

Data Privacy in Practice: Final Project Assignment

Group Members:

Basel Al Musalhi (19-0398)

Hamood Al Marhoubi (19-0503)

Nadir Al Batrani (19-0309)

Omar Al Hudaifi (19-0508)

Due: January 6, 2024

Professor: Dr. Abdelaziz Korba
Winter Semester 2023

Importing the Required Data and Libraries.....	5
Code Snippet	5
Description of the imports.....	5
1. Implement a linking attack to recover Brenn's data from the adult deid dataset.....	6
1.1 Task 1 - Part 1.....	6
1.2 Task 1 - Part 2.....	6
1.3 Task 1 - Part 3.....	7
1.4 Task 1 Summary	7
2. Conduct a differencing attack to determine Brenn McNeely's age using only the mean aggregation function over large groups	8
2.1 Task 2 - Part 1.....	8
2.2 Task 2 - Part 2.....	9
2.3 Task 2 - Part 3.....	9
2.4 Task 2 Summary	10
3. Alter the Zip column in the adult dataset to achieve k-Anonymity for k=2	11
3.1 Scenario 1	11
3.1.1 Code.....	11
3.1.2 Explanation	12
3.2 Scenario 2	12
3.2.1 Code.....	12
3.2.2 Explanation	13
3.3 Scenario 3	13
3.3.1 Code.....	14
3.3.2 Explanation	15
4. Determine whether Karrie Trusslove's age is 39 using a differentially private counting query. Add Laplace noise to ensure differential privacy with $\epsilon = 1.0$.	15
4.1 Task 4 - Part 1.....	15
4.1.1 code	15
4.1.2 explanation.....	16
4.2 Task 4 - Part 2.....	16
4.2.1 code	16
4.2.2 Explanation	16
5. Conduct a linkage attack using adult pii and adult deid dataframes to recover as many names as possible from adult deid. Parameterize your solution by the columns used in the attack.)	17
5.1 Code	17
5.2 Explanation	18

6. Determine the number of individuals uniquely identified by their Zip code and age in the dataset.....	19
6.1 Code	19
6.2 Explanation	20
7. Calculate a differentially private sum of ages for various values of b, clipped to each b, with $\epsilon = 0.1$.	20
7.1 Code	20
7.2 Explanation	21
8. Develop an algorithm to automatically select a clipping parameter for the age column that satisfies differential privacy.....	22
8.1 Task 8 part 1	22
8.1.1 Code.....	22
8.1.2 Explanation	22
8.2 Task 8 part 2	22
8.2.1 Code.....	22
8.2.2 Explanation	24
9. Create a differentially private contingency table for the Education and Sex columns of the adult dataset.	25
9.1 Code	25
9.2 Explanation	26
References	27

List of Figures

<i>Figure 1 Output of Brenns row</i>	<i>7</i>
<i>Figure 2 Output showing Brenns recovered row</i>	<i>7</i>
<i>Figure 3 Output of grouping Education.....</i>	<i>8</i>
<i>Figure 4 Output of grouping Occupation & Education.....</i>	<i>9</i>
<i>Figure 5 Output of Brenns age.....</i>	<i>10</i>
<i>Figure 6 Output of modified dataset & K-anonymity verification</i>	<i>12</i>
<i>Figure 7 Output of the K-Anonymity Iteration: 'Zip' Column Generalized</i>	<i>13</i>
<i>Figure 8 Output of K-Anonymity Enforcement & Data Generalized.....</i>	<i>15</i>
<i>Figure 9 Linkage Attack Representation.....</i>	<i>19</i>
<i>Figure 10 Output of individuals found by Zip Code & Age</i>	<i>20</i>

Importing the Required Data and Libraries

Code Snippet

```
# Load the data and libraries
import pandas as pd
import numpy as np

adult = pd.read_csv('https://github.com/jnear/cs211-data-privacy/raw/master/homework/adult_with_pii.csv')
```

Description of the imports

1. First, we imported Pandas as pd, a useful open-source library that can be used for Python data analysis and modelling. It has features that make analysing data structures simpler (Great Learning, 2023).
2. Next, we imported Numpy as np, which is typically utilised in scientific computing and essentially offers support for the complex mathematical computations needed for dataset analysis (Great Learning, 2023).
3. We then using pandas have uploaded the dataset adult which be retrieved in github, 'https://github.com/jnear/cs211-data-privacy/raw/master/homework/adult_with_pii.csv'

1. Implement a linking attack to recover Brenn's data from the adult deid dataset

1.1 Task 1 - Part 1

We begin with the actual task in hand where we will first need to start by constructing a version of the dataset that is de-identified

```
def Deidentify_adult(): # We hereby start by constructing a de-identified `adult`
dataset as the following function does...

    return adult.drop(['Name', 'SSN', 'DOB'], axis=1) # We remove only the Personally
Identifiable Information (PII), in this case it is the: Name, SSN, & DOB Columns.
(Zip is quasi identifier)
    raise NotImplementedError()

adult_deid = Deidentify_adult() # Now we simply store the de-identified dataset in
'adult_deid'
```

Now we are simply checking if the 3 columns have successfully been displaced from the 'adult_deid' dataset

```
assert 'Name' not in adult_deid.columns
assert 'DOB' not in adult_deid.columns
assert 'SSN' not in adult_deid.columns
```

Now Since the 3 columns have successfully been displaced from the 'adult_deid' dataset, whereas they were needed to be removed as they were direct identifiers to the dataset while most of the others that weren't removed were indirect identifiers or quasi-identifiers which weren't unique for each individual and couldn't fully de-identify the individuals if removed. According to HIPAA, PHI, & PII: Institutional Review Board (IRB) Office - Northwestern University.

1.2 Task 1 - Part 2

We then will make an expression that will return just the row containing information about Brenn McNeely. Since the goal would be to perform a linking attack and retrieve Brenns data from the de-identified data set earlier, through a linking attack. This following part would basically retrieve the original data of Brenn.

```
def Get_brenns_row(): # This function will get Brenns row from the 'adult' DataFrame
    return adult[adult['Name'] == 'Brenn McNeely'] # Here we are filtering the Dataframe
to find Brenn from the 'Name' column
brenns_row = Get_brenns_row()[['Name', 'DOB', 'Zip', 'Age']] # Function is called and
the following: (name, DOB, ZIP, and Age) all will be stored the in 'brenns_row'
```

```
assert len(brenns_row) == 1 # Ensuring that brenns row is only 1 of length and that
only 1 would matche the rows conditions
assert brenns_row['Zip'].iloc[0] == 95668 # Ensuring that Brenns 'Zip' is 95668
assert brenns_row['Age'].iloc[0] == 38 # Ensuring that Brenns 'Age' is 38
```

```
brenns_row # We simply display brenns row
```

	Name	DOB	Zip	Age
2	Brenn McNeely	8/6/1991	95668	38

Figure 1 Output of Brenns row

Which shows us Brenn's row clearly so it clarifies that the original data of Brenn was retrieved successfully.

1.3 Task 1 - Part 3

Now for the Main part of the task, where we are going to conduct the linking attack to recover Brenn's data from the 'adult_deid' dataset. To start with in this part we acknowledge that we have some auxiliary information including the Zip and Age of Brenn

```
def Recover_brenns_row(): # This function will recover brenns row
    # Auxilary info
    zipofBrenn = int(brenns_row['Zip'].iloc[0]) # Here Brenns Zipcode was extracted
    from the Dataset
    ageofBrenn = int(brenns_row['Age'].iloc[0]) # Here Brenns Age was extracted from
    the Dataset
    return adult_deid[(adult_deid['Zip']== zipofBrenn) & (adult_deid['Age']==
    ageofBrenn)] # This acts as a filter to find rows where The Zip and Age of Brenn
    match to to the one in the de-identified set

brenns_recovered_row = Recover_brenns_row() # Brenns row here is then called and
recovered from the de-identified dataset.
```

Then we show Brenns recovered row which shows that the linkage attack was successful:

```
brenns_recovered_row # Brenns recovered row showing the linkage attack was successful
```

brenns_recovered_row # Brenns recovered row showing the linkage attack was successful

✓ 0.0s

	Zip	Age	Workclass	fnlwgt	Education	Education-Num	Marital Status	Occupation	Relationship	Race	Sex	Capital Gain	Capital Loss	Hours per week	Country	Target
2	95668	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K

Figure 2 Output showing Brenns recovered row

And just some assertion lines:

1. Checks that only 1 row was returned, showing that there is a unique match in the de-identified dataset.
2. Checks if Brenns Education level actually matches and so therefore it was successful.

```
assert len(brenns_recovered_row) == 1 # Checking that only 1 row was returned,
showing that there is a unique match in the de-identified dataset.
assert brenns_recovered_row['Education'].iloc[0] == 'HS-grad' # Checks if Brenns
Education level actually matches and so therefore it was successful
```

1.4 Task 1 Summary

1. We began by defining the "adult" dataset, which included some PII's by default.

2. After that, the dataset was retrieved and the PII's (Name, SSN, and DOB in particular) were removed in order to de-identify it. At this point, we created a new DataFrame called "adult_deid" that had no PII's, which is typically de-identified to preserve privacy.
3. The next stage was to specify Brenns Data, which was important since it allowed us to identify the exact person we were aiming for and some details about him (auxiliary information), which included his age, ZIP, and DOB
4. Here, with the help of the auxiliary information that was retrieved, we were able to successfully execute the linking attack on the de-identified dataset. The filtering step from the 'adult_deid' DataFrame, where Brenn's zip code and age were matched to fully locate and recover Brenn's De-identified data by matching, was an essential and vital part of this attack. Additionally, each of these steps required testing using assertion commands.

2. Conduct a differencing attack to determine Brenn McNeely's age using only the mean aggregation function over large groups

2.1 Task 2 - Part 1

First we will start by grouping the `adult` dataset by a single column and counting the number of members in each group. This is necessary as It would begin here to give good insights towards the structure of the available information and can therefore play a good part in privacy attacks.

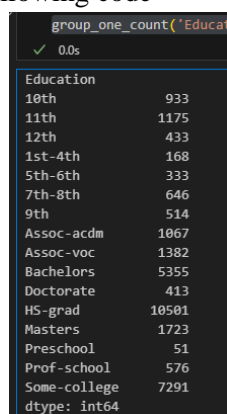
```
def group_one_count(col): # We defined this function so that we can group the 'adult'
dataset by a single column & have the numbers of each entry being counted
    a = adult.groupby(col).size() # We use the size function here to do the counting
    return a

raise NotImplementedError()
```

Basically the function grouped the 'adult' dataset by a single column & the numbers of each entry being counted

```
group_one_count('Education') # Education is hereby grouped by 1 column
```

So the following figure is the output of the following code



Education	Count
10th	933
11th	1175
12th	433
1st-4th	168
5th-6th	333
7th-8th	646
9th	514
Assoc-acdm	1067
Assoc-voc	1382
Bachelors	5355
Doctorate	413
HS-grad	10501
Masters	1723
Preschool	51
Prof-school	576
Some-college	7291

Figure 3 Output of grouping Education

Then it was stored in S as its building up for the final outcome

```
s = group_one_count('Education') # Its stored in 's'
# verification checks... that the levels are as expected
assert s['10th'] == 933
```



```
assert s['9th'] == 514
assert s['Some-college'] == 7291
```

2.2 Task 2 - Part 2

Then we will group the `adult` dataset by two columns and count the number of members in each group. This is also a necessary step when it comes to performing differencing attacks as this can possibly show more available patterns and possible vulnerabilities that can be attacked.

```
def group_two_count(col1, col2): # This function unlike the first would take 2 columns
as input and then it would group the 'adult' Dataframe using the 2 columns
    grouped_data_2 = adult.groupby([col1, col2]).size() #Similar to the first part were
they are counted using the size function and this time its for 2 specified columns
    return grouped_data_2 # Counted data is returned...
```

```
group_two_count('Occupation', 'Education') # The input of the previously named function
here would be 'Occupation' and 'education'
```

The following figure is the output of the following code

Occupation	Education	
Adm-clerical	10th	38
	11th	67
	12th	38
	5th-6th	6
	7th-8th	11
Transport-moving	...	
	Doctorate	1
	HS-grad	825
	Masters	10
	Prof-school	3
	Some-college	283

Length: 201, dtype: int64

Figure 4 Output of grouping Occupation & Education

```
s = group_two_count('Occupation', 'Education') # The 2 columns are here stored inside
's'
assert s['Transport-moving']['Doctorate'] == 1 # Verifying the values matched
assert s['Adm-clerical']['10th'] == 38 # Verifying the values matched
```

Similar to the first part where they are counted using the size function and this time it's for 2 specified columns

2.3 Task 2 - Part 3

Finally, we will perform a differencing attack to determine Brenn McNeely's age using only the `mean` aggregation function over large groups, which is basically the attack. This is the main step involved, as the mean will hereby be calculated before and after the differencing attack. Using that information and getting the difference between those 2 will help us retrieve Brenn's Age.

```
def get_brenns_age():
    # Before Brenn's participation, all of the means are being calculated
    mean_before = adult.loc[adult['Name'] != "Brenn McNeely", 'Age'].mean()
    # Mean After Brenn's participation is now going to be calculated (So this time with
    Brenn)
    mean_after = adult['Age'].mean()
    '''
```

```

Here we perform the Differencing attack to retrieve Brenn's Age
firstly we would get the total age of everyone (Excluding Brenn)
We achieve that by multiplying Adults total number by the mean age (all excluding
Brenn):
'''
sum_age_before = mean_before * len(adult[adult['Name'] != "Brenn McNeely"])
'''
Then we need to calculate the total age of all adults
this is achieved by getting the multiplication of total number of adults by the
mean age (Here it includes Brenn):
'''
sum_age_after = mean_after * len(adult)
'''
The accurate version of Brenns age can be found by subtracting the total age after
and before his participation:
'''
ageBren = sum_age_after - sum_age_before

return ageBren

brenns_age = get_brenns_age() # function gets called to update Brenns age

```

Here we perform the Differencing attack to retrieve Brenn's Age firstly we would get the total age of everyone (Excluding Brenn) We achieve that by multiplying Adults total number by the mean age (all excluding Brenn) $\text{sum_age_before} = \text{mean_before} * \text{len}(\text{adult}[\text{adult}['\text{Name}'] \neq \text{"Brenn McNeely"}])$ Then we need to calculate the total age of all adults this is achieved by getting the multiplication of total number of adults by the mean age (here it includes Brenn) $\text{sum_age_after} = \text{mean_after} * \text{len}(\text{adult})$ the accurate version of Brenn's age can be found by subtracting the total age after and before his participation $\text{print}(\text{brenns_age})$ # His age is printed here

✓ 0.0s
38.0

Figure 5 Output of Brenns age

```
assert brenns_age == 38.0 #verifying that his age is 38.0 and this ensures that
```

2.4 Task 2 Summary

1. In Part 1, 'adult' was grouped by a single column, and the function was created to count the total number of entries from each group. This section focuses on grouping and counting of only that.
2. After being constructed in a way that was similar to that of Part 1, Part 2 concentrated on two column groupings. The difference was that two columns 'Occupation' and 'Education' were used this time to help highlight some patterns or even some potential vulnerabilities.
3. The real differencing attack to determine Brennan's age takes place in Part 3, which is the last part. The main way used to accomplish this was the 'get_brenns_age' function, which determined the age mean

both before and after Brenn's participation. Which at the end then helped retrieve Brenn's accurate age and was then verified with his actual age.

3. Alter the Zip column in the adult dataset to achieve k-Anonymity for $k=2$

K-anonymity is known as a form of privacy technique. Its main purpose is to formalize important or sensitive data in a database to protect individual privacy through anonymization. In this task, we are going to present three scenarios.

3.1 Scenario 1

In this scenario, we have written the below code to generalise the Zip column once by keeping the first 2 digits of the number and setting the rest to zeros (as shown below in the code).

3.1.1 Code

```
# This function is to generalize the 'Zip' column
def generalize_zip(zip, i=2):
    # Here will only keep the first i digits in the 'Zip' number and set the rest to
    zeros
    # For example if the number is 62340 and i = 2, then the number will be 62000.
    return int(zip // 10 ** (len(str(zip)) - i)) * 10 ** (len(str(zip)) - i)

# Apply the generalization function to the 'Zip' column then name it 'G_Zip'
adult['G_Zip'] = adult['Zip'].apply(lambda x: generalize_zip(x, i=2)) # G_Zip =
'Generalized Zip'

# Display the modified dataset with the selected columns (Zip and G_Zip)
k_anonymous_dataset = adult[['Zip', 'G_Zip']]

# Check if the modified dataset achieves k-anonymity with k=2
def k_anonymized(adult, column_name, k):
    counts = adult.groupby(column_name).size()
    return all(count >= k for count in counts)

k_value = 2 # Selected k value
is_k_anonymized_result = k_anonymized(k_anonymous_dataset, 'G_Zip', k_value)

# Printing the modified Dataset
print("Modified Dataset:")
print(k_anonymous_dataset)

# Check if the modified dataset is k-anonymized
print(f"\nIs the modified dataset k-anonymized with k={k_value}?
{is_k_anonymized_result}")
```

This is the output:

```

Modified Dataset:
      Zip      G_Zip
0      64152  64000.0
1      61523  61000.0
2      95668  95000.0
3      25503  25000.0
4      75387  75000.0
...      ...      ...
32556  41328  41000.0
32557  94735  94000.0
32558  49628  49000.0
32559   8213   8200.0
32560  86153  86000.0

[32561 rows x 2 columns]

Is the modified dataset k-anonymized with k=2? False

```

Figure 6 Output of modified dataset & K-anonymity verification

3.1.2 Explanation

Based on the output that we got, we can see that we have a new column called '**G_Zip**'. In this column, we have applied the generalization function to generalize the Zip column to achieve k-Anonymity $k=2$ as mentioned above. We tried to generalize or, in another word, to formalize the zip column, hoping that we could achieve $k=2$ where two zip numbers from two different rows share the same number in the **G_Zip** column.

But based on the results we obtained, we can see that the modified dataset hasn't achieved $k=2$ even after generalizing the whole column. This means that every row shares a unique value ($k=1$), therefore, achieving $k=2$ is not possible in this scenario.

3.2 Scenario 2

In this second scenario, we have created a new program with a function that will keep generalizing until $k=2$ is achieved.

3.2.1 Code

```

# This function is to generalize the 'Zip' column
def generalize_zip(zip_code, i=2):
    # Here will only keep the first i digits in the 'Zip' number and set the rest to
    zeros
    # For example if the number is 62340 and i = 2, then the number will be 62000.
    return int(zip_code // 10 ** (len(str(zip_code)) - i)) * 10 **
(len(str(zip_code)) - i)

# Apply the generalization function to the 'Zip' column then name it 'G_Zip'
adult['G_Zip'] = adult['Zip'].apply(lambda x: generalize_zip(x, i=2)) # G_Zip =
'Generalized Zip'

# Selected k value
k_value = 2

# Checking the dataset if it's k-anonymized according to the unique G_Zip column
values.
def k_anonymized(adult, G_Zip, k):

```

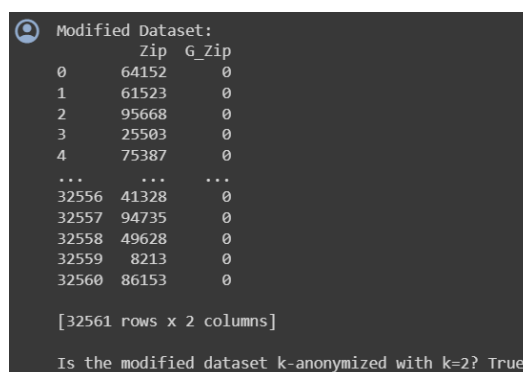
```
# Grouping the G_Zip column and count the occurrences of each unique values.
counts = adult.groupby(G_Zip).size()
# Check if all counts are greater than or equal to k then return
return all(count >= k for count in counts)

# Continue generalizing until k-anonymity is achieved
while not k_anonymized(adult, 'G_Zip', k_value):
    adult['G_Zip'] = adult['G_Zip'].apply(lambda x: generalize_zip(x, i=2))

# Print the modified dataset
print("Modified Dataset:")
print(adult[['Zip', 'G_Zip']])

# Check if the modified dataset is k-anonymized
is_k_anonymized_result = k_anonymized(adult, 'G_Zip', k_value)
print(f"\nIs the modified dataset k-anonymized with k={k_value}? {is_k_anonymized_result}")
```

This is the output:



```
Modified Dataset:
   Zip  G_Zip
0  64152     0
1  61523     0
2  95668     0
3  25503     0
4  75387     0
...    ...
32556 41328     0
32557 94735     0
32558 49628     0
32559 8213     0
32560 86153     0

[32561 rows x 2 columns]

Is the modified dataset k-anonymized with k=2? True
```

Figure 7 Output of the K-Anonymity Iteration: 'Zip' Column Generalized

3.2.2 Explanation

From the output that we got, we can see that all the generalised zips (**G_Zip**) are zeros because the code tried to achieve $k=2$ by generalising each row, which led to overkilling the data because it contains outliers that are different from the rest; therefore, these zip values are difficult to categorise, even after generalisation. In only that way, the dataset can achieve $k=2$.

3.3 Scenario 3

In this scenario, we will be removing all the outliers that are affecting us to achieve $k=2$ without overkilling the data.

3.3.1 Code

```
# Limit or clip the 'Zip' column values to be within a specific range (in
this scenario, from 10000 to 99999)
lower_bound = 10000
upper_bound = 99999
adult['Zip'] = np.clip(adult['Zip'], lower_bound, upper_bound)

# This function is to generalize the 'Zip' column
def generalize_zip(zip_code, i=2):
    return int(zip_code // 10 ** (len(str(zip_code)) - i)) * 10 **
(len(str(zip_code)) - i)

# Apply the generalization function to the 'Zip' column then name it 'G_Zip'
adult['G_Zip'] = adult['Zip'].apply(lambda x: generalize_zip(x, i=2))

# Selected k value
k_value = 2

# Checking the dataset if it's k-anonymized according to the unique G_Zip
column values.
def k_anonymized(adult, G_Zip, k):
    counts = adult.groupby(G_Zip).size()
    return all(count >= k for count in counts)

# Continue generalizing until k-anonymity is achieved
while not k_anonymized(adult, 'G_Zip', k_value):
    adult['G_Zip'] = adult['G_Zip'].apply(lambda x: generalize_zip(x, i=2))

# Print the modified dataset
print("Modified Dataset:")
print(adult[['Zip', 'G_Zip']])

# Check if the modified dataset is k-anonymized
is_k_anonymized_result = k_anonymized(adult, 'G_Zip', k_value)
print(f"\nIs the modified dataset k-anonymized with k={k_value}?
{is_k_anonymized_result}")
```

This is the output:

```

Modified Dataset:
      Zip  G_Zip
0      64152 64000
1      61523 61000
2      95668 95000
3      25503 25000
4      75387 75000
...      ...   ...
32556  41328 41000
32557  94735 94000
32558  49628 49000
32559  10000 10000
32560  86153 86000

[32561 rows x 2 columns]

Is the modified dataset k-anonymized with k=2? True

```

Figure 8 Output of K-Anonymity Enforcement & Data Generalized

3.3.2 Explanation

We have removed all the outliers from the dataset by using '**np.clip**', which is a **Numpy** function to clip (limit) the values in a specified range, in our case, from 10000 to 99999, as shown in the code. Then apply the 'clip' function so that if the number is below 10000, it will be set to 10000, and if it is above 99999, it will be set to 99999. Now the values are restricted to the defined range.

Finally, from the output, we can see that we have achieved k-anonymity where k=2 without overkilling the zip values.

4. Determine whether Karrie Trusslove's age is 39 using a differentially private counting query. Add Laplace noise to ensure differential privacy with $\epsilon = 1.0$.

4.1 Task 4 - Part 1

The imported dataframe contains a very large dataset about different people, although we can filter out the data by using unique identifiers and additional information that can narrow down the output, we can look for all the data about a specified person. In our case, it is Karrie Trusslove, with the additional information related to her, which is her age (39 years old).

4.1.1 code

```

# this will extract all the rows with the name of Karrie Trusslove within the
database
karries_row= adult[adult['Name']== 'Karrie Trusslove']
#this will point out karrie Trusslove with the age of 39
karries_row[karries_row['Age']== 39].shape[0]

```

>>The output: 1

4.1.2 explanation

This code snippet will provide us with the actual data to compare it with the modified data, to simulate Laplace noise modification to the data and satisfy confidential privacy.

4.2 Task 4 - Part 2

Since we already know that there is 1 Karrie Trusslove in the database with the age of 39, we can proceed with the second stage which is the addition of Laplace noise. Laplace noise will help in securing the actual data by anonymizing the data output and ensure confidential privacy.

4.2.1 code

```
#defining the sensitivity of the Laplace
sensitivity = 1
#defining the privacy (epsilon) for the laplace
epsilon = 1

#This is a recall to show how the karries_row was created
karries_row= adult[adult['Name']== 'Karrie Trusslove']

#defining the database result
diff_priv_output = karries_row[karries_row['Age']== 39].shape[0] +
np.random.laplace(loc=0,scale = sensitivity/epsilon)
    # "karries_row[karries_row['Age']== 39].shape[0]" will provide the information of
the Query (Karrie, with the age of 39) which is = 1 as you can see above...
    # Loc = 0 --> laplace distribution is set to 0
    # scale will represent the sensitivity and the epsilon of the laplace
    # The main purpose of "np.random.laplace(loc=0,scale = sensitivity/epsilon)" is
to create noise to the data...

#output of the data query (with noise)
diff_priv_output
#the output will contain noise, as the epsilon is making noise for the data to be
unclear.
```

>>Output is: **1.3635312917590396** (with noise)

#Note: the number will vary and will never stay consistent.

4.2.2 Explanation

Based on the output of the program, we have successfully managed to add noise to the output utilising the Laplace mechanism. This will create randomness to the output resulting in having a different value from the original value which satisfies the differential privacy.

In our case, we specified the value of **sensitivity = 1** and the value of **epsilon (€) = 1**. Note that the sensitivity will **always be equal to 1 in a counting queries**.

$$\text{Noise} = \text{Laplace} \left(\frac{\text{Sensitivity}}{\epsilon} \right)$$

This means that the laplace scale has a balanced setting for differential privacy. The balance between privacy protection and data utility.

$$F(x) = f(x) + \text{Noise}$$

The actual data will be combined with the noise produced from the laplace mechanism and provide us with the final output. In our case the final output was **1.3635312917590396**. Keep in mind that the value will always be different after every query (*even if the exact query was used again*).

5. Conduct a linkage attack using adult pii and adult deid dataframes to recover as many names as possible from adult deid. Parameterize your solution by the columns used in the attack.)

Linking attack is a form of security threat, it happens when the attacker can link two datasets together to re-identify an individual after being de-identified. This form of attack mainly consists of **merging** two datasets which are in our case: **adult_pii**, and **adult_deid**. Personally identifiable information (**PII**) consists of the name, Date of Birth, social security number, Zip code, and age. This information is considered as **sensitive information** and can be useful for the attacker. On the other hand, we have the de-identified data (**DEID**), this kind of dataset is used to share information for research and statistics purposes as it doesn't contain any sensitive information or information that gives off any lead to an individual as it protects the individual's privacy.

In this program, linkage attack will be demonstrated as the program will create two datasets **PII** and **DEID** and will merge them together to correlate information to re-identify individuals.

5.1 Code

```
#Creating two dataframes, the data within the dataframes are defined by us, and in
this case, we have [Name, Date of birth, Social Security Number, ZIP code, and Age]
for the Personally identifiable information.
pii_adult_db = adult[['Name', 'DOB', 'SSN', 'Zip', 'Age']]
#And we will drop the two key identifiers to help in de-identifying the data records
for the De-identified dataframe.
deid_adult_db = adult.drop(columns=['Name', 'SSN'])

#Creating the linkage attack funtion that utilizes the two dataframes (pii_adult_db
and deid_adult_db) and the list of columns that will be used for linking in the
attack.
def linkage_Atk(pii_adult_db, deid_adult_db, cols):

    # Perform the linkage attack using an inner join on the specified columns
    # the inner join will will combine the data from the pii_adult_db and
deid_adult_db from the specified column/s (cols)
    linkage_output = pd.merge(pii_adult_db, deid_adult_db, how="inner", on=cols)

    # after linking the two tables, we can return the linked data records.
    return linkage_output

# Example usage:
# This will make sure that the Zip is in both dataframes
link_columns = ['Zip'] # join the two dataframes through the Zip and Age column only
```

```
result_df = linkage_Atk(pii_adult_db, deid_adult_db, link_columns )# use the defined
attack function
print("Linked using Zip Code: ",len(result_df))

# this will make sure that Zip and Date of birth are both in the dataframes
link_columns = ['Zip', 'DOB'] # join the two dataframes through the Zip and the date
of birth columns only
result_df = linkage_Atk(pii_adult_db, deid_adult_db, link_columns)# use the defined
attack function
print("Linked using Zip Code, and Date of Birth: ",len(result_df))

# this will make sure that Zip and Date of birth are both in the dataframes
link_columns = ['Zip', 'Age'] # join the two dataframes through the Zip and Age
columns only
result_df = linkage_Atk(pii_adult_db, deid_adult_db, link_columns) # use the defined
attack function
print("Linked using Zip Code, and Age: ",len(result_df))

# this will make sure that Zip, Age, and Date of birth are in the dataframes
link_columns = ['Zip', 'Age', 'DOB'] # join the two dataframes through the Zip and
Age and Date of birth columns
result_df = linkage_Atk(pii_adult_db, deid_adult_db, link_columns) # use the defined
attack function
print("Linked using Zip Code, Date of Birth, and Age: ",len(result_df))
# As we increase the Joined columns, in most cases, it will narrow down the volume of
data...
```

>>Output is:

Linked using Zip Code: 43191
Linked using Zip Code, and Date of Birth: 32563
Linked using Zip Code, and Age: 32755
Linked using Zip Code, Date of Birth, and Age: 32561

5.2 Explanation

After executing the forged python program, we have obtained the final output which states that after linking the datasets with different sets of linking columns, we have successfully obtained the number of records between the **PII** and the **DEID** databsets that matches up in the **zip code by its own** (presented on the first line of the output), **Linkage using Zip Code and Date of Birth** (second line of the output), **Linkage using Zip Code and Age** (third line of the output), and finally, **Linkage using Zip Code, Date of Birth, and Age** (last line of the output).

Linkage attempts provided us with different results, although, **as we increase the columns** to be linked between the two datasets, the **matching data decreases**.lets break down why this is happening.

The first linking attempt (only the **ZIP code**) was only single column linkage, note that a large number of matching records can be found here as there is only one condition to be met, yet many individuals can share the same zip code.

As we increase the columns, there are **more conditions to be met**. This will reduce the chances of finding records that satisfy the conditions (two specified columns within the linkage attempt), Therefore, giving us a smaller result.

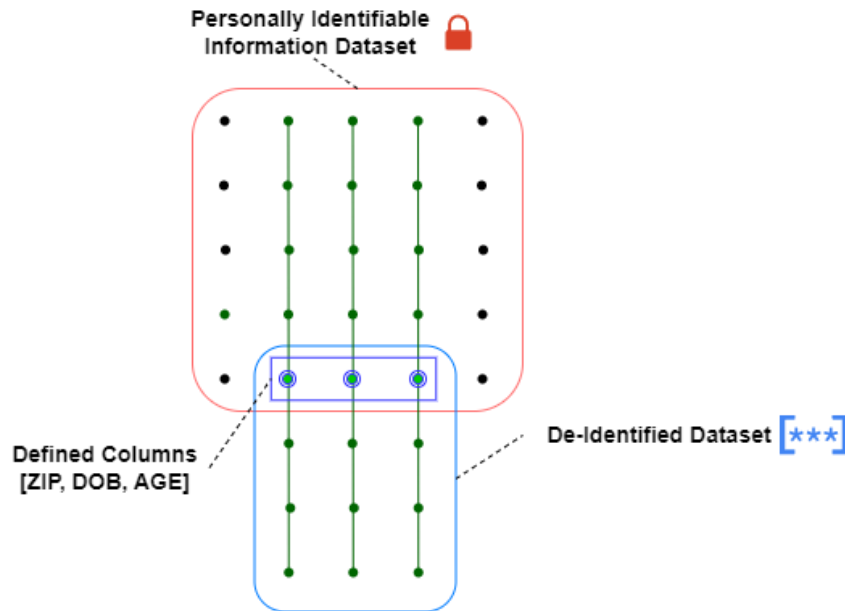


Figure 9 Linkage Attack Representation

6. Determine the number of individuals uniquely identified by their Zip code and age in the dataset.

In this task, we will be writing Python code to calculate the number of individuals who are uniquely identified by their zip code and age.

6.1 Code

```
# This is function to calculate the number of uniquely identified individuals for Zip
and Age
def uniq_count(adult): # uniq_count = unique individuals count
    # Grouping the DataFrame by 'Zip' and 'Age' columns and count frequency repeated
    uniq_comb = adult.groupby(['Zip', 'Age']).size().reset_index(name='Count') #
    uniq_comb = unique_combinations

    # Filtering the combinations in which the count equals 1, indicating persons who
    are uniquely identifiable
    uniquely_identified_count = uniq_comb[uniq_comb['Count'] == 1].shape[0]

    # Return the count of uniquely identified individuals
    return uniquely_identified_count

# Display the number of uniquely identified individuals
# Calling the function with the DataFrame 'adult' and store the result in
count_unique_individuals
count_unique_individuals = uniq_count(adult)

# Print the result
```

```
print(f"Number of individuals uniquely identified by Zip code and age:
{count_unique_individuals}")
```

This is the output:

```
Number of individuals uniquely identified by Zip code and age: 32367
```

Figure 10 Output of individuals found by Zip Code & Age

6.2 Explanation

The function **'uniq_count'** will take the database as an input to calculate the unique number that is identified by individuals based on the **zip** and **age** columns. Inside the function, we used the **'groupby'** method to group zip and age columns together so we could count the unique frequency combination using **'size()'** and save it to **'uniq_comb'**.

Then we will filter this combination so the count can reach 1, which means that there is only one unique occurrence of this combination. Then we will be checking the shape of the modified dataset to get the number of rows.

Finally, the function will be stored in the variable **'count_unique_individuals'**, then it will be printed out as is shown above.

7. Calculate a differentially private sum of ages for various values of b, clipped to each b, with $\epsilon = 0.1$.

7.1 Code

```
# Required imports
from numpy.random import laplace

def dp_sum_of_ages(adult, b, epsilon):
    # Clip the age data
    clipped_data = np.clip(adult['Age'], 0, b)

    # Calculate the true sum
    true_sum = clipped_data.sum()

    # Calculate the scale for Laplace noise
    sensitivity = b
    scale = sensitivity / epsilon

    # Add Laplace noise
    noisy_sum = true_sum + laplace(0, scale)

    return noisy_sum

b_values = [30, 40, 50, 60, 70, 80, 90, 100]
for b in b_values:
```

```
dp_sum = dp_sum_of_ages(adult, b, 0.1)
print(f"Differentially private sum for b={b}: {dp_sum}")
```

```
Differentially private sum for b=30: 912761.3165251508
Differentially private sum for b=40: 1096388.6471775067
Differentially private sum for b=50: 1195918.9340472273
Differentially private sum for b=60: 1237618.5675611878
Differentially private sum for b=70: 1251891.3878004993
Differentially private sum for b=80: 1255631.6090031029
Differentially private sum for b=90: 1257186.3522960853
Differentially private sum for b=100: 1256017.2250096698
```

7.2 Explanation

We were tasked to Calculate a differentially private sum of ages for various values of b , clipped to each b , with $\epsilon = 0.1$ so The code implements the concept of calculating a differentially private sum of ages for various clipping values b , with a privacy parameter $\epsilon = 0.1$. It defines a function `dp_sum_of_ages` that first clips ages in the dataset to a maximum of b and then computes their sum. To ensure differential privacy, Laplacian noise scaled by b and ϵ is added to this sum. The process is repeated for multiple values of b , demonstrating how the private sum varies with different clipping levels. This method balances data utility with privacy protection, ensuring individual ages do not affect the aggregated sum. So As " b " increases, the level of privacy protection likely increases, but at the same time, the amount of noise introduced into the result also increases. This trade-off between privacy and accuracy is a fundamental aspect of differential privacy.

The code goes as follows:

1.Import Laplace Function: Import the `laplace` function from `numpy.random`.

2.Define `dp_sum_of_ages` Function:

- Parameters: The function takes three arguments:
 - `adult`: The dataset.
 - `b`: The clipping parameter.
 - `epsilon`: The privacy budget.
- Process:
 - Clip Ages: Clip the 'Age' column in the dataset to a maximum of b .
 - Calculate True Sum: Compute the sum of the clipped ages.
 - Determine Laplace Noise Scale: Set the scale of the Laplace noise as the sensitivity (b) divided by `epsilon`.
 - Add Laplace Noise: Add Laplacian noise to the true sum for differential privacy.
- Return: The function returns the noisy sum, which is the differentially private sum of ages.

3.Iterate Over `b`_values:

- Define a list of `b` values: `[30, 40, 50, 60, 70, 80, 90, 100]`.
- For each `b` in this list, call `dp_sum_of_ages` with the dataset, the current `b` value, and a fixed `epsilon` of 0.1.
- Output: Print the differentially private sum for each `b`, demonstrating how the sum changes with different clipping values.

8. Develop an algorithm to automatically select a clipping parameter for the age column that satisfies differential privacy.

8.1 Task 8 part 1

Clipping in database security is used to enforce the upper and the lower bound of the query values. We cannot set a specific clipping bounds, or it will give away data leads and expose individual's information and not satisfy the differential privacy.

8.1.1 Code

```
# this small code will provide us with the actual value of the summation
operation with the defined age.
def age_sum_query(df, b):
    return df['Age'].clip(lower=0, upper=b).sum()

age_sum_query(adult, 20)
# we can use this output as a reference to our final output on the next code
part
```

>> Output is: **648223**

8.1.2 Explanation

This code will provide us with the exact summation value of individuals from the age of 0 and up to 20 years old. Note that the lower bound is set to 0, this is because the human age cannot be lower than 0. The upper bound was set to 20, as we are requesting for the summation of individuals between the age of 0 and exactly 20, the output will be consistent as the data is 100% accurate and no noise was introduced here.

We can use the output as a reference for the next stage of this task (task 8).

8.2 Task 8 part 2

The requirement is to create a python program that dynamically selects an upper bound based on the age value range we are trying to perform summation operation. The objective here is to achieve differential privacy with the algorithm.

8.2.1 Code

```
# This function will run the laplace mechanism for the differential privacy.
def laplace_operation(query_result, sensitivity, epsilon):
    # This line will generate Laplace noise
    # The input of the defined function will be set from the function call
    when setting up the upperbound.
    noise = np.random.laplace(loc=0, scale=sensitivity/epsilon)

    # Add noise to the query result
    # Query result will already be calculated as an int
    private_result = query_result + noise
```

```

    # this will return the query with the noise
    return private_result

# This function will calculate the sum of the age column in the dataframe
with a fixed lower bound = 0, this is because there is no age below 0.
# The upperbound will be defined using the "select_upper_bound" function.
def calculate_age_sum(df, upper_bound):
    # Calculate and return the sum of the 'Age' column with clipping
    return df['Age'].clip(lower=0, upper=upper_bound).sum()

# This defined function will automatically select the upperbound for the age,
by utilizing laplace mechanism and the age sum value.
def select_upper_bound(query_function, dataframe, privacy_budget):
    # Define a range of potential upper bounds (starting from 1, and ending
    with 999, with a pace of 10 counts per hop)
    bounds = range(1, 1000, 10)
    # The best value represents the best value as the upper bound for the
    query, the initial value is 0 (it will change as the program starts
    processing values and calculations)
    best = 0
    # The threshold will help to control the loop and stop when the best
    bound and the new bound difference is lower than the threshold.
    threshold = 10
    # This is the epsilon value that will be used repeatedly (note that the
    value might change after each iteration as the bound will change through each
    loop).
    epsilon_i = privacy_budget / len(bounds)

    # this loop will go through the bounds and detect which bound is
    considered the best for this query.
    for bound in bounds:
        # Apply Laplace mechanism to query result with varying upper bounds
        result = laplace_operation(query_function(dataframe, bound), bound,
        epsilon_i)

        # If the new bound is close to the best bound, stop
        if result - best <= threshold:
            return bound
        # Otherwise, update the "best" answer to be the current one
        else:
            best = result
    # This will return the last bound set within the index, which is the best
    bound set by the loop...
    return bounds[-1]

# Example query with dynamic upper bound setting
# "select_upper_bound" function is called here with the defined parameters,
and assigned the value to the "final_upper_bound" variable/

```

```
final_upper_bound = select_upper_bound(calculate_age_sum, adult, 1)
print("The upper bound was set automatically to:", final_upper_bound)

# data with the extracted bound.
print("Final data of adults with the age of 20 with the extracted upper bound
is:", age_sum_query(adult, final_upper_bound))
```

>> Output is:

The upper bound was set automatically to: 81

Final data of adults with the age of 20 with the extracted upper bound is: 1255772

8.2.2 Explanation

Based on the output, we have successfully achieved an upper bound value that was not set by us. Note that the upper bound value will always be above the requested age range, this will include 100% of the individuals who meets the requirement.

By default, the upper bound will not reach above 125 years old, as the maximum human life span is 125 years, and no data will be found beyond that age.

Let us break down how this program achieved the most suitable upper bound while satisfying differential privacy. Going back to the program, “best = 0” is the initial point for the upper bound, the algorithm increases the bound with a pace of 10, until the output stops changing or have a very slight changes when compared to the previous bound. Once the best bound is achieved, it then sets it as the final bound, which is the final output. Note that this algorithm might not be applicable to all sets of data, for instance, financial data.

Table 1 - Possible Upper Bounds with the corresponding Data

Bound	Data
11	358171
21	678374
31	935798
41	1108069
51	1201865
61	1241700
71	1253315
81	1255772
91	1256257
101	1256257
111	1256257
121	1256257

This data was gathered from the adult data frame, we went through all the possible upper bounds to see which upper bound is the best. As highlighted in red in the table above, we can see from the readings that the value of individuals is slightly changing at **bound 71, 81, and 91**. And if we used the program to find us the best upper bound, the most common best upper bound will be set on 71, 81, or 91. Although it is possible to get 101, 111, and 121 as an upper bound because they also satisfy the bound selection criteria.

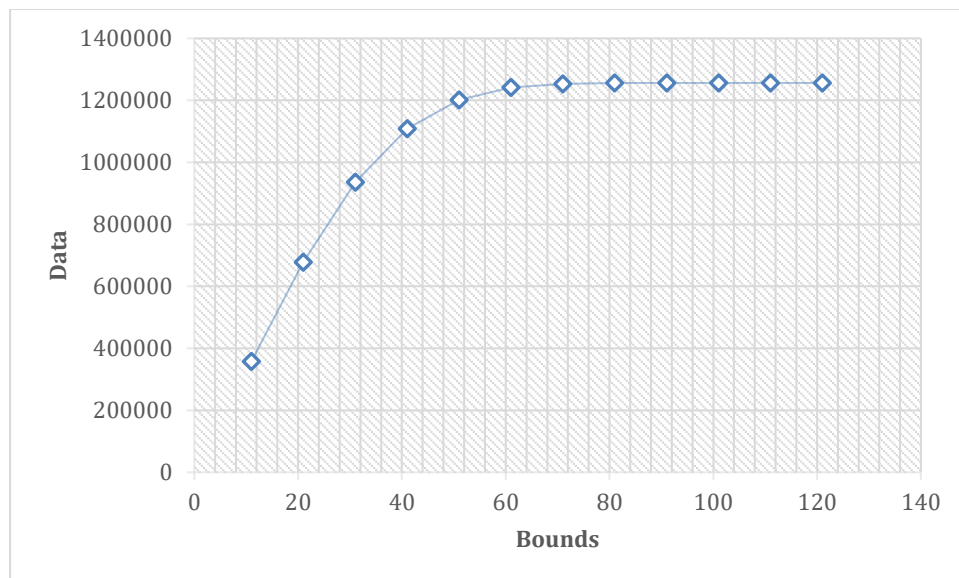


Figure 11 Clipping Upper Bound for Age

This diagram shows us the line where the data is being consistent, the algorithm will identify the first point where the value is slightly changing or heading for a straight line.

Using this information, we can set the clipping bounds automatically, in our case, we only needed the upper bound for the clipping. The lower bound was already set by default as 0 (human age cannot be lower than 0).

9. Create a differentially private contingency table for the Education and Sex columns of the adult dataset.

9.1 Code

```
# Function to create a differentially private contingency table
def dp_contingency_table(adult, col1, col2, epsilon=0.1):
    # Create the contingency table
    contingency_table = pd.crosstab(adult[col1], adult[col2])

    # Calculate the scale of the Laplace noise
    sensitivity = 1 # Sensitivity for counting queries
    scale = sensitivity / epsilon

    # Add Laplacian noise to the counts in the contingency table
    noisy_table = contingency_table + np.random.laplace(0, scale,
contingency_table.shape)

    return noisy_table

epsilon = 0.1
noisy_table = dp_contingency_table(adult, 'Education', 'Sex', epsilon)
print(noisy_table)
```

Sex	Female	Male
Education		

10th	297.941783	634.794645
11th	467.138142	750.405575
12th	152.341079	280.600131
1st-4th	82.255145	115.647376
5th-6th	116.852077	236.279214
7th-8th	156.052616	506.870213
9th	143.462437	365.295141
Assoc-acdm	436.927457	635.040056
Assoc-voc	508.833429	888.324949
Bachelors	1605.355487	3734.686269
Doctorate	88.907363	352.964898
HS-grad	3381.668822	7102.790357
Masters	507.325601	1183.306376
Preschool	22.743116	19.164031
Prof-school	86.499586	494.998343
Some-college	2807.211598	4476.179487

9.2 Explanation

We were tasked to Create a differentially private contingency table for the Education and Sex columns of the adult dataset and to make a table using data from two columns in our case is Education and Sex , and then add laplace noise in it.

The code format goes as follows:

1. Define `dp_contingency_table` Function:

- Parameters: The function accepts four arguments:

- adult: The dataset.
- col1 and col2: The names of two columns in the dataset for which the contingency table is to be created.
- epsilon: The privacy budget.

- Process:

- Create Contingency Table: Generate a contingency table (cross-tabulation) between col1 and col2.
- Determine Laplace Noise Scale: Calculate the scale of the Laplace noise based on a sensitivity of 1 (typical for counting queries) and the provided epsilon.
- Add Laplacian Noise: Apply Laplacian noise to each count in the contingency table to ensure differential privacy.

- Return: The function returns the noisy contingency table, which has differential privacy guarantees.

2. Apply Function to Dataset:

- Set epsilon to 0.1.
- Call `dp_contingency_table` with the adult dataset, specifying 'Education' and 'Sex' as the columns for the contingency table, and the defined epsilon.
- Output: Print the resulting differentially private contingency table.

References

1. Near, J. P., & Abuah, C. (2023, October 26). Programming Differential Privacy. <https://programming-dp.com/book.pdf>
2. Top 30 Python Libraries To Know in 2024. (2023, November 8). Great Learning Blog: Free Resources What Matters to Shape Your Career! <https://www.mygreatlearning.com/blog/open-source-python-libraries/>