UNIVERSITÄT ZU LÜBECK

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Learning advanced differential privacy mechanisms with fixed data order

Master Thesis

Nadja Aoutouf

Thursday 30th September, 2021

Supervised by:
Prof. Peter L. Bühlmann, ETH Zürich
Prof. Esfandiar Mohammadi, Universität zu Lübeck
Dr. David Sommer, ETH Zürich

Department of Mathematics, ETH Zürich

Abstract

Companies, governments, and institutes such as hospitals, have a strong interest in gathering vast amounts of data about individuals. The potential benefits from analyzing these data are tremendous. Since most of this information is private or sensitive, the main challenge is to reveal useful information about the data while protecting the privacy of individual contributors. The framework of differential privacy provides a measurable level of privacy protection for such data. Thus, a differential privacy-based approximation ensures that statistical conclusions are reached deniable if an individual is included in the data set or not, by using suitable additive noising strategies. In this work, we consider the problem of producing a general and problem-dependent mechanism that learns a suitable differentially private approximation based on public data only. Given a first prototype derived in previous work, the aim of this work is to develop a more accurate and applicable advanced prototype version. In this process, first potential points of improvement are identified. Later, techniques which incorporate these improvements into the advanced prototype version are developed and implemented.

Acknowledgements

Herewith, I would like to thank Esfandiar Mohammadi and David Sommer for their generous support throughout this work as well as their valuable feedback.

Contents

Chapter 1

Introduction

There is a strong interest among governments, companies, and individuals in collecting and analyzing digital data [5] [6]. Machine learning (ML) models based on these data offer functions that are tailored to specific Big Data applications, allowing to gain valuable insights into the behavior of groups or individuals. Yet, many ML models do not have a concise representation and might have unexpected behavior for specific data sets. Such behavior can leak information about the input data set and thereby lead to privacy violations. In order to preserve privacy while maintaining the benefits of analyzing data, there is great interest in a technique providing both privacy protection and data accuracy.

Differential privacy (DP) is a mathematical notion for analyzing whether a function exhibits privacy leakage. By constructing a DP approximation of a function offered by an ML model, a strong privacy guarantee is provided. More precisely, DP approximations ensure that statistical conclusions are reached deniable of whether an individual is included in the data set, thus by adding random noise [4]. When developing a DP approximation, it is crucial to find an appropriate balance between accuracy and privacy. To consider this, a common method to design a DP approximation is to limit the maximal change of the output value after replacing one entry of a data set in the input space. Such a maximum change in the output space is called sensitivity. There are several approaches that use this method to develop a mechanism that constructs a suitable DP approximation for a given problem using the specific training data as well as problem depending information. This very problem dependency leads to the fact that once a mechanism has been developed, it is not applicable to new problems. A generic approach for learning DP approximation is one solution to construct a problem-dependent mechanism from training data alone. In this work, we make significant progress towards developing such a generic mechanism, concentrating on the theoretical aspects. Given an ML model, we construct a mechanism that

learns an approximation that has bounded sensitivity and thus directly leads to differential privacy.

In prior work, we derived a first prototype towards learning a DP approximation from arbitrary public data, for high-dimensional input spaces (with one attribute per data-point) and one-dimensional output spaces [2] [10]. This prototype is based on a mechanism that divides the input space into so-called areas. More precisely, it structures the domain into a multi-dimensional chessboard-like model composed of equal-sized hypercubes which has the crucial advantage that DP-neighbours (i.e. a set of databases that differ in at most one entry) can be identified more efficiently. The mechanism is based on this structure to generate consistent output ranges that satisfy the sensitivity constraints. Then, the output values of the approximation are limited to especially these ranges to ensure that the sensitivity bounds are never exceeded. Finally, an appropriate amount of random noise is added to the output, leading to a differential private approximation for new data sets with the same statistic as the public data set. This prototype has been proven to be both general and problem-dependent. Moreover, testing this prototype shows promising results. Nevertheless, there is still potential of improvement which is not addressed yet.

Our Contribution. In this work, we will focus on deriving an advanced version of this first prototype, by identifying and addressing potential points of improvement. More precisely, the prototype version derived in this work will incorporate the following points:

- **Sparse Data.** We derive and implement a procedure that generates an accurate and valid DP approximation even if the number of areas is significantly larger than the number of data sets.
- **Optimal Abort Condition.** We propose and implement an abort condition that ensures that our mechanism stops exactly when the sensitivity is best exploited.
- **Multivariate Input Data.** We generalize the approach used to construct the first prototype from high-dimensional input data $\mathbb{N}^n$ to multivariate input data $\mathbb{N}^{nxd}$. Then, given a data set D contained in the input space, each row is a record associated with an individual, and each column represents one attribute.
- **Smoothed Sensitivity.** Our first prototype version learns a DP approximation based on the global sensitivity S_{global}. More precisely, it uses global sensitivity to measure the amount of noise that has to be added to the statistic g to be privacy-preserving. In this work, we consider the so-called smooth sensitivity while learning a DP approximation. This

smooth sensitivity additionally depends on the data set D making the advanced prototype usable to a broader range of applications.

Furthermore, throughout this work, the derived prototype will be adapted to supervised learning methods such as a decision tree.

Applications. With the proposed mechanism in this work, it is possible to learn a differential private approximation for a non-commutative function, that offers good practical accuracy without any trusted server. Additionally, our advanced prototype expands the applicability range by adding a smaller amount of noise if possible making the output of the resulting DP approximation more accurate. The median is one example of a function where this property is crucial.

Structure. This work is structured as follows. In the beginning, fundamental definitions and concepts which form the basis of this work are introduced. More precisely, the framework of differential privacy as well as the construction of the first prototype version of our prior work are recalled. Next, the problem statement of this work is presented in detail in chapter 3. In chapter 4 the conceptual work that forms the foundation for the development of the advanced prototype version is discussed. Then, in chapter 5 the process of integrating our approaches from chapter 4 in the prototype is explained in detail. Afterwards, the correctness, as well as the performance of the proposed prototype version, is proven. In chapter 6, ways to adapt the prototype model to other learning mechanisms are demonstrated. Thereafter, in chapter 7, the functionality of our constructed prototype is evaluated. Further, possible directions for future work are specified in chapter 8. Finally, the results of this work are summarized in chapter 9.

Chapter 2

Preliminaries

This chapter is organized into four parts. In the first part, the notations used in this thesis are presented. Next, the fundamental concept behind the term differential privacy is introduced. Thereafter, the prototype developed in our previous work, which forms the basis of this work, is described. Finally, we have a closer look at related work to consider other approaches which address the problem statement of this work.

2.1 Notation

Throughout this work the following notations are used:

Notation	**Explanation**
I_{train}	Training data
$D = (x_1, \ldots, x_n)$	Data set $D \in I$ with data points $x_1, \ldots, x_n$ being a multivariate time series such that $x_i = (a_1, \ldots, a_d)$ where $a_i \in \mathbb{N}$
$M_{I_{train}}$	A differentially private approximation, based on I_{train}
g	A function $g\colon I \to R$ with $I \subseteq \mathbb{N}^{nxd}$ and $R \subseteq \mathbb{R}$, which maps multivariate data sets to real numbers.
$\tilde{g}$	Represents a restricted "copy" of the function g in such a way that the sensitivity bounds are fulfilled.
S_{global}	Global sensitivity of g

Table 2.1: Notation

Notation	Explanation
$S_{local}(D)$	Local sensitivity of g on data set D
$\tilde{S}$	Smooth sensitivity approximation $\tilde{S}\colon I \to \mathbb{R}_+$ of g
$areas$	The set of all elements forming a partition of the domain I of g.
$area\_of\_data\_point(\cdot)$	A function which takes as input a data set $D \in I$ and returns the $area$ in I in which D is contained.
$area_{cv}$	Center Value of $area \in areas$.
$(area_{lb}, area_{ub})$	Lower buffer and upper buffer value of $area \in areas$.

Table 2.2: Notation

2.2 Differential Privacy

Before the definition of differential privacy is introduced let us first define some fundamental definitions, we will need later. Most of the definitions, theorems, and proofs are based on chapter 3 of [4] and [8], other references will be noted separately.

Definition 2.1 (neighbourhood) *Let $D = (x_1, \ldots, x_n)$ be a dataset. We call D and D' neighbouring datasets if they differ in at most one dimension x_i and x_i' . The set of neighbouring data sets to a fixed data set D, is called (direct) neighbourhood of D and is denoted with D_{direct_nghb}.*

This definition can be generalized as follows:

Definition 2.2 (k-neighbourhood) *Let D and D' are datasets. We call D and D' k-neighbouring data sets if they differ in exactly k dimensions. The set of k-neighbouring data sets to a fixed data set D, is called k-neighbourhood of D and is denoted with D_{k_nghb}.*

Definition 2.3 (DP-neighbourhood) *Let $D = (x_1, \ldots, x_n) \in I \subseteq \mathbb{N}^n$, then a DP-neighbourhood of D in dimension $i \in [n]$ is referring to a number of elements in I that differ only in the i-te dimension, i.e. $\{D' \in I\colon x_1' = x_1, \ldots, x_{i-1}' = x_{i-1}, x_{i+1}' = x_{i+1}, \ldots, x_n' = x_n\}$.*

Notice that the union of all DP-neighbourhoods of x are the same as the direct neighbourhood of x.

Definition 2.4 ((ε, δ)-differentially private) *Let $\varepsilon, \delta \geq 0$ and $I \subseteq \mathbb{N}^n$. A randomized mechanism $g\colon I \to R$ is called (ε, δ)-differentially private if for all $S \subseteq Range(g)$ and for all $D, D' \in I$ such that $D' \in D_{direct\_nghb}$:*

$$Pr[g(D) \in S] \leq e^{\varepsilon} Pr[g(D') \in S] + \delta.$$

If $\delta = 0$, we say that M is ε-differentially private.

More intuitively, differential privacy will guarantee that a randomized algorithm behaves indistinguishably on similar input data sets (i.e. data sets that differ in at most one dimension).

2.2.1 Sensitivity

Given two neighbouring data sets, there is a need to estimate the maximal possible change in the output range that can be expected, as this reveals the amount of noise that has to be added to preserve privacy. The sensitivity is such a measure. It captures the magnitude by which one element can change the output of a function.

Definition 2.5 (l_1-Global-Sensitivity) *Let $g\colon I \to R$ be a function and $D, D' \in I$ be two arbitrary data sets such that their differ in at most one data point (i.e., $D' \in D_{direct\_nghb}$). Then the l_1-global-sensitivity of g is defined by:*

$$S_{global} := \max_{D,D' \in I} ||g(D) - g(D')||_1.$$

Notice that by using the concept of global sensitivity described above, the noise magnitude depends on g and the privacy parameter ε, but not on the instance D. For many functions, such as the median g_{median}, this approach adds an excessive amount of noise, making the result meaningless. With the concept of smoothed sensitivity, this can be prevented. To compute the smoothed sensitivity function $\tilde{S}\colon I \to \mathbb{R}_+$ certain criteria must be respected. For instance, the smoothed sensitivity function should not change quickly in any neighbourhood of its input space. This criterion guarantees that no information is leaked even though different sensitivities are used within the data sets.

Before formally introducing the definition of smoothed sensitivity let us first propose a simpler version of a local measure of sensitivity:

Definition 2.6 (Local Sensitivity) *For $g\colon I \to R$ and $D \in I \subseteq \mathbb{N}^{nxd}$, the local sensitivity of g at D is defined by:*

$$S_{local}(D) := \max_{D'\colon D' \in D_{direct\_nghb}} ||g(D) - g(D')||_1.$$

Remark 2.7 *Notice that the maximal value of the local sensitivity of g is equal to the global sensitivity of g:*

$$S_{global} = \max_{D \in I} S_{local}(D).$$

The magnitude of added noise defined by the local sensitivity function can vary to the point that information about the data set is revealed. This can be prevented by using smoothed sensitivity instead.

Definition 2.8 (Smooth Bound) *For $\gamma > 0$ and $I \subseteq \mathbb{N}^{nxd}$, a function $\tilde{S} : I \to \mathbb{R}_+$ is a γ-smooth upper bound on the local sensitivity of g if it satisfies the following requirements:*

$$\forall D \in I : \tilde{S}(D) \geq S_{local}(D)$$
$$\forall D, D' \in I, d(D, D') = 1 : \tilde{S}(D) \leq e^{\gamma}\tilde{S}(D')$$

A function $\tilde{S}$ that is the smallest to satisfy the above definition 2.8 is called the smooth sensitivity of g.

2.2.2 Additive Noising Strategies

Given a function g and its bounded sensitivity S_g, one can convert g into a differential private approximation by adding the right amount of random noise. Such random noise could for an instant be drawn from the Laplace distribution. Then, the scale of the noise will be calibrated to the sensitivity of g (divided by ε). The Laplace mechanism describes exactly this procedure. In the following, we will first define the Laplace distribution to later formally introduce the Laplace mechanism

Definition 2.9 (Laplace Distribution) *The Laplace distribution (centered at 0) with scale b is the distribution with probability density function:*

$$Lap(x|b) = \frac{1}{2b}exp(-\frac{|x|}{b})$$

Note that we will write $Lap(0, b)$ to denote the Laplace distribution (centered at 0) with scale b.

Definition 2.10 (Laplace Mechanism) *Let $I \subseteq \mathbb{N}^{nxd}$ and $R \subseteq \mathbb{R}^k$. Given a function $g\colon I \to R$ with bounded sensitivity (i.e. $S_g < \infty$) and $D \in I$. The Laplace mechanism is defined as follow:*

$$M_L(D, g(\cdot), S_g, \varepsilon) = g(D) + (Y_1, \ldots, Y_d)$$

where Y_i are i.i.d. random variables drawn from the Laplace distribution centered around zero with scale $\frac{S_g}{\varepsilon}$ (i.e., $Y_i \sim Lap(0, \frac{S_g}{\varepsilon})$).

Notice that the amount of added noise increases proportionally with the magnitude of the sensitivity. Moreover, the value of epsilon reveals the degree of privacy guaranteed. Regarding the Laplace mechanism, this means that we add more noise for smaller epsilon values.

Theorem 2.11 *Let $\varepsilon > 0$ and $Y_i \sim Lap(0, \frac{S_g}{\varepsilon})$ for all $i \in [d]$. Then for any function $g\colon I \to R$ with bounded sensitivity S and $D \in I$, the Laplace mechanism*

$$M_L(D, g(\cdot), S, \varepsilon) = g(D) + (Y_1, \ldots, Y_d)$$

is $(\varepsilon, 0)$-differentially private.

For the proof of Theorem 2.11, we refer for instant to [4], Chapter 3.

Assuming a smooth sensitivity is given, the following theorem can be applied on functions that return a single real value:

Theorem 2.12 *Let $g\colon I \to R$ be any real-valued function and let $\tilde{S} : I \to \mathbb{R}_+$ be a γ-smooth upper bound on the local sensitivity of g. Then if $\gamma \leq \frac{\varepsilon}{2ln(\frac{2}{\delta})}$ and $\delta \in (0,1)$, the mechanism*

$$M_L(D, g(\cdot), \tilde{S}, \varepsilon) = g(D) + \frac{2\tilde{S}(D)}{\varepsilon} \cdot Y$$

where $Y \sim Lap(1)$, is (ε, δ)- differentially private.

The proof of Theorem 2.12, can be found in [8], Chapter 2.2.

2.3 DP Mechanism with High-Dimensional Input

The goal of this work is to develop a mechanism based on public data to learn a suitable differential private approximation. In this section, we present the approach used to construct our first prototype of such a mechanism derived in prior work [2]. Understanding this first prototype is essential since the mechanism constructed in this work is an extended version of the first prototype.

This first version aims to be applicable to different settings by considering individual statistics of public data to obtain a more reliable approximation. To realize this, the domain of the function g is divided into so-called areas. More precisely, the domain is structured into equal-sized hypercubes which has the crucial advantage that the DP-neighbours can be identified more efficiently. The former prototype is based on this structure to first find suitable center values for each hypercube to later generate consistent output ranges that satisfy the sensitivity constraints. After obtaining the output ranges for each area, the output values of the approximated function of g are limited to especially these ranges. This ensures that the sensitivity bounds are never exceeded. Finally, the appropriate amount of Laplace noise is added to the result, leading to a differential private approximation for new data sets with the same statistic as I_{train}.

Overall, this prototype is based on three basic steps. First, identifying appropriate center values for each area. Then, using these center values to determine valid output ranges. And finally, adding the appropriate amount of noise to the individual output values using the Laplace mechanism. In the following, these three basic steps are explained in more detail.

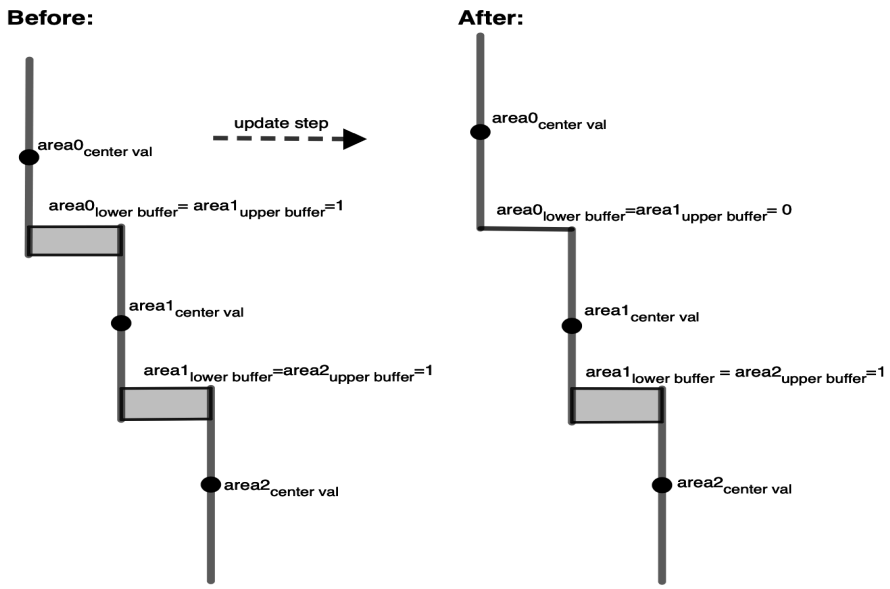


Figure 2.1: This illustration demonstrates the buffer idea while showing how shifting the center values would work for $area0, area2 \in area1_{direct\_nghb}$ such that $area0 \notin area2_{direct\_nghb}$. In this example $area0_{center\_val}$ can be shifted up without adjusting the center value of the direct neighbour $area1$, due to the fact that there was enough buffer left between $area0_{center\_val}$ and $area1_{center\_val}$.

2.3.1 Finding Center Values

To ensures compliance with the privacy constraints (i.e. sensitivity bounds) while maximizing accuracy, center values for each area are introduced. These center values are shifted during the mechanism towards the mean value of all output values within one area. Furthermore, to guarantee that the sensitivity bounds are respected, so-called buffers are initiated for each area. These buffers indicate the extent to which the center value can be shifted up or down without violating global consistency. In other words, these buffers represent a measure of the flexibility of each center value. Before moving a center value in one direction, the buffers of the affected area, as well as the (directly) neighbouring areas, indicate whether this shift is compatible with the sensitivity constraints. Besides that, the buffers reveal which of the neighbouring areas have to be shifted additionally in order to maintain consistency (as illustrated in figure 2.1 and figure 2.2). Furthermore, a synchronization function is used to ensure that flexibility is exploited more effectively. In detail, a synchronisation is performed if either not enough buffer is left to shift the center value of the area or in the case, the buffers of thus (direct) neighbouring areas, which must be updated to maintain consistency, are depleted. In this synchronization step, all center values are set equal to the mean value of the current center values. Therefore, after performing a synchronisation all center values are identical. Consequently,

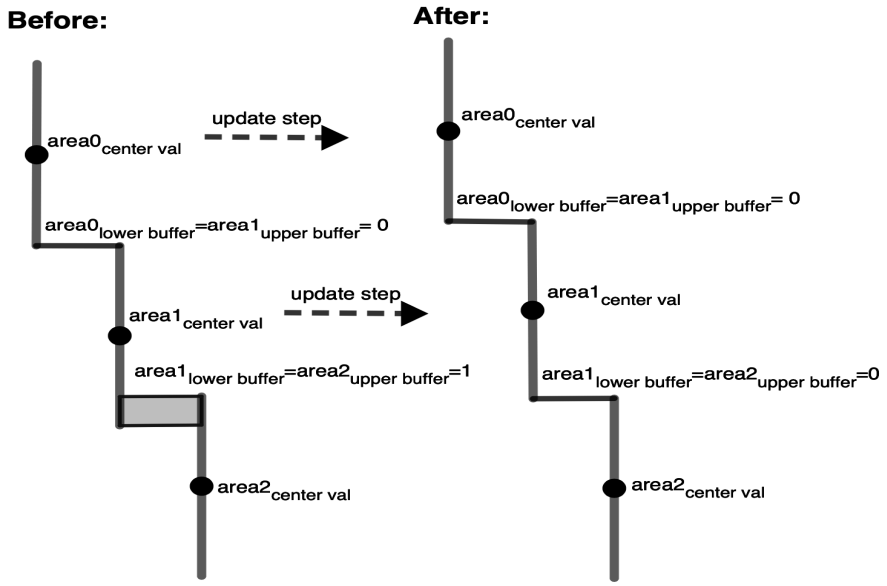


Figure 2.2: This illustration demonstrates the buffer idea while showing how shifting the center values would work for $area0, area2 \in area1_{direct\_nghb}$ with $area0 \notin area2_{direct\_nghb}$. In this example $area0_{center\_val}$ can only be shifted up if $area1_{center\_val}$ has enough lower buffer left to be adjusted, this is due to the fact that there was no buffer left between $area0_{center\_val}$ and $area1_{center\_val}$.

the buffers can be filled up again. To ensure the termination of this procedure only a given maximum number of performed synchronizations is allowed. Ultimately, this approach determines center values which indicates a suitable output value within the individual area as accurately as possible while respecting the privacy constraints.

2.3.2 Finding Valid Output Ranges

To find a consistent output range for each area it is essential that the center values were chosen to be no further apart than sensitivity S if they belong to areas that are within a DP-neighbourhood. As mentioned, the center value of each area represents a valid and suitable mean output value within the area. Therefore, by iteratively going over all areas, the minimum and maximum center values of all neighbouring areas can be used to set the first limits for the output ranges, which are no further than S wide. Next, these output ranges can be extended symmetrically up to a length of S, to maximize the flexibility.

2.3.3 Add Noise

Finally, by following both approaches, the output ranges of each area are obtained, such that the resulting approximation has sensitivity S. It remains

to apply appropriate noising techniques to obtain a mechanism that learns a DP approximation using the given public data, only. The Laplace Mechanism is such a noising technique.

2.3.4 Prototype of the General Problem-Dependent Mechanism

A first prototype version of a general problem-dependent mechanism, based on the previous three subsections, is formally introduced. This prototype outputs a valid DP approximation $M_{I_{train}}$, based on given public data.

Algorithm 1 General Problem-Dependent Mechanism

Input: $I_{train} = (I_{train_{input}}, I_{train_{output}}), count_{max}, EPOCHS, S, \varepsilon, g$
Output: DP approximation: $M_{I_{train}}$

1: // Finding center values
2: $\{area_{cv} : area \in areas\} \leftarrow$ **Finding_CV**$(I_{train}, count_{max}, EPOCHS, S)$
3: $\Omega_{all\_center\_values} \leftarrow \{area_{cv} : area \in areas\}$
4: // Finding valid output ranges
5: $\{area_{output\_range} : area \in areas\} \leftarrow$ **Output_Ranges**$(\Omega_{all\_center\_values}, S)$
6: // Add noise and define the resulting DP approximation $M_{I_{train}}$
7: **function** $M_{I_{train}}(D)$
8: $\quad area \leftarrow area\_of\_data\_set(D)$
9: $\quad (l_{area}, u_{area}) \leftarrow area_{output\_range}$
10: $\quad$ **if** $g(D) \leq l_{area}$ **then**
11: $\quad\quad \tilde{g}(D) \leftarrow l_{area}$
12: $\quad$ **else if** $u_{area} \leq g(D)$ **then**
13: $\quad\quad \tilde{g}(D) \leftarrow u_{area}$
14: $\quad$ **else**
15: $\quad\quad \tilde{g}(D) \leftarrow g(D)$
16: $\quad$ **end if**
17: $\quad Y \leftarrow Lap(\frac{S}{\varepsilon})$
18: $\quad M_{I_{train}}(D) \leftarrow \tilde{g}(D) + Y$
19: **end function**
20: **return** $M_{I_{train}}$

The input of this prototype is composed out of the maximal number of synchronizations that is allowed $count_{max}$, the sensitivity S, the number of iterations over the whole data set $EPOCHS$ as well as the public data set $I_{train} = (I_{train_{input}}, I_{train_{output}})$, which will be used as training data. As mentioned the prototype outputs a valid DP approximation $M_{I_{train}}$. First, a valid center value for each area is obtained using the function **Finding_CV**(). This function is based on the approach explained in section 2.3.1 (lines 1-2). Then, the function described in section 2.3.2 and denoted with **Output_Ranges**() is applied on the computed center values to find suitable output ranges

for each area (lines 4-5). Finally, based on the Laplace mechanism and the calculated output ranges, a suitable DP approximation $M_{I_{train}}$ is constructed and returned (lines 6-19).

Theorem 2.13 *Let a data set I_{train} be given. Furthermore, let $count_{max}, EPOCHS \in \mathbb{Z}_{\geq 0}$ and $S, \varepsilon \in \mathbb{R}_{\geq 0}$ be finite and given.*
*Then **Algorithm 1**$(I_{train}, count_{max}, EPOCHS, S, \varepsilon)$ returns a mechanism $M_{I_{train}}$ that is $(\varepsilon, 0)$-differentially private.*

For the proof of Theorem 2.13, we refer to [2], Chapter 5.

2.4 Related Work

This work aims to produce a general and problem-dependent mechanism that learns a suitable differentially private approximation based on public data. In this section, related work which addresses the problem statement will be discussed. The more theoretical work of Dwork and McSherry [3], proves that privacy can be preserved by calibrating the standard deviation of the noise according to the global sensitivity of a general function g. More precisely, a mechanism is presented which obtains ε-indistinguishability by adding noise according to a certain distribution. The results of this work are of great interest, creating the foundation for our approach. The work of Nissim, Raskhodnikova, and Smith [8] introduces a problem-specific mechanism which uses additive noising techniques to guarantee differential privacy of the resulting approximation. More detailed, in this work a mechanism that derives a DP approximation to the sample median is discussed. This mechanism provides accuracy while being privacy-preserving by adding noise proportional to the smoothed sensitivity. Nevertheless, it is not generic applicable. Furthermore, the so-called Bernstein functional mechanism determines a suitable DP approximation that depends on public data, only [1]. This generic mechanism is based on an approximation by Bernstein basis polynomials.

Chapter 3

Problem Statement

The aim of this work is to derive a generic mechanism, which learns a DP approximation that preserves the individual privacy of new data. This mechanism should be completely characterized by a given set of training data. An example to illustrate the usefulness of such a mechanism is as follows. Suppose that a store wants to analyze the shopping behavior of its customers in order to improve its services. In this analysis, confidential data such as time, visit time, purchase amount, gender, age, etc. is collected over a period of time from different customers to subsequently draw valuable conclusions about the shopping behavior. For such an private data analysis a DP approximation is needed, which is constructed based on training data, only.

So far, there is no reliable solution to this problem. According to state-of-the-art, there are two strategies to develop a differential private approximation. One of the strategies uses the specific statistics of the training data to find a suitable mechanism. However, it should be emphasized that this approach relies on information about the statistics of the data. Moreover, with this strategy one results in a mechanism that is not applicable to other statistics and therefore does not provide a proper solution. Thus, we are interested in a prototype that is adaptable to different problems and consequently to different statistics, while being based on training data, only. The other strategy is indeed applicable to different problems, but there are no existing realizations of such mechanisms which are satisfactorily reliable and accurate [10].

In our previous work, we have already developed a first prototype version, which offers a novel solution to the described problem. More detailed, given any set of training data and an arbitrary statistic S as a black-box (i.e., input and output is given, while no internal information about the statistic is provided), this prototype returns an DP approximation, which allows making unknown databases with the same statistic S differential private. In

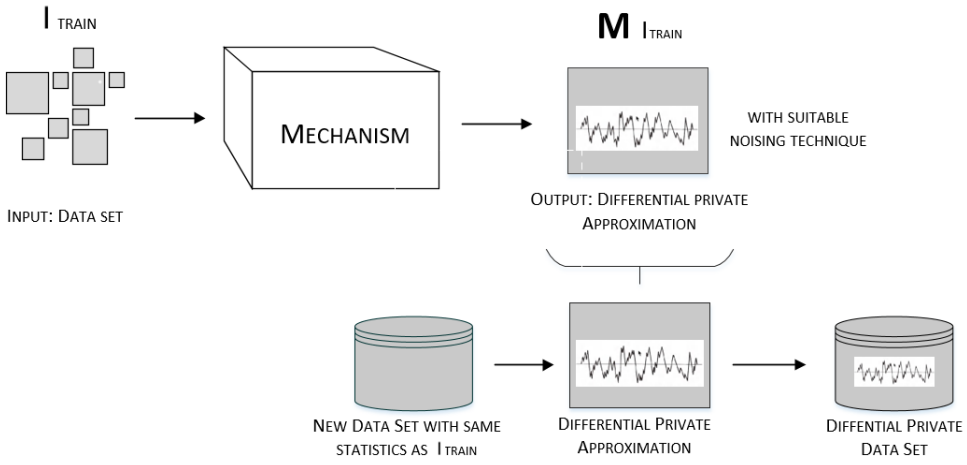


Figure 3.1: Upper illustration: The general problem dependent mechanism is shown, which is trained on a public data set I_{train} to finally return a differential private approximation $M_{I_{train}}$. Lower illustration: Here, it is shown, how the learned approximation $M_{I_{train}}$ can be used on new data sets.

other words, our prototype is both general and problem-dependent on high-dimensional input data (see figure 3.1). Beyond that, this first version respects the order of the input data, making the prototype particularly qualified for applications on time series data. However, this prototype is not yet fully developed. In this work, we will identify and address shortcomings of this first prototype and finally develop and implement an extended prototype version.

More specifically, in this work, we will develop an advanced prototype version based on the first prototype, while tackling the following weaknesses:

- **Sparse Data.** Currently, the approximation error tends to increase if the number of areas is significantly larger than the number of data sets. This described scenario leads to sparsity inside the domain. In the case of sparsity, the output ranges will not be adjusted by a significant number of areas. Resulting in a strong inaccuracy within the areas. However, even the areas that are not directly affected by this are forced to comply with the sensitivity bounds, which may prevent them from achieving high accuracy. As this can be the case if the dimension of the domain is increased, while the size of the training data set remains constant, there is a strong interest in finding an efficient solution for the problem of sparsity regarding this first prototype.
- **Optimal Abort Condition.** The prototype terminates directly as a given number of iterations has been performed. By implementing an elaborated abort condition, the algorithm stops when the sensitivity is

best exploited, which results in better accuracy.

- **Multivariate Input Data.** The first mechanism works for high-dimensional input datasets $\mathbb{N}^n$. Generalise this to multivariate input datasets $\mathbb{N}^{nxd}$, where each row is a record associated with some individual, and the columns are attributes, broadens the range of applications. Then, each attribute (column) can be thought of as a dimension, and each individual record as a point in the multidimensional attribute space.
- **Smoothed Sensitivity.** The sensitivity determines how much perturbation is required to preserve privacy. Our first prototype version derives an DP approximation by considering the global sensitivity S_{global}, such that enough noise is added to cover the "worst case" for the function g we want to approximate. Note that this global sensitivity does only depend on g. In fact, there are functions, as for example the Median function, that are not applicable on our prototype. More precisely, the prototype would add so much noise that the result is meaningless. By considering the smooth sensitivity, which additionally to the statistic g depends on the data set D, a suitable local sensitivity can be added to the output, making the prototype applicable for such functions.

Furthermore, this first version of the prototype is constructed to work on chessboard like model structures. In this work, we will adapt the prototype on structures of more general learning methods (e.g., decision trees).

Overall, this work aims to develop a more general version of the first prototype in which the improvements described above are considered as well as ways to adapt the derived prototype to other learning methods.

Chapter 4

Measures for Improvement

In this chapter, weaknesses of the former prototype are identified. In addition, strategies to counteract these weaknesses are derived.

In the beginning of this chapter it is discussed to which extent non-discrete center values can lead to inaccurate conclusions due to a limited amount of digits. Further a form of discretization is proposed. The next chapter presents techniques such that the prototype is better applicable to sparse data. After that, in subsection 4.3 an optimal termination condition is derived and its correctness is proven. In part 4.4, the generalization of the former prototype to multivariate input spaces is discussed. Finally, an approach is derived which allows the prototype to learn a DP approximation based on smooth sensitivity.

4.1 Discretization of Center Values

Floating point numbers are represented in a limited amount of digits. This has the consequence that not all real numbers can be represented accurately. More precisely, if a real number requires more than the maximum amount of digits and this number is stored as a float point number, then it will be rounded. Throughout the mechanism described in the first prototype the center values are repeatedly compared in absolute terms while their are stored as floating point numbers. Consequently, wrong conclusions can occur due to the rounding errors of the center values.

By converting the center values into discrete values, this issue can be addressed in our advanced version of the prototype. For the discretization of the center values the given variable $update\_step$ specifies the potential distance in which the center values are shifted after one update. Note that the smaller the value of $update\_step$ is chosen, the more flexibility can be used. However, at the same time the runtime for finding the desired center values increases.

Therefore, the goal is to choose a suitable size for $update\_step$ that reduces the runtime while providing best possible accuracy. Note that the optimal size of $update\_step$ varies depending on the function to be approximated.

In the following, we adapt the definition of consistent center values for discrete center values.

Definition 4.1 (Consistency for Discrete Center Values) *Let two areas area_1 and area_2 are k-neighbours. Then the corresponding center values area_1$_{CV}$ and area_2$_{CV}$ are consistent with each other, if their fulfill:*

$$abs((area\_1_{CV} - area\_2_{CV}) \cdot update\_step) \leq k \cdot S_{global}$$

Note that the definition of how valid buffers are defined remains the same.

Definition 4.2 (Valid Buffers) *Let area $\in$ areas. Then we call the lower buffer, denoted with area$_{lb}$, and the upper buffer, denoted with area$_{ub}$, valid buffers for area, if the maximum shift of the center values that is allowed by these buffers does not lead to inconsistency.*

4.2 Extension to Sparse Data

By first introducing a rank to each area a better understanding of what is meant with sparse data can be achieved.

Definition 4.3 (Rank) *All areas containing a data set that is included in the training data are defined to be of rank 0. More generally, an area is assigned to rank k ($k \in \mathbb{N}_0$) if this area is k direct neighbourhoods away from an area of rank 0. Accordingly, the set of direct neighbours of all areas of rank 0 are areas of rank 1 (or areas of rank 0, if two areas of rank 0 are direct neighbours).*

Due to the construction of our first prototype, only center values of areas of rank 0 and 1 are adjusted. Therefore, in the case of sparsity, the output ranges will not be adjusted by a significant number of areas. As a consequence, the approximation error of the currently implemented method tends to increase if the number of areas is significantly larger than the number of data sets. Hence, there is a strong interest in finding an efficient solution for the problem of sparsity regarding the first prototype.

A solution to the problem is to find a reasonable way to consider and adjust the center values of areas with a rank greater than 1. This can be achieved by defining criteria that specify such a reasonable approach. As a starting point, the synchronization could be modified in such a way that instead of setting all center values to the same mean value (as it is the case in the synchronization of the first prototype), only the center values of areas with a given rank k are shifted step-wise to the average of the center values of their

direct neighbours with rank $k-1$. Following this approach, an overview of the buffers of the individual areas can be kept while shifting the center values in such a way that the flexibility can be better exploited afterwards.

Before we explain the exact implementation of the advanced synchronization, two helper functions that are needed within the synchronization are presented. These functions specify how an update of one center value is implemented as well as how we can re-establish consistency in case of missing buffers by using the $undo\_update()$ function.

Algorithm 2 $update()$ - discrete version:

Input: $buffer, area, distance, adapted\_lower\_buffer\_areas,$
$adapted\_upper\_buffer\_areas$
Output: Set of areas with modified center values, lower and upper buffer

1: $area_{CV} \leftarrow area_{CV} + sign(distance)$
2: **if** $area_{lb} - sign(distance) \leq buffer$ **then**
3: $\quad area_{lb} \leftarrow area_{lb} - sign(distance)$
4: $\quad adapted\_lower\_buffer\_areas.append(area)$
5: **end if**
6: **if** $area_{ub} - sign(distance) \leq buffer$ **then**
7: $\quad area_{ub} \leftarrow area_{ub} + sign(distance)$
8: $\quad adapted\_upper\_buffer\_areas.append(area)$
9: **end if**
10: **return**

The $update()$ function takes as input the initial buffer size $buffer$ as well as the *area* and the direction *distance* in which we want to move the center value $area_{CV}$. Moreover, the two lists $adapted\_lower\_buffer\_areas$ and $adapted\_upper\_buffer\_areas$ keep track of the areas from which the buffers are adjusted during the $update()$ function. In the beginning, the discrete center value of *area* is shifted by one unit towards distance (line 1). Then, the lower or upper buffer of *area* is decreased by one unit depending on the direction of the shift (lines 2-9). Note that the value of each buffer can never be less than zero, this is because $update()$ is only performed if both buffers are greater or equal to one. Finally, in case the lower buffer $area_{lb}$ (resp. upper buffer $area_{ub}$) needs to be adjusted, *area* is added to the list $adapted\_lower\_buffer\_areas$ (resp. $adapted\_upper\_buffer\_areas$) (lines 4 and 8).

As the $update()$ function has now been described in more detail, the construction of the $undo\_update()$ function can be discussed. The $undo\_update()$ function represents the counterpart of the $update()$ function. Similar to $update()$, the function $undo\_update()$ requires as input the initial buffer size $buffer$, the considered area and the direction *distance* in which the center value

Algorithm 3 $undo\_update()$ - discrete version:

Input: $buffer, area, adapted\_areas, adapted\_lower\_buffer\_areas,$
$adapted\_upper\_buffer\_areas, distance$
Output: Set of areas with modified center values, lower and upper buffer

1: $area_{CV} \leftarrow area_{CV} - sign(distance)$
2: **for** $n\_area \in adapted\_areas$ **do**
3: $n\_area_{CV} \leftarrow n\_area_{CV} - sign(distance)$
4: **if** $n\_area \in adapted\_lower\_buffer\_areas$ **then**
5: $n\_area_{lb} \leftarrow n\_area_{lb} + sign(distance)$
6: **end if**
7: **if** $n\_area \in adapted\_upper\_buffer\_areas$ **then**
8: $n\_area_{ub} \leftarrow n\_area_{ub} - sign(distance)$
9: **end if**
10: **end for**
11: **return**

$area_{CV}$ was shifted. Moreover, the two lists $adapted\_lower\_buffer\_areas$ and $adapted\_upper\_buffer\_areas$ are crucial to reverse the adjustment of the buffer during the execution of the $update()$ function. Additionally, $undo\_update()$ needs a list which contains all neighbour areas for which the buffer has been adjusted. Therefore the list $adapted\_areas$ is also needed as input.

Using these two helpers functions the advanced synchronization can be described as follows. The advanced synchronization needs as input max_width, the initial buffer size of each area $buffer$, the step size $update\_step$ as well as lists consisting of areas with rank k (resp. rank $k+1$) saved in $areas\_rank\_k$ (resp. $areas\_rank\_k+1$). The synchronization returns areas with updated center values (resp. lower and upper buffers) such that more flexibility for areas with rank $k-1$ and k is created. First of all, we update the buffers of all areas of rank $k+1$, considering that the synchronization will be executed after shifting the center values of areas with rank k (in the main algorithm of the prototype) (line 1). Next, we iterate over all $areas$ of rank $k+1$. While iterating over $area \in area\_rank\_k+1$, the distance $dist$ between the center value of $area$ and the average of the center values of all areas of rank k which are neighbours of $area$ is calculated (lines 3-4). Afterwards, it will be checked if the calculated distance can be reduced by shifting the center value of $area$ by $update\_step$ (lines 5-7). If this is the case and $area$ has enough lower and upper buffer left, the center value of $area$ is shifted towards $dist$ by $update\_step$ (line 9). Thereafter, neighbour areas of $area$ are shifted in the same direction as $area_{CV}$ was shifted to maintain consistency (lines 13-17). In case such a neighbour area has not enough buffer left, all modifications made in the iteration over $area$ are reverted to maintain consistency (line 19). After adapting the neighbour areas, the buffers of $area$ are refreshed (line 23).

Finally, all buffers of rank $k, k+1$, and $k+2$ are adjusted (line 25).

Algorithm 4 Advanced Synchronization()

Input: $max\_width, buffer, areas\_rank\_k, areas\_rank\_k + 1, update\_step$
Output: Set of areas with new center values, lower and upper buffer

1: Update the buffers of all areas of rank $k+1$
2: **for** $area \in areas\_rank\_k + 1$ **do**
3: $mean_{rk\_k} \leftarrow mean(n\_area_{CV} | n\_area \in area[neighbours] \cap areas\_rank\_k)$
4: $dist \leftarrow mean_{rk\_k} - area_{CV}$
5: **if** $abs(dist) \cdot update\_step \leq \frac{update\_step}{2}$ **then**
6: **continue**
7: **end if**
8: **if** $min(area_{lb} - sign(dist), area_{ub} + sign(dist)) \geq 0$ **then**
9: $area_{CV} \leftarrow area_{CV} + sign(dist)$
10: $adapted\_areas \leftarrow []$
11: $adapted\_lower\_buffer\_areas \leftarrow []$
12: $adapted\_upper\_buffer\_areas \leftarrow []$
13: **for** $n\_area \in area[neighbours]$ **do**
14: **if** $abs((area_{CV} - n\_area_{CV}) \cdot update\_step) > max\_width$ **then**
15: **if** $min(n\_area_{lb} - sign(dist), n\_area_{ub} + sign(dist)) \geq 0$ **then**
16: Use $update()$ with variables $buffer, n\_area, dist, update\_step$, $adapted\_lower\_buffer\_areas$ and $adapted\_upper\_buffer\_areas$ to shift n_area_{CV} towards $dist$ by one and to adjust the buffers of n_area accordingly.
17: $adapted\_areas.append(n\_area)$
18: **else**
19: Use $undo\_update()$ with variables $buffer, area, adapted\_areas$, $adapted\_lower\_buffer\_areas, adapted\_upper\_buffer\_areas, dist$, and $update\_step$ to undo all previous steps for $area$ to maintain consistency.
20: **end if**
21: **end if**
22: **end for**
23: Adjust the buffers of $area$.
24: **end if**
25: Update all buffers of areas with rank k,k+1 and k+2
26: **end for**
27: **return**

Remark 4.4 *Note that the buffers within the for-loop in the function Advanced_Synchronisation() might be slightly incorrect. However, this does not lead to inconsistency, since all center values are shifted towards the same direction.*

Theorem 4.5 *Assume all input areas have consistent center values with each other and valid corresponding lower resp. upper buffer. Let* $max\_width \leq S_{global}$. *Furthermore, let* $update()$ *and* $undo\_update()$ *be to valid functions. Then the advanced synchronization described in algorithm* 4 *is correct and satisfies the following properties:*

(i) The resulting center values are consistent with each other.

(ii) The resulting buffers of each area are valid.

Proof Assume that all input areas have consistent center values with each other and valid corresponding lower resp. upper buffer. To prove correctness, we show that properties (i) and (ii) hold after each iteration of the for-loop over all areas with rank $k+1$. Due to the assumption, it suffices to show that if both properties (i) and (ii) hold at the beginning of an iteration, then they hold at the end of the iteration as well. First of all, notice that if not enough buffer is left or $abs(dist) \cdot update\_step \leq \frac{update\_step}{2}$ is fulfilled, then due to the assumption that (i) and (ii) are satisfied at the beginning of the iteration and the fact that no center value or buffer is modified, there is nothing to prove. Thus, let *area* be of rank $k+1$ such that enough buffer is left and $abs(dist) \cdot update\_step > \frac{update\_step}{2}$ is fulfilled. Then the center value of *area* will be shifted by one unit towards *dist*.
Now assume that there exist no neighbour of *area* (denoted with *n_area*) such that $abs(area_{CV} - n\_area_{CV}) \cdot update\_step \geq max\_width$ is satisfied after moving $area_{CV}$ by one unit. Then, only the center value of *area* is modified. Thus, to show property (i), it is enough to prove that the updated value of $area_{CV}$ has a maximum distance of S_{global} to all center values of neighbouring areas. Based on the assumption, we know that $abs(area_{CV} - n\_area_{CV}) \cdot update\_step < max\_width$ holds. Moreover, we have $max\_width \leq S_{global}$. Therefore, property (i) is fulfilled. To prove property (ii) it is enough to verify that the buffers of *area* are correctly adjusted since only the center value of *area* is modified. Due to the fact, that the buffers are adjusted after iterating over the direct neighbourhood of *area* (line 23), we can follow that property (ii) is fulfilled as well.
Next assume there exist at least one direct neighbour *n_area* of *area* such that $abs(area_{CV} - n\_area_{CV}) \cdot update\_step \geq max\_width$, where $area_{CV}$ denotes the shifted center value (see line 10). Consider the case when there exists a direct neighbour *n_area* which does not have enough buffer left, then the function $undo\_update()$ is executed as described in line 19. With the correctness of $undo\_update()$ we can follow that all modifications executed during the iteration over *area* are reversed. Hence, with the assumption that properties (i) and (ii) are fulfilled at the beginning of the iteration, it follows that both properties are fulfilled after the iteration over *n_area*, as well. Now consider the other case (i.e. all direct neighbours have enough buffer left) then only the function $update()$ is executed as described in line 17. The correctness

of $update()$ implies that the neighbours of *area* have valid buffers even after the update of the center values. In addition, line 23 ensures that *area* has valid buffers as well. Therefore, we can conclude that property (ii) is met after iterating over *area*. To see that property (i) is satisfied, notice that a center value is only modified, in case enough buffer is left. The validity of the buffers implies that the center values of neighbouring areas have a maximum distance of $\frac{S_{global}}{update\_step}$. Hence, property (i) is fulfilled. □

Theorem 4.6 *The running time of algorithm 4 is* $\mathcal{O}(|areas|^3)$.

Proof The running time of the advanced synchronization is dominated by the costs of the two nested for-loops. In addition, the functions $update()$ and $undo\_update()$ can be performed within the inner for-loop. Both functions need at most running time $\mathcal{O}(|areas|)$. Furthermore, inside the outer for-loop the buffers are adjusted. To adjust all buffers a running time of $\mathcal{O}(|areas|^2)$ is needed. Consequently, the following running time is required in total:

$$\mathcal{O}(|areas| \cdot (|areas|^2 + |areas| \cdot |areas|)) = \mathcal{O}(|areas|^3). \qquad \square$$

4.3 Optimal Abort Condition

The termination of the former prototype version was ensured by limiting the number of synchronizations that can be performed. While this approach guarantees that the algorithm terminates, in general, it fails to exploit the ideal amount of flexibility. In the following, one potential optimal termination condition will be described. Here, optimal refers to maximizing the accuracy of the returned DP approximation while ensuring that the mechanism stops as soon as this point is reached.

When deriving such an abort condition, note that the algorithm should stop when the majority of the areas reach optimal center values. However, it should be noted that this is not the case if one area alone leads to inconsistency due to a lack of flexibility for the corresponding center value. Furthermore, the distance between the "target values" of the individual center values is not qualified as a stopping criterion, since this distance differs depending on the statistic g. Thus, a measure that reflects the amount of flexibility used across all areas must be defined. The total shift of the center values after an iteration provides exactly such a measure. With the total shift of all center values during one iteration as well as a given lower bound *min_change*, which defines the allowed minimum displacement after one iteration, an optimal abort condition can be characterized with the following approach. First, verify after each iteration that the total amount of shifts of the center values is greater than *min_change*. If this is the case, continue with the next iteration. Otherwise, increase the variable *count* by one and then continue

with the next iteration. Finally, if the variable *count* has reached a given value *count_max*, stop the iteration and return the center values that have been calculated up to this point.

Theorem 4.7 *If* $min\_change = 0$*, then the abort condition described above is optimal.*

Proof To prove the optimality of the termination condition, it is enough to show once the abort conditions are reached the center values do not gain accuracy in further iterations. Assume, that the abort conditions are met. Thus $change \leq min\_change = 0$ holds. In other words, the center values were not shifted in the recent iteration. Now assume during following iterations the center values are shifted closer to the "target values". However, this is technically not possible, because the prototype determines the amount of shift based on the current center values as well as the buffers allocated to each area. In the case, that the center values have not been shifted within an iteration, both values did not change. Therefore, it can conclude that once there is no shift of the center values within one iteration, there will be no more shifts in following iterations. This implies that once the termination condition is satisfied further iterations will not lead to increased accuracy of the DP approximation. This proves the optimality of our abort condition. □

Next, a helper function will be defined, which will be used to implement the described abort condition in our advanced prototype.

Algorithm 5 $termination\_fct()$

Input: $old\_center\_val, new\_center\_val, min\_change, count, count\_max, break\_out\_flag, len(data), num\_areas$
Output: A tuple which symbolizes whether the algorithm should terminate

1: $change \leftarrow mean(abs(old\_center\_val - new\_center\_val))$
2: **if** $min\_change \geq change \cdot \frac{len(data)}{num\_areas}$ **then**
3: **if** $count \leq count\_max$ **then**
4: $count \leftarrow count + 1$
5: **else**
6: $break\_out\_flag \leftarrow$ TRUE
7: **end if**
8: **end if**
9: **return** $(count, break\_out\_flag)$

In order to integrate the abort method in the prototype, a function (denoted with $termination\_fct()$) is needed. By using this function after each iteration over the training set an overview of whether or not it is desirable to output the center values determined so far is given. This function takes as input both

the center values before *old_center_val* and after *new_center_val* the iteration. In addition, a minimal bound of the average shift per center value within one iteration *min_change*, the number of iterations in which this minimal shift was not reached *count*, as well as the upper bound *count_max* for *count*, is required as input. Besides that, the function uses the boolean variable $break\_out\_flag$, which indicates whether the mechanism stops, the length of the training set $len(data)$ as well as the number of areas *num_areas*, as input. Using this input, first the average shift per center value within the considered epoch is calculated and stored under *change* (line 1). In case, the average displacement per area is smaller than *min_change*, there are two ways to symbolize that the displacement in this epoch was smaller than desired. Either *count* will be increased by one, this is the case if this does not cause *count* to exceed the upper bound *count_max* (lines 3-4), or $break\_out\_flag$ is set to the value $TRUE$ (lines 5-7). Finally, the returned tuple $(count, break\_out\_flag)$ reveals if our prototype should terminate (line 9).

4.4 Extension to Multivariate Data

The first prototype can be applied only to univariate input data. With the aim of making our prototype applicable to more problems, it is of strong interest to adapt our prototype to multivariate input data. The problem statement of this work can be defined as follows for multivariate input data. Given a model M and a data set $D = (x_1, \ldots, x_n)$ with data points $x_1, \ldots, x_n$ being multivariate time series such that $x_i = (a_1, \ldots, a_d)$. Our prototype learns a DP approximation $\tilde{M}$, such that the influence on the output $\tilde{M}(D)$ of every data point of a new data set is protected.

In chapter 2 the neighbourhood relation is defined for univariate input data. Since this definition is fundamental to find a valid DP approximation it is necessary to extend that definition to multivariate input data. One potential extended neighbourhood definition is as follows:

Definition 4.8 (neighbourhood) *Let $D = (x_1, \ldots, x_n)$ be a dataset with $x_1, \ldots, x_n$ being a multivariate time series such that $x_i = (a_1, \ldots, a_d)$. We call D and D' neighbouring datasets if they differ in at most one data point x_i and x_i' . The set of neighbouring data sets to a fixed data set D, is called (direct) neighbourhood of D and is denoted with D_{direct_nghb}.*

In other words, two areas are considered as direct neighbours if they differ in exactly one data point. Note that, the number of attributes in which this data point differs is irrelevant. Using this neighbourhood definition, all other definitions defined in Chapter 2.2 are directly transferable.

To adapt the original prototype to multivariate input data, we have to clarify the definition of an area. Based on our definition of a neighbourhood, we consider an area to be a *nxd*-dimensional hypercube.

Finally, the extended prototype with multivariate input data can be implemented directly using these definitions.

4.5 Introducing Smoothed Sensitivity

Up to this point, we have derived a DP approximation by considering the global sensitivity S_{global}, such that enough noise is added to cover the "worst-case" for the statistic g. This global sensitivity does only depend on g (the function we want to approximate). By considering the smooth sensitivity, which additionally depends on the data set D, a more suitable local sensitivity can be added to the output.

Let k be the number of areas and $D \in \mathbb{R}^{nxd}$. Then the smooth sensitivity $\tilde{S}$ can be defined as piece-wise linear function as follow:

$$\tilde{S}(D) = \begin{cases} \tilde{S}_{area_1} & , D \in area_1 \\ \tilde{S}_{area_2} & , D \in area_2 \\ \quad \vdots & \\ \tilde{S}_{area_k} & , D \in area_k \end{cases}$$

4.5.1 Finding a Suitable Smoothed Sensitivity Function

To construct a valid sensitivity function we proceeded as follows. First, the local sensitivity of each area is calculated. Next, the initial value of the smooth sensitivity for each area is set equal to the global sensitivity. Then, the smooth sensitivity of each area is step-wise decreased by γ, as long as it stays greater or equal than the local sensitivity of the considered area. During this last process, it is checked if the smooth sensitivity of the direct neighbours of the considered area can be lowered as well. If this is not the case, the value of the sensitivity function on this area is fixed to maintain γ-smoothness.

In the following, it is presented how to implement this approach to finally obtain a valid smooth sensitivity function. After that, the correctness of the proposed smooth sensitivity function will be proven.

Algorithm 6 takes the set of areas, the global sensitivity as well as γ as input values and returns a suitable and valid smooth sensitivity function. In the beginning, we initialize a smooth sensitivity as well as a local sensitivity for each area (lines 1-7). More precisely, the smooth sensitivity is set equal to the global sensitivity S_{global}, which represents the upper bound for the smooth sensitivity. Note that during this algorithm the value of the smooth sensitivity can only decrease, this ensures that the smooth sensitivity never exceeds the value of the global sensitivity. The local sensitivity is approximated by the maximum distance between the neighbouring center values. Next, the

Algorithm 6 *smooth_sensitivity*()

Input: $areas$, S_{global}, γ
Output: $\tilde{S} : \mathbb{R}^{nxd} \rightarrow \mathbb{R}_+$

for $area \in areas$ **do**
  $\tilde{S}_{area} \leftarrow S_{global}$
  $S_{local(area)} \leftarrow 0$
  for $n\_area \in area['neighbours']$ **do**
    $S_{local(area)} \leftarrow \max\{S_{local(area)}, area_{cv} - n\_area_{cv}\}$
  end for
end for
$final\_area \leftarrow []$
$i \leftarrow 1$
while $areas \setminus final\_areas \neq \emptyset$ **do**
  for $area \in areas \setminus final\_areas$ **do**
    if $S_{global} - S_{local(area)} \geq i \cdot \gamma$ **then**
      $\tilde{S}_{area} \leftarrow \tilde{S}_{area} - \gamma$
    else
      $final\_areas.append(area)$
    end if
  end for
  $neighbour\_areas \leftarrow []$
  for $area \in final\_areas$ **do**
    for $n\_area \in area['neighbours']$ **do**
      if $n\_area \notin final\_areas \cup neighbour\_areas$ **then**
        $neighbour\_areas.append(n\_area)$
      end if
    end for
  end for
  $final\_areas.append(neighbour\_areas)$
  $i \leftarrow i + 1$
end while
return

smooth sensitivity is calculated (lines 10-28). For this, the first for-loop inside the while-loop (lines 11-17) ensures that the smooth sensitivity of one area is decreased only if the local sensitivity is still not undercut and the distance between the considered area to all direct neighbours maintains less than γ. The second loop ensures that if one area has reached its maximal lower bound while fulfilling the smooth sensitivity properties, the smooth sensitivity of neighbouring areas of one area can not be decreased by more than γ. Furthermore, due to the while-condition (line 10), the smooth sensitivity is reduced as long as possible while maintaining the characteristics of a smooth sensitivity function.

Lemma 4.9 *The function $\tilde{S} : \mathbb{R}^{nxd} \rightarrow \mathbb{R}_+$ describes a smooth sensitivtiy of g. Therefore the following properties are satisfied:*

(i) $\tilde{S}_{area} \geq S_{local(area)} : \forall area \in areas$

(ii) $\gamma \geq \max\{|\tilde{S}_{area} - \tilde{S}_{n\_area}| : \forall area \in areas, n\_area \in area['neighbours']\}$

This smooth sensitivity function $\tilde{S}$ has a running time of $\mathcal{O}(|areas|^2)$.

Proof To prove correctness, we show that both properties (i) and (ii) hold after each iteration of the while-loop over all data points. Due to the initialization (lines 1-8), all areas are initialized with a smooth sensitivity equal to S_{global} and a local sensitivity smaller or equal than S_{global}. To see the last one, assume there exist one area such that $S_{local(area)} > S_{global}$, then due to the construction there must exist a neighbouring area $n\_area \in area['neighbours]$ such that the center values of these areas have distance greater than S_{global} (i.e., $|area_{cv} - n\_area_{cv}| > S_{global}$). But this contradicts the construction of our prototype, in which center values of neighbouring areas are computed such that there have at most difference $\frac{S_{global}}{2}$. Hence, properties (i) and (ii) hold at the beginning of the iteration. Therefore, it suffices to show that if both properties (i) and (ii) hold at the beginning of an iteration, then they hold at the end of the iteration as well. Since in each iteration over the while-loop the value of the variable i increases by one unit, let us assume that both properties are fulfilled at the beginning of the i'te iteration. To show that they hold at the end of the i'te iteration, notice that the smooth sensitivity is only adopted by areas that are not yet contained in the list $final\_areas$ and satisfy the if-condition in line 12 (lines 11-16).

Let $area \in areas$ such that $area \notin final\_areas$ and $S_{global} - S_{local(area)} \geq i \cdot \gamma$ (line 13) is satisfied. To see that $area$ fulfilles property (i) at the end of the iteration, it is enough to show $\tilde{S}_{area} - \gamma \geq S_{local(area)}$. First notice that $S_{global} - S_{local(area)} \geq (i-1) \cdot \gamma$ must be satisfied since $area \notin final\_areas$. Furthermore, by assumption $S_{global} - S_{local(area)} \geq i \cdot \gamma$ and $S_{local(area)} \leq \tilde{S}_{area}$

holds. Hence,

$$\begin{aligned} S_{local(area)} &\leq S_{global} - i \cdot \gamma \\ &= S_{global} - (i-1) \cdot \gamma - \gamma \\ &\leq S_{local(area)} - \gamma \\ &\leq \tilde{S}_{area} - \gamma. \end{aligned}$$

Now assume $area \in final\_areas$ or $S_{global} - S_{local(area)} < i \cdot \gamma$ is fulfilled, then the value of $\tilde{S}_{area}$ remains unchanged, implying that (i) is satisfied. This proves that property (i) holds.

To show property (ii) notice that whenever the smooth sensitivity of $area \in areas$ remains unchanged during one iteration (i.e., $area \notin final\_areas$ or $S_{global} - S_{local(area)} < i \cdot \gamma$), then the third for-loop (lines 19-26) ensures that the smooth sensitivity of all neighbouring areas of *area* can not be decreased in the next iteration. In other words, the maximal distance between the smooth sensitivities of a pair of neighbouring areas is bounded by γ, which is exactly property (ii). This proves the correctness of the algorithm 6.

Let us now consider the running time of algorithm 6. The first two nested for-loops and the following while loop dominate the run-time of algorithm 6. The nested for-loops need $\mathcal{O}(|areas|^2)$ to be executed. Whereas the while loop requires $\mathcal{O}(|areas|^2)$ time. Consequently, the total running time is $\mathcal{O}(|areas|^2 + |areas|^2) = \mathcal{O}(|areas|^2)$. □

Chapter 5

Our Advanced Prototype

In the previous chapter, we presented various points of improvement for the former prototype as well as approaches to tackle these points of improvement. In this chapter, based on these findings, an advanced prototype is derived step by step, which is both general and problem-dependent. Afterwards, the advantages and limitations of the proposed mechanism are discussed.

5.1 Implementation of the Advanced Prototype

In this section, the idea of developing an improved version of the original mechanism will be put into practice.

5.1.1 Finding Center Values

In the following, the procedure used by the advanced prototype for finding valid center values is described. Thereby, the approaches used in the previous chapter are considered with the aim of producing accurate results even on sparse data. More, the termination condition described in section 4.3 is included in the advanced prototype. In addition, the discretization of the center values is considered.

Before presenting an algorithm that computes valid center values, it is necessary to introduce the function $simple\_syn()$. This function is relevant if only areas of rank 0 and 1 exist. Note that $simple\_syn()$ is identical to the synchronization function of the former prototype.

In the following, the construction of the simple synchronization function will be described in more detail. The function $simple\_syn()$ takes the initial buffer size denoted with $buffer\_val$ as input. It returns refreshed areas that have been assigned to new center values and refilled buffers. In the beginning, the mean value of the current center values of all areas is calculated (line 1). This mean center value is then assigned to all areas as the new center value (line

3). Since all areas are assigned to the same center value, the upper and lower buffers can be refilled (line 4).

Algorithm 7 *simple_syn*

Input: $buffer\_val$
Output: new center values and full buffers for all areas

1: $mean\_val \leftarrow mean(areas_{cv})$
2: **for all** $area \in areas$ **do**
3: $\quad area_{cv} \leftarrow mean\_val$
4: $\quad (area_{lb}, area_{ub}) \leftarrow (buffer\_val, buffer\_val)$
5: **end for**

Based on the simple and advanced synchronization, one can construct an advanced function that determines suitable center values as follow.

Algorithm 8 requires as an input the training set I_{train} and a lower bound *min_change*, which specifies the amount of the desired shifting of the center values per iteration. Besides that, a maximum number *count_max* that indicates how often the shift of the center values in one iteration is allowed to be smaller than *change_min* is needed. Moreover, the maximum number of iterations over the training set $EPOCHS$, the global sensitivity S_{global}, and the initial buffer value $buffer\_val$ are required as input. Then, this algorithm returns valid center values for each area. To determine these center values, first variables which are needed throughout the algorithm are initialized. More precisely, the variable *count* is initialized to zero (line 2) in order to count the number of periods in which the shift of the center values has been on average less than *min_change*. In addition, in case there are no areas with rank bigger or equal than two, the variable *count* counts the number of performed simple synchronizations. Then the variable $break\_out\_flag$ is initialized with the boolean value $FALSE$ (line 3). This variable is used to stop the algorithm according to the abort condition. Moreover, the variables *max_width* and *update_step* are initialized with $\frac{S_{global}}{2}$ and $\frac{S_{global}}{len(I_{train})}$ (lines 4-5). Next, the center values, as well as the buffers of the individual areas, are defined (lines 6-10). In addition, the rank of each area is specified (line 11). Afterwards, we iterate up to $EPOCHS$ times over all data points in I_{train}. At the beginning of each iteration, the current center values are stored in the list *old_center_values* (line 13). Thereafter, the area in which the input of the data set D_{input} is located is computed (line 15). The distance between the discrete center value of this area and D_{output} is stored in *dist* (line 16). Next, in case *area* has enough buffer left (line 17), *dist* will be used to specify the direction in which the center value of *area* is shifted (line 18). After adjusting the corresponding buffers to the shift of the center value, three empty lists are initialized (lines 19-23). These lists serve to keep track of

Algorithm 8 Finding center values

Input: $I_{train} = (I_{train\_input}, I_{train\_output})$, $count_{max}, min\_change, EPOCHS$, $S_{global}, buffer\_val$
Output: center value for each area

// Initialization of variables:
$count \leftarrow 0$
$break\_out\_flag \leftarrow$ FALSE
$max\_width \leftarrow \frac{S_{global}}{2}$
$update\_step \leftarrow \frac{S_{global}}{len(I_{train})}$
// Initialization of all areas:
for all $area \in areas$ **do**
  $area_{cv} \leftarrow round(\frac{mean(I_{output})}{update\_step})$
  $(area_{lb}, area_{ub}) \leftarrow (\frac{len(I_{train})}{2}, \frac{len(I_{train})}{2})$
end for
Assign ranks to each area
for all *epoche* in range($EPOCHS$) **do**
  $old\_center\_values \leftarrow$ store all current center values
  for all $D = (D_{input}, D_{output})$ in I_{train} **do**
    $area \leftarrow area\_of\_data\_set(D_{input})$
    $dist \leftarrow D_{output} - area_{cv} \cdot update\_step$
    if $\min(area_{lb} - sign(dist), area_{ub} - sign(dist)) \geq 0$ **then**
      $area_{cv} \leftarrow area_{cv} + sign(dist)$
      $area_{lb} \leftarrow \min(area_{lb} - sign(dist), buffer\_val)$
      $area_{ub} \leftarrow \min(area_{ub} + sign(dist), buffer\_val)$
      $adapted\_areas \leftarrow []$
      $adapted\_lb\_areas \leftarrow []$
      $adapted\_ub\_areas \leftarrow []$
      for all $n\_area \in areas['neighbours']$ **do**
        if $abs((n\_area_{cv} - area_{cv}) \cdot update\_step) < max\_width$ **then**
          if $\min(n\_area_{lb} - sign(dist), n\_area_{ub} + sign(dist)) \geq 0$ **then**
            $adapted\_areas.append(n\_area)$
            $update(buffer\_val, area, dist, adapted\_lb\_areas,$
                $adapted\_ub\_areas)$

```
30:             else
31:               // Advanced Synchronization:
32:               if adapted_areas_rank2 ≠ [] then
33:                 undo_update(buffer_val, area, adapted_areas,
34:                             adapted_lb_areas, adapted_ub_areas, dist)
35:                 advanced_syn(max_width, buffer_val, areas_rk2, areas_rk3,
36:                              update_step)
37:                 advanced_syn(max_width, buffer_val, areas_rk3, areas_rk4,
38:                              update_step)
39:                 advanced_syn(max_width, buffer_val, areas_rk4, areas_rk5,
40:                              update_step)
41:                 go to line 55
42:                 // Simple Synchronization:
43:               else if count ≤ count_max then
44:                 simple_syn(buffer_val)
45:                 count ← count + 1
46:                 go to line 55
47:               else
48:                 undo_update(buffer_val, area, adapted_areas,
49:                             adapted_lb_areas, adapted_ub_areas, dist)
50:                 go to line 68
51:               end if
52:             end if
53:           end if
54:         end for
55:         update all buffers
56:       end if
57:     end for
58:     new_center_values ← store all new center values
59:     (count, break_out_flag) ← termination_fct(old_center_values,
60:                                               new_center_values,
61:                                               min_change, count,
62:                                               count_max, len(data),
63:                                               len(areas))
64:     if break_out_flag == TRUE then
65:       break
66:     end if
67: end for
68: return area_cv for all areas
```

which neighbouring areas, from the currently considered area, need to have their center values adjusted in order to maintain consistency. Finally, we iterate over all direct neighbours of *area* and adjust the center values as well as the buffers if necessary. In case one neighbouring area cannot be adjusted due to resulting inconsistency, a synchronization will be performed (lines 24-46). The type of synchronization depends on whether there exist areas of rank greater than or equal to two. In case, all areas are either of rank 0 or 1, the variable *count* is used to restrict the number of simple synchronizations (lines 47-50). After a synchronization is performed, the buffers of each area are updated (line 55). Once an iteration over one epoch is completed the function $termination\_fct()$ is used to assess whether to iterate over another epoch (lines 58-66). Note that, this function is based on the amount of change of the center values created in one epoch. Finally, the computed center values for each area are returned (line 68).

Lemma 5.1 *Algorithm 8 is correct, which means it satisfies the following properties:*

(i) For all area $\in$ areas, the values of the corresponding buffers $(area_{lb}, area_{ub})$ never take on a negative value.

(ii) The center values of (directly) neighbouring areas have a maximum distance of sensitivity S_{global} from each other.

Further, algorithm 8 has a running time of $\mathcal{O}(EPOCHS \cdot |I_{train}| \cdot |areas|^3)$.

Proof To prove correctness, we show that properties (i) and (ii) hold after each iteration of the for-loop over all data sets contained in the training set. Due to the initialization, both properties (i) and (ii) hold at the beginning of the iteration. Therefore, it is sufficient to show that if properties (i) and (ii) hold at the beginning of an iteration, then both properties hold at the end of the iteration as well. First of all, notice that if there is not enough buffer left and $count > count_{max}$ (i.e., the abort conditions are met), then due to the assumption that properties (i) and (ii) are satisfied at the beginning of the iteration, there is nothing to prove. Thus, consider $area\_0 \in areas$ has enough buffer to update the corresponding center value $area\_0_{cv}$. Then either all direct neighbours $area\_0_{direct\_nghb}$ are adjusted to maintain consistency or, if not enough buffer is left, a synchronization is performed. In the first case, any neighbouring area $n\_area \in area\_0_{direct\_nghb}$ will be updated if the new center value $area\_0_{cv}$ and the unmodified center value n_area_{cv} would be further than $\frac{S_{global}}{2}$ away from each other, i.e. if consistency is not preserved. This ensures that property (ii) remains fulfilled. To see that property (i) is satisfied, notice that whenever the buffers are updated, we first check if enough buffer is left in $area\_0$ and $area\_0_{direct\_nghb}$ to adjust the center values as described in the if-condition. Therefore, after performing an update the buffers of $area\_0$ and $area\_0_{direct\_nghb}$ are at least zero. This proves property (i). In the other case, a synchronization is performed. More

precisely, if there exists one area with rank 2, then the inconsistency caused by the missing buffer of a neighbouring area is reversed with the function $undo\_update()$, before performing an advanced synchronization. Based on the correctness of the advanced synchronization (see theorem 4.5) we can conclude that the properties (i) and (ii) remain fulfilled after the execution of the advanced synchronization. Else if there exists no area with rank 2 and $count \leq count_{max}$ holds, then the simple synchronization is performed. In other words, all center values and buffers are reinitialized. Similar to the first initialization, properties (i) and (ii) are satisfied. Finally, if no area with rank 2 exist and $count > count_{max}$ holds, then by using the function $undo\_update()$ consistency is re-established (i.e., properties (i) and (ii) are fulfilled).

In the following, we will prove the run-time of algorithm 8. Notice that the run-time is dominated by the for-loop starting at line 12. In the worst case, the for-loop is executed $EPOCHS$ times. Within each iteration, we iterate $|I_{train}|$ times over neighbouring areas, which means a maximum number of $|areas| - 1$ times. Here, in the worst case with respect to the runtime the functions $undo\_update$ and $advanced\_syn$ are executed. Given that $undo\_update()$ (resp. $advanced\_syn()$) has a running time of $\mathcal{O}(|areas|)$ (resp. $\mathcal{O}(|areas|^2)$) the total running time of algorithm 8 is

$$\mathcal{O}(EPOCHS \cdot |I_{train}| \cdot |areas| \cdot (|areas| + |areas|^2))$$

$$= \mathcal{O}(EPOCHS \cdot |I_{train}| \cdot |areas|^3)$$

5.1.2 Finding Valid Output Ranges

In the previous section, a procedure to find valid discrete center values, which obtain high accuracy even on sparse data, was presented. This section discusses in detail how these center values can be used to find an output range for each area that has smooth sensitivity. After that, the correctness and performance of the proposed procedure is proven.

Let γ be a constant that determines the allowed change between the smoothed sensitivity values of two neighbouring areas. Then, given such a γ, the global sensitivity S_{global}, and consistent center values, suitable output ranges for each area can be calculated as follow. At the beginning of the process, the smooth sensitivity of each area is determined using the function $smooth\_sensitivity()$ (line 1). Similar to the former prototype, we iterate over each area to find suitable output ranges. Here, only the global sensitivity is replaced by the smooth sensitivity computed in line 1. In more detail, while iterating over $area \in areas$ a list denoted with $output\_range$ is created to store potential boundaries for the output range of $area$. In the beginning, this list is initialized with $\tilde{S}_{area}$-long interval constructed around the center value $area_{cv}$ (lines 2-3). Then, a valid interval is iteratively generated for each

Algorithm 9 Finding valid output ranges for each area

Input: (consistent) $area_{center\_val}$ for all areas, global sensitivity S_{global}, γ
Output: consistent output ranges for each area

1: $smooth\_sensitivity(areas, S, \gamma)$
2: **for** *area* in *areas* **do**
3: $output\_range \leftarrow [(area_{cv} - \frac{\tilde{S}_{area}}{2}, area_{cv} + \frac{\tilde{S}_{area}}{2})]$
4: **for** *dim* in $range(num\_dim)$ **do**
5: $lower\_bound \leftarrow \min_{n\_area \in area_{direct\_nghb}[dim]} (n\_area_{cv})$
6: $upper\_bound \leftarrow \max_{n\_area \in area_{direct\_nghb}[dim]} (n\_area_{cv})$
7: $diff \leftarrow upper\_bound - lower\_bound$
8: $lower\_bound \leftarrow lower\_bound - \frac{\tilde{S}_{area} - diff}{2}$
9: $upper\_bound \leftarrow upper\_bound + \frac{\tilde{S}_{area} - diff}{2}$
10: $interval \leftarrow output\_range.append(lower\_bound, upper\_bound)$
11: **end for**
12: $lower\_bound \leftarrow \max_{i \in range(num\_dim)} (interval[i, 0])$
13: $upper\_bound \leftarrow \min_{i \in range(num\_dim)} (interval[i, 1])$
14: $area_{output\_range} \leftarrow (lower\_bound, upper\_bound)$
15: **return** $area_{output\_range}$
16: **end for**

DP-neighbourhood of *area*, which includes all center values of the areas within the DP-neighbourhood. This interval will be extended to the length $\tilde{S}_{area}$ to obtain the highest possible flexibility. Furthermore, the calculated interval of each DP-neighbourhood of *area* will be added to the initialized list $output\_range$ (lines 4-11). In the next step, by choosing the lower (resp. upper) bound of $area_{output\_range}$ as the maximum lower (resp. minimum upper) bound of the valid intervals of the DP-neighbourhoods, we ensure that $area_{output\_range}$ is valid in every DP-neighbourhood which contains *area*. Notice, that this can be easily calculated by using the previously computed list $output\_range$ (lines 12-14). Finally, the output ranges are returned (lines 15-16).

Lemma 5.2 *Algorithm 9 is correct, meaning the following statements hold:*

(i) Two neighbouring areas are consistent with each other in the output range, i.e. for every area ∈ areas and n_area ∈ $area_{direct\_nghb}$ we have:

$$\max_{\substack{g(D) \in area_{output\_range} \\ g(D') \in n\_area_{output\_range}}} |g(D) - g(D')| \leq \tilde{S}_{area}.$$

(ii) For all area ∈ areas, the length of the output range satisfies:

$$len(area_{output\_range}) \leq \tilde{S}_{area} \leq S_{global}$$

Furthermore, algorithm 9 has a running time of $\mathcal{O}(|areas|^2 \cdot num\_dim)$.

Proof To show property (i), assume for the sake of contradiction, that there exist $g(D) \in area_{output\_range}$ and $g(D') \in n\_area_{output\_range}$ such that $|g(D) - g(D')| > min(\tilde{S}_{area}, \tilde{S}_{n\_area})$ holds. Moreover, assume w.l.o.g. that $area_{cv} \leq n\_area_{cv}$ holds. Based on the construction of the output ranges, we know that the following is satisfied:

$$g(D) \in [(area_{cv} - \frac{min(\tilde{S}_{area}, \tilde{S}_{n\_area})}{2}, area_{cv} + \frac{min(\tilde{S}_{area}, \tilde{S}_{n\_area})}{2})]$$
$$g(D') \in [(n\_area_{cv} - \frac{min(\tilde{S}_{area}, \tilde{S}_{n\_area})}{2}, n\_area_{cv} + \frac{min(\tilde{S}_{area}, \tilde{S}_{n\_area})}{2})]$$

From this, an upper bound for the distance between $g(D)$ and $g(D')$ can estimated as followed:

$$\begin{aligned} |g(D) - g(D')| &\leq (area_{cv} - \frac{min(\tilde{S}_{area}, \tilde{S}_{n\_area})}{2}) - (n\_area_{cv} + \frac{min(\tilde{S}_{area}, \tilde{S}_{n\_area})}{2}) \\ &= area_{cv} - n\_area_{cv} - min(\tilde{S}_{area}, \tilde{S}_{n\_area}) \\ &\leq min(\tilde{S}_{area}, \tilde{S}_{n\_area}) \\ &\leq \tilde{S}_{area} \end{aligned}$$

The first inequality follows by the assumption that $area_{cv} \leq n\_area_{cv}$ holds. Since we have chosen *n_area* to be a direct neighbour of *area* the distance between the center values is limited by $min(\tilde{S}_{area}, \tilde{S}_{n\_area})$. It follows, that $|g(D) - g(D')| \leq min(\tilde{S}_{area}, \tilde{S}_{n\_area})$ is fulfilled. This is a contradiction to our assumption. Thus, we can conclude that property (i) holds. It remains to show the property (ii). First, notice that the center values within a DP-neighbourhood regarding an area *area* are no farther apart than $\tilde{S}_{area}$ (i.e. $diff \leq \tilde{S}_{area}$). This implies that the bounds of the output ranges of each DP-neighbourhood are at most $\tilde{S}_{area}$ long. Second, if possible, the boundaries of each DP-neighbourhood are symmetrically extended to the length of the corresponding smooth sensitivity. Therefore, the lower (resp. upper) bound of the output range is set as the maximum lower (resp. minimum upper) bound of the previously computed smooth sensitivity long intervals. By the construction of our smooth sensitivity function, we can conclude that the output range of a given area reaches most the length of the corresponding smooth sensitivity. Thus, the correctness of the algorithm is shown.

The smooth sensitivity is constructed in $\mathcal{O}(|areas|^2)$ time (see lemma 4.9). However, the two nested for-loops dominate the running time of algorithm

9. Whereas $\mathcal{O}(|areas|)$ time is needed to find the minimum and maximum center value within the two for-loops. Therefore, in total the algorithm needs a running time of

$$\mathcal{O}(|areas|^2 + |areas| \cdot num\_dim \cdot |areas|) = \mathcal{O}(|areas|^2 \cdot num\_dim). \qquad \square$$

5.1.3 Adding Noise

In the previous subsections, a method to compute output ranges for each area that are both accurate and consistent with each other has been derived. Using these output ranges, an approximation of $g\colon I \to R$ can be defined, which is bounded by the corresponding smooth sensitivity. This restricted "copy" of g will be denoted by $\tilde{g}$. As in the original prototype, the co-domain R of $\tilde{g}$ must be restricted to the union of the output ranges, in order to fulfill smooth sensitivity for $\tilde{g}$. This can be realized by replacing the output values of g that exceed the output range of the corresponding area. More formally, let $D \in I$ and the output range of the area which contains D be $[l_D, u_D] \subseteq R$. Then one could define $\tilde{g}$ as follow:

$$\tilde{g}(D) = \begin{cases} l_D & \text{for } g(D) \leq l_D \\ g(D) & \text{for } l_D < g(D) < u_D \\ u_D & \text{for } u_D \leq g(D) \end{cases} \tag{5.1}$$

Given this approximation of g with forced output ranges and bounded sensitivity, differential privacy can be easily achieved by adding calibrated noise to each area. For instance, consider the Laplace mechanism $M_L(\cdot)$ introduced in section 2.2.2, which calibrates the added noise to the smooth sensitivity divided by ε. By applying theorem 2.12 we can conclude, that $M_{I_{train}} := M_L \circ \tilde{g}(I_{train})$ preserves differential privacy and therefore meets our desired criteria for the wanted differential private approximation.

5.1.4 Combing all Achievements to the Advanced Mechanism

In this section, all findings from the last three subsections are combined to construct an advanced prototype of a general problem-dependent mechanism, based on the first prototype version. First, this prototype is formally introduced. Thereafter, the correctness and performance of the advanced prototype is proven. More precisely, we prove that the derived advanced prototype indeed outputs a valid DP approximation $M_{I_{train}}$, based on given public data.

Based on the training data I_{train}, $count\_max$, $EPOCHS$, the global sensitivity S_{global}, the constants ε and γ, as well as the function g offered by the ML model, the algorithm 10 generates a valid DP approximation $M_{I_{train}}$. First,

Algorithm 10 Advanced General Problem-Dependent Mechanism

Input: $I_{train} = (I_{train\_input}, I_{train\_output}), count_{max}, EPOCHS, S_{global}, \varepsilon, g, \gamma$
Output: DP approximation: $M(I_{train})$

1: $\{area_{cv} : area \in areas\} \leftarrow$ **Algorithm 8**($I_{train}, count_{max}, EPOCHS, S_{global},$
2: $buffer\_val$)
3: $\Omega_{all\_center\_values} \leftarrow \{area_{cv} : area \in areas\}$
4: $\{area_{output\_range} : area \in areas\} \leftarrow$ **Algorithm 9**($\Omega_{all\_center\_values}, S_{global}, \gamma$)
5: **function** $M_{I_{train}}(D)$
6: $area \leftarrow area\_of\_data\_set(D)$
7: $(l_{area}, u_{area}) \leftarrow area_{output\_range}$
8: **if** $g(D) \leq l_{area}$ **then**
9: $\tilde{g}(D) \leftarrow l_{area}$
10: **else if** $u_{area} \leq g(D)$ **then**
11: $\tilde{g}(D) \leftarrow u_{area}$
12: **else**
13: $\tilde{g}(D) \leftarrow g(x)$
14: **end if**
15: $Y \leftarrow Lap(1)$
16: $M_{I_{train}}(D) \leftarrow \tilde{g}(D) + \frac{2 \cdot S_{smooth(area)}}{\varepsilon} \cdot Y$
17: **end function**
18: **return** $M_{I_{train}}$

algorithm 8 returns valid center values (lines 1-2), which are used in algorithm 9 to identify valid output ranges for each area (line 4). Given these output ranges, the DP approximation $M_{I_{train}}$ is defined throughout the lines 5-17. More precisely, the output values of g are restricted to the computed output ranges as described in equation 5.1. Furthermore, the optimal amount of Laplace noise will be added to the restricted output values of $\tilde{g}$.

Before proceeding with the analysis of the correctness and performance of algorithm 10, it is worth to recall an important property of the Laplace mechanism of section 2.2.2:

Theorem 2.12 *Let $g\colon I \to R$ be any real-valued function and let $\tilde{S} : I \to \mathbb{R}_+$ be a γ-smooth upper bound on the local sensitivity of g. Then if $\gamma \leq \frac{\varepsilon}{2ln(\frac{2}{\delta})}$ and $\delta \in (0,1)$, the mechanism*

$$M_L(D, g(\cdot), \tilde{S}, \varepsilon) = g(D) + \frac{2\tilde{S}(D)}{\varepsilon} \cdot Y$$

where $Y \sim Lap(1)$, is (ε, δ)- differentially private.

With the help of this theorem, we are now able to show the main theorem of this chapter.

Theorem 5.3 *Let a training set I_{train} be given. Furthermore, let $count_{max}$, S_{global}, $\varepsilon, \gamma \in \mathbb{R}_{\geq 0}$ and $EPOCHS \in \mathbb{Z}_{\geq 0}$ be finite and given. Let $\delta \in (0,1)$ such that $\gamma \leq \frac{\varepsilon}{2ln(\frac{2}{\delta})}$. Then **Algorithm 10**$(I_{train}, count_{max}, EPOCHS, S_{global}, \varepsilon, \gamma)$ returns a mechanism $M_{I_{train}}$ that is (ε, δ)-differentially private.*

Proof To prove that **Algorithm 10**$(I_{train}, count_{max}, EPOCHS, S_{global}, \varepsilon, \gamma)$ returns a mechanism $M_{I_{train}}$ that is (ε, δ)-differentially private, first note that within the algorithm consistent output ranges are computed for each area. This follows directly from lemma 5.1 and lemma 5.2. Therefore, it remains to verify that these output ranges are properly used to define the differentially private approximation $M_{I_{train}}$. In our advanced prototype, these consistent output ranges are used to ensure that the approximated function $\tilde{g}$ has bounded sensitivity. Therefore, according to theorem 2.12, by adding noise drawn from the Laplace distribution centered around zero with scale $\frac{2 \cdot \tilde{S}(D)}{\varepsilon}$ to $\tilde{g}$ one finally obtain an approximation $M_{I_{train}}$ which is (ε, δ)-differentially private. □

Lemma 5.4 *Let the running time of the function $area\_of\_data\_set(\cdot)$ be negligible. Then algorithm 10 has running time*

$$\mathcal{O}(|EPOCHS| \cdot |I_{train}| \cdot |areas|^3 + |areas|^2 \cdot num\_dim).$$

Proof Given that the running time of performing $area\_of\_data\_point(\cdot)$ is negligible. Then, the running time is dominated by the approximation of g, which depends on the run-time of algorithm 8 and algorithm 9. Therefore, the run-time of algorithm 10 is

$$\mathcal{O}(EPOCHS \cdot |I_{train}| \cdot |areas|^3 + |areas|^2 \cdot num\_dim). \qquad \square$$

5.2 Advantages and Limitations

In the last section, we derived an extended prototype version that approximates the statistic g of an arbitrary public data set I_{train} such that it satisfies differential privacy constraints. In this section, we will illustrate some advantages and weaknesses of this new prototype version.

5.2.1 Advantages

An important advantage of this prototype is that a more accurate DP approximation can be generated based on a small set of training data. To illustrate this, an approximation of the function g representing a hyperplane was applied to our prototype. Based on a training set of size two, our prototype generated the DP approximation illustrated in Figure 5.1. In this figure, it can be seen that the new approach allows more flexibility for areas of rank less or equal to two by controlling the center values of the other areas as well.

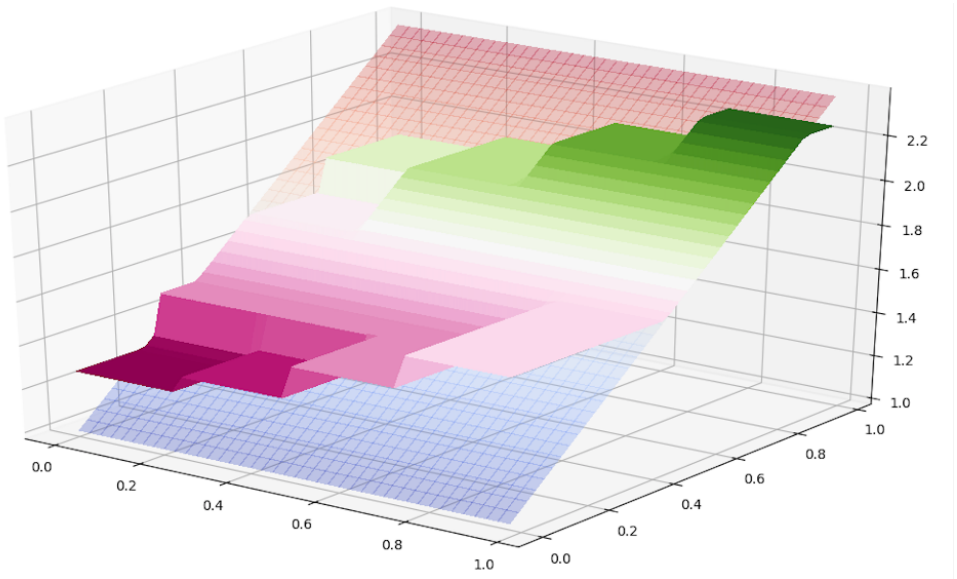


Figure 5.1: This figure shows both the linear function g and the implemented approximation $\tilde{g}$ (without added noise) based on two data points. Note that in order to represent g and $\tilde{g}$ graphically, the four-dimensional domain is illustrated in two dimensions.

Another important advantage of the derived prototype is that the order of the input data is respected. This is of great importance for applications with time-series data.

5.2.2 Limitations

One weakness of our mechanism concerns the neighbourhood definition in the multivariate input space. Due to the fact that we did not distinguish between the number of attributes in which two data points differ. Consequently, in the case of two data sets that differ only within the same data point in two or more attributes, the flexibility cannot be fully exploited. This limitation could, however, be alleviated with an adjusted neighboring relation.

Another weakness of our prototype is that a reasonable structure on the input space must be given. This structure defines how to split the input

space into a set of disjoint areas which need to be both reducing the size of the corresponding DP-neighbourhood as well as minimizing the output ranges defined by the input points inside of each area. Note, that the output ranges defined by one area can be reduced by understanding the learning mechanism which was used to learn the function g. However, some of the resulting structures created by learning mechanisms cannot be represented easily (e.g. neural networks). Therefore, our prototype is not yet optimized for such learning mechanisms.

Finally, it is important to point out that the efficiency and accuracy of the prototype depend strongly on the number of areas. This dependency has a strong effect on the usefulness of our advanced prototype. Unfortunately, with increasing complexity, more areas need to be considered resulting in longer running time.

Chapter 6

Adopting the Advanced Prototype on the Decision Tree Learning Method

In this chapter, we will take a closer look at the structure in which the input space is partitioned. Since this structure defines the shape of each area and therefore the corresponding neighbourhood relations, the form of the structure is crucial for the accuracy of the resulting DP approximation as well as for the efficiency of the mechanism. First, the structure used in the original prototype will be described. Afterwards, a potential structure, which is suitable for a decision tree learning method, is proposed. Later, this new structure is implemented and adapted to the advanced prototype derived in the previous chapter.

6.1 Chessboard-Like Model

First, it is essential to better understand the structure into which the domain was divided in the original prototype. To make the prototype more efficient, the input space was divided into areas that have the form of equal-sized hypercubes. This simplification has the advantage that it is possible to iterate over all areas instead of each point while remaining efficient. In addition, by choosing the areas as hypercubes, neighbourhoods of areas can be identified more easily. This chessboard-like structure is very suitable if the supervised learning method used to determine the function g divides the input space into a highly complicated structure that is not exactly comprehensible. For instance, when dealing with complex neural networks, this is the case. However, if a less complex learning method is used to determine g, it can lead to more accuracy if the structure of the domain is adapted to the learning method.

6.2 Decision-Tree Model

In this subsection, we adapt our advanced prototype to the structure generated by a regression decision tree. By the selection of a suitable supervised learning method, it was crucial that the resulting structure of the input space is not too complex. In order to apply our prototype to a given decision tree, we need to define a set of areas that participate the domain in such a way that the corresponding output values are predictable. For this, it is essential to understand the way a decision tree divides its input space and to clarify how this structure can be used to define areas in the input space. Finally, an approach to efficiently identify direct neighbours of each area is derived and implemented.

6.2.1 Partitioning of an Input Space by a Decision Tree

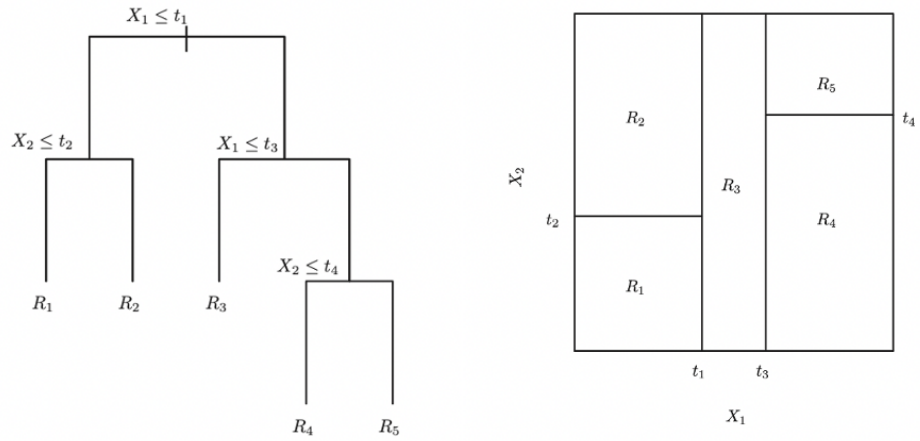


Figure 6.1: Left figure: A binary decision tree with three inner nodes and four leaf nodes. Right figure: The corresponding partition of the decision tree. (Source: [7])

Within this subsection, a better understanding of how the input space of a decision tree is partitioned will be given. A decision tree is a supervised learning method for classifying data and solving decision problems. A decision tree consists of a root node and an arbitrary number of inner nodes as well as at least two leaf nodes. Where each node represents a decision rule and each leaf represents an answer to the decision problem. For simplicity, assume in the following, that a binary decision tree with a two-dimensional input space is considered. An example of a binary decision tree is shown in figure 6.1. Observe that each input data point is assigned uniquely to one region (in Fig. 6.1 denoted with R_1, R_2, R_3, R_4). Moreover, within a region,

the output behavior is known. In figure 6.1 (right) it is illustrated how the input space is partitioned according to the corresponding decision tree.

6.2.2 Defining Areas on the Input Space

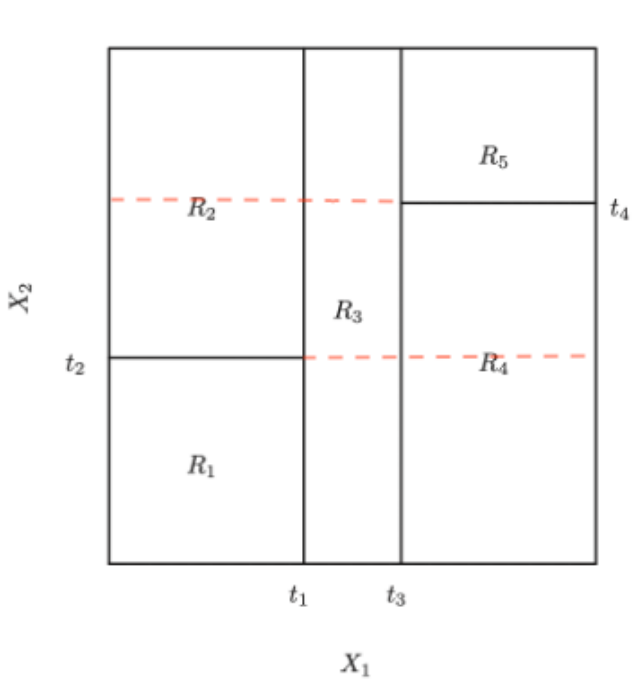


Figure 6.2: The partition of the input space of a binary decision tree into 9 different areas.

Using this structure created by the partition of the domain, a proper characterization of the areas can be described as follow. A decision tree splits the input space into disjointed orthotopes. One possible approach to define areas is to treat each region as one area. However, this can lead to a large number of direct neighbours for one area. Consequently, such an area has many constraints which must be considered when determining a valid output range. As a result, the accuracy of the DP approximation is often unsatisfactory. To overcome this problem we divide the input space into more areas. More precisely, the input space is completely spitted by all decision rules regardless of which sub-tree the inner node is located. This leads to an input space that is partitioned into a set of disjoint orthotopes of different sizes as demonstrated in figure 6.2. Due to the fact that an area is completely contained in one region, the output value range of the individual areas can be predicted, as desired.

6.2.3 Identifying Neighbourhood Relations

Using the characterization of the set of areas, we can start to derive an approach to identify direct neighbours. For this, a formal definition of two neighbouring areas is introduced first.

Definition 6.1 *Let $area_a, area_b \subseteq \mathbb{R}^{nxd}$ are two different areas. Define*

$$S := \{i \in [n] | D_i \neq D'_i : \forall D \in area_a, D' \in area_b\}.$$

Then $area_a$ and $area_b$ are called k-neighbours if $|S| = k$. We call $area_a$ and $area_b$ direct neighbours if $k = 1$.

One can simply find the direct neighbourhood for each area by pairwise comparison, if a function, that determines for two given areas whether these areas are direct neighbours, is given. Therefore, in the following, a function that determines if two given areas are direct neighbours is presented.

Algorithm 11 $iface.areas\_are\_neighbours()$

Input: $area_{id_a}, area_{id_b}$
Output: if $area_a, area_b$ differ in exactly one data point (i.e. *ite* data point) then output $[i]$ otherwise $[]$

```
1: threshold_a ← areas[area_id_a]['threshold']
2: threshold_b ← areas[area_id_b]['threshold']
3: equal_thresholds ← zeros(iface.hypercube_dim)
4: for i ∈ range(iface.hypercube_dim) do
5:    equal_thresholds[i] ← threshold_a[i] == threshold_b[i]
6: end for
7: non_equal_thresholds ← logical_not(equal_thresholds)
8: non_equal_data_points ← []
9: for i, j ∈ enumerate(non_equal_thresholds) do
10:   if j == 1 then
11:      non_equal_data_points.append(i)
12:   end if
13: end for
14: if len(non_equal_data_points) == 1 then
15:   return [non_equal_data_points]
16: else
17:   return []
18: end if
```

The function $iface.areas\_are\_neighbours()$ takes the area ids of $area_a$ and $area_b$ as input and returns a list. This returned list reveals if $area_a$ and $area_b$ are direct neighbours. In case both areas are neighbours, the returned list will indicate in which data point these two areas differ from each other. If both

areas are not directly neighbouring, an empty list is returned. First, all the boundaries defining $area_a$ (resp. $area_b$) are stored using the variable $threshold_a$ (resp. $threshold_b$) (lines 1-2). Then, an array *equal_thresholds* (and later *non_equal_thresholds*) of length n is used to store if the thresholds which define $area_a$ and $area_b$ differ (lines 3-7). Finally, in the initially empty list *non_equal_data_points* all dimensions in which the thresholds differ are saved (lines 8-13). If both areas differ in exactly one threshold this list will be returned. Otherwise, an empty list is returned (lines 14-15).

Lemma 6.2 *Let $area_a$ and $area_b$ are two different areas with area ID's $area_{id_a}$ and $area_{id_b}$. Then the function*

$$iface.areas\_are\_neighbours(area_{id_a}, area_{id_b})$$

correctly returns a list with all elements in S if $area_a$ and $area_b$ are direct neighbours and an empty list otherwise.

Proof Assume that $area_a$ and $area_b$ are direct neighbours. Then by definition there exist $i \in [n]$ such that $D_i \neq D'_i$ for all data sets $D \in area_a$ and $D' \in area_b$. Consequently, the thresholds in the *ite* data point must differ in at least one attribute, which is considered in lines 4 to 7. Furthermore, since $area_a$ and $area_b$ are direct neighbours we have $|S| = 1$. In other words $S = \{i\}$ or only the *ite* data points always differ in at least one attribute. By considering lines 14 and 15 the function correctly returns a list which does only contain the element i. Note that this element indicates correctly which data point of the data sets differ.
Now assume that $area_a$ and $area_b$ are k-neighbours with $k \geq 2$. By definition there exist a set $S = \{i \in [n] | D_i \neq D'_i : \forall D \in area_a, D' \in area_b\}$ such that $|S| \geq 2$. Therefore, for every $i \in S$ the thresholds in the *ite* data point must differ in at least one attribute, which is considered in lines 4 to 7. By the fact that $|S| \geq 2$, it can be conclude that there are more that one element such that $D_i \neq D'_i$. Consequently, an empty list is returned (lines 14-17). □

Chapter 7

Evaluation on Statistical and Machine Learning Models for Time Series Prediction

In this chapter, we will evaluate and analyze the performance of our prototype. For this purpose, the prototype will be applied to data used for traffic forecasting. In the process, emerging problems will be addressed. Moreover, some interesting characteristics of our prototype will be discussed.

7.1 Traffic Forecasting using Long Short-Term Memory Neural Network

Let a data set that measures the number of cars on four different lanes for every five minutes, as well as the resulting sum of these cars, over one month be given [9]. And split this data set in a training set and a test set. In addition, a long short-term memory (LSTM) model trained on this training data is given [9]. This model predicts the total number of cars in the lanes within the next time step using four time steps directly back in time. Then, by applying this forecasting model to our derived prototype, a DP approximation can be learned based on the training set. Note that the complexity of the structure into which an LSTM model divides the input space is not easily understood. Consequently, classifying the input space according to the underlying structure is not straightforward. Therefore, it is simpler to split the input space into a chessboard-like structure, as described in chapter 6.

Setting. Due to efficiency reasons, each dimension of the four-dimensional input space (one dimension corresponds to exactly one of the four timestamps) is divided into three equal-sized parts. This separation results in a

total of 81 disjoint areas. The test data set consists out of 283 different timestamps in which the total number of cars at one timestamp varies between 0 and 633. We want to protect the identity of one car. Thus, two data sets are considered as neighbours if they differ from each other by exactly one car. Furthermore, the test data are distributed over the input space in such a way that 21 areas contain at least one data element in the test set, (i.e. they are of rank 0). By using the neighbourhood relation described above, the direct neighbours of these rank 0 areas are in total 36 areas which are of rank 1. Moreover, 21 areas are assigned to rank 2 and the remaining 3 areas are assigned to rank 3. Additionally, the sensitivity value is set equal to 10, the lower bound $min\_change = 0.3$, which specifies the amount of desired shifting of the center values per iteration, and the maximal number of iterations in which min_change could not be achieved is is limited by $count_{max} = 3$.

Observation. Let us now describe the results obtained by applying our advanced mechanism on the traffic forecasting data, based on the setting specified above. First of all, the implemented mechanism computes the center values for each area. In the considered example, these values are between 295.901 and 316.431. While calculating the center values, 5388 advanced synchronizations have been performed during 42 iterations over the data set. Next, based on the obtained center values, the mechanism determines the best possible output ranges for each area. These output ranges allow overall a minimum output value of 294.876 and a maximum output value of 317.933. In other words, the calculated approximation of the statistic derived with the LSTM model based on the training data allows a maximum difference of 23.057 within the function values.

Evaluation. Note that the resulting output ranges allow a maximum difference that is significantly larger than the global sensitivity S_{global}. Furthermore, considering that the domain is limited to four dimensions as well as the fact that the domain is partitioned into a relatively small number of areas, a large dependency between the areas in terms of DP-neighbourhoods occurs. Therefore, it can be concluded that due to the limitation of our setting the flexibility was exploited to the best possible extent. Nevertheless, it is of great interest to perform this application in a setting that splits the input space into significantly more areas and uses a smaller threshold value for min_change. However, the number of performed synchronizations rapidly increases with the number of areas. Therefore, when performing such an application, it is strongly recommended to ensure that the computational power of the underlying system is adequate.

Chapter 8

Future Work

In this chapter potential directions for future work are presented.

8.1 Generalize to High-Dimensional Co-Domain

Within this work, a mechanism was derived, which outputs a suitable DP approximation for given data. However, this mechanism is limited by the fact that the output value is contained in a one-dimensional space. In order to broaden the range of applications, it is of great interest to generalize this mechanism in such a way that it can be applied to arbitrary dimensional output spaces. For this purpose, the approach based on our prototype has to be extended to a high-dimensional co-domain. To do this, it is no longer practical to use the L_1-distance as we did in our prototypes, since the distance becomes arbitrarily large with increasing dimension. However, one way to avoid this is by using the L_2-distance. In this setting, it is necessary to redefine the meaning of fulfilling the sensitivity constraints in high-dimensional output space.

8.2 Sustainability

In the previous chapter, our advanced prototype was applied to a traffic forecasting example. By applying this prototype to other examples as well, the functionality of the prototype can be validated. Furthermore, this can identify potential weaknesses that have not yet been considered.

8.3 More Detailed Definition of Neighbourhood Relations for Multivariate Input Data

For extending the former prototype to multivariate input data two data sets are considered to be neighbours if they differ in exactly one data point. For simplicity, we did not distinguish between the number of attributes in which the two data points differ. However, in order to obtain more accuracy for the resulting DP approximation, it is of interest to make such a distinction. More precisely, a more accurate approach would be to characterize two data sets as neighbours if they differ in exactly one attribute in the same data point.

8.4 Group Privacy

Our prototype does not meet the group privacy constraints. In other words, it does not protect the privacy of time-series data which contains sensitive information spread across several data points. It only protects the influence of one data point (i.e., x_i) on the output value. To be able to apply our prototype to a broader class of time series, it is of interest to extend our prototype to meet the group privacy requirements.

Chapter 9

Conclusion

Throughout this work, a more generally applicable and accurate version of a former mechanism that learns a differentially private approximation $M_{I_{train}}$ based on a given publicly available data set I_{train} has been developed and implemented. Furthermore, this advanced mechanism version is adapted to common ML models such as the decision tree.

In prior work, we developed a generic mechanism to learn a DP approximation for high-dimensional input spaces that is both general and problem independent i.e., it provides accurate results while being applicable to different problems [2]. This former mechanism, using a domain-based approach, divides the input space of a statistic g into hypercubes of equal sizes which has the crucial advantage that the DP-neighbours can be identified more efficiently. To the best of our knowledge, this approach has not yet been researched more.

Based on this work, we have derived an advanced version of the mechanism that addresses a number of limitations regarding its first implementation. Concretely, the advanced mechanism derived in this work is now applicable to multivariate input spaces. Furthermore, this mechanism has an optimal abort condition, which minimizes the running time while increasing the accuracy of the resulting DP approximation. The advanced synchronization function implemented in this new mechanism allows us to output an accurate DP approximation even with sparse training data. Moreover, the mechanism can be applied to a larger class of statistics, since it computes the amount of added noise based on the smooth sensitivity which depends on the data set D, besides the statistic g. Finally, we derived an approach to adapt the prototype of this mechanism on structures of more general learning methods like decision trees.

In conclusion, the advanced mechanism constructed in this work is applicable to a broader range of statistics and therefore a first prototype version

applicable to real-world problems.

Appendix A

Appendix

In the following, the implemented advanced prototype for an decision tree learning mechanism, which was derived in chapter 5 as well as in chapter 6 and evaluated in chapter 7, is provided. Please note that within the code the areas in the domain are labelled with nodes. In addition, the statistics of the data points are denoted with f instead of g.

A.1 Problem Instantiations (Model: Decision Tree)

```
import numpy as np
from matplotlib import pyplot as plt
import pandas as pd
import itertools
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn import tree
from keras.models import model_from_json

class InterfaceDecisionTree(DecisionTreeRegressor):
    def __init__(self, hypercube_dim, num_attribute):
        self.hypercube_dim = hypercube_dim
        self.num_attribute = num_attribute

    def number_of_nodes(self):
        if self.hypercube_dim == 0:
            number_of_nodes = 0
        else:
            num_of_nodes = 1
            for i in range(self.hypercube_dim):
                for j in range(self.num_attribute):
                    num_of_nodes *= len(thresholds[i][j])
        return num_of_nodes

    def number_of_dimensions(self): # minimal number of subtrees
        return self.hypercube_dim
```

```
def nodes_are_neighbours(self, node_id_a, node_id_b):

    assert node_id_a != node_id_b #cubes must no overlap

    threshold_a = nodes[node_id_a]['threshold']
    threshold_b = nodes[node_id_b]['threshold']

    equal_thresholds = np.zeros(iface.hypercube_dim)
    for i in range(iface.hypercube_dim):
        equal_thresholds[i] = threshold_a[i] == threshold_b[i]

    non_equal_dimensions = np.arange(iface.hypercube_dim)
                           [np.logical_not(equal_thresholds)]

    return (list(non_equal_dimensions) if len(non_equal_dimensions) == 1
                                       else [])

# Approach for get_node_id()
# 1. given the data_set, find the thresholds which describe the
#    corresponding node
# 2. given the thresholds, find the corresponding node
def get_node_id(self, data_set):
    node_of_data_set = [[1 for _ in range(iface.num_attribute)].copy()
                           for _ in range(iface.hypercube_dim)]
    for dim in range(self.hypercube_dim):
        for attribute in range(self.num_attribute):
            i = 0
            while thresholds[dim][attribute][i] <= data_set[dim][attribute]
                               and i <= len(thresholds[dim][attribute])-1:
                i +=1
            node_of_data_set[dim][attribute] = thresholds[dim][attribute][i]
    for id, node in enumerate(nodes):
        if node_of_data_set == node['threshold']:
            node_id = id
    return node_id

def evaluate_f(self, data_set):
    # this is something like a circular wave originating from 0, with minimum
    # -1 and maximum 1,
    # and multiple up and downs, characterised by SPEED
    SPEED = 4
    return np.sin(np.pi * np.linalg.norm(data_set, ord=2) * SPEED )

def generate_data(self, length, seed):
    # helper function
    np.random.seed(seed)
    return np.random.uniform(low=0, high=1, size=(length, self.hypercube_dim,
                                                  self.num_attribute))

# note dim = 3 is dimension 4. In general dim = n-1 represents the tree for
# dimension n
def build_tree(self, data_X):
    X = self.bring_data_in_valid_form(data_X)
    y = [self.evaluate_f(data_set) for data_set in data_X]
```

```
        clf = DecisionTreeRegressor(random_state=0)
        #max_leaf_nodes = 2 ** self.num_attribute, random_state=0)
        fitted_tree = clf.fit(X, y)
        return fitted_tree

    def bring_data_in_valid_form(self, data_X):
        X = np.array([data_set for data_set in data_X])
        nsample, n_dim ,n_attribute = X.shape
        X = X.reshape((nsample, n_dim * n_attribute))
        return X

iface = InterfaceDecisionTree(hypercube_dim=3, num_attribute = 2)
data = iface.generate_data(length=10, seed=10)

#creating the tree
decisiontree = iface.build_tree(data)

# return the set of leaves of a given binary tree
def classified_node_ids(decisiontree):

    set_of_leaves = []
    splitting_nodes = []
    path_to_leaf = [0]

    n_nodes_in_tree = decisiontree.tree_.node_count
    children_left = decisiontree.tree_.children_left
    children_right = decisiontree.tree_.children_right

    node_depth = np.zeros(shape=n_nodes_in_tree, dtype=np.int64)
    is_leaves = np.zeros(shape=n_nodes_in_tree, dtype=bool)
    stack = [(0, 0)]  # start with the root node id (0) and its depth (0)
    while len(stack) > 0:
        # `pop` ensures each node is only visited once
        node_id, depth = stack.pop()
        node_depth[node_id] = depth

        # If the left and right child of a node is not the same we have a split
        # node
        is_split_node = children_left[node_id] != children_right[node_id]
        # If a split node, append left and right children and depth to `stack`
        # so we can loop through them
        if is_split_node:
            stack.append((children_left[node_id], depth + 1))
            stack.append((children_right[node_id], depth + 1))
        else:
            is_leaves[node_id] = True

    for i in range(n_nodes_in_tree):
        if is_leaves[i]:
            set_of_leaves.append(i)
        else:
            splitting_nodes.append(i)

        # check if all leaves are contained in set_of_leaves
        # assert len(set_of_leaves) == tree.get_n_leaves()
```

```
    return set_of_leaves, splitting_nodes

# assign all splitting nodes to one dimension
# IMPORTANT to find all neighbour nodes

leave_nodes, splitting_nodes = classified_node_ids(decisiontree)

dim_of_splitting_node = []
attribute_of_splitting_node = []
threshold_of_splitting_node = []

# note: dimension n is stored as n-1 ind dim_of_splitting_node
#       (same for attribute_of_splitting_node)

for id in splitting_nodes:
    # find cooresponding threshold which is relevant for the splitting node
    threshold = decisiontree.tree_.threshold[id]
    threshold_of_splitting_node.append(threshold)
    # find corresponding dimension which is relevant for the splitting node
    dim = int(decisiontree.tree_.feature[id] / iface.num_attribute)
    dim_of_splitting_node.append(dim)
    # find corresponding attribute which is relevant for the splitting node
    attribute = (((decisiontree.tree_.feature[id] -
    ((dim+1)*iface.num_attribute)))%iface.num_attribute)
    attribute_of_splitting_node.append(attribute)

# find all thresholds in each dimension for each attribute
# assume that each attribute takes a value between [0,1]

# thresholds is a matrix with column = #attributes and rows = #dimensions
thresholds = [[None for _ in range(iface.num_attribute)].copy()
              for _ in range(iface.hypercube_dim)]

for dim in range(iface.hypercube_dim):
    for attribute in range(iface.num_attribute):
        threshold_in_dim_and_attribute = [1]
        for i, node_id in enumerate(splitting_nodes):
            if dim == dim_of_splitting_node[i]
               and attribute == attribute_of_splitting_node[i]:
                threshold_in_dim_and_attribute.append(threshold_of_splitting_node[i])
        thresholds[dim][attribute] = sorted(threshold_in_dim_and_attribute)

# Given a leaf id i, get_path(i) returns path (from root to leaf i)
# Output is a list with tree_node_ids of increasing depth (starts with root and
# ends with leaf node i)
# if k is the id of an internal node, get_path(k) will return None

def allpaths(start_node):
    # a class method which finds all the tree paths from the root to the leaf
    if start_node == None:
        return []
    elif start_node in leave_nodes: # in set of leaf nodes
        return [start_node]
```

```
    else:
        return [str(start_node) + "-->" + str(l) for l in
                allpaths(decisiontree.tree_.children_left[start_node])
                 + allpaths(decisiontree.tree_.children_right[start_node])]

def get_path(k):
    """tree method to get the desired path from root to leaf"""
    l = allpaths(0)
    for x in l:
        my_list = [int(y) for y in x.split("-->")]
        if my_list[-1] == k:
            return(my_list)
        else:
            pass

# returns if the direction in which the given path continues after each
# splitting node
# True : left child - splitting threshold is respected
# False: right child - splitting threshold is exceeded
# note: this function works only for binary trees
def splitting_node_direction(path):

    assert path != None # input path is not valid

    decisions = []
    for i, id in enumerate(path[:len(path)-1]):
        if decisiontree.tree_.children_left[id] == path[i+1]:
            decisions.append(True)
        else:
            decisions.append(False)
    return(decisions)
```

A.2 Initialization

```
dim = iface.number_of_dimensions()
S = 0.6
EPOCHS = 500  # at least len(data)
max_syn = 3
update_step = S / (len(data))
buffer_value = S/(2*update_step)
max_update_steps = iface.number_of_dimensions()
max_width = S/2
count_syn = 0
count = 0
count_max = 2 # S/update_step
break_out_flag = False
break_out_flag_2 = False
minimal_change = 0.0001
forced_syn = True
stop_syn1 = False
count_sny2 = 0

# finally, nodes will contain {'center_val': value that has to be contained
```

```
# in all neighbouring intervals, 'neighbours': a (dim x undef) list with node
# indexes that are direct neighbours. }
nodes = []

# build up node map and instantiate all nodes with start values
start_f_val = np.mean([iface.evaluate_f(d) for d in data[:100] ])
# take average of first 100 values
discret_start_f_val = round(start_f_val/update_step)

### range(iface.number_of_nodes()) = {0,...,iface.number_of_nodes()-1}

for i in range(iface.number_of_nodes()):
    nodes.append({'center_val': discret_start_f_val,
                  'lower_buffer': buffer_value, 'upper_buffer': buffer_value})

# create a list all possible combinations of node IDs and assign each of them
# to a node:

# find all combinations of thresholds
dimension_combination = []

# find all combinations per dimensions
for dim in range(iface.hypercube_dim):
    dimension_combination.append(list(itertools.product(*thresholds[dim][:])))

threshold_combinations = list(itertools.product(*dimension_combination))

for i, node in enumerate(nodes):
    threshold_node = [list(x) for x in threshold_combinations[i]]
    # save it as list of lists instead as tuple of tuples
    node['threshold'] = threshold_node

# build up node <--> data_point reference for later use
node_of_data_set = []
for data_set in data:
    node_ids = iface.get_node_id(data_set)
    node_of_data_set.append(node_ids)

# compute f value for each datapoint
f_of_data_set = []
for data_set in data:
    f_of_data_set.append(iface.evaluate_f(data_set))

# build up two lists one which contains all DP-neighbours
# (nodes that share an interval in any dimension)
# now one node_id has the length of num_attribute

for id_a, node in enumerate(nodes):

    neighbours = [[] for _ in range(iface.number_of_dimensions()) ]
    for id_b in range(iface.number_of_nodes()):

        if id_a == id_b:
            # we do not want the node itself listed as neighbour
            continue
```

```
        else:
            neighbour_dims = iface.nodes_are_neighbours(id_a, id_b)
            [ neighbours[dim].append(id_b) for dim in neighbour_dims ]

    node['neighbours'] = neighbours
```

A.2.1 Assign Nodes to Ranks

```
adapted_nodes_total = [] # to save all adjusted nodes with arbitrary rank
adapted_nodes_rank0 = [] # to save all adjusted nodes with rank 0
                         # i.e. nodes which contain a trainingpkt
adapted_nodes_rank1 = [] # to save all adjusted nodes with rank 1
adapted_nodes_rank2 = [] # to save all adjusted nodes with rank 2
adapted_nodes_rank3 = [] # to save all adjusted nodes with rank 3
adapted_nodes_rank4 = [] # to save all adjusted nodes with rank 4
adapted_nodes_rank5 = [] # to save all adjusted nodes with rank 5

# collect all nodes with rank 0
for i_d, _ in enumerate(data):
    node_id = node_of_data_set[i_d]
    node = nodes[node_id]
    if node_id not in adapted_nodes_total:
        adapted_nodes_total.append(node_id)
        adapted_nodes_rank0.append(node_id)

# collect all nodes of rank 1
for i_d, _ in enumerate(data):
    node_id = node_of_data_set[i_d]
    node = nodes[node_id]
    for dim in range(iface.number_of_dimensions()):
        for n_node_rk1_id in node['neighbours'][dim]:
            n_node_rk1 = nodes[n_node_rk1_id]

            if n_node_rk1_id not in adapted_nodes_total:
                adapted_nodes_total.append(n_node_rk1_id)
                adapted_nodes_rank1.append(n_node_rk1_id)

# collect all nodes of rank 2
for n_node_rk1_id in adapted_nodes_rank1:
    for dim2 in range(iface.number_of_dimensions()):
        n_node_rk1 = nodes[n_node_rk1_id]
        for n_node_rk2_id in n_node_rk1['neighbours'][dim2]:
            n_node_rk2 = nodes[n_node_rk2_id]

            if n_node_rk2_id not in adapted_nodes_total:
                adapted_nodes_total.append(n_node_rk2_id)
                adapted_nodes_rank2.append(n_node_rk2_id)

# collect all nodes of rank 3
for n_node_rk2_id in adapted_nodes_rank2:
    for dim3 in range(iface.number_of_dimensions()):
        n_node_rk2 = nodes[n_node_rk2_id]
        for n_node_rk3_id in n_node_rk2['neighbours'][dim3]:
```

```
            n_node_rk3 = nodes[n_node_rk3_id]

            if n_node_rk3_id not in adapted_nodes_total:
                adapted_nodes_total.append(n_node_rk3_id)
                adapted_nodes_rank3.append(n_node_rk3_id)

# collect all nodes of rank 4
for n_node_rk3_id in adapted_nodes_rank3:
    for dim4 in range(iface.number_of_dimensions()):
        n_node_rk3 = nodes[n_node_rk3_id]
        for n_node_rk4_id in n_node_rk3['neighbours'][dim4]:
            n_node_rk4 = nodes[n_node_rk4_id]

            if n_node_rk4_id not in adapted_nodes_total:
                adapted_nodes_total.append(n_node_rk4_id)
                adapted_nodes_rank4.append(n_node_rk4_id)

# collect all nodes of rank 5 or higher
for id,_ in enumerate(nodes):
    if id not in adapted_nodes_total:
        adapted_nodes_total.append(id)
        adapted_nodes_rank5.append(id)
```

A.3 Helper Functions

```
### evaluate f in the computed sensitivity bound ###
def evaluate_f_hat(data_set, nodes, iface):
    node_ids = iface.get_node_id(data_set)
    node_id_index = iface.node_id_to_index(node_ids)
    lower_bound, upper_bound = nodes[node_id_index]['bounds']

    f = iface.evaluate_f(data_set)

    if f > upper_bound:
        return upper_bound
    elif f < lower_bound:
        return lower_bound
    return f

def evaluate_lower_bound(data_set, nodes, iface):
    node_ids = iface.get_node_id(data_set)
    node_id_index = iface.node_id_to_index(node_ids)
    lower_bound, _ = nodes[node_id_index]['bounds']

    return lower_bound

def evaluate_upper_bound(data_set, nodes, iface):
    node_ids = iface.get_node_id(data_set)
    node_id_index = iface.node_id_to_index(node_ids)
    _, upper_bound = nodes[node_id_index]['bounds']

    return upper_bound
```

```
def update_buffers(old_buffer_nodes_as_ids, new_buffer_nodes, update_step, max_width):

    for node_id in old_buffer_nodes_as_ids:
        node = nodes[node_id]
        new_upper_buffer = [node['upper_buffer']]
        new_lower_buffer = [node['lower_buffer']]
        for dim in range(iface.number_of_dimensions()):
            for n_node_id in node['neighbours'][dim]:
                n_node = nodes[n_node_id]
                if n_node in new_buffer_nodes:
                    if (node['center_val']-n_node['center_val'])* update_step
                        > max_width:
                        new_lower_buffer.append(n_node['upper_buffer'])
                    elif (n_node['center_val']-node['center_val'])* update_step
                         > max_width:
                        new_upper_buffer.append(n_node['lower_buffer'])
        # switch of lower <-> upper is intentional
        if new_upper_buffer != []:
            node['lower_buffer'] = min(new_upper_buffer)
        if new_lower_buffer != []:
            node['upper_buffer'] = min(new_lower_buffer)

    return

def advanced_synchronization(max_width, buffer, nodes_rank1, nodes_rank2,
                             nodes_rank3, update_step):


    # adjust the buffers of all nodes of rank 1 (since we have already changed the
    # buffers by shifting the center values of nodes with rank 1)
    # find valued upper/lower buffer for node of rank 2

    update_buffers(nodes_rank1, nodes, update_step, max_width)
    update_buffers(nodes_rank1, nodes, update_step, max_width)


    # adjust the buffers of all nodes of rank 2 (since we have already changed the
    # buffers by shifting the center values of nodes with rank 1)
    # find valued upper/lower buffer for node of rank 2

    update_buffers(nodes_rank2, nodes_rank1, update_step, max_width)
    update_buffers(nodes_rank2, nodes_rank1, update_step, max_width)


    # RANK 2: update all center values of nodes with rank 2
    for node_rk2_id in nodes_rank2:
        node_rk2 = nodes[node_rk2_id]
        break_out_flag1 = False
        # mean value of all nghb. center values of rank 1:
        # note that every node of rank 2 has at least one nghb node of rank 1
        # (otherwise it would be of rank 3)
        rank1_nghb_centerval = []
        rank1_nghbs = []
        for dim in range(iface.number_of_dimensions()):
```

```
        for n_node_rk1_id in node_rk2['neighbours'][dim]:
            n_node_rk1 = nodes[n_node_rk1_id]
            if n_node_rk1_id in nodes_rank1:
                rank1_nghb_centerval.append(n_node_rk1['center_val'])
                rank1_nghbs.append(n_node_rk1)
                mean_nghb_centerval = round(sum(rank1_nghb_centerval)
                                            /len(rank1_nghbs))
            # if we apply this syn2 on nodes of rank2, rank3, rank3 it
            # could happen that rank1_nghb_centerval is empty!!
    if rank1_nghb_centerval == []:
        mean_nghb_centerval = node_rk2['center_val']
        continue

    dist1 = mean_nghb_centerval - node_rk2['center_val']

    if abs(dist1 * update_step) <= update_step/2:
        continue

    # if possible: shift center_val by step towards the mean value of its
    # nghb of rank 1 + adjust the lower and upper buffer
    if min(node_rk2['lower_buffer']-np.sign(dist1),
           node_rk2['upper_buffer']+np.sign(dist1)) >= 0:
        node_rk2['center_val'] += np.sign(dist1)

        # update all neighboring center_vals that are outside of
        # center_val +- max_dist + adjust the lower and upper buffer
        # note that this neighboring notes are probably nodes
        # of rank 2 or 3 not of rank 1

        upper_list1 = [min(node_rk2['upper_buffer']+np.sign(dist1),buffer)]
        lower_list1 = [min(node_rk2['lower_buffer']-np.sign(dist1),buffer)]
        adapted_nodes1 = [] # to save all adjusted ngbh nodes
        adapted_lower_buffer_nodes1 = []
        adapted_upper_buffer_nodes1 = []
        for dim in range(iface.number_of_dimensions()):
            for n_node_rk1_id in node_rk2['neighbours'][dim]:
                n_node_rk1 = nodes[n_node_rk1_id]
                if abs((node_rk2['center_val'] - n_node_rk1['center_val'])
                        * update_step) > max_width:
                    if n_node_rk1['lower_buffer']-np.sign(dist1) >= 0 and
                       n_node_rk1['upper_buffer']+ np.sign(dist1) >= 0:
                        update(buffer,node_rk2, n_node_rk1,dist1,update_step,
                               adapted_lower_buffer_nodes1,
                               adapted_upper_buffer_nodes1,nodes_rank3)
                        adapted_nodes1.append(n_node_rk1)

                    else:
                        # Undo previous steps to maintain consistency
                        undo_update(buffer, node_rk2, adapted_nodes1,
                                    adapted_lower_buffer_nodes1,
                                    adapted_upper_buffer_nodes1,
                                     dist1, update_step)
                        break_out_flag1 = True
                        break
```

```
                    if (node_rk2['center_val']-n_node_rk1['center_val'])
                        * update_step > max_width:
                        lower_list1.append(n_node_rk1['lower_buffer'])
                    elif (n_node_rk1['center_val']-node_rk2['center_val'])
                          * update_step > max_width:
                        upper_list1.append(n_node_rk1['upper_buffer'])


                if (break_out_flag1 == True):
                    break
            if (break_out_flag1 == False):
                # switch of lower <-> upper is intentional
                if upper_list1 != []:
                    node_rk2['lower_buffer'] = min(upper_list1)

                if lower_list1 != []:
                    node_rk2['upper_buffer'] = min(lower_list1)


        ########## update all buffers of nodes with rank 1,2 & 3 ########
        # update buffers of rank 2 (Buffers of nodes with rank 2 are
        # up-to-date/adjusted)
        # to be sure, update all buffers of nodes with rank 2 depending on
        # nodes of rank 2

        update_buffers(nodes_rank2, nodes_rank2, update_step, max_width)
        update_buffers(nodes_rank2, adapted_nodes_total, update_step, max_width)
        update_buffers(nodes_rank2, adapted_nodes_total, update_step, max_width)

        # update buffers of nodes with rank 3
        update_buffers(nodes_rank3, nodes_rank2, update_step, max_width)
        update_buffers(nodes_rank3, adapted_nodes_total, update_step, max_width)


        # update Buffers of nodes with rank 1
        update_buffers(nodes_rank1, nodes_rank2, update_step, max_width)
        update_buffers(nodes_rank1, adapted_nodes_total, update_step, max_width)

    return

def simple_syn(buffer):

    current_center_val = []
    for node in nodes:
        current_center_val.append(node['center_val'])
    new_center_val = int(np.mean(current_center_val))
    for node in nodes:
        node['center_val'] = new_center_val
        node['upper_buffer'] = buffer
        node['lower_buffer'] = buffer
    return

def update(buffer,current_node,ngbh_node,distance,update_step,
           modified_lower_buffer_nghb_nodes,modified_upper_buffer_nghb_nodes,
```

```
            adapted_nodes_rank_bigger_ids):

        ngbh_node['center_val'] += np.sign(distance)


        if ngbh_node['lower_buffer'] - np.sign(distance) <= buffer:
            ngbh_node['lower_buffer'] -= np.sign(distance)
            modified_lower_buffer_nghb_nodes.append(ngbh_node)
        else:
            ngbh_node['lower_buffer'] = buffer

        if ngbh_node['upper_buffer'] + np.sign(distance) <= buffer:
            ngbh_node['upper_buffer'] += np.sign(distance)
            modified_upper_buffer_nghb_nodes.append(ngbh_node)
        else:
            ngbh_node['upper_buffer'] = buffer

        # adjust the buffers of nghb nodes of nghb_node, in case their have rank
        # equal or smaller than ngbh_node
        for dim1 in range(iface.number_of_dimensions()):
            for ngbh_ngbh_node_id in ngbh_node['neighbours'][dim]:
                ngbh_ngbh_node = nodes[ngbh_ngbh_node_id]
                if ngbh_ngbh_node not in [current_node,ngbh_node]:
                    if ngbh_ngbh_node_id not in adapted_nodes_rank_bigger_ids:
                        update_buffers([ngbh_ngbh_node_id], nodes,
                                                            update_step, max_width)
          return

    def undo_update(buffer, current_node, modified_nghb_nodes,
                    modified_lower_buffer_nghb_nodes,
                    modified_upper_buffer_nghb_nodes, distance, update_step):

        current_node['center_val'] -= np.sign(distance)
        # include all neighbors, which we have already modified
        for modified_nghb_node in modified_nghb_nodes:
            modified_nghb_node['center_val'] -= np.sign(distance)

            if modified_nghb_node in modified_lower_buffer_nghb_nodes:
                    modified_nghb_node['lower_buffer']
                            = min(modified_nghb_node['lower_buffer']
                                  + np.sign(distance), buffer)

            if modified_nghb_node in modified_upper_buffer_nghb_nodes:
                    modified_nghb_node['upper_buffer']
                            = min(modified_nghb_node['upper_buffer']
                                  - np.sign(distance), buffer)

        return

    def termination_fct(old_cv,new_cv,min_change,count,count_max,flag):

        change = np.mean(abs(np.array(old_cv) - np.array(new_cv)))
        if min_change > change *(len(data)/len(adapted_nodes_total)):
            if count <= count_max:
                count += 1
```

```
        else:
            flag = True

    return(count,flag)
```

A.4 Algorithm

A.4.1 Finding Suitable Center Values

```
for epoche in range(EPOCHS):

    # center_values before
    old_center_values = []
    new_center_values = []
    if new_center_values == []:
        for node in nodes:
            old_center_values.append(node['center_val'])
    else:
        old_center_values = new_center_values

    for i_d, _ in enumerate(data):
        break_out_flag = False
        break_out_flag_2 = False
        node_id = node_of_data_set[i_d]
        node = nodes[node_id]
        distance = f_of_data_set[i_d] - node['center_val'] * update_step

        # we have to accept an inaccuracy of update_step/2
        if abs(distance) <= update_step/2:
            continue

        # if possible: shift center_val by step towards its optimal value
        # and adjust the lower and upper buffer
        if min(node['lower_buffer']-np.sign(distance),node['upper_buffer']
               +np.sign(distance)) >= 0:
            node['center_val'] += np.sign(distance)
            node['lower_buffer']= min(node['lower_buffer']-np.sign(distance),
                                      buffer_value)
            node['upper_buffer']= min(node['upper_buffer']+np.sign(distance),
                                      buffer_value)

            # update all neighboring center_vals that are outside of center_val
            # +- max_dist and adjust the lower and upper buffer
            # upper_list = []
            # lower_list = []
            adapted_nodes = [] # to save all adjusted ngbh nodes
            adapted_lower_buffer_nodes = []
            adapted_upper_buffer_nodes = []

            for dim in range(iface.number_of_dimensions()):
                for n_node_id in node['neighbours'][dim]:
                    n_node = nodes[n_node_id]
                    if abs((n_node['center_val']
```

```
        - node['center_val']) * update_step)
       > max_width:
    if n_node['lower_buffer']-np.sign(distance)>= 0 and
       n_node['upper_buffer'] +np.sign(distance) >= 0:

        adapted_nodes.append(n_node)

        # update function
        update(buffer_value,node,n_node,distance,
               update_step, adapted_lower_buffer_nodes,
               adapted_upper_buffer_nodes,
            adapted_nodes_rank2)

    else:

        if adapted_nodes_rank2 != []:

            # undo all updates of the current node to get
            # consistency
            undo_update(buffer_value,node,adapted_nodes ,
                        adapted_lower_buffer_nodes,
                        adapted_upper_buffer_nodes,
                        distance,update_step)

            advanced_synchronization(max_width,
            buffer_value, adapted_nodes_rank0,
            adapted_nodes_rank1, adapted_nodes_rank2,
            update_step)
            advanced_synchronization(max_width,
            buffer_value, adapted_nodes_rank1,
            adapted_nodes_rank2, adapted_nodes_rank3,
            update_step)
            advanced_synchronization(max_width,
            buffer_value, adapted_nodes_rank2,
            adapted_nodes_rank3, adapted_nodes_rank4,
            update_step)
            advanced_synchronization(max_width,
            buffer_value, adapted_nodes_rank3,
            adapted_nodes_rank4, adapted_nodes_rank5,
            update_step)
            advanced_synchronization(max_width,
            buffer_value, adapted_nodes_rank1,
            adapted_nodes_rank2, adapted_nodes_rank3,
            update_step)
            forced_syn = False
            break_out_flag = True
            count_sny2 += 1

        elif count_syn < count_max:
            simple_syn(buffer_value)
            break_out_flag = True
            count_syn += 1

        else:
```

```
                            # undo all updates of the current node to get
                            # consistency
                            undo_update(buffer_value,node,adapted_nodes,
                                        adapted_lower_buffer_nodes,
                                        adapted_upper_buffer_nodes,
                                        distance, update_step)
                            stop_syn1 = True

                if (break_out_flag == True):
                    break

                elif (stop_syn1 == True):
                    break

            # find valued upper/lower buffer for node
            update_buffers([node_id], nodes, update_step, max_width)

        if (break_out_flag == True):
            continue

        elif (stop_syn1 == True):
            break

    # center_values after
    for node in nodes:
        new_center_values.append(node['center_val'])

    (count, break_out_flag_2) = termination_fct(old_center_values, new_center_values,
                                                minimal_change, count, count_max,
                                                break_out_flag_2)

    if count >= count_max -1 and forced_syn == True:
        forced_syn = False
        advanced_synchronization(max_width, buffer_value, adapted_nodes_rank0,
        adapted_nodes_rank1, adapted_nodes_rank2, update_step)
        advanced_synchronization(max_width, buffer_value, adapted_nodes_rank1,
        adapted_nodes_rank2, adapted_nodes_rank3, update_step)
        advanced_synchronization(max_width, buffer_value, adapted_nodes_rank2,
        adapted_nodes_rank3, adapted_nodes_rank4, update_step)
        advanced_synchronization(max_width, buffer_value, adapted_nodes_rank3,
        adapted_nodes_rank4, adapted_nodes_rank5, update_step)
        advanced_synchronization(max_width, buffer_value, adapted_nodes_rank1,
        adapted_nodes_rank2, adapted_nodes_rank3, update_step)
        count_sny2 += 1

    if (break_out_flag_2 == True):
        break

    elif (stop_syn1 == True):
        break
```

A.4.2 Finding Output Ranges

Construction of Smoothed Sensitivity Function

```
gamma = S/5

# 1. Find/Define the local sensitivity for each node and set the
#    initial smooth sensitivity = global sensitivity
for node in nodes:
    node['smooth_sensitivity'] = S
    node['local_sensitivity'] = 0
    for dim in range(iface.number_of_dimensions()):
        for n_index in node['neighbours'][dim]:
            n_node = nodes[n_index]
            node['local_sensitivity'] = max(node['local_sensitivity'],
                                            abs(node['center_val']
                                                - n_node['center_val'])
                                                * update_step)

# 2. Define a suitable smooth sensitivity
unfinished_nodes = range(len(nodes))
final_nodes = []
i = 1
while len(nodes) != len(final_nodes):
    # elements in list unfinished_nodes that are not in list final_node
    for node_index in np.setdiff1d(unfinished_nodes,final_nodes):
        node = nodes[node_index]
        if S - node['local_sensitivity'] >= i * gamma:
            node['smooth_sensitivity'] -= gamma
        else:
            final_nodes.append(node_index)

    for node_index in final_nodes:
        node = nodes[node_index]
        for dim in range(iface.number_of_dimensions()):
            for n_index in node['neighbours'][dim]:
                if n_index not in final_nodes:
                    final_nodes.append(n_index)
    i += 1
```

Computing Output Ranges

```
for node in nodes:

    da_intervals = []
    for dim in range(iface.number_of_dimensions()):
        da_vals = np.array([nodes[n_index]['center_val']
                        for n_index in node['neighbours'][dim] ])
        da_vals = np.append(node['center_val'],da_vals)
        da_interval = [np.min(da_vals), np.max(da_vals)]

        # extend to S
        diff = da_interval[1] - da_interval[0]
        # MOST IMPORTANT SANITY CHECK in terms of DP
```

```
        assert diff*update_step <= node['smooth_sensitivity']
        assert diff >= 0

        # here we approximate optimal solution by extending intervals symmetrically
        da_interval[0] -= (int(node['smooth_sensitivity']/update_step + 1/2) - diff)/ 2
        da_interval[1] += (int(node['smooth_sensitivity']/update_step + 1/2) - diff)/ 2

        da_intervals.append(da_interval)

    da_intervals = np.array(da_intervals)
    # maximal lower bound of all neighbouring cells
    lower_bound = np.max(da_intervals[:,0])* update_step
    # minimal upper bound of all neighbouring cells
    upper_bound = np.min(da_intervals[:,1])* update_step

    assert lower_bound <= upper_bound, (lower_bound, upper_bound)

    node['bounds'] = (lower_bound, upper_bound)
```

Bibliography

[1] Francesco Alda and Benjamin I.P. Rubinstein. *The Bernstein Mechanism: Function Release under Differential Privacy*. AAAI-17, 2017.

[2] Nadja Aoutouf. *Research Project: Learning differential privacy mechanisms with high-dimensional input*. 2021.

[3] C. Dwork, F. McSherry, K. Nissim, and A. Smith. *Calibrating Noise to Sensitivity in Private Data Analysis*. Springer-Verlag Berlin Heidelberg, 2006.

[4] C. Dwork and A. Roth. *The Algorithmic Foundations of Differential Privacy*. Now Foundations and Trends, 2014.

[5] I. Grujic, S. Bogdanovic-Dinic, and L. Stoimenov. *Collecting and Analyzing Data from E-Government Facebook Pages*. ICT Innovations 2014 Web Proceedings ISSN 1857-7288, 2014.

[6] R. B. HUBERT, E. ESTEVEZ, and A. MAGUITMAN. *Analyzing and Visualizing Government-Citizen Interactions on Twitter to Support Public Policy-making*. Universidad Nacional del Sur Bahía Blanca, Argentina, 2020.

[7] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning with Applications in R*. Springer New York Heidelberg Dordrecht London, 2013.

[8] K. Nissim, S. Raskhodnikova, and A. Smith. *Smooth Sensitivity and Sampling in Private Data Analysis*. ACM, 2007.

[9] Peng Zheng. Predict Trafficflow by LSTM in Keras. https://github.com/ZhengPeng7/Predict_Trafficflow_by_LSTM_in_Keras. [Online; last accessed 23-September-2021].

[10] Sheila Zingg. *Research Project: A Novel, General, Problem-Dependent Noising Strategy for Differential Privacy*. 2020.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Learning advanced differential privacy mechanisms with fixed data order

Authored by (in block letters):

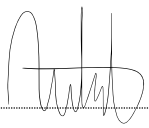
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Aoutouf	Nadja

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 30.09.2021	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.