differential_privacy_composition_concepts

June 18, 2020

1 Introduction

This notebook is a continuation of the notebook differential_privacy_basic_concepts, we show more advanced concepts in Differential Privacy (DP) such as the composition theorems and how to use them in Sherpa.FL. Before diving in, we recommend reading section 3.5 from The Algorithmic Foundations of Differential Privacy and everything related to Privacy Filters from this paper Privacy Odometers and Filters: Pay-as-you-Go Composition.

2 Composition theorems

A great property of DP is that private mechanisms can be composed while preserving DP, the new values of ϵ and δ can be computed according to the composition theorems. Before the composition theorems are provided, we state an experiment with an adversarial which proposes a composition scenario for DP.

Composition experiment $b \in \{0, 1\}$ for adversary A with a given set, M, of DP-mechanisms

For $i = 1, \ldots, k$:

1. A generates two neighbouring databases x_i^0 and x_i^1 and selects a mechanism $\mathcal{M}_i$ from M.
2. A receives the output $y_i \in \mathcal{M}_i(x_i^b)$, which is stored in V^b

Remark that the Adversary is stateful, that is, it stores the output in each iteration and selects the DP-mechanism based on the observed outputs.

2.0.1 Note on neighbouring databases:

It is important to know that when it comes to numeric databases, such as two arrays, $A = [1, 2, 3, 4]$ and $B = [1, 2, 3, 8]$ (which by the way is the main use case of DP for Sherpa.FL). They are neighbouring databases if they differ in only one component as much as 1 (the must have the same length), therefore A and B aren't neighbouring databases but $C = [1, 28, 91]$ and $D = [2, 28, 91]$ are.

[1]:
```
from shfl.private.node import DataNode
from shfl.differential_privacy.dp_mechanism import LaplaceMechanism,␣
↪GaussianMechanism
from math import log, exp
import numpy as np
```

```python
def run_composition_experiment(M, db_storage, secret):
    # Number of runs equals to the number of mechanism provided
    k = len(M)

    # Adversary's view in experiment 1
    A_view1 = np.empty(shape=(k,))
    # Adversary's view in experiment 2
    A_view2 = np.empty(shape=(k,))

    # Neighbouring databases are created
    db1 = "db1"
    db2 = "db2"
    db_storage.set_private_data(name=db1, data=secret)
    db_storage.set_private_data(name=db2, data=secret+1)

    # In the following loop, we reproduce both experiments for b=0 and for b=1
    for i in range(k):
        # The adversarial selects the dp-mechanism
        db_storage.configure_data_access(db1, M[i])
        db_storage.configure_data_access(db2, M[i])
        # The outputs are stored in the adversary's view in each experiment
        A_view1[i] = db_storage.query(db1)
        A_view2[i] = db_storage.query(db2)

    return A_view1, A_view2
```

As you can see in the following piece of code privacy is preserved as it is not possible to tell in which database the secret is stored, but if this experiment is run for enough time, the probabilities of telling apart the difference increase, so what is the privacy budget spent in these experiments? This is the fundamental question that composition theorems answer

[2]:
```python
# Setup storage for all databases
db_storage = DataNode()

# List of DP-mechanisms
M = [LaplaceMechanism(1, epsilon=0.5),
     LaplaceMechanism(1, epsilon=1),
     GaussianMechanism(1, epsilon_delta=(0.5, 0.01))]

A_view1, A_view2 = run_composition_experiment(M, db_storage, 1)

print("Adversary's view from Experiment 1: {}, mean: {}".format(A_view1, np.
 ↪mean(A_view1)))
print("Adversary's view from Experiment 2: {}, mean: {}".format(A_view2, np.
 ↪mean(A_view2)))
```

```
Adversary's view from Experiment 1: [ 0.58698426 -0.0048289   3.71970278], mean:
1.433952713786833
```

```
Adversary's view from Experiment 2: [-3.02339064  1.99909929  5.16732688], mean:
1.381011844103406
```

As expected if the experiment is carried on for enough rounds we can decide in which database the secret is stored

[3]:
```python
# Setup storage for all databases
db_storage = DataNode()

# List of DP-mechanisms
M = [LaplaceMechanism(1, epsilon=0.5),
     LaplaceMechanism(1, epsilon=1),
     GaussianMechanism(1, epsilon_delta=(0.5, 0.01))]*1000

A_view1, A_view2 = run_composition_experiment(M, db_storage, 1)

print("Adversary's view from Experiment 1 mean: {}".format(np.mean(A_view1)))
print("Adversary's view from Experiment 2 mean: {}".format(np.mean(A_view2)))
```

```
Adversary's view from Experiment 1 mean: 0.9724678402050658
Adversary's view from Experiment 2 mean: 1.9266741884751446
```

The first and most basic theorem that can be employed for composition is:

Basic composition theorem

The composition of a sequence $\{\mathcal{M}_k\}$ of (ϵ_i, δ_i)-differentially private mechanisms under the Composition experiment with $M = \{\mathcal{M}_k\}$, is $(\sum_{i=1}^{k} \epsilon_i, \sum_{i=1}^{k} \delta_i)$-differentially private.

In other words, it states that the resulting privacy budget is the sum of the privacy budget spent in each access. Therefore the budget expend in the experiment before is:

[4]:
```python
epsilon_delta_access = [m.epsilon_delta for m in M]
epsilon_spent, delta_spent = map(sum, zip(*epsilon_delta_access))
print("{} epsilon was spent".format(epsilon_spent))
print("{} delta was spent".format(delta_spent))
```

```
2000.0 epsilon was spent
9.999999999999831 delta was spent
```

The main disadvantage of this theorem is that is assumes a worst case scenario. A composition theorem with a better bound can be stated:

Advanced composition theorem

For all $\epsilon, \delta, \delta' \geq 0$ the composition of a sequence $\{\mathcal{M}_k\}$ of (ϵ, δ)-differentially private mechanisms under the Composition experiment with $M = \{\mathcal{M}_k\}$, satisfies (ϵ', δ'')-DP with:

$$\epsilon' = \sqrt{2k \ln(1/\delta')} + k\epsilon(e^{\epsilon} - 1) \quad \text{and} \quad \delta'' = k\delta + \delta'$$

In other words, for an small sacrifice δ' in the global δ spent, we can achieve a better bound for the global ϵ spent. However, the theorem assumes that the same dp-mechanism is used in each access:

[5]:
```python
from math import sqrt, log, exp

# Basic theorem computations
def basic_theorem_expense(epsilon, delta, k):
    epsilon_spent = k*epsilon
    delta_spent = k*delta
    return epsilon_spent, delta_spent

# Advanced theorem computations
def advanced_theorem_expense(epsilon, delta, delta_sacrifice, k):
    epsilon_spent = sqrt(2*k*log(1/delta_sacrifice)) + k * epsilon *␣
 ↪(exp(epsilon) - 1)
    delta_spent = k*delta + delta_sacrifice
    return epsilon_spent, delta_spent


epsilon = 0.5
delta = 0
k = 3
delta_sacrifice = 0.1

basic = basic_theorem_expense(epsilon, delta, k)
advanced = advanced_theorem_expense(epsilon, delta, delta_sacrifice, k)

print("Epsilon: {} vs {} (basic theorem vs advanced theorem) ".format(basic[0],␣
 ↪advanced[0]))
print("Delta: {} vs {} (basic theorem vs advanced theorem) ".format(basic[1],␣
 ↪advanced[1]))
```

```
Epsilon: 1.5 vs 4.690004094900031 (basic theorem vs advanced theorem)
Delta: 0 vs 0.1 (basic theorem vs advanced theorem)
```

But wait, the epsilon spent is worse with the new theorem, is it useless? Of course not, let's see what happens when we increase the number of iterations:

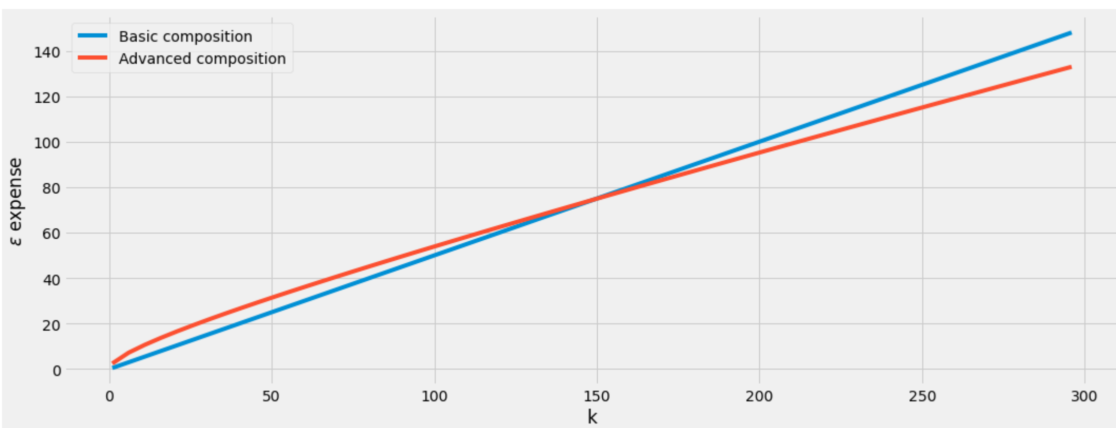
[6]:
```python
from math import sqrt, log, exp

epsilon = 0.5
delta = 0
k = 350
delta_sacrifice = 0.1

basic = basic_theorem_expense(epsilon, delta, k)
advanced = advanced_theorem_expense(epsilon, delta, delta_sacrifice, k)

print("Epsilon: {} vs {} (basic theorem vs advanced theorem) ".format(basic[0],␣
 ↪advanced[0]))
```

```
print("Delta: {} vs {} (basic theorem vs advanced theorem) ".format(basic[1],␣
↪advanced[1]))
```

```
Epsilon: 175.0 vs 153.67357054267973 (basic theorem vs advanced theorem)
Delta: 0 vs 0.1 (basic theorem vs advanced theorem)
```

So we can conclude that the benefits of the advanced theorem are only noticeable when the number of mechanism access are huge. In particular, we can observe that for values of k close to 150 (and $\delta = 0.1$), the theorems are almost indentical

[7]:
```
import matplotlib.pyplot as plt

plt.style.use('fivethirtyeight')

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(16,6))
k_values = np.arange(1, 300, 5)
ax.plot(k_values, [basic_theorem_expense(epsilon, delta, k)[0] for k in␣
↪k_values], label = "Basic composition")
ax.plot(k_values, [advanced_theorem_expense(epsilon, delta, delta_sacrifice,␣
↪k)[0] for k in k_values], label = "Advanced composition")
ax.set_xlabel('k')
ax.set_ylabel('$\epsilon$ expense')

plt.legend(title = "", loc="upper left")
plt.show()
```



While composition theorems are quite useful, they require some parameters to be defined upfront, such as the number of mechanisms to be composed. Therefore, no intermediate result can be observed and the privacy budget can be wasted. In such situations it is required a more fine grained composition technique which allows to observe the result of each mechanism without compromising the privacy budget spent. In order to remove some of the stated constraints a more flexible experiment of composition is introduced:

Adaptive composition experiment $b \in \{0, 1\}$ **for adversary** A

For $i = 1, \dots, k$:

1. A generates two neighbouring databases x_i^0 and x_i^1 and selects a mechanism $\mathcal{M}_i$ that is (ϵ_i, δ_i)-differentially private.
2. A receives the output $y_i \in \mathcal{M}_i(x_i^b)$

Remark that in these situations the ϵ_i and δ_i of each mechanism is adaptively selected based on the outputs of previous iterations.

Now we introduce the privacy filter which can be used to guarantee that with high probability in the Adaptive composition experiments, the stated privacy budget ϵ_g is never exceeded. Privacy filters have similar composition theorems to the ones given previously:

Basic composition for Privacy Filters

For any $\epsilon_g, \delta_g \geq 0$, $\texttt{COMP}_{\epsilon_g,\delta_g}$ is valid Privacy Filter:

$$\texttt{COMP}_{\epsilon_g,\delta_g}(\epsilon_1, \delta_1, ..., \epsilon_k, \delta_k) = \begin{cases} \texttt{HALT} & \text{if } \sum_{i=1}^k \delta_i > \delta_g \quad \text{or} \quad \sum_{i=1}^k \epsilon_i > \epsilon_g, \\ \texttt{CONT} & \text{otherwise} \end{cases}$$

Advanced composition for Privacy Filters

We define $\mathcal{K}$ as follows:

$$\mathcal{K} := \sum_{j=1}^{k} \epsilon_j \left(\frac{\exp(\epsilon_j) - 1}{2} \right) + \sqrt{\left(\sum_{i=1}^{k} \epsilon_i^2 + H \right) \left(2 + \ln\left(\frac{1}{H} \sum_{i=1}^{k} \epsilon_i^2 + 1 \right) \right) \ln(2/\delta_g)}$$

with

$$H = \frac{\epsilon_g^2}{28.04 \ln(1/\delta_g)}$$

Then $\texttt{COMP}_{\epsilon_g,\delta_g}$ is a valid Privacy Filter for $\delta_g \in (0, 1/e)$ and $\epsilon_g > 0$, where:

$$\texttt{COMP}_{\epsilon_g,\delta_g}(\epsilon_1, \delta_1, ..., \epsilon_k, \delta_k) = \begin{cases} \texttt{HALT} & \text{if } \sum_{i=1}^k \delta_i > \delta_g/2 \quad \text{or} \quad \mathcal{K} > \epsilon_g, \\ \texttt{CONT} & \text{otherwise} \end{cases}$$

The value of $\mathcal{K}$ might be strange at first sight, however if we assume $\epsilon_j = \epsilon$ for all j, it remains:

$$\mathcal{K} = \sqrt{(k\epsilon^2 + H) \left(2 + \ln\left(\frac{k\epsilon^2}{H} + 1 \right) \right) \ln(2/\delta)} + k\epsilon^2 \left(\frac{\exp(\epsilon) - 1}{2} \right)$$

which is quite similar to the expression given in the advanced composition theorem.

2.1 Privacy filters in Sherpa.FL

This frameworks implements Privacy Filters and transparently applies both theorems stated before, so that there is no need to constantly check which theorem ensures a better (ϵ, δ) expense. When the fixed privacy budget is surpassed, an exception ExceededPrivacyBudgetError is thrown. The following example shows two equivalent implementations of the Adaptive composition experiment, stated before.

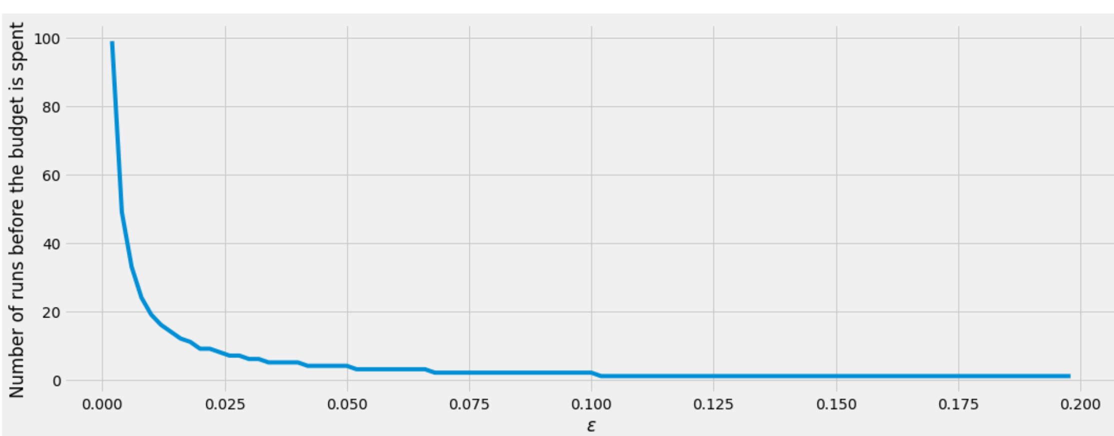
[8]:

```python
from shfl.private.node import DataNode
from shfl.differential_privacy.composition_dp import AdaptiveDifferentialPrivacy
from shfl.differential_privacy.composition_dp import ExceededPrivacyBudgetError
from shfl.differential_privacy.dp_mechanism import LaplaceMechanism
import numpy as np
import matplotlib.pyplot as plt


def run_adaptive_comp_experiment_v1(global_eps_delta, eps_delta_access):
    # Define a place to store the data
    node_single = DataNode()

    # Store the private data
    node_single.set_private_data(name="secret", data=np.array([1]))

    # Chooose your favourite differentially_private_mechanism
    dpm = LaplaceMechanism(sensitivity=1, epsilon=eps_delta_access)

    # Here we are specifing that we want to use composition theorems for␣
 ↪Privacy Filters
    # dp-mechanis
    default_data_access = AdaptiveDifferentialPrivacy(global_eps_delta,␣
 ↪differentially_private_mechanism=dpm)
    node_single.configure_data_access("secret", default_data_access)

    result_query = []
    while True:
        try:
            # Queries are performed using the Laplace mechanism
            result_query.append(node_single.query(private_property="secret"))
        except ExceededPrivacyBudgetError:
            # At this point we have spent all our privacy budget
            break

    return result_query

def run_adaptive_comp_experiment_v2(global_eps_delta, eps_delta_access):
    # Define a place to store the data
    node_single = DataNode()

    # Store the private data
    node_single.set_private_data(name="secret", data=np.array([1]))

    # Chooose your favourite differentially_private_mechanism
    dpm = LaplaceMechanism(sensitivity=1, epsilon=eps_delta_access)
```

```
    # Here we are specifing that we want to use composition theorems for␣
↪Privacy Filters
    default_data_access = AdaptiveDifferentialPrivacy(global_eps_delta)
    node_single.configure_data_access("secret", default_data_access)

    result_query = []
    while True:
        try:
            # DP-mechanism is specified at query time, in this case the Laplace␣
↪mechanism
            # if no mechanism is specified an exception is thrown
            result_query.append(node_single.query(private_property="secret",␣
↪differentially_private_mechanism=dpm))
        except ExceededPrivacyBudgetError:
            # At this point we have spent all our privacy budget
            break

    return result_query
```

In the following plot we can see that the privacy budget is spent notoriously faster as ϵ moves away from 0

[9]:
```
global_epsilon_delta = (2e-1, 2e-30)
epsilon_values = np.arange(2e-3, 2e-1, 2e-3)
plt.style.use('fivethirtyeight')

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(16,6))
y_axis=[len(run_adaptive_comp_experiment_v1(global_epsilon_delta, e)) for e in␣
↪epsilon_values]
ax.plot(epsilon_values, y_axis)
ax.set_xlabel('$\epsilon$')
ax.set_ylabel('Number of runs before the budget is spent')

plt.show()
```



2.2 Note

These experiments are run with the same dp-mechanism for simplification. If you want to access your data with different dp-mechanism we recommend using a schema similar to the one shown in the following function *run_adaptive_comp_experiment_v2*

[ ]:

```

```