a lot of the methods in this section will have a big risk of introducing a bias which would be detrimental for the usability of the dataset.

Data Swapping For this method the database is replaced with a new database with approximately the same t-order statistics. "A t-order statistic is some statistical quantity that can be computed from the values of exactly t attributes, for example, the number of patients whose Sex = Male and Disease = AIDS is two-order frequency count." [4] From this a random database is created which can be queried and get approximately the same return values. This method is best for a static database as with a dynamic database the statistical properties of it will change on each mutation, thus requiring the dummy database to be updated as well. A similar principle is still being applied, but the way the sample is calculated is different. A recent example of this is AnomiGAN [11], which we will talk about more in section 2.3.4.

Method by Liew et al. (1985) This method works to protect one private variable. The idea is that the probability density function from the private variable is calculated, and from that a new sample of data is taken. The original data is consecutively replaced with the generated data in the same rank order. This way comparison relations are mostly preserved, but exact values are gone.

Fixed data perturbation Where in the two previous methods the data was replaced by a representative sample looking at statistical properties fixed data perturbation works directly with the real data. For numerical attributes one could just add the noise straight to each value, rather than only adding it in while processing the query result. This could mean directly adding the noise to the data, but could also be a multiplication so the impact of the perturbation is relatively the same size for different values.

2.1.3 Output perturbation

This method relies on perturbing the output of the query to preserve privacy. This perturbation will add, or subtract, from the actual output which will make it less accurate. There are a few techniques available, with the main one being differential privacy. We will also take a brief look at rounding.

Rounding For rounding the answer is rounded up or down to the nearest multiple of a certain base value. There are three types of rounding, random, systematic and controlled. Systematic rounds to the nearest multiple, so up if it is closer to that value and down otherwise. Random takes a certain probability and rounds up with that probability, and otherwise down. Controlled rounding is a little more intricate. While the rounding of the cell itself is roughly the same, other cell values are rounded as well to preserve the row and column counts to their true values. For this purpose you can imagine cells as being one value in a larger table of values. For instance if we take a table that shows the number of dogs owned by people in certain age ranges, we can have columns 1-10, 11-20, 21-30 for the age ranges etc and rows 1, 2, 3 etc for the number of dogs. At the intersection of each of these columns and rows is a single cell. Controlled rounding looks to maintain total values, so there will not be more 21-30 year old people after rounding than before, neither will there be more people owning 2 dogs. Of course these will not be the exact counts in the data, but it will remain as close as possible to it.

Each of the three have a slightly different way of going about things. For instance systematic rounding could round 13 to 15 (nearest multiple of 5), random could do it to 10 or 15 (randomly going up or down to the nearest multiple of 5 on both sides) and controlled also to either 10 or 15, however, as controlled rounding can also affect other cells in the row and column to preserve counts, these values are all but guaranteed. This could also be done for e.g. base 10, with 13 rounding to 10 for systematic, 10 or 20 for random and the same for controlled.

The main issue with rounding is that, whilst counts cannot be used to guess counts naively, if data changes it can happen that the rounding changes, giving away the data it is trying to protect. This makes it not usable for a dynamic database. In addition to this, combining several rounded tables, or even several records in the same table, together, can still lead to discovery of private information.

2.2 Differential Privacy

This section is largely based on the work of Dwork and Roth[5].

Differential privacy promises that all adjacent datasets will return almost the same result for the same function[5]. ***Adjacent datasets*** are two datasets that differ in at most one element. This element could be either a full row added or removed, or a value in a row being changed. If almost the same result is returned in all cases the conclusions based on these adjacent datasets will also be the same. Take, for example, research into the adverse health effects of smoking. When a person decides to participate in this research they will consider if there is any harm for them in doing so. In this research a smoker could decide against it as they are afraid their health insurance premium will go up when the research shows smoking is bad for your health. If you want to ensure that as many people as possible participate in the research you will have to find a way so that the harm for a person participating is minimised. Here differential privacy comes in, as adjacent datasets will return almost the same result, it does not matter if an individual participates or not. There might still be harm to the person from the conclusion of the research, but this harm will not stem from their personal participation but only from the conclusions that can be made when considering everyone who participated. The mathematical definition of differential privacy is: a function f is a (ϵ, δ)-differentially private function on $\mathbb{N}^{|\mathcal{X}|}$ if for all $S \subseteq Range(f)$ and for all $x, y \in \mathbb{N}^{|\mathcal{X}|}$ such that $||x - y||_1 \leq 1$ (adjacent datasets):

$$Pr[f(x) \in S] \leq e^{\epsilon} * Pr[f(y) \in S] + \delta$$

Looking at each component in layman's terms, an algorithm is (ϵ, δ)-differentially private when adjacent datasets have a probability density that is nearly indistinguishable from one another. The amount it can differ is tunable by changing ϵ and δ. If $\delta = 0$ a function is ϵ-differentially private.

2.2.1 Perturbing output

Numerical queries Differential privacy has been defined for different types of data. For numerical data we can add noise from a Laplace distribution to perturb the output[5]. This noise has to be tuned so when we add or remove one record from the dataset the result stays the same. This means that the noise depends on how much the result of our function can change when moving to an adjacent dataset. We call this change the ***sensitivity of a function***. An example of the sensitivity of a function is when we try to sum the binary records. As a binary record can have value 0 or 1, the maximum the function can change when moving to an adjacent dataset is 1. We write the sensitivity of function f as Δf. Once we know the sensitivity of the function we use this to calibrate the noise we will add to the output. In our numerical example this would be done through the following operations: $f(x) + y$ where y is drawn from $Lap(\Delta f/\epsilon)$. The ϵ in the Laplace distribution is the ϵ that is set to ensure ϵ-differential privacy. By adding noise calibrated this way we get a result that is $(\epsilon, 0)$-differentially private.

We will now look at an example to gain a better understanding of how this works. If we once again take a sum on some binary data, we can immediately say the sensitivity of the function is 1. Now we will need to pick ϵ . There are no set in stone ways for how ϵ needs to be chosen, a lower ϵ will give higher privacy guarantees while a higher ϵ leads to less noise being added. For our example we will pick $\epsilon = 0.01$. Now firstly we will execute our query and get the true answer. In our case the true answer is 25. To make it differentially private we will need to add our noise from $Lap(\Delta f/\epsilon)$. We know $\Delta f = 1$ and $\epsilon = 0.01$ so our noise will come from $Lap(1/0.01) = Lap(100)$. We will add this noise to our true answer to get a $(0.01, 0)$-differentially private answer. This means the answer we return to the user will be $25 + Lap(100)$.

Categorical queries When working with categorical data there are different considerations. For instance if you want to find the most common eye color in a population adding flat noise, as was done in the previous paragraph, would not provide the best result. Rather than adding noise we change how the correct value is returned. In the previous mechanism we added noise to the correct answer, but for categorical data we add noise in returning the correct answer. This mechanism is called the exponential mechanism[5]. The exponential mechanism uses a utility function u, dataset X, the set of outcomes R and outcome r where $r \in R$. Using these parameters it aims to achieve that with a probability proportional to $exp(\epsilon u(x, r)/\Delta u)$ the mechanism returns the result r. The utility function should return a higher result for an answer closer to the truth in $r \in R$. When this is implemented correctly, the mechanism has a higher chance of returning the correct answer than any other answer, but there still is a chance to return the wrong answer.

We can look at our example with the most common eye colour to explain how this works. Take a community with three eye colours, brown, green and blue. 100 people have brown eyes, 99 have green eyes and 10 have blue eyes. If we were to query this dataset for the most common eye colour we would always get brown back as the most common eye colour. However, when one person with brown eyes moves out of our community we will have an equal number of people with brown and green eyes, so our query will return both. This change will tell you that the person who moved out has brown eyes, and thus reveal personal information. Now if we use the exponential mechanism there is a large chance a max query will return brown, a smaller chance it will return green and an even smaller chance it will return blue. This way, each answer is always possible, no matter the composition of our community. This means that when the same person moves out again and our query returns green rather than brown, it could just as well be because of the randomness in the mechanism. Therefore, no private information is disclosed.

A more basic implementation of the exponential mechanism is randomised response[5]. This mechanism uses two coin flips to decide between two possible answers. If we once again take the example of a smoker, they could decide to answer the question “Are you a smoker?” through randomised response. The way to go about this is first flipping one coin, if it returns heads the smoker should tell the truth. If it comes up tails, the smoker should tell the truth if it comes up heads again, and lie if it comes up tails. This method is most likely to return the truth, but as there is the possibility of a lie, the person asking the question can never be sure whether the answer is the truth. This mechanism is $(ln(3), 0)$-differentially private[5].

2.2.2 Sensitivity of a function

The sensitivity of a function is the amount the output can change when one person is added or removed from the data. This creates an issue when we have a function with a potentially big domain, but which primarily operates on a much smaller domain. Take for instance a database compromised of the net worth of every person on earth. When taking the sum on this table the sensitivity of the function is dictated by the person with the highest net worth, which is in the 100 billion region, while the biggest share of people is under 1 million, and a lot even lower. Even though the value for this one person is way higher than the rest, making sure everyone’s data remains private entails adding in noise that protects the privacy of this person as well. This will lead to us having to add a large amount of noise to the answer, making the end result almost useless.

To prevent this we can instead look at local sensitivity. Where ***global sensitivity*** takes the worst global case into account, ***local sensitivity*** looks only at the current dataset. Thus the local sensitivity will need to be computed for each query. However, there are no current mechanisms that satisfy both the concept of differential privacy and calculate the local sensitivity. The main reason for this is that the use of local sensitivity can give away information by the magnitude of the noise. As the noise depends on the sensitivity, when the noise gets smaller or bigger one can say whether they added or removed a value that changed the sensitivity. We will now look at an example for how the accuracy of the answer changes with different sensitivities.

Figure 1 shows the different errors at different sensitivities. The query used is “SELECT SUM(num_medications) FROM diabetes”. This query returns the total sum of medications used by all people in the diabetes dataset, with the true answer to this query being 1630479, which is roughly $10^{6.2}$. We take the true answer and use the Laplace mechanism to randomise it and create a differentially private result. The sensitivity values used for the graph are 10, 20, 35, 50, 75, 100, 2^8, 1000, 10000, 2^{16}, 2^{20}, 2^{24}, 2^{28} and 2^{32}. These sensitivities have been arbitrarily chosen, except

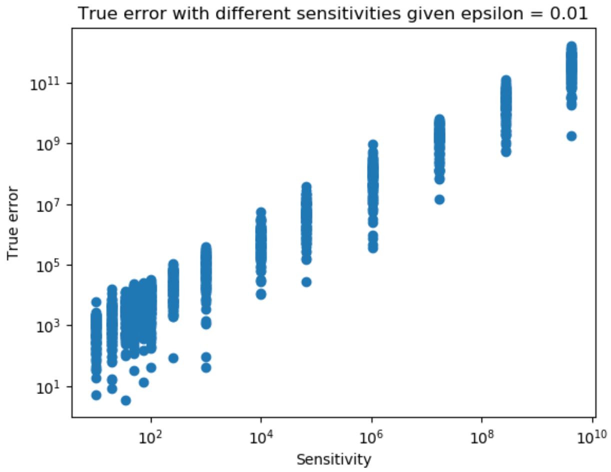


Figure 1: True error for different sensitivities

for 2^{32} which is the global sensitivity value. We have chosen to demonstrate the results with this range of values to demonstrate how a higher sensitivity value affects the result. Each sensitivity has been used in 100 different samples and the results are displayed. Each error value is calculated by the following formula

$$|dp\_result - true\_answer|$$

There is a visible linear relation in this log-log plot, meaning that the actual relation is exponential. This shows how a bigger sensitivity leads to a worse and worse accuracy exponentially, and the current situation where global sensitivity is used (2^{32}) is very inaccurate. From around the sensitivity value of 10^4 we can see that the noise is getting even bigger than the result itself. From this we can see that by reducing the sensitivity value used in differential privacy can lead to a big gain in accuracy.

In this thesis we will consider static datasets exclusively. Static datasets are datasets that will not be changed anymore, no records will be added removed or altered. This allows us to calculate the local sensitivity of a dataset quite easily. If we were to consider dynamic datasets, which can be changed, we would have to factor possible future records into our sensitivity calculation as well. If we once again take a `SUM()` query we can see the difference. On a static dataset we take the maximum of the current dataset, which will be the local sensitivity. If our dataset were dynamic quite often this is not a hard guarantee. A new record can come in with a value higher than the current maximum. This value will not be protected by the local sensitivity that was used so far. While we can protect it once it is in the dataset, when new results are combined with old results this can disclose information about the added record.

Current methods for local sensitivity estimate We will now look at two methods that are currently used to estimate local sensitivity, while preserving differential privacy. We will look at smooth sensitivity, which has been around since 2007 [13], and elastic sensitivity, which was published in 2018[7].

Smooth sensitivity is calculated using a function $S(x)$ with the constraints that $\forall x : S(x) \geq LS(x)$ and $\forall x, y \;\&\; ||x-y||_1 \leq 1 : S(x) \leq S(y) * e^{\beta}$. This can be combined into one formula $S^*(x)$ for the β-smooth sensitivity of X, being:

$$S^*(x) = max(LS(y) * e^{-\beta * ||x-y||_1})$$

Elastic sensitivity computes the sensitivity by using properties of the dataset. It is specifically designed to work with joins and exploits properties of joining in computing the sensitivity value. Rather than looking at sensitivity directly, elastic sensitivity considers the elastic stability ($\hat{S}_R^{(||x-y||_1)}$) of the function. The exact computation of the elastic stability can be found in sections 3.3 and 3.4 of the paper by Johnson et al[7]. Once this has been computed to get to a sensitivity value once again smooth sensitivity is used, but replacing $LS(y)$ with the elastic stability, leaving us with the following function

$$S^*(x) = max(\hat{S}_R^{(||x-y||_1)} * e^{-\beta * ||x-y||_1})$$

More detailed explanations of either of those techniques and their proofs are in their respective papers [7][13]. Both these methods still work on only the real dataset, thus creating a tight link between the sensitivity used and the real local sensitivity. The method we will introduce does not have a clear relation between the sensitivity we use and the real local sensitivity, potentially allowing for a more accurate sensitivity value.

2.2.3 Uses of differential privacy

Differential privacy is already being used by some big tech companies. We will give a brief overview of some of them and how they currently use differential privacy.

Google designed RAPPOR to get data from users while still preserving differential privacy [14]. RAPPOR is based on the randomised response mechanism, returning the truth or a lie depending on a coin flip.

Apple uses a different technique for differential privacy [15]. This is also meant to get user data,. They do this by hashing the relevant data so it fits in a matrix and randomly flipping bits. After this they return a random row in the matrix. An analysis of this technique has been done as well, citing a lack of transparency and no clear indication of the possible privacy loss [16].

Facebook uses differential privacy for research into the influence of social media on elections [17].This allows researchers to use data from Facebook while the privacy of users is still guaranteed. An extensive paper on this method has also been published by Facebook, where they descrive requirements for using an application employing differential privacy in production [18].

Microsoft uses differential privacy for a host of applications, telemetry in Windows [19], advertising queries on Linkedin [20], and suggested replies in Office [21]. The telemetry is once again a similar approach as we have seen from Google and Apple. The other two use more advanced forms of differential privacy and propose new mechanisms to solve their problems.

Elastic sensitivity, as discussed before, is being used by Uber for internal data analysis.

The US Census bureau will also use differential privacy for the 2020 census [22]. This way they hope to preserve the privacy of everyone who submits data to the census. As the 2020 census is not yet published results of this are not yet known.

A lot of research into differential privacy is also currently on going. A lot of sources from the previous section are from the last 3 years, with a considerable amount even being released in the first half of 2020. Differential privacy is currently a hot field with a lot of changes happening as we speak.

2.3 Generative Adversarial Networks

2.3.1 Uses

GANs are very popular in image generation, being used to improve the quality of images [9], or even generate images based on text [23]. However it has also been used in generating realistic structured data [24][8], and more specifically medical data as well [25] [26]. It has also been compared to differential privacy [11], showing that reasonable privacy can be achieved by replacing the whole population. However, this paper does not take into account that the distribution itself can also be privacy sensitive, as with a dynamic dataset every time the data changes the distribution changes. This change in distribution can then be used to figure out the exact value of the record that has been removed. This is also the issue discussed, and to and extend solved, in the paper on smooth sensitivity [13].

2.3.2 Structure

Generative Adversarial Networks (GAN) are essentially two different neural networks [27]. These two networks compete with each other in a game. This game consists of one of the networks, the generator, generating data based on noise and the second network, the discriminator, will try to spot whether the provided sample is from the training data or generated by the second network. The networks in the GAN can be either regular networks, but also recurrent networks, as shown in the paper by Mogren [28].

2.3.3 Training

Training a GAN is more complex than training a single neural network. Part of this complexity comes from the fact that two networks are being trained at once, but another part is deciding when training is done. Training the networks happens separately, while the generator is going through a few epochs the discriminator stays the same and vice versa. This is so each network has a chance to train itself without the other continuously getting better.

There are a few known issues with GAN training, failure to converge, mode collapse and diminished gradient. An important improvement in this has been made with the introduction of Wasserstein GANs [29]. Rather than saying if an image is real or fake it judges the realness, or fakeness, of an image. While this is already more stable in training, more improvements have been made in the training [30]. To this even more improvements were made [31], showing that this is very much a hotly researched topic which is still very relevant.

2.3.4 Implementations

In this section we will discuss the two GAN implementations which are relevant for this thesis. anomiGAN, which is a different approach to preserving privacy using a GAN, and CTGAN, which is a GAN created to replicate tabular data.

anomiGAN The purpose of anomiGAN is not necessarily to create data that is the same, but rather data which results in the same result in some target classifier [11]. The target classifier, and starting dataset, can be anything. In the paper they use a machine learning model as target classifier, but as we are interested in queries we will use them instead.

In our case it means the following: rather than executing our query on the real dataset we execute the query on a dataset with the same number of rows. The result of a query on synthetic data can then be compared to the result when doing the same query on the real data. Another possible application of this is when training a classifier. Take for instance a fraud detection problem. You want to create an as accurate as possible classifier, but financial data is very privacy sensitivity. By replacing the data the classifier trains on a lot of privacy concerns are alleviated, while the result should stay roughly the same.

While there is source code provided with anomiGAN this did not function properly for our use case. As anomiGAN is designed to be used with a machine learning model the published source code does not easily translate to an application where we want to answer queries. Therefore, we decided to use the principle behind anomiGAN rather than the exact implementation. For our synthetic data we use the data generated by CTGAN, which will be discussed in the next section. This means that there is no direct comparison with anomiGAN, but rather just with the idea of replacing all data as proposed by anomiGAN.

CTGAN CTGAN was specifically created as a model that can recreate tabular data [8]. CTGAN is a continuation of previous approaches that did not yet use a GAN for this task, and has its source available as a Python library[3]. This makes it a very easy to use framework, while providing accurate synthetic data.

CTGAN works for both discreet and continuous data, making it work well on most datasets. There are a host of issues with creating a GAN that works properly on tabular data, for instance the mix of continuous and discreet data means that different generation techniques have to be used in the same GAN. Another issue is that the distribution in tabular data is quite different from image data, which GANs have so far had most success in. Even more challenges are described in the paper on CTGAN.

[3] `https://pypi.org/project/ctgan/`

3 Datasets and models

We will now briefly discuss the datasets we will use for our research. Firstly we will use the datasets also used in the anomiGAN paper [11]. These are the Wisconsin breast cancer set and the chronic kidney disease set from the UCI Machine Learning repository [10]. A third dataset we will use is the Diabetes dataset, which also comes from the same repository. We will look at certain aspects of the datasets that are relevant when training our GAN. These are the amount of unique values for each variable, the cardinality of each variable and the amount of missing values. We will not discuss every variable independently, but only look at the variables that are most interesting in each group.

After discussing the datasets we will discuss the GAN model we trained on it. For this we will compare to the real data on the cardinality of variables and the uniqueness of variables. Another metric that can be considered when comparing the GAN to the real data is the sensitivity of the real data versus the generated data. This comparison is the most important for us, as it is the purpose we use the GAN for, and will be included in the Results section in Section 7.3.

Each GAN was created using the CTGAN library and trained on the full dataset, after removing all the rows containing null values. We used all default settings in training our GANs. For each GAN, after generating a dataset, we transform every numerical, non-categorical value to its absolute value. This is as in the real datasets negative numbers never occur, while the GAN can produce them.

3.1 Wisconsin breast cancer

The Wisconsin breast cancer dataset has 569 records with 32 variables. The Wisconsin data set is almost fully real numbers. There are 31 numerical variables and one categorical variable (`diagnosis`) with two possible values. There are no missing values in this dataset. `id` is the only truly unique variable in the dataset, however all variables other than `diagnosis` have a high amount with unique values. The variable with the lowest amount of unique values, spare `diagnosis`, is `smoothness_worst` which has 72.2% unique values. The variable with the highest amount of unique values, spare `id`, is `smoothness_se` which has 96.1% unique values. Most other variables are in the 87% to 95% uniqueness range.

Variable	Type	Real Data	Generated Data
id	real	100%	64.3%
diagnosis	cat	0.4%	0.4%
smoothness_worst	real	72.2%	100%
smoothness_se	real	96.1%	100%

Table 1: Uniqueness of variables of the Wisconsin breast cancer dataset, and a dataset produced by a GAN model trained on it

In Table 1 we compare the most interesting variables of the real dataset with those same variables in a generated dataset. Our generated dataset has 596 rows, the same amount as the real data. Three things stand out in this comparison. Firstly, for `diagnosis` we see that the uniqueness is the same. This is due to it being a categorical variable with only two values. The GAN produces these same two values in the generated dataset. Secondly, both `smoothness_worst` and `smoothness_se` are fully unique in the generated dataset. This is due to the values in the generated dataset having a higher precision. Thirdly, we see that the generated dataset does not produce unique values for `id` while they are fully unique in the real dataset. There is no constraint on the GAN to create unique `id` values, and as the `id` values are close together in the real dataset as well, collisions happen in the generated dataset.

A final interesting note here is the correlations between variables on the real and the generated datasets. This dataset includes groups of variables, such as `smoothness_mean`, `smoothness_se` and `smoothness_worst`. In the real dataset there are high correlations between these variables. Our current GAN is not able to produce these correlations between variables. As our current research does not depend on accurate correlations we will not be delving into this further.

3.2 Kidney disease

The kidney disease dataset has 400 records with 25 variables. The kidney disease dataset is fully categorical. The variable with the highest amount of different values is `bgr`, which has 147 values. The variable with the lowest amount of different values is `class`, which only has two.

This dataset has missing values in 0.9% of rows. Other rows also miss values, but they are expressed as "?" rather than null. 8 variables have a higher amount of "?" than any other value, with `rbcc` being the worst offender with 32% "?". The variable with most "?" is `rbc` with 38% "?", however due to only 3 values occurring for this variable "normal" still occurs more often at 50.2%.

Variable	Type	Real Data	Generated Data	Generated Data 10k
bgr	cat	147	107	145
class	cat	2	2	2
rbcc	cat	48	47	48
rbc	cat	3	3	3

Table 2: Cardinality of variables of the Kidney disease dataset, and two datasets produced by a GAN model trained on it, the first one with 400 rows, the second one with 10000 rows

Table 2 shows the amount of different values for the four variables we discussed. We see that for the values with a low cardinality in the original dataset all values occur even in a small generated dataset, When we look at `bgr` we see that when we have only 400 rows (the same as the original dataset) we only see 107 unique values. However, when we take 10,000 rows we get 145 values, out of 147 in the original dataset. When we look at `rbcc` we can see the same, in 400 rows we see 47 out of 48 values, while with 10,000 we see all 48 values.

Variable	Type	Real Data	Generated Data	Generated Data 10k
bgr	cat	11%	18.5%	15.8%
class	cat	0%	0%	0%
rbcc	cat	32%	20.8%	23.3%
rbc	cat	38%	33.5%	32.4%

Table 3: Amount of "?" for variables of the Kidney disease dataset, and two datasets produced by a GAN model trained on it, the first one with 400 rows, the second one with 10,000 rows

Table 3 shows the amount of "?" that show up in the different datasets. There are no noteworthy conclusions to take away from this data. There is no clear pattern for the amount, nor any obvious relation between the amounts we see in different datasets. We also cannot say that in the generated dataset we see less "?" as in the real dataset, as for bgr there seem to be more. However, one thing to consider is that as we removed all rows with null values from the data before training our GAN the exact percentages of the dataset the GAN trained on will be different from the percentages shown in Table 3.

3.3 Diabetes

The diabetes dataset has 101766 records with 50 variables. This dataset contains a mix of categorical and real data. There are 36 categorical variables, 13 numeric and 1 boolean. Two variables are constant, examide and citoglipton. There is a very low amount of unique values for all variables, spare encounter_id and patient_nbr. The variable with the highest amount of unique values is diag_3 with 790 unique values, diag_2 and diag_1 have 749 and 717 values respectively. Most of the other variables have less than 20 unique values.

Missing values in this dataset are once again symbolised by "?". This dataset has no null values. The variable with the highest amount of "?" is medical_specialty with 49.1% of all values being "?". payer_code comes in second with 39.6% of values missing.

There are 4 variables with a high amount of zeroes, number_emergency coming out on top with 88.8% of all values being 0. This variable is highly skewed towards 0, however the maximum value is 76, which only occurs once. The 5 highest values, ranging from 46 to 76, all occur only once. This makes this specific variable very suitable for testing how our method handles outliers.

Variable	Type	Real Data	Generated Data 10k	Generated Data 50k	Generated Data 100k
encounter_id	real	100,000	10,000	49,995	99,997
patient_nbr	real	71,518	9999	49,986	99,906
diag_3	cat	790	280	434	499
medical_speciality	cat	73	57	69	70
number_emergency	real	33	4	4	4
payer_code	cat	18	18	18	18

Table 4: Cardinality of variables of the Diabetes dataset, and three datasets produced by a GAN model trained on it, the first one with 10,000 rows, the second one with 50,000 rows and the last one with 100,000 rows

In Table 4 we can see the cardinality of different variables in the real and generated datasets. When looking at encounter_id we can see that, similar to what we saw in the Wisconsin dataset, values generated by the GAN are not necessarily unique. With only 3 double values in 100,000 rows this dataset does better at creating uniqueness than the Wisconsin dataset, but when assuming encounter_id to be unique it could still pose an issue.

On the flipside of this we have patient_nbr. This value is unique in about 70% of the cases in the real dataset, but in the generated datasets it is just a bit less often unique as encounter_id. In the real dataset a patient can be admitted multiple times, getting the same patient_nbr but a different encounter_id. This rarely occurs in our generated dataset.

diag_3 has 790 unique values in the real dataset, which our generated datasets do not come close to. We see an increase in the amount of values when we increase the size of the generated dataset, but with 100,000 records we still only have 499 unique values. By increasing the dataset size even further we will probably get even more unique values.

`medical_speciality` exhibits the same behaviour as `diag_3`, showing increased cardinality with bigger datasets. Once we get to 100,000 rows we get 70 unqieu values to the 73 unique values in the real data. Once again if we increase the size of the set even further we might get even more unique values, but for this variable the benefits of doing so are a lot less obvious than for `diag_3`.

`number_emergency` is a very interesting variable. As mentioned before it consists for 88.8% of 0. When we look at the generated data we see the same, with almost all values being 0. The other 3 values that occur here are 1, 2 and 3. This means we are lacking the high value of 76 in all our generated datasets, and so far we do not see any improvement in diversity of values when we increase the size of our generated dataset.

`payer_code` already gets all its values in the smallest dataset, and keeps having all values in all other sizes of the dataset. This is quite interesting, as the value "FR" only occurs one in the real dataset, but still occurs immediately in the smallest sample as well. When we look at `number_emergency` we do not see values that occur only once back in the generated dataset. This might indicate that there is more to it than just how often a value occurs. It is also noteworthy that `payer_code` is categorical, while `number_emergency` is a real number. This might explain why uncommon values do occur in `payer_code` but not in `number_emergency`.

4 Method

Our method has a few important steps, which in order are:

1. Training a GAN model on the dataset
2. Process user query
3. Using the GAN model to get a local sensitivity value
4. Using the local sensitivity value to return a differentially private result

We will now talk about each of these steps in order. We will use an example query to exemplify what happens in each step.

4.1 Training a GAN model

For our GAN we use the CTGAN Python package, which is a GAN specifically designed to recreate tabular data[8]. The only thing that needs to be done when training a GAN is specifying which columns are categorical data, after which the package does all the work. Once the GAN is trained it can be saved to disk and used at a later point when we need to create synthetic data. We have discussed the models more in depth in section 3.

Example

We want to know the total wealth of everyone in our street. To do this we have compiled the financial data of the people in our street and put it in a dataset `streetWealth`. Each person has their own line, only consisting of one value `wealth`. The highest wealth in our street is 50000 and there are 50 people living in our street. The total wealth is 250000. Before we start querying our dataset we train a GAN on it. This GAN will then be saved to disk, so we can use it later.

4.2 Processing user query

Before we can do anything with differential privacy we have to find the true answer to the user's query. To do this we first load our dataset into a pandas frame. Once we have loaded our data we apply all filters specified by the user on the dataset and save the result of this as a temporary second instance of the dataset. We then execute the function as specified by the user on the correct column of the filtered data and get the true answer. With the true answer saved we can move on to the next steps to provide our user with an accurate, yet differentially private, result.

Example

In this step we have to specify our query. As we want to know the total wealth of everyone in our street, our query will be "`SUM(wealth) FROM streetWealth`". As we have not specified a `WHERE`-clause, no filters will need to be applied to the data. We take the sum of our unfiltered data, and this will be the true answer. The true answer to this query is 250000,

If we wanted to know the average wealth of people who have a wealth under 10000 our query would be "`AVG(wealth) FROM streetWealth WHERE wealth < 10000`". For this query we do have to apply a filter. After this filter is applied we take the average of the filtered data, and make this our true answer.

4.3 Estimating local sensitivity value with GAN

```
1  generatedData = model.sample(100000)
2  realSensitivity = CalcSens(data)
3  generatedSensitivity = CalcSens(generatedData)
4
5  difference = abs(realSensitivity - generatedSensitivity)
6  estimatedSensitivity = generatedSensitivity + (np.random.normal() + 3) * difference
7
8  if(np.random.binomial(1, 0.75) == 1):
9      usedSensitivity = realSensitivity
10 else:
11     usedSensitivity = estimatedSensitivity
12
13 usedSensitivity += (np.random.uniform(0.015, 0.06)  * generatedSensitivity)
```

Listing 1: Sensitivity calculation

The code used to calculate the local sensitivity can be seen in Listing 1. First we will discuss what this code does and then in section 5 we will prove that this method is appropriate for differential privacy.

In line 1 fake data is generated from our GAN model. The model we use here is the model trained on the dataset that is being queried. We have chosen for 100000 rows here. We will discuss the exact parameter choice in section 7.5.

Line 2 and 3 are essentially the same computation on two different datasets. Both lines calculate the local sensitivity on their respective dataset: line 2 on the real data and line 3 on the fake. The sensitivity calculation is dependant on the query and therefore will be kept abstract. However, *realSensitivity*, which is calculated on the real data, is assumed to be the correct local sensitivity value. This means that the sensitivity calculation has to be the correct one for the query, and thus for each query type the sensitivity calculation needs to be specified. *generatedSensitivity*, which is calculated on the generated dataset, is the sensitivity of the generated dataset.*sampleSensitivity* does not have to be the same as *realSensitivity*, and might change between different iterations of the algorithm on the same query due to the randomness in generating data.

Line 5 computes the absolute difference between the two datasets. This is needed for later use in the computation to make sure *estimatedSensitivity* is not too low. By taking the absolute value we can ensure no too low value is being used, if *generatedSensitivity* is higher than *realSensitivity* there is no chance of privacy leak. However, if it is too low it can disclose private information in the eventual differential privacy calculation, so we want to prevent any too low *estimatedSensitivity* from occurring.

In line 6 we multiply the $difference$ with a value randomly taken from a Gaussian distribution with a mean of 3. The result of this will then be added to the *generatedSensitivity* and stored as *estimatedSensitivity*. This step is what allows us to guarantee that *estimatedSensitivity* $\geq$ *realSensitivity* in at least 97.7% of cases, for which a proof is provided in section 5.

Line 8 through 11 use the randomised response mechanism as seen in the work by Dwork [5]. This gives plausible deniability for returning *realSensitivity* or *estimatedSensitivity*. This method flips two coins, if either one is heads we return *realSensitivity*, otherwise we return *estimatedSensitivity*. This mechanism prevents any information from leaking due to a single query. Because of the chance of the fake sensitivity being used a person will never be able to tell whether the true or fake sensitivity has been used when only executing one query. Our implementation does not explicitly flip two coins but still has a 75% chance of returning *realSensitivity* and 25% chance of returning *estimatedSensitivity*.

Line 13 adds some noise to *usedSensitivity* as a final random step. This will decrease accuracy a bit, but improve privacy even further as the real sensitivity will almost never be used. The noise we add here is random in both parts, as *generatedSensitivity* is also random for each iteration. This means that the final value of *usedSensitivity* has a lot of randomness, however it is still loosely based on *realSensitivity*

Example

The first thing we do here is take our GAN which we trained and saved to disk in the first step and take 100000 rows of data from is, which we save into the variable *generatedData*.

We then need to calculate the local sensitivity on both our real and fake data. The sensitivity of a function is the amount a function can change by changing one record. Our first query is a `SUM()` query, so the sensitivity is the maximum value in the column being queried, as this is the maximum amount our query can change by changing one record. We will take the maximum value of our unfiltered data as the local sensitivity of this query, which we will save to the variable *realSensitivity*. As the richest person in our street has 50000, $realSensitivity = 50000$.

If we take our `AVG()` query the sensitivity calculation is a little different. The maximum amount an average can change by changing one record is the maximum value in the colomn being queried divided by the total amount of rows in the data. A second difference between our two queries was that with our `AVG()` query we specified a condition. While this condition is relevant to the true answer, it is not relevant to the sensitivity. If we expand the filter we can get a record with a higher value, which we have to take into account. Therefore, the sensitivity calculation has to be done on the *unfiltered* data, as otherwise the result might be too low. As the richest person in our street has 50000 and there are 50 people living in our street, the sensitivity of this query is 1000.

We will now calculate the sensitivity of *generatedData*. The calculation we use here is the same as we used on the real data. In the case of the `SUM()` query that means the maximum value of the queried column in *generatedData* is the sensitivity, which we save into *genratedSensitivity*. Our GAN generated a dataset where the richest person has 47500, and thus $generatedSensitivity = 47500$.

When we look at the `AVG()` query again there is a slight twist. In a generated dataset with the richest person having 47500, we then need to divide this number by the amount of people in the *real dataset* again. If we divide this number by the amount of people in the real dataset we get $generatedSensitivity = 950$. If we were to divide by the amount of rows in *generatedData*, we would get $generatedSensitivity = 0.475$. By dividing by the amount of rows in the real data, rather than in *generatedData*, *generatedSensitivity* is closer to *realSensitivity*.

From now on we will only consider the `SUM()` query, as all steps will be the same no matter the query once we have our sensitivity. This means $realSensitivity = 50000$ and $generatedSensitivity = 47500$.

We will then compute the absolute difference between our two sensitivities. In our case this will be $abs(realSensitivity - generatedSensitivity) = 2500$. After getting the difference we will have to get *generatedSensitivity* near the value of $realSensitivity$. To do so we use a random number taken from a Gaussian normal distribution. This value will range from -4 to 4, with a very small change of a lower or higher value. The most common value is 0. so we will use that here. Once we have this random number we add 3 to it. We then take this result and multiply the difference by it. We add this final result to *generatedSensitivity* to get *estimatedSensitivity*. In our example that would mean $estimatedSensitivity = generatedSensitivity + difference * (random() + 3) = 47500 + (2500 * 3) = 55000$. From this we can see that $estimatedSensitivity \geq realSensitivity$ because $55000 \geq 50000$.

After computing *estimatedSensitivity* we use a randomised response mechanism to decide *usedSensitivity*. This mechanism has a 75% chance of returning $usedSensitivity = realSensitivity$ and a 25% chance of returning $usedSensitivity = estimatedSensitivity$. This means that in our case *usedSensitivity* has a 75% chance of being 50000 and a 25% chance of being 55000. We will say that $usedSensitivity = 55000$.

The last step in computing the sensitivity is adding a bit of random noise. While this step is not strictly necessary, and indeed makes the sensitivity higher than required, it introduces just a bit more randomness to strengthen the privacy guarantees. The extra noise we add is based on *estimatedSensitivity*, which can change between queries, and a random value in a uniform distribution between 0.015 and 0.6. If we get a random value of 0.3 from our uniform distribution, the noise we will add to *usedSensitivity* is $47500 * 0.03 = 1425$. This leaves us with a final *usedSensitivity* values of $55000 + 1425 = 56425$. We will use this sensitivity value in the last step to get our differentially private answer.

4.4 Using local sensitivity to return a differentially private result

When we take *sensitivity* as decided by the code in the previous section we are almost done. The final step is to input it in the Laplace mechanism and get the result. For the differential privacy mechanism we use the diffprivlib[4] Python package developed by IBM [12]. This package contains an implementation of the Laplace mechanism which we use to get our final result. We set the sensitivity of the mechanism to the sensitivity decided by the previous code and then randomise our true answer to get to the final result.

Example

This last step is quite easy. All we do here is apply the Laplace mechanism to our query. As we said before, the total wealth in our street is 250000, and the sensitivity we will use for differential privacy is 56425. This means our differentially private result will be $250000 + Lap(56425)$. The result of this computation will be disclosed as the wealth of our street, while preserving the privacy of everyone in our street.

[4] `https://diffprivlib.readthedocs.io/en/latest/`

5 Proof

In this section we will prove that our system provides an estimated sensitivity value which preserves differential privacy in practice in the case where each neighbouring dataset has the same local sensitivity value. As the minimum requirement for a differentially private computation is that the sensitivity for such computation is at least as high as the true local sensitivity, we will have to prove our system can preserve differential privacy in practice. For this proof we will show $P(estimatedSensitivity \geq realSensitivity)$ is high enough to not be a privacy risk.

5.1 Definitions

Neighbouring/Adjacent datasets : Datasets that are different in at most 1 element

Query : The current query

Real dataset: The dataset on which the query is done

$realSensitivity$: The sensitivity of the real dataset for the query

Generated dataset : The dataset generated by the GAN

$generatedSensitivity$: The sensitivity of the generated dataset for the query

$difference$: The absolute difference between $realSensitivity$ and $generatedSensitivity$, calculated using $abs(realSensitivity - generatedSensitivity)$

$estimatedSensitivity$: The sensitivity estimated by our system after adding $difference * (random() + 3)$ to $generatedSensitivity$

$usedSensitivity$: The sensitivity our system uses after randomised response

5.2 Proof

If we once again look at listing 1 and consider line 6 we will now prove that this line allows us to guarantee 97.7% that $estimatedSensitivity \geq realSensitivity$. The way we do this is taking a value from a Gaussian normal distribution, adding 3 to it, multiplying it by $difference$ and adding it to $generatedSensitivity$. This can be written as:

$$estimatedSensitivity = generatedSensitivity + (difference * (random() + 3)) \quad (1)$$

In this equation random() will return a value from a Gaussian normal distribution with mean 0 and a standard deviation of 1.

When we look at the above equation we can see that if we add exactly $difference$ to $generatedSensitivity$ $P(estimatedSensitivity \geq realSensitivity) = 1$. However, plainly adding $difference$ will just leave us with the real sensitivity value again and all issues that come with it. The $random() + 3$ in Equation 1 aims to give us values that are (mostly) high enough to still leave us with $estimatedSensitivity \geq realSensitivity$ while removing any direct link between $realSensitivity$ and $estimatedSensitivity$. Using values returned from $random() + 3$ to decide how many times we want to add the difference leads us to the following theorem.

Theorem 5.1 *When neighbouring datasets share the same local sensitivity then* $P(estimatedSensitivity \geq realSensitivity) \geq 0.977$

We will now prove theorem 5.1. There are two cases to consider, the case $generatedSensitivity < realSensitivity$ and the case $generatedSensitivity \geq realSensitivity$ We can say

$$P(estimatedSensitivity \geq realSensitivity | generatedSensitivity < realSensitivity) = P(random() {+} 3 \geq 1) \tag{2}$$

To prove Equation 2 we need to look back to Equation 1. As said in the previous section, if we do

$$estimatedSensitivity = generatedSensitivity + (difference * 1) \tag{3}$$

$estimatedSensitivity$ will always be equal to $realSensitivity$, and therefore $P(estimatedSensitivity \geq realSensitivity) = 1$. Now if we replace the 1 in Equation 3 with $random() + 3$ we can say that for every $random() + 3 \geq 1$ $estimatedSensitivity \geq realSensitivity$. From this we can show Equation 2 to be true.

To understand why theorem 5.1 is true we will have to look into how a Gaussian distribution works.

A Gaussian normal distribution abides to the 68-95-99.7 rule. This rule tells us that 68% of all values will lie within 1 standard deviation from the mean, 95% within 2 standard deviations and 99.7% within 3 standard deviations. The standard deviation of the Gaussian normal distribution we use is 1. This means that for our Gaussian normal distribution with mean 0 the chance of a random value taken from this distribution being between -1 and 1 is 68%.

While we do first sample a value from our Gaussian normal distribution with mean 0, we then add 3 to the result. This way we shift the mean of the result of this computation to 3, while preserving a standard deviation of 1. This means that we can now say that 68% of values are between 2 and 4, and 95% of values between 1 and 5. From this we can say that at least 95% of the values returned from $random() + 3$ in Equation 1 are between 1 and 5. This already gives us $P(random() + 3 \geq 1) \geq 0.95$, but we are not quite there yet. In addition to all values between 1 and 5 giving a high enough $estimatedSensitivity$, all values over 5 also do so. Adding all values over 5 as well gives us

$$P(random() + 3 \geq 1) = 0.977 \tag{4}$$

If we combine Equation 2 and Equation 4 we get

$$P(estimatedSensitivity \geq realSensitivity \,|\, generatedSensitivity < realSensitivity) = 0.977 \tag{5}$$

With $generatedSensitivity < realSensitivity$ out of the way, we can now look at $generatedSensitivity \geq realSensitivity$. In this case

$$P(estimatedSensitivity \geq realSensitivity \mid generatedSensitivity \geq realSensitivity) \geq 0.9999 \tag{6}$$

Equation 6 follows from the only way of $estimatedSensitivity < realSensitivity$ if $generatedSensitivity \geq realSensitivity$ is if $random() + 3 <= -1$. This can be shown by looking at Equation 1. As $difference$ is absolute, the only way that $estimatedSensitivity < realSensitivity$ is when $difference$ is multiplied by at least -1. This leads us to

$$\text{P(estimatedSensitivity} \geq realSensitivity \mid generatedSensitivity \geq realSensitivity) = P(random() + 3 > -1) \tag{7}$$

As a value of -1 is 4 standard deviations from the mean we can say

$$P(random() + 3 > -1) \geq 0.9999 \tag{8}$$

Combining Equation 7 and Equation 8 we get Equation 6
Combining Equation 5 and Equation 6 gives us $P(estimatedSensitivity \geq realSensitivity) \geq 0.977$ and thus proves theorem 5.1.

5.3 Value of estimated sensitivity

In Section 5 we show guarantees on $P(estimatedSensitivity \geq realSensitivity)$. In this section we will look more in depth what the actual value of $estimatedSensitivity$ is. If we look at Equation 1 we can see that the most likely value

$$estimatedSensitivity = generatedSensitivity + (difference * 3) \tag{9}$$

This is due to the mean of our distribution being 0, and thus 0 being the theoretical most likely value. Now the question is, what does this actually say about the value of $estimatedSensitivity$?

There is actually still very little we can say about $estimatedSensitivity$. The most important thing to think about here is what the value of $generatedSensitivity$ and $difference$ are. The only real thing we know about these two variables is

$$generatedSensitivity + difference \geq realSensitivity \tag{10}$$

However, we do not know how much of $realSensitivity$ is in either one of those two variables. Therefore, we cannot give any guarantees about the value of Equation 1, other than what was done in Section 5.

There is one thing we can still say about $estimatedSensitivity$, it is most likely higher than $realSensitivity$. As we can see in Equation 9 it is very likely that $3 * difference$ is added on. The only time this will not guarantee $estimatedSensitivity \geq realSensitivity$ is when $estimatedSensitivity = realSensitivity$. However, as we cannot say whether $estimatedSensitivity = realSensitivity$ this is still not a guarantee.

This means that in any case where we only look at one query result the estimated sensitivity value says nothing about the real sensitivity. As we will see in 5.4, in production settings there should not be any type of unlimited querying and thus the estimated sensitivity will be very secure.

5.4 Including randomised response

Section 5 showed how our mechanism works without randomised response. In this section we will look how it works with randomised response. Randomised responses takes two values and randomly returns one or the other. Our implementation takes $realSensitivity$ and $estimatedSensitivity$, where

$$P(usedSensitivity = realSensitivity) = 0.75 \tag{11}$$

Where $usedSensitivity$ is the result of the randomised response mechanism.
From Equation 11 we can deduce that

$$P(usedSensitivity = estimatedSensitivity) = 0.25 \tag{12}$$

$P(usedSensitivity \geq realSensitivity)$ can be calculated using the following equation based on regular rules of probability

$$\begin{split} P(usedSensitivity \geq realSensitivity) \geq P(usedSensitivity = realSensitivity) + \\ P(usedSensitivity = estimatedSensitivity) * P(estimatedSensitivity \geq realSensitivity) \end{split} \tag{13}$$

If we substitute Equations 11, 12 and 5.1 into Equation 13 we can say

$$P(usedSensitivity \geq realSensitivity) \geq eq11 + eq12 * eq5.1 \tag{14}$$

Filling in the values for each equation we get $0.75 + 0.25 * 0.977 = 0.99425$ and thus we can make the following theorem.

Theorem 5.2 *When neighbouring datasets share the same local sensitivity and we use a randomised response mechanism with a 75% chance of returning $realSensitivity$ and a 25% chance of returning $estimatedSensitivity$ then* $P(usedSensitivity \geq realSensitivity) \geq 0.99425$

Using this theorem we can say that the mechanism as discussed in Section 4 there is a 99.425% chance of $usedSensitivity \geq realSensitivity$.

5.5 Implications for differential privacy

In Section 5.4 we show that for the mechanism as proposed in Section 4 the chance of $usedSensitivity \geq realSensitivity$ is 99.425%. While this looks quite high, the question is still what happens in the remaining 0.575% of cases. Differential privacy is not guaranteed if the sensitivity used to calibrate the noise is too low. This sounds like a big issue, however quite a lot of the damage of this can be mitigated. As Dwork puts it "All small epsilons are alike"[5]. This means that if the ϵ is sufficiently low there is a limited privacy leak if the result is 2ϵ-differentially private rather than ϵ-differentially private.

Additionally, if a query is executed twice and one of the results uses a high enough sensitivity while the other does not, there is still no privacy leak. Differential privacy protects from all linkage attacks, so a differentially private answer will not disclose any more information based on auxiliary knowledge of the user[5]. Due to this having one differentially private result is enough to not cause a privacy leak, as the differential privacy of this result means the information gained from the non-differentially private result cannot be used to discover more information.

Now this still leaves us with the question of why we would not just get a 100% guarantee, as then there is no risk whatsoever. The idea behind this stems from how a system like this would actually work in implementation. Currently queries are just executed without any sort of processing happening. In a production system there would be a lot more happening, for instance adding in caching so the same query on the same data always returns the same result. This type of extra protections will mean that, in general, doing the same query multiple times is a lot harder. Now, by setting the chance of $usedSensitivity \geq realSensitivity$ to a value slightly under 100% it means that from 1 query in no way can any information be found. If we set it at 100% we can say that the real sensitivity will never be lower than the value we see now, while with the value of 99.425% this information is also not disclosed.

Another question is why we would use randomised response at all, and not just $estimatedSensitivity$. As we saw in Section 5.3 there is very limited information on $estimatedSensitivity$. To understand why randomised response is still valuable we have to look at the exact value of $usedSensitivity$

rather than at $p(usedSensitivity \geq realSensitivity)$. Looking at that probability we only consider if the $usedSensitivity$ value will allow us to get a differentially private result but not if the result is as good as possible. $realSensitivity$ will always give us the best result while still preserving differential privacy, and thus the more we can use $realSensitivity$ the better it is for the accuracy of our final result. And as is shown in Equation 9, $estimatedSensitivity$ is most likely higher than $realSensitivity$ and thus give us lower accuracy.

The randomised response mechanism allows us to get the best of both worlds. There is a chance of using $estimatedSensitivity$ which says nothing about $realSensitivity$, or of using $realSensitivity$ which gives the most accurate result possible. Once again in a limited amount of queries it is impossible to tell which value is being used for $usedSensitivity$ and thus differential privacy is preserved in practice. While we do not hold to the exact constraints of differential privacy academically, our system is differentially private in at least 99.425% of cases. While the remaining cases are not differentially private, they are rare and when they do occur the chance of a privacy leak is still quite low. Therefore, our system will preserve differential privacy in practice even though it does not fully abide to the theoretical constraints.

6 Experiment setup

In this section we will discuss the benchmarks and metrics we will use to evaluate our system.

6.1 Metrics

There are a few metrics we will consider for our benchmarks. This section serves to explain them in more depth.

The first metric is the ***accuracy of the estimated sensitivity***. To check this we will look at the error of the estimated sensitivity versus the real sensitivity. This will allow us to check whether our mechanism produces sensitivity values near the local sensitivity. This method is not yet used by any paper, as the method of calculating local sensitivity in all papers so far is based on a mathematical calculations. This means that the accuracy of the sensitivity is directly related to the function used to create it, and therefore there is no need to examine the accuracy of the sensitivity.

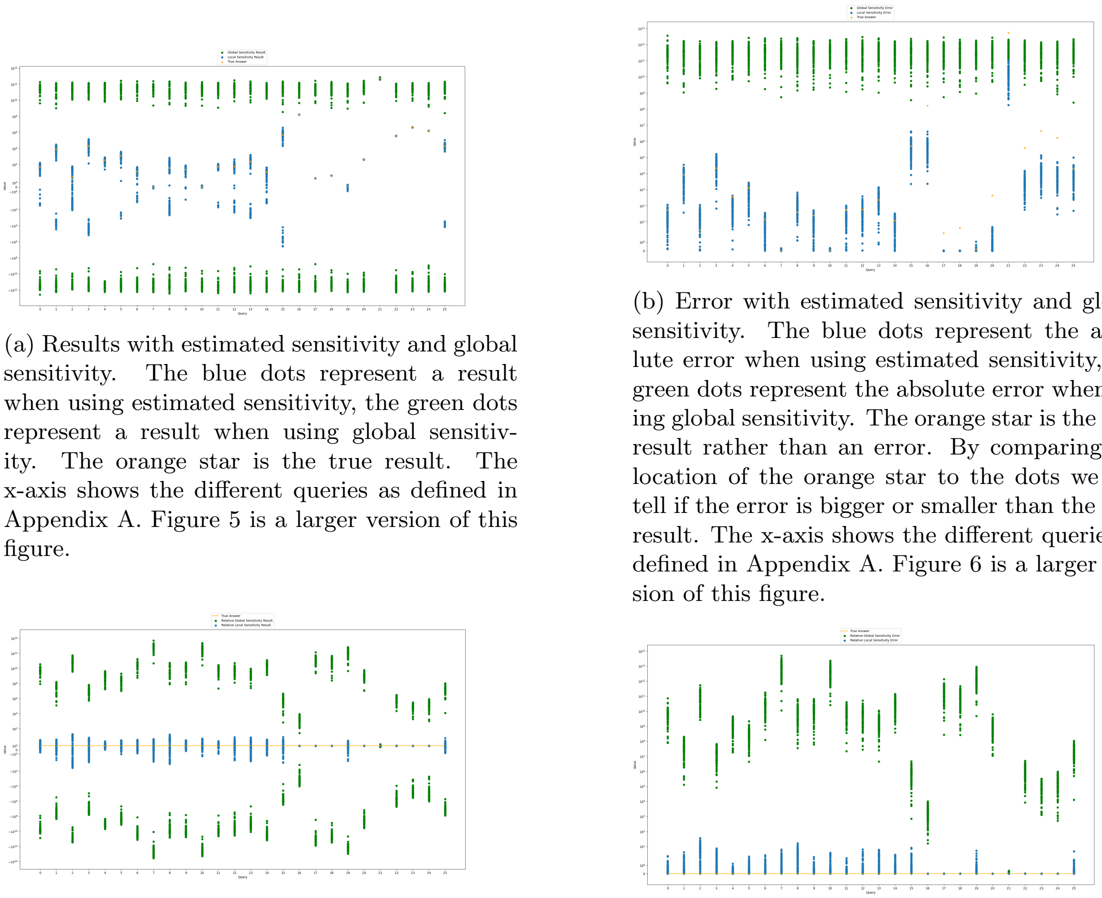
The second metric is ***accuracy of the result of the query using estimated sensitivity***. The idea of the research is that by using local sensitivity the value of the query result will be a lot closer to the true, unperturbed, value. This can be verified by putting the local sensitivity and the global sensitivity in a head to head contest, comparing the accuracy of each pair of values. This test might also be unnecessary, given that adding in noise with a lesser magnitude will automatically provide better accuracy, in which case we only need to verify whether the local sensitivity value is lower than the global sensitivity, as the noise is directly related to the sensitivity. This method is also used by both anomiGAN and elastic sensitivity [7][11].

The third metric would be ***time taken to answer a query***. As our method needs to generate a new synthetic dataset each time it answer a query, it will be slower than direct querying. If answering queries takes too long our system loses usability as it takes too long to get an answer to a query.

6.2 Benchmarks

We will use two benchmarks to compare our system against, differential privacy with global sensitivity and anomiGAN. In this section we will look at them both more in depth. The first benchmark is ***differential privacy using global sensitivity***[5]. Global sensitivity is the de facto standard for differential privacy at the moment. However, it is possible that our local sensitivity estimate is higher than the global sensitivity. If our sensitivity value is higher than the global sensitivity value rather than adding less noise, we will add more. This means our accuracy will go down, rather than up. We will have to verify that our local sensitivity actually improves upon global sensitivity. Comparison with this method would be on accuracy of the result.

The second benchmark will be ***anomiGAN***[11]. This approach mimics what we try to achieve, but does not give the guarantees of differential privacy. While it does generate data, rather than doing differential privacy as well, it just swaps the real data for the generated data. Comparison with this method would also be based on accuracy of the result. As the method of anomiGAN is based on replacing the real data with fake data and then executing the same operation on the fake data as the real data, we will use the GANs trained for local sensitivity estimation to get fake data and execute our queries on this data, taking the same number of rows of data as there are in the real dataset. While the code for the paper has been published we did not manage to get it working. So rather than using the exact implementation of anomiGAN we used the CTGAN models we also used to estimate the local sensitivity as data replacement models. This preserves the idea behind anomiGAN to replace the data with data generated by GANs that have been trained on the real data, and therefore we judge it as a good stand in for the true implementation.



(a) Results with estimated sensitivity and global sensitivity. The blue dots represent a result when using estimated sensitivity, the green dots represent a result when using global sensitivity. The orange star is the true result. The x-axis shows the different queries as defined in Appendix A. Figure 5 is a larger version of this figure.

(b) Error with estimated sensitivity and global sensitivity. The blue dots represent the absolute error when using estimated sensitivity, the green dots represent the absolute error when using global sensitivity. The orange star is the true result rather than an error. By comparing the location of the orange star to the dots we can tell if the error is bigger or smaller than the true result. The x-axis shows the different queries as defined in Appendix A. Figure 6 is a larger version of this figure.

(c) Relative result with estimated sensitivity and global sensitivity. For the values in this figure we have divided the results with estimated and global sensitivity by the true result. The orange line is at y=1 and represents the true result. The x-axis shows the different queries as defined in Appendix A. Figure 7 is a larger version of this figure.

(d) Relative error with estimated sensitivity and global sensitivity. This figure shows the errors divided by the true result. The orange line is at y=0 and represents the true result. The x-axis shows the different queries as defined in Appendix A. Figure 8 is a larger version of this figure.

Figure 2: Differential privacy with global and estimated sensitivity

7 Results

In this section we will discuss the results when using our mechanism. We will compare our mechanisms to (0.01, 0)-differential privacy with global sensitivity. We always have $\epsilon = 0.01$ and $\delta = 0$. In Section 7.5 we will show results for $epsilon = 0.1$ as well. For each query we have run both our algorithm and (0.01, 0)-differential privacy with global sensitivity (2^{32}) 100 times. The ground truth for all our queries can be found in Appendix A. When we refer to "estimated sensitivity" in this section, we refer to the result of our mechanism after randomised response.

7.1 Accuracy of answer

7.1.1 Differential privacy using global sensitivity

For the accuracy of the answer we look at the value returned by the (0.01, 0)-differential privacy mechanism for both the global sensitivity, which we set at 2^{32}, and the estimated sensitivity, which depends on the query and is dynamically generated by our mechanism. Figure 2a shows the results for both estimated and global sensitivity, and Figure 2b shows the absolute error of these results. Looking at both of them we can see that by using estimated sensitivity we are almost always closer to the answer than when using global sensitivity.

Figure 2c and Figure 2d, respectively, show the same data as the previous two figures, but relative to the true answer. Both these figures drive the difference home even better. Whilst the estimated sensitivity is always quite close to the true result, the global sensitivity is almost always far away from it. We can look at Table 17 for global sensitivity and Table 21 for estimated sensitivity for more details on this difference. Here we can see that the highest mean relative error is 6.85 for query 2, while the lowest is 0.0025 for query 20. If we look at the global sensitivity we see the highest mean relative error is $8.62 * 10^{12}$, while the lowest is 0.084.

7.1.2 anomiGAN

(ϵ,$sampleSize$)	Mean	Median	Min	Max
(0.01, 100000)	0.8372	0.2225	1.1918×10^{-5}	37.1703
(0.1, 100000)	0.0818	0.0213	7.5100×10^{-8}	2.8010
(0.01, 10000)	0.7929	0.2107	4.7808×10^{-9}	30.2441
(0.1, 10000)	0.0811	0.0211	1.7165×10^{-7}	3.6687
anomiGAN	0.1415	0.0439	3.4464×10^{-7}	0.8507

Table 5: Relative error of our method and anomiGAN method

We will now compare the accuracy of our result when using estimated sensitivity with the accuracy of the result when we used the anomiGAN principle. anomiGAN replaces all data by data generated by a GAN. For this we can use the GAN we trained to estimate the local sensitivity value. We then take as many rows of fake data as there are in the real data and execute the query on it. This then leaves us with a final result, for which we can once again look at the accuracy.

Table 26 shows the per query breakdown of the relative error when we use the anomiGAN method. Now in Table 5 we show the mean, median, min and max of the relative errors over all queries for our method and anomiGAN. Looking at this table we can see that anomiGAN gives a better accuracy than our method when $\epsilon = 0.01$, but when using $\epsilon = 0.1$ our system gets the edge. However, the maximum error of anomiGAN is in all cases better than our method.

7.1.3 Elastic sensitivity

Johnson et al. report for elastic sensitivity median errors between 10000% and 0.0001%, decreasing with the size of the population[7]. When we look at our median error, as seen in Table 21, we see that our median error is 1.0752 (107.52%) at highest and 0.4521 (45.21%) at lowest. There is no clear link between population size and error with our queries, which can be seen when we compare the errors for the different datasets in Table 21. While the Kidney Disease and Wisconsin Breast Cancer datasets have similar populations, with 400 and 569 rows respectively, the mean error on the Wisconsin Breast Cancer dataset is higher than on the Kidney Disease dataset. As with elastic sensitivity the lowest median errors are with a population size of 100 million, we cannot compare our lowest generated sensitivity error easily, as our maximum population is around 100000. However, we can say that with a low population size our system performs better than elastic sensitivity, as the error for elastic sensitivity is highest with a low population while for our method the mean error is as accurate on a small population as on a larger population.

7.2 Accuracy of estimated sensitivity

Section 7.1.1 shows the gain there is in accuracy of the result when we can use our method for estimating the sensitivity over global sensitivity. To verify whether our method meets the constraints set forth for differential privacy we have to check whether our estimated sensitivities are higher than real sensitivity in most cases, and how close they are to the real sensitivity.

First we will look if the estimated values are equal to or higher than the real sensitivity. As set out in section 5.5 not all estimated sensitivity values have to meet this constraint, so if a few fail this constraint the algorithm still passes. However, if more than 0.575% of the estimated sensitivity values fail this constraint, the algorithm does not hold to the standards as proposed in section 5. We have a total of 2600 runs with estimated sensitivity, spread across 26 queries. In these 2600 runs, 3 estimated sensitivity values are lower than the real sensitivity. This gives us a total of 0.115% too low values. As this is less than 0.575% our method passes this check.

Second, we want to explore how accurate the estimated values are. Figure **??** shows the estimated sensitivity values relative to the real sensitivity. In this graph we can already see that there is a high concentration of values near y=1, which represents the real sensitivity. If we look at Table 20, which contains more detailed statistics on the estimated sensitivity relative to the real sensitivity, we see that the mean estimated sensitivity value relative to the real sensitivity for each query lies between 1 and 1.6. The lowest mean is 1.035 and the highest is 1.59. 50% of the queries have a mean of less than 1.10, meaning that there is only a 10% difference between the estimated sensitivity and the real sensitivity. More data can be found in Figures 9 through 12 and in Tables 18 through 21, which show both absolute estimated sensitivity values and errors and relative to the real sensitivity.

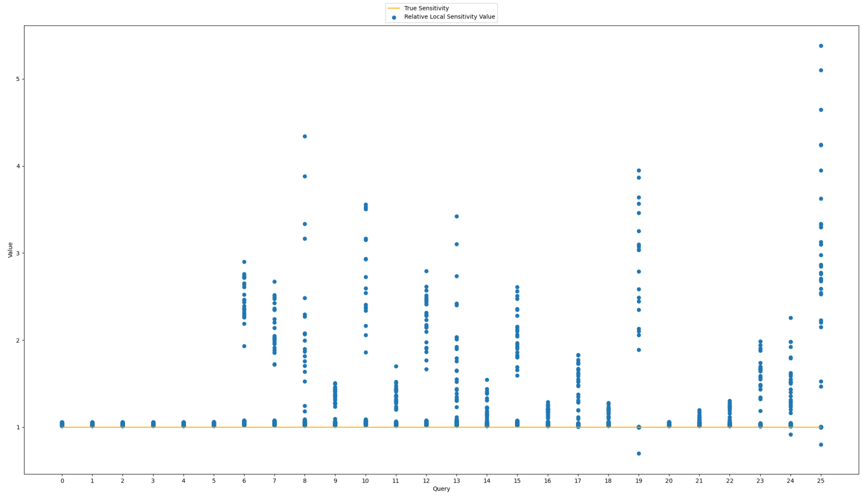
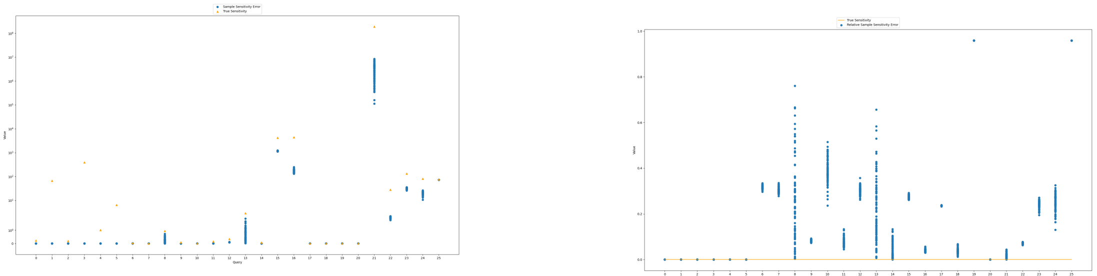


Figure 3: Relative estimated sensitivity. This figure shows the estimated sensitivity value divided by the real sensitivity. The orange line is at y=1 and represents the real sensitivity. The x-axis shows the different queries as defined in Appendix A. Figure 11 is a larger version of this figure.



(a) Generated sensitivity error. The blue dots show a generated sensitivity error, the orange triangles show the real sensitivity. Comparing the location of the blue dot with the orange triangle shows us whether the error is more or less than the real sensitivity. The x-axis shows the different queries as defined in Appendix A. Figure 14 is a larger version of this figure

(b) Relative generated sensitivity error. This figure shows the generated sensitivity error value divided by the real sensitivity. The orange line is at y=0 and represents the real sensitivity. The x-axis shows the different queries as defined in Appendix A. Figure 16 is a larger version of this figure

Figure 4: Generated sensitivity error

7.3 Accuracy of generated sensitivity

As we have seen in the last section, the estimated sensitivity used in the calculation is both high enough and quite accurate. Now we will dive into this a bit deeper and consider the underlying generated sensitivity, the sensitivity of the sample generated by our GAN model. The most important aspect here is the error of the generated sensitivity, as this is the distance we use in our method.

When we look at the absolute error between the generated and the real sensitivity in Figure 4a it looks like quite a few generated sensitivities are close to the truth, and all that are not have an error lower than the real sensitivity. While this is quite promising, if we look at Figure 4b, which shows the same data but relative to the real sensitivity, there are more differences than there appeared to be in the previous figure. However, looking at this figure we once again see that the relative error is always under 1, and for most queries under 0.4. This means that, while there is an error, the generated sensitivity values are not very far off the real sensitivity.

When we then look at Table 25 we can see there are 7 queries where the generated sensitivity is equal to the real sensitivity in all of our runs, and the other 19 queries all have a difference between the real and the generated sensitivity in each run. More details on the absolute generated sensitivity values and the values relative to the real sensitivity are shown in Figures 13 through 16 and in Tables 22 through 25.

7.4 Bringing it all together

We have looked at 3 aspects of the mechanism, the final result, the estimated sensitivity value used, and the sensitivity on the dataset generated by our GAN model. Until now we have looked at them individually, but if we look at them all together there are some effects visible. The most important connection is between the generated sensitivity and the estimated sensitivity value. We will use Table 21 for the relative estimated sensitivity error and Table 25 for the relative generated sensitivity error. In this section we will be comparing groups of queries which have similar errors for the generated sensitivity values.

For the 7 queries where the generated sensitivity is always equal to the real sensitivity we see that the estimated sensitivity value is off by about 3.5-3.8%. This is due to the flat noise we add on after the randomised response mechanism, and is to be expected. If we then look at the two queries with a high error on the generated sensitivity, queries 19 and 25, we see something quite different. While they both have a mean error of 95.91% on the generated sensitivity relative to the real sensitivity, the mean error on the estimated sensitivity value relative to the real sensitivity is quite different. Query 19 clocks in at a mean relative error of 36.25%, while query 25 has a mean relative error of 59.21%. If we look at other queries where the mean relative generated sensitivity is similar (queries 6, 7 and 12 and queries 23 and 24) we once again see that the difference in estimated sensitivity value, while closer, is not the same and in all cases different by at least 1.5%. If we look at queries 7 and 12, which are just 0.14% apart in relative generated sensitivity error, they are 2.3% apart in relative estimated sensitivity error.

When we then look at the result of the query, we see even more differences. We will look at Table 16 for the error of the result of the query relative to the true answer for each query. If we once again look at the queries were the generated sensitivity is always equal to the real sensitivity, we see a big difference in accuracy of the result. For query 2, the mean relative error is 685.47%, while for query 20 it is only 0.25%. This means that the error in the final result cannot be only due to the error in the generated sensitivity, and there are other factors at play as well. This can be confirmed

(ϵ,*sampleSize*)	Mean	Median	Min	Max
(0.01, 100000)	0.8372	0.2225	1.1918×10^{-5}	37.1703
(0.1, 100000)	0.0818	0.0213	7.5100×10^{-8}	2.8010
(0.01, 10000)	0.7929	0.2107	4.7808×10^{-9}	30.2441
(0.1, 10000)	0.0811	0.0211	1.7165×10^{-7}	3.6687

Table 6: Relative accuracy with different epsilons and sample sizes

(ϵ,*sampleSize*)	Mean	Median	Min	Max
(0.01, 100000)	0.1946	0.0783	0	0.9597
(0.1, 100000)	0.1949	0.0794	0	0.9595
(0.01, 10000)	0.1951	0.0774	0	0.9600
(0.1, 10000)	0.1952	0.0789	0	0.9600

Table 7: Relative generated sensitivity error with different epsilons and sample sizes

by once again looking at the other groups of values with mean relative generated sensitivity errors close together, like query 7 and 12, Query 7 has a mean relative error on the result of 104.89%, while query 12 has one of 74.65%.

By looking at these different groups of queries with start with a mean relative generated sensitivity error which is quite close, but end up with mean relative errors on the results of the queries which are farther apart, we can say there is no direct link between mean relative generated sensitivity error and mean relative result error.

7.5 Parameter choice

The epsilon and amount of data generated for our experiment do not have a strong rationale behind them. ϵ choice impacts the privacy of the result, so needs to be selected depending on the needs of the dataset. The size of the generated data also is quite arbitrary. While with a higher amount of generated data we are more likely to get outliers, in essence it does not matter for our method if those outliers are or are not generated, so the choice in size is actually also quite irrelevant. The only advantage one might get from a higher sample size is the possibility of a more accurate generated sensitivity, which in turn also leads to a more accurate end result. This is also up to the end user to configure. considering time taken versus quality of the answer. We will now show some important data for when we change ϵ to 0.1 rather than 0.01, *sampleSize* to 10000 rather than 100000 and those two changes combined.

Firstly we will look at the effect of changing ϵ and *sampleSize* on the accuracy of our final result. The data for this, compiled over all queries, can be seen in Table 6. Here we can see a large improvement when we change ϵ from 0.01 to 0.1. However, we do not see a nearly as big an improvement when changing *sampleSize*. We do see as small improvement but this might be due to the inherent randomness that is present in the method. More extensive testing will need to be done to see if this improvement comes from the changed *sampleSize* or from the randomness in the method.

Secondly, we will consider the effect of changing ϵ and *sampleSize* on the accuracy of our generated sensitivity. We once again compiled data over all queries, and display the results for this in Table 7. Here we see all values are quite close. From this table we can deduce that, with these

(ϵ,*sampleSize*)	Mean	Median	Min	Max
(0.01, 100000)	9.1471	7.3797	3.7775	4805.3389
(0.1, 100000)	7.6755	7.9737	3.6358	57.3049
(0.01, 10000)	0.7289	0.6129	0.3665	3.5788
(0.1, 10000)	0.7762	0.6771	0.3226	37.5916

Table 8: Time taken to answer a query for each combination in seconds

(ϵ,*sampleSize*)	Mean	Median	Min	Max
(0.01, 100000)	7.2908	7.3819	3.7775	21.3477
(0.1, 100000)	7.6640	7.9661	3.6358	57.3049
(0.01, 10000)	0.7275	0.6128	0.3665	1.2460
(0.1, 10000)	0.7610	0.6761	0.3226	1.8619

Table 9: Time taken to answer a query for each combination in seconds without initialisation

values, there is no large difference in the accuracy of the generated sensitivity. Now that leaves us with the question of which values to actually use.

Firstly let us look at ϵ. ϵ only has an effect on the accuracy of the final result as it is used to scale the noise. Setting ϵ is always an important choice when using differential privacy. As we can see in Table 6 a higher epsilon will lead to a higher accuracy. However, it will also lead to lower privacy. Therefore, the choice of ϵ has to be based on the privacy and accuracy needs of the system.

Now we will take a look at *sampleSize*. *sampleSize*'s main effect is on the accuracy of the generated sensitivity. As we see in Table 7 this is very minor for the different values. Based on this we cannot necessarily say anything about which value to use. However, there is something we have yet to consider. Most of the time taken to answer queries comes from generating the fake data. This means that using less rows of generated data will allow us to answer queries faster. This can be seen in Table 8. This table shows how long it takes to answer queries, however, there is one note to make. In general, the first time we execute a query on specific dataset will take longest to answer, as the model has to initialise. Therefore, Table 9 shows the same data as Table 8, only without the first execution. From either one of those tables we can see that $sampleSize = 10000$ is roughly 10 times as fast as $sampleSize = 100000$. Therefore, in the general use case, using a lower *sampleSize* is better for speed. There might be a drop in generated sensitivity accuracy when *sampleSize* drops too low, but this has not been researched. All our experiments were run on a Windows 10 machine with a 2.2GHz Intel Core I7 and 8GB of RAM, using Python 3.8.2.

8 Conclusion

In this thesis we explored a way to increase the utility of datasets by using differential privacy combined with GANs. Our main goal was to find a method that allowed us to use a sensitivity value based on the local sensitivity of a dataset, while preserving differential privacy. This led us to the following main research question:

Can GANs be used to estimate the local sensitivity of a function for use in differential privacy?

We then specified 3 subquestions, to which we will briefly summarise the answers here.

Can GANs estimate a sensitivity value which satisfies the requirements of differential privacy? We have demonstrated how our method can generate sensitivity values that satisfy the requirements of differential privacy in many cases. It is possible for our method to generate values that do not satisfy the requirements of differential privacy, but it only rarely does so. In our experiment only 0.115% of the estimated sensitivity values were too low. The privacy risk of these too low values are also minimal.

Does our estimated sensitivity value disclose anything about the dataset? By allowing sensitivity values that do not strictly satisfy the requirements of differential privacy we completely remove any link between our estimated sensitivity and the real sensitivity. This allows for us to say that the estimated sensitivity value does not disclose any information about the real sensitivity value, or the real dataset.

Does our estimated sensitivity value give us a higher accuracy on the result of the query than when using global sensitivity? We have shown that by using our method we get more accurate results over using global sensitivity. This change was most obvious in cases where the real sensitivity was quite low, as in those cases the difference between the global and the estimated sensitivity was the largest.

Conclusion of the main research question

We have shown that by using a GAN we can increase the accuracy of the result of an answer, and our estimated sensitivity does not disclose anything about the real dataset. However, we have not created a method that strictly abides by the definition of differential privacy. By allowing too low sensitivities to be used we have introduces a potential privacy risk, however we have also shown this risk to be minimal and the benefits of allowing too low sensitivities. While our system does not abide to the theoretical definition of differential privacy, we believe that in practice differential privacy will be preserved.

However, to deploy our method into the real world more work is needed. Partly this is on the research side, there are still many unanswered questions on how to use this system and the exact benefits of it. Our system can be used as a great starting point for further research in how approaches combining real and generated data can help preserve privacy as best as possible, while providing researchers with the valuable data they need. The other part that still needs to be done is all the auxiliary work that comes with a differential privacy application. If such an application is set up properly our system for generating the local sensitivity value can be used, rather than using global sensitivity. But if our system gets used without any of the requirements set out for a differential privacy system, which we did not go into in this thesis, privacy risks are still inevitable. Differential privacy has a great potential to preserve privacy while increasing utility, but it has to be used with care.

9 Discussion

The method we proposed is still in its infancy, there are many directions left to research. We will look at limitations in two distinct categories. Firstly, we will discuss the current limitations of our method with differential privacy. Secondly, we will discuss the limitations of our GAN and how this can be improved. After considering the limitations we will propose future research topics.

9.1 Limitations

9.1.1 Limitations concerning differential privacy

So far we have only explored how our mechanism interacts with the Laplace mechanism. This also means that the queries that can be answered by our mechanism now are rather limited. However, there are other mechanism out there for differential privacy, such as the exponential mechanism. By adding support for the exponential mechanism other types of queries can also be supported, such as `MIN()` and `MAX()` queries.

Another limitation is the exact constraints of differential privacy. We have shown that our value is almost always higher than the real sensitivity, however differential privacy dictates this always be the case. Therefore, we cannot give a strict guarantee of differential privacy in the academic sense. However, in practice differential privacy should be all but guaranteed, as long as other required measures are being used to protect the privacy.

Directly building onto this is the fact that we have only tested our system in isolation. A differential privacy system in practice should have mechanisms to prevent unlimited querying and other actions that can infringe on this. Incorporating this is of utmost importance before the system can be used in practice.

Another important limitation is that we have only included static datasets in our experiment. Static datasets are less complex to preserve the privacy of than dynamic datasets. Due to no new data being added, by taking the sensitivity of the full dataset we know the highest possible sensitivity. In a dynamic dataset new data can be added that make the sensitivity of the function higher, thus leading to issues when trying to preserve differential privacy.

9.1.2 Limitations concerning GANs

We have not taken an extensive look into the exact properties of the GAN, but rather left it as rather a black box. We have shown that our method will preserve privacy no matter what is returned by the GAN, but we have also seen that the expected value returned is $realSensitivity + 2 * difference$. Our method works for a truly random value, however the accuracy will be quite low. As we would still like our accuracy to be as high as possible, extra guarantees on the GAN might improve the method even further.

We have also only looked at a single dataset, without including any joins from additional datasets. Adding joins will add in more complexity in generating data. A very important question here is how will we do a join on generated data, and what will the effects of it be on the result. As in many real world applications multiple datasets get linked in a query, this is still an important limitation of the system.

Another aspect of the GAN we have not looked into in depth is the correlations between variables in the generated dataset. We mentioned briefly, whilst comparing the real data and a generated dataset, that correlations are still lacking in the GAN. When introducing more complex queries to our system it might be necessary that the lack of accurate correlations can disclose information on whether we are using a generated sensitivity rather than a real sensitivity.

9.2 Future research

There are a lot of interesting questions still left open after this research, both on the differential privacy and the machine learning side.

One such question is how our system can be integrated in different differential privacy mechanisms. As our method only calculates the sensitivity it should be able to work with any of the currently proposed differential privacy mechanisms, but this has not yet been researched.

Another question that can be explored further is the effect of the size of the generated dataset. We have shown for 100000 and for 10000 rows that accuracy is quite close, while using fewer rows gives a big boost in speed. More research can be done to see what the trade-offs of using more, or fewer, rows is on the accuracy of both *generatedSensitivity* and the final answer.

A third interesting question is if our method can also work with dynamic databases. As we use a generated dataset to estimate the sensitivity, it could be possible that the generated set also allows us to cover dynamic datasets. As dynamic datasets are currently still a big issue for differential privacy, research into this question could be very interesting.

Something we have not discussed at all in this thesis is the time taken to train a GAN, nor have we explored how a GAN trained on different subsets of the data can influence *sampleSensitivity*. An interesting question would be what would happen if we trained a GAN only on outliers of the data. Training a GAN on fewer records makes the training go faster, and the most interesting data for privacy are the outliers. A question that can follow from this would be what would happen if we combined several GANs to create our dataset, rather than just using one. We have briefly seen that training a GAN on fewer records leads to the outliers in this data to be more visible when we generate a new dataset. If we were to combine a GAN trained on outliers with a GAN trained on the whole dataset, would we get a more accurate generated dataset than we do now? There are many more questions possible in this direction, and exploring how to increase the accuracy of the generated data is most likely one of the most important aspects for increasing the accuracy of the method in general.

Another thing to look into could be the probability distribution used in the estimated sensitivity calculation. We currently use a Gaussian normal distribution, however this can be replaced with any other probability distribution as long as $P(estimatedSensitivity \geq realSensitivity)$ is still high enough. An example here could be using a Laplace distribution rather than a Gaussian normal distribution, which might lead to more accurate estimated sensitivity on average.

Appendix A Query Data

Number	Query	Dataset	Query Type	True Result	True Sensitivity
0	SELECT AVG(age) FROM kidney	kidney	AVG	51.4834	0.2256
1	SELECT AVG(wbcc) FROM kidney	kidney	AVG	8425.7730	66.1654
2	SELECT AVG(sc) FROM kidney	kidney	AVG	3.0725	0.1905
3	SELECT SUM(bu) FROM kidney	kidney	SUM	2.1879×10^4	391
4	SELECT SUM(sg) FROM kidney	kidney	SUM	359.1450	1.0250
5	SELECT SUM(rbcc) FROM kidney	kidney	SUM	1249.1000	6.5000
6	SELECT AVG(radius_mean) FROM breast	breast	AVG	14.1273	0.0495
7	SELECT AVG(concave_points_mean) FROM breast	breast	AVG	0.0489	3.5400×10^{-4}
8	SELECT AVG(area_se) FROM breast	breast	AVG	40.3371	0.9546
9	SELECT AVG(texture_worst) FROM breast	breast	AVG	25.6772	0.0872
10	SELECT AVG(fractal_dimension_worst) FROM breast	breast	AVG	0.0839	3.6500×10^{-4}
11	SELECT SUM(smoothness_mean) FROM breast	breast	SUM	54.8290	0.1634
12	SELECT SUM(compactness_mean) FROM breast	breast	SUM	59.3700	0.3454
13	SELECT SUM(radius_se) FROM breast	breast	SUM	230.5429	2.8730
14	SELECT SUM(symmetry_se) FROM breast	breast	SUM	11.6886	0.0790
15	SELECT SUM(area_worst) FROM breast	breast	SUM	5.0105×10^5	4254
16	SELECT AVG(encounter_id) FROM diabetes	diabetes	AVG	1.6520×10^8	4361.6880
17	SELECT AVG(admission_type_id) FROM diabetes	diabetes	AVG	2.0240	7.8600×10^{-5}
18	SELECT AVG(time_in_hospital) FROM diabetes	diabetes	AVG	4.3960	1.3800×10^{-4}
19	SELECT AVG(number_emergency) FROM diabetes	diabetes	AVG	0.1978	7.4700×10^{-4}
20	SELECT AVG(diag_2) FROM diabetes	diabetes	AVG	438.6749	0.0098
21	SELECT SUM(patient_nbr) FROM diabetes	diabetes	SUM	5.5290×10^{12}	1.8950×10^8
22	SELECT SUM(discharge_disposition_id) FROM diabetes	diabetes	SUM	3.7813×10^5	28
23	SELECT SUM(num_lab_procedures) FROM diabetes	diabetes	SUM	4.3857×10^6	132
24	SELECT SUM(num_medications) FROM diabetes	diabetes	SUM	1.6305×10^6	81
25	SELECT SUM(number_emergency) FROM diabetes	diabetes	SUM	2.0133×10^4	76

Appendix B Results Graphs

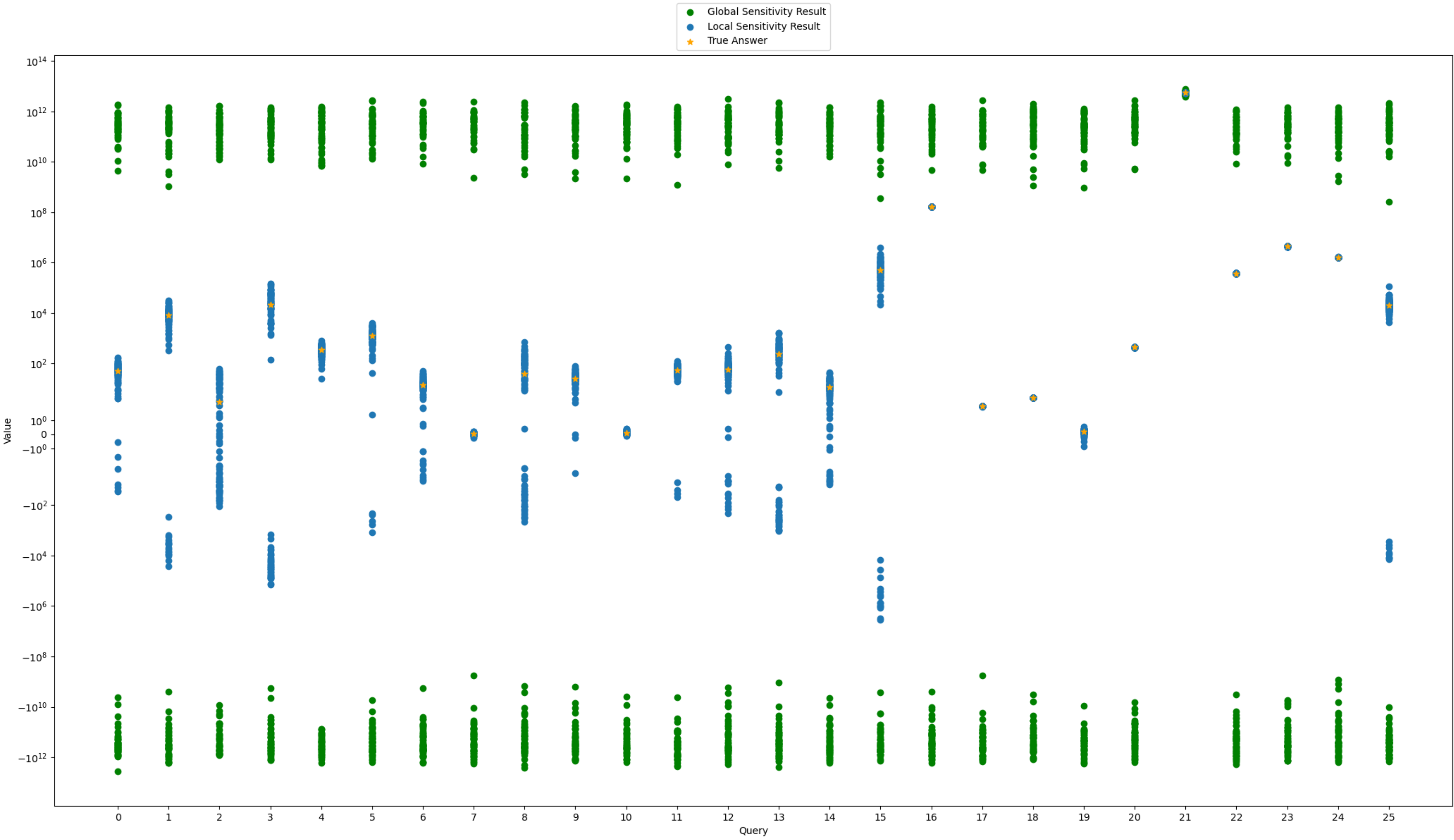


Figure 5: Results with estimated sensitivity and global sensitivity. The blue dots represent a result when using estimated sensitivity, the green dots represent a result when using global sensitivity. The orange star is the true result. The x-axis shows the different queries as defined in Appendix A.

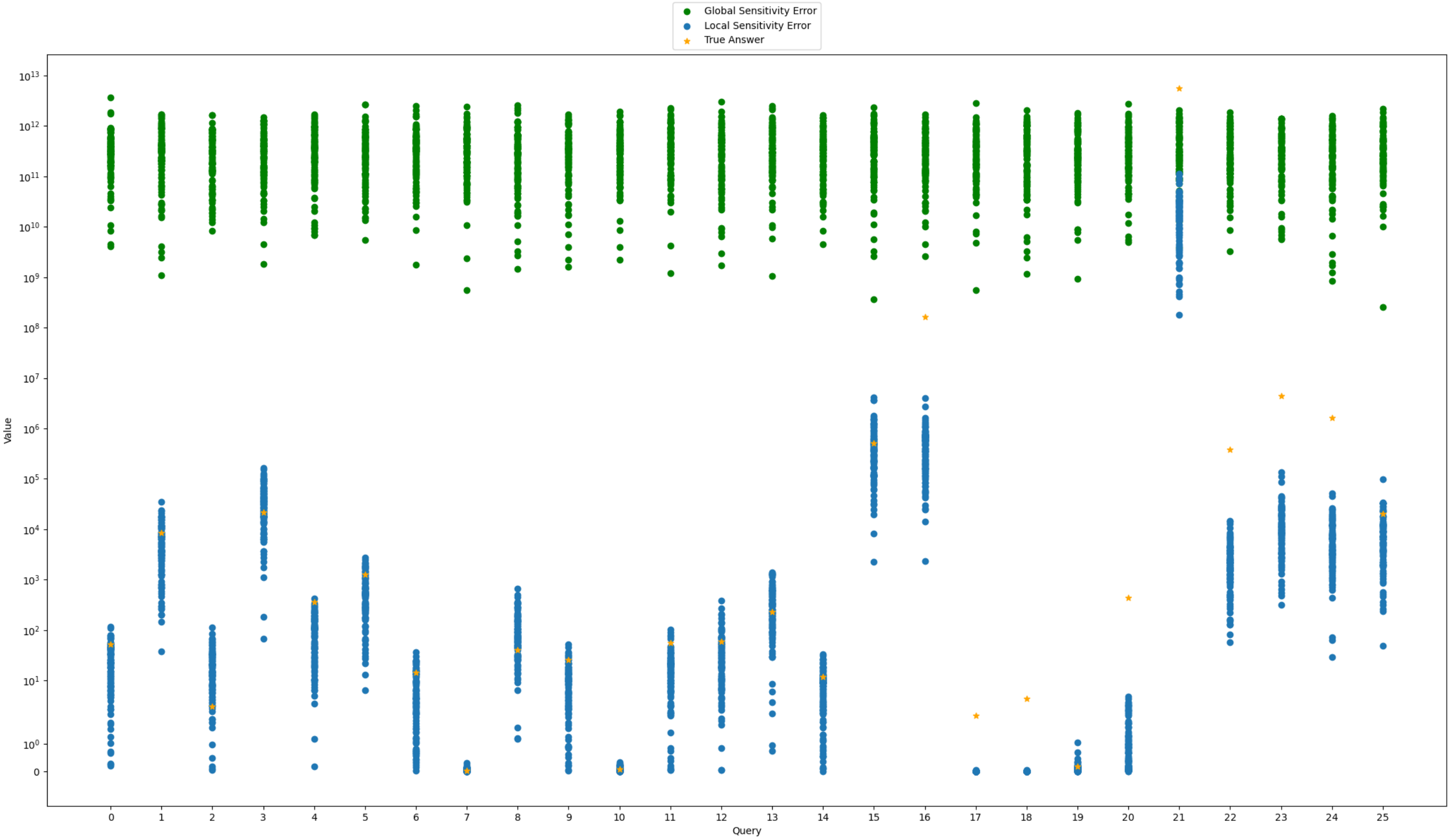


Figure 6: Error with estimated sensitivity and global sensitivity. The blue dots represent the absolute error when using estimated sensitivity, the green dots represent the absolute error when using global sensitivity. The orange star is the true result rather than an error. By comparing the location of the orange star to the dots we can tell if the error is bigger or smaller than the true result. The x-axis shows the different queries as defined in Appendix A.

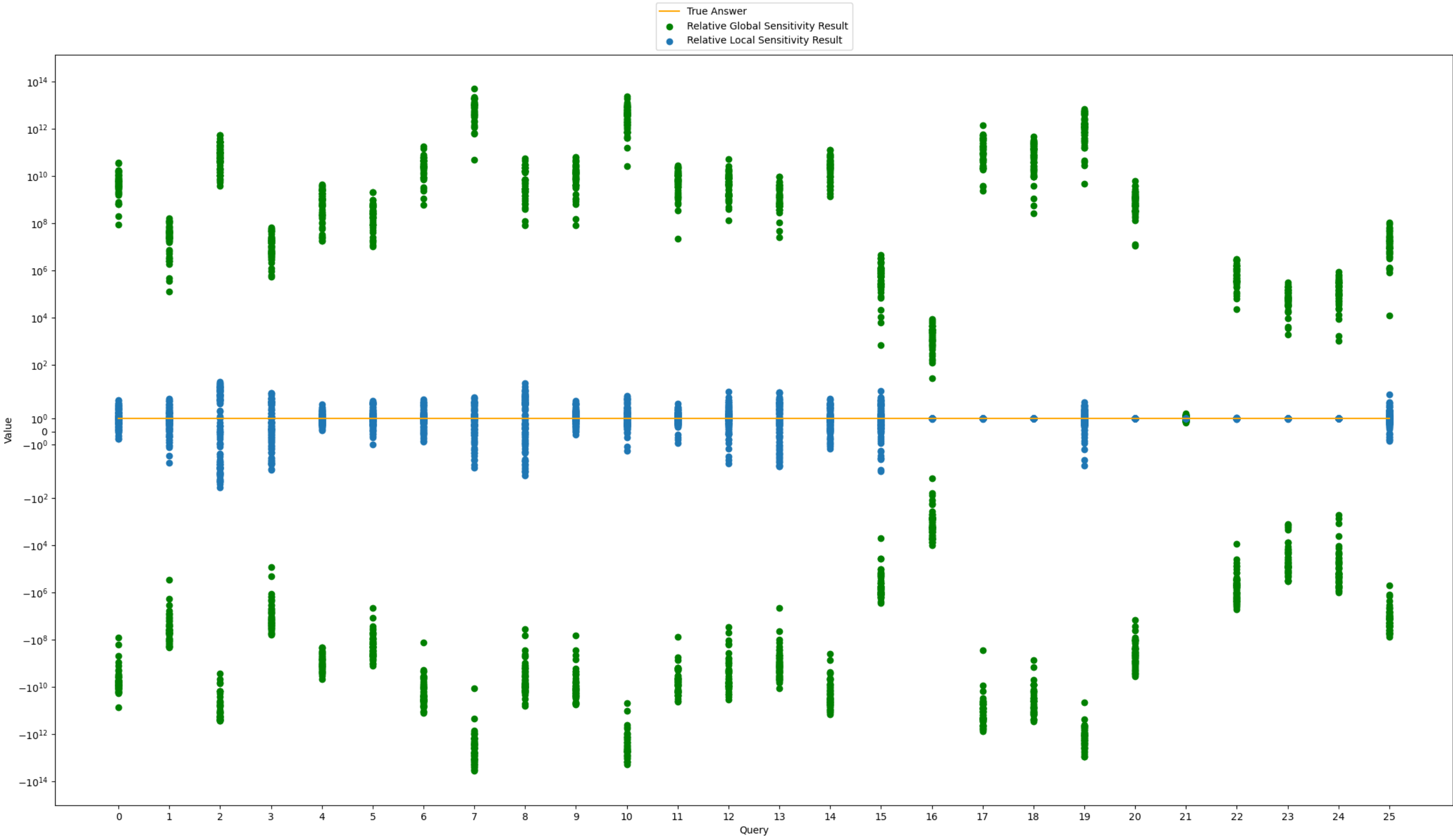


Figure 7: Relative result with estimated sensitivity and global sensitivity. For the values in this figure we have divided the results with estimated and global sensitivity by the true result. The orange line is at y=1 and represents the true result. The x-axis shows the different queries as defined in Appendix A.

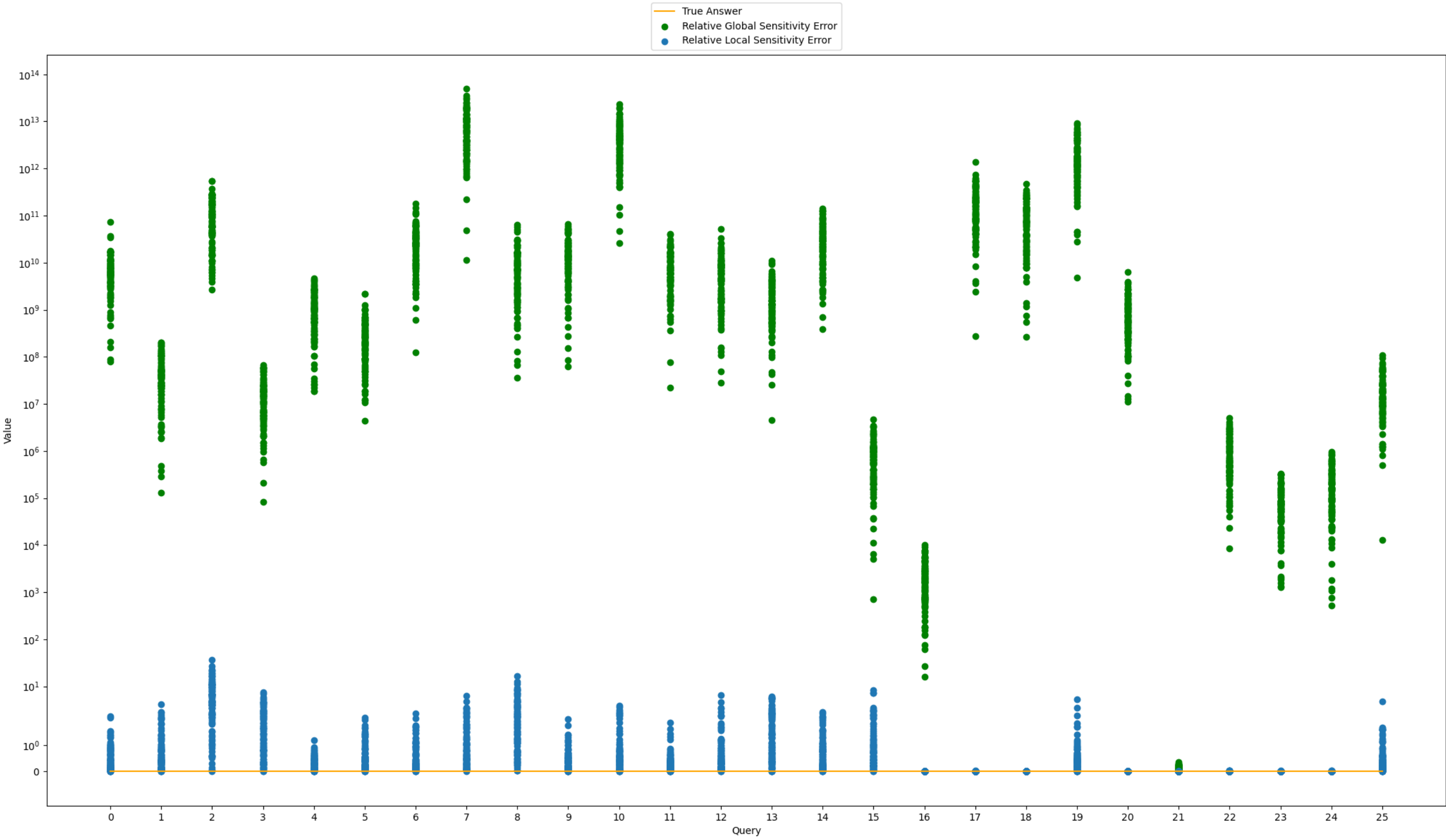


Figure 8: Relative error with estimated sensitivity and global sensitivity. This figure shows the errors divided by the true result. The orange line is at y=0 and represents the true result. The x-axis shows the different queries as defined in Appendix A.

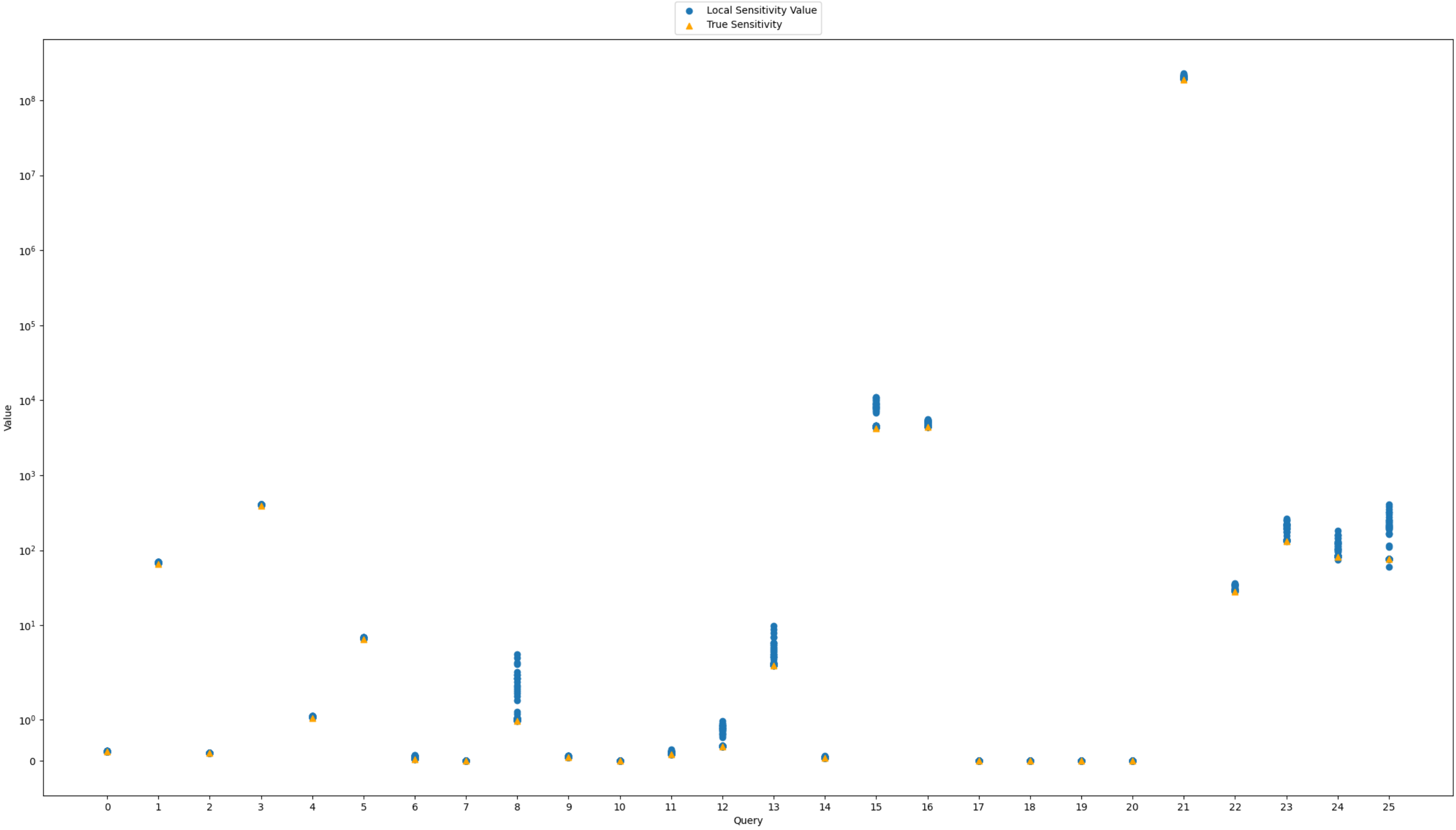


Figure 9: Estimated sensitivity value. The blue dots show a estimated sensitivity value, the orange triangles the real sensitivity. The x-axis shows the different queries as defined in Appendix A.

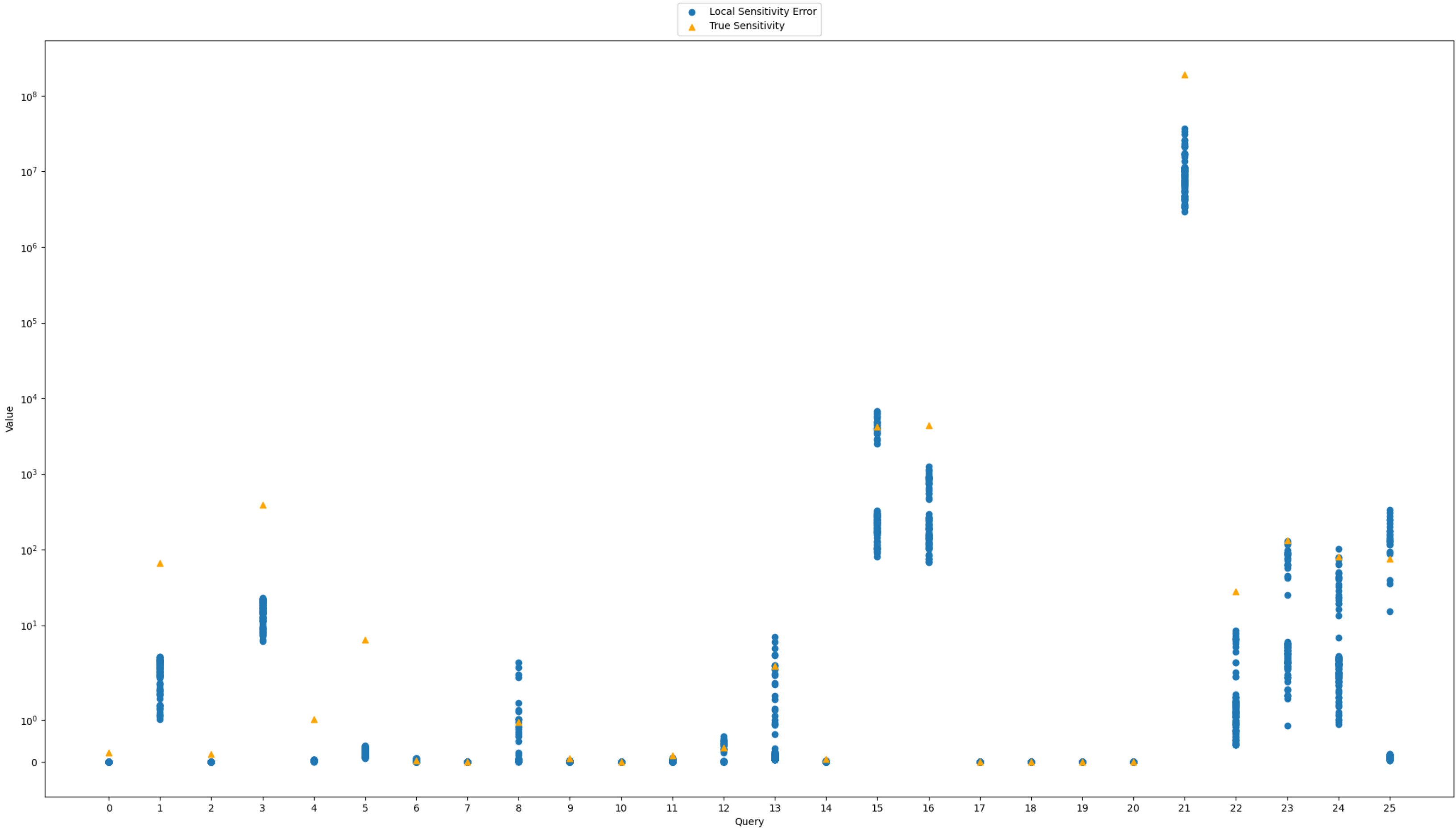


Figure 10: Estimated sensitivity error. The blue dots show a estimated sensitivity error, the orange triangles show the real sensitivity. Comparing the location of the blue dot with the orange triangle shows us whether the error is more or less than the real sensitivity. The x-axis shows the different queries as defined in Appendix A.

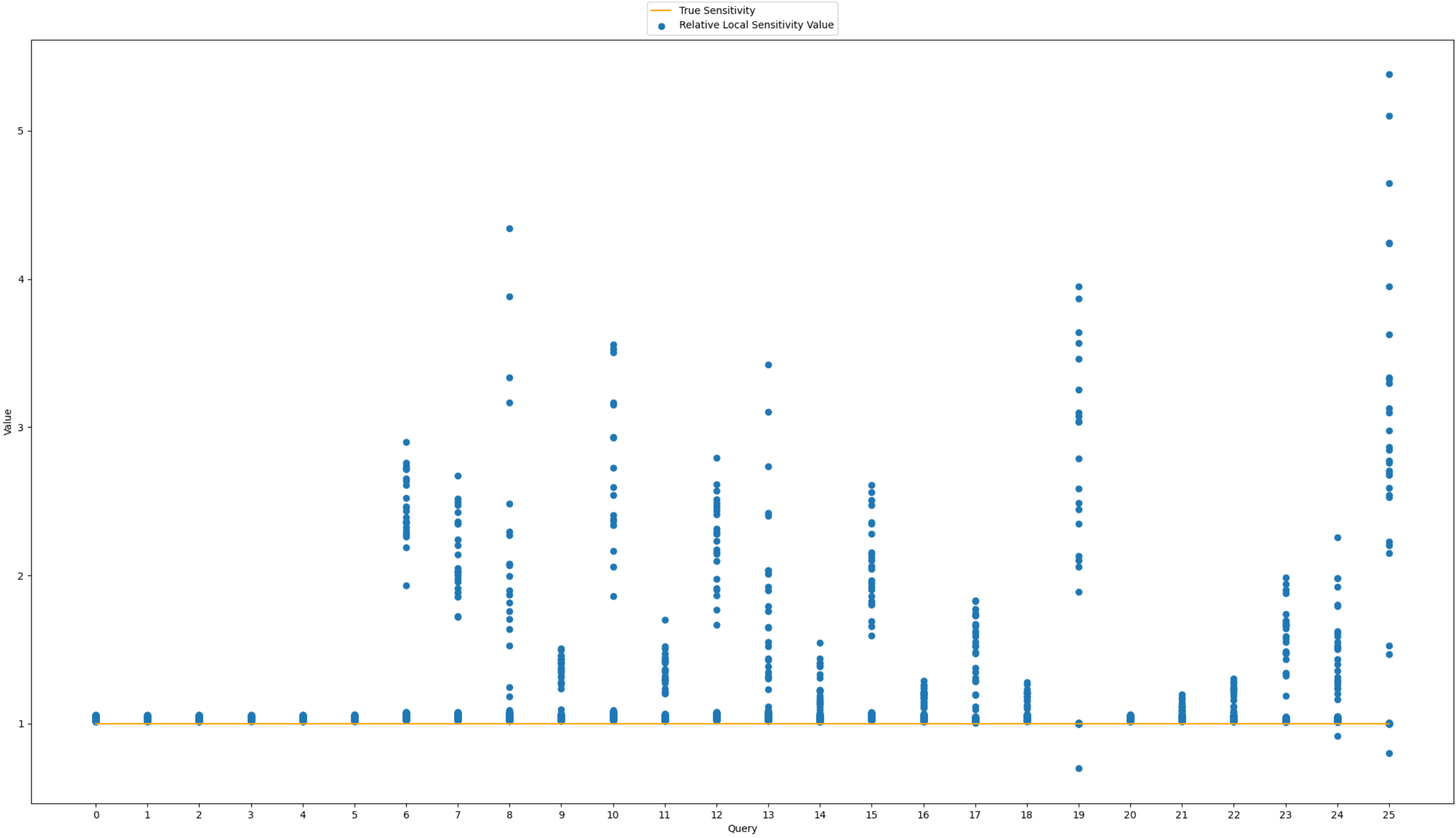


Figure 11: Relative estimated sensitivity. This figure shows the estimated sensitivity value divided by the real sensitivity. The orange line is at y=1 and represents the real sensitivity. The x-axis shows the different queries as defined in Appendix A.

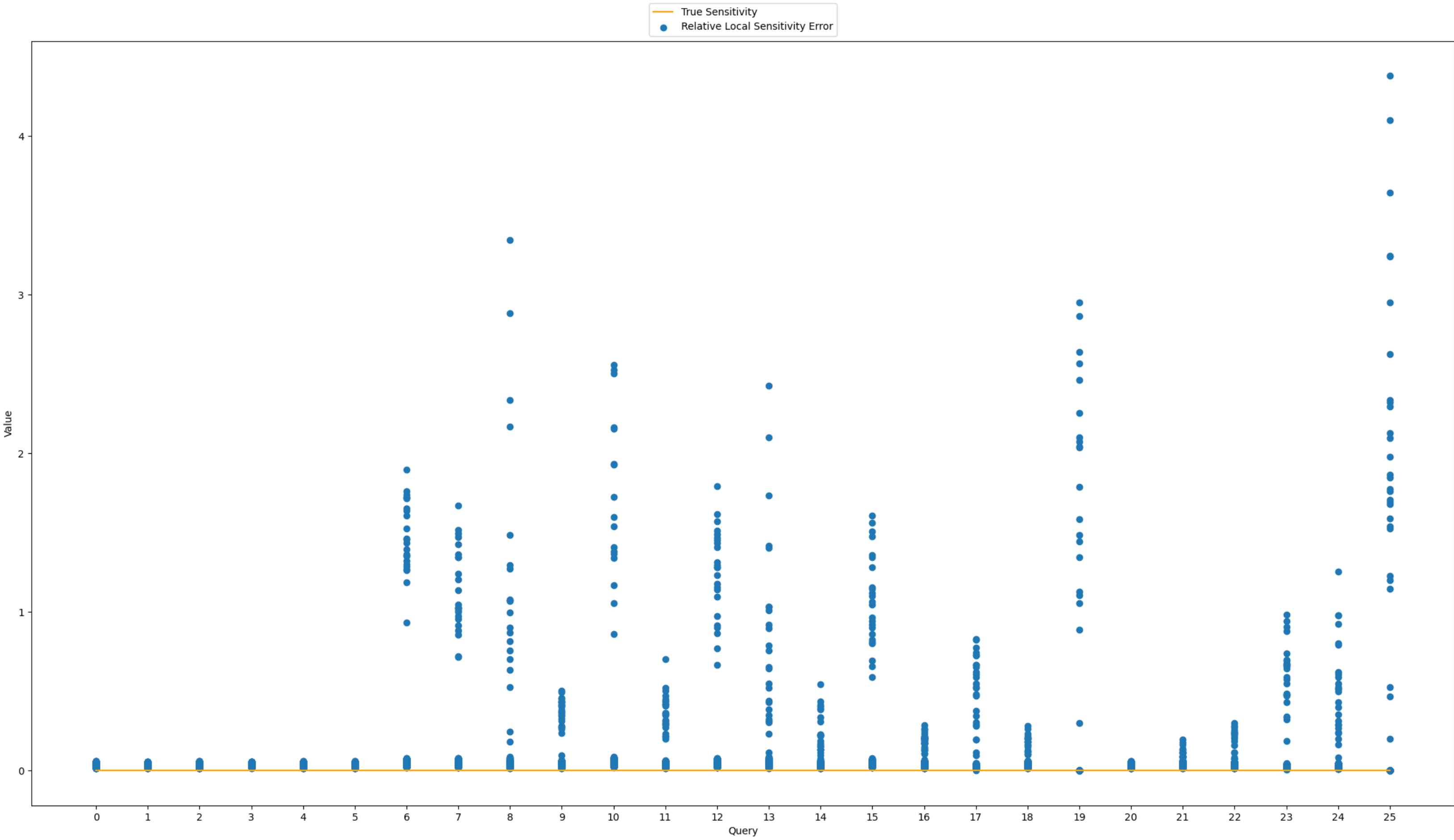


Figure 12: Relative estimated sensitivity error. This figure shows the estimated sensitivity error value divided by the real sensitivity. The orange line is at y=0 and represents the real sensitivity. The x-axis shows the different queries as defined in Appendix A.

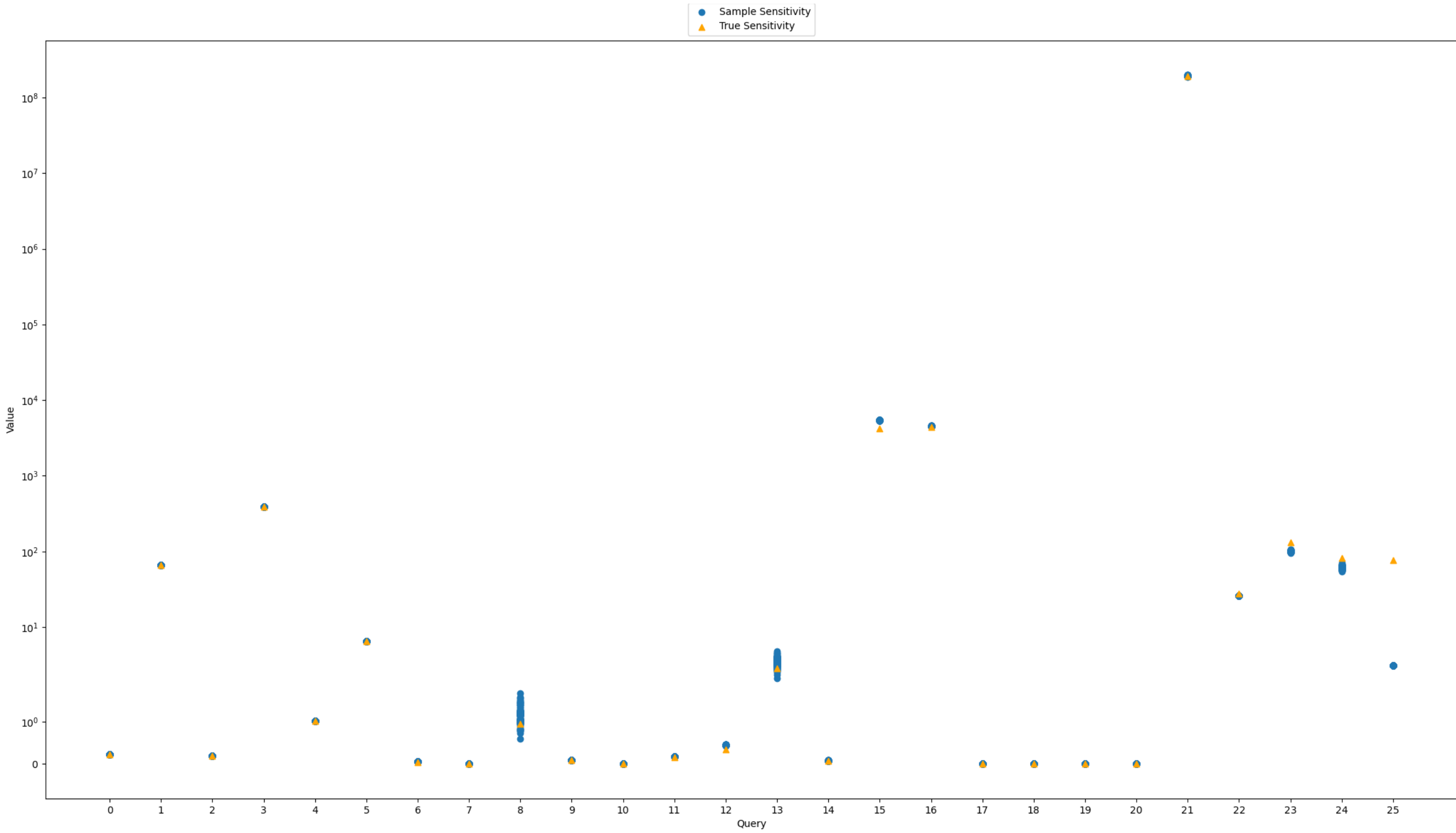


Figure 13: Generated sensitivity. The blue dots show a generated sensitivity value, the orange triangles the real sensitivity. The x-axis shows the different queries as defined in Appendix A.

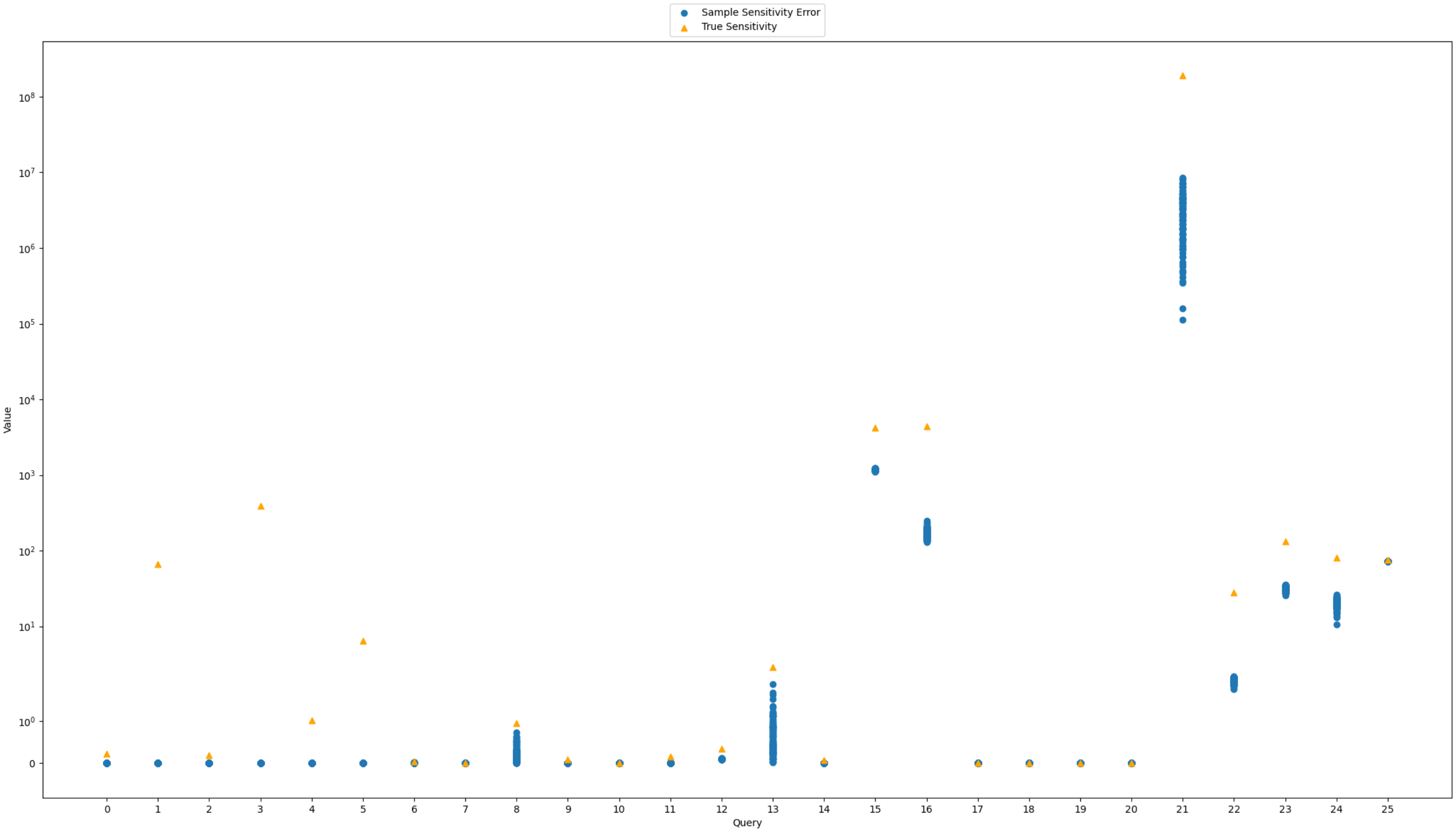


Figure 14: Generated sensitivity error. The blue dots show a generated sensitivity error, the orange triangles show the real sensitivity. Comparing the location of the blue dot with the orange triangle shows us whether the error is more or less than the real sensitivity. The x-axis shows the different queries as defined in Appendix A.

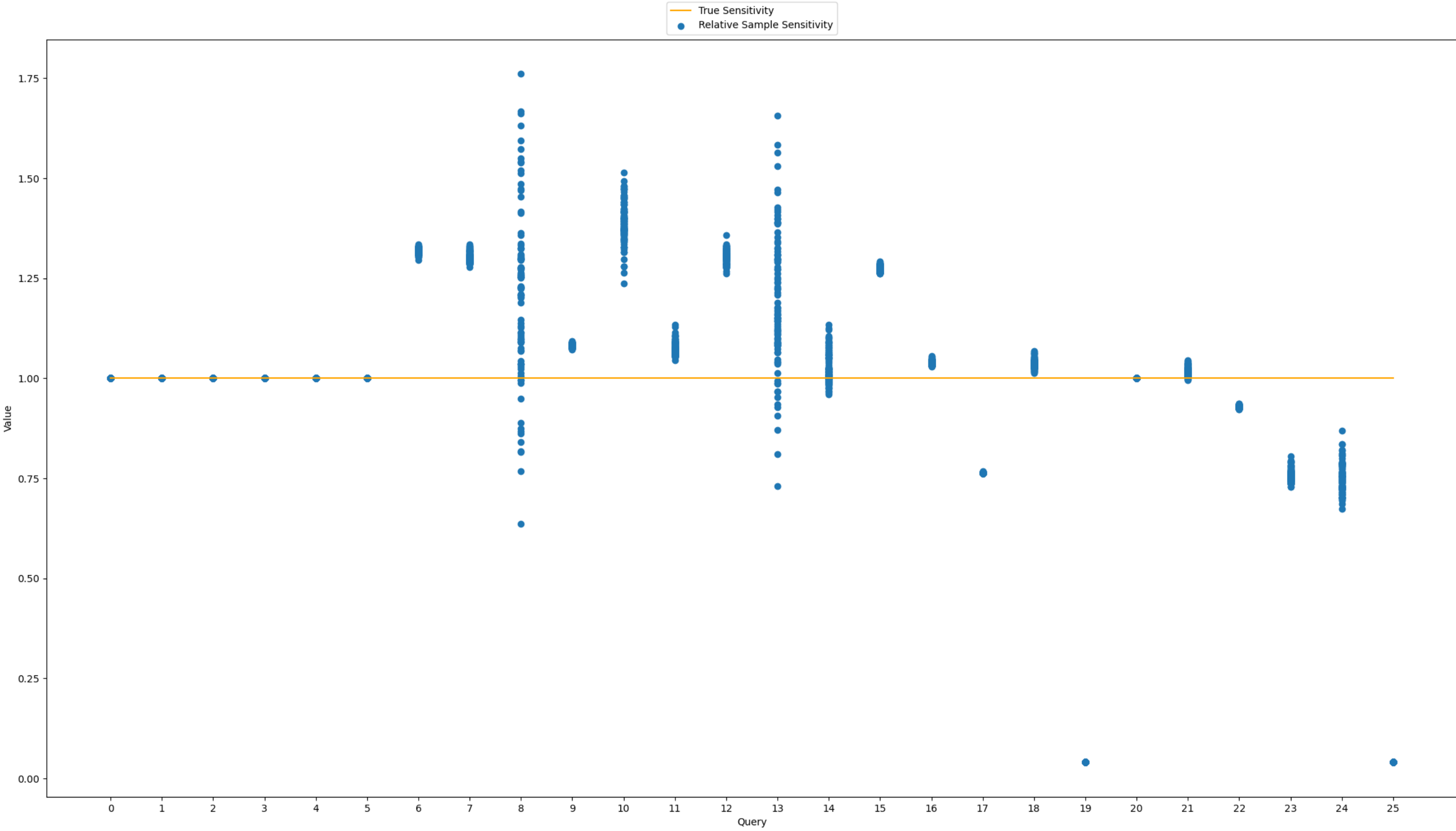


Figure 15: Relative generated sensitivity. This figure shows the generated sensitivity value divided by the real sensitivity. The orange line is at y=1 and represents the real sensitivity. The x-axis shows the different queries as defined in Appendix A.

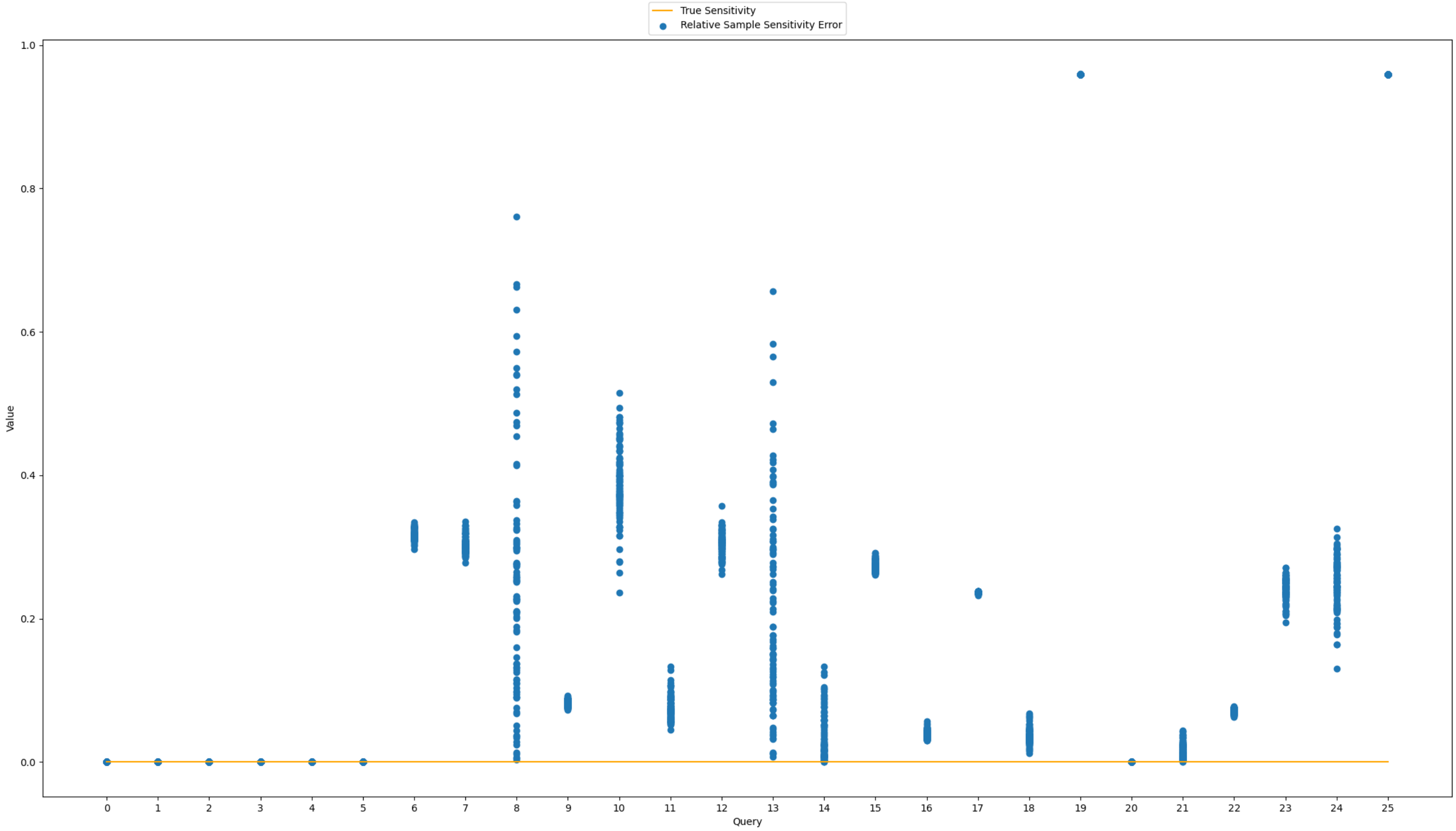


Figure 16: Relative generated sensitivity error. This figure shows the generated sensitivity error value divided by the real sensitivity. The orange line is at y=0 and represents the real sensitivity. The x-axis shows the different queries as defined in Appendix A.

Appendix C Results Tables

Query	Mean	Median	Min	Max
Kidney Disease				
0	53.0339	53.1269	−28.3770	168.9653
1	6898.3550	7675.0669	-2.6759×10^4	3.1618×10^4
2	0.6362	1.3891	−111.1317	61.0103
3	1.1239×10^4	1.0392×10^4	-1.4426×10^5	1.4851×10^5
4	353.0570	345.5466	24.4689	788.5061
5	1310.3838	1210.7484	−1212.4208	3950.3043
Wisconsin Breast Cancer				
6	14.4918	14.5600	−10.5881	51.1592
7	0.0353	0.0435	−0.2642	0.2183
8	52.1439	36.5612	−460.4581	708.1086
9	28.4219	26.7126	−5.4028	77.3469
10	0.1037	0.0873	−0.1209	0.4110
11	55.1543	56.6731	−48.4247	128.4143
12	62.0883	63.8328	−210.9819	440.8945
13	164.9987	198.4357	−1043.1245	1622.5042
14	10.8598	10.9874	−15.0645	44.9209
15	4.7149×10^5	4.9811×10^5	-3.5955×10^6	4.1003×10^6
Diabetes				
16	1.6517×10^8	1.6512×10^8	1.6115×10^8	1.6791×10^8
17	2.0222	2.0221	1.9600	2.0684
18	4.3936	4.3931	4.3478	4.4498
19	0.1729	0.1850	−0.8471	0.5376
20	438.4659	438.5793	433.9349	442.2971
21	5.5217×10^{12}	5.5260×10^{12}	5.4150×10^{12}	5.6372×10^{12}
22	3.7852×10^5	3.7848×10^5	3.6343×10^5	3.9251×10^5
23	4.3872×10^6	4.3884×10^6	4.2489×10^6	4.4989×10^6
24	1.6297×10^6	1.6289×10^6	1.6043×10^6	1.6816×10^6
25	2.1210×10^4	2.1091×10^4	-1.3850×10^4	1.1695×10^5

Table 10: Estimated sensitivity result statistics per query. The values in this table show statistics on the results to the queries when using our proposed method for estimating sensitivity

Query	Mean	Median	Min	Max
Kidney Disease				
0	-3.2735×10^{10}	2.0366×10^{8}	-3.7259×10^{12}	1.8980×10^{12}
1	-6.6537×10^{10}	3.6083×10^{9}	-1.7160×10^{12}	1.4184×10^{12}
2	8.9096×10^{10}	5.9307×10^{10}	-8.3587×10^{11}	1.6421×10^{12}
3	2.3227×10^{10}	3.7460×10^{10}	-1.3293×10^{12}	1.4841×10^{12}
4	9.3072×10^{10}	4.7872×10^{10}	-1.6924×10^{12}	1.5684×10^{12}
5	1.6366×10^{10}	-4.2746×10^{10}	-1.5553×10^{12}	2.6885×10^{12}
Wisconsin Breast Cancer				
6	-5.1778×10^{10}	-6.2523×10^{10}	-1.7045×10^{12}	2.5044×10^{12}
7	-6.5739×10^{10}	-7.2561×10^{10}	-1.7450×10^{12}	2.3987×10^{12}
8	-1.6640×10^{10}	-4.2296×10^{10}	-2.5583×10^{12}	2.2999×10^{12}
9	1.2438×10^{10}	1.0542×10^{10}	-1.3785×10^{12}	1.7074×10^{12}
10	6.9391×10^{10}	7.1559×10^{10}	-1.5799×10^{12}	1.9387×10^{12}
11	-3.1123×10^{10}	-1.5191×10^{9}	-2.2529×10^{12}	1.6051×10^{12}
12	7.8819×10^{10}	6.1212×10^{10}	-1.9535×10^{12}	3.0609×10^{12}
13	-3.7042×10^{10}	-5.2903×10^{10}	-2.5285×10^{12}	2.2496×10^{12}
14	-4.2920×10^{10}	-2.7584×10^{10}	-1.6515×10^{12}	1.4907×10^{12}
15	5.8678×10^{10}	4.4699×10^{9}	-1.3748×10^{12}	2.3239×10^{12}
Diabetes				
16	-6.0342×10^{10}	-8.3972×10^{10}	-1.6745×10^{12}	1.5141×10^{12}
17	-2.7234×10^{10}	-2.3675×10^{10}	-1.4768×10^{12}	2.8242×10^{12}
18	1.4081×10^{11}	4.4398×10^{10}	-1.2050×10^{12}	2.0493×10^{12}
19	1.4266×10^{10}	3.5177×10^{10}	-1.7887×10^{12}	1.3219×10^{12}
20	8.5688×10^{10}	5.2235×10^{9}	-1.5510×10^{12}	2.7315×10^{12}
21	5.5234×10^{12}	5.4621×10^{12}	3.7962×10^{12}	7.5977×10^{12}
22	-9.8252×10^{10}	-7.1174×10^{10}	-1.8803×10^{12}	1.1847×10^{12}
23	-1.0071×10^{10}	-6.3491×10^{9}	-1.4140×10^{12}	1.4210×10^{12}
24	-3.3413×10^{10}	-1.5890×10^{9}	-1.5745×10^{12}	1.4433×10^{12}
25	-2.3498×10^{10}	-3.6935×10^{10}	-1.4714×10^{12}	2.1980×10^{12}

Table 11: Global sensitivity statistics per query. The values in this table show statistics on the results of each query when using global sensitivity.

Query	Mean	Median	Min	Max
Kidney Disease				
0	1.0301	1.0319	−0.5512	3.2819
1	0.8187	0.9109	−3.1759	3.7525
2	0.2071	0.4521	−36.1703	19.8572
3	0.5137	0.4750	−6.5936	6.7876
4	0.9830	0.9621	0.0681	2.1955
5	1.0491	0.9693	−0.9706	3.1625
Wisconsin Breast Cancer				
6	1.0258	1.0306	−0.7495	3.6213
7	0.7217	0.8890	−5.4014	4.4624
8	1.2927	0.9064	−11.4153	17.5548
9	1.1069	1.0403	−0.2104	3.0123
10	1.2355	1.0404	−1.4405	4.8958
11	1.0059	1.0336	−0.8832	2.3421
12	1.0458	1.0752	−3.5537	7.4262
13	0.7157	0.8607	−4.5246	7.0378
14	0.9291	0.9400	−1.2888	3.8431
15	0.9410	0.9941	−7.1758	8.1834
Diabetes				
16	0.9998	0.9995	0.9755	1.0164
17	0.9991	0.9990	0.9684	1.0219
18	0.9995	0.9993	0.9890	1.0122
19	0.8737	0.9352	−4.2817	2.7175
20	0.9995	0.9998	0.9892	1.0083
21	0.9987	0.9995	0.9794	1.0196
22	1.0011	1.0009	0.9611	1.0380
23	1.0004	1.0006	0.9688	1.0258
24	0.9995	0.9990	0.9840	1.0314
25	1.0535	1.0476	−0.6879	5.8088

Table 12: Estimated sensitivity result statistics per query relative to true answer. This table shows the same data as Table 10, only divided by the true answer to each query as stated in Appendix A. A value of 1 represents a completely accurate result.

Query	Mean	Median	Min	Max
Kidney Disease				
0	-6.3583×10^{8}	3.9559×10^{6}	-7.2371×10^{10}	3.6866×10^{10}
1	-7.8969×10^{6}	4.2825×10^{5}	-2.0366×10^{8}	1.6834×10^{8}
2	2.8998×10^{10}	1.9303×10^{10}	-2.7205×10^{11}	5.3446×10^{11}
3	1.0616×10^{6}	1.7121×10^{6}	-6.0755×10^{7}	6.7830×10^{7}
4	2.5915×10^{8}	1.3330×10^{8}	-4.7124×10^{9}	4.3671×10^{9}
5	1.3102×10^{7}	-3.4222×10^{7}	-1.2451×10^{9}	2.1524×10^{9}
Wisconsin Breast Cancer				
6	-3.6651×10^{9}	-4.4257×10^{9}	-1.2066×10^{11}	1.7727×10^{11}
7	-1.3438×10^{12}	-1.4833×10^{12}	-3.5671×10^{13}	4.9034×10^{13}
8	-4.1253×10^{8}	-1.0486×10^{9}	-6.3424×10^{10}	5.7018×10^{10}
9	4.8441×10^{8}	4.1058×10^{8}	-5.3684×10^{10}	6.6497×10^{10}
10	8.2662×10^{11}	8.5244×10^{11}	-1.8821×10^{13}	2.3095×10^{13}
11	-5.6764×10^{8}	-2.7706×10^{7}	-4.1089×10^{10}	2.9275×10^{10}
12	1.3276×10^{9}	1.0310×10^{9}	-3.2904×10^{10}	5.1556×10^{10}
13	-1.6068×10^{8}	-2.2947×10^{8}	-1.0967×10^{10}	9.7577×10^{9}
14	-3.6719×10^{9}	-2.3599×10^{9}	-1.4129×10^{11}	1.2754×10^{11}
15	1.1711×10^{5}	8920.9939	-2.7439×10^{6}	4.6380×10^{6}
Diabetes				
16	-365.2622	-508.2999	-1.0136×10^{4}	9165.3226
17	-1.3456×10^{10}	-1.1697×10^{10}	-7.2962×10^{11}	1.3953×10^{12}
18	3.2031×10^{10}	1.0100×10^{10}	-2.7412×10^{11}	4.6618×10^{11}
19	7.2112×10^{10}	1.7781×10^{11}	-9.0414×10^{12}	6.6816×10^{12}
20	1.9533×10^{8}	1.1908×10^{7}	-3.5357×10^{9}	6.2267×10^{9}
21	0.9990	0.9879	0.6866	1.3742
22	-2.5984×10^{5}	-1.8823×10^{5}	-4.9728×10^{6}	3.1330×10^{6}
23	-2296.3710	-1447.6935	-3.2241×10^{5}	3.2401×10^{5}
24	-2.0493×10^{4}	-974.5597	-9.6567×10^{5}	8.8523×10^{5}
25	-1.1671×10^{6}	-1.8346×10^{6}	-7.3084×10^{7}	1.0917×10^{8}

Table 13: Global sensitivity result statistics per query relative to true answer. This table shows the same data as Table 11, only divided by the true answer to each query as stated in Appendix A. A value of 1 represents a completely accurate result.

Query	Mean	Median	Min	Max
Kidney Disease				
0	26.7087	20.7529	0.2236	117.4819
1	6350.7946	3935.4706	37.6311	3.5185×10^4
2	21.0610	14.7670	0.0671	114.2041
3	3.8793×10^4	3.0546×10^4	66.6267	1.6614×10^5
4	103.3523	89.7511	0.1785	429.3611
5	676.1217	512.0601	6.3121	2701.2043
Wisconsin Breast Cancer				
6	7.4007	4.2478	0.0364	37.0319
7	0.0513	0.0330	3.3736×10^{-4}	0.3131
8	111.9850	70.1569	1.1830	667.7715
9	10.5778	8.1888	0.0259	51.6697
10	0.0603	0.0380	1.0312×10^{-4}	0.3270
11	18.6675	14.5125	0.0573	103.2537
12	44.3178	22.9034	0.0668	381.5245
13	363.1467	250.5404	0.7313	1391.9613
14	8.2501	6.6645	0.0156	33.2323
15	5.6481×10^5	3.5446×10^5	2283.4060	4.0965×10^6
Diabetes				
16	5.0888×10^5	3.9812×10^5	2347.5416	4.0478×10^6
17	0.0107	0.0074	4.0604×10^{-4}	0.0640
18	0.0121	0.0098	1.6698×10^{-4}	0.0538
19	0.1065	0.0777	1.2453×10^{-4}	1.0449
20	1.0875	0.8497	0.0052	4.7400
21	2.1331×10^{10}	1.6525×10^{10}	1.7894×10^8	1.1399×10^{11}
22	2796.7119	1980.9561	57.1948	1.4691×10^4
23	1.5564×10^4	9642.9233	311.7239	1.3678×10^5
24	7584.5482	4416.4285	28.8194	5.1150×10^4
25	9894.9066	6350.9685	48.4201	9.6816×10^4

Table 14: Estimated sensitivity result error statistics per query. The values in this table show statistics on the error between the true answer as shown in Appendix A and the result when using our estimated sensitivity mechanism.

Query	Mean	Median	Min	Max
Kidney Disease				
0	4.1261×10^{11}	3.1134×10^{11}	4.0807×10^{9}	3.7259×10^{12}
1	4.2611×10^{11}	3.1064×10^{11}	1.0842×10^{9}	1.7160×10^{12}
2	3.4497×10^{11}	2.2956×10^{11}	8.2650×10^{9}	1.6421×10^{12}
3	3.7349×10^{11}	2.6530×10^{11}	1.8168×10^{9}	1.4841×10^{12}
4	4.3094×10^{11}	3.3060×10^{11}	6.7813×10^{9}	1.6924×10^{12}
5	4.1407×10^{11}	2.6382×10^{11}	5.5151×10^{9}	2.6885×10^{12}
Wisconsin Breast Cancer				
6	4.4109×10^{11}	3.5407×10^{11}	1.7730×10^{9}	2.5044×10^{12}
7	4.2187×10^{11}	2.9705×10^{11}	5.5681×10^{8}	2.3987×10^{12}
8	3.9777×10^{11}	2.6435×10^{11}	1.4573×10^{9}	2.5583×10^{12}
9	4.0740×10^{11}	3.1515×10^{11}	1.6239×10^{9}	1.7074×10^{12}
10	3.9534×10^{11}	3.1698×10^{11}	2.2212×10^{9}	1.9387×10^{12}
11	5.1015×10^{11}	3.7313×10^{11}	1.1987×10^{9}	2.2529×10^{12}
12	4.1097×10^{11}	2.7732×10^{11}	1.6983×10^{9}	3.0609×10^{12}
13	4.6236×10^{11}	2.9790×10^{11}	1.0564×10^{9}	2.5285×10^{12}
14	4.0149×10^{11}	2.9412×10^{11}	4.4963×10^{9}	1.6515×10^{12}
15	4.2522×10^{11}	3.3090×10^{11}	3.6017×10^{8}	2.3239×10^{12}
Diabetes				
16	3.8426×10^{11}	2.7234×10^{11}	2.5980×10^{9}	1.6747×10^{12}
17	3.9692×10^{11}	2.4614×10^{11}	5.5998×10^{8}	2.8242×10^{12}
18	4.1592×10^{11}	2.8542×10^{11}	1.1749×10^{9}	2.0493×10^{12}
19	3.8474×10^{11}	2.4497×10^{11}	9.3524×10^{8}	1.7887×10^{12}
20	4.1981×10^{11}	2.6449×10^{11}	4.9489×10^{9}	2.7315×10^{12}
21	4.6876×10^{11}	2.7135×10^{11}	3.8043×10^{9}	2.0688×10^{12}
22	3.9484×10^{11}	2.2616×10^{11}	3.2176×10^{9}	1.8803×10^{12}
23	3.7883×10^{11}	2.9001×10^{11}	5.5968×10^{9}	1.4210×10^{12}
24	3.5756×10^{11}	2.4848×10^{11}	8.4646×10^{8}	1.5745×10^{12}
25	4.1597×10^{11}	2.6188×10^{11}	2.5500×10^{8}	2.1980×10^{12}

Table 15: Global sensitivity result error statistics per query. The values in this table show statistics on the error between the true answer as shown in Appendix A and the result when using global sensitivity.

Query	Mean	Median	Min	Max
Kidney Disease				
0	0.5188	0.4031	0.0043	2.2819
1	0.7537	0.4671	0.0045	4.1759
2	6.8548	4.8063	0.0219	37.1703
3	1.7730	1.3961	0.0030	7.5936
4	0.2878	0.2499	4.9703×10^{-4}	1.1955
5	0.5413	0.4099	0.0051	2.1625
Wisconsin Breast Cancer				
6	0.5239	0.3007	0.0026	2.6213
7	1.0489	0.6739	0.0069	6.4014
8	2.7762	1.7393	0.0293	16.5548
9	0.4120	0.3189	0.0010	2.0123
10	0.7188	0.4522	0.0012	3.8958
11	0.3405	0.2647	0.0010	1.8832
12	0.7465	0.3858	0.0011	6.4262
13	1.5752	1.0867	0.0032	6.0378
14	0.7058	0.5702	0.0013	2.8431
15	1.1272	0.7074	0.0046	8.1758
Diabetes				
16	0.0031	0.0024	1.4210×10^{-5}	0.0245
17	0.0053	0.0037	2.0061×10^{-4}	0.0316
18	0.0027	0.0022	3.7984×10^{-5}	0.0122
19	0.5385	0.3925	6.2944×10^{-4}	5.2817
20	0.0025	0.0019	1.1918×10^{-5}	0.0108
21	0.0039	0.0030	3.2365×10^{-5}	0.0206
22	0.0074	0.0052	1.5126×10^{-4}	0.0389
23	0.0035	0.0022	7.1078×10^{-5}	0.0312
24	0.0047	0.0027	1.7675×10^{-5}	0.0314
25	0.4915	0.3155	0.0024	4.8088

Table 16: Estimated sensitivity result error statistics per query relative to true answer. This table shows the same data as Table 14, only divided by the true answer to each query as stated in Appendix A. A value of 0 represents a completely accurate result.

Query	Mean	Median	Min	Max
Kidney Disease				
0	8.0144×10^{9}	6.0474×10^{9}	7.9262×10^{7}	7.2371×10^{10}
1	5.0572×10^{7}	3.6867×10^{7}	1.2867×10^{5}	2.0366×10^{8}
2	1.1228×10^{11}	7.4714×10^{10}	2.6900×10^{9}	5.3446×10^{11}
3	1.7071×10^{7}	1.2126×10^{7}	8.3038×10^{4}	6.7830×10^{7}
4	1.1999×10^{9}	9.2051×10^{8}	1.8882×10^{7}	4.7124×10^{9}
5	3.3150×10^{8}	2.1120×10^{8}	4.4153×10^{6}	2.1524×10^{9}
Wisconsin Breast Cancer				
6	3.1223×10^{10}	2.5063×10^{10}	1.2550×10^{8}	1.7727×10^{11}
7	8.6239×10^{12}	6.0723×10^{12}	1.1382×10^{10}	4.9034×10^{13}
8	9.8610×10^{9}	6.5534×10^{9}	3.6129×10^{7}	6.3424×10^{10}
9	1.5866×10^{10}	1.2274×10^{10}	6.3242×10^{7}	6.6497×10^{10}
10	4.7095×10^{12}	3.7760×10^{12}	2.6460×10^{10}	2.3095×10^{13}
11	9.3043×10^{9}	6.8054×10^{9}	2.1863×10^{7}	4.1089×10^{10}
12	6.9222×10^{9}	4.6711×10^{9}	2.8606×10^{7}	5.1556×10^{10}
13	2.0055×10^{9}	1.2922×10^{9}	4.5822×10^{6}	1.0967×10^{10}
14	3.4349×10^{10}	2.5163×10^{10}	3.8467×10^{8}	1.4129×10^{11}
15	8.4866×10^{5}	6.6041×10^{5}	718.8294	4.6380×10^{6}
Diabetes				
16	2326.0144	1648.5036	15.7262	1.0137×10^{4}
17	1.9611×10^{11}	1.2161×10^{11}	2.7667×10^{8}	1.3953×10^{12}
18	9.4614×10^{10}	6.4928×10^{10}	2.6727×10^{8}	4.6618×10^{11}
19	1.9448×10^{12}	1.2383×10^{12}	4.7274×10^{9}	9.0414×10^{12}
20	9.5700×10^{8}	6.0294×10^{8}	1.1281×10^{7}	6.2267×10^{9}
21	0.0848	0.0491	6.8806×10^{-4}	0.3742
22	1.0442×10^{6}	5.9811×10^{5}	8509.2346	4.9728×10^{6}
23	8.6380×10^{4}	6.6127×10^{4}	1276.1478	3.2401×10^{5}
24	2.1930×10^{5}	1.5240×10^{5}	519.1511	9.6568×10^{5}
25	2.0661×10^{7}	1.3007×10^{7}	1.2666×10^{4}	1.0917×10^{8}

Table 17: Global sensitivity result error statistics per query relative to true answer. This table shows the same data as Table 15, only divided by the true answer to each query as stated in Appendix A. A value of 0 represents a completely accurate result.

Query	Mean	Median	Min	Max
Kidney Disease				
0	0.2340	0.2343	0.2290	0.2390
1	68.6517	68.6197	67.1859	70.0351
2	0.1976	0.1977	0.1934	0.2018
3	405.9473	406.1261	397.2029	413.8628
4	1.0648	1.0680	1.0406	1.0863
5	6.7398	6.7297	6.6008	6.8891
Wisconsin Breast Cancer				
6	0.0688	0.0524	0.0505	0.1435
7	4.6441×10^{-4}	3.7443×10^{-4}	3.6131×10^{-4}	9.4625×10^{-4}
8	1.2234	1.0023	0.9705	4.1461
9	0.0982	0.0914	0.0886	0.1313
10	4.8831×10^{-4}	3.8726×10^{-4}	3.7265×10^{-4}	0.0013
11	0.1833	0.1719	0.1661	0.2781
12	0.4609	0.3648	0.3525	0.9648
13	3.5687	3.0206	2.9236	9.8389
14	0.0860	0.0827	0.0802	0.1220
15	5503.3406	4487.1131	4334.9970	1.1095×10^{4}
Diabetes				
16	4716.7367	4553.3166	4429.5550	5621.5278
17	9.0565×10^{-5}	8.1200×10^{-5}	7.8800×10^{-5}	1.4393×10^{-4}
18	1.4665×10^{-4}	1.4384×10^{-4}	1.3977×10^{-4}	1.7627×10^{-4}
19	0.0010	7.4813×10^{-4}	5.2069×10^{-4}	0.0029
20	0.0102	0.0102	0.0100	0.0104
21	1.9936×10^{8}	1.9777×10^{8}	1.9247×10^{8}	2.2676×10^{8}
22	29.7807	29.1837	28.4073	36.4772
23	154.6091	136.2794	132.8698	261.9760
24	93.5270	83.9532	74.1707	182.7525
25	120.6916	76.1390	60.7001	408.8314

Table 18: Estimated sensitivity statistics. This table shows statistics on the estimated sensitivity values as generated by our mechanism.

Query	Mean	Median	Min	Max
Kidney Disease				
0	0.0376	0.0386	0.0150	0.0597
1	0.0376	0.0371	0.0154	0.0585
2	0.0376	0.0382	0.0153	0.0596
3	0.0382	0.0387	0.0159	0.0585
4	0.0389	0.0420	0.0152	0.0598
5	0.0369	0.0353	0.0155	0.0599
Wisconsin Breast Cancer				
6	0.3908	0.0585	0.0214	1.8994
7	0.3111	0.0570	0.0200	1.6713
8	0.2817	0.0500	0.0167	3.3434
9	0.1259	0.0479	0.0164	0.5049
10	0.3367	0.0601	0.0201	2.5558
11	0.1220	0.0520	0.0165	0.7018
12	0.3344	0.0562	0.0206	1.7933
13	0.2421	0.0514	0.0176	2.4246
14	0.0895	0.0478	0.0154	0.5449
15	0.2937	0.0548	0.0190	1.6081
Diabetes				
16	0.0814	0.0439	0.0156	0.2888
17	0.1520	0.0326	0.0019	0.8308
18	0.0660	0.0456	0.0160	0.2813
19	0.3625	0.0018	6.4139×10^{-4}	2.9501
20	0.0350	0.0342	0.0153	0.0599
21	0.0520	0.0436	0.0157	0.1966
22	0.0636	0.0423	0.0145	0.3028
23	0.1713	0.0324	0.0066	0.9847
24	0.1563	0.0371	0.0111	1.2562
25	0.5921	0.0018	6.2931×10^{-4}	4.3794

Table 19: Estimated sensitivity error statistics. This table shows statistics on the error of the estimated sensitivity values produced by our mechanism. This error is taken relative to the true sensitivity value for each query as seen in Appendix A.

Query	Mean	Median	Min	Max
Kidney Disease				
0	1.0376	1.0386	1.0150	1.0597
1	1.0376	1.0371	1.0154	1.0585
2	1.0376	1.0382	1.0153	1.0596
3	1.0382	1.0387	1.0159	1.0585
4	1.0389	1.0420	1.0152	1.0598
5	1.0369	1.0353	1.0155	1.0599
Wisconsin Breast Cancer				
6	1.3908	1.0585	1.0214	2.8994
7	1.3111	1.0570	1.0200	2.6713
8	1.2817	1.0500	1.0167	4.3434
9	1.1259	1.0479	1.0164	1.5049
10	1.3367	1.0601	1.0201	3.5558
11	1.1220	1.0520	1.0165	1.7018
12	1.3344	1.0562	1.0206	2.7933
13	1.2421	1.0514	1.0176	3.4246
14	1.0895	1.0478	1.0154	1.5449
15	1.2937	1.0548	1.0190	2.6081
Diabetes				
16	1.0814	1.0439	1.0156	1.2888
17	1.1522	1.0331	1.0025	1.8311
18	1.0660	1.0455	1.0160	1.2813
19	1.3565	1.0018	0.6972	3.9501
20	1.0350	1.0342	1.0153	1.0599
21	1.0520	1.0436	1.0157	1.1966
22	1.0636	1.0423	1.0145	1.3028
23	1.1713	1.0324	1.0066	1.9847
24	1.1547	1.0365	0.9157	2.2562
25	1.5880	1.0018	0.7987	5.3794

Table 20: Estimated sensitivity statistics relative to true sensitivity. This table shows the same data as Table 18, only divided by the true sensitivity of each query as stated in Appendix A. A value of 1 represents a completely accurate result.

Query	Mean	Median	Min	Max
		Kidney Disease		
0	0.0376	0.0386	0.0150	0.0597
1	0.0376	0.0371	0.0154	0.0585
2	0.0376	0.0382	0.0153	0.0596
3	0.0382	0.0387	0.0159	0.0585
4	0.0389	0.0420	0.0152	0.0598
5	0.0369	0.0353	0.0155	0.0599
		Wisconsin Breast Cancer		
6	0.3908	0.0585	0.0214	1.8994
7	0.3111	0.0570	0.0200	1.6713
8	0.2817	0.0500	0.0167	3.3434
9	0.1259	0.0479	0.0164	0.5049
10	0.3367	0.0601	0.0201	2.5558
11	0.1220	0.0520	0.0165	0.7018
12	0.3344	0.0562	0.0206	1.7933
13	0.2421	0.0514	0.0176	2.4246
14	0.0895	0.0478	0.0154	0.5449
15	0.2937	0.0548	0.0190	1.6081
		Diabetes		
16	0.0814	0.0439	0.0156	0.2888
17	0.1520	0.0326	0.0019	0.8308
18	0.0660	0.0456	0.0160	0.2813
19	0.3625	0.0018	6.4139×10^{-4}	2.9501
20	0.0350	0.0342	0.0153	0.0599
21	0.0520	0.0436	0.0157	0.1966
22	0.0636	0.0423	0.0145	0.3028
23	0.1713	0.0324	0.0066	0.9847
24	0.1563	0.0371	0.0111	1.2562
25	0.5921	0.0018	6.2931×10^{-4}	4.3794

Table 21: Estimated sensitivity error statistics relative to true sensitivity. This table shows the same data as Table 19, only divided by the true sensitivity of each query as stated in Appendix A. A value of 0 represents a completely accurate result.

Query	Mean	Median	Min	Max
Kidney Disease				
0	0.2256	0.2256	0.2256	0.2256
1	66.1654	66.1654	66.1654	66.1654
2	0.1905	0.1905	0.1905	0.1905
3	391.0000	391.0000	391.0000	391.0000
4	1.0250	1.0250	1.0250	1.0250
5	6.5000	6.5000	6.5000	6.5000
Wisconsin Breast Cancer				
6	0.0652	0.0652	0.0642	0.0660
7	4.6171×10^{-4}	4.6117×10^{-4}	4.5266×10^{-4}	4.7291×10^{-4}
8	1.1616	1.1736	0.6081	1.6809
9	0.0944	0.0944	0.0936	0.0953
10	5.0801×10^{-4}	5.0661×10^{-4}	4.5167×10^{-4}	5.5353×10^{-4}
11	0.1758	0.1754	0.1708	0.1852
12	0.4507	0.4512	0.4359	0.4690
13	3.4308	3.3474	2.0992	4.7597
14	0.0813	0.0805	0.0758	0.0895
15	5417.9027	5414.7704	5366.2138	5497.2020
Diabetes				
16	4532.8456	4528.9512	4491.9021	4609.4718
17	6.0025×10^{-5}	6.0000×10^{-5}	5.9900×10^{-5}	6.0300×10^{-5}
18	1.4260×10^{-4}	1.4250×10^{-4}	1.3934×10^{-4}	1.4693×10^{-4}
19	3.0488×10^{-5}	3.0500×10^{-5}	3.0100×10^{-5}	3.0900×10^{-5}
20	0.0098	0.0098	0.0098	0.0098
21	1.9257×10^{8}	1.9227×10^{8}	1.8875×10^{8}	1.9794×10^{8}
22	25.9952	25.9940	25.8100	26.2397
23	100.1035	99.8330	96.1602	106.3065
24	60.8049	60.6423	54.6275	70.4203
25	3.1026	3.1017	3.0764	3.1398

Table 22: Generated sensitivity statistics. This table shows statistics on the generated sensitivity for each query.

Query	Mean	Median	Min	Max
Kidney Disease				
0	0.0000	0.0000	0.0000	0.0000
1	0.0000	0.0000	0.0000	0.0000
2	0.0000	0.0000	0.0000	0.0000
3	0.0000	0.0000	0.0000	0.0000
4	0.0000	0.0000	0.0000	0.0000
5	0.0000	0.0000	0.0000	0.0000
Wisconsin Breast Cancer				
6	0.0157	0.0157	0.0147	0.0165
7	1.0748×10^{-4}	1.0694×10^{-4}	9.8437×10^{-5}	1.1868×10^{-4}
8	0.2395	0.2303	0.0036	0.7263
9	0.0072	0.0072	0.0063	0.0081
10	1.4270×10^{-4}	1.4130×10^{-4}	8.6349×10^{-5}	1.8821×10^{-4}
11	0.0124	0.0120	0.0074	0.0218
12	0.1053	0.1058	0.0905	0.1236
13	0.6131	0.4995	0.0200	1.8867
14	0.0030	0.0019	1.7450×10^{-5}	0.0105
15	1163.9027	1160.7704	1112.2138	1243.2020
Diabetes				
16	171.1572	167.2628	130.2137	247.7834
17	1.8575×10^{-5}	1.8600×10^{-5}	1.8300×10^{-5}	1.8700×10^{-5}
18	5.0288×10^{-6}	4.9250×10^{-6}	1.7640×10^{-6}	9.3560×10^{-6}
19	7.1633×10^{-4}	7.1632×10^{-4}	7.1592×10^{-4}	7.1672×10^{-4}
20	0.0000	0.0000	0.0000	0.0000
21	3.1053×10^{6}	2.7687×10^{6}	1.1262×10^{5}	8.4417×10^{6}
22	2.0048	2.0060	1.7603	2.1900
23	31.8965	32.1670	25.6935	35.8398
24	20.1951	20.3577	10.5797	26.3725
25	72.8974	72.8983	72.8602	72.9236

Table 23: Generated sensitivity error statistics. This table shows statistics on the error of the generated sensitivity per query. The error is relative to the true sensitivity as seen in Appendix A

Query	Mean	Median	Min	Max
Kidney Disease				
0	1.0000	1.0000	1.0000	1.0000
1	1.0000	1.0000	1.0000	1.0000
2	1.0000	1.0000	1.0000	1.0000
3	1.0000	1.0000	1.0000	1.0000
4	1.0000	1.0000	1.0000	1.0000
5	1.0000	1.0000	1.0000	1.0000
Wisconsin Breast Cancer				
6	1.3173	1.3165	1.2964	1.3342
7	1.3034	1.3019	1.2779	1.3351
8	1.2169	1.2294	0.6370	1.7609
9	1.0822	1.0820	1.0727	1.0926
10	1.3906	1.3868	1.2364	1.5152
11	1.0762	1.0732	1.0454	1.1335
12	1.3048	1.3063	1.2621	1.3578
13	1.1942	1.1651	0.7307	1.6567
14	1.0302	1.0192	0.9599	1.1335
15	1.2736	1.2729	1.2615	1.2922
Diabetes				
16	1.0392	1.0383	1.0299	1.0568
17	0.7637	0.7634	0.7621	0.7672
18	1.0366	1.0358	1.0128	1.0680
19	0.0408	0.0408	0.0403	0.0414
20	1.0000	1.0000	1.0000	1.0000
21	1.0162	1.0146	0.9960	1.0445
22	0.9284	0.9284	0.9218	0.9371
23	0.7584	0.7563	0.7285	0.8054
24	0.7507	0.7487	0.6744	0.8694
25	0.0408	0.0408	0.0405	0.0413

Table 24: Generated sensitivity statistics relative to true sensitivity. This table shows the same data as Table 22, only divided by the true sensitivity of each query as stated in Appendix A. A value of 1 represents a completely accurate result.

Query	Mean	Median	Min	Max
Kidney Disease				
0	0.0000	0.0000	0.0000	0.0000
1	0.0000	0.0000	0.0000	0.0000
2	0.0000	0.0000	0.0000	0.0000
3	0.0000	0.0000	0.0000	0.0000
4	0.0000	0.0000	0.0000	0.0000
5	0.0000	0.0000	0.0000	0.0000
Wisconsin Breast Cancer				
6	0.3173	0.3165	0.2964	0.3342
7	0.3034	0.3019	0.2779	0.3351
8	0.2509	0.2412	0.0038	0.7609
9	0.0822	0.0820	0.0727	0.0926
10	0.3906	0.3868	0.2364	0.5152
11	0.0762	0.0732	0.0454	0.1335
12	0.3048	0.3063	0.2621	0.3578
13	0.2134	0.1739	0.0069	0.6567
14	0.0376	0.0241	2.2103×10^{-4}	0.1335
15	0.2736	0.2729	0.2615	0.2922
Diabetes				
16	0.0392	0.0383	0.0299	0.0568
17	0.2363	0.2366	0.2328	0.2379
18	0.0366	0.0358	0.0128	0.0680
19	0.9592	0.9592	0.9586	0.9597
20	0.0000	0.0000	0.0000	0.0000
21	0.0164	0.0146	5.9432×10^{-4}	0.0445
22	0.0716	0.0716	0.0629	0.0782
23	0.2416	0.2437	0.1946	0.2715
24	0.2493	0.2513	0.1306	0.3256
25	0.9592	0.9592	0.9587	0.9595

Table 25: Generated sensitivity error statistics relative to true sensitivity. This table shows the same data as Table 23, only divided by the true sensitivity of each query as stated in Appendix A. A value of 0 represents a completely accurate result.

Query	Mean	Median	Min	Max
Kidney Disease				
0	0.0391	0.0371	0.0042	0.0732
1	0.0183	0.0172	1.0064×10^{-4}	0.0551
2	0.1165	0.1069	3.7190×10^{-4}	0.4023
3	0.1329	0.1307	0.0255	0.2770
4	0.0142	0.0119	5.5688×10^{-5}	0.0344
5	0.1070	0.1090	0.0072	0.1972
Wisconsin Breast Cancer				
6	0.3435	0.3440	0.3141	0.3678
7	0.5321	0.5358	0.4112	0.6147
8	0.3938	0.3963	0.3365	0.4642
9	0.3841	0.3844	0.3501	0.4222
10	0.2379	0.2376	0.2159	0.2646
11	0.0533	0.0533	0.0407	0.0717
12	0.1548	0.1534	0.1010	0.2337
13	0.2894	0.2894	0.2492	0.3372
14	0.1161	0.1171	0.0906	0.1394
15	0.7411	0.7387	0.5943	0.8507
Diabetes				
16	1.9092×10^{-6}	1.2230×10^{-6}	4.3393×10^{-7}	7.2236×10^{-6}
17	4.7643×10^{-6}	1.3213×10^{-6}	4.4651×10^{-7}	3.1562×10^{-5}
18	2.4747×10^{-6}	1.3683×10^{-6}	5.0274×10^{-7}	9.7818×10^{-6}
19	0.0019	6.8457×10^{-6}	2.2918×10^{-6}	0.0168
20	8.4847×10^{-7}	8.4516×10^{-7}	3.7720×10^{-7}	1.3417×10^{-6}
21	2.1823×10^{-6}	1.6978×10^{-6}	5.4054×10^{-7}	6.6182×10^{-6}
22	6.0752×10^{-6}	3.6098×10^{-6}	1.0643×10^{-6}	2.4479×10^{-5}
23	4.0581×10^{-6}	1.0317×10^{-6}	3.4464×10^{-7}	3.1358×10^{-5}
24	1.0174×10^{-5}	1.8598×10^{-6}	6.1488×10^{-7}	5.4307×10^{-5}
25	0.0028	7.8852×10^{-6}	2.3413×10^{-6}	0.0171
Overall	0.1415	0.0439	3.4464×10^{-7}	0.8507

Table 26: Error when using the anomiGAN method relative to true answer. This error is relative to the true answer as stated in Appendix A

References

[1] European Union. *General Data Protection Regulation.* 2016 (accessed June 25, 2020). URL: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679.

[2] State of California. *California Consumer Privacy Act.* 2018 (accessed June 25, 2020). URL: https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180SB1121.

[3] Arthur R Miller. "Computers, Data Banks and Individual Privacy: An Overview". In: *Colum. Hum. Rts. L. Rev.* 4 (1972), p. 1.

[4] Nabil R Adam and John C Worthmann. "Security-control methods for statistical databases: a comparative study". In: *ACM Computing Surveys (CSUR)* 21.4 (1989), pp. 515–556.

[5] Cynthia Dwork, Aaron Roth, et al. "The algorithmic foundations of differential privacy". In: *Foundations and Trends® in Theoretical Computer Science* 9.3–4 (2014), pp. 211–407.

[6] Cynthia Dwork et al. "Calibrating noise to sensitivity in private data analysis". In: *Theory of cryptography conference.* Springer. 2006, pp. 265–284.

[7] Noah Johnson, Joseph P Near, and Dawn Song. "Towards practical differential privacy for SQL queries". In: *Proceedings of the VLDB Endowment* 11.5 (2018), pp. 526–539.

[8] Lei Xu et al. "Modeling tabular data using conditional gan". In: *Advances in Neural Information Processing Systems.* 2019, pp. 7333–7343.

[9] Kevin Schawinski et al. "Generative adversarial networks recover features in astrophysical images of galaxies beyond the deconvolution limit". In: *Monthly Notices of the Royal Astronomical Society: Letters* 467.1 (2017), pp. L110–L114.

[10] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository.* 2017. URL: http://archive.ics.uci.edu/ml.

[11] Ho Bae, Dahuin Jung, and Sungroh Yoon. "AnomiGAN: Generative adversarial networks for anonymizing private medical data". In: *arXiv preprint arXiv:1901.11313* (2019).

[12] Naoise Holohan et al. "Diffprivlib: The IBM Differential Privacy Library". In: *arXiv preprint arXiv:1907.02444* (2019).

[13] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. "Smooth sensitivity and sampling in private data analysis". In: *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing.* ACM. 2007, pp. 75–84.

[14] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. "Rappor: Randomized aggregatable privacy-preserving ordinal response". In: *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security.* 2014, pp. 1054–1067.

[15] Apple Inc. *Differential Privacy.* 2016 (accessed June 25, 2020). URL: https://www.apple.com/privacy/docs/Differential_Privacy_Overview.pdf.

[16] Jun Tang et al. "Privacy loss in apple's implementation of differential privacy on macos 10.12". In: *arXiv preprint arXiv:1709.02753* (2017).

[17] Chaya Nayak. *New privacy-protected Facebook data for independent research on social media's impact on democracy.* 2020 (accessed June 25, 2020). URL: https://research.fb.com/blog/2020/02/new-privacy-protected-facebook-data-for-independent-research-on-social-medias-impact-on-democracy/.

[18] Daniel Kifer et al. “Guidelines for Implementing and Auditing Differentially Private Systems”. In: *arXiv preprint arXiv:2002.04049* (2020).

[19] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. “Collecting telemetry data privately”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 3571–3580.

[20] Ryan Rogers et al. “LinkedIn’s Audience Engagements API: A Privacy Preserving Data Analytics System at Scale”. In: *arXiv preprint arXiv:2002.05839* (2020).

[21] Sivakanth Gopi et al. “Differentially private set union”. In: *arXiv preprint arXiv:2002.09745* (2020).

[22] Michael Hawes. *Title 13, Differential Privacy, and the 2020 Decennial Census*. 2019 (accessed June 25, 2020). URL: `https://www2.census.gov/about/policies/2019-11-paper-differential-privacy.pdf`.

[23] Han Zhang et al. “Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 5907–5915.

[24] Noseong Park et al. “Data synthesis based on generative adversarial networks”. In: *Proceedings of the VLDB Endowment* 11.10 (2018), pp. 1071–1083.

[25] Piper Jackson and Marco Lussetti. “Extending a Generative Adversarial Network to Produce Medical Records with Demographic Characteristics and Health System Use”. In: *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE. 2019, pp. 0515–0518.

[26] Edward Choi et al. “Generating multi-label discrete patient records using generative adversarial networks”. In: *arXiv preprint arXiv:1703.06490* (2017).

[27] Ian Goodfellow et al. “Generative adversarial nets”. In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[28] Olof Mogren. “C-RNN-GAN: Continuous recurrent neural networks with adversarial training”. In: *arXiv preprint arXiv:1611.09904* (2016).

[29] Martin Arjovsky, Soumith Chintala, and Léon Bottou. “Wasserstein gan”. In: *arXiv preprint arXiv:1701.07875* (2017).

[30] Ishaan Gulrajani et al. “Improved training of wasserstein gans”. In: *Advances in neural information processing systems*. 2017, pp. 5767–5777.

[31] Tero Karras et al. “Progressive growing of gans for improved quality, stability, and variation”. In: *arXiv preprint arXiv:1710.10196* (2017).