Harvard Privacy Tools Project

Data Processing Notes

Contents

1 Overview

This document contains notes on our choices regarding "data processing", which I am using to stand in for all things related to data bookkeeping (data bounds, imputation, sample size calculations, etc.).

2 Clamping

2.1 Introduction

Many operations in the library require the definition of a set of possible data values we would like to use for said operation. For numeric variables, this typically takes the form of a closed interval $[min, max] \in \mathbb{R}$ or $\mathbb{Z}$. For categorical variables, this is a discrete set of elements. The *clamp* component sets these properties for a given analysis. In our system, we have slightly different notions of clamping for numeric and categorical variables. We consider $f64$ and $i64$ to be allowable numeric types, and $i64, bool$, and $String$ to be allowable categorical types.

Our method for clamping numeric types should be familiar; it requires data, a lower bound, and an upper bound, and maps data elements outside of the interval $[min, max]$ to the nearest point in the interval. Null values (represented as NAN for the $f64$ type and which do not exist for the $i64$ type) are preserved by numeric clamping.

Our categorical clamping requires data, a set of feasible values, and a null value. Data elements that are outside the set of feasible values are mapped to the null value. Null values are preserved by the categorical clamping.

2.2 Algorithm Statement

The component arguments are as follows:

1. X: The data of size $n \times m$ to be clamped.
2. categories: A vector of length m, where each element is the set of plausible data values for column m of X. If not provided, defaults to None.
3. null_value: A vector of length m, where each element is the designated "null" for column m of X. That is, elements $\notin$ categories are mapped to the null_value. If not provided, defaults to None.
4. lower: A vector of length m, where each element is a proposed minimum value for column m of X. If not provided, defaults to None.
5. upper: A vector of length m, where each element is a proposed maximum value for column m of X. If not provided, defaults to None.

Algorithm 1 Clamping: clamp(X, categories, lower, upper)

for $j \in [m]$ **do** ▷ Iterate over columns
  for $i \in [n]$ **do** ▷ Iterate over rows
    if categories $\neq$ None **then** ▷ Proceed with categorical clamping
      if $X_{i,j} \notin categories[j]$ **then**
        $X_{i,j} \leftarrow null\_value[j]$
      end if
    else ▷ Proceed with numeric clamping
      if $X_{i,j} < lower[j]$ **then**
        $X_{i,j} \leftarrow lower[j]$
      else if $X_{i,j} > upper[j]$ **then**
        $X_{i,j} \leftarrow upper[j]$
      end if
    end if
  end for
end for
return X

3 Imputation

3.1 Introduction

Some operations in the library require the private data to be non-null. The *impute* component sets the non-null property for an analysis.

Much like clamping, we have slightly different notions of imputation for numeric and categorical variables. In this case, we consider only $f64$ to be an allowable numeric type, while $f64, i64, bool$, and $String$ are allowable categorical types. We do not allow for numeric imputation on $i64$ because $i64$ does not support NAN values and thus are non-null by default.

Numeric imputation is parameterized by min, max, and a supported probability distribution (Gaussian or Uniform) with appropriate arguments. NAN values are replaced with a draw from the provided distribution.

Categorical imputation is parameterized by categories, probabilities, and a null value. Each data element equal to the null value is replaced with a draw from the set of categories, according to the probabilities provided.

3.2 Algorithm Statement

The component arguments are as follows:

1. X: The data of size $n \times m$ for which values should be imputed.
2. categories: A vector of length m, where each element is the set of plausible data values for column m of X. If not provided, defaults to None.
3. weights: A vector of length m, where each element is a set of imputation weights associated with the categories for column m of X. If not provided, defaults to None.
4. null_value: A vector of length m, where each element is the designated "null" for column m of X. If an element is equal to the corresponding null_value for its column, it is imputed from the probability distribution over the categories defined by the weights.

5. distribution: A string providing the distribution from which elements will be imputed. Currently supported options are "Uniform" and "Gaussian". Used only if *categories* == None. Defaults to "Uniform".
6. shift: A vector of length m, where each element is the expectation of the Gaussian distribution used for imputing missing values in column m of X. Used only if *distribution* == "Gaussian". Defaults to None.
7. scale: A vector of length m, where each element is the standard deviation of the Gaussian distribution used for imputing missing values in column m of X. Used only if *distribution* == "Gaussian". Defaults to None.
8. lower: A vector of length m, where each element is a proposed minimum value for column m of X. Acts as a lower bound for the distribution from which elements are imputed. If not provided, defaults to None.
9. upper: A vector of length m, where each element is a proposed maximum value for column m of X. Acts as a lower bound for the distribution from which elements are imputed. If not provided, defaults to None.

Let Categorical$(k, (p_1, \ldots, p_k))$ be the categorical distribution, which chooses a single element from the set $\{1, 2, \ldots, k\}$ where each element i has probability of being chosen p_i. Then we have

Algorithm 2 Imputation:
impute(X, categories, weights, null_value, distribution, shift, scale, lower, upper)

```
for j ∈ [m] do                                              ▷ Iterate over columns
    for i ∈ [n] do                                          ▷ Iterate over rows
        if categories ≠ None then                           ▷ Proceed with categorical imputation
            if X_{i,j} == null_value[j] then
                sampling_index ← Categorical(|weights[j]|, weights[j])
                X_{i,j} ← categories[j][sampling_index]
            end if
        else                                                ▷ Proceed with numeric imputation
            if X_{i,j} == NAN then
                if distribution == "Uniform" then
                    X_{i,j} ← Uniform(lower[j], upper[j])
                else if distribution == "Gaussian" then
                    X_{i,j} ← Gaussian(shift[j], scale[j])
                    if X_{i,j} < lower[j] then
                        X_{i,j} ← lower[j]
                    else if X_{i,j} > upper[j] then
                        X_{i,j} ← upper[j]
                    end if
                end if
            end if
        end if
    end for
end for
return X
```

4 Resize

4.1 Motivation

We want our library to support cases both in which the analyst does and does not have access to the number of records in the data. Doing the latter well is an under-explored problem, so we propose the *resize* framework as a possible solution.

4.2 Goals

The *resize* framework is a means of jointly achieving a few different goals:

1. guarantee known n for analyses that require it,
2. give users flexibility in how they trade off between bias and variance, and
3. allow for both c-stability and privacy amplification from subsampling within the same privacy calculus.

4.3 Algorithm Statement

The *resize* function takes the following inputs:

1. X: The private underlying data.
2. $\tilde{n}$: The size of the private underlying data.
3. n: The desired size of the new data.
4. p: The proportion of the underlying data that can be used to construct the new data. p can be > 1.
5. ...: Various arguments explaining imputation rules (these will follow the rules from section 3).

Let $sample(Y, m)$ be a function that samples m elements from data set Y without replacement. Let $Aug(Y, m, \ldots)$ be a function that imputes new elements independent of the data (using imputation parameters given by $\ldots$) for a data set Y until it is of size m. The algorithm will look something like the following:

Algorithm 3 Resize: resize(X, n, p, neighboring, ...)

$c \leftarrow \lceil p \rceil$ ▷ sets c-stability property
$s \leftarrow p/c$ ▷ sets subsampled_proportion property
$X_c \leftarrow \bigcup_{i=1}^{c} X$ ▷ create new database of size $c\tilde{n}$, composed of c copies of X
if neighboring == "replace one" **then**
    $m \leftarrow \lfloor sc\tilde{n} \rfloor$ ▷ number of records that can be filled using subsampled private data
else if neighboring == "add/remove one" **then**
    $m \leftarrow Binomial(c\tilde{n}, s)$ ▷ number of records that can be filled using subsampled private data
end if
$(\epsilon', \delta') \leftarrow \left(\log\left(1 + s\left(e^{c\epsilon} - 1\right)\right), s\left(\sum_{i=0}^{c-1} e^{i\epsilon}\right)\delta\right)$ ▷ privacy amplification via subsampling
$X' \leftarrow sample\left(X_c, \max(m, n)\right) \bigcup \left(Aug(\varnothing, \max(0, n - m), \ldots)\right)$
return (X', ϵ', δ')

The ϵ', δ' terms come from first applying the group privacy definition with group size c to the database X_c to get $\left(c\epsilon, \left(\sum_{i=0}^{c-1} e^{i\epsilon}\right)\delta\right)$ [Vad17] and then applying privacy amplification by subsampling results from Theorems 8 and 9 of [BBG18]. Note that, for the "replace

one" definition, we could be using $\frac{m}{c\tilde{n}}$ instead of s in the privacy calculation. Using s gives us a very slightly worse privacy guarantee (the only difference is the $\lfloor \cdot \rfloor$ we used to get m), but is nice for consistency between the methods and not having to keep track of m as an extra property.

4.4 Functional Privacy Parameters

We established in section 4.3 that a user asking for an (ϵ, δ)-DP guarantee will get an (ϵ', δ')-DP guarantee with respect to the original private data. What we'd really like, however, is for the user to ask for an (ϵ, δ)-DP gurantee and have the library come up with what we will call a *functional* (ϵ_f, δ_f) that will ensure (ϵ, δ)-DP on the original data. Any components that operate on the resized data will use the *functional* (ϵ_f, δ_f) internally instead of the parameters passed by the user.

Theorem 1. *A mechanism that respects*

$$(\epsilon_f, \delta_f) = \left(\frac{1}{c} \log\left(\frac{e^{\epsilon} - 1}{s} + 1 \right), \frac{\delta}{s\left(\sum_{i=0}^{c-1} e^{i\epsilon}\right)} \right) \text{-}DP$$

on the resized data respects (ϵ, δ)-DP on the true private data.

Proof. We know that an (ϵ, δ)-DP on the resized data corresponds to a

$$(\epsilon', \delta') = \left(\log\left(1 + s\left(e^{c\epsilon} - 1\right)\right), s\left(\sum_{i=0}^{c-1} e^{i\epsilon}\right)\delta \right)$$

guarantee on the original data. But we want our mechanism to respect (ϵ_f, δ_f) on the resized data such that it respects (ϵ, δ) on the original data. So we replace (ϵ, δ) with ϵ_f, δ_f and (ϵ', δ') with (ϵ, δ). Then we can invert the function to get what we want.

Let's start with ϵ, ϵ_f:

$$\begin{aligned} \epsilon &= \log\left(1 + s\left(e^{c\epsilon_f} - 1\right)\right) \\ e^{\epsilon} &= 1 + s\left(e^{c\epsilon_f} - 1\right) \\ \frac{e^{\epsilon} - 1}{s} &= e^{c\epsilon_f} - 1 \\ \frac{1}{c}\log\left(\frac{e^{\epsilon} - 1}{s} + 1\right) &= \epsilon_f. \end{aligned}$$

We carry out a similar calculation for δ, δ_f:

$$\begin{aligned} \delta &= s\left(\sum_{i=0}^{c-1} e^{i\epsilon}\right)\delta_f \\ \frac{\delta}{s\left(\sum_{i=0}^{c-1} e^{i\epsilon}\right)} &= \delta_f. \end{aligned}$$

□

So, in order for a mechanism to respect (ϵ, δ)-DP on the original data, it must respect $\left(\frac{1}{c}\log\left(\frac{e^{\epsilon}-1}{s} + 1\right), \frac{\delta}{s\left(\sum_{i=0}^{c-1} e^{i\epsilon}\right)} \right)$-DP on the resized data.

We now present an mini-algorithm for finding the *functional* (ϵ, δ).

Algorithm 4 Finding Functional (ϵ_f, δ_f): get_func_priv(p, ϵ, δ)

$c \leftarrow \lceil p \rceil$ ▷ sets c-stability property

$s \leftarrow p/c$ ▷ sets subsampled_proportion property

$\epsilon_f \leftarrow \frac{1}{c} \log\left(\frac{e^{\epsilon}-1}{s} + 1\right)$

$\delta_f \leftarrow \frac{\delta}{s\left(\sum_{i=0}^{c-1} e^{i\epsilon}\right)}$

return (ϵ_f, δ_f)

4.5 Examples

Let X be such that $\tilde{n} = |X| = 100$. We can look at a few examples of calls to the resize function (which will return X') and check the behavior.

1. resize(X, 150, 1, ...): X' will be made up of the 100 true elements of X and 50 imputed values. The functional privacy parameters are identical to the ones the user provides.
2. resize(X, 100, 0.75, ...): X' will be made up of 75 true elements of X and 25 imputed values. The functional privacy parameters will benefit (lower noise) from amplification via subsampling.
3. resize(X, 90, 1.5, ...): X' will be a random sample of $X \bigcup X$ of size 90. The functional privacy parameters will lead to greater noise than what the user provides, as they have to take into account the new c-stability of 2. This example is illustrative in that is shows that the functional privacy usage is affected only by p – it has nothing to do with the relative sizes of the X and X'.

4.6 How to choose n

To this point, we have ignored how a user would go about choosing n. If no trusted public metadata exist, we suggest estimating the size of the private database (e.g. using the Geometric mechanism) and using that release for n.

4.7 Accuracy/Error

The accuracy/error calculations in the library are always relative to the resized data X', rather than the true private data X. We are currently thinking about how best to extend guarantees from X' to X.

References

[BBG18] Borja Balle, Gilles Barthe, and Marco Gaboardi. Privacy amplification by subsampling: Tight analyses via couplings and divergences. *CoRR*, abs/1807.01647, 2018.

[Vad17] Salil Vadhan. The complexity of differential privacy. In *Tutorials on the Foundations of Cryptography*, pages 347–450. Springer, 2017.