differential_privacy_binary_average_attack

June 18, 2020

1 Permanent randomized response

The basic idea of differential privacy is to make a particular query stochastic so that the underlying data is kept private. The average attack consists of performing the same query many times in order to reliably estimate the underlying data. This is, of course, not desirable so we should either limit the number of queries or design algorithms that are not vulnerable under this kind of attack.

In this notebook we present a simple example of this phenomenon based on a single node that contains one binary number n that encodes whether the node is guilty ($n = 1$) or innocent ($n = 0$). A randomized response algorithm (see The Algorithmic Foundations of Differential Privacy, Section 3.2, and the notebook Basic concepts) is used to query the node in order to preserve the privacy of n.

1.0.1 Average attack

We start by creating a single node that contains a binary number. In this case, we set this number to 1 (guilty). By setting f1=0.8, we make sure that 80% of the times we query the node whose data is 1, we get an answer of 1. On the remaining 20% of the cases we obtain an answer of 0.

[1]:
```python
import numpy as np
from shfl.private.node import DataNode
from shfl.differential_privacy.dp_mechanism import RandomizedResponseBinary
from math import log, exp

n = 1 #the node is guilty

node_single = DataNode()
node_single.set_private_data(name="guilty", data=np.array([n]))
data_access_definition = RandomizedResponseBinary(f0=0.8, f1=0.8,␣
 ↪epsilon=log(4) + 1e-10)
node_single.configure_data_access("guilty", data_access_definition)
```

If we perform the query just once, we cannot be sure that the result matches the true data.

[2]:
```python
result = node_single.query(private_property="guilty")
print("The result of one query is: " + str(int(result)))
```

```
The result of one query is: 0
```

If we perform the query N times and take the average, we can estimate the true data with an error that goes to zero as N increases.

[3]:
```
N = 500
result_query = []
for i in range(N):
    result_query.append(node_single.query(private_property="guilty"))
result_query = np.array(result_query)
print(np.mean(result_query))
```

```
0.834
```

We see that the average result of the query is close to 0.8. This allows us to conclude that the raw answer is most probably 1. Otherwise, the result would've been close to 0.2.

1.0.2 Permanent randomized response

A possible solution to this problem (e.g. see RAPPOR technology and Jiang 2019) is to create a node that contains two pieces of information: the true data and a **permanent randomized response** (PRR). The latter is initialized to None and, once the node receives the first query, it creates a random binary number following the algorithm described above which is saved as the PRR. The result of the query is then a randomized response using the PRR as input. This way, even if the query is performed a large number of times, the attacker may only guess with certainty the PRR, but not the true data.

[4]:
```
node_single_prr = DataNode()
data_binary = np.array([1])  #the node is guilty
node_single_prr.set_private_data(name="guilty", data=np.array([n]))
node_single_prr.configure_data_access("guilty", data_access_definition)

permanent_response = node_single_prr.query(private_property="guilty")
print("The PRR is: " + str(int(permanent_response)))

# we save the prr as a new piece of information
node_single_prr.set_private_data(name="guilty", data=np.append(data_binary,␣
 ↪permanent_response))
```

```
The PRR is: 1
```

From now on, all the external queries are done over the permanent randomized data, while the raw data remains completely hidden.

[5]:
```
N = 500
result_query = np.empty(shape=(N,))
for i in range(N):
    result_query[i] = node_single_prr.query(private_property="guilty")[1]
print(np.mean(result_query))
```

```
0.796
```

The result is not always close to 0.8, since the permanent response might be 0. The average attack may, at best, identify the permanent randomized response, but not the raw data.

1.0.3 Adaptive Differential Privacy

Adaptive Differential Privacy is a different approach to manage multiple queries against same data. The basic idea consists in register all the interactions until a global epsilon is spent.

[6]:
```
from shfl.differential_privacy.composition_dp import AdaptiveDifferentialPrivacy
from shfl.differential_privacy.composition_dp import ExceededPrivacyBudgetError

global_epsilon_delta = (7, 0)

node_single = DataNode()
node_single.set_private_data(name="guilty", data=np.array([n]))

data_access_definition = AdaptiveDifferentialPrivacy(global_epsilon_delta)

differentially_private_mechanism = RandomizedResponseBinary(f0=0.8, f1=0.8,␣
 ↪epsilon=log(4) + 1e-10)
node_single.configure_data_access("guilty", data_access_definition)

N = 500
result_query = []
for i in range(N):
    try:
        result_query.append(node_single.query(private_property="guilty",
                                                                      ␣
 ↪differentially_private_mechanism=differentially_private_mechanism))
    except ExceededPrivacyBudgetError:
        print("The privacy budget has been spent at run {}".format(i+1))
        break
```

```
The privacy budget has been spent at run 6
```