differential_privacy_exponential

June 19, 2020

1 Exponential Mechanism

In this notebook we introduce the exponential mechanism (see The Algorithmic Foundations of Differential Privacy, Section 3.4). We consider three different examples that show how general the mechanism is.

Let $\mathcal{X}$ be the set of possible rows in a database so that we can represent a particular database by a histogram $x \in \mathbb{N}^{|\mathcal{X}|}$. Given an arbitrary response space $\mathcal{R}$ and a utility function

$$u : \mathbb{N}^{|\mathcal{X}|} \times \mathcal{R} \to \mathbb{R}\,,$$

we define the exponential mechanism as the collection of conditional probabilities given by

$$P(r|x; u, \alpha) = \frac{e^{\alpha u(x,r)}}{\sum_{r' \in \mathcal{R}} e^{\alpha u(x,r')}}\,.$$

In the case of uncountable $\mathcal{R}$, the sum in the denominator should be replaced with an integral and the expression above gives a probability density.

The exponential mechanism is an ϵ-differentially private randomization algorithm with $\epsilon = 2\alpha\Delta u$ and

$$\Delta u = \max_{r \in \mathcal{R}} \max_{||x-y||_1 \leq 1} |u(x,r) - u(y,r)|\,.$$

The utility function can be thought of as a generalization of a numeric query. In fact, one can show that this mechanism reduces to simpler ones for particular types of functions u. In order to illustrate this, we show how this works in two well-known examples.

Randomized response (see also the notebook Differential privacy basic concepts) Consider the case in which $\mathcal{R} = \{0, 1\}$ and $\mathcal{X} = \{\text{truly innocent}, \text{truly guilty}\}$ with a utility function such that

$$\begin{aligned} u(0,0) &= 0 \qquad & u(1,0) &= 0 \\ u(0,1) &= \beta \qquad & u(1,1) &= \gamma \end{aligned}$$

where β, γ are real constants. The exponential mechanism reduces in this case to the simplest randomized response algorithm.

Laplace Mechanism (see also the notebook Laplace mechanism) Consider the case in which $\mathcal{R} = \mathbb{R}$ and the utility function is given by $u(x, r) = -|f(x) - r|$ with

$$f : \mathbb{N}^{|\mathcal{X}|} \to \mathbb{R}\,.$$

This is, by definition, the Laplace mechanism with $b = \alpha^{-1}$.

1.1 Example: Pricing

The input dataset x is a set of bids for the purchase of an abundant supply of a product. The problem is to identify the best price $r \in \mathcal{R} = [r_{min}, r_{max}]$ such as to maximize the revenue, without revealing the bids. In this case the revenue is our utility function u, defined as

$$u(x, r) = r|S_r|$$

with

$$S_r = \{i : x_i \geq r\}$$

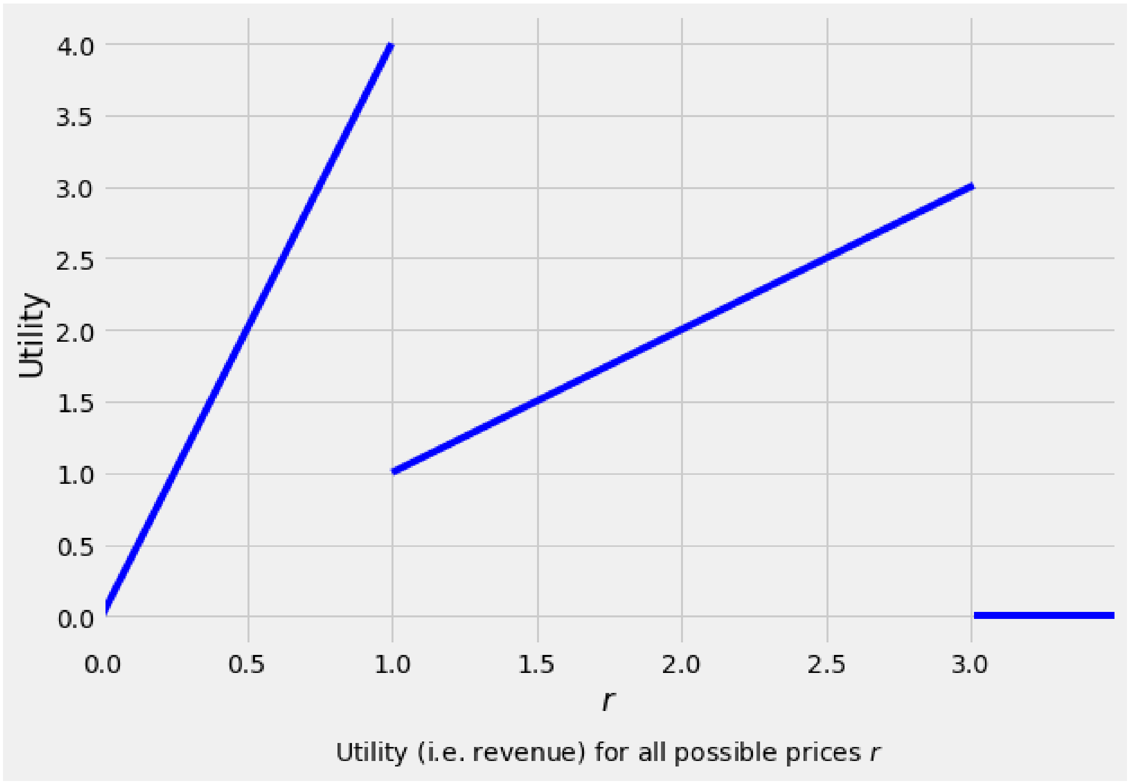
the set of people that are willing to buy at a price r.

In general, it is not possible to compute Δu analytically and one must resort to statistical estimations of “typical” values of Δu (see Rubinstein 2017). This is also the case when Δu is not bounded. In our particular case, we can compute the sensitivity with respect to the utility, which is given by

$$\Delta u = r_{max}.$$

The complication in this case arises from the fact that the output price may not be directly perturbed. In Example 3.5 of The Algorithmic Foundations of Differential Privacy, they suppose there are four bidders: A, F, I, K, where A, F, I each bid 1.00 and K bids 3.01, so $x = \{1, 1, 1, 3.01\}$. Fixing the price to 3.01 the revenue is $u = 3.01$, at 3.00 we have $u = 3.00$, at 1.00 we have $u = 4.00$. However, if we fix the price at 3.02 the revenue is zero! The revenue plot for this specific demand defined by x is shown below.

[1]:
```python
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm

def u(x, r):
    '''
    Utility function

    Arguments:
        x: list. True bids.
        r_range: array of possible values for the price.
    '''
    output = np.zeros(len(r))
    for i in range(len(r)):
        output[i] = r[i] * sum(np.greater_equal(x, r[i]))
    return output

x = [1.00, 1.00, 1.00, 3.01] # Input dataset: the true bids
r = np.arange(0, 3.5, 0.001) # Set the interval of possible outputs r

# Plot the utility (revenue) for each possible output r (price)
utility = u(x, r)
plt.style.use('fivethirtyeight')
fig, ax = plt.subplots(figsize=(9, 6))
x_intervals = np.sort(np.unique(np.append(x, [r.min(), r.max()])))
for i_interval in range(len(x_intervals) - 1):
    indices_interval = [all(union) for union in zip(
        r > x_intervals[i_interval], r <= x_intervals[i_interval + 1])]
    ax.plot(r[indices_interval], utility[indices_interval], color = "blue")

ax.set_xlim([r.min(), r.max()])
ax.set_xlabel('$r$')
ax.set_ylabel('Utility')
label="Utility (i.e. revenue) for all possible prices $r$"
ax.text(r.max()/2, -1, label, ha='center')

plt.show()
```

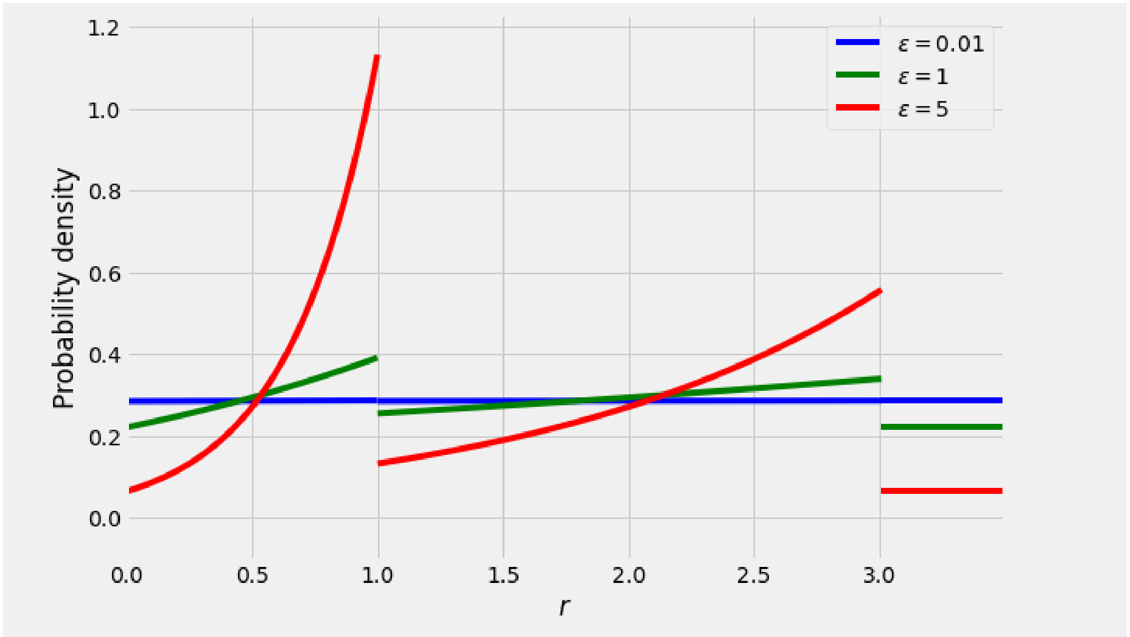


As explained above, the exponential mechanism is defined with respect to the utility function $u : \mathbb{N}^{|\mathcal{X}|} \times \mathcal{R} \to \mathbb{R}$, which maps database/output *pairs* to utility scores. For a fixed database x, the exponential mechanism outputs each possible $r \in \mathcal{R}$ with probability proportional to $\exp\left(\frac{\epsilon u(x,r)}{2\Delta u}\right)$. The resulting probability distribution is shown below.

It can be observed that for low values of ϵ the probability resembles a flat horizontal curve of the *uniform* probability, thus the privacy increases. On the contrary, for higher values of ϵ the probability curve exponentially reveals the jumps in the revenue, implying less privacy.

[2]:
```python
def PDF(x, r, epsilon):
    """
    PDF associated to the database x
    """
    r_min = r.min()
    r_max = r.max()
    x_intervals = np.sort(np.unique(np.append(x, [r_min, r_max])))
    area = 0
    for i in range(len(x_intervals) - 1):
        S = epsilon/(2*r_max) * sum(np.greater(x, x_intervals[i]))
        if S > 0:
            area_int = 1/S * (np.exp(S * x_intervals[i + 1]) - np.exp(S *␣
 ↪x_intervals[i]))
        elif S == 0:
            area_int = x_intervals[i + 1] - x_intervals[i]
        area = area + area_int

    u_prob_norm = np.exp(epsilon * u(x, r) / (2 * r_max)) / area
    return u_prob_norm


epsilon_range = [0.01, 1, 5]

fig, ax = plt.subplots(figsize=(9,6))
color_list = ["b", "g", "r", "c", "m"]

x_intervals = np.sort(np.unique(np.append(x, [r.min(), r.max()])))
for i_epsilon in range(len(epsilon_range)):
    u_prob_norm = PDF(x, r, epsilon_range[i_epsilon])
    for i_interval in range(len(x_intervals) - 1):
        indices_interval = [all(union) for union in zip(
            r > x_intervals[i_interval], r <= x_intervals[i_interval + 1])]
        ax.plot(r[indices_interval],
                u_prob_norm[indices_interval],
                color = color_list[i_epsilon],
                label = '$\epsilon = $' + str(epsilon_range[i_epsilon]))
```

```
handles, labels = ax.get_legend_handles_labels()
handles = handles[0:len(handles):(len(x)-1)]
labels = labels[0:len(labels):(len(x)-1)]
ax.set_xlim([r.min(), r.max()])
ax.set_ylim([-0.1, max(u_prob_norm) + 0.1])
ax.set_xlabel('$r$')
ax.set_ylabel('Probability density')
ax.legend(handles, labels)
label="Probability density function for different levels of privacy $\epsilon$.␣
 ↪For higher levels of privacy (i.e. lower $\epsilon$), \n\
the probability resembles the uniform distribution, while for lower levels of␣
 ↪privacy (i.e. higher $\epsilon$), \n\
the probability reveals the jumps in revenue and thus the bids."
ax.text(r.max()/2, -0.5, label, ha='center')

plt.show()
```

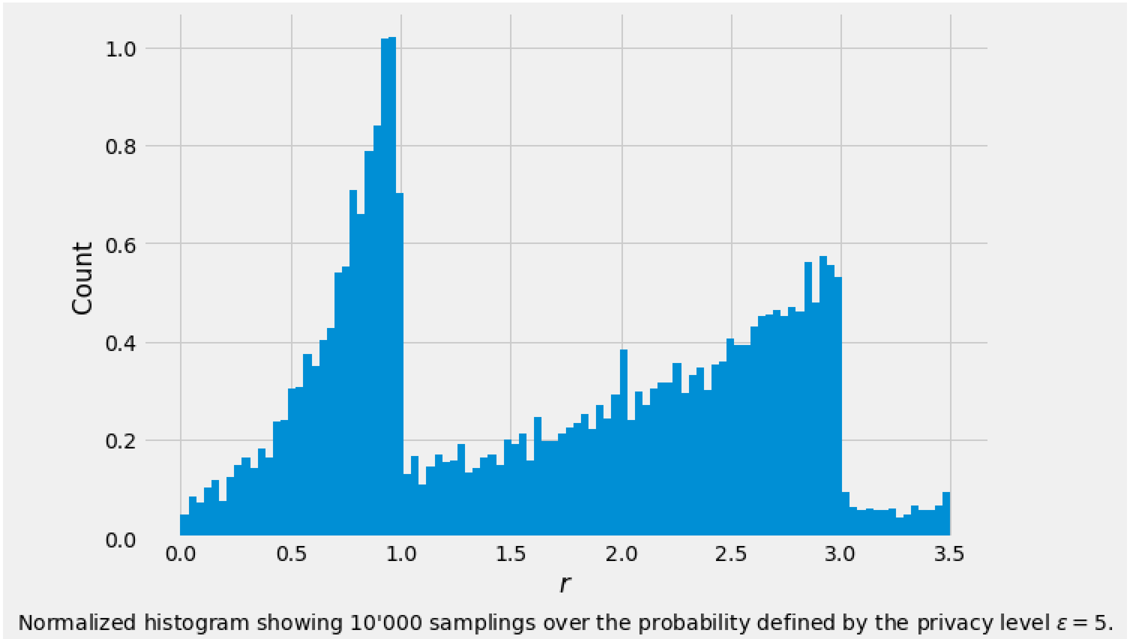


Probability density function for different levels of privacy ε. For higher levels of privacy (i.e. lower ε), the probability resembles the uniform distribution, while for lower levels of privacy (i.e. higher ε), the probability reveals the jumps in revenue and thus the bids.

We create a node that contains the true bids and choose to access the data using the exponential mechanism. We repeat the experiment 10000 times to check that the resulting distribution for the output price matches the one shown earlier.

```
[3]: from shfl.differential_privacy.dp_mechanism import ExponentialMechanism
     from shfl.private.node import DataNode

     node = DataNode()     # Create a node
     node.set_private_data(name="bids", data=np.array(x)) # Store the database x in␣
      ↪the node
     delta_u = r.max()     # In this specific case, Delta u = max(r)
     epsilon = 5           # Set a value for epsilon
     size = 10000          # We want to repeat the query this many times

     data_access_definition = ExponentialMechanism(u, r, delta_u, epsilon, size)
     node.configure_data_access("bids", data_access_definition)
     result = node.query("bids")
```

```
[4]: fig, ax = plt.subplots(figsize=(9,6))
     plt.hist(result, bins = int(round(np.sqrt(len(result)))), density = True)
     ax.set_xlabel('$r$')
     ax.set_ylabel('Count')
     label="Normalized histogram showing 10'000 samplings over the probability␣
      ↪defined by the privacy level $\epsilon=5$. \n\
     The sampled prices recover the actual distribution of the prices (see curve␣
      ↪$\epsilon=5$ in the previous picture)."
     ax.text(r.max()/2, -0.25, label, ha='center')

     plt.show()
```



Normalized histogram showing 10'000 samplings over the probability defined by the privacy level $\varepsilon = 5$.
The sampled prices recover the actual distribution of the prices (see curve $\varepsilon = 5$ in the previous picture).

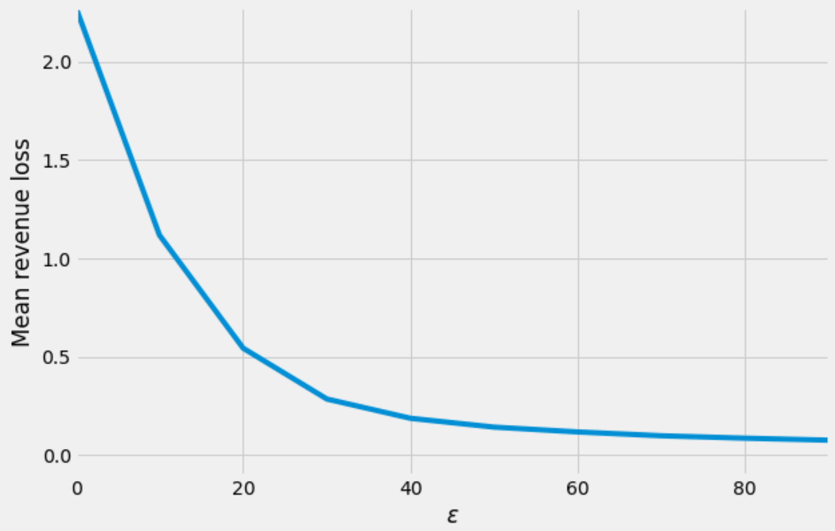
The trade-off between accuracy and privacy can be assessed by computing the mean revenue loss, which is defined as

$$\text{loss} = |u_{\text{OPT}}(x) - u(x, r_{\text{sample}})|$$

where $u_{\text{OPT}}(x)$ is the highest possible revenue for a fixed database x, and $u(x, r_{\text{sample}})$ is the revenue at the sampled price r_{sample}.

This is a measure of the loss in utility that we get when using the exponential mechanism compared to the case without privacy at all. The plot below shows that as ϵ increases (i.e. privacy decreases) the revenue loss goes to zero.

[5]:
```
epsilon_range = np.arange(0.001, 100, 10)
optimum_price = float(r[utility == max(utility)])
optimum_revenue = u(x, [optimum_price])
mean_loss = np.zeros(len(epsilon_range))

for i in range(len(epsilon_range)):
    epsilon = epsilon_range[i]
    node = DataNode()
    node.set_private_data(name="bids", data=np.array(x))
    data_access_definition = ExponentialMechanism(u, r, delta_u, epsilon, size)
    node.configure_data_access("bids", data_access_definition)
    result = node.query("bids")
    mean_loss[i] = np.mean(abs(u(x, result) - optimum_revenue))

fig, ax = plt.subplots(figsize=(9,6))
ax.plot(epsilon_range, mean_loss)
ax.set_xlim([0, max(epsilon_range)])
ax.set_ylim([-0.1, max(mean_loss)])
ax.set_xlabel('$\epsilon$')
ax.set_ylabel('Mean revenue loss')
label="Mean revenue loss for different levels of privacy $\epsilon$. \
As privacy decreases (i.e. higher $\epsilon$), the revenue loss goes to zero."
ax.text(epsilon_range.max()/2, -0.5, label, ha='center')

plt.show()
```



Mean revenue loss for different levels of privacy ε. As privacy decreases (i.e. higher ε), the revenue loss goes to zero.

1.1.1 Example: Randomized response from the exponential mechanism (see also the notebook Differential privacy basic concepts)

Consider the case in which $\mathcal{R} = \{0, 1\}$ and $\mathcal{X} = \{\text{truly innocent}, \text{truly guilty}\}$ with a utility given by

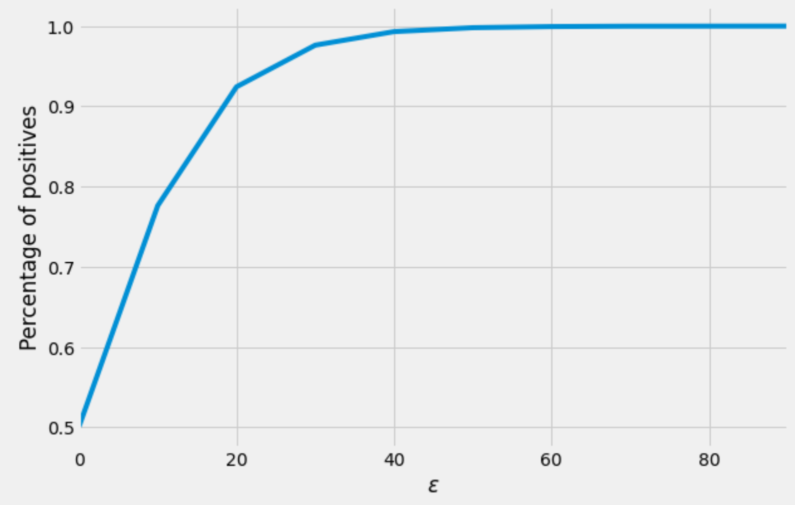
$$\begin{aligned} u(0,0) &= 0 \qquad & u(1,0) &= 0 \\ u(0,1) &= \beta \qquad & u(1,1) &= -\beta \end{aligned}$$

This means that $\Delta u = 2|\beta|$ so $\epsilon = 4\alpha|\beta|$. For concreteness, in the following we choose $\beta = -1/4$ so that $\epsilon = \alpha$.

In the figure below we show the percentage of times the query returns the same value as the true value as a function of ϵ. We observe, that this percentage approaches one as the privacy decreases. For $\epsilon = 0$, which gives maximal privacy, the result of the query is completely uninformative since the probability of getting either result is independent of the true value.

[6]:
```python
def u_randomized(x, r):
    '''
    Utility function for randomized mechanism

    Arguments:
        x: binary number
        r: array of binaries
    '''
    output = -1/4 * (1 - 2*x) * r
    return output
```

```
r = np.array([0,1]) # Set the interval of possible outputs r
x = 1               # Set a value for the dataset
delta_u = 1         # We simply set it to one
size = 100000       # We want to repeat the query this many times

epsilon_range = np.arange(0.001, 100, 10)
positives = np.zeros(len(epsilon_range))

for i in range(len(epsilon_range)):
    epsilon = epsilon_range[i]
    node = DataNode()               # Create a node
    node.set_private_data(name="identity", data=np.array(x))  # Store the␣
 ↪database x in the node
    data_access_definition = ExponentialMechanism(u_randomized, r, delta_u,␣
 ↪epsilon, size)
    node.configure_data_access("identity", data_access_definition)
    result = node.query("identity")
    positives[i] = result.sum() / size

fig, ax = plt.subplots(figsize=(9,6))
ax.plot(epsilon_range, positives)
ax.set_xlim([0, max(epsilon_range)])
ax.set_xlabel('$\epsilon$')
ax.set_ylabel('Percentage of positives')
label="Randomized response: percentage of times the query returns the true␣
 ↪value, as a function of the privacy level $\epsilon$.\n\
For lower privacy (i.e. higher $\epsilon$), the query returns the true value␣
 ↪almost 100% of the times. On the contrary, \n\
for higher privacy (i.e. lower $\epsilon$), the query returns either true or␣
 ↪false 50% of the times, thus being completely uninformative."
ax.text(epsilon_range.max()/2, 0.3, label, ha='center')

plt.show()
```



Randomized response: percentage of times the query returns the true value, as a function of the privacy level ε.
For lower privacy (i.e. higher ε), the query returns the true value almost 100% of the times. On the contrary,
for higher privacy (i.e. lower ε), the query returns either true or false 50% of the times, thus being completely uninformative.

1.1.2 Example: Laplace mechanism from the exponential mechanism (see also the notebook Laplace mechanism)

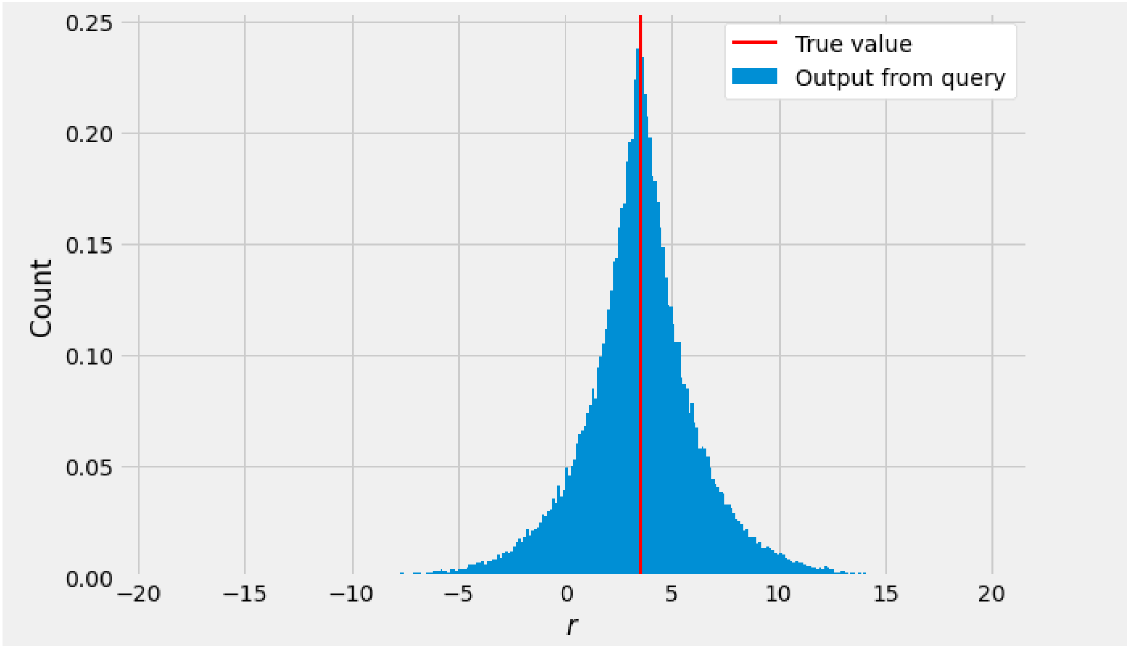
We choose the utility function to be $u(x, r) = -|f(x) - r|$ with $r \in \mathbb{R}$. Substituting into the exponential mechanism, we obtain an algorithm with conditional probability

$$P(r|x; f, \alpha) = \frac{\alpha}{2} e^{-\alpha|f(x)-r|} \,.$$

In this example we pick $f(x)$ to be the identity. We then only need to define the utility function. In the following we show a (normalized) histogram of the result of performing the query with Laplace noise repeatedly. As we can see, it gives the Laplace distribution centered at the true value $f(x)$.

[7]:
```python
def u_laplacian(x, r):
    '''
    Utility function for Laplacian mechanism

    Arguments:
        x: float.
        r: array of reals.
    '''
    output = -np.absolute(x - r)
    return output

# Define some example values:
r = np.arange(-20, 20, 0.001) # Set the interval of possible outputs r
x = 3.5                       # Set a value for the dataset
delta_u = 1                   # We simply set it to one
```

```
epsilon = 1                         # Set a value for epsilon
size = 100000                       # We want to repeat the query this many times

node = DataNode()                   # Create a node
node.set_private_data(name="identity", data=np.array(x)) # Store the database x␣
 ↪in the node

data_access_definition = ExponentialMechanism(u_laplacian, r, delta_u, epsilon,␣
 ↪size)
node.configure_data_access("identity", data_access_definition)
result = node.query("identity")

fig, ax = plt.subplots(figsize=(9,6))
plt.hist(result, bins = int(round(np.sqrt(len(result)))), density = True,␣
 ↪label="Output from query")
plt.axvline(x=x, linewidth=2, color='r', label= "True value")
ax.set_xlabel('$r$')
ax.set_ylabel('Count')
ax. legend(facecolor='white', framealpha=1, frameon=True)
label="Laplace mechanism: normalized histogram showing 10'000 identity queries,␣
 ↪for $\Delta u = 1$ and $\epsilon = 1$. \n\
Due to the noise added by the Laplace mechanism, the values returned by the␣
 ↪query recover\n\
the Laplace distribution centered at the true value."
ax.text((r.max()+r.min())/2, -0.07, label, ha='center')

plt.show()
```



Laplace mechanism: normalized histogram showing 10'000 identity queries, for $\Delta u = 1$ and $\varepsilon = 1$. Due to the noise added by the Laplace mechanism, the values returned by the query recover the Laplace distribution centered at the true value.