FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Differential Privacy

Implementation and examination of four differential privacy models

Friis, Jonas
jof1007@gmail.com
xdr476@alumni.ku.dk

June 14, 2021

1 Abstract

In this thesis, we present and discuss central and local differential privacy for range counting and the amount of noise added to the output; We do this by implementing different central and local differential private data structures. We implement a flat solution in both central and local DP. We furthermore examine two variations of hierarchical histograms. The data structures are implemented in Python. We then analyze and benchmark the data structures against each other by measuring the error over different range queries.

Contents

2 Introduction

Current companies collect and use their users/customers' data to improve and develop their services. It makes sense for the company to want their users/customers' data. In that way, the company can develop products that the users/customers want to utilize instead of guessing (speculating) what they may want. However, the data that the companies collect about their users could potentially be personal information that the users do not want to share. Therefore the users will demand privacy guarantees.

There have been several naive attempts to preserve the privacy of the users. One simple naive approach would be to simply strip the data from personally identifying information, such as names, addresses, social security numbers, etc. This approach usually fails; the leftover data can be used in connection to other data/information from different datasets to identify people uniquely. This linkage attack was first done back in 2000 by Latanya Sweeney, who identified the former Governor of Massachusetts William Weld's health records using only his date of birth, gender, and zip code (Sweeney 2000). Another example of this occurred in 2007 when Netflix released a dataset of 100,480,507 ratings that 480,189 users gave to 17,770 movies, where they removed personally identifying information and changed some ratings. Attackers were able to recover 99% of personal data using auxiliary data from IMBD (Narayanan and Shmatikov 2006).

This tells us that all data can potentially be personally identifying information. We can not make the assumption that an adversary sees the dataset in isolation. Neither do we know what kind of auxiliary data the adversary has access to or how an adversary plans to use the data. Therefore it does not make sense to focus on making some specific data set private; instead, we should focus on the algorithms/techniques that we use to analyze the data; when doing so, we will get more meaningful guarantees about privacy. These analysis algorithms/techniques are called differential private.

This thesis examines the fundamentals of differential privacy and implements different differential privacy data structures, and measure the noise added to the data.

3 Problem definition

When releasing datasets to the world that contain sensitive personal attributes, the aggregator who releases the data should make sure that no information from an individual subject of the dataset could be gained from an adversary. To archive this, the aggregator can use different mathematical techniques that yield differential privacy. This project focuses on range counting, which is defined as processing an object S, in order to determine how much of the object intersects with a query called the range. Examples could be how many individuals of a dataset are male or how many are between the ages 20 and 25. The general technique to archive differential privacy when releasing answer to range counting is by the addition of random noise to it. When doing this, we get private range counting.

In this project, the aim is to examine some of the techniques to archive differential privacy and how these techniques can be used in combination. Furthermore, we want to implement these techniques in a localized differential privacy model and a centralized differential privacy model. Use the implementations to make experiments on how much noise they add to the answer and how effective they are on a real dataset when doing private range counting.

4 Notation

A short list of notations used throughout the thesis:

- DP = differential privacy

- F = Frequency oracle
- V_F = Variance of frequency oracle
- $\hat{\theta}[j]$ = Estimator of point j when using a frequency oracle
- $\theta[j]$ = True value of point j
- $Lap(b)$ to denote a random variable $X \sim Lap(b)$.

5 Differential Privacy

The purpose of this section is to first introduce concepts about differential private algorithms and then give a informal definition. Next, we lead this into a mathematical definition of differential private algorithms, and explain a central and local model of computation.

5.1 Sensitivity of a DP algorithm

Before we talk about the sensitivity a DP algorithms, we will first introduce how we think about databases. We will think of databases z as being collections of records from a domain $\mathcal{Z}$. The way we want to use databases is that we would want to think about their histograms, which are $z \in N^{|\mathcal{Z}|}$, each entry z_i represents the number of elements in the database z of type $i \in \mathcal{Z}$. We will now introduce a measure of the distance between two databases z and y. We will be using the the 1 norm/distance. The 1 norm of a database x is

$$||z||_1 = \sum_{i=1}^{|\mathcal{Z}|} |z|_1$$

Then the 1 norm of two databases x and y is $\ell_1 = ||x - y||_1$

$$||x - y||_1 = \sum_{i=1}^{|\mathcal{Z}|} |x|_1 - |y|_1$$

The 1 norm is a measure of how many records differ between x and y. The sensitivity of a function f is defined by the 1 norm. The 1 norm captures how much a single record, a individual persons data can change the function f in the worst case. The magnitude of the 1 norm is the 'amount of randomness' we need to introduce in the function f, in order to preserve the privacy the participation of a single individual. A formal definition would be; the sensitivity of a function gives an upper bound on how much we must perturb its output to preserve privacy (Dwork and Roth 2014, p. 17).

5.2 An Informal Definition

One way to try defining privacy from the context of data analysis is to require that the adversary does not know more about any of the individuals in the data set after the analysis is performed than the adversary knew before she/he got the analysis results. This goal is formalized by requiring that the adversary's prior knowledge and posterior knowledge about an individual should not be 'too different', or that access to the database should not change the adversary's views about any individual 'too much'. The appeal of this notion to defining privacy is that if nothing is learned about an individual, then the individual cannot be harmed by the analysis. However, that is impossible; if this requirement should be achieved, the database does not contain any information, and then why should the adversary query this database for information? Thus, this notion of privacy is unachievable (Dwork and Roth 2014, p. 13).

5.3 A Formal Definition

We first explain where the privacy comes from. The privacy comes from a process, where there is introduced some randomness, which is often done via random sampling, adding noise and linear transformations. With this stated we move on to a technical definition of differential privacy.

5.3.1 Definition Of Epsilon differential privacy

To define what it means for an algorithm to be differential private, we must first define a randomized algorithm.
A randomized algorithm M with domain A and discrete range B is associated with a mapping $M : A \rightarrow \Delta(B)$. On input $a \in A$, the algorithm M outputs $M(a) = b$ with probability $(M(a))_b$ for each $b \in B$. A randomized algorithm M gives $(\epsilon, 0)$ -differential privacy if for all databases D and D' differing on at most one row, and any $S \subseteq Range(M)$,

$$\Pr[\mathcal{M}(D) \in \mathcal{S}] \leq \exp(\epsilon) \Pr[\mathcal{D}'(x) \in \mathcal{S}]$$

(Dwork and Roth 2014, p. 17) We can see there are two quantities we must consider in DP algorithms:
ϵ: The metric of privacy loss at a differentially change in data (adding, removing one entry). The smaller the value is, the better privacy protection.
Accuracy: How much the output of DP algorithms differ from the true output. We can think of the parameter ϵ as determining the overall privacy protection. This roughly translates to the increase of the risk of individuals' privacy has of being compromised. A smaller value for ϵ implies better protection. This also translates the other way around; a larger value for ϵ gives worse protection. If we let $\epsilon = 0$ we would have perfect privacy; not a single individual in the analysis have risked their privacy at all. However, if we have $\epsilon = 0$ in the real world, the output would only consist of noise and, therefore, the analysis would be useless (Kobbi et al. 2018, p. 23).

5.4 What differential privacy does promise

Differential privacy gives a promise to the data holder from the aggregator, that any participant will not be inflicted with any harm stemming from the fact that they released their data to aggregator's private database x. Thus, a participant could still face harm. However differential privacy gives the guarantee that the probability of harm was not significantly increased by their choice to release their data.

5.5 What differential privacy does not promise

While differential privacy does deliver a strong guarantee about preserving privacy, it can not promise there can not be done harm. It can not create privacy out of thin air. Differential privacy does not guarantee that secrets disclosed in the survey will remain secret. It ensures that an individual's participation in a survey will not in itself be disclosed, nor will participation lead to the disclosure of any results that one has answered within the survey. However, if there are enough participants, the survey will disclose statistical information about the population who took the survey. The statistical information the survey obtains can then be used to draw conclusions.

The purpose of any survey is to discover statistical information about a population so we can draw conclusions about the population; if any of these conclusions hold for a given individual, it does not mean that we have violated differential privacy; Forall intends and purposes the individual may not even have participated in the survey. Differential privacy sets a guarantee that these results would be obtained with a very similar probability of whether or not the given individual participated in the survey (Dwork and Roth 2014, p. 22).

5.6 Composition theorems

We will now examine what happens if we use two differentially private algorithms in combination. We will see that the output of this will also be differentially private, this comes of the consequence that ϵ will degrade and leading to a increase of the accuracy. Consider that we repeatedly compute the same thing/statistic, with the Laplace mechanism or a frequency oracle from the sections 7.1 and 8.1 respectively. The mean of all the answers given by the DP algorithms will eventually converge to the true value of the thing/statistic, and so we cannot avoid the fact that the strength of our privacy guarantee will degrade with repeated use.
To proof this we will first show that two independent use of an ϵ_1-differentially private algorithm and an ϵ_2-differentially private algorithm, used together, is $(\epsilon_1 + \epsilon_2)$-differentially private. The proof follows here:
We let $M_1 : N^{|X|} \to R_1$ be an ϵ_1-differentially private algorithm, and $M_2 : N^{|X|} \to R_2$ be an ϵ_2-differentially private algorithm. Then their combination, defined to be $M_{1,2} : N^{|X|} \to R_1 \times R_2$ by the mapping: $M_{1,2}(x) = (M_1(x), M_2(x))$ is $\epsilon_1 + \epsilon_2$-differentially private algorithm. We fix the databases $x, y \in N^{|X|}$, we fix two points in the domain $(r_1, r_2) \in R_1 \times R_2$

$$
\begin{aligned}
\frac{\Pr[\mathcal{M}_{1,2}(x) = (r_1, r_2)]}{\Pr[\mathcal{M}_{1,2}(y) = (r_1, r_2)]} &= \frac{\Pr[\mathcal{M}_1(x) = r_1]\Pr[\mathcal{M}_2(x) = r_2]}{\Pr[\mathcal{M}_1(y) = r_1]\Pr[\mathcal{M}_2(y) = r_2]} \\
&= \left(\frac{\Pr[\mathcal{M}_1(x) = r_1]}{\Pr[\mathcal{M}_1(y) = r_1]}\right)\left(\frac{\Pr[\mathcal{M}_2(x) = r_1]}{\Pr[\mathcal{M}_2(y) = r_1]}\right) \\
&\leq \exp(\epsilon_1)\exp(\epsilon_2) \\
&= \exp(\epsilon_1 + \epsilon_2)
\end{aligned}
$$

We use this fact to show that i independent M_i mechanisms that are e_i-differentially private algorithms, used in combination, will result in a mechanism M_k that are $\sum_i^K \epsilon_i$ e_i-differentially private.
Let $\mathcal{M}_i : \mathbb{N}^{|X|} \to \mathcal{R}_i$ be an $(\epsilon_i, 0)$ -differentially private algorithm for $i \in [k]$. Then if $\mathcal{M}_{[k]} : \mathbb{N}^{|X|} \to \prod_{i=1}^k \mathcal{R}_i$ is defined to be $\mathcal{M}_{[k]}(x) = (\mathcal{M}_1(x), \ldots, \mathcal{M}_k(x))$, then $\mathcal{M}_{[k]}$ is $\left(\sum_{i=1}^k \epsilon_i, 0\right)$ -differentially private (Dwork and Roth 2014, p. 42).

5.7 Central Differential Privacy

In central differential privacy, there is a trusted aggregator who holds the entire dataset of all individual users; these can be thought of as rows. Each user then reports their row to this aggregator. The aggregator then wants to keep every individuals row private. The aggregator then uses a DP algorithm on the data which has been sent. Here we only add randomness in one place, which makes this model very accurate. The disadvantage is that the aggregator knows all actual data, which means the user really has to trust the aggregator enough to share its data with it. To obtain the trust needed for this can however be difficult. The aggregator could be an untrustworthy company or government. In the central model, the aggregator collects all the data in one place. This increases the risk of catastrophic failure e.g. if the aggregator gets compromised and leaks all the data. This model of computation can be seen on Figure 1.

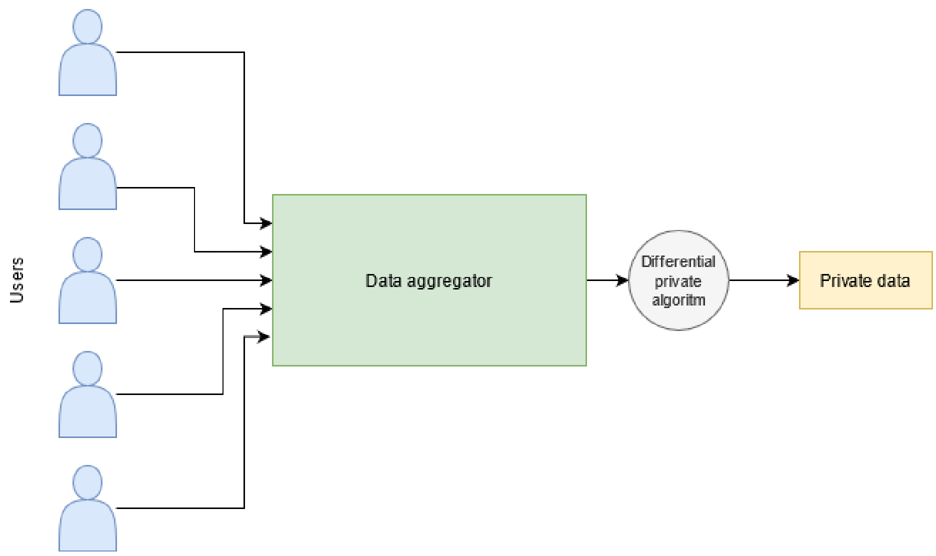


Figure 1: Model of central differential privacy

5.8 Local Differential Privacy

Initial work on differential privacy assumed the presence of a trusted aggregator, who curates all the private information of individuals, and releases information through a DP algorithm; this was explained in the previous section. In practice, individuals may be reluctant to share private information with a data aggregator. This could be because the user does not trust the aggregator or worries that the aggregator could be sold to someone they do not trust in the future. It could also be that it is hard to gain trust due to the nature of the information, e.g., a survey about illegal activities. The local variant of differential privacy is where the user only knows their data, a local view of the dataset. The users then independently release information about their data through a DP algorithm. This model of computation can be seen on Figure 2. They can even release the whole dataset while still be being differential private. Since each user adds noise to their data, this will increase the overall noise by a considerable margin over the central model. This generally means we need a lot more users to report their data to learn something about the population.

When working with local differential privacy, there arises another problem/consideration; the communication cost needs to be considered. The communication cost is the amount of data the aggregator, and a user have to exchange. We would want the communication cost to be as low as possible. This could come at the expense of some computation on both the user and aggregator ends (Kulkarni, Cormode, and Srivastava 2018, p. 3).

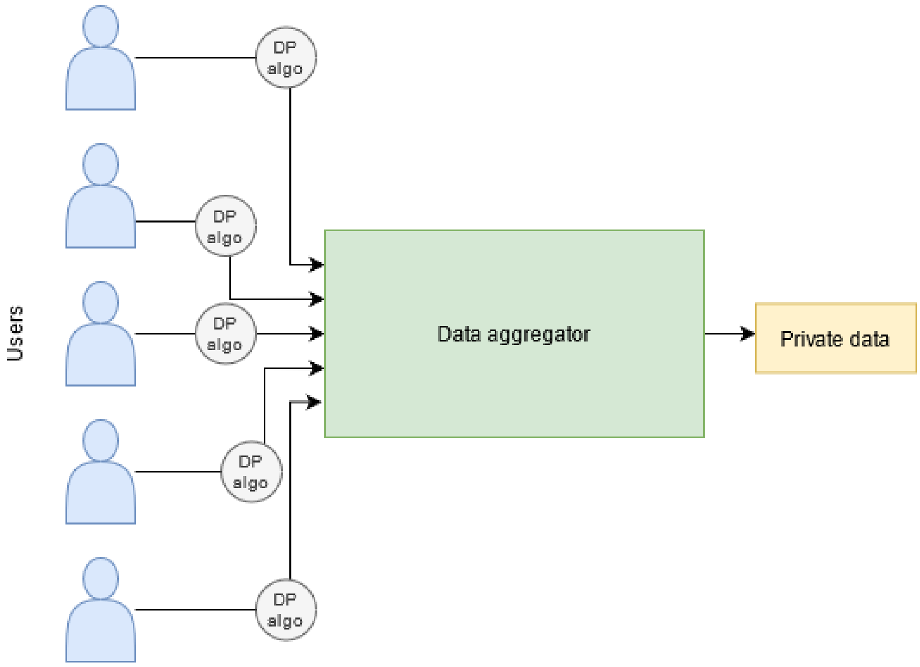


Figure 2: Model of local differential privacy

6 Point and range queries

What we would want to in this project is to focus on differential private range counting. Range counting, is defined as processing an object S, in order to determine how much of the object intersects with a query called the range. We will start by looking at point queries instead of range queries, as point queries are just range queries with range 1. With point queries, we try to estimate the frequency of of a element single element z in a domain $\mathcal{Z}$.

Then if we want to a range query, we can just sum up the z in our range. A range query has the sensitivity of 1, an individual data can only change the count by one.

7 Data Structures For Central Differential Privacy

In the central model, we know the frequency of all the elements in our domain. We now wish to make this frequency differential private. To make the frequency of all the elements differential private, we will introduce some random noise. The noise will be drawn from the Laplace distribution. We will then add this noise to the true frequency z; this will be named the Laplace mechanism.

Definition 7.1 (Laplace mechanism)**.** Given any function $f : \mathbb{N}^{|Z|} \to \mathbb{R}^k$, the Laplace mechanism is defined as:

$$f(x) + (Y_1, ..., Y_k)$$

where Y_i are i.i.d. random variables drawn from Lap$(\frac{\Delta f}{\epsilon})$ and Δf is the sensitivity of the function, in the case for counting, the sensitivity is $\Delta f = 1$.

The proof for Laplace mechanism preserves $(\epsilon,\ 0)$ differential privacy is as follows. Let $x \in \mathbb{N}^{|X|}$ and$y \in \mathbb{N}^{|X|}$ be such that $||x - y||_1 \leq 1$, and let f(·) be some function $x \in \mathbb{N}^{|X|} \to \mathbb{R}^k$. Let p_x denote the probability density function of $ML(x, f, \epsilon)$, and let p_y denote the probability density function of

$ML(y, f, \epsilon)$. We then compare the two at some arbitrary point $z \in \mathbb{R}^K$

$$\frac{p_x(z)}{p_y(z)} = \prod_{i=1}^{k} \left(\frac{\exp\left(-\frac{\epsilon |f(x)_i - z_i|}{\Delta f}\right)}{\exp\left(-\frac{\epsilon |f(y)_i - z_i|}{\Delta f}\right)} \right)$$
$$= \prod_{i=1}^{k} \exp\left(\frac{\epsilon \left(|f(y)_i - z_i| - |f(x)_i - z_i|\right)}{\Delta f} \right)$$

Using the triangle inequality gives us

$$\leq \prod_{i=1}^{k} \exp\left(\frac{\epsilon\, |f(x)_i - f(y)_i|}{\Delta f} \right)$$
$$= \exp\left(\frac{\epsilon \cdot ||f(x) - f(y)||_1}{\Delta f} \right)$$

We have that $\Delta f = 1$ and $||x - y||_1 \leq 1$

$$\leq \exp(\epsilon)$$

Which is fulfills the definition of ϵ-differentially private algorithm (Dwork and Roth 2014, p. 23).

7.1 Central Flat Solution

An obvious way to support range queries would be to use the Laplace mechanism Lap(ϵ) at each of the true frequencies and then simply sum up the estimated frequency in the range. The variance at each frequency is, therefore, $V_\mu = \frac{2}{\epsilon^2}$. We let $|r|$ denote the number of frequencies we want to sum up. Then we get that the expected error is $\text{Var}(E_m(r)) = r \cdot V_\mu$, which means the variance is linear with respect to the length of the query. The average length of the interval N is $\frac{\sum_{i=1}^{N} i(N-i+1)}{N(N+1)/2} = \frac{(N+2)}{3}$ which means the average error would be $\frac{(N+2)}{3} \cdot V_\mu$.

7.2 Continuous Observation

A different way to support range queries would be to support continuous observation of a count; in our case, each day would be the domain $\mathcal{Z}$ and store the counts at each element in the domain $\mathcal{Z}$. Then to support range queries, we would simply need to subtract the count at the last element of the range query and subtract the first count in the range.

In the book Dwork and Roth 2014 at page 243, they have a data structure to support exactly this. It works that we need the domain $|\mathcal{Z}|$ to be a power of 2. Then every interval are the natural ones corresponding to the labels on a complete binary tree with $|\mathcal{Z}|$ leaves, the leaves are labeled, starting from the left and going right, with the intervals $[0, 0], [1, 1], ..., [T - 1, T - 1]$ and each parent is labeled with the union of the interval of its children. To compute the noisy count for each leave $[t_1, t_2]$; that is, the value corresponding to the label $[t_1, t_2]$, we have a tree of the same dimensions where each leave is i.i.d Lap($\frac{1+\log_2 |\mathcal{Z}|}{\epsilon}$), we then compute the path down to our node $[t_1, t_2]$ and add each Laplace variable on the way, this is then the noisy count for leave $[t_1, t_2]$. To learn the count of an element in the domain, we sum the nodes in the B-adic decomposition of the range. To answer a range query, we do need two elements in the domain and subtract them from each other. The pseudocode for the algorithm can be seen in Figure 12.1 on page 243 in the book Dwork and Roth 2014.

Now we show why this ensures $(\epsilon, 0)$-differential privacy; we know each element in the domain appears at most $1 + \log_2 |\mathcal{Z}|$ intervals as the height of the tree is $log_2|\mathcal{Z}|$. So every element in the domain can only affect the output $1 + \log_2 |\mathcal{Z}|$ times. Then if we add noise to each leaf distributed according to $\mathrm{Lap}(\frac{1+\log_2 |\mathcal{Z}|}{\epsilon})$ it ensures $(\epsilon, 0)$-differential privacy. This argument is easily extended for other than a binary tree. This data structure is almost identical to the local hierarchical histogram we will describe later on.

It is important reuse the same Laplace variables for the noisy counts and not sample some new ones. First I implemented a data structure where I sampled new Laplace variables for every count. This is however not DP, because we can let the size domain get really big to increase the height of the tree. Then the law of large numbers tells us the Laplace variables would cancel out as they mean 0.

8 Data Structures For Local Differential Privacy

In contrast to the central case, we do not know the true count at each point. Each user i holds a private element z_i from the domain $\mathcal{Z}$. This domain can be describes as a unknown discrete distribution θ, where θ_z is the probability that a randomly sampled input element is equal to $z \in \mathcal{Z}$. We can see this domain as a vector with a one in the index where z_i belongs and zeros every where else. We want have a local DP protocol so the aggregator can estimate θ as $\hat{\theta}$ as accurately as possible.

8.1 Frequency Oracle

Solutions for this problem are referred to as providing a frequency oracle. There have been several suggestions of frequency oracles described in recent years. In each case, the users perturb their input on their own data (locally), often via linear transformation or random sampling, and send the result to the aggregator. The noisy reports each user reports are aggregated together and corrected by the expected noise to reconstruct the frequency for each item in $\mathcal{Z}$. The estimators for these mechanisms are unbiased and have the same variance with the same bias V_f for all items in the input domain (Kulkarni, Cormode, and Srivastava 2018, p. 3).

Three different versions of frequency oracle is described on page 3 of Kulkarni, Cormode, and Srivastava 2018 . Where the focus of this thesis will be of the one they call modified Optimal Local Hashing (OLH). In the paper they use hashing to reduce the communication cost between the individuals and the aggregator. In contrast, when I will be using the frequency oracle, I will both be the individuals and the aggregator so there are no communication cost. There is also a flaw in the paper as it seems they ran out of symbols to use, where they use g for two things, both the hashing range and the variable to minimise the variance of the frequency oracle in a way that contradict each other. The frequency oracle works as such, with probability $\frac{e^\epsilon}{e^\epsilon+g-1}$ the individual answer truthfully or else reports a value sampled u.a.r from the domain. The aggregator then collect all the individuals reports and computes a frequency vector for all items in the original domain, based on what was reported from all N individuals. All N such histograms are added together to give $T \in \mathbb{R}^D$. The aggregator then uses the unbiased estimator $\hat{\theta}(i) = \frac{T[i]-(1-p)\cdot\frac{N}{g}}{p}$ to get the frequency for all elements in the original domain, the variance should is $V_f = O\left(\frac{e^\epsilon}{N(e^\epsilon-1)^2}\right)$ (Kulkarni, Cormode, and Srivastava 2018, p. 3).

8.2 Local Flat Solution

The flat solution for local DP, is almost the same as with central DP, instead of adding the noise of Laplace, the permutation comes from the frequency oracle. We can see that for an interval $[a, b]$, we let the range be $R_{[a,b]} = \sum_{i=a}^{b} f_i$, where is our estimated frequency f_i of $i \in \mathcal{Z}$. This frequency is estimated by our frequency oracle. Therefore a simple approach is to sum up estimated frequencies for every item in the range. The error behaves exactly the same way as in the central flat solution.

8.3 Local Hierarchical Histogram

Before looking at hierarchical histogram, we introduce the notion of B-adic intervals and a property of B-adic decompositions. An B-adic intervals if it is on the form $kB^j \ldots (k+1)B^j - 1$ for $j \in [log_B D]$ and $B \in \mathbb{N}^+$. Any subrange of in a B-adic intervals of length r can be decomposed into $\max{(B-1)}(2\log_B r + 1)$ sub intervals.

If we look at the range query problem as representing answering collections of histograms, each element in the domain is a bin. In the local flat solution we have bin for each element. This leads to an error that is linear in the length of the range. We can ask oneself what if we keep some bins for the subranges of the domain instead of having a bin for all of the domain. A way to do this is impose a hierarchy on the domain items in such away that the frequency of each item contributes to multiple bins. With this structure we should be able to answer, queries by adding counts from a smaller number of bins.

In the hierarchical histograms, we arrange the intervals into a tree with branching factor b, where the unit-length intervals are the leaves, and each node in the tree corresponds to an interval that is the union of the intervals of its children. An illustration of the nodes subranges in a hierarchical histograms with an degree of two can be seen on Figure 3.

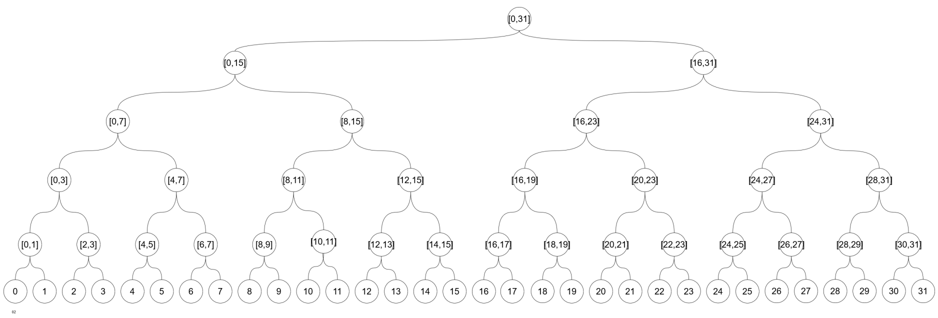


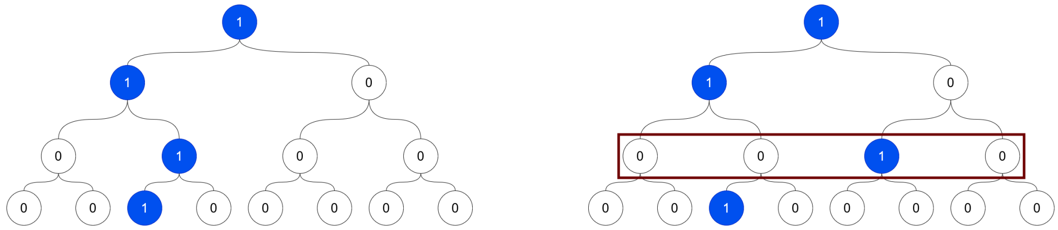
Figure 3: Hierarchical histograms representation

Each user i has an item z_i from the domain, z_i will match with a leaf in the bottom of the tree. The user i arranges their input $z_i \in \mathcal{Z}$ as a full tree of height with degree B. z_i will have a unique path from leaves to the root with a weight of 1 attached to each node on the path, and zero elsewhere, see Figure 4a for the local view with $z_i = 2$ and $b = 2$. We can therefore see each level in the tree as a vector with 1 in one place and zero everywhere else. Hence, we can use our frequency oracle from section 8.1, all the levels.

I will now present the steps that the users has to do.
User i samples a random level with probability p_l, then perturbs this vector using the frequency oracle, reports the perturbed information to the aggregator along with what level it was, see Figure 4b for an example of a perturbation of level 3 with $z_i = 2$.

I will now present the steps that the aggregator has to do.
The aggregator builds the same empty tree and adds what each individual contributes to the corresponding nodes. The aggregator answer a range query by using the frequency oracle estimator on the nodes from the B-adic decomposition of the range and times it with height of the tree to compensate for the random sampling of levels (Kulkarni, Cormode, and Srivastava 2018, p. 5). Something to note here is that we loose a little piece of information by doing this. We only add data to one node in the path, but we still know the path up in the tree as it is identified by all parents of the nodes. We could not do something similar with the children of this node as we do not known which of the

children leaf we should contribute to.



(a) Local view with $z_i = 2$ (b) Local view with $z_i = 2$ and perpetuated level 3

Figure 4: Local view with $z_i = 2$ and perpetuation

8.3.1 Error for Local Hierarchical Histograms

We denote hierarchical histograms with degree B as HH_B. First we show that overall variance can be expressed by the variance of the frequency oracle, V_f. We remember that the variance of the frequency oracle is $V_f = O\left(\frac{e^\epsilon}{N(e^\epsilon-1)^2}\right)$ where N is the amount of people contributing to the frequency oracle. Here we observe that the variance does not depend on the domain, which in our case is size of the levels, the variance depends only the amount of people who contributed to the frequency oracle. We can then introduce $V_F \leq \psi_F(\epsilon)/N$ where $\psi_F(\epsilon)$ is a constant for method the frequency oracle that depends on ϵ. This gives us the ability to fix the variance for all nodes in any to be in the same level to be V_l. V_l the variance of a given level is determined by how many users randomly sample that level N_l. We remember that the range query of length r is decomposed into at $2(B-1)$ nodes at each level, we can at maximum use $\alpha = \lceil log_B r \rceil + 1$ levels. This gives the bound of total variance to be

$$\sum_{l=1}^{\alpha}(2B-1)V_l = \sum_{l=1}^{\alpha}(2B-1)V_F/p_l = (2B-1)V_F\sum_{l=1}^{\alpha}\frac{1}{p_l}$$

It can further be proven that to minimise variance we need to set $p_l = \frac{1}{h}$, which means the users sample their level uniformly at random (Kulkarni, Cormode, and Srivastava 2018, p. 5).

We can further optimise the error by adjusting our degree B of the HH_B. This comes from the fact that a larger degree will reduce the height of the tree. This will increases accuracy of estimation per node since larger fraction of the users is allocated to each level. But doing this, it also means that we require more nodes at each level to evaluate a query which we have shown increases the total error.

9 Implementation

The code can be found in the appendix and also on the GitHub page: `https://github.com/jfriisKU/Bachelors-Thesis`, the zip file and in appendix here 14.1. The code was implemented in python 3.6 The following python libraries have been used in the implementations:

- Numpy
- Pandas
- datetime
- scipy
- matplotlib
- os
- Re
- sys
- psycopg2
- unittest

The numpy library allows us to quickly and nicely manipulate arrays.
The domain of the dataset dates, therefore we need a library to handle that, pythons standard library datetime was chosen to handle this.
The psycopg2 library allows us to interact with a PSQL database. This was needed as we loaded our dataset into PSQL database to access of the data in different python files.
The library scipy allows us get an implementation of the Laplace distribution where we could control the scale of the distribution. This was need in both of the implementation of the central models, which relies on the Laplace distribution.
The library matplotlib made plotting and showing the results much quick and easy.
Os, Re and sys was mainly used for saving and loading the results from the experiments on the data structures for further analysis.
Unittest was used to perform the unit tests on the data structures.

9.1 Generally for all data structures

In all the implemented data structures, I have saved the true count of every element. This was done to make the experiments and the later analysis much easier, as we can ask the data structure what the true answer was and saved it with the estimation.

9.2 Central flat solution

The implementation for the Central flat solution, takes three arguments ϵ, the domain and the counts of each element in the domain. When running the implementation, the data structure adds Lap(ϵ) to every count and saves this count in a new array. This new array is then used to answer the range queries, by summing up every element given in the range.

9.3 Local flat solution

The implementation for the local flat solution, takes three arguments ϵ, the domain and the counts of each element in the domain. We represent the domain as a 1d array, where each element in the array corresponds to element in the domain. When running the implementation, the data structure, estimates the frequency in the domain, we use the frequency oracle on every count in each element in of the domain. The frequency oracle reports element of the domain, we then add one our estimation of the frequency in the domain. The array of noisy counts the frequency oracle produced is then, used to answer the range queries, we use the unbiased estimator on every element given in the range, and sum them up when doing a range query.

9.4 Answering Queries In Hierarchical Histograms

As noted in the previous section about continuous observation and local hierarchical histograms. The two data structures do not differ that much in concept. They both answer range queries by making use B-adic decomposition. In both the Continuous Observation and local Hierarchical Histograms, I did not make the B-adic decomposition, but i opted for another way for finding the leaves in the B-adic decomposition.

One can instead look at it as a binary search down the tree after the nodes just before and after the range. When we search for a node just before our range then every time we go to the left child, we need to count the node at our right. When we search for the node just after our range, then every time we go to the right child, we need to count the node at our left. After doing this search we have found all the nodes we need in our B-adic decomposition.

An example of this is given here, say we are interested in the range $[2, 22]$ and we have $B = 2$ and $D = 32$, this range can be decomposed into $|2, 3| \cup |4, 7| \cup |8, 15| \cup |16, 19| \cup |20, 21| \cup |22, 22|$.

This search in the tree is illustrated in Figure 5, where the green line is the search for node right before our query and the purple line is the search for our node right after our range. The nodes in the red circles are the ones we want to count.

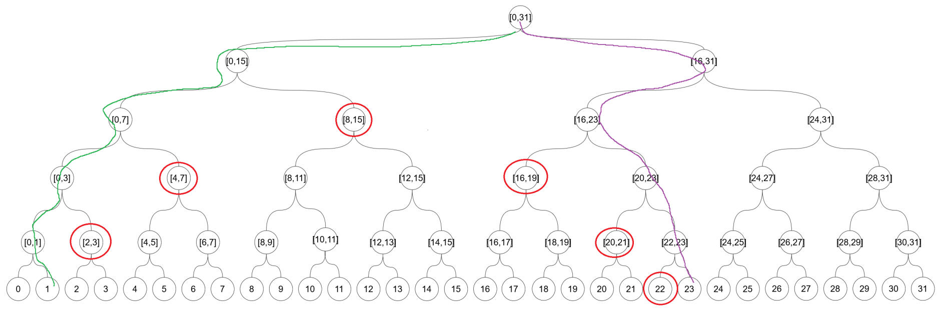


Figure 5: Range query search in hierarchical histograms

9.5 Continuous Observations

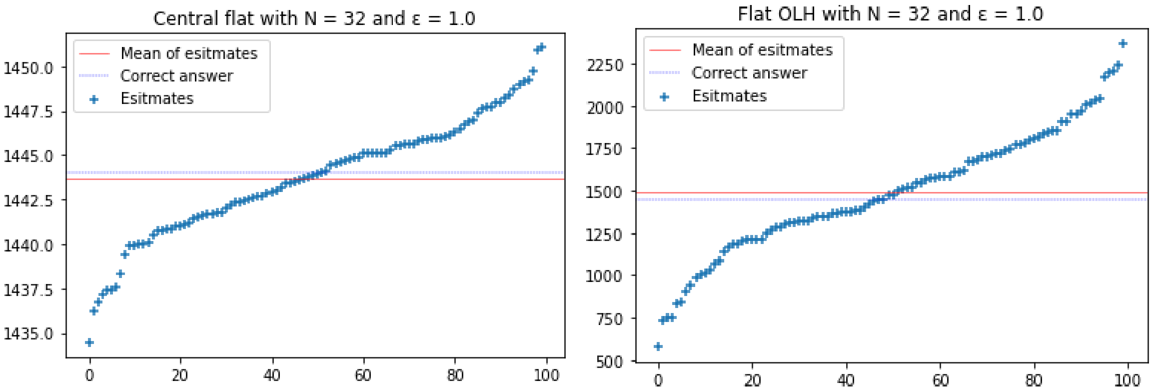
The implementation for the continuous observations solution, takes four arguments ϵ, the domain, the counts of each element in the domain and a degree. When running the implementation, the data structure makes a true continuous observations hierarchical histogram of the domain, and a tree with the same dimension, where it stores in the leaves the sum of all the Laplace variables, needed for the corresponding leaf in true histogram. The true histogram and the tree with Laplace variables are then added together. To answer a range query, we need the continuous count from the last element in the range and subtract it from continuous count from the first element of the range. To get the continuous count we use the method described in section 9.4.

9.6 Local Hierarchical Histograms

The implementation for the Local Hierarchical Histograms, takes 4 arguments ϵ, the domain, the counts of each element in the domain and a degree. When running the implementation, the data structure makes an empty full B-ary tree with the given degree. It then goes for every count of every element in the domain. We sample a random level and use the frequency oracle on this level and, and adds this to the corresponding level of the full B-ary tree. To answer a range query, we need to get the nodes corresponding to the B-adic decomposition, as described in section 9.4. We then sum up what the frequency estimator response with on the nodes and times this with the height of the tree.

9.7 Testing of the implementation

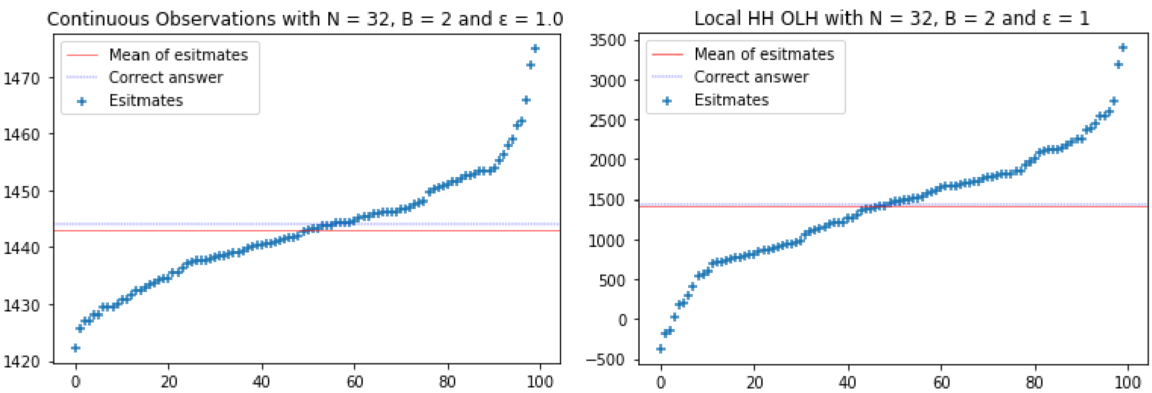
The testing strategy was unit testing. Some of the things in the data structures can be quite difficult to use unit testing, as they rely on randomness. To test the random things I instead chose to do some sanity checks and self inspect to see if the randomness performed as 'expected'. An example of this was testing the 'randomness' to see if we got the correct result. We could run the same range query on 100 data structure with the same parameters. We could then get a CDF of the results and check if they match with the expectation. Also as a consequence of the composition theorem, if we average the 100 results we would get a DP algorithm with privacy guarantee of $100 \cdot \epsilon$ that we used in our data structure. These sanity check for the data structure implementations can be seen in Figures 6 and 7. Some other sanity checks of the randomness, would the mean of all the levels in a local HH, the mean should roughly to be the same. The unit testing can be found in appendix 14.2 and on the GitHub page.



(a) Sanity check central flat data structures

(b) Sanity check local flat data structures

Figure 6: Sanity checks of data structures



(a) Sanity check continuous observations structures

(b) Sanity check local HH data structures

Figure 7: Sanity checks of data structures

10 Experiments and results

10.1 Dataset

The dataset used for experiments describes the number of people visiting different public libraries in Aarhus. The data consist of three columns. The first column contains the dates with timestamps in the format 'Y-m-dTtimestamp', and the second column is the number of people who went into the library in the given hour. The third column is ID string which identifies what library in Aarhus the datapoint belongs to. There were 16 different libraries in the dataset. I chose to sum up the number of visitors on a day instead of having them as individual hours. In the experiments the data from library with id code of 775147.

In the data, some dates were missing due to holidays and etc. This was corrected by adding the missing dates and set the number of visitors for that day to 0. The data was loaded into a Postgres database. This allows for easier access to the data when doing experiments.

The dataset comes from `https://www.opendata.dk/city-of-aarhus/besogstal-og-abningstider-for-aarhus-kommunes-biblioteker#resource-bes%C3%B8g` under section Besøg (2014-2019).

10.2 Generation of range queries of specific a length

For the experiments and test of the different data structures, we would need different range queries of the same length. These queries was generated at random, by sampling a date uniformly at random of the domain and adding the length of the range query to this date. We then check if the new date is still in the domain of the dates. If not we re-sample and do the same thing until a range has been sampled.

10.3 Flat solutions with varying length of queries

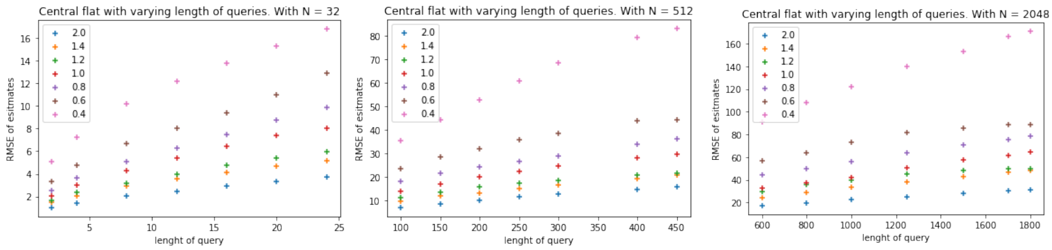
As we have shown in both the sections 7.1 about the central flat solution, the error of flat solutions scales linearly with the length of the queries. We also expect the error to scale with ϵ. We made data structures with varying domain sizes N and different privacy variables ϵ, and ran range queries of varying length on the models to test this. The error measurement was the root mean error. We had the parameters, $\epsilon' s = [2, 1.4, 1.2, 1, 0.8, 0.6, 0.4, 0.2]$, $n' s = [32, 128, 256, 512, 1024, 2048]$. The length r for the queries varied depending on the domain size N. The testing was performed with seven different lengths of queries. The lengths can be seen on table 1. A total of 2500 queries being estimated by 25 different data structures, 100 queries each data structure.

Domain size	r_1	r_2	r_3	r_4	r_5	r_6	r_7
32	2	4	8	12	16	20	24
128	20	40	50	60	70	80	90
512	40	60	80	100	140	200	220
1024	200	300	400	500	600	800	900
2048	600	800	1000	1250	1500	1700	1800

Table 1: length of queries depending on N

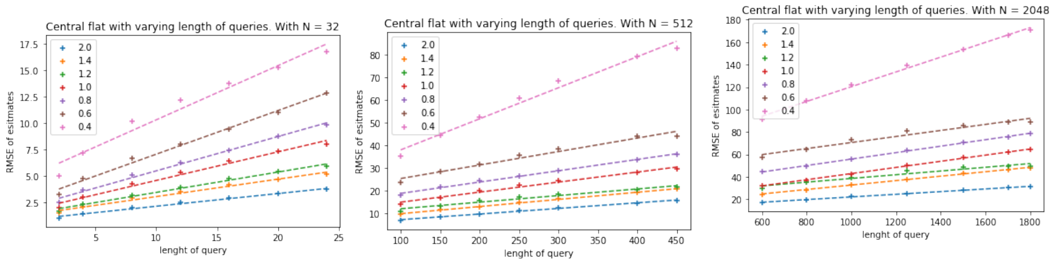
10.3.1 Central flat solution

Here we present the results for the central flat solution. All the plots of the errors can be seen in here 14.3.1. I have chosen to display three of plots with the domain size N being $[32, 512, 2048]$. All the plots show the same tendencies, so there is no reason to look at them all. Plots of the results can be seen on Figure 8. We can quite clearly observe that the error grows linearly with the length of the query, as we would expect. This is further examined in plots of Figure 9. Here the errors (dependent variable) and the length of the queries (interdependent variable) are fitted with linear regression. All linear fitted functions have an R^2 value in the high .9 with the lowest r^2 being 0.9715. We can also see that smaller ϵ values gives a higher error, which its also in line with what we would expect.



(a) RMSE over r for $N = 32$ (b) RMSE over of r for $N = 512$ (c) RMSE over r for $N = 2028$

Figure 8: Central flat plots with RMSE over r



(a) Linear fit of error for $N = 32$ (b) Linear fit of error for $N = 512$ (c) Linear fit of error for $N = 2028$

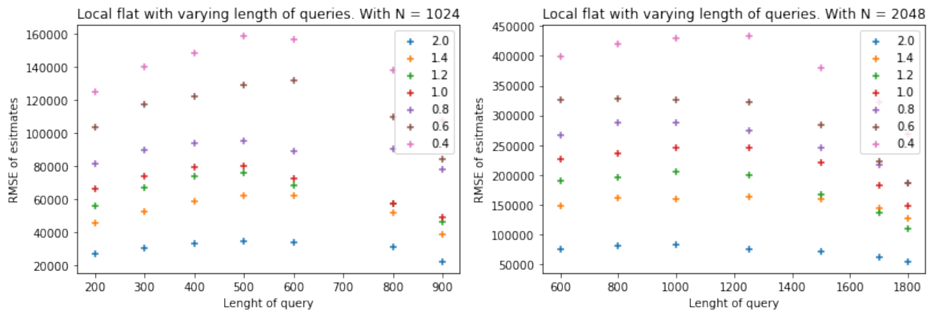
Figure 9: Central flat plots with RMSE as function of r

10.3.2 Local Flat Solution

Here we present the results for the local flat solution. All the plots of the errors can be seen in here 14.3.2. I have chosen to display 3 of them with the domain size N being $[32, 512, 2048]$. All the plots show the same tendencies, so there is no reason to look at them all. Plots of the results can be seen on Figure 8. We can here observe that the error does not scale linearly with the length of the query, as we would expect, and as the theory states. It looks like it is scaling length of the query at the start but then stops, and the error decreases all of the sudden. It error seems more so to follow a second degree polynomial. This back up by the plots in Figures 12 and 11. We can see the linear regression fits really poorly and the second degree polynomial fits much better.

To understand why this is happening we have to look at the frequency oracle, so the frequency oracle either responds with the correct z with a certain probability or it chooses a random $z \in \mathcal{Z}$, because of element z of $\mathcal{Z}$ is counted once, where it belongs in the domain is just random. When we make a range query over a large percentage of the domain, it will not matter if responded truthfully about the location in the domain, as long as our u.a.r response was still in the part of the domain that is a part of the current range query, because it will then counted in this range either way. This is also evidently by when i did a range query over the whole domain with the flat solution. Every time it answered with the true answer (bearing in mind some floating point operations errors), because we count every element in the domain, but at wrong locations.

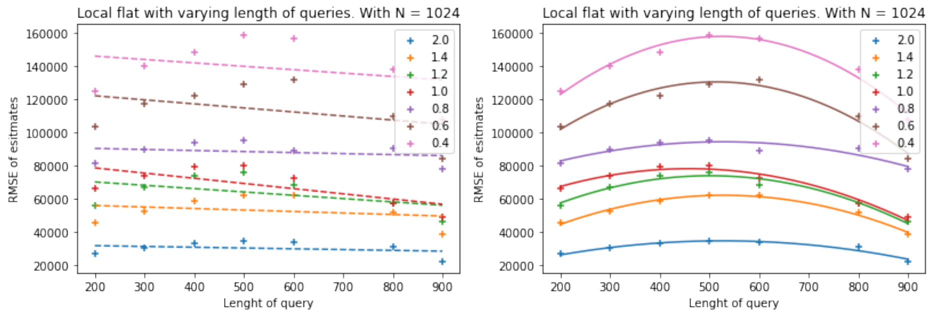
This gives me some doubt about the differential privacy of this frequency oracle. If we assume a company did local DP survey where all their users responds with this implementation. We let the company do a range query on the whole domain, about how many of users satisfies property y. After one single day they get just a single new user, and run the same query. If the count of how many satisfies property y goes up by one, we know with 100% certainty that this new user satisfies property y.



(a) RMSE over r for $N = 1024$

(b) RMSE over r for $N = 2028$

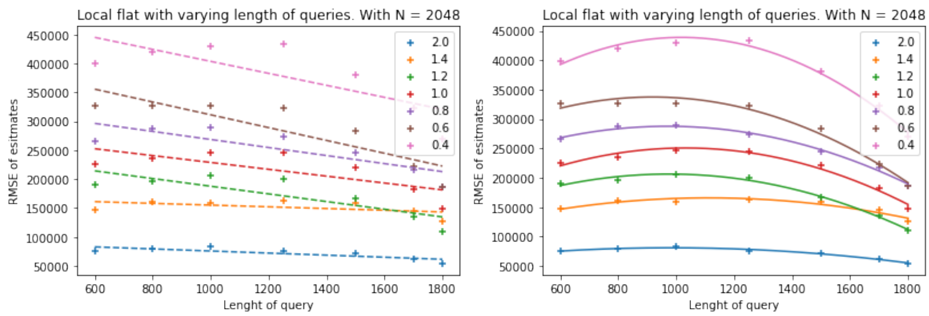
Figure 10: Local flat plots with RMSE over r



(a) Linear fit of error for $N = 1024$

(b) Poly fit of error with $N = 1024$

Figure 11: Linear regression fit of RMSE as function of r



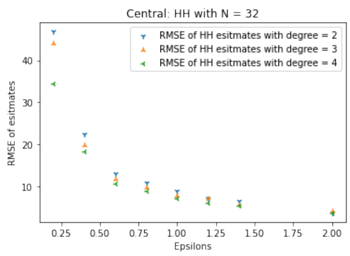
(a) Linear fit of error with $N = 2048$

(b) Poly fit of error with $N = 2048$

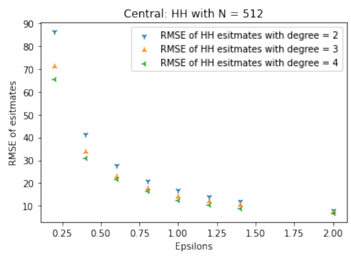
Figure 12: Second degree polynomial fit of RMSE as function of r

10.4 Continuous observation impact of ϵ and degree

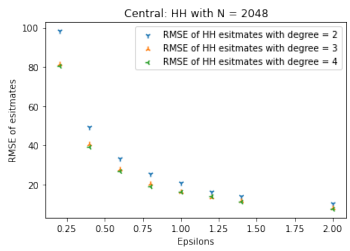
In this section we want to measure the impact of ϵ, and the degree of our tree has on the error on the continuous observation data structure. A total of 2500 queries being estimated by 25 different data structures, 100 queries each data structure. The plots of the error over 8 different epsilon values ranging from $[0.2, 2]$ can be seen on Figure 13. We can see from the plots that the smaller the epsilon value, the higher an RMSE value we get, which correlates with the theory from section 5.3. The continuous observation with the higher degree also tends to score a lower RMSE value. This is because the degree increase will decrease the tree's height, and therefore we need fewer Laplace variables. It looks like the RMSE values, when plotted, follow an exponential decreasing function by inspecting the graphs. This gives reason to plot the RMSE error with $\frac{1}{\epsilon}$ as the x-axis. The plots with RMSE over $\frac{1}{\epsilon}$ can be seen on Figure 14. This indicates linear dependence between RMSE and $\frac{1}{\epsilon}$. This is confirmed when fit a line regression model of RMSE and $\frac{1}{\epsilon}$, as seen on Figure 15. All of these regression models have R^2 value of .99. We can therefore deduce that in the continuous observation the error grows accordingly to $\frac{1}{\epsilon}$.



(a) Continuous observation with $N = 32$

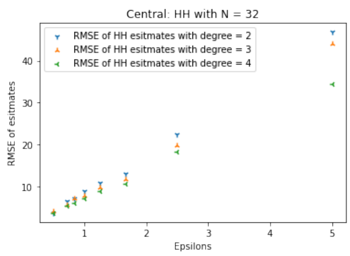


(b) Continuous observation with $N = 512$

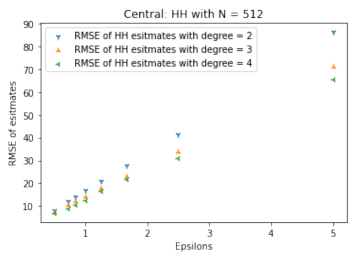


(c) Continuous observation with $N = 2048$

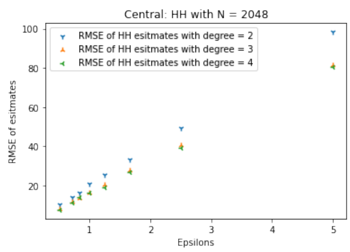
Figure 13: RMSE of continuous observation with different height and domain size



(a) Continuous observation with $N = 32$

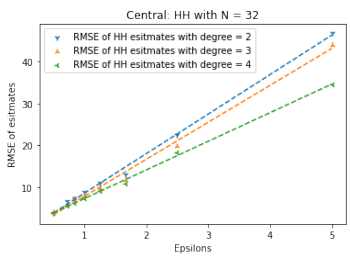


(b) Continuous observation with $N = 512$

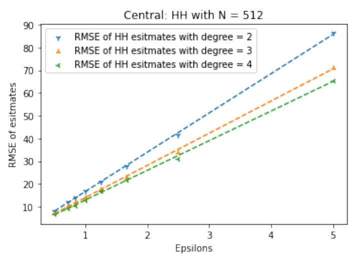


(c) Continuous observation with $N = 2048$

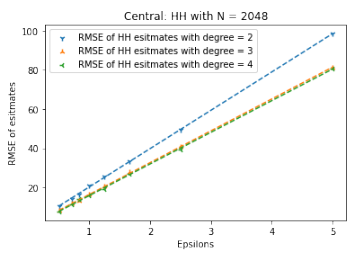
Figure 14: RMSE of continuous observation over $\frac{1}{\epsilon}$



(a) Continuous observation with $N = 32$



(b) Continuous observation with $N = 512$

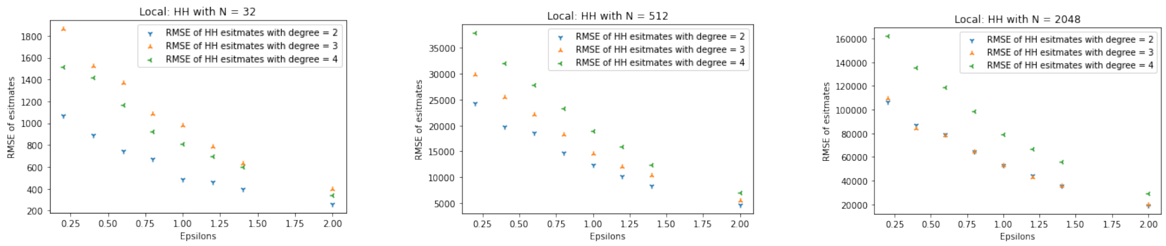


(c) Continuous observation with $N = 2048$

Figure 15: Linear fit between RMSE and $\frac{1}{\epsilon}$

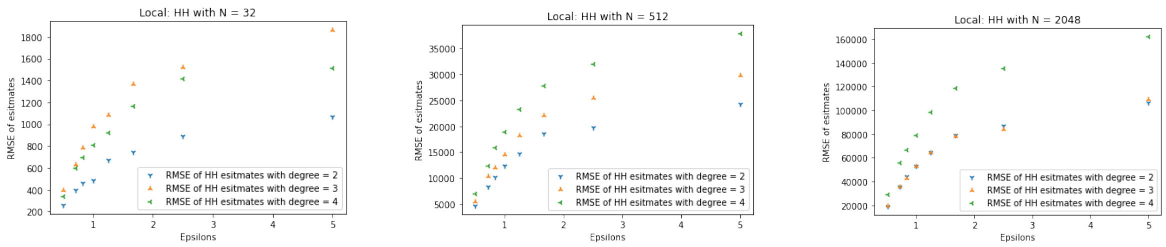
10.5 Local HH impact of ϵ and degree

In this section, we want to measure the impact of ϵ and degree has on the error on the local HH data structure. The plots of this can be seen on 16. We can see from the plots that the smaller the epsilon value, the higher an RMSE value we get, which again correlates with the theory from section 5.3. In the continuous observation data structure, the one data structure with the higher degree also tends to score a lower RMSE. This is not the case with local HH data structures. Here it is generally the lower degree HH that gets a lower RMSE value. This is not what we would expect; the theory from 8.3, clearly states that if we increase the height, we should count a lower amount of leaves in the tree. I am not really sure why this is the case. Just by inspecting the graphs, it looks like the linear function for small ϵ values. However, if we apply the same trick as before by plotting the RMSE over the $\frac{1}{\epsilon}$. We can observe the RMSE grows cube function of $\frac{1}{\epsilon}$, see Figure 17. Which we can confirm by setting the x-axis to log scale and fitting a linear regression model. The semi-log plots be seen on Figure 18. We can therefore deduce, that in the local HH, the error grows accordingly to $\frac{1}{\epsilon^2}$.



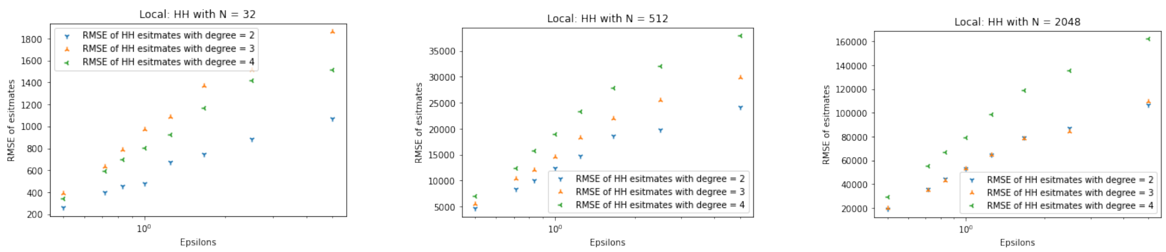
(a) Local HH with $N = 32$ (b) Local HH with $N = 512$ (c) Local HH with $N = 2048$

Figure 16: RMSE of local HH with different height and domain size



(a) Local HH with $N = 32$ (b) Local HH with $N = 512$ (c) Local HH with $N = 2048$

Figure 17: Local HH; RMSE over $\frac{1}{\epsilon}$



(a) Local HH with $N = 32$ (b) Local HH with $N = 512$ (c) Local HH with $N = 2048$

Figure 18: Semi log plot of RMSE over $\frac{1}{\epsilon}$

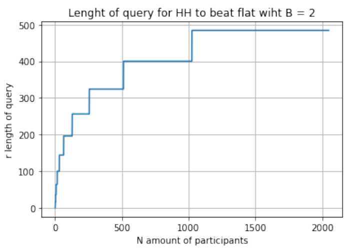
10.6 Flat solutions vs HH solutions

There were a total of 2500 queries being estimated by 25 different data structures in these experiments, 100 queries each data structure. As explained earlier, the local hierarchical histogram data structures resemble the data structures for continuous observation very closely. They both build on the same thought of imposing a hierarchy on intervals in the domain. I will therefore denote the continuous observation as a hierarchical histogram for the experiments in the next section.

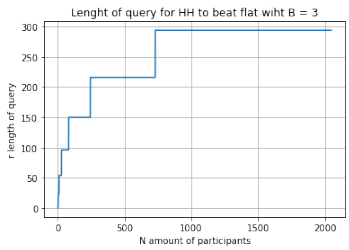
10.6.1 When does hierarchical histogram beat the flat solutions?

The benefit of the hierarchical histogram approach over the baseline flat method comes from the fact that we, at some point, need to visit fewer nodes in HH than in the flat solutions. Sometimes we could answer a range that is defined by a single node in the HH where we would need many more lot of point estimations in the flat solution. We have shown in a previous section that the number of nodes we need to visit in the HH depends on the height of the HH, which depends on the degree of the HH. The number of nodes we needed to visit in a HH is $(2B-1)h\alpha$ versus in the flat approach is the quantity r, when answering a range query. We have that $h = \log_B(|\mathcal{Z}|) + O(1)$ and $\alpha = \log_B(r) + O(1)$ from our previous section about local hierarchical histogram. Therefore we obtain an improvement over flat methods when $r > 2 \cdot B \log_B^2(|\mathcal{Z}|)$.

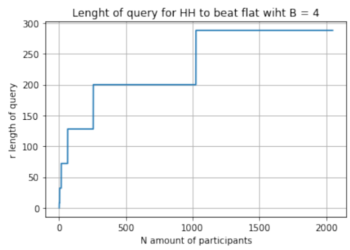
I have plotted the maximum length of the queries that hierarchical histogram needs to beat flat solutions for various degrees of a hierarchical histogram. The plots of this can be seen in figure 19. The HHs could potentially beat the flat solution earlier if the range matches the interval of one leaf in the tree. We can see from the plots that the HH might not beat the flat solutions for some combinations of B and $|\mathcal{Z}|$ for example. When if we pick $|\mathcal{Z}| = 64$ and $B = 2$, the length for when the HH starts beating the flat solution is 144, which is impossible as it can maximum be 64. The length used for the experiments were HHs beats the flat can be seen in table 2. The length used for the experiments were flat beats the HHs can be seen in table 3.



(a) Length needed for $B = 2$



(b) Length needed for $B = 3$



(c) Length needed for $B = 4$

Figure 19: Length needed for HH to beat flat for various degrees

N	Length
256	Only possible to on whole range
512	324
1024	400
2048	484

N	Length
256	216
512	216
1024	294
2048	294

N	Length
256	128
512	200
1024	200
2048	288

Table 2: Length of queries for HH to beat flat with $B = 2$, $B = 3$ and $B = 4$ respectively

N	Length
32	6
128	9
256	24
512	35
1024	44
2048	64

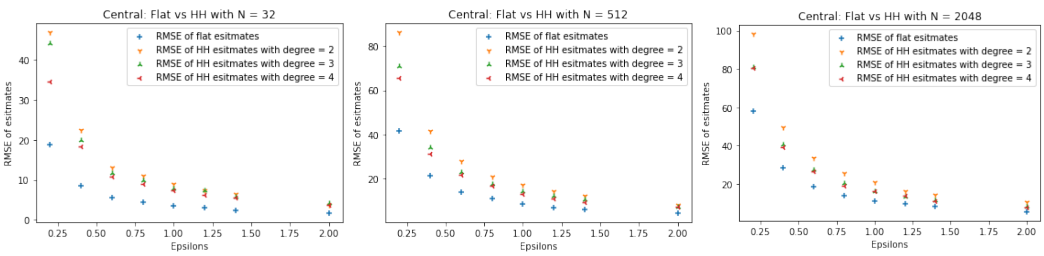
Table 3: Length of queries for flat to beat HH with different domain sizes

10.6.2 Flat solutions beating the hierarchical histogram solutions

Here we present the results for when the flat solutions should beat hierarchical histogram solutions in both local and central differential privacy models. All the plots of the errors can be seen here 14.3.3. We have chosen to present six plots, three for both the local and central solutions. As all the plots show the same tendencies, there is no reason to look at them all. I have chosen to look at the plots with a domain sizes $N \in [32, 512, 2048]$, the length of the queries would be $N \in [6, 35, 64]$ in the respective domain sizes. The RMSE value for the queries of a specific length is plotted over the eight different epsilon values in the plots. ϵ takes the values $\epsilon \in [2, 1.4, 1.2, 1, 0.8, 0.6, 0.4, 0.2]$. The plots for the central models can be seen in figure 20. The plots for the local models can be seen on figure 21.

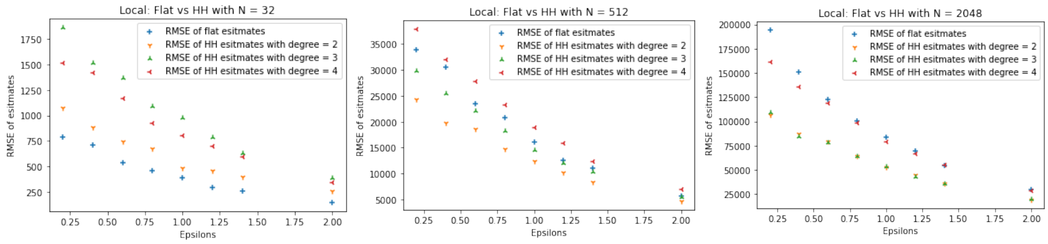
We can see that the central models follow what we would expect, but the local models do not. In the central case, we can clearly see that the flat solution beats the hierarchical histogram solution. We can also see that the lowest RMSE value of the hierarchical histogram data structures is the one with the largest degree, this line with the theory.

In the local case only for $N = 32$, the local flat solution beats the local hierarchical histogram. For $N = 512$, it is beaten hierarchical histogram solution with degrees two and three. For $N = 2048$ it is beaten by every hierarchical histogram solution. One potential reason for this could be that the range queries represented exactly a single node in the hierarchical histogram. However, if this were the case, we would expect similar results for the central models. This leads me to think something else is the cause for this result. Another potential reason could be that we know from the section 8.1, the variance of an element in the domain depends upon how many contribute to the mechanism. So the variance would be high for each element in the flat solution as everyone contributes to the mechanism if there is a lot in the domain. Where in the hierarchical histogram, the variance would be spread uniformly out over the mechanism for each level. Therefore the variance of mechanism at each of the levels would be lower. So when we count a few high variance values vs. some more medium variance values, the overall variance would be smaller from the medium variance.



(a) $N = 32$ (b) $N = 512$ (c) $N = 2048$

Figure 20: Central flat beating central HH

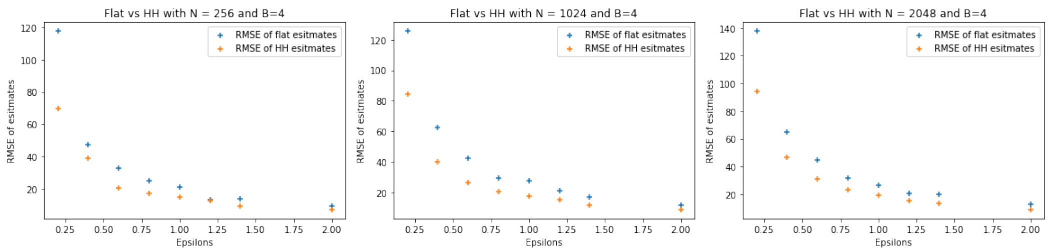


(a) $N = 32$ (b) $N = 512$ (c) $N = 2048$

Figure 21: Local flat beating local HH

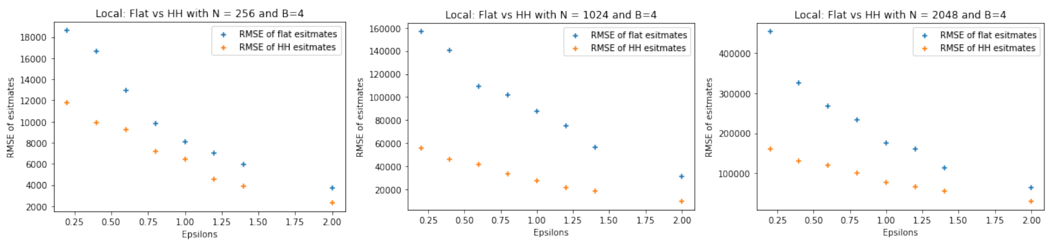
10.6.3 Hierarchical histogram solutions beating the flat solutions

Here we present the results for when the flat solutions should beat hierarchical histogram solutions in both local and central differential privacy models. We have chosen to present six plots, three for both the local and central solutions. As all the plots show the same tendencies, there is no reason to look at them all. I have chosen to look at we domain sizes $N \in [256, 512, 2048]$ all with degree $B = 4$; this means the length of the queries would be $N \in [128, 200, 288]$ respectively for the domain sizes. The RMSE value for the queries of a specific length is plotted over the eight different epsilon values in the plots. ϵ takes the values $\epsilon \in [2, 1.4, 1.2, 1, 0.8, 0.6, 0.4, 0.2]$. The plots for the central models can be seen in figure 22. The plots for the local models can be seen in figure 23. We can clearly see that the hierarchical histogram solutions beat flat solutions as expected. The margin is not that wide with the central models, where the margin for the local models is far more extensive. We would assume this gap to get bigger the longer the ranges become.



(a) $N = 256$ and $B = 4$ (b) $N = 512$ and $B = 4$ (c) $N = 2048$ and $B = 4$

Figure 22: Central hierarchical histogram beating central flat solution

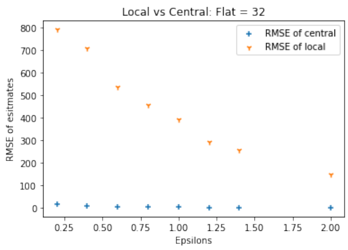


(a) $N = 256$ and $B = 4$ (b) $N = 512$ and $B = 4$ (c) $N = 2048$ and $B = 4$

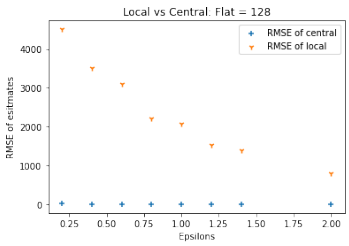
Figure 23: Local hierarchical histogram beating local flat solution

10.7 Local vs Central

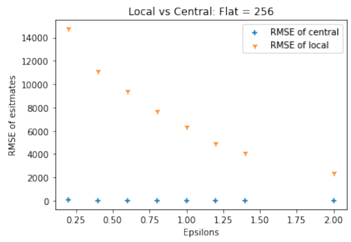
In this section we want to see what happens in the error when we go from the central to the local model. The queries used to examine this is the same from the previous section. We $\epsilon \in [2, 1.4, 1.2, 1, 0.8, 0.6, 0.4, 0.2]$ in our data structures. We would expect that the error would be worse in the local solutions. This is also what we observe in the plots both for the flat solutions in Figure 24 and the HH solution in Figure 25. The extend of how much worse local solution is quite surprising. In the plots the errors for the central solution is pretty much a straight line at the bottom of the plot around 0 to 200, wheres the with the errors of the local solutions are are in the 10s of thousands. This could be because the error grows both in proportion with the ϵ and the number of individuals in the domain.



(a) $N = 32$

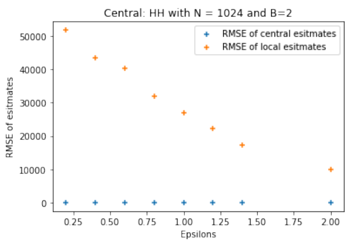


(b) $N = 128$

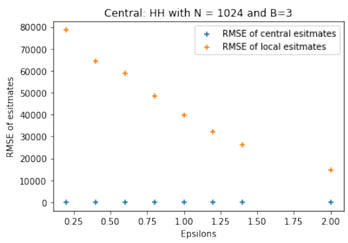


(c) $N = 256$

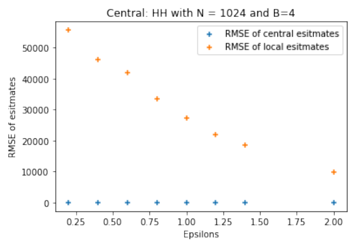
Figure 24: Central flat solution vs local flat solution



(a) $N = 1024$ and $B = 2$



(b) $N = 1024$ and $B = 3$



(c) $N = 1024$ and $B = 4$

Figure 25: Central HH solution vs local HH solution

11 Conclusion

This thesis examined differential privacy, described some of the concepts used in differential privacy, and made an informal and formal definition of it. We proved the composition theorem about differential privacy. We have then described four different data structures that support differential privacy for range counting. Two for both local and central differential privacy.

The two models in local and central differential privacy build upon the same idea; one has a 'flat' approach the other has the idea of imposing a hierarchy of the ranges. After implementing these four data structures, we measured the error they estimation different range queries of varying length and privacy parameters. They were then bench mark the local and central data structures against each other. We found that local had a significantly higher error than the central ones; this agrees with the theory. We saw that the idea of imposing a hierarchy of the ranges would answer range queries more accurately for larger ranges. We found out the in both the flat solutions, the error grows linear with the length of the query, with the caveat the with my implementation of the frequency oracle the error starts to drop of at some point. We found out that in error grows accordingly to $\frac{1}{\epsilon}$ in our central solution utilizing a hierarchy of the ranges. We found out that in error grows accordingly to $\frac{1}{\epsilon^2}$ in our local solution utilizing a hierarchy of the ranges.

12 Future work

Due to lack of time, some tests and experiments were left out. I would have liked to have test the implemented data structures against some know attacks, reconstruction attack in particular, firstly to check if they were differential private, and then see what effect ϵ, would have on the attack. We could also have shown how to break my bad implementation of continuous observation. Another interesting experiment was figuring out when the local flat solution began to decrease in error again. Was it when we meet a certain percentage of the domain size, or a certain percentage of all the individuals have reported. These two variables, percentage of domain size and a certain percentage of all the individuals have reported, are not necessarily dependant on each other, 100% of the individuals could have visited the library on the same day.

13 Bibliography

References

Dwork, Cynthia and Aaron Roth (Aug. 2014). "The Algorithmic Foundations of Differential Privacy". In: *Found. Trends Theor. Comput. Sci.* 9.3–4, pp. 211–407. ISSN: 1551-305X. DOI: 10.1561/0400000042. URL: https://doi.org/10.1561/0400000042.

Kobbi, Nissim et al. (2018). "Differential Privacy:A Primer for a Non-technical Audience*". In: URL: https://privacytools.seas.harvard.edu/files/privacytools/files/pedagogical-document-dp_new.pdf.

Kulkarni, Tejas, Graham Cormode, and Divesh Srivastava (2018). "Answering Range Queries Under Local Differential Privacy". In: *CoRR* abs/1812.10942. arXiv: 1812.10942. URL: http://arxiv.org/abs/1812.10942.

Narayanan, Arvind and Vitaly Shmatikov (2006). "How To Break Anonymity of the Netflix Prize Dataset". In: *CoRR* abs/cs/0610105. arXiv: cs/0610105. URL: http://arxiv.org/abs/cs/0610105.

Sweeney, Latanya (2000). "Simple Demographics Often Identify People Uniquely". Working paper. URL: http://dataprivacylab.org/projects/identifiability/.

14 Appendix

14.1 Implementation

14.1.1 Central flat solution

```python
import numpy as np
import pandas as pd
from scipy.stats import laplace
from datetime import datetime
from datetime import timedelta

class central_flat:
    def __init__(self, epsilon, dates, counts):
        """Setup of the datastructere
        Parameters:
        epsilon (float): The epsilon for differintial privacy
        dates (Array): The dates of the stream
        counts (Array): The count for each of the dates
        Returns:
        A epsilon differintial datastructure
        """

        self.epsilon = epsilon
        self.all_dates = dates
        self.all_counts = counts

        if len(dates) < (dates[-1]-dates[0]).days:
            self.all_dates = self.__add_missing_dates(dates)
            self.all_counts = self.__add_missing_counts(counts,dates)

        #Make dict for date indexing
        values = np.arange(0,len(self.all_dates))
        zip_iterator = zip(self.all_dates, values)
        self.idx_dict =  dict(zip_iterator)

        self.noisy_counts = self.__process(self.all_counts)

    def __add_missing_dates(self, old_dates):
        """Add missing dates in a list
        Parameters:
        old_dates (list of datetime.date): List of dates that is not countious
        Returns:
        List of countious starting with the first value of
        """
        start_date = old_dates[0]
        end_date = old_dates[-1]
        all_dates = pd.date_range(start = start_date, end = end_date).to_pydatetime().tolist()
        return [(date.date()) for date in all_dates]

    def __add_missing_counts(self, old_counts, old_dates):
        """Adds 0 to the list of counts where there was missing dates
        Parameters:
        old_counts (list of int): List counts for each day with
        old_dates (list of datetime.date): List of dates that is not countious
        Returns:
        List of countious starting with the first value of
        """
        zip_iterator = zip(old_dates, old_counts)
        missing_dict =  dict(zip_iterator)
        all_counts = np.zeros(len(self.all_dates))
        for i, date in enumerate(self.all_dates):
            val = missing_dict.get(date, 0)
```

```
                all_counts[i] = val

        return all_counts

    def __process(self, counts):
        N = len(counts)
        laplaces = laplace(scale=1/self.epsilon).rvs(N)
        noise_counts = counts + laplaces

        return noise_counts

    def answer(self, dates):
        """Calculates the differintial private answear

        Parameters:
        dates (tuple of string): Two dates in the format string 2000-12-19.

        Returns:
        float: The private range count
        """
        if (len(dates) < 2):
            #There is only one date
            date_obj_0 = datetime.strptime(dates[0],'%Y-%m-%d').date()

            idx = self.idx_dict[date_obj_0]
            noise_count = self.noisy_counts[idx]
            return noise_count

        else:
            date_obj_0 = datetime.strptime(dates[0],'%Y-%m-%d').date()
            date_obj_1 = datetime.strptime(dates[1],'%Y-%m-%d').date()

            noise_sum = np.sum(self.noisy_counts[self.idx_dict[date_obj_0]: self.idx_dict[date_obj_1]+1])
            return noise_sum

    def real_answer(self, dates):
        """Calculates the real answer to a range query

        Parameters:
        dates (tuple of string): Two dates in the format string 2000-12-19.

        Returns:
        float: The real range count
        """
        if len(dates) < 2:
            date_obj_0 = datetime.strptime(dates[0],'%Y-%m-%d').date()
            return self.all_counts[self.idx_dict[date_obj_0]]
        else:
            date_obj_0 = datetime.strptime(dates[0],'%Y-%m-%d').date()
            date_obj_1 = datetime.strptime(dates[1],'%Y-%m-%d').date()
            sum_ = np.sum(self.all_counts[self.idx_dict[date_obj_0]: self.idx_dict[date_obj_1]+1])
            return sum_
```

14.1.2 Local flat solution

```
import numpy as np
import pandas as pd
from scipy.stats import laplace
from datetime import datetime
from datetime import timedelta


class OLH_flat:
```

```python
    def __init__(self, epsilon, dates, counts):
        """Setup of the datastructere
        Parameters:
        T (int): The length of the stream
        epsilon (float): The height of the full binary tree.
        dates (Array): The dates of the stream
        counts (Array): The count for each of the dates
        Returns:
        A epsilon differintial datastructe
        """

        self.epsilon = epsilon
        self.all_dates = dates
        self.all_counts = counts

        if len(dates) < (dates[-1] - dates[0]).days:
            self.all_dates = self.__add_missing_dates(dates)
            self.all_counts = self.__add_missing_counts(counts, dates)

        # Make dict for date indexing
        values = np.arange(0, len(self.all_dates))
        zip_iterator = zip(self.all_dates, values)
        self.idx_dict = dict(zip_iterator)

        self.noise_counts = self.__process(self.all_dates, self.all_counts)
        # Check if we are we have missing dates.

        self.p = np.exp(self.epsilon) / (np.exp(self.epsilon) + len(self.all_dates) - 1)

    def __add_missing_dates(self, old_dates):
        """Add missing dates in a list
        Parameters:
        old_dates (list of datetime.date): List of dates that is not countious
        Returns:
        List of dates starting with the first value of
        """
        start_date = old_dates[0]
        end_date = old_dates[-1]
        all_dates = pd.date_range(start=start_date, end=end_date).to_pydatetime().tolist()
        return [(date.date()) for date in all_dates]

    def __add_missing_counts(self, old_counts, old_dates):
        """Adds 0 to the list of counts where there was missing dates
        Parameters:
        old_counts (list of int): List counts for each day with
        old_dates (list of datetime.date): List of dates that is not countious
        Returns:
        List of all counts starting with the first value of
        """
        zip_iterator = zip(old_dates, old_counts)
        missing_dict = dict(zip_iterator)
        all_counts = np.zeros(len(self.all_dates))
        for i, date in enumerate(self.all_dates):
            val = missing_dict.get(date, 0)
            all_counts[i] = val

        return all_counts

    def OLH_func(self, x, g):
        if np.random.uniform(0, 1) < np.exp(self.epsilon) / (np.exp(self.epsilon) + g - 1):
            return x
        else:
            return np.random.randint(low=0, high=g)
```

```python

    def OLH_aggre(self, count, N, g):
        p = np.exp(self.epsilon) / (np.exp(self.epsilon) + g - 1)
        return (count - (1 - p) * N / g) / (p)

    def __process(self, dates, counts):
        olh_count = np.zeros(len(counts))
        D = len(dates)

        for idx, count in enumerate(counts):
            for i in range(0, int(count)):
                response = self.OLH_func(idx, D)
                olh_count[response] = olh_count[response] + 1

        return olh_count

    def answer(self, dates):
        """Calculates the path of index in full binary string

        Parameters:
        dates (tuple of string): Two dates in the format string 2000-12-19.

        Returns:
        float: The private range count
        """
        N = np.sum(self.noise_counts)
        D = len(self.noise_counts)
        if len(dates) < 2:
            # There is only one date
            date_obj_0 = datetime.strptime(dates[0], '%Y-%m-%d').date()

            idx = self.idx_dict[date_obj_0]
            noise_count = self.noise_counts[idx]
            return self.OLH_aggre(noise_count, N, D)

        else:
            date_obj_0 = datetime.strptime(dates[0], '%Y-%m-%d').date()
            date_obj_1 = datetime.strptime(dates[1], '%Y-%m-%d').date()
            idx_0 = self.idx_dict[date_obj_0]
            idx_1 = self.idx_dict[date_obj_1]
            # print(idx_0)
            # print(idx_1)
            # idx_0 is not 0
            noise_sum = 0.0
            for i in range(idx_0, idx_1 + 1):
                # print(i)
                # print(self.OLH_answer(self.noise_counts[i], N, D))
                noise_sum = noise_sum + self.OLH_aggre(self.noise_counts[i], N, D)
            return noise_sum

    def real_answer(self, dates):
        if len(dates) < 2:
            date_obj_0 = datetime.strptime(dates[0], '%Y-%m-%d').date()
            return self.all_counts[self.idx_dict[date_obj_0]]
        else:
            date_obj_0 = datetime.strptime(dates[0], '%Y-%m-%d').date()
            date_obj_1 = datetime.strptime(dates[1], '%Y-%m-%d').date()
            sum_ = np.sum(self.all_counts[self.idx_dict[date_obj_0]: self.idx_dict[date_obj_1] + 1])
            return sum_
```

14.1.3 Continuous Observation

```python
import numpy as np
import pandas as pd
from scipy.stats import laplace
from datetime import datetime

class con_obs:

    def __init__(self, epsilon, degree, dates, counts):
        """Setup of the datastructere

        Parameters:
        T (int): The lenght of the stream
        epsilon (float): The height of the full binary tree.
        dates (Array): The dates of the stream
        counts (Array): The count for each of the dates
        Returns:
        A epsilon differintial datastructe
        """

        self.degree = degree

        self.all_dates = dates
        self.all_counts = counts
        #Check if we are we have missing dates.
        if len(dates) < (dates[-1]-dates[0]).days:
            self.all_dates = self.__add_missing_dates(self.all_dates)
            self.all_counts = self.__add_missing_counts(self.all_counts,dates)

        self.all_dates = self.pad_dates(self.all_dates)
        self.all_counts = self.pad_counts(self.all_counts)

        #Make dict for date indexing
        values = np.arange(0,len(self.all_dates))
        zip_iterator = zip(self.all_dates, values)
        self.idx_dict =  dict(zip_iterator)

        # We need the stream to be a power of the degree
        self.T = int(np.ceil(np.log(len(self.all_counts)) / np.log(self.degree))+1)
        self.epsilon = epsilon
        self.zeta = (np.log2(self.T))/epsilon

        # The height of the "adic tree"
        self.n_layers = int(np.log(self.T)/np.log(self.degree))
        self.h = int(np.ceil(np.log(len(self.all_dates)) / np.log(degree)))

        # Get laplace for each node
        self.laplaces = self.init_laplace()
        self.histogram = self.build_histogram()
        self.tree_levels = self.__process(self.all_counts)

    def init_laplace(self):
        """
        returns: list of arrays with the correct size of laplaces variabels.
        """
        laplaces = []
        for i in np.arange(0,self.T):
            rvs = laplace(scale=self.zeta).rvs(int(self.degree**np.ceil(i)))
            laplaces.append(rvs)

        for i in np.arange(0,self.T-1):
            for j in np.arange(0,len(laplaces[i])):

```

```
                ch1, ch2 = self.get_children(j,i)
                laplaces[i+1][ch1] = laplaces[i+1][ch1] + laplaces[i][j]
                laplaces[i+1][ch1] = laplaces[i+1][ch2] + laplaces[i][j]

        return laplaces

    def build_histogram(self):
        #print(counts)
        #print(get_group(counts,degree))
        tree = []
        left = self.all_counts
        for level in np.arange(0,self.T):
            split_ratio = self.degree**level
            left = np.array_split(self.all_counts, split_ratio)

            sums = [np.sum(a) for a in left]
            tree.append(sums)

        tree.append(self.all_counts)
        return tree

    def __add_missing_dates(self, old_dates):
        """Add missing dates in a list
        Parameters:
        old_dates (list of datetime.date): List of dates that is not countious
        Returns:
        List of countious starting with the first value of
        """
        start_date = old_dates[0]
        end_date = old_dates[-1]
        all_dates = pd.date_range(start = start_date, end = end_date).to_pydatetime().tolist()
        return [(date.date()) for date in all_dates]

    def __add_missing_counts(self, old_counts, old_dates):
        """Adds 0 to the list of counts where there was missing dates
        Parameters:
        old_counts (list of int): List counts for each day with
        old_dates (list of datetime.date): List of dates that is not countious
        Returns:
        List of countious starting with the first value of
        """
        zip_iterator = zip(old_dates, old_counts)
        missing_dict =  dict(zip_iterator)
        all_counts = np.zeros(len(self.all_dates))
        for i, date in enumerate(self.all_dates):
            val = missing_dict.get(date, 0)
            all_counts[i] = val

        return all_counts

    def pad_counts(self, counts):
        levels = int(np.ceil(np.log(len(counts)) / np.log(self.degree)))

        n_missing_counts =  self.degree**levels -len(counts)

        missing = np.zeros(n_missing_counts, dtype=int)
        new_counts = np.concatenate((counts,missing))
        return new_counts

    def pad_dates(self, dates):
        levels = int(np.ceil(np.log(len(dates)) / np.log(self.degree)))
        n_missing_dates =  self.degree**levels - len(dates)

```

```
        start_date = datetime.strptime(str(dates[-1]),'%Y-%m-%d').date()
        result = pd.date_range(start = start_date, periods = n_missing_dates).to_pydatetime().tolist()

        new_dates = np.concatenate((dates,result))
        return new_dates

    def __process(self, counts):
        """
        def __process(self, counts):

            noise_counts = np.zeros(len(self.dates))
            for idx, date_count in enumerate(counts):
                indices = self.get_index(idx,self.n_layers)
                indices.reverse()
                laplace_sum = 0.0
                for laplace_idx, laplace_row in enumerate(self.laplaces):
                    laplace_sum = laplace_sum + laplace_row[indices[laplace_idx]]
                noise_counts[idx] = date_count  +  noise_counts[idx-1] + laplace_sum
            return noise_counts

        """
        hh = []
        for i in range(0,len(self.laplaces)):

            level = self.laplaces[i] + self.histogram[i]
            hh.append(level)
        return hh

    def get_index(self, date_idx, n_layers):
        """Calculates the path of index in full binary string

        Parameters:
        date_idx (int): The node in the bouttom layer we want to calculate a path to.
        The bottom layer has index from 0 to 2**h-1
        n_layers (int): The height of the full binary tree.

        Returns:
        list: of index in the path from the starting from the bottom and going up

        """
        idx = []
        for i in np.arange(0,self.h):
            if i == 0:
                idx.append(int(date_idx))
            else:
                idx.append(int(idx[i-1]//self.degree))
        idx.append(0)
        return idx

    def get_children(self, idx, level):
        """Calculates the path of index in full binary string

        Parameters:
        date_idx (int): The node in the bouttom layer we want to calculate a path to.
        The bottom layer has index from 0 to 2**h-1
        n_layers (int): The height of the full binary tree. 0 index

        Returns:
        list: of index in the path from the starting from the bottom and going up

        """
        child_1 = idx*self.degree
        child_2 = idx*self.degree + 1
```

```

        return child_1, child_2

    def get_group(self, idx, level):
        """Calculates the path of index in full binary string

        Parameters:
        date_idx (int): The node in the bouttom layer we want to calculate a path to.
        The bottom layer has index from 0 to 2**h-1
        n_layers (int): The height of the full binary tree. 0 index

        Returns:
        list: of index in the path from the starting from the bottom and going up

        """
        if level == 0:
            return id
        elif idx == 0:
            return np.arange(0,self.degree)
        else:
            group_index = idx //self.degree
            level_indicis = np.arange(0,self.degree**level)

            split_ratio = (len(level_indicis) // self.degree)
            level_indicis_split = np.array_split(level_indicis, split_ratio)

            return level_indicis_split[group_index]

    def turns_right(self, path):
        #0 is left 1 is right
        direction_lst = []
        for i in range(len(path)-1):
            #print(f'i = {i}')
            current = path[i]
            nxt = path[i+1]

            if nxt == 0:
                #We went left
                direction_lst.append(0)

            elif nxt == current*self.degree + self.degree - 1:
                #We went right
                direction_lst.append(1)

            else:
                direction_lst.append(0)

        return direction_lst


    def turns_left(self, path):
        #0 is left 1 is right
        direction_lst = []
        for i in range(len(path)-1):
            #print(f'i = {i}')
            current = path[i]
            nxt = path[i+1]

            #Checks
            if nxt == 0:
                #We went left
                direction_lst.append(1)
            #Checks
```

```
            elif current == 0 and current < nxt:
                #We went right
                direction_lst.append(0)
            elif nxt == self.degree * current:
                #We went left
                direction_lst.append(1)
            else:
                #We went right
                direction_lst.append(0)

        return direction_lst

    def answer(self, dates):
        """Calculates the path of index in full binary string

        Parameters:
        dates (tuple of string): Two dates in the format string 2000-12-19.

        Returns:
        float: The private range count
        """

        date_obj_0 = datetime.strptime(dates[0],'%Y-%m-%d').date()
        date_obj_1 = datetime.strptime(dates[1],'%Y-%m-%d').date()

        idx_0 = self.idx_dict[date_obj_0]
        idx_1 = self.idx_dict[date_obj_1]

        idx_left = idx_0-1
        idx_right = idx_1+1

        path_to_left = np.flip(np.array(self.get_index(idx_left,self.h+1)))
        path_to_right = np.flip(np.array(self.get_index(idx_right,self.h+1)))

        turns_left_lst = self.turns_left(path_to_right)
        turns_right_lst = self.turns_right(path_to_left)

        range_count = 0.0

        if idx_0 == 0 and idx_1 == np.max(np.fromiter(self.idx_dict.values(), dtype = int)):
            node = self.tree_levels[0]
            range_count = node

        elif idx_0 == 0:

            level_offset = 1

            for i in range(len(turns_left_lst)):

                if turns_left_lst[i] == 0:
                    group = self.get_group(path_to_right[i+level_offset], i+level_offset)
                    idx_sss = np.where(group == path_to_right[i+level_offset])[0][0]

                    count_nodes = self.tree_levels[i+level_offset][group[:idx_sss]]

                    for node in count_nodes:
                        range_count = range_count + node

        elif idx_1 == np.max(np.fromiter(self.idx_dict.values(), dtype = int)):

            level_offset = 1

            for i in range(len(turns_right_lst)):
```

```
                if turns_right_lst[i] == 0:

                    group = self.get_group(path_to_left[i+level_offset], i+level_offset)
                    idx_sss = np.where(group == path_to_left[i+level_offset])[0][0]

                    count_nodes = self.tree_levels[i+level_offset][group[idx_sss+1:]]
                    for node in count_nodes:
                        range_count = range_count + node

        else:

            level_offset = 1
            left_count = []
            left_count_group = []

            level_offset = 1
            left_count = []
            left_count_group = []

            for i in range(len(turns_left_lst)):
                if turns_left_lst[i] == 0:
                    group = self.get_group(path_to_right[i+level_offset], i+level_offset)
                    idx_sss = np.where(group == path_to_right[i+level_offset])[0][0]

                    left_count_group.append(group[:idx_sss])

                    count_nodes = self.tree_levels[i+level_offset][group[:idx_sss]]
                    left_count.append(count_nodes)

                else:
                    left_count_group.append(np.array([]))
                    left_count.append(np.array([]))

            #The search right side
            right_count = []
            right_count_group = []

            for i in range(len(turns_right_lst)):
                if turns_right_lst[i] == 0:

                    group = self.get_group(path_to_left[i+level_offset], i+level_offset)
                    idx_sss = np.where(group == path_to_left[i+level_offset])[0][0]

                    right_count_group.append(group[idx_sss+1:])

                    count_nodes = self.tree_levels[i+level_offset][group[idx_sss+1:]]
                    right_count.append(count_nodes)

                else:
                    right_count_group.append(np.array([]))
                    right_count.append(np.array([]))

            for i in range(len(left_count_group)):
                if left_count_group[i].size != 0 and right_count_group[i].size != 0:
                    #Both not zero
                    group_left = self.get_group(left_count_group[i][0], i+ level_offset)
                    group_right = self.get_group(right_count_group[i][0], i+ level_offset)

                    if not (np.array_equal(group_left,group_right)):
                        for node in left_count_group[i]:
                            range_count = range_count + self.tree_levels[i+level_offset][node]

                        for node in right_count_group[i]:
```

```python
                            range_count = range_count + self.tree_levels[i+level_offset][node]

                    else:
                        count_nodes = np.intersect1d(left_count_group[i], right_count_group[i])
                        for node in count_nodes:
                            range_count = range_count + self.tree_levels[i+level_offset][node]

                if left_count_group[i].size != 0 and right_count_group[i].size == 0:
                    #Left not zero
                    for node in left_count_group[i]:
                        if path_to_left[i] != path_to_right[i]:
                            range_count = range_count + self.tree_levels[i+level_offset][node]

                if right_count_group[i].size != 0 and left_count_group[i].size == 0:
                    #Right not zero
                    for node in right_count_group[i]:
                        if path_to_left[i] != path_to_right[i]:
                            range_count = range_count + self.tree_levels[i+level_offset][node]
        return range_count

    def real_answer(self, dates):
        if len(dates) < 2:
            date_obj_0 = datetime.strptime(dates[0],'%Y-%m-%d').date()
            return self.all_counts[self.idx_dict[date_obj_0]]
        else:
            date_obj_0 = datetime.strptime(dates[0],'%Y-%m-%d').date()
            date_obj_1 = datetime.strptime(dates[1],'%Y-%m-%d').date()
            sum_ = np.sum(self.all_counts[self.idx_dict[date_obj_0]: self.idx_dict[date_obj_1]+1])
            return sum_

```

14.1.4 Local Hierarchical Histograms

```python
import numpy as np
import pandas as pd

from datetime import datetime

class HH_OLH:
    def __init__(self, epsilon, degree, dates, counts):
        """Setup of the datastructere
        Parameters:
        T (int): The lenght of the stream
        epsilon (float): The height of the full binary tree.
        dates (Array): The dates of the stream
        counts (Array): The count for each of the dates
        Returns:
        A epsilon differintial datastructe
        """
        self.epsilon = epsilon
        self.all_dates = dates
        self.all_counts = counts
        #Check if we are we have missing dates.
        if len(dates) < (dates[-1]-dates[0]).days:
            #print('here')
            self.all_dates = self.__add_missing_dates(dates)
            self.all_counts = self.__add_missing_counts(counts,dates)

        #Make dict for date indexing
        values = np.arange(0,len(self.all_dates))
        zip_iterator = zip(self.all_dates, values)
        self.idx_dict =  dict(zip_iterator)

```

```
        self.degree = degree
        self.h = int(np.ceil(np.log(len(self.all_dates)) / np.log(degree)))
        self.level_prob = np.full(self.h+1,1/(self.h+1))

        self.tree_levels = self.__process(self.all_dates, self.all_counts)

    def __add_missing_dates(self, old_dates):
        """Add missing dates in a list
        Parameters:
        old_dates (list of datetime.date): List of dates that is not countious
        Returns:
        List of countious starting with the first value of
        """
        start_date = old_dates[0]
        end_date = old_dates[-1]
        all_dates = pd.date_range(start = start_date, end = end_date).to_pydatetime().tolist()
        return [(date.date()) for date in all_dates]

    def __add_missing_counts(self, old_counts, old_dates):
        """Adds 0 to the list of counts where there was missing dates
        Parameters:
        old_counts (list of int): List counts for each day with
        old_dates (list of datetime.date): List of dates that is not countious
        Returns:
        List of countious starting with the first value of
        """
        zip_iterator = zip(old_dates, old_counts)
        missing_dict =  dict(zip_iterator)
        all_counts = np.zeros(len(self.all_dates))
        for i, date in enumerate(self.all_dates):
            val = missing_dict.get(date, 0)
            all_counts[i] = val

        return all_counts

    def __process(self, dates, counts):
        tree_levels = []
        for i in np.arange(0,self.h+1):
            level = np.zeros(int(self.degree**np.ceil(i)))
            tree_levels.append(level)

        for index, (date, day_count) in enumerate(zip(dates, counts)):
            idxs = self.get_index(index,self.h)
            idxs.reverse()
            for person in range(int(day_count)):
                level = np.random.choice(np.arange(0, self.h+1), p = self.level_prob )

                if level != 0:
                    response = self.OLH_func(idxs[level], (self.degree**level))
                else:
                    response = 0
                tree_levels[level][response] = tree_levels[level][response] + 1

        return tree_levels

    def get_index(self, date_idx, n_layers):
        """Calculates the path of index in full binary string

        Parameters:
        date_idx (int): The node in the bouttom layer we want to calculate a path to.
        The bottom layer has index from 0 to 2**h-1
        n_layers (int): The height of the full binary tree.

```

```
        Returns:
        list: of index in the path from the starting from the bottom and going up

        """
        idx = []
        for i in np.arange(0,self.h):
            if i == 0:
                idx.append(int(date_idx))
            else:
                idx.append(int(idx[i-1]//self.degree))
        idx.append(0)
        return idx

    def get_group(self, idx, level):
        """Calculates the path of index in full binary string

        Parameters:
        date_idx (int): The node in the bouttom layer we want to calculate a path to.
        The bottom layer has index from 0 to 2**h-1
        n_layers (int): The height of the full binary tree. 0 index

        Returns:
        list: of index in the path from the starting from the bottom and going up

        """
        if level == 0:
            return id
        elif idx == 0:
            return np.arange(0,self.degree)
        else:
            group_index = idx //self.degree
            level_indicis = np.arange(0,self.degree**level)

            split_ratio = (len(level_indicis) // self.degree)
            level_indicis_split = np.array_split(level_indicis, split_ratio)

            return level_indicis_split[group_index]

    def OLH_func(self, x, g):
        if np.random.uniform(0,1) < np.exp(self.epsilon)/(np.exp(self.epsilon)+g-1):
            return x
        else:
            return np.random.randint(low = 0, high = g)

    def OLH_aggre(self, count, N, g):
        p = np.exp(self.epsilon)/(np.exp(self.epsilon)+g-1)
        #print(p - 1/g)
        #print(f'p = {p}')
        return (count - (1-p)*N/g) / (p)

    def turns_right(self, path):
        #0 is left 1 is right
        direction_lst = []
        for i in range(len(path)-1):
            #print(f'i = {i}')
            current = path[i]
            nxt = path[i+1]

            if nxt == 0:
                #We went left
                direction_lst.append(0)

            elif nxt == current*self.degree + self.degree - 1:
```

```
                #We went right
                direction_lst.append(1)

            else:
                direction_lst.append(0)

        return direction_lst


    def turns_left(self, path):
        #0 is left 1 is right
        direction_lst = []
        for i in range(len(path)-1):
            #print(f'i = {i}')
            current = path[i]
            nxt = path[i+1]

            #Checks
            if nxt == 0:
                #We went left
                direction_lst.append(1)
            #Checks
            elif current == 0 and current < nxt:
                #We went right
                direction_lst.append(0)
            elif nxt == self.degree * current:
                #We went left
                direction_lst.append(1)
            else:
                #We went right
                direction_lst.append(0)

        return direction_lst

    def answer(self, dates):
        """Calculates the path of index in full binary string

        Parameters:
        dates (tuple of string): Two dates in the format string 2000-12-19.

        Returns:
        float: The private range count
        """

        date_obj_0 = datetime.strptime(dates[0],'%Y-%m-%d').date()
        date_obj_1 = datetime.strptime(dates[1],'%Y-%m-%d').date()


        idx_0 = self.idx_dict[date_obj_0]
        idx_1 = self.idx_dict[date_obj_1]


        idx_left = idx_0-1
        idx_right = idx_1+1

        path_to_left = np.flip(np.array(self.get_index(idx_left,self.h+1)))
        path_to_right = np.flip(np.array(self.get_index(idx_right,self.h+1)))

        turns_left_lst = self.turns_left(path_to_right)
        turns_right_lst = self.turns_right(path_to_left)

        range_count = 0.0

```

```
        if idx_0 == 0 and idx_1 == np.max(np.fromiter(self.idx_dict.values(), dtype = int)):
            node = self.tree_levels[0]
            range_count = self.OLH_aggre(node, np.sum(self.tree_levels[0]), 1)

        elif idx_0 == 0:

            level_offset = 1

            for i in range(len(turns_left_lst)):

                if turns_left_lst[i] == 0:
                    group = self.get_group(path_to_right[i+level_offset], i+level_offset)
                    idx_sss = np.where(group == path_to_right[i+level_offset])[0][0]

                    count_nodes = self.tree_levels[i+level_offset][group[:idx_sss]]

                    for node in count_nodes:
                        range_count = range_count + self.OLH_aggre(node, np.sum(self.tree_levels[i+level_offse

        elif idx_1 == np.max(np.fromiter(self.idx_dict.values(), dtype = int)):

            level_offset = 1

            for i in range(len(turns_right_lst)):
                if turns_right_lst[i] == 0:

                    group = self.get_group(path_to_left[i+level_offset], i+level_offset)
                    idx_sss = np.where(group == path_to_left[i+level_offset])[0][0]

                    count_nodes = self.tree_levels[i+level_offset][group[idx_sss+1:]]
                    for node in count_nodes:
                        range_count = range_count + self.OLH_aggre(node, np.sum(self.tree_levels[i+level_offse


        else:

            level_offset = 1
            left_count = []
            left_count_group = []

            for i in range(len(turns_left_lst)):
                if turns_left_lst[i] == 0:
                    group = self.get_group(path_to_right[i+level_offset], i+level_offset)
                    idx_sss = np.where(group == path_to_right[i+level_offset])[0][0]

                    left_count_group.append(group[:idx_sss])

                    count_nodes = self.tree_levels[i+level_offset][group[:idx_sss]]
                    left_count.append(count_nodes)

                else:
                    left_count_group.append(np.array([]))
                    left_count.append(np.array([]))

            #The search right side
            right_count = []
            right_count_group = []

            for i in range(len(turns_right_lst)):
                if turns_right_lst[i] == 0:

                    group = self.get_group(path_to_left[i+level_offset], i+level_offset)
                    idx_sss = np.where(group == path_to_left[i+level_offset])[0][0]
```

```python

                    right_count_group.append(group[idx_sss+1:])

                    count_nodes = self.tree_levels[i+level_offset][group[idx_sss+1:]]
                    right_count.append(count_nodes)

                else:
                    right_count_group.append(np.array([]))
                    right_count.append(np.array([]))

            for i in range(len(left_count_group)):

                if left_count_group[i].size != 0 and right_count_group[i].size != 0:
                    #Both not zero
                    group_left = self.get_group(left_count_group[i][0], i+ level_offset)
                    group_right = self.get_group(right_count_group[i][0], i+ level_offset)

                    if not (np.array_equal(group_left,group_right)):
                        for node in left_count_group[i]:
                            range_count = range_count + self.OLH_aggre(self.tree_levels[i+level_offset][node],

                        for node in right_count_group[i]:
                            range_count = range_count + self.OLH_aggre(self.tree_levels[i+level_offset][node],

                    else:
                        count_nodes = np.intersect1d(left_count_group[i], right_count_group[i])
                        for node in count_nodes:
                            range_count = range_count + self.OLH_aggre(self.tree_levels[i+level_offset][node],

                if left_count_group[i].size != 0 and right_count_group[i].size == 0:
                    #Left not zero
                    for node in left_count_group[i]:
                        if path_to_left[i] != path_to_right[i]:
                            range_count = range_count + self.OLH_aggre(self.tree_levels[i+level_offset][node],

                if right_count_group[i].size != 0 and left_count_group[i].size == 0:
                    #Right not zero
                    for node in right_count_group[i]:
                        if path_to_left[i] != path_to_right[i]:
                            range_count = range_count + self.OLH_aggre(self.tree_levels[i+level_offset][node],

        return range_count * (self.h+1)

    def real_answer(self, dates):
        if len(dates) < 2:
            date_obj_0 = datetime.strptime(dates[0],'%Y-%m-%d').date()
            return self.all_counts[self.idx_dict[date_obj_0]]
        else:
            date_obj_0 = datetime.strptime(dates[0],'%Y-%m-%d').date()
            date_obj_1 = datetime.strptime(dates[1],'%Y-%m-%d').date()
            sum_ = np.sum(self.all_counts[self.idx_dict[date_obj_0]: self.idx_dict[date_obj_1]+1])
            return sum_
```

14.2 Unit test

14.2.1 Continuous Observation

```python
import unittest

import numpy as np

from psql_functions import execQuery, execRangeQuery
from miss_data import add_missing_dates, add_missing_counts
```

```python
from sample_range_query import load_range_queries_n_split

param_dic = {
    "host"      : "localhost",
    "database"  : "bachelorBesoeg2014",
    "user"      : "postgres",
    "password"  : "password",
    "port"      : "5432"
}

query = """select time_ from _775147;"""
result = execQuery(param_dic, query)
dates = [(date[0]) for date in result]

query = """select count_ from _775147;"""
result = execQuery(param_dic, query)

counts = [(count[0]) for count in result]

data_dates = add_missing_dates(dates)
data_counts =  add_missing_counts(counts, dates, data_dates)

from con_obs import con_obs

class test_con_obs(unittest.TestCase):

    def test_height(self):
        model = con_obs(1.0, 2, data_dates[:32], data_counts[:32])
        self.assertEqual(model.h, 5)

        model = con_obs(1.0, 3, data_dates[:27], data_counts[:27])
        self.assertEqual(model.h, 3)

        model = con_obs(1.0, 4, data_dates[:64], data_counts[:64])
        self.assertEqual(model.h, 3)

    def test_padding(self):
            model = con_obs(1.0, 2, data_dates[:28], data_counts[:28])
            self.assertEqual(len(model.all_counts), 32)

            model = con_obs(1.0, 3, data_dates[:20], data_counts[:20])
            self.assertEqual(len(model.all_counts), 27)

            model = con_obs(1.0, 4, data_dates[:64], data_counts[:64])
            self.assertEqual(model.h, 3)

    def test_tree_lengths(self):
            model = con_obs(1.0, 2, data_dates[:28], data_counts[:28])
            self.assertEqual(len(model.tree_levels[0]), 1)
            self.assertEqual(len(model.tree_levels[1]), 2)
            self.assertEqual(len(model.tree_levels[2]), 4)
            self.assertEqual(len(model.tree_levels[3]), 8)
            self.assertEqual(len(model.tree_levels[4]), 16)
            self.assertEqual(len(model.tree_levels[5]), 32)

            model = con_obs(1.0, 3, data_dates[:28], data_counts[:28])
            self.assertEqual(len(model.tree_levels[0]), 1)
            self.assertEqual(len(model.tree_levels[1]), 3)
            self.assertEqual(len(model.tree_levels[2]), 9)
            self.assertEqual(len(model.tree_levels[3]), 27)

            model = con_obs(1.0, 4, data_dates[:60], data_counts[:60])
            self.assertEqual(len(model.tree_levels[0]), 1)
```

```python
        self.assertEqual(len(model.tree_levels[1]), 4)
        self.assertEqual(len(model.tree_levels[2]), 16)
        self.assertEqual(len(model.tree_levels[3]), 64)

    def test_histogram(self):
        model = con_obs(1.0, 2, data_dates[:32], np.ones(32))
        self.assertEqual(sum(model.histogram[0]), 32)
        self.assertEqual(sum(model.histogram[1]), 32)
        self.assertEqual(sum(model.histogram[2]), 32)
        self.assertEqual(sum(model.histogram[3]), 32)
        self.assertEqual(sum(model.histogram[4]), 32)
        self.assertEqual(sum(model.histogram[5]), 32)

        self.assertTrue((model.histogram[0] == [32]))
        self.assertTrue((model.histogram[1] == [16,16]))
        self.assertTrue((model.histogram[2] == [8,8,8,8]))
        self.assertTrue((model.histogram[3] == [4,4,4,4,4,4,4,4]))
        self.assertTrue((model.histogram[4] == [2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2]))
        self.assertTrue((model.histogram[5] == [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,


        model = con_obs(1.0, 3, data_dates[:28], data_counts[:28])
        self.assertEqual(len(model.tree_levels[0]), 1)
        self.assertEqual(len(model.tree_levels[1]), 3)
        self.assertEqual(len(model.tree_levels[2]), 9)
        self.assertEqual(len(model.tree_levels[3]), 27)

        model = con_obs(1.0, 4, data_dates[:60], data_counts[:60])
        self.assertEqual(len(model.tree_levels[0]), 1)
        self.assertEqual(len(model.tree_levels[1]), 4)
        self.assertEqual(len(model.tree_levels[2]), 16)
        self.assertEqual(len(model.tree_levels[3]), 64)


    def test_get_index(self):
        model = con_obs(1.0, 2, data_dates[:28], data_counts[:28])
        self.assertEqual(model.get_index(0,4), [0,0,0,0,0,0])
        self.assertEqual(model.get_index(31,4), [31,15,7,3,1,0])

        #with self.assertRaises(IndexError): model.get_index(44,4)

        model = con_obs(1.0, 3, data_dates[:20], data_counts[:20])
        self.assertEqual(model.get_index(0,4), [0,0,0,0])
        self.assertEqual(model.get_index(20,4), [20,6,2,0])

        model = con_obs(1.0, 4, data_dates[:64], data_counts[:64])
        self.assertEqual(model.get_index(0,3), [0,0,0,0])
        self.assertEqual(model.get_index(63,4), [63,15,3,0])

    def test_turns_left(self):
        model = con_obs(1.0, 2, data_dates[:28], data_counts[:28])
        self.assertEqual(model.turns_left([0,0,0,0,0,0]), [1,1,1,1,1])
        self.assertEqual(model.turns_left([31,15,7,3,1,0]), [0,0,0,0,1])

    def test_turns_right(self):
        model = con_obs(1.0, 2, data_dates[:28], data_counts[:28])
        self.assertEqual(model.turns_right([0,0,0,0,0,0]), [0,0,0,0,0])
        self.assertEqual(model.turns_right([31,15,7,3,1,0]), [0,0,0,0,0])

    def test_get_group(self):
        model = con_obs(1.0, 2, data_dates[:28], data_counts[:28])
        self.assertEqual(model.get_group(0,0), 0)
        self.assertEqual(model.get_group(1,0), 1)
```

```
            self.assertTrue((model.get_group(0,5) == [0,1]).all())

            model = con_obs(1.0, 4, data_dates[:64], data_counts[:64])
            self.assertEqual(model.get_group(0,0), 0)
            self.assertEqual(model.get_group(1,0), 1)

            self.assertTrue((model.get_group(1,1) == [0,1,2,3]).all())


if __name__ == '__main__':
    unittest.main()
```

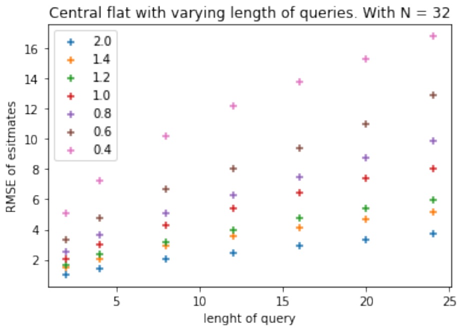
14.2.2 Local Hierarchical Histograms

```
import unittest

from psql_functions import execQuery, execRangeQuery
from miss_data import add_missing_dates, add_missing_counts
from sample_range_query import load_range_queries_n_split

param_dic = {
    "host"      : "localhost",
    "database"  : "bachelorBesoeg2014",
    "user"      : "postgres",
    "password"  : "password",
    "port"      : "5432"
}

query = """select time_ from _775147;"""
result = execQuery(param_dic, query)
dates = [(date[0]) for date in result]

query = """select count_ from _775147;"""
result = execQuery(param_dic, query)

counts = [(count[0]) for count in result]

data_dates = add_missing_dates(dates)
data_counts =  add_missing_counts(counts, dates, data_dates)

from local_hh import HH_OLH

class test_con_obs(unittest.TestCase):

    def test_height(self):
        model = HH_OLH(1.0, 2, data_dates[:32], data_counts[:32])
        self.assertEqual(model.h, 5)

        model = HH_OLH(1.0, 3, data_dates[:27], data_counts[:27])
        self.assertEqual(model.h, 3)

        model = HH_OLH(1.0, 4, data_dates[:64], data_counts[:64])
        self.assertEqual(model.h, 3)

    def test_tree_lengths(self):
            model = HH_OLH(1.0, 2, data_dates[:28], data_counts[:28])
            self.assertEqual(len(model.tree_levels[0]), 1)
            self.assertEqual(len(model.tree_levels[1]), 2)
            self.assertEqual(len(model.tree_levels[2]), 4)
            self.assertEqual(len(model.tree_levels[3]), 8)
            self.assertEqual(len(model.tree_levels[4]), 16)
            self.assertEqual(len(model.tree_levels[5]), 32)
```

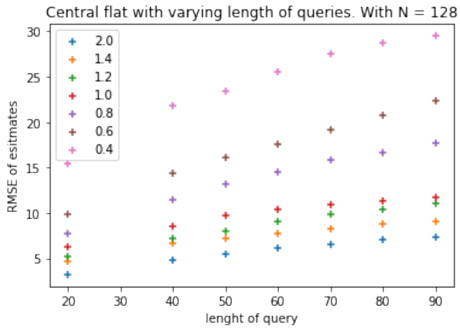
```
            model = HH_OLH(1.0, 3, data_dates[:28], data_counts[:28])
            self.assertEqual(len(model.tree_levels[0]), 1)
            self.assertEqual(len(model.tree_levels[1]), 3)
            self.assertEqual(len(model.tree_levels[2]), 9)
            self.assertEqual(len(model.tree_levels[3]), 27)

            model = HH_OLH(1.0, 4, data_dates[:64], data_counts[:64])
            self.assertEqual(len(model.tree_levels[0]), 1)
            self.assertEqual(len(model.tree_levels[1]), 4)
            self.assertEqual(len(model.tree_levels[2]), 16)
            self.assertEqual(len(model.tree_levels[3]), 64)



    def test_get_index(self):
            model = HH_OLH(1.0, 2, data_dates[:28], data_counts[:28])
            self.assertEqual(model.get_index(0,4), [0,0,0,0,0,0])
            self.assertEqual(model.get_index(31,4), [31,15,7,3,1,0])

            #with self.assertRaises(IndexError): model.get_index(44,4)

            model = HH_OLH(1.0, 3, data_dates[:20], data_counts[:20])
            self.assertEqual(model.get_index(0,4), [0,0,0,0])
            self.assertEqual(model.get_index(20,4), [20,6,2,0])

            model = HH_OLH(1.0, 4, data_dates[:64], data_counts[:64])
            self.assertEqual(model.get_index(0,3), [0,0,0,0])
            self.assertEqual(model.get_index(63,4), [63,15,3,0])

    def test_turns_left(self):
            model = HH_OLH(1.0, 2, data_dates[:28], data_counts[:28])
            self.assertEqual(model.turns_left([0,0,0,0,0,0]), [1,1,1,1,1])
            self.assertEqual(model.turns_left([31,15,7,3,1,0]), [0,0,0,0,1])

    def test_turns_right(self):
            model = HH_OLH(1.0, 2, data_dates[:28], data_counts[:28])
            self.assertEqual(model.turns_right([0,0,0,0,0,0]), [0,0,0,0,0])
            self.assertEqual(model.turns_right([31,15,7,3,1,0]), [0,0,0,0,0])

    def test_get_group(self):
            model = HH_OLH(1.0, 2, data_dates[:28], data_counts[:28])
            self.assertEqual(model.get_group(0,0), 0)
            self.assertEqual(model.get_group(1,0), 1)

    def test_sum_tuple(self):
        self.assertEqual(sum((2, 2, 2)), 6, "Should be 6")

if __name__ == '__main__':
    unittest.main()
```

14.3 Benchmark results

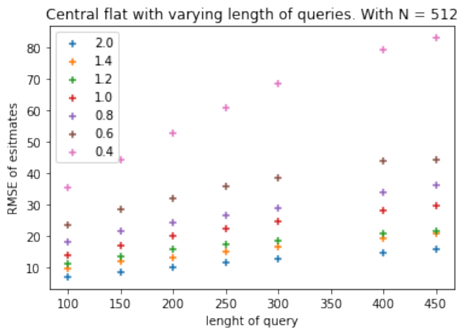
14.3.1 Central flat plots with varying r



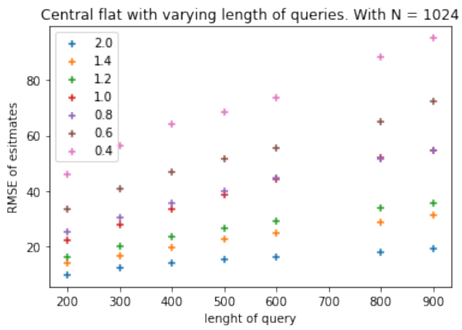
(a) RMSE as function of r for $N = 32$



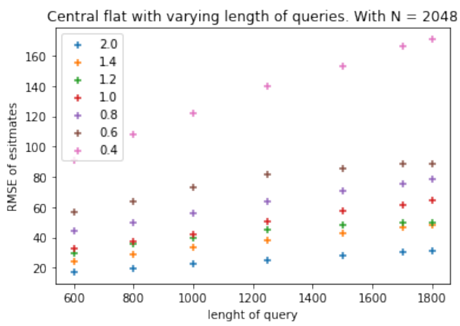
(b) RMSE as function of r for $N = 128$



(c) RMSE as function of r for $N = 512$



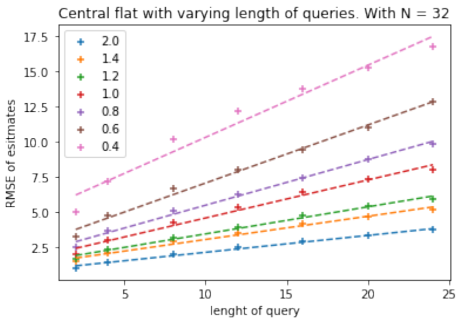
(d) RMSE as function of r for $N = 1024$



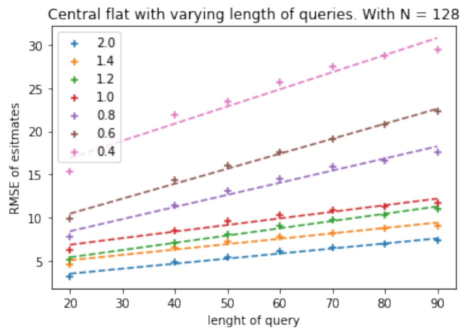
(e) RMSE as function of r for $N = 2028$

(f) RMSE as function of r and N

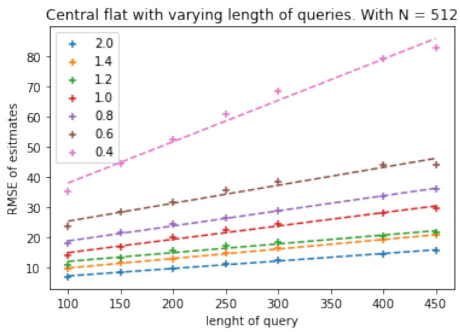
Figure 26: RMSE as function of r and N



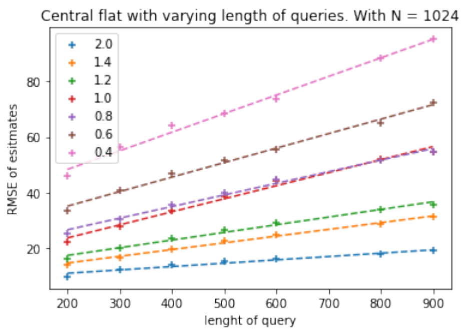
(a) RMSE as function of r for $N = 32$



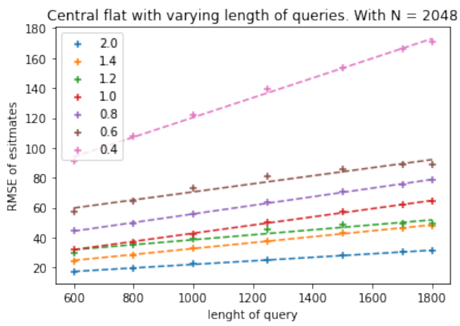
(b) RMSE as function of r for $N = 128$



(c) RMSE as function of r for $N = 512$



(d) RMSE as function of r for $N = 1024$

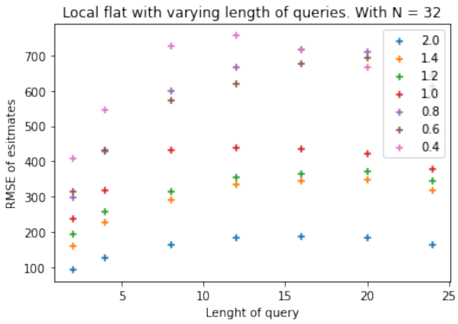


(e) RMSE as function of r for $N = 2028$

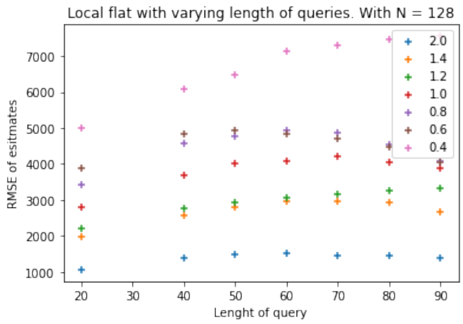
(f) RMSE as function of r and N

Figure 27: RMSE as function of r and N

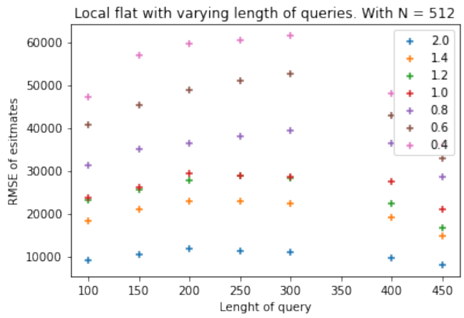
14.3.2 Local flat plots



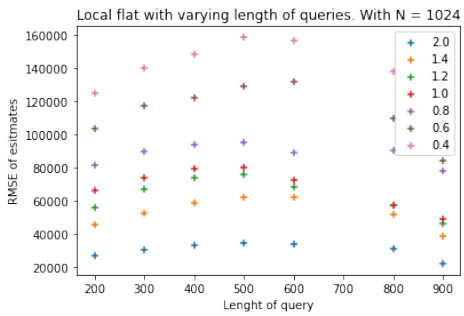
(a) RMSE as function of r for $N = 32$



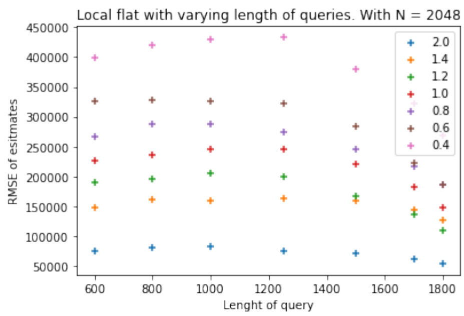
(b) RMSE as function of r for $N = 128$



(c) RMSE as function of r for $N = 512$



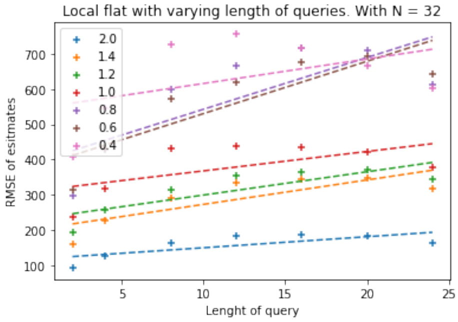
(d) RMSE as function of r for $N = 1024$



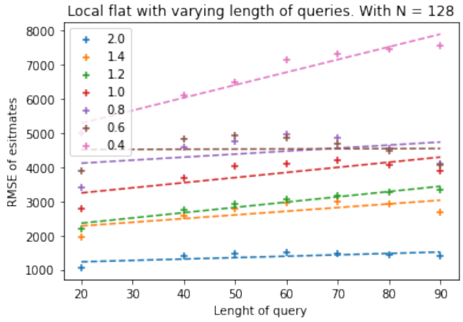
(e) RMSE as function of r for $N = 2028$

(f) RMSE as function of r and N

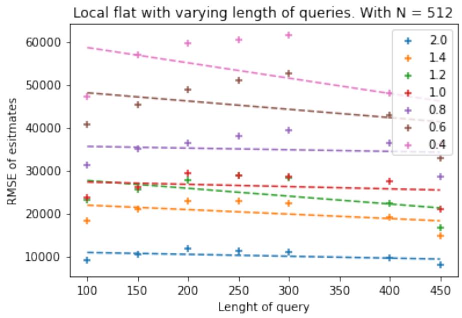
Figure 28: RMSE as function of r and N



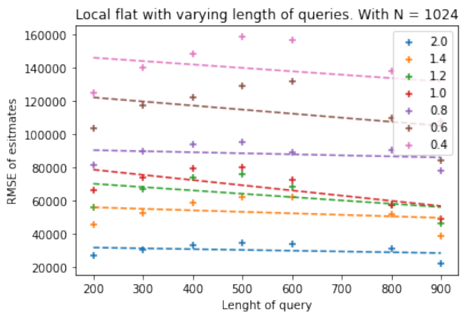
(a) RMSE as function of r for $N = 32$



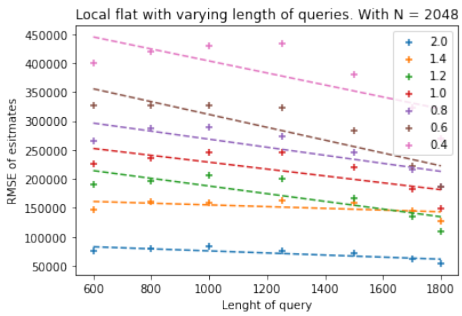
(b) RMSE as function of r for $N = 128$



(c) RMSE as function of r for $N = 512$



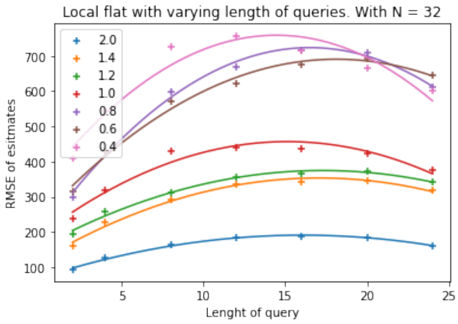
(d) RMSE as function of r for $N = 1024$



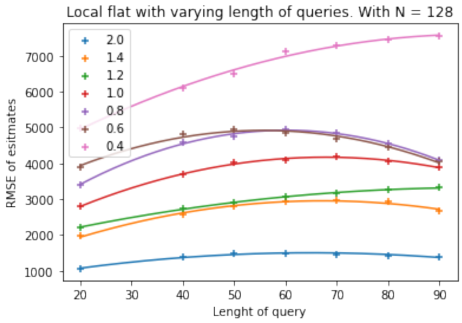
(e) RMSE as function of r for $N = 2028$

(f) RMSE as function of r and N

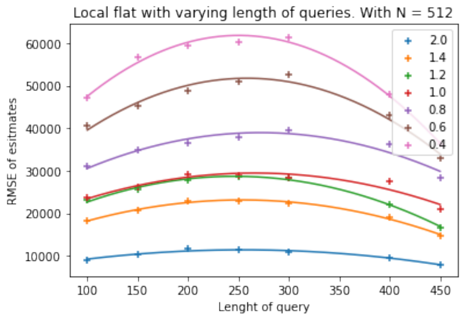
Figure 29: RMSE as function of r and N



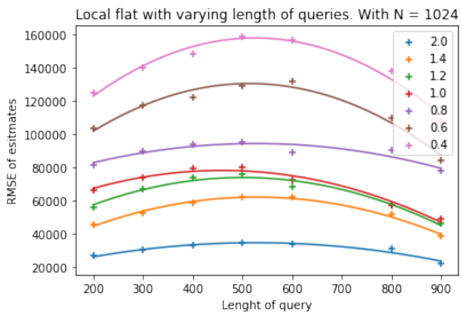
(a) RMSE as function of r for $N = 32$



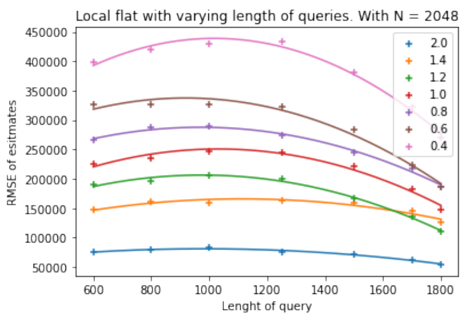
(b) RMSE as function of r for $N = 128$



(c) RMSE as function of r for $N = 512$



(d) RMSE as function of r for $N = 1024$

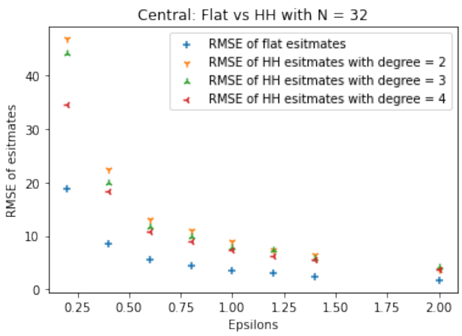


(e) RMSE as function of r for $N = 2028$

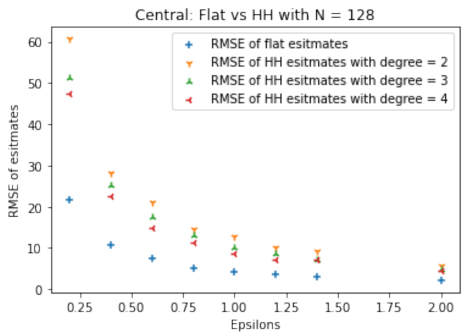
(f) RMSE as function of r and N

Figure 30: RMSE as function of r and N

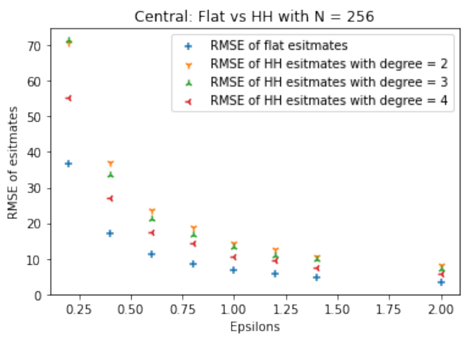
14.3.3 Central flat beating continuous observation



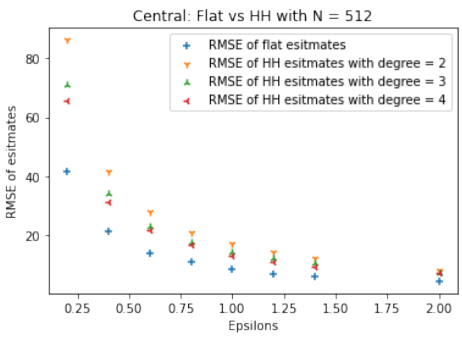
(a) Linear fit of error for $N = 32$



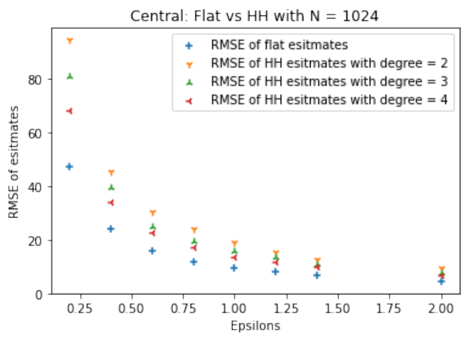
(b) Linear fit of error for $N = 128$



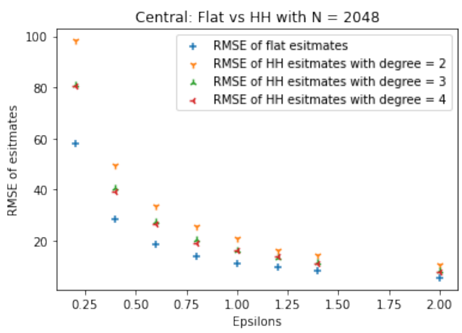
(c) Linear fit of error for $N = 256$



(d) Linear fit of error for $N = 512$



(e) Linear fit of error for $N = 1024$



(f) Linear fit of error for $N = 2048$

Figure 31: Central flat beating continuous observation