Section 10: Distributed Models of Differential Privacy

CS 208 Applied Privacy for Data Science, Spring 2022

April 13, 2022

1 Agenda

- Recap: Local Differential Privacy.
- Discovering Unknown Values in Histograms.
- Shuffle DP.
- Multiparty DP.

2 Local Differential Privacy

Recall some of the (trust) models of DP: central and local.

Central DP The data is held by a (trusted) curator (e.g., the U.S. Census Bureau). For example, if the data is $x_1, \ldots, x_n \in \{-1, 1\}$ and the curator wishes to compute the mean in an ϵ-DP way, she releases the following unbiased estimate of the mean:

$$\frac{1}{n}\sum_{i=1}^{n} x_i + \text{Lap}\left(0, \frac{2}{\epsilon n}\right),$$

with standard deviation of $O(\frac{1}{\epsilon n})$ for the noisy estimate.

Local DP The data is held by the individuals and the noisy estimates of the data is released to the curator. For example, if $x_1, \ldots, x_n \in \{-1, 1\}$, then the randomized response on each datapoint is

$$f(x_i) = \begin{cases} +x_i & \text{w.p. } \frac{e^\epsilon}{e^\epsilon+1} \\ -x_i & \text{w.p. } \frac{1}{e^\epsilon+1} \end{cases},$$

and the released estimate is

$$\frac{1}{n}\sum_{i=1}^{n} f(x_i).$$

The standard deviation of the noisy estimate is $O(\frac{1}{\epsilon\sqrt{n}})$ for the noisy estimate.

2.1 Implementing Local DP

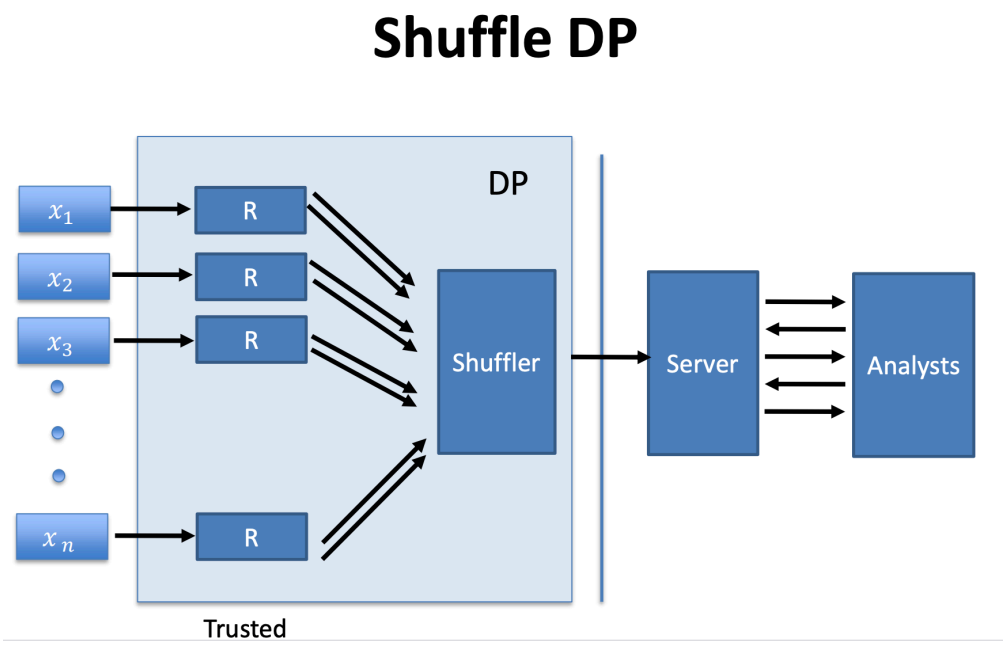
Exercise 2.1. Generate some synthetic data from a multinomial distribution. The multinomial distribution models the outcome of n experiments, where the outcome of each trial has a categorical distribution supported on $\{1, \ldots, k\}$. Each trial could, for example, correspond to rolling a k-sided die.

Exercise 2.2. Write some python code to compute the histogram, supported on k bins, of data generated from Exercise 2.1.

Exercise 2.3. Write some python code to compute the DP histogram—in the central model—of data generated from Exercise 2.1.

Exercise 2.4. Write some python code to compute the DP histogram—in the local model—of data generated from Exercise 2.1.

3 Shuffle model



Theorem 3.1 (Privacy Amplification by Shuffling (Informal))**.** *If R is ϵ_0-DP, then for every $\delta \in (0, 1]$, there exists ϵ such that $M(x_1, \ldots, x_n) = \text{Shuffle}(R(x_1), \ldots, R(x_n))$ is (ϵ, δ)-DP.*

For the binary sums protocol for Shuffle DP:

$$f(x_i) = \begin{cases} x_i & \text{w.p. } 1 - p \\ \text{Bern}(1/2) & \text{w.p. } p \end{cases}.$$

This results in error of, roughly,

$$O\left(\frac{\sqrt{\log 1/\delta}}{\epsilon n}\right),$$

instead of

$$O\left(\frac{\sqrt{\log 1/\delta}}{\epsilon \sqrt{n}}\right).$$

4 Applying Randomized Response to Histograms

Let every participant i have a bit $x_i \in \{1, \ldots, D\}$. We want to estimate the histogram $f(x) = (n_1, \ldots, n_D)$, where $n_j = \#\{i : x_i = j\}$. We will map each x_i to a "1-hot" indicator vector of length D and feed this into a local randomizer $Q(x_i)$, which works as follows.

1. Constructs "1-hot" vector $e_{x_i} = (0, \ldots, 0, 1, 0, \ldots, 0) \in \{0, 1\}^D$.
2. Applies $(\epsilon/2)$-DP randomized response to each coordinate to get $y_i \in \{0, 1\}^D$:

$$y_i = \begin{cases} e_{x_i}[j] & \text{w.p. } \frac{e^{\epsilon/2}}{1+e^{\epsilon/2}} \\ 1 - e_{x_i}[j] & \text{o.w.} \end{cases}.$$

3. Sends y_i to server.

Without privacy, the server would just sum up these indicator vectors to estimate the histogram. Since the sums are noisy, however, the server needs to add a multiplicative scaling factor c_ϵ (and, if we were using ± 1, an additive term) to correct the expectations of the sums. Thus, the server estimates $\hat{f}$ of the histogram f as follows:

$$\hat{f}(x) = c_\epsilon \sum_{i=1}^{n} y_i.$$

The expected error per bin is $\pm O(\sqrt{n}/\epsilon)$, using a similar analysis as we did when looking at basic randomized response, except now with counts instead of means. The expected max error over all D bins can be shown to be $\pm O(\sqrt{n \log D}/\epsilon)$.

The problem with this approach for estimating histograms is that the vectors can get quite long for the aggregator to store, which is why companies have used hashing and other compression techniques in their deployments.

5 Practical aspects of Deployments

5.1 Handling Large Domains

Imagine that D is very large. How can we compress the vectors y_i? One common technique is to use hashing.

The server chooses random hash function: $h : \{1, \ldots, D\} \to \{1, \ldots, m\}$ for $m \ll D$ and sends h to all users. Then, the users apply h to x_i and run the randomized response protocol on $h(x_i)$. The servers obtain the approximate histogram $\hat{f} \in \mathbb{Z}^m$ of $h(x_i)$'s.

At first, we might think that the server can estimate the frequency v using the shifting and scaling factors we calculated in the normal randomized response histogram. However, we need some additional scaling to correct for the possibility of hash collisions.

If we are using a good random hash function, for any value $v' \neq v$,

$$\Pr[h(v') = h(v)] = \frac{1}{m}$$

which means that

$$E[\hat{f}(v)] = \frac{m-1}{m} f(v) + \frac{n}{m}$$

Solving for $f(v)$, we have that for a value $v \in \{1, \ldots, D\}$, the server can finally estimate the frequency of v as

$$\frac{m \cdot \hat{f}[h(v)] - n}{m-1}$$

5.2 Reducing the Variance

Using a hash function reduces the size of the vectors. However, it increases the variance of the estimate due to the high probability of a hash collision (especially when hashing to a small space such as $m = 1024$). To reduce the variance, Google and Apple actually use multiple hash functions.

First, the users are randomly partitioned into k cohorts, where each cohort uses a different hash function h_j. Then, the randomized response is applied and the server sums up the estimator across all the cohorts.

How does using multiple hash functions reduce the variance of the estimate? The probability of a collision occurring decays exponentially with k, thereby reducing the error of our estimator. For example, for $v \neq v'$,

$$\Pr[h_1(v) = h_1(v') \wedge h_2(v) = h_2(v') \wedge \ldots \wedge h_k(v) = h_k(v')] = \prod_{i=1}^{k} \frac{1}{m} = \frac{1}{m^k}$$

5.3 Discovering Unknown Values

Before, we relied on the server knowing a small set of values for which it wanted to know the frequencies. But how can we identify high-frequency values that are not anticipated in advance? Using the above mechanisms, we might know that strings that hash to 17 occur frequently, but there might be a huge number of strings that fit this description. This space may be too large to enumerate over.

Our goal is to modify the mechanisms above such that we can actually reconstruct original values from the hashes, one symbol at a time. To do so, we will split up users into subgroups within each cohort. Users in the first subgroup will add the first symbol of their value to their hash, users in the second will add the second symbol, and so on.

For example, the string "HELLO" would hash to "137H" in the first subgroup, "137E" in the second subgroup, and so on. Thus, if our domain is ASCII strings of length 5, we are mapping from a space of size 256^5 to a much smaller space of $256m$. Then, we do randomized response to estimate the frequency of these values.

Now, if the server sees that a hash value of "137" occurs frequently, it can dig further to see that "137H, 137E, 137L" all occur frequently but "137G" does not. Thus, it would know that the string "HELLO," rather than another string "GOODBYE" that also hashes to "137", is the string that is actually frequently occurring.

5.4 Controlling Privacy Loss over Time

Companies aim to use local differential privacy to collect sensitive data from their users over a long period of time. How do they control the privacy loss?

Google has implemented the following heuristic method to control privacy loss. The idea is to conserve their privacy budget on values that are not expected to change often, such as default settings or more permanent attributes of the user.

Thus, they conduct two levels of randomized response. First, they apply an ϵ_1-randomized response on the more permanent data. The output of this permanent randomized response is reused until the underlying data changes. Then, they apply a ϵ_2-randomized response, where $\epsilon_2 > \epsilon_1$, on the output of the first randomized response. This second layer is called an instantaneous randomized response because it is re-randomized at every data-reporting time period.

The hope is that although many instantaneous releases will reveal the output of the permanent randomized response, this will still preserve ϵ_1-DP. And since the first layer of data is not expected to change often, the privacy loss will not grow too fast over time.

Some problems are that if the underlying permanent data is changing in an expected fashion (such as age), adversaries will be able to tell when the first randomized response changes and will be able to learn more about the underlying data.

6 Multiparty DP

Multiparty DP is an intermediate model where there are k curators C_j, each with a dataset $D^{(j)}$ on n_j subjects. Each curator is allowed to access the raw data of its own subjects, but not of the subjects of other curators. The k curators interact to carry out a joint analysis.

Definition 6.1 ((ϵ, δ)-Multiparty Differential Privacy)**.** For all $D^{(j)}, D^{(j)'}$ differing on one row, and for all polynomial-time adversaries A,

$$\Pr[A(C_j(D^{(j)})) = \text{ YES }] \leq e^{\epsilon} \cdot \Pr[A(C_j(D^{(j)'})) = \text{ YES }] + \delta.$$

We can also quantify over only "honest-but-curious" or "semi-honest" adversaries that follow the protocol but may try to glean sensitive information or do extra computations afterwards. This may be a reasonable model for aggregators such as Apple or Google, who want to protect themselves from subpeonas but who still have incentives to learn sensitive data. Other approaches are to use "threshold" adversaries or to anonymize the participants using a mixnet and verifiable shuffle for a boost in privacy.