Automatic, Fine-Grained Algorithmic Choice for DP

Jacob Imola
School of Computer Science, Carnegie Mellon University

Jean Yang
School of Computer Science, Carnegie Mellon University

February 10, 2018

1 Introduction

Why is algorithmic choice in Differential Privacy important?

The rapid technological increase in data collection, speed, and storage has brought about revolutionary insights and ideas and will continue to do so. However, with huge amounts of private data comes the concern of preventing data from ending up in the wrong hands. In order to prevent data leakage, we must lay a strong privacy foundation and give data analysts the tools they need without forcing them to become privacy experts.

Consider a healthcare database with records like patient weight, age, some genetic information, or whether they are HIV positive. Granting access rights to just the patients and their doctors protects privacy perfectly and developing tools for verification is an interesting question in its own right. However, sometimes it's okay to release a few general statistics about a database so that an analyst can find risk factors for people who have HIV. On the other side of the spectrum, publicly releasing all the information doesn't protect an individual's information. It is therefore necessary to use a middle ground tool where the amount of information released can be reliably controlled. Perhaps the most promising tool is Differential Privacy.

Differential Privacy (DP) has been the interest of many researchers and programmers since its conception in 2006. Its goal is to provide strong bounds on the amount of information being released from a dataset. Previous attempts were not mathematically rigorous and did not reliably prevent attacks. Most notably, before DP, researchers were able to reidentify users in a Netflix dataset given an auxiliary dataset from IMDB and form a generalized attack against the state-of-the-art privacy algorithms of the time [5]. The strong privacy guarantee of DP, on the other hand, has a rigorous mathematical foundation that makes it impervious to the post-processing attacks that exploited the Netflix dataset, and more recently, AirBnB and Instagram. As our world becomes more and more data-driven, it becomes ever more important to protect privacy in a sturdy way.

However, just building a suite of DP algorithms is not satisfactory. We are motivated by the need to build tools that make the analyst's life easier, and an analyst who is not well-versed in privacy would be daunted by the mountains of DP algortihms out there. Privacy adds a new layer complexity to the data analyst's job because the performance of algorithms may vary wildly depending on the database or other input parameters. This leads to the central problem of this work:
Problem 1. *How can a data analyst make the correct algorithmic choice given that a many tasks may be solved by more than privacy algorithm?*

This problem is noted by the authors of DPComp [2], who comment that currently, "the practitioner is lost and incapable of deploying the right algorithm". A data analyst should not responsible for making this choice. Little research has been conducted on how to help make the proper algorithmic choice; the most related work so far has been on visualization tools such as DPComp. However, we take a programming-language based approach for the following reasons:

- **Abstraction** Abstracting privacy in a programming language allows for the suppression of complicated privacy machinery, allowing programmers to view privacy as a black box. Programmers write code faster with fewer bugs when they can think about complicated ideas as black boxes.
- **Generalization** It is infeasible to conduct research on all use cases that a programmer may encounter. Writing a sophisticated programming language that automatically makes this choice would perform on par with the state-of-the-art methods. Effectively, we would allow anyone to stand on the shoulders of the giants who invented these algorithms.

```
def f(D):
    NoisyIf(g D):
        do A
    else:
        do B
```

Figure 1: Example `NoisyIf` statement.

The programming language could be termed *privacy agnostic* because it allows programmers and data practitioners to be agnostic to the advanced studies that have gone into developing differentially-private algorithms yet still attain the level of performance of the highest-grade algorithms in the field. Because the programming language will automatically make decisions during execution in an attempt to maximize accuracy, we call it `Jostle`.

The most central idea in `Jostle` is the `NoisyIf` statement, illustrated in Figure 1. A programmer will use `NoisyIf` when they are unsure of which choice to make at a certain point in their code. If they have certain beliefs that may aid `Jostle` in making the choice, then they may specify an *advice* function (denoted by `g` in Figure 1). Jostle will combine this advice along with previous executions of this particular `NoisyIf` to make the best decision.

We will begin with a background section on DP and related work. Then we will introduce decision trees and highlight the existence of Problem 1 with them. Thirdly, we will solve the decision tree problem with different `NoisyIf` statements, and finally, we will illustrate how `NoisyIf` statements can be generalized to make `Jostle`.

2 Background

2.1 DP

DP is based on the intuition of a user being asked to participate in a study involving an algorithm $\mathcal{A}$ being run on a database D. If $\mathcal{A}(D)$ changes as a result of the user participating, then this poses a privacy concern. An attacker may be able to infer something about the participant's input. This means that non-trivial deterministic algorithms are already unacceptable; their output may change depending on just a single addition to D. The strength ϵ of a DP guarantee is the maximum factor that the randomized output of $\mathcal{A}$ changes for two databases D and D' which differ in one row. Mathematically, we are saying:

Definition 1. *$\mathcal{A}$ satisfies ϵ-DP if for all D and D' such that $|D - D'|_1 = 1$ and for all o in the range of $\mathcal{A}$,*

$$\Pr\left(\mathcal{A}(D) = o\right) \le e^{\epsilon} \Pr\left(\mathcal{A}(D') = o\right)$$

There is also a weaker definition:

Definition 2. *$\mathcal{A}$ satisfies (ϵ, δ)-DP if for all D and D' such that $|D - D'|_1 = 1$ and for all o in the range of $\mathcal{A}$,*

$$\Pr\left(\mathcal{A}(D) = o\right) \le e^{\epsilon} \Pr\left(\mathcal{A}(D') = o\right) + \delta$$

For much of this paper, we will focus on ϵ-DP, but it is worth knowing the more general case so we can import the well-known privacy theorems in their most general form. Below is perhaps the most important result, and its proof comes cleanly from the definition of DP.

Theorem 1. *(Post-Processing) If $\mathcal{A}$ satisfies (ϵ, δ)-DP, and F is any function that takes the output of $\mathcal{A}$ as input, then $F(\mathcal{A})$ satisfies (ϵ, δ)-DP.*

This theorem is the reason why DP is such a useful guarantee. Data analysts can be sure that once they run their algorithm $\mathcal{A}$ and release its output, then the DP guarantee gets no weaker *no matter what an adversary does with the data.* This prevents the headaches where an analyst realizes retroactively that the data he released can be combined in some way to reveal much more information than was intended. Another useful, intuitive result is:

Theorem 2. *(Composition) Given algorithms M_1 and M_2 satisfying ϵ_1 and ϵ_2 DP, respectively, along with a database D, the algorithm $M = (M_1(D), M_2(D))$ has $(\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)$ DP.*

Composition is like the union bound from probability; it's convenient to apply but often is a pessimistic bound, as we will see later. Because of composition, we often refer to ϵ as a privacy budget—if we string together many private computations, it's like we spend some of our budget on them, and our total budget is ϵ.

Finally, we can talk about the disjointness of our queries—if D is split into disjoint parts, before mechanisms are applied to it, then out of all its possible neighbors, only one of the parts will be different. Thus, only the worst mechanism will affect the guarantee:

Theorem 3. *(Disjointness) Given disjoint subsets D_1, D_2 of D with two mechanisms M_1 and M_2 providing (ϵ_1, δ_1) and (ϵ_2, δ_2)-privacy, then $((M_1(D_1), M_2(D_2))$ satisfies $(\max\{\epsilon_1, \epsilon_2\}, \max\{\delta_1, \delta_2\})$-DP.*

So, what's a simple example of a DP algorithm? Suppose each row of our database D is 0 or 1, so $D \in \{0,1\}^n$, and that we are trying to release the sum of the elements of D. If this sum is S, then all neighboring databases D' have sum S or $S+1$. We can add noise to S so that it looks very similar in distribution to $S+1$. The distribution we are looking for is the Laplace distribution:

Definition 3. *The Laplace(λ) distribution has probability mass function $f(x) = \frac{1}{2\lambda}e^{-|x|/\lambda}$.*

This distribution fits perfectly with the definition of DP because of the exponentials. If X, Y are i.i.d. from Laplace $\left(\frac{1}{\epsilon}\right)$, then it is straightforward to show that the distributions of $S+X$ and $S+1+X$ satisfy $(\epsilon, 0)$ DP. To generalize this statement, we will use the following definition:

Definition 4. *(Sensitivity) A function f is Δ-sensitive if for all x, y such that $|x-y|_1 = 1$, we have*

$$|f(x) - f(y)| \leq \Delta$$

This can equivalently be rephrased as

$$\max_{|x-y|_1=1} |f(x) - f(y)| = \Delta$$

We will denote the sensitivity of f by $\Delta(f)$.

This gives us the following mechanism:

Algorithm 1: Laplace Mechanism

```
Input  : $D$, a database; $f$, a function; and $\epsilon$
Output: An estimate for $f(D)$ satisfying $\epsilon$ DP.
1 $X \sim$ Laplace $\left(\frac{\Delta(f)}{\epsilon}\right)$;
2 return X+f(D)
```

Theorem 4. *The Laplace Mechanism 1 satisfies $(\epsilon, 0)$ DP.*

For the counting or histogram queries such as our example above, we have $\Delta = 1$ so we add Laplace $\left(\frac{1}{\epsilon}\right)$ noise to our function.

However, what if we wanted to compute the maximum value in a set? If we had n elements, we certainly wouldn't want to apply Composition n times,

```
double NoisyCount(double epsilon){
    if(myagent.apply(epsilon)){
        return mysource.Count() + Laplace(1.0/epsilon);
    }else{
        throw new Exception("Access Denied")
    }
}
```

Figure 2: NoisyCount Implemented in PINQ.

obtaining $n\epsilon$-DP, just to have `Laplace` $\left(\frac{1}{\epsilon}\right)$ noise added to our answer. A better way is to use the exponential mechanism 2. Instead of paying $n\epsilon$, the exponential mechanism allows us to pay ϵ for essentially the same noise on our answer, and it works for any utility function.

```
Algorithm 2: Exponential Mechanism
Input  : $D \in \mathcal{D}$; $\mathcal{X}$, a domain; $f : \mathcal{X} \times \mathcal{D} \to \mathbb{R}$, a utility function, $\epsilon$
Output: $x \in \mathcal{X}$ where $f(x, D)$ is more likely to be high.
1 Pick $x \in \mathcal{X}$ where $\Pr(x = k) \propto \exp\left(\frac{\epsilon f(k,D)}{\Delta(f)}\right)$;
2 return $x$
```

2.2 Related Work

Perhaps the best-known example of a language that attempts to help practitioners use DP is PINQ [4]. PINQ builds off Microsoft's C-sharp LINQ database system and provides an interface between the database and the programmer that can accomplish simple differential-privacy mechanisms. PINQ makes extensive use of the Laplace Mechamism 1 to noise histogram queries such as `Count`, `Average`, and `Median`. It also uses Composition 2 extensively when many of these queries are executed, and it attempts to abstract the composition into a `PINQAgent` class which keeps track of privacy budget. For example, the `NoisyCount` function is implemented in Figure 2 The functionality of PINQ is limited, although a lot of code can still be written, and McSherry provides k-means and Social Networking examples to prove its power. Also, the interface is a step in the right direction of thinking about privacy as a black box. However, the drawback of PINQ is that an analyst must still think about noise at every step of the computation. If an

`Access is denied` error is thrown or the results are unacceptably noisy, the analyst will have no idea how to fix their code without diving into privacy.

Several languages have been built off PINQ to try to increase its functionality [6] [3]. wPINQ [6] uses a weighted database which is a function $f : \mathcal{D}^n \to \mathbb{R}$, which can be thought of a generalization of regular databases where each element appears a natural number of times. Proserpio et al. define `Join` on two weighted databases in a way that greatly reduces the weights of elements that appear many times in the resulting join.

For instance, in a complete bipartite graph with n vertices, there are no triangles. However, the addition of a single edge can add $O(n)$ triangles to the graph, so the triangle-counting function is very sensitive. If we were to count the number of triangles given a database D of all the edges, we could start with the standard outer Join, $\texttt{Join}(D, D)$, to produce paths of length 2. Indeed, this produces a database where $O(n)$ elements could vary when a single edge is changed. To prevent this problem, the authors define a new type of Join, where the weight of an element in $\texttt{Join}(D, D)$ is inversely proportional to the number of times each vertex in the path appears in any path of length 2. This curbs the sensitivity immensely, and the authors succeed in counting the number of triangles (or any design) using their version of `Join`.

The authors of [3] overcome the large sensitivity of `Join` by forcing the predicate of the Join to be just equality of keys. This greatly reduces the

Static methods

2.3 Decision Trees

Decision Trees are a powerful tool for data mining due to their high human interpretability, non-parametric design, low computational cost, ability to discover non-linear relationships among attributes, resilience to missing values, ability to handle both discrete and continuous data, and ability to handle non-binary labels [1]. Let $\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ be a set of attributes for each input, where $\mathcal{A}_i$ is a set of possible values for the ith coordinate. Let $\mathcal{C}$ be the output attribute, or class. A decision tree classifies points by branching on attribute $\mathcal{A}_i$, forming $|\mathcal{A}_i|$ subtrees. Once certain criteria are met, no more branching occurs, and instead a leaf node predicts a branch. An example Decision Tree is given in Figure 3.

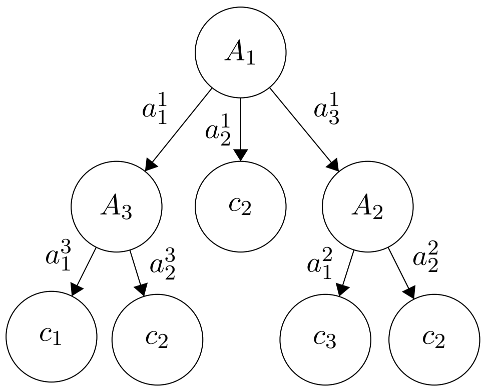

Figure 3: Example Decision Tree.

The most widely-used algorithm for training decision trees is the C4.5 algorithm. C4.5 grows trees top-down, and it creates a branch up to a certain depth specified by the user. For each branch, it selects the attribute that produces the lowest conditional entropy and splits the dataset on it. Conditional Entropy is defined as:

Definition 5. *(Entropy) The Entropy of a discrete random variable X which attains k values with probabilities $p_1, p_2, \ldots, p_k$ is*

$$H(X) = -\sum_{i=1}^{k} p_i \log(p_i)$$

The conditional entropy of X given a discrete random variable Y which attains values $a_1, \ldots, a_\ell$ is

$$H(X \mid Y) = \sum_{i=1}^{\ell} \Pr[Y = i] H(X \mid Y = i)$$

Conditional entropy, being a measure for information, is minimized so as to prioritize those attributes which produce large information gain. For leaf nodes, the class that has the largest representation in the remaining dataset is selected. The C4.5 algorithm appears in Algorithm 4. We denote by D_k to be the kth column of database D and use bracket notation $D[i]$ to represent those rows of D where conditional i is satisfied. When converting this mechanism to the differentially-private version, the user is left with several questions as noted in [1]:

```
def DTree(D, att, class, d):
    if(d = 0):
        return Leaf(prediction=get_most_common_class(D, class))
    else:
        E_now     = Entropy(D[k+1])
        best_att = arg_max(att, lambda a: c_entropy(D, class, a))
        C = map(best_att, lambda a: DTree(D[best_att == a],
            att[att != best_att], class, d-1))
        return Node(att=best_att, children=C)
```

Figure 4: C4.5 Algorithm.

- How large of a budget has been alotted or should be alotted? There isn't a clear way to decide the budget, as the performance of the algorithm may vary wildly with the budget.
- How many times should the data be queried? How would pick d in C4.5? Should one alter line (1) to something different?
- Might the sensitivity of some of the queries prevent an accurate choice? Specifically, even though it is widely agreed that the entropy function performs best in the non-private setting, could a lower-quality function be substituted because its computation is more accurate?
- How does the size of D impact performance? Is there enough data to provide accurate results in the private setting?

As we will see, the answers to these questions are data-dependent and there is never one answer that always dominates.

3 Decision Tree Examples

For recursive decision tree algorithms set up like **??**, the recursive calls on the same depth of the tree all operate on disjoint subsets of D. Therefore, a conservative estimate of the privacy usage, using composition and disjointness, is

$$\sum_{i=0}^{d} \max_{n \in \texttt{Nodes on lvl } i} \epsilon_n \tag{1}$$

where ϵ_n is the privacy used by node n. We will go into details on how to pick ϵ_n later The most naive way to create a DP version of C4.5 is to take all the histogram queries in algorithm 4 and change them to `NoisyCount`. This results in the following code for `c_entropy`

```
def c_entropy(D, class, attribute):
    N = NoisyCount(D)
     for a in attribute:
         count_a = NoisyCount(D[attribute == a])
```

Theorem 5. *The entropy function on disjoint histogram counts $a_1, a_2, \ldots, a_n$ has sensitivity*

Proof. Let $A = \sum_{i=1}^{n} a_i$. Then, the entropy is

$$\sum_{i=1}^{n} \frac{a_i}{A} \log\left(\frac{A}{a_i}\right) = \frac{1}{A}\sum_{i=1}^{n} a_i \log A - \frac{1}{A}\sum_{i=1}^{n} a_i \log(a_i) = \log(A) - \frac{1}{A}\sum_{i=1}^{n} a_i \log(a_i)$$

Suppose bucket a_j is reduced by 1, and the entropy change is

$$\begin{aligned}
&\log(A) - \log(A-1) - \frac{1}{A}a_j \log(a_j) + \frac{1}{A-1}(a_j - 1)\log(a_j - 1) \\
&\leq \frac{1}{\ln(2)(A-1)} - \frac{1}{A}(a_j - 1)\log(a_j - 1) + \frac{1}{A-1}(a_j - 1)\log(a_j - 1) \\
&= \frac{1}{\ln(2)(A-1)} + \frac{1}{A(A-1)}(a_j - 1)\log(a_j - 1) \leq \frac{1}{\ln(2)(A-1)} + \frac{1}{A}\log(A)
\end{aligned}$$

□

References

[1] Sam Fletcher and Md Zahidul Islam. Decision tree classification with differential privacy: A survey. *CoRR*, abs/1611.01919, 2016.

[2] Michael Hay, Ashwin Machanavajjhala, Gerome Miklau, Yan Chen, Dan Zhang, and George Bissias. Exploring privacy-accuracy tradeoffs using dpcomp. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 2101–2104, New York, NY, USA, 2016. ACM.

[3] Noah M. Johnson, Joseph P. Near, and Dawn Xiaodong Song. Practical differential privacy for SQL queries using elastic sensitivity. *CoRR*, abs/1706.09479, 2017.

[4] Frank McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. *Commun. ACM*, 53(9):89–97, September 2010.

[5] Arvind Narayanan and Vitaly Shmatikov. How to break anonymity of the netflix prize dataset. *CoRR*, abs/cs/0610105, 2006.

[6] Davide Proserpio, Sharon Goldberg, and Frank McSherry. Calibrating data to sensitivity in private data analysis: A platform for differentially-private analysis of weighted datasets. *Proc. VLDB Endow.*, 7(8):637–648, April 2014.