differential_privacy_basic_concepts

June 19, 2020

1 Differential privacy basic concepts

There is a common situation in sociology where a researcher wants to develop a study about certain property on a population. The point is that this property is very sensitive and must remain private because could be embarrassing or illegal. However the researcher is able to get insights about the population without compromising their privacy. There is a simple mechanism in which every person follows a randomization algorithm that provides "plausible deniability". Consider the following algorithm (see The Algorithmic Foundations of Differential Privacy, Section 2.3):

1.- Flip a coin;

2.- If tails, then respond truthfully;

3.- If heads, then flip a second coin and respond "Yes" if heads and "No" if tails.

Even if the user answers "Yes", he could argue that this was due to the randomization algorithm and that this is not the true answer.

Imagine now that the researcher wants to use the previous algorithm to estimate the average number of people that have broken a particular law. The interesting point here is that we are able to obtain a very good approximation of the value preserving the privacy of every individual. Let's see a concrete example to get more familiar with the situation.

First of all we are going to generate a binary random vector with a concrete mean given by p.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import shfl

p = 0.2
data_size = 10000
array = np.random.binomial(1,p,size=data_size)
np.mean(array)
```

[1]:
```
0.2016
```

Let's suppose that every number represents whether the person has broken the law or not. Now we are going to generate federated data from this array. Every node is assigned one value, simulating, for instance, a piece of information in their mobile device.

[2]:
```
from math import log, exp

federated_array = shfl.private.federated_operation.federate_array(array,␣
↪data_size)
```

Now, we want every node to execute the previously defined algorithm and to return the result.

[3]:
```
from shfl.differential_privacy import RandomizedResponseCoins

# Define allowed access to data.
data_access_definition = RandomizedResponseCoins()
federated_array.configure_data_access(data_access_definition)

# Query data
result = federated_array.query()
```

Let's compare the mean of the original vector with the mean of the returned vector.

[4]:
```
print("Generated binary vector with mean: " + str(np.mean(array)))
print("Differential query mean result: " + str(np.mean(result)))
```

```
Generated binary vector with mean: 0.2016
Differential query mean result: 0.3507
```

What happened? Obviously, we have modified the true mean of the vector applying the randomization algorithm. We need to remove the influence of the latter to get the correct result. Fortunately, we can reverse the process to get a good estimation.

We are introducing noise half of the times with mean 0.5. So the expected value will be:

Expected estimated mean = actual mean * 0.5 + random mean * 0.5

[5]:
```
np.mean(array) * 0.5 + 0.5*0.5
```

[5]:
```
0.3508
```

Pretty close. Isn't it? To get the corrected estimated mean value we only need to use the following expression:

Corrected estimated mean = (estimated mean - random mean * 0.5) / 0.5

[6]:
```
(np.mean(result) - 0.5*0.5)/0.5
```

[6]:
```
0.20140000000000002
```

Right! We have obtained an estimation which is very similar to the true mean. However, of course, introducing randomness is not for free. In the next sections, we are going to formalize the error introduced in the estimation by the differential privacy mechanism and to study the effect of the population size and the privacy in the algorithm performance.

1.1 Differential Privacy definition

We introduce the notion of differential privacy (see The Algorithmic Foundations of Differential Privacy, Definition 2.4).

Let $\mathcal{X}$ be the set of possible rows in a database so that we can represent a particular database by a histogram $x \in \mathbb{N}^{|\mathcal{X}|}$. A randomized algorithm is a collection of conditional probabilities $P(z|x)$ for all x and $z \in \mathcal{Z}$, where $\mathcal{Z}$ is the response space.

Differential privacy is a property of some randomized algorithms. In particular, a randomized algorithm is ϵ-differentially private if, for all $z \in \mathcal{Z}$,

$$\frac{P(z|x)}{P(z|y)} \leq e^{\epsilon}$$

for all databases $x, y \in \mathbb{N}^{|\mathcal{X}|}$ such that $||x - y||_1 = 1$. In words, this definition means that for any pair of similar databases (i.e., differing in one element only), the probability of getting the same result after randomization is similar (i.e., probabilistically bounded).

1.2 Digging into the Randomized Response

Let's get back to sociological studies.

Suppose a group of N people take part in a study that wants to estimate the proportion of the population that commits fraud when paying their taxes. Since this is a crime, the participants might be worried about the consequences of telling the truth, so they are told to follow the algorithm described previously.

This procedure is, in fact, an ϵ-differentially private randomized mechanism. In particular, given this differential privacy mechanism, we have that

$$P(\text{respond yes} \,|\, \text{actual yes}) = \frac{3}{4} \qquad P(\text{respond no} \,|\, \text{actual yes}) = \frac{1}{4}$$

$$P(\text{respond yes} \,|\, \text{actual no}) = \frac{1}{4} \qquad P(\text{respond no} \,|\, \text{actual no}) = \frac{3}{4}\,,$$

and a direct computation shows that $\epsilon = \log(3)$.

The probability of responding "Yes" is given by

$$P(\text{respond yes}) = \frac{1}{4} + \frac{p}{2}$$

where $p = P(\text{actual yes})$ is the quantity of interest in the study. Using that $P(\text{respond yes})$ is estimated to be $r_P = \frac{\#(\text{respond yes})}{N}$, we have an estimate for the proportion of the population that commits fraud, namely

$$\hat{p} = 2r_P - \frac{1}{2}\,.$$

1.2.1 Trade-off between accuracy and privacy

Now, we are going to create a set of useful functions to execute some experiments and try to understand better the behavior of the previous expressions. The framework provides an implementation of this algorithm using two parameters, namely the probabilities of getting "heads" in each of the two coin tosses. For simplicity, we are going to consider the case in which the first coin is biased but the second one remains fair.

More explicitly, we consider the conditional probabilities given by

$$P(\text{respond yes}|\text{actual yes}) = f$$

$$P(\text{respond yes}|\text{actual no}) = 1 - f\,.$$

with $f \in [1/2, 1]$. This corresponds to taking the first coin with $p(\text{heads}) = 2(1-f)$ and the second coin to be unbiased. In this case we have that the amount of privacy depends on the bias of the first coin,

$$\epsilon = \log \frac{f}{1-f}\,.$$

For $f = 1/2$, we have that $\epsilon = 0$ and the algorithm is maximally private. On the other hand, for $f = 1$, ϵ tends to infinity, so the algorithm is not private at all.

The relation between the number of positive responses and p is given by

$$\hat{p} = \frac{r_P + f - 1}{2f - 1}\,.$$

In order to properly see the trade-off between utility and privacy, we may look at the uncertainty of the estimate, which is given by

$$\Delta p = \frac{2}{2f-1}\sqrt{\frac{r_P(1-r_P)}{N}}\,.$$

This expression shows that, as the privacy increases (f approaches $1/2$), the uncertainty of the estimate increases. On the other hand, for $f = 1$, the uncertainty of the estimate is purely due to the finite size of the sample N. In general, we see that the price we pay for being private is in terms of accuracy of the estimate (for a fixed sample size). (The expression for the uncertainty is computed using the normal approximation to the error of a binomial distribution.)

[7]:
```
def get_prob_from_epsilon(epsilon):
    f = np.exp(epsilon) / (1 + np.exp(epsilon))
    return 2*(1-f)
```

We also define the uncertainty function.

[8]:
```
def uncertainty(dp_mean, n, epsilon):
    f = np.exp(epsilon) / (1 + np.exp(epsilon))
    estimation_uncertainty = 2/(2*f-1) * np.sqrt(dp_mean*(1-dp_mean) / n)
    return estimation_uncertainty
```

Finally, we define a function to execute the experiments n_{runs} times so that we can study the average behavior.

[9]:
```
def experiment(epsilon, p, size):
    array = np.random.binomial(1,p,size=size)
    federated_array = shfl.private.federated_operation.federate_array(array,␣
 ↪size)

    prob = get_prob_from_epsilon(epsilon)
    # Define allowed access to data.
    data_access_definition = RandomizedResponseCoins(prob_head_first=prob)
    federated_array.configure_data_access(data_access_definition)
    # Query data
    result = federated_array.query()

    estimated_mean = (np.mean(result) - prob * 0.5)/(1-prob)
    estimation_uncertainty = uncertainty(np.mean(result), size, epsilon)
    return estimated_mean, estimation_uncertainty

def run_n_experiments(epsilon, p, size, n_runs):
    uncertainties = 0
    p_est = 0
    for i in range(n_runs):
        estimated_mean, uncertainty = experiment(epsilon,p,size)
        p_est = p_est + estimated_mean
        uncertainties = uncertainties + uncertainty
    p_est = p_est / n_runs
    uncertainties = uncertainties / n_runs
    return p_est, uncertainties
```

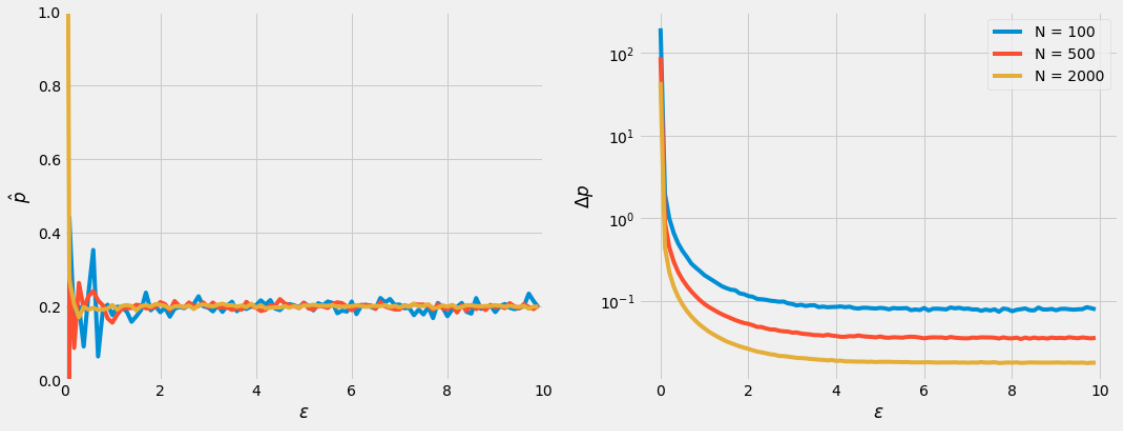
Now, we are going to execute the experiment and save the results.

[10]:
```
epsilon_range = np.arange(0.001, 10, 0.1)
n_range = [100, 500, 2000]
p_est = np.zeros((len(n_range), len(epsilon_range)))
uncertainties = np.zeros((len(n_range), len(epsilon_range)))
for i_n in range(len(n_range)):
    for i_e in range(len(epsilon_range)):
        p_est_i, uncertainty_i = run_n_experiments(epsilon_range[i_e], p,␣
 ↪n_range[i_n], 10)
        p_est[i_n, i_e] = p_est_i
        uncertainties[i_n, i_e] = uncertainty_i
```

We can visualize the results.

[11]:
```
plt.style.use('fivethirtyeight')
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(16,6))
for i_n in range(len(n_range)):
    ax[0].plot(epsilon_range, p_est[i_n], label = "N = " + str(n_range[i_n]))
ax[0].set_xlabel('$\epsilon$')
ax[0].set_ylabel('$\hat p$')
ax[0].set_xlim([0., 10])
ax[0].set_ylim([0, 1])
caption="Left:   The accuracy of the estimated proportion $\hat{p}$ decreases␣
 ↪as privacy increases, \
i.e. with smaller $\epsilon$ (the true proportion is $p=0.2$) \n          \
Moreover, larger sample sizes $N$ are associated to higher accuracy of the␣
 ↪estimated proportion $\hat{p}$.\
\nRight: For a fixed sample size $N$, the uncertainty in the estimate $\Delta␣
 ↪p$ \
grows as privacy increases, i.e. with smaller $\epsilon$. Moreover, \n          ␣
 ↪\
larger sample sizes $N$ are associated to lower uncertainty $\Delta p$."
ax[0].text(0.5, -.4, caption, ha='left')

for i_n in range(len(n_range)):
    ax[1].plot(epsilon_range, uncertainties[i_n], label = "N = " +␣
 ↪str(n_range[i_n]))
ax[1].set_xlabel('$\epsilon$')
ax[1].set_ylabel('$\Delta p$')
ax[1].set_yscale('log')
plt.legend(title = "", loc="upper right")
plt.show()
```



Left: The accuracy of the estimated proportion $\hat{p}$ decreases as privacy increases, i.e. with smaller ε (the true proportion is $p = 0.2$)
Moreover, larger sample sizes N are associated to higher accuracy of the estimated proportion $\hat{p}$.
Right: For a fixed sample size N, the uncertainty in the estimate Δp grows as privacy increases, i.e. with smaller ε. Moreover,
larger sample sizes N are associated to lower uncertainty Δp.