CHW1-Report

Saman Soleimani 400206284

Question 1: I used the attack mentioned in 1, designed by [2]. The attacker, besides the query on the dataset, only knows y and z. The one that should be checked is whether it is in the dataset and the reference sample, sampled from the population, respectively. So, d is the dimension of the query, δ is the statistical significance, τ is the critical value, and the difference of the inner product of y and z with the query is the statistic that is used.

$\mathcal{A}_{\delta,d}(y,q,z)$

1. Input: $y,z \in \{\pm 1\}^d$ and $q \in [-1,1]^d$.
2. Compute $\langle y,q\rangle = \sum_{j\in[d]} y^j \cdot q^j$ and $\langle z,q\rangle = \sum_{j\in[d]} z^j \cdot q^j$.
3. If $\langle y,q\rangle - \langle z,q\rangle > \tau := \sqrt{8d\ln(1/\delta)}$, output IN; otherwise output OUT.

Figure 1: Hypothesis test of the membership inference attack

I set $d = 2000$, $\delta = 0.05$.Based on 1, the critical value should be 218.93.

```
significance_level = 0.05
dimension = 2000
critical_value = np.sqrt(8 * dimension * np.log(1 /
significance_level))

```

I used the PUMS as the population. Because the algorithm needs the boolean data, we should hash the data. For this purpose, I used boolean projection function from [1].

```
import numpy as np
import pandas as pd

def make_boolean_projection(d_in, d_out=1):
  """Returns a (pseudo)random vector predicate function by
hashing data."""
  prime = 691
  desc = np.random.randint(prime, size=(d_in, d_out))
  """this predicate maps data into a ndarray of booleans of
size [n, d_out]        (where '@' is the dot product and '%'
modulus)"""
  return lambda data: ((data.values @ desc) % prime % 2).astype
(bool)

population: pd.DataFrame = pd.read_csv(
"https://raw.githubusercontent.com/opendp/cs208/main/spring2022
/data/FultonPUMS5sample100.csv")

projection = make_boolean_projection(len(population.columns),
d_out=dimension)

```

In each simulation, I took 5 samples as the dataset, and the reference sample should be taken randomly from the population. Under the null hypothesis, the target sample should be taken from the dataset. The utility function simulates output test statistics for a random target sample from the dataset or outside of the dataset and a random reference sample from the population.

```
    import numpy as np
    import pandas as pd

    import numpy as np
    import pandas as pd

    def random_dataset(data, sample_size):

      """Returns ‘sample_size‘ rows randomly from ‘data‘, and the
    rest"""
      indices = np.arange(len(data))
      np.random.shuffle(indices)

      sampled_data = data.iloc[indices[:sample_size]]
      remaining_data = data.iloc[indices[sample_size:]]

      return sampled_data, remaining_data

     def simulating_function(population,hypothesis="null",
    sample_size=5):

       dataset, rest_population = random_dataset(population,
    sample_size)


       projected_query = projection(dataset).mean(axis=0)


       projected_reference_sample = projection(population.sample(1)
    .iloc[0])

       def get_statistics(projected_target_sample, projected_query,
     projected_reference_sample):

         projected_target_sample = projected_target_sample * 2 - 1
         projected_query = projected_query * 2 - 1
         projected_reference_sample = projected_reference_sample *
    2 - 1


         target_similarity = projected_target_sample.dot(
    projected_query)
         reference_similarity = projected_reference_sample.dot(
    projected_query)
         statistics = target_similarity - reference_similarity
         return statistics

       if hypothesis=="null":
         projected_target_sample = projection(dataset.sample(1).
    iloc[0])
         statistics = get_statistics(projected_target_sample,
    projected_query, projected_reference_sample)


```

```
        elif hypothesis=="both":
          statistics = []
          projected_target_sample = projection(dataset.sample(1).
    iloc[0])

           statistics.append(get_statistics(projected_target_sample,
    projected_query, projected_reference_sample))
          projected_target_sample = projection(rest_population.
    sample(1).iloc[0])
          statistics.append(get_statistics(projected_target_sample,
    projected_query, projected_reference_sample))

        else:
          projected_target_sample = projection(rest_population.
    sample(1).iloc[0])
          statistics = get_statistics(projected_target_sample,
    projected_query, projected_reference_sample)

        return statistics


```

I used "seaborn" package for defining graph function.

```
    import seaborn as sns

    def graph_function(df):
    plot = sns.histplot(df,kde=True)
    plot.set_xlabel("test statistics")
    plot. axvline(x=critical_value)


```

I conduct 1000 simulations under the null hypothesis and plot the histogram of the test statistics.

```
  def simulate(population,hypothesis="null",dimension=100,
    sample_size=5,num_simulations = 1000):
    all_statistics =[]
    for _ in range(num_simulations):
    all_statistics.append(simulating_function(population,hypothesis
    ,dimension,sample_size))
    return all_statistics

    null_list = simulate(population,hypothesis="null",dimension
    =100,sample_size=5,num_simulations = 1000)

    dict = {'null_distribution': null_list}

    df = pd.DataFrame(dict)

    graph_function(df)


```

2 shows the null distribution partitioned by the critical region. Statistics greater than the critical value will be correctly accepted, while statistics less than that will lead to the incorrect rejection of the null hypothesis.
Now I simulate the complete attack. In one graph two distribution of the test statistics under the null hypothesis and alternative hypothesis is shown.

```
        all_list = simulate(population,hypothesis="both",dimension
    =100,sample_size=5,num_simulations = 1000)
```

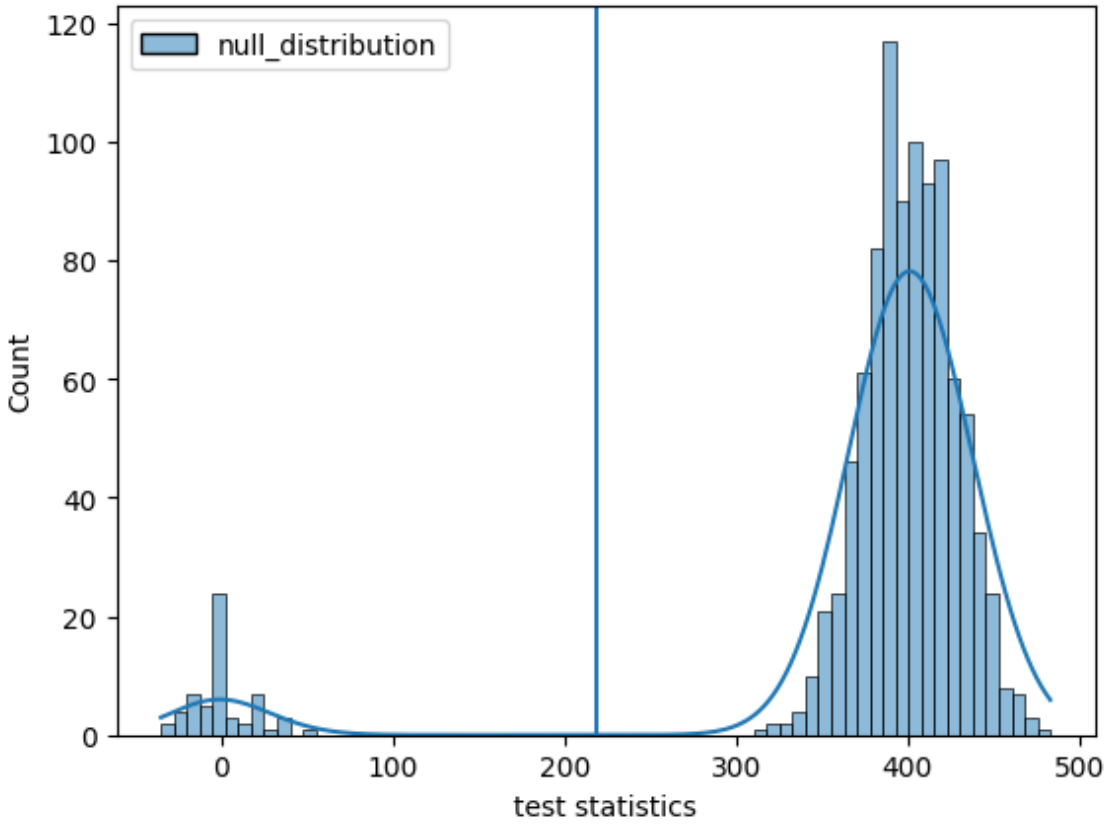

Figure 2: Null distribution of the test statistics

```
    dict = {'in_dataset': [l[0] for l in all_list], 'out_dataset
': [l[1] for l in all_list]}

    df = pd.DataFrame(dict)

    graph_function(df)


```

3 illustrates the discrimination of test statistics under two hypotheses. The null hypothesis posits that the observation (target sample) is in the dataset, while the alternative hypothesis suggests that the observation does not belong to the dataset. This graph demonstrates the effectiveness of hypothesis testing, as the proportion of samples incorrectly classified as belonging to or not belonging to the dataset is very small.

Question 2: First, I built the necessary functions, including a random sampler, calmp, mean query, and Laplace mechanism applied to the mean query.

```
def random_sampler(data,sample_size):
  indices = np.arange(len(data))
  np.random.shuffle(indices)
  sampled_data = data.iloc[indices[:sample_size]]
  return sampled_data
def clip(dataset,min_margin,max_margin):
  return dataset.clip(min_margin,max_margin)
def mean_query(dataset):
  return dataset.mean()
def lap_noise(query,scale):
  return np.random.laplace(query, scale)
```

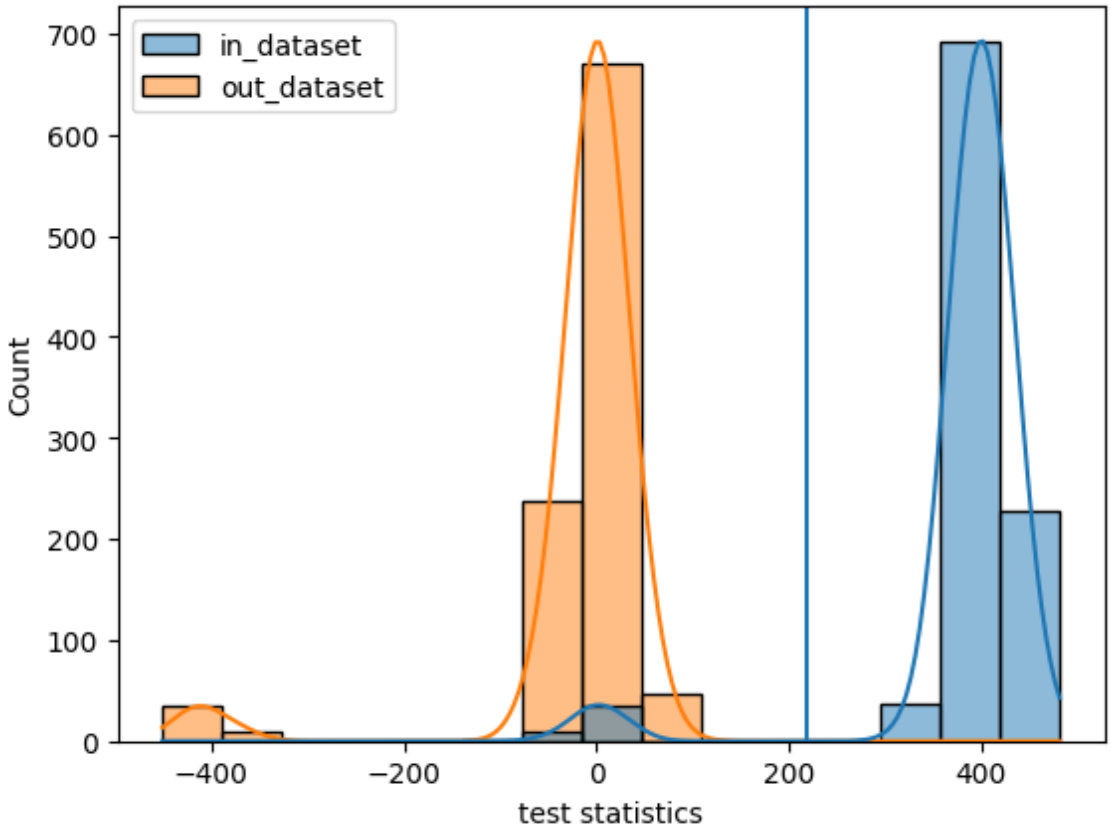

Figure 3: Null distribution of the test statistics

```
    def lap_mech(dataset,epsilon,sensitivity,min_margin=0,
max_margin=20):
        scale = sensitivity/epsilon
        cliped_dataset = clip(dataset,min_margin,max_margin)
        query = mean_query(cliped_dataset)
        private_query = lap_noise(query,scale)
        return private_query, query

```

Next, I downloaded the CSV file and converted it into a Pandas DataFrame. I sampled from the "educ" column of the DataFrame and created a dataset with a size of 500.

```
    np.random.seed(seed=42)
    sampled_data_frame = random_sampler(data,500)
    sampled_data_set = sampled_data_frame["educ"]

```

I employed Hamming distance, resulting in a sensitivity of 2 for the histogram. Additionally, I assumed $\epsilon = 0.5$, making the scale of the Laplace noise 4.

```
    # I used hamming distance for neighboring definition, so the
sensitivity of the histogram will be 2
    np.random.seed(seed=42)
    epsilon = 0.5
    sensitivity = 2
    scale = sensitivity/epsilon
    max_margin =20
    min_margin = 0
    hist, bins = np.histogram(sampled_data_set, bins =np.arange(
min_margin,max_margin))
```

```
    private_hist = lap_noise(hist,scale).round()

    print("Educational level counts:\n", hist)
    print("DP Laplace Educational level counts:\n", private_hist)

    df = pd.DataFrame({'bins': bins[:-1], 'hist': hist, 'private
    hist': private_hist})
    df.plot(x="bins", y=["hist", "private hist"], kind="bar", rot
    =0)

```

The true histogram and the private histogram are depicted in 4.

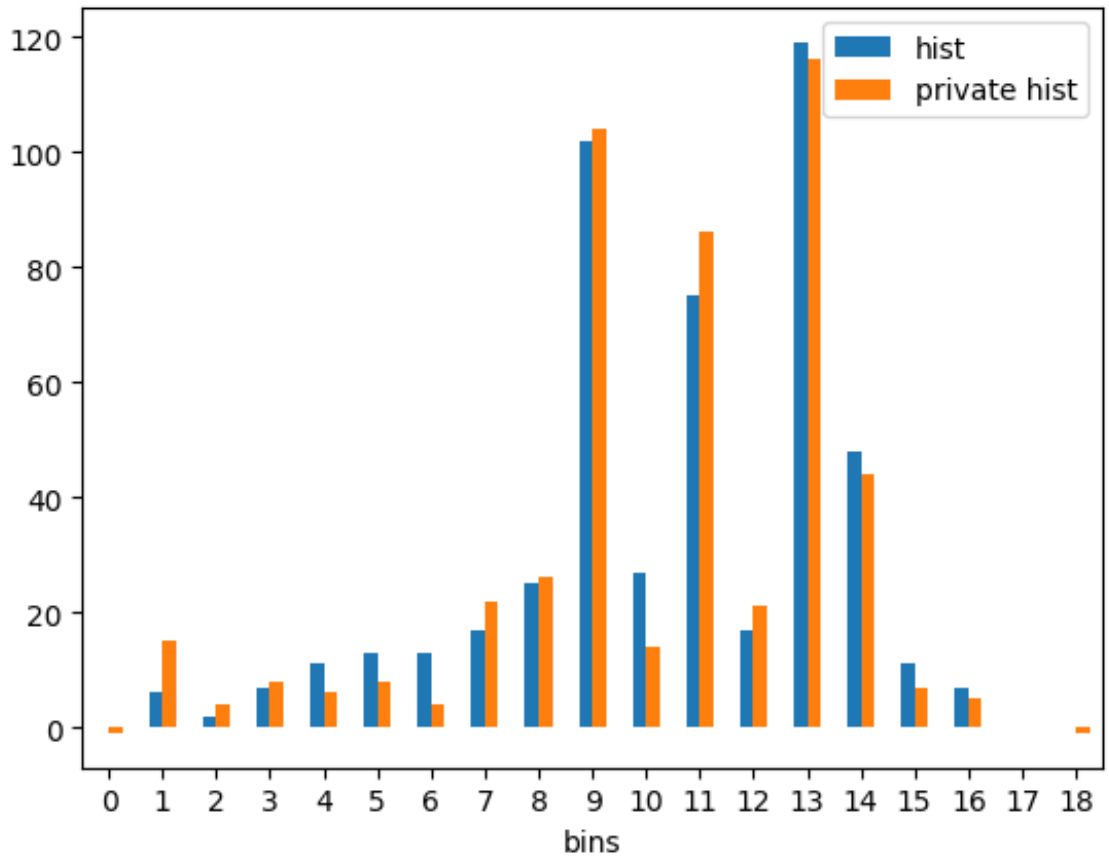

Figure 4: The true histogram and the private histogram with a Laplace noise scale of 4.

Now I want to use the bootstrap method to sample from the empirical probability mass function of the dataset to create new datasets. First, I grasped the empirical probability mass function.

```
    val, cnt = np.unique(sampled_data_set, return_counts=True)
    pmf = cnt / len(sampled_data_set)
    df = pd.DataFrame({'values': val, 'probability': pmf})
    plot = df.plot(x="values", y=["probability"], kind="bar", rot
    =0)
    plot.set_xlabel("Emprical PMF of the dataset")

```

5 shows the empirical probability mass function.
With the assistance of this distribution, I defined a function for bootstrapping and the Laplace mechanism.

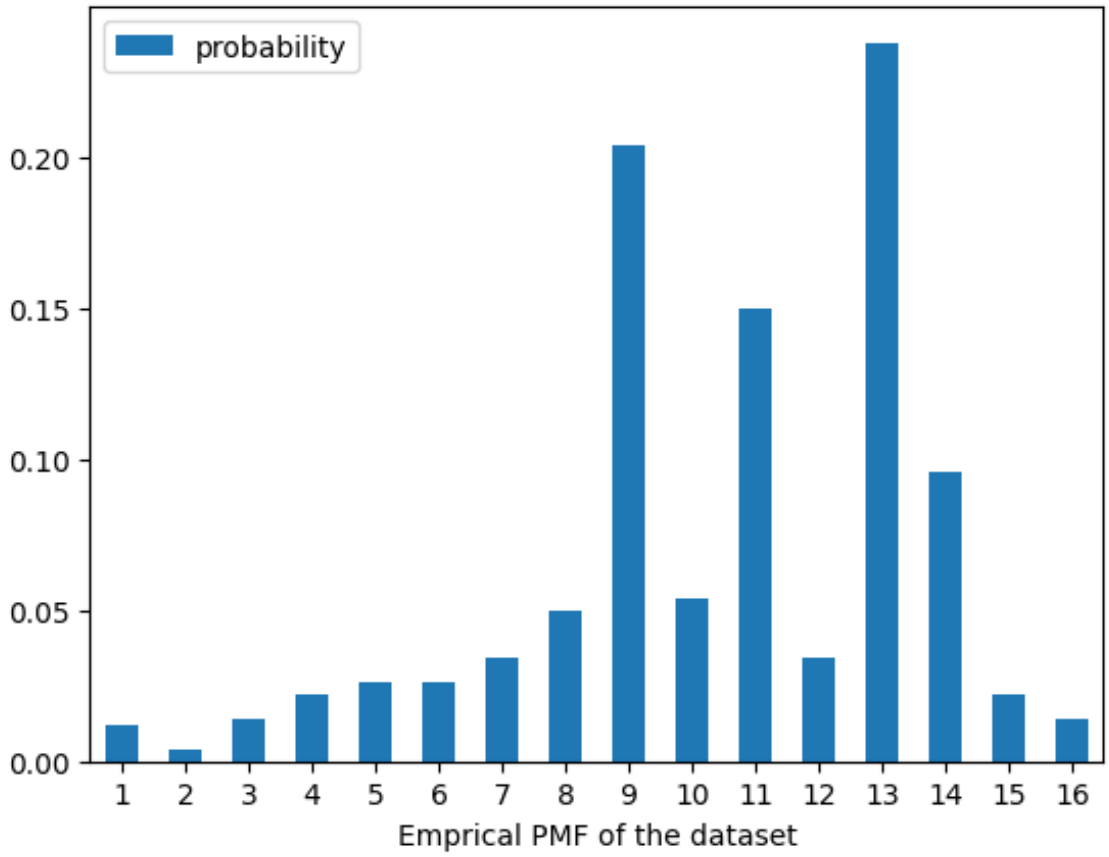

Figure 5: Empirical PMF of the dataset

```
np.random.seed(seed=42)
def boot_strap_laplace(val,pmf,epsilon,dataset_size,
simulation_number,min_margin=0,max_margin=20) :
  sensitivity = (max_margin - min_margin)/dataset_size
  private_means = []
  true_means = []
  for _ in range(simulation_number):
  rand_data = np.random.choice(val, dataset_size, p=pmf)
  private_mean, true_mean = lap_mech(rand_data,epsilon,
sensitivity,min_margin,max_margin)
  private_means.append(private_mean)
  true_means.append(true_mean)
  return sum(private_means)/simulation_number,sum(true_means)/
simulation_number

```

6 plots the error of the Laplace mechanism for the mean query with ϵ = [0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0] and dataset sizes = $[5, 10, 20, 50, 100, 200, 500, 1000]$, with a simulation number of 1000 for each ϵ and dataset size.

```
np.random.seed(seed=42)
epsilons = [0.01,0.05,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.]
dataset_sizes = [5,10,20,50,100,200,500,1000]
df = pd.DataFrame(index=epsilons)
for dataset_size in dataset_sizes:
  all_private_mean = []
  for epsilon in epsilons:
    all_private_mean.append(boot_strap_laplace(val,pmf,epsilon,
```

```
    dataset_size,simulation_number=1000))
        error = [abs(mean[0] - mean[1]) for mean in all_private_mean]
        df[f'Dataset Size={dataset_size}'] = error

    plot = df.plot()
    plot.set_xlabel("Epsilon")
    plot.set_ylabel("Absolute Error")

```

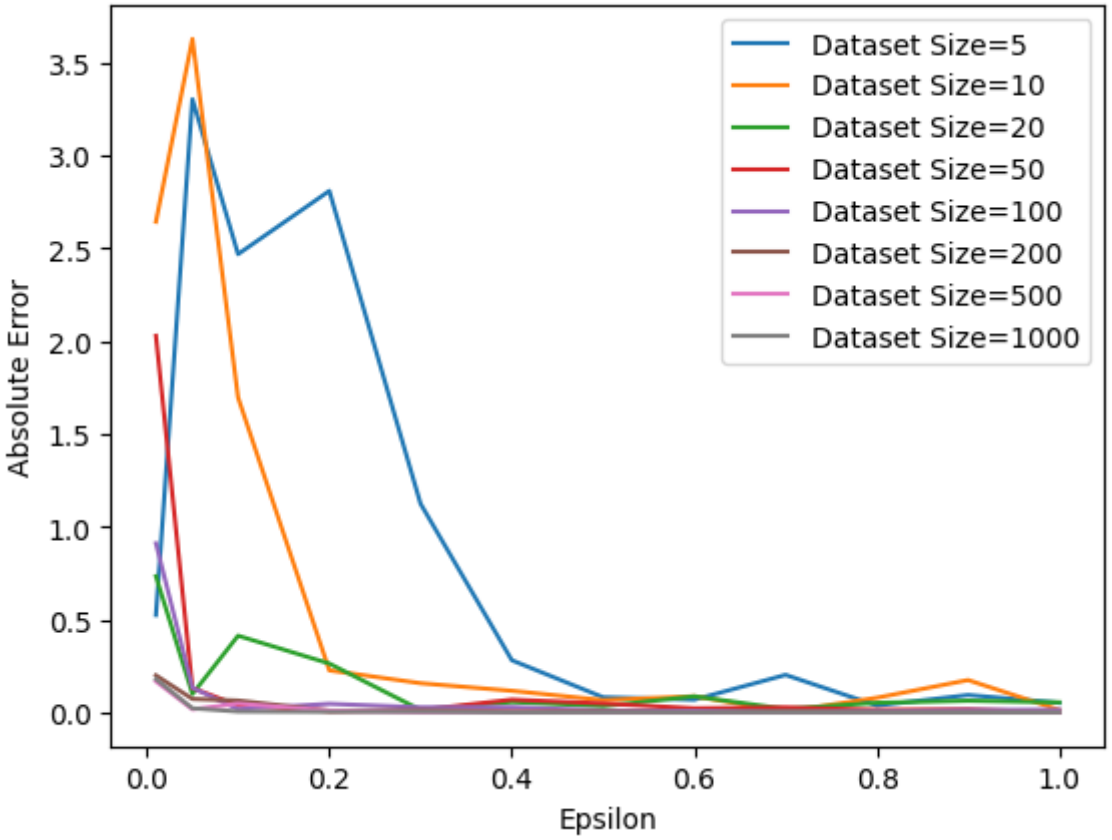

Figure 6: Absolute error for the range of epsilons and dataset sizes

Question 3: First I imported packages I need.

```
    import opendp
    from  opendp import prelude as dp
    from opendp.mod import enable_features
    enable_features('contrib')

```

Then, I downloaded the CSV file and converted the column "educ" to a list. I took this step because when using `"dp.t.make_split_dataframe(separator=",", col_names=col_names) >> dp.t.make_select_column(key="educ", TOA=str)"`, it was selecting the wrong column, not the "educ" column.

```
    data_list = data["educ"].tolist()

```

utilized the input space with the input domain `"dp.atom_domain(T=int)"` and input metric `"dp.symmetric_distance()"`. Subsequently, I applied `"then_cat_default"` to change the type to float, followed by `"then_clamp"` to restrict the data between 0 and 20. Using `"then_resize"` I created a dataset of 500 samples. Finally, I employed `"then_mean"` to generate the `"sized_bounded_mean"` For the creation

of "base_laplace" I utilized a scale of 0.5 and set the output domain and output metric of "sized_bounded_mean" as the input domain and input metric for "base_laplace" respectively.

```
    min_margin = 0.
    max_margin = 20.
    edu_bounds =( min_margin,max_margin)
    count_release = 500

    input_space = dp.vector_domain(dp.atom_domain(T=int)), dp.
   symmetric_distance()
    preprocess = (input_space >>
    dp.t.then_cast_default(TOA=float) >>
    dp.t.then_clamp(bounds=edu_bounds) >>
    dp.t.then_resize(size=count_release, constant=10.))

    true_mean = dp.t.then_mean()
    sized_bounded_mean = preprocess >> true_mean

    base_laplace = dp.m.make_laplace(sized_bounded_mean.
   output_domain, sized_bounded_mean.output_metric, scale=0.5)

```

So, the differentially private mean of the dataset, comprising 500 data points, can be shown below.

```
    # chain with the base_laplace measurement
    dp_mean= sized_bounded_mean >> base_laplace

    # release a dp mean
    dp_mean(data_list)

```

Then, I check if the distance of inputs of the "base_laplace" is 0.15 based on the input metric (sensitivity). The output distance, based on the output metric (ϵ), will be 0.3 or not.

```
    # Check that when sensitivity is 0.15, the privacy usage is .30
    base_laplace.check(d_in=.15, d_out=.30)

```

The output is `True`. Therefore, with this sensitivity, $\epsilon = 0.3$ is achieved because the scale is set to 0.5. If we increase the sensitivity beyond 0.15, we can no longer achieve $\epsilon = 0.3$. For instance, setting sensitivity to 0.20 results in an epsilon value greater than 0.4. The following code yields `False`.

```
    base_laplace.check(d_in=.2, d_out=.30)

```

If I set `max_influence`, it means that each person only affects one data point. By mapping the transformation `sized_bounded_mean`, we obtain the sensitivity of this transformation. The following code checks whether, when `max_influence` is set to 1, the sensitivity is equal to or lower than 0.15.

```
    max_influence = 1
    eps = 0.3
    sensitivity = 0.15
    sized_bounded_mean.check(d_in=max_influence, d_out=sensitivity)


```

The code above returns `True`. Now, we examine the privacy expenditure of the entire Laplace mechanism when `max_influence` is set to one. The following code also outputs `True`.

```
dp_mean.check(d_in=max_influence, d_out=eps)

```

I wrote the code below to find the true histogram and private histogram of a random dataset with a sample size of 500, using the `"then_count_by_categories"` and `"OpenDP"` package.

```
# release a histogram with laplace noise
min_margin = 0
max_margin = 20
categories = [str(category) for category in np.arange(min_margin, max_margin)]
input_space = dp.vector_domain(dp.atom_domain(T=int)), dp.symmetric_distance()
histogram = (input_space >> dp.t.then_cast_default(TOA=float) >>
dp.t.then_resize(size=count_release, constant=10.)>>
dp.t.then_cast_default(TOA=str) >>
dp.t.then_count_by_categories(categories=categories))
noisy_histogram = histogram >> dp.m.then_laplace(scale=0.5)
hist = histogram(data_list)
private_hist = noisy_histogram(data_list)

print("Educational level counts:\n", hist[:-1])
print("DP Laplace Educational level counts:\n", private_hist[:-1])

df = pd.DataFrame({'bins': np.arange(min_margin, max_margin), 'hist': hist[:-1], 'private hist': private_hist[:-1]})

df.plot(x="bins", y=["hist", "private hist"], kind="bar", rot=0)

```

7 plots true and private histogram with Laplace noise scale of 0.5.

Now I do the same for geometric noise of scale 0.5. 8 plots true and private histogram with Geometric noise scale of 0.5.

Question 4: I followed the same procedure as in `Question2`, but I constructed the Gaussian mechanism for this question. First, I built the necessary functions, including a random sampler, clamp, mean query, and the Laplace mechanism applied to the mean query.

```
def random_sampler(data,sample_size):
  indices = np.arange(len(data))
  np.random.shuffle(indices)
  sampled_data = data.iloc[indices[:sample_size]]
  return sampled_data
def clip(dataset,min_margin,max_margin):
  return dataset.clip(min_margin,max_margin)
def mean_query(dataset):
  return dataset.mean()
def gaussian_noise(query,variance):
  return np.random.normal(query, np.sqrt(variance))
def gaussian_mech(dataset,epsilon,sensitivity,min_margin=0,max_margin=20,delta=1e-8):
  variance = (2*np.power(sensitivity,2)*np.log(2/delta))/np.power(epsilon,2)
```

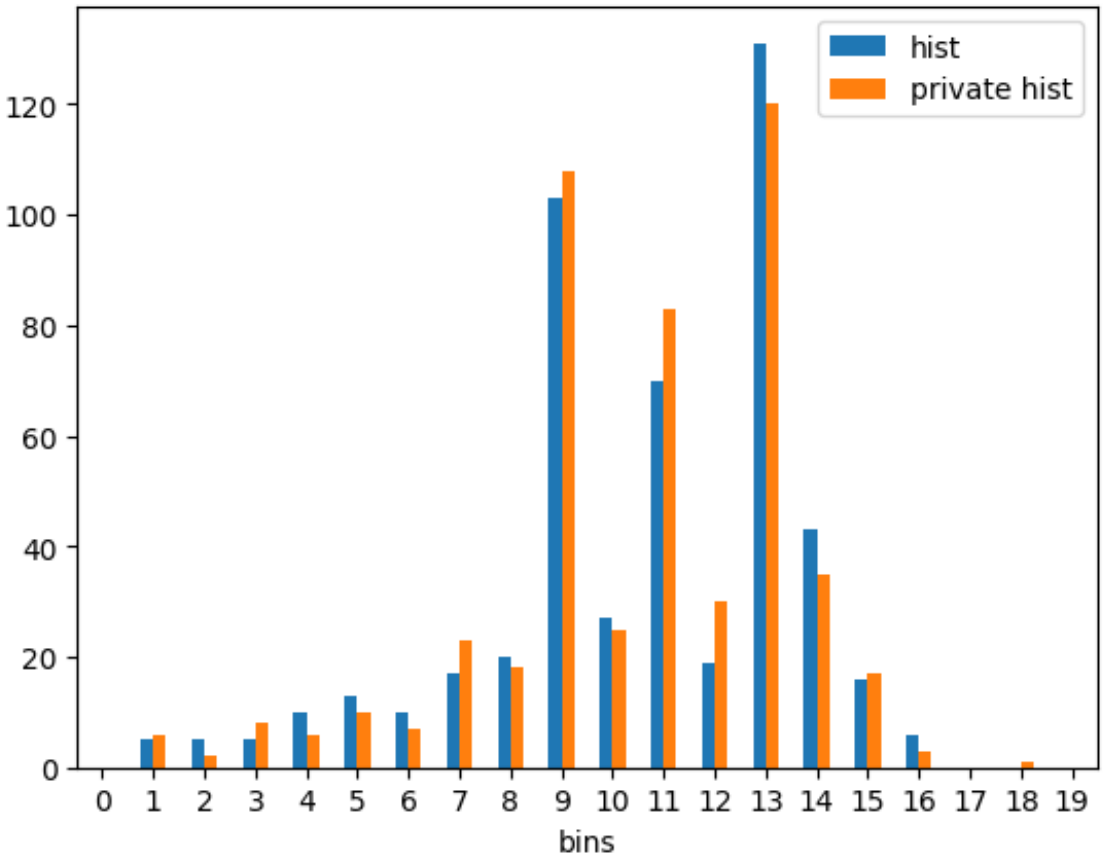

Figure 7: The true histogram and the private histogram generated by `OpenDP` with a Laplace noise scale of 0.5.

```
    cliped_dataset = clip(dataset,min_margin,max_margin)
    query = mean_query(cliped_dataset)
    private_query = gaussian_noise(query,variance)
    return private_query, query

```

Subsequently, I obtained the CSV file and converted it into a Pandas DataFrame. I then extracted a sample from the "educ" column, resulting in a dataset of 500 entries. Now, I aim to employ the bootstrap method to sample from the empirical probability mass function of the dataset and generate new datasets.

```
np.random.seed(seed=42)
sampled_data_frame = random_sampler(data,500)
sampled_data_set = sampled_data_frame["educ"]
val, cnt = np.unique(sampled_data_set, return_counts=True)
pmf = cnt / len(sampled_data_set)
df = pd.DataFrame({'values': val, 'probability': pmf})
plot = df.plot(x="values", y=["probability"], kind="bar", rot
=0)
plot.set_xlabel("Emprical PMF of the dataset")

```

[language=Python] 9 shows the empirical probability mass function. Utilizing this distribution, I formulated a function for bootstrapping, incorporating both Laplace and Gaussian mechanisms with a parameter $\delta = 1e - 8$.

```
np.random.seed(seed=42)
def boot_strap_gaussian_laplace(val,pmf,epsilon,dataset_size,
simulation_number,min_margin=0,max_margin=20) :
```

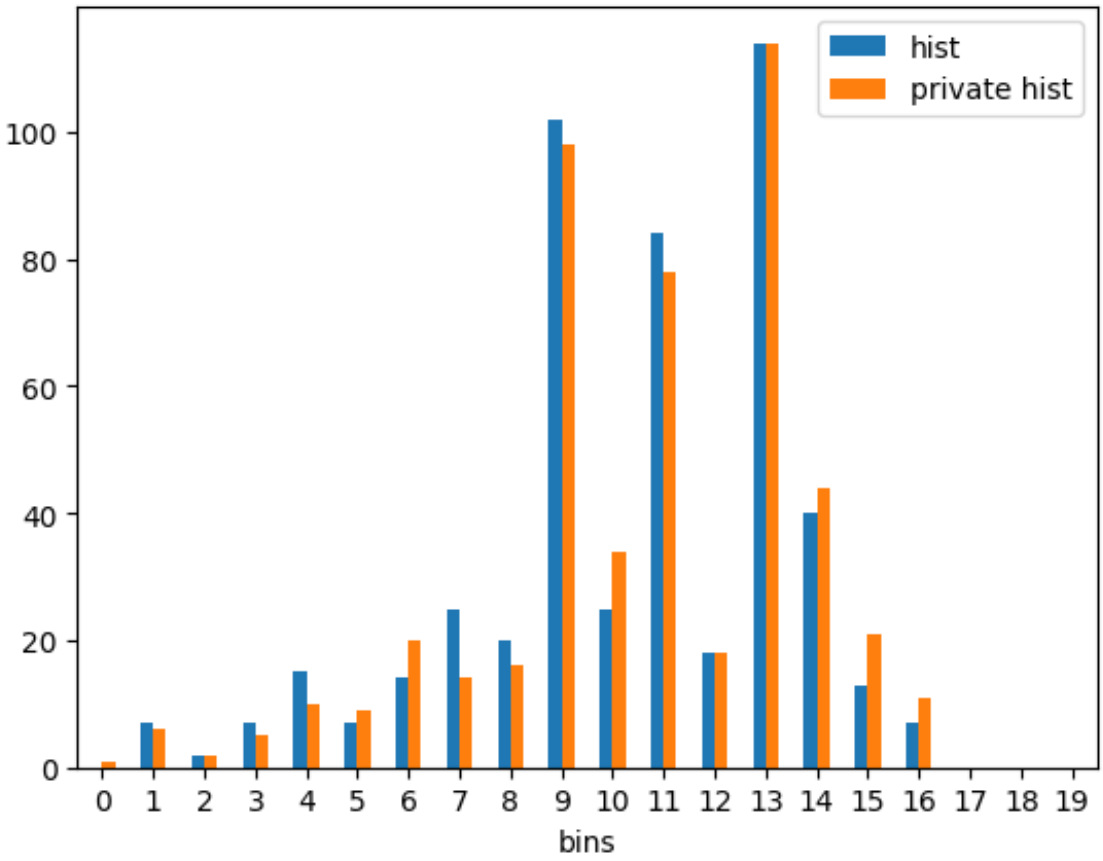

Figure 8: The true histogram and the private histogram generated by `OpenDP` with a Geometric noise scale of 0.5.

```
    sensitivity = (max_margin - min_margin)/dataset_size
    private_means_gaussian = []
    private_means_laplace = []
    true_means = []
    for _ in range(simulation_number):
      rand_data = np.random.choice(val, dataset_size, p=pmf)
      private_mean_gaussian, true_mean = gaussian_mech(rand_data,
epsilon,sensitivity,min_margin,max_margin)
      private_mean_laplace, _ = lap_mech(rand_data,epsilon,
sensitivity,min_margin,max_margin)
      private_means_gaussian.append(private_mean_gaussian)
      private_means_laplace.append(private_mean_laplace)
      true_means.append(true_mean)
   return sum(private_means_gaussian)/simulation_number,sum(
private_means_laplace)/simulation_number,sum(true_means)/
simulation_number

```

In Figure 10, the error of the Laplace mechanism and Gaussian Mechanism for the mean query is plotted with $\epsilon = [0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]$ and dataset sizes $= [5, 10, 20, 50, 100, 200, 500, 1000]$. The simulation is conducted 1000 times for each ϵ and dataset size.

```
np.random.seed(seed=42)
epsilons = [0.01,0.05,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.]
dataset_sizes = [5,10,20,50,100,200,500,1000]
df = pd.DataFrame(index=epsilons)
for dataset_size in dataset_sizes:
```

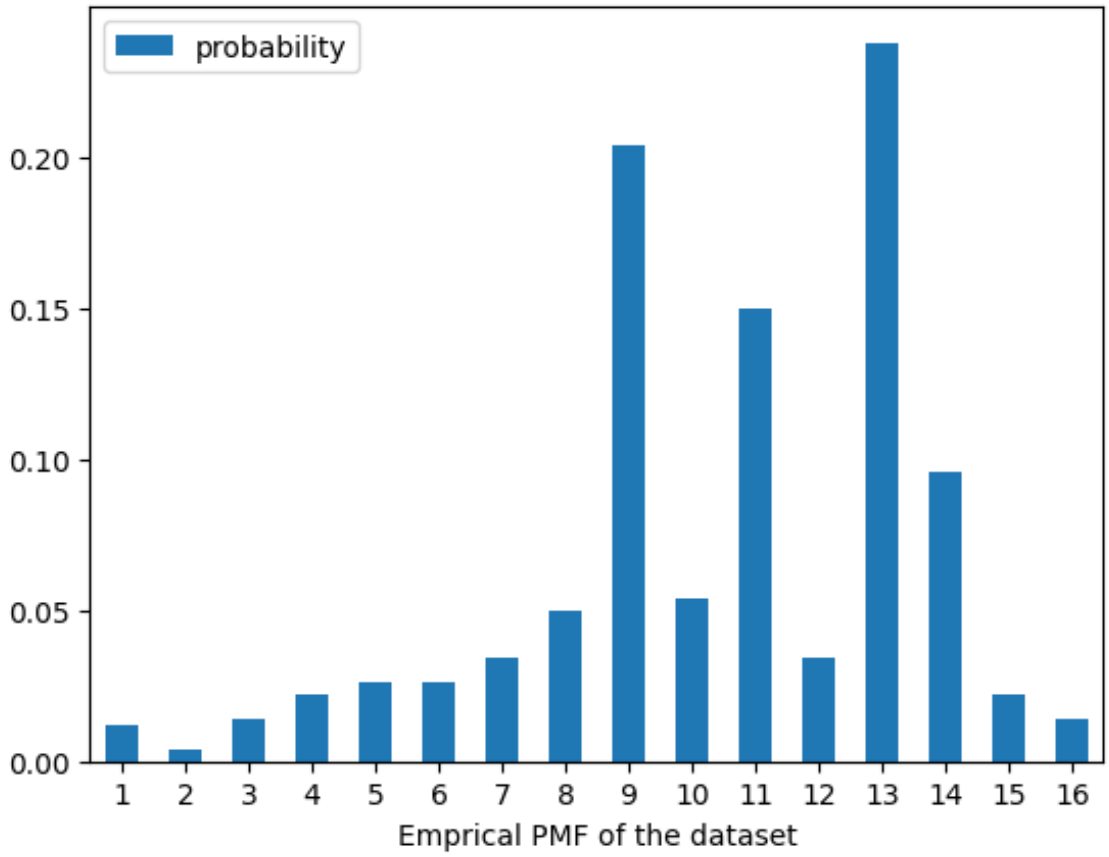

Figure 9: Empirical PMF of the dataset

```
    all_private_mean = []
    for epsilon in epsilons:
      all_private_mean.append(boot_strap_gaussian_laplace(val,pmf
,epsilon,dataset_size,simulation_number=1000))

    error_gaussian = [abs(mean[0] - mean[2]) for mean in
all_private_mean]
    error_laplace = [abs(mean[1] - mean[2]) for mean in
all_private_mean]
    df[f'Dataset Size(gaussian,delata=1e-8)={dataset_size}'] =
error_gaussian
    df[f'Dataset Size(laplace)={dataset_size}'] = error_laplace

 plot = df.plot(figsize=(10, 10))
 plot.set_xlabel("Epsilon")
 plot.set_ylabel("Absolute Error")

```

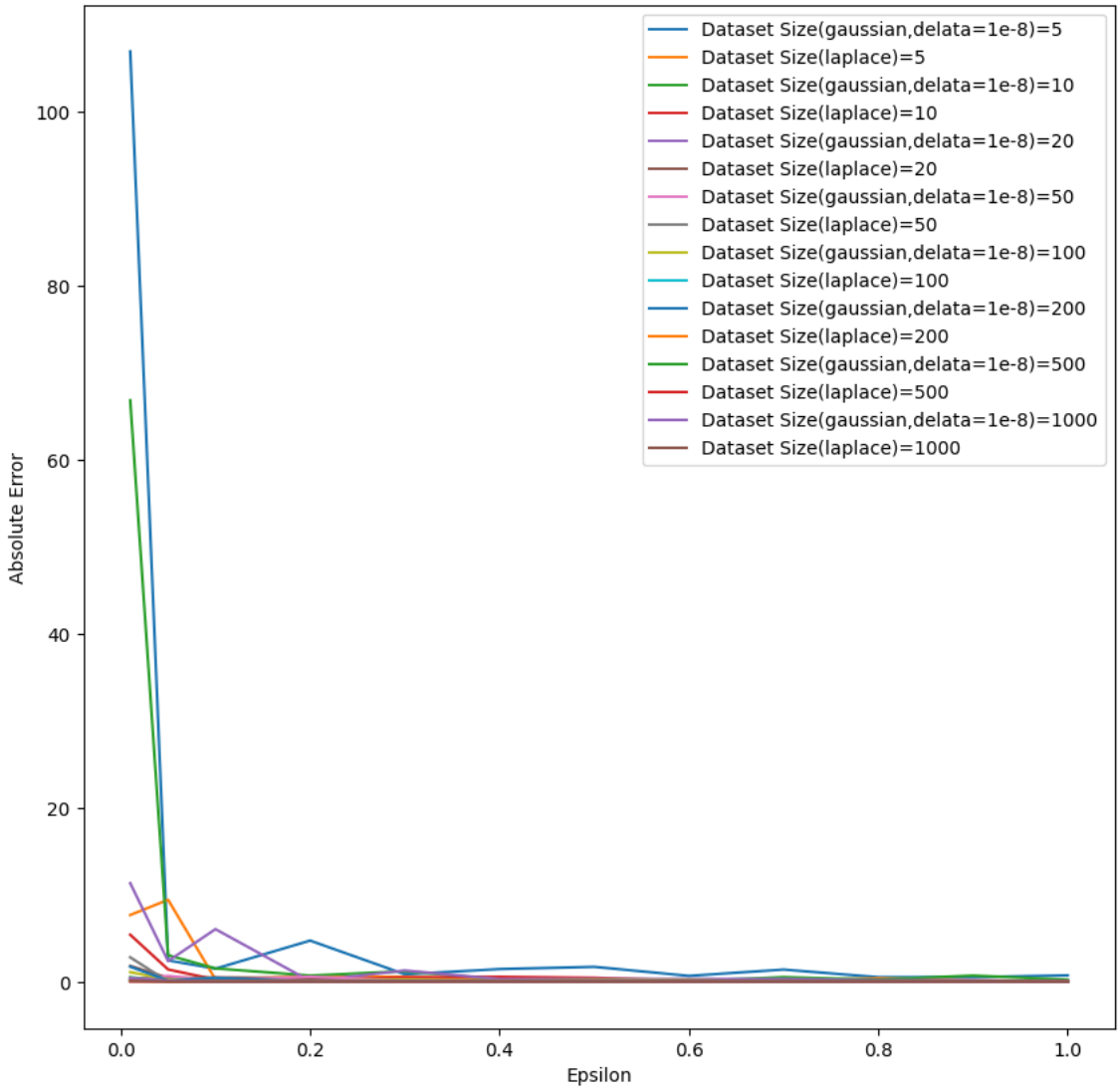

Figure 10: Absolute error of the Laplace and Gaussian mechanisms for the range of epsilons and dataset sizes

References

[1] OpenDP. `https://docs.opendp.org/en/stable/examples/attacks/membership.html`. Accessed: 2023-12-14.

[2] Cynthia Dwork, Adam Smith, Thomas Steinke, Jonathan Ullman, and Salil Vadhan. Robust traceability from trace amounts. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 650–669. IEEE, 2015.