Introduction

This is a book about differential privacy, for programmers. It is intended to give you an introduction to the challenges of data privacy, introduce you to the techniques that have been developed for addressing those challenges, and help you understand how to implement some of those techniques.

The book contains numerous examples *as programs*, including implementations of many concepts. Each chapter is generated from a self-contained Jupyter Notebook. You can click on the "download" button at the top-right of the chapter to download the notebook for that chapter, and you'll be able to execute the examples yourself. Many of the examples are generated by code that is hidden (for readability) in the chapters you'll see here. You can show this code by clicking the "Click to show" labels adjacent to these cells.

This book assumes a working knowledge of Python, as well as basic knowledge of the pandas and NumPy libraries. You will also benefit from some background in discrete mathematics and probability - a basic undergraduate course in these topics should be more than sufficient.

This book is open source, and the latest version will always be available online here. The source code is available on GitHub. If you would like to fix a typo, suggest an improvement, or report a bug, please open an issue on GitHub.

The techniques described in this book have developed out of the study of *data privacy*. For our purposes, we will define data privacy this way:

> **Definition**
>
> *Data privacy* techniques have the goal of allowing analysts to learn about *trends* in sensitive data, without revealing information specific to *individuals*.

This is a broad definition, and many different techniques fall under it. But it's important to note what this definition *excludes*: techniques for ensuring *security*, like encryption. Encrypted data doesn't reveal *anything* - so it fails to meet the first requirement of our definition. The distinction between security and privacy is an important one: privacy techniques involve an *intentional* release of information, and attempt to control *what can be learned* from that release; security techniques usually *prevent* the release of information, and control *who can access* data. This book covers privacy techniques, and we will only discuss security when it has important implications for privacy.

This book is primarily focused on differential privacy. The first couple of chapters outline some of the reasons why: differential privacy (and its variants) is the only formal approach we know about that seems to provide robust privacy protection. Commonly-used approaches that have been used for decades (like de-identification and aggregation) have more recently been shown to break down under sophisticated privacy attacks, and even more modern techniques (like k-Anonymity) are susceptible to certain attacks. For this reason, differential privacy is fast becoming the gold standard in privacy protection, and thus it is the primary focus of this book.

De-identification

De-identification is the process of removing *identifying information* from a dataset. The term *de-identification* is sometimes used synonymously with the terms *anonymization* and *pseudonymization*.

Learning Objectives

After reading this chapter, you be able to:

- Define the following concepts:
  - De-identification
  - Re-identification
  - Identifying information / personally identifying information
  - Linking attacks
  - Aggregation and aggregate statistics
  - Differencing attacks
- Perform a linking attack
- Perform a differencing attack
- Explain the limitations of de-identification techniques
- Explain the limitations of aggregate statistics

Identifying information has no formal definition. It is usually understood to be information which would be used to identify us uniquely in the course of daily life - name, address, phone number, e-mail address, etc. As we will see later, it's *impossible* to formalize the concept of identifying information, because *all* information is identifying. The term *personally identifiable information (PII)* is often used synonymously with identifying information.

How do we de-identify information? Easy - we just remove the columns that contain identifying information!

```
adult_data = adult.copy().drop(columns=['Name', 'SSN'])
adult_pii = adult[['Name', 'SSN', 'DOB', 'Zip']]
adult_data.head(1)
```

	DOB	Zip	Age	Workclass	fnlwgt	Education	Education-Num	Marital Status	Occupatic
0	9/7/1967	64152	39	State-gov	77516	Bachelors	13	Never-married	Adn cleric

We'll save some of the identifying information for later, when we'll use it as *auxiliary data* to perform a *re-identification* attack.

Linking Attacks

Imagine we want to determine the income of a friend from our de-identified data. Names have been removed, but we happen to know some auxiliary information about our friend. Our friend's name is Karrie Trusslove, and we know Karrie's date of birth and zip code.

To perform a simple *linking attack*, we look at the overlapping columns between the dataset we're trying to attack, and the auxiliary data we know. In this case, both datasets have dates of birth and zip codes. We look for rows in the dataset we're attacking with dates of birth and zip codes that match Karrie's date of birth and zip code. If there is only one such row, we've found Karrie's row in the dataset we're attacking. In databases, this is called a *join* of two tables, and we can do it in Pandas using `merge`.

```
karries_row = adult_pii[adult_pii['Name'] == 'Karrie Trusslove']
pd.merge(karries_row, adult_data, left_on=['DOB', 'Zip'], right_on=['DOB', 'Zip'])
```

	Name	SSN	DOB	Zip	Age	Workclass	fnlwgt	Education	Education-Num	M S
0	Karrie Trusslove	732-14-6110	9/7/1967	64152	39	State-gov	77516	Bachelors	13	N m

Indeed, there is only one row that matches. We have used auxiliary data to re-identify an individual in a de-identified dataset, and we're able to infer that Karrie's income is less than $50k.

How Hard is it to Re-Identify Karrie?

This scenario is made up, but linking attacks are surprisingly easy to perform in practice. How easy? It turns out that in many cases, just one data point is sufficient to pinpoint a row!

```
pd.merge(karries_row, adult_data, left_on=['Zip'], right_on=['Zip'])
```

	Name	SSN	DOB_x	Zip	DOB_y	Age	Workclass	fnlwgt	Education	Edu
0	Karrie Trusslove	732-14-6110	9/7/1967	64152	9/7/1967	39	State-gov	77516	Bachelors	

So ZIP code is sufficient **by itself** to allow us to re-identify Karrie. What about date of birth?

```
pd.merge(karries_row, adult_data, left_on=['DOB'], right_on=['DOB'])
```

	Name	SSN	DOB	Zip_x	Zip_y	Age	Workclass	fnlwgt	Education	Educa
0	Karrie Trusslove	732-14-6110	9/7/1967	64152	64152	39	State-gov	77516	Bachelors	
1	Karrie Trusslove	732-14-6110	9/7/1967	64152	67306	64	Private	171373	11th	
2	Karrie Trusslove	732-14-6110	9/7/1967	64152	62254	46	Self-emp-not-inc	119944	Masters	

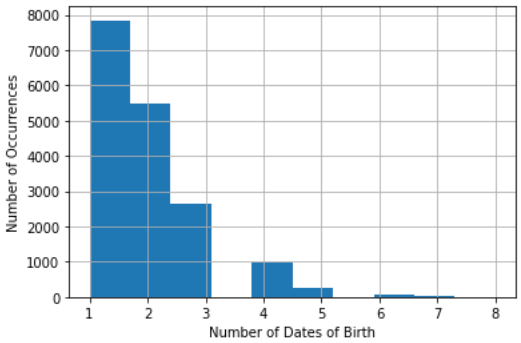
This time, there are three rows returned - and we don't know which one is the real Karrie. But we've still learned a lot!

- We know that there's a 2/3 chance that Karrie's income is less than $50k
- We can look at the differences between the rows to determine what additional auxiliary information would *help* us to distinguish them (e.g. sex, occupation, marital status)

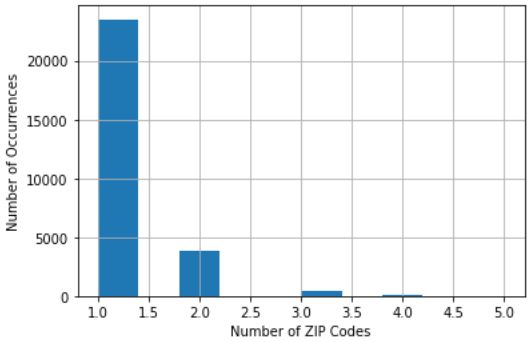
Is Karrie Special?

How hard is it to re-identify others in the dataset? Is Karrie especially easy or especially difficult to re-identify? A good way to guage the effectiveness of this type of attack is to look at how "selective" certain pieces of data are - how good they are at narrowing down the set of potential rows which may belong to the target individual. For example, is it common for birthdates to occur more than once?

We'd like to get an idea of how many dates of birth are likely to be useful in performing an attack, which we can do by looking at how common "unique" dates of birth are in the dataset. The histogram below shows that *the vast majority* of dates of birth occur 1, 2, or 3 times in the dataset, and *no date of birth* occurs more than 8 times. This means that date of birth is fairly *selective* - it's effective in narrowing down the possible records for an individual.

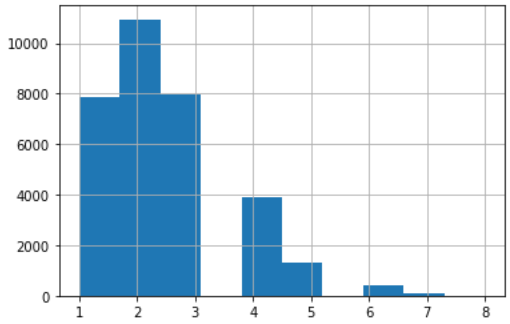


We can do the same thing with ZIP codes, and the results are even worse - ZIP code happens to be *very* selective in this dataset. Nearly all the ZIP codes occur only once.

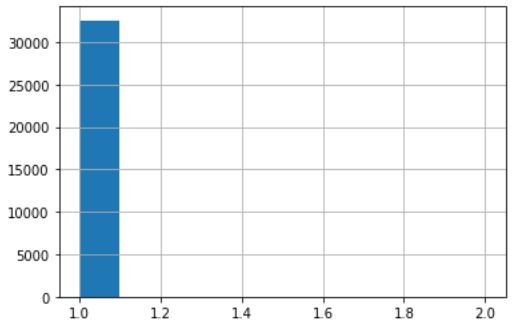


How Many People can we Re-Identify?

In this dataset, how many people can we re-identify uniquely? We can use our auxiliary information to find out! First, let's see what happens with just dates of birth. We want to know how many *possible identities* are returned for each data record in the dataset. The following histogram shows the number of records with each number of possible identities. The results show that we can uniquely identify almost 7,000 of the data records (out of about 32,000), and an additional 10,000 data records are narrowed down to two possible identities.



So it's not possible to re-identify a majority of individuals using *just* date of birth. What if we collect more information, to narrow things down further? If we use both date of birth and ZIP, we're able to do much better. In fact, we're able to uniquely re-identify basically the whole dataset.



When we use both pieces of information, we can re-identify **essentially everyone**. This is a surprising result, since we generally assume that many people share the same birthday, and many people live in the same ZIP code. It turns out that the *combination* of these factors is **extremely** selective. According to Latanya Sweeney's work, 87% of people in the US can be uniquely re-identified by the combination of date of birth, gender, and ZIP code.

Let's just check that we've actually re-identified *everyone*, by printing out the number of possible data records for each identity:

```
Antonin Chittem     2
Barnabe Haime       2
Smith Scholfield    1
Andy Keer           1
Pincas Darling      1
Name: Name, dtype: int64
```

Looks like we missed two people! In other words, in this dataset, only **two people** share a combination of ZIP code and date of birth.

Aggregation

Another way to prevent the release of private information is to release only *aggregate* date.

```
adult['Age'].mean()
```

```
38.58164675532078
```

Problem of Small Groups

In many cases, aggregate statistics are broken down into smaller groups. For example, we might want to know the average age of people with a particular education level.

```
adult[['Education-Num', 'Age']].groupby('Education-Num').mean().head(3)
```

	Age
Education-Num	
1	42.764706
2	46.142857
3	42.885886

Aggregation is supposed to improve privacy because it's hard to identify the contribution of a particular individual to the aggregate statistic. But what if we aggregate over a group with just *one person* in it? In that case, the aggregate statistic reveals one person's age *exactly*, and provides no privacy protection at all! In our dataset, most individuals have a unique ZIP code - so if we compute the average age by ZIP code, then most of the "averages" actually reveal an individual's exact age.

```
adult[['Zip', 'Age']].groupby('Zip').mean().head()
```

	Age
Zip	
4	55.0
12	24.0
16	59.0
17	42.0
18	24.0

The US Census Bureau, for example, releases aggregate statistics at the *block level*. Some census blocks have large populations, but some have a population of zero! The situation above, where small groups prevent aggregation from hiding information about individuals, turns out to be quite common.

How big a group is "big enough" for aggregate statistics to help? It's hard to say - it depends on the data and on the attack - so it's challenging to build confidence that aggregate statistics are really privacy-preserving. However, even very large groups do not make aggregation completely robust against attacks, as we will see next.

Differencing Attacks

The problems with aggregation get even worse when you release multiple aggregate statistics over the same data. For example, consider the following two summation queries over large groups in our dataset (the first over the whole dataset, and the second over all records except one):

```
adult['Age'].sum()
```

```
1256257
```

```
adult[adult['Name'] != 'Karrie Trusslove']['Age'].sum()
```

```
1256218
```

If we know both answers, we can simply take the difference and determine Karrie's age completely! This kind of attack can proceed even if the aggregate statistics are over *very large groups*.

```
adult['Age'].sum() - adult[adult['Name'] != 'Karrie Trusslove']['Age'].sum()
```

```
39
```

This is a recurring theme.

- Releasing *data* that is useful makes ensuring *privacy* very difficult
- Distinguishing between *malicious* and *non-malicious* queries is not possible

Summary

- A *linking attack* involves combining *auxiliary data* with *de-identified data* to *re-identify* individuals.
- In the simplest case, a linking attack can be performed via a *join* of two tables containing these datasets.
- Simple linking attacks are surprisingly effective:
  - Just a single data point is sufficient to narrow things down to a few records
  - The narrowed-down set of records helps suggest additional auxiliary data which might be helpful
  - Two data points are often good enough to re-identify a huge fraction of the population in a particular dataset
  - Three data points (gender, ZIP code, date of birth) uniquely identify 87% of people in the US

k-Anonymity

k-Anonymity is a *formal privacy definition*. The definition of k-Anonymity is designed to formalize our intuition that a piece of auxiliary information should not narrow down the set of possible records for an individual "too much." Stated another way, k-Anonymity is designed to ensure that each individual can "blend into the crowd."

> **Learning Objectives**
>
> After reading this chapter, you will understand:
>
> - The definition of k-Anonymity
> - How to check for k-Anonymity
> - How to generalize data to enforce k-Anonymity
> - The limitations of k-Anonymity

Informally, we say that a dataset is "k-Anonyized" for a particular k if each individual in the dataset is a member of a group of size at least k, such that each member of the group shares the same *quasi-identifiers* (a selected subset of all the dataset's columns) with all other members of the group. Thus, the individuals in each group "blend into" their group - it's possible to narrow down an individual to membership in a particular group, but not to determine which group member is the target.

Definition

Formally, we say that a dataset D satisfies k-Anonymity for a value of k if:

- For each row $r_1 \in D$, there exist at least $k-1$ other rows $r_2 \ldots r_k \in D$ such that $\Pi_{qi(D)} r_1 = \Pi_{qi(D)} r_2, \ldots, \Pi_{qi(D)} r_1 = \Pi_{qi(D)} r_k$

where $qi(D)$ is the quasi-identifiers of D, and $\Pi_{qi(D)} r$ represents the columns of r containing quasi-identifiers (i.e. the projection of the quasi-identifiers).

Checking for k-Anonymity

We'll start with a small dataset, so that we can immediately see by looking at the data whether it satisfies k-Anonymity or not. This dataset contains age plus two test scores; it clearly doesn't satisfy k-Anonymity for $k > 1$. Any dataset trivially satisfies k-Anonymity for $k = 1$, since each row can form its own group of size 1.

	age	preTestScore	postTestScore
0	42	4	25
1	52	24	94
2	36	31	57
3	24	2	62
4	73	3	70

To implement a function to check whether a dataframe satisfies k-Anonymity, we loop over the rows; for each row, we query the dataframe to see how many rows match its values for the quasi-identifiers. If the number of rows in any group is less than k, the dataframe does not satisfy k-Anonymity for that value of k, and we return False. Note that in this simple definition, we consider *all* columns to contain quasi-identifiers; to limit our check to a subset of all columns, we would need to replace the `df.columns` expression with something else.

```
def isKAnonymized(df, k):
    for index, row in df.iterrows():
        query = ' & '.join([f'{col} == "{row[col]}"' for col in df.columns])
        rows = df.query(query)
        if rows.shape[0] < k:
            return False
    return True
```

As expected, our example dataframe does *not* satisfy k-Anonymity for $k = 2$, but it does satisfy the property for $k = 1$.

```
isKAnonymized(df, 1)
```

```
True
```

```
isKAnonymized(df, 2)
```

```
False
```

Generalizing Data to Satisfy k-Anonymity

The process of modifying a dataset so that it satisfies k-Anonymity for a desired k is generally accomplished by *generalizing* the data - modifying values to be less specific, and therefore more likely to match the values of other individuals in the dataset. For example, an age which is accurate to a year may be generalized by rounding to the nearest 10 years, or a ZIP code might have its rightmost digits replaced by zeros. For numeric values, this is easy to implement. We'll use the `apply` method of dataframes, and pass in a dictionary named `depths` which specifies how many digits to replace by zeros for each column. This gives us the flexibility to experiment with different levels of generalization for different columns.

```
def generalize(df, depths):
    return df.apply(lambda x: x.apply(lambda y: int(int(y/(10**depths[x.name]))*
(10**depths[x.name]))))
```

Now, we can generalize our example dataframe. First, we'll try generalizing each column by one "level" - i.e. rounding to the nearest 10.

```
depths = {
    'age': 1,
    'preTestScore': 1,
    'postTestScore': 1
}
df2 = generalize(df, depths)
df2
```

	age	preTestScore	postTestScore
0	40	0	20
1	50	20	90
2	30	30	50
3	20	0	60
4	70	0	70

Notice that even after generalization, our example data *still* does not satisfy k-Anonymity for $k = 2$.

```
isKAnonymized(df2, 2)
```

```
False
```

We can try generalizing more - but then we'll end up removing *all* of the data!

```
depths = {
    'age': 2,
    'preTestScore': 2,
    'postTestScore': 2
}
generalize(df, depths)
```

	age	preTestScore	postTestScore
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0

This example illustrates one of the key challenges of achieving k-Anonymity:

> **Challenge**
>
> Achieving k-Anonymity for meaningful values of k often requires removing quite a lot of information from the data

Does More Data Improve Generalization?

Our example dataset is too small for k-Anonymity to work well. Because there are only 5 individuals in the dataset, building groups of 2 or more individuals who share the same properties is difficult. The solution to this problem is more data: in a dataset with more individuals, less generalization will typically be needed to satisfy k-Anonymity for a desired k.

Let's try the same census data we examined for de-identification. This dataset contains more than 32,000 rows, so it should be easier to achieve k-Anonymity.

We'll consider the ZIP code, age, and educational achievement of each individual to be the quasi-identifiers. We'll project just those columns, and try to achieve k-Anonymity for $k = 2$. The data is already k-Anonymous for $k = 1$.

Notice that we take just the first 100 rows from the dataset for this check - try running `isKAnonymized` on a larger subset of the data, and you'll find that it takes a very long time (for example, running the $k = 1$ check on 5000 rows takes about 20 seconds on my laptop). For $k = 2$, our algorithm finds a failing row quickly and finishes fast.

```
df = adult_data[['Age', 'Education-Num']]
df.columns = ['age', 'edu']
isKAnonymized(df.head(100), 1)
```

```
True
```

```
isKAnonymized(df.head(100), 2)
```

```
False
```

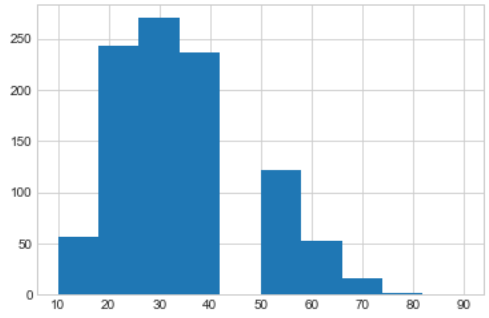
Now, we'll try to generalize to achieve k-Anonymity for $k = 2$. We'll start with generalizing both age and educational attainment to the nearest 10.

```
# outliers are a real problem!
depths = {
    'age': 1,
    'edu': 1
}
df2 = generalize(df.head(1000), depths)
isKAnonymized(df2, 2)
```

```
False
```

The generalized result still does not satisfy k-Anonymity for $k = 2$! In fact, we can perform this generalization on all 32,000 rows and still fail to satisfy k-Anonymity for $k = 2$ - so adding more data does not necessarily help as much as we expected.

The reason is that the dataset contains *outliers* - individuals who are very different from the rest of the population. These individuals do not fit easily into any group, even after generalization. Even considering *only* ages, we can see that adding more data is not likely to help, since very low and very high ages are poorly represented in the dataset.



Achieving the optimal generalization for k-Anonymity is very challenging in cases like this. Generalizing each row *more* would be overkill for the well-represented individuals with ages in the 20-40 range, and would hurt utility. However, more generalization is clearly needed for individuals at the upper and lower ends of the age range. This is the kind of challenge that occurs regularly in practice, and is difficult to solve automatically. In fact, optimal generalization for k-Anonymity has been shown to be NP-hard.

Challenge

Outliers make achieving k-Anonymity very challenging, even for large datasets. Optimal generalization for k-Anonymity is NP-hard.

Removing Outliers

One solution to this problem is simply to clip the age of each individual in the dataset to lie within a specific range, eliminating outliers entirely. This can also hurt utility, since it replaces real ages with fake ones, but it can be better than generalizing each row more. We can use Numpy's `clip` method to perform this clipping. We clip ages to be 60 or below, and leave educational levels alone (by clipping them to a very large value).

```
# clipping away outliers
depths = {
    'age': 1,
    'edu': 1
}
dfp = df.clip(upper=np.array([60, 10000000000000]), axis='columns')
df2 = generalize(dfp.head(500), depths)
isKAnonymized(df2, 7)
```

```
True
```

Now, the generalized dataset satisfies k-Anonymity for $k = 7$! In other words, our level of generalization was appropriate, but outliers prevented us from achieving k-Anonymity before, even for $k = 2$.

Summary

- k-Anonymity is a property of data, which ensures that each individual "blends in" with a group of at least k individuals.
- k-Anonymity is computationally expensive even to check: the naive algorithm is $O(n^2)$, and faster algorithms take considerable space.
- k-Anonymity can be achieved by modifying a dataset by *generalizing* it, so that particular values become more common and groups are easier to form.
- Optimal generalization is extremely difficult, and outliers can make it even more challenging. Solving this problem automatically is NP-hard.

Differential Privacy

> **Learning Objectives**
>
> After reading this chapter, you will be able to:
>
> - Define differential privacy
> - Explain the importance of the privacy parameter ϵ
> - Use the Laplace mechanism to enforce differential privacy for counting queries

Like k-Anonymity, *differential privacy* is a formal notion of privacy (i.e. it's possible to prove that a data release has the property). Unlike k-Anonymity, however, differential privacy is a property of *algorithms*, and not a property of *data*. That is, we can prove that an *algorithm* satisfies differential privacy; to show that a *dataset* satisfies differential privacy, we must show that the algorithm which produced it satisfies differential privacy.

> **Definition**
>
> A function which satisfies differential privacy is often called a *mechanism*. We say that a *mechanism* F satisfies differential privacy if for all *neighboring datasets* x and x', and all possible outputs S,
>
> \begin{equation} \frac{\mathsf{Pr}[F(x) = S]}{\mathsf{Pr}[F(x') = S]} \leq e^\epsilon \end{equation}

Two datasets are considered neighbors if they differ in the data of a single individual. Note that F is typically a *randomized* function, so that the probability distribution describing its outputs is not just a point distribution.

The important implication of this definition is that F's output will be pretty much the same, *with or without* the data of any specific individual. In other words, the randomness built into F should be "enough" so that an observed output from F will not reveal which of x or x' was the input. Imagine that my data is present in x but not in x'. If an adversary can't determine which of x or x' was the input to F, then the adversary can't tell whether or not my data was *present* in the input - let alone the contents of that data.

The ϵ parameter in the definition is called the *privacy parameter* or the *privacy budget*. ϵ provides a knob to tune the "amount of privacy" the definition provides. Small values of ϵ require F to provide *very* similar outputs when given similar inputs, and therefore provide higher levels of privacy; large values of ϵ allow less similarity in the outputs, and therefore provide less privacy.

How should we set ϵ to prevent bad outcomes in practice? Nobody knows. The general consensus is that ϵ should be around 1 or smaller, and values of ϵ above 10 probably don't do much to protect privacy - but this rule of thumb could turn out to be very conservative. We will have more to say on this subject later on.

The Laplace Mechanism

Differential privacy is typically used to answer specific queries. Let's consider a query on the census data, *without* differential privacy.

"How many individuals in the dataset are 40 years old or older?"

```
adult[adult['Age'] >= 40].shape[0]
```

```
14237
```

The easiest way to achieve differential privacy for this query is to add random noise to its answer. The key challenge is to add enough noise to satisfy the definition of differential privacy, but not so much that the answer becomes too noisy to be useful. To make this process easier, some basic *mechanisms* have been developed in the field of differential privacy, which describe exactly what kind of - and how much - noise to use. One of these is called the *Laplace mechanism*.

Definition

According to the Laplace mechanism, for a function $f(x)$ which returns a number, the following definition of $F(x)$ satisfies ϵ-differential privacy:

\begin{equation} F(x) = f(x) + \textsf{Lap}(\frac{s}{\epsilon}) \end{equation}

where s is the *sensitivity* of f, and $\text{Lap}(S)$ denotes sampling from the Laplace distribution with center 0 and scale S.

The *sensitivity* of a function f is the amount f's output changes when its input changes by 1. Sensitivity is a complex topic, and an integral part of designing differentially private algorithms; we will have much more to say about it later. For now, we will just point out that *counting queries* always have a sensitivity of 1: if a query counts the number of rows in the dataset with a particular property, and then we modify exactly one row of the dataset, then the query's output can change by at most 1.

Thus we can achieve differential privacy for our example query by using the Laplace mechanism with sensitivity 1 and an ϵ of our choosing. For now, let's pick $\epsilon = 0.1$. We can sample from the Laplace distribution using Numpy's `random.laplace`.

```
sensitivity = 1
epsilon = 0.1

adult[adult['Age'] >= 40].shape[0] + np.random.laplace(loc=0,
scale=sensitivity/epsilon)
```

```
14264.4863909206
```

You can see the effect of the noise by running this code multiple times. Each time, the output changes, but most of the time, the answer is close enough to the true answer (14,235) to be useful.

How Much Noise is Enough?

How do we know that the Laplace mechanism adds enough noise to prevent the re-identification of individuals in the dataset? For one thing, we can try to break it! Let's write down a malicious counting query, which is specifically designed to determine whether Karrie Trusslove has an income greater than $50k.

```
karries_row = adult[adult['Name'] == 'Karrie Trusslove']
karries_row[karries_row['Target'] == '<=50K'].shape[0]
```

```
1
```

This result definitely violates Karrie's privacy, since it reveals the value of the income column for Karrie's row. Since we know how to ensure differential privacy for counting queries with the Laplace mechanism, we can do so for this query:

```
sensitivity = 1
epsilon = 0.1

karries_row = adult[adult['Name'] == 'Karrie Trusslove']
karries_row[karries_row['Target'] == '<=50K'].shape[0] + \
  np.random.laplace(loc=0, scale=sensitivity/epsilon)
```

```
-16.11794116092802
```

Is the true answer 0 or 1? There's too much noise to be able to reliably tell. This is how differential privacy is *intended* to work - the approach does not *reject* queries which are determined to be malicious; instead, it adds enough noise that the results of a malicious query will be useless to the adversary.

Properties of Differential Privacy

Learning Objectives

After reading this chapter, you will be able to:

- Explain the concepts of sequential composition, parallel composition, and post processing
- Calculate the cumulative privacy cost of multiple applications of a differential privacy mechanism
- Determine when the use of parallel composition is allowed

This chapter describes three important properties of differentially private mechanisms that arise from the definition of differential privacy. These properties will help us to design useful algorithms that satisfy differential privacy, and ensure that those algorithms provide accurate answers. The three properties are:

- Sequential composition
- Parallel composition
- Post processing

Sequential Composition

The first major property of differential privacy is *sequential composition*, which bounds the total privacy cost of releasing multiple results of differentially private mechanisms on the same input data. Formally, the sequential composition theorem for differential privacy says that:

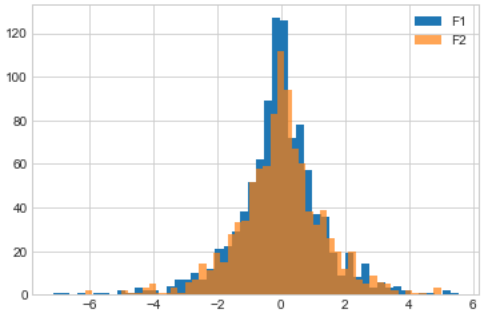
- If $F_1(x)$ satisfies ϵ_1-differential privacy
- And $F_2(x)$ satisfies ϵ_2-differential privacy
- Then the mechanism $G(x) = (F_1(x), F_2(x))$ which releases both results satisfies $\epsilon_1 + \epsilon_2$-differential privacy

Sequential composition is a vital property of differential privacy because it enables the design of algorithms that consult the data more than once. Sequential composition is also important when multiple separate analyses are performed on a single dataset, since it allows individuals to bound the *total* privacy cost they incur by participating in all of these analyses. The bound on privacy cost given by sequential composition is an *upper* bound - the actual privacy cost of two particular differentially private releases may be smaller than this, but never larger.

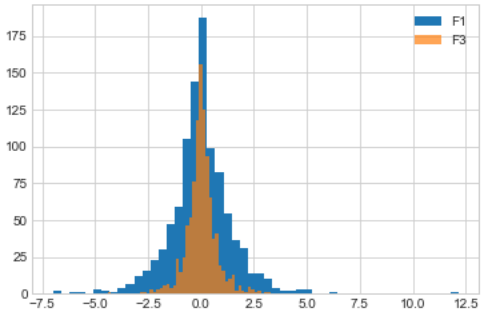
The principle that the ϵs "add up" makes sense if we examine the distribution of outputs from a mechanism which averages two differentially private results together. Let's look at some examples.

```
epsilon1 = 1
epsilon2 = 1
epsilon_total = 2

# satisfies 1-differential privacy
def F1():
    return np.random.laplace(loc=0, scale=1/epsilon1)

# satisfies 1-differential privacy
def F2():
    return np.random.laplace(loc=0, scale=1/epsilon2)

# satisfies 2-differential privacy
def F3():
    return np.random.laplace(loc=0, scale=1/epsilon_total)

# satisfies 2-differential privacy, by sequential composition
def F_combined():
    return (F1() + F2()) / 2
```

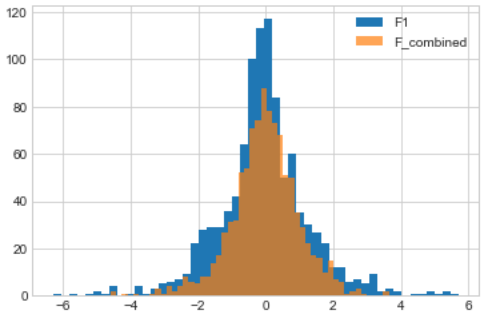
If we graph `F1` and `F2`, we see that the distributions of their outputs look pretty similar.



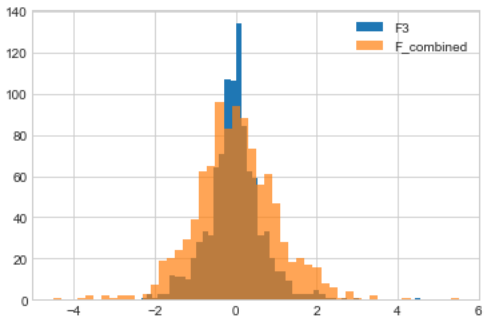
If we graph `F1` and `F3`, we see that the distribution of outputs from `F3` looks "pointier" than that of `F1`, because its higher ϵ implies less privacy, and therefore a smaller liklihood of getting results far from the true answer.



If we graph `F1` and `F_combined`, we see that the distribution of outputs from `F_combined` is pointier. This means its answers are more accurate than those of `F1`, so it makes sense that its ϵ must be higher (i.e. it yields less privacy than `F1`).



What about `F3` and `F_combined`? Recall that the ϵ values for these two mechanisms are the same - both have an ϵ of 2. Their output distributions should look the same.



In fact, `F3` looks "pointier"! Why does this happen? Remember that sequential composition yields an *upper* bound on the total ϵ of several releases, the actual cumulative impact on privacy might be lower. That's the case here - the actual privacy loss in this case appears to be somewhat lower than the upper bound ϵ determined by sequential composition. Sequential composition is an extremely useful way to control total privacy cost, and we will see it used in many different ways, but keep in mind that it is not necessarily an exact bound.

Parallel Composition

The second important property of differential privacy is called *parallel composition*. Parallel composition can be seen as an alternative to sequential composition - a second way to calculate a bound on the total privacy cost of multiple data releases. Parallel composition is based on the idea of splitting your dataset into disjoint chunks and running a differentially private mechanism on each chunk separately. Since the chunks are disjoint, each individual's data appears in *exactly* one chunk - so even if there are k chunks in total (and therefore k runs of the mechanism), the mechanism runs exactly once on the data of each *individual*. Formally,

- If $F(x)$ satisfies ϵ-differential privacy
- And we split a dataset X into k disjoint chunks such that $x_1 \cup \ldots \cup x_k = X$
- Then the mechanism which releases all of the results $F(x_1), \ldots, F(x_k)$ satisfies ϵ-differential privacy

Note that this is a much better bound than sequential composition would give. Since we run F k times, sequential composition would say that this procedure satisfies $k\epsilon$-differential privacy. Parallel composition allows us to say that the total privacy cost is just ϵ.

The formal definition matches up with our intuition - if each participant in the dataset contributes one row to X, then this row will appear in *exactly* one of the chunks $x_1, \ldots, x_k$. That means F will only "see" this participant's data *one time*, meaning a privacy cost of ϵ is appropriate for that individual. Since this property holds for all individuals, the privacy cost is ϵ for everyone.

Histograms

In our context, a *histogram* is an analysis of a dataset which splits the dataset into "bins" based on the value of one of the data attributes, and counts the number of rows in each bin. For example, a histogram might count the number of people in the dataset who achieved a particular educational level.

	Education
HS-grad	10501
Some-college	7291
Bachelors	5355
Masters	1723
Assoc-voc	1382

Histograms are particularly interesting for differential privacy because they automatically satisfy parallel composition. Each "bin" in a histogram is defined by a possible value for a data attribute (for example, `'Education' == 'HS-grad'`). It's impossible for a single row to have *two* values for an attribute simultaneously, so defining the bins this way

guarantees that they will be disjoint. Thus we have satisfied the requirements for parallel composition, and we can use a differentially private mechanism to release *all* of the bin counts with a total privacy cost of just ϵ.

```
epsilon = 1

# This analysis has a total privacy cost of epsilon = 1, even though we release many
results!
f = lambda x: x + np.random.laplace(loc=0, scale=1/epsilon)
s = adult['Education'].value_counts().apply(f)
s.to_frame().head(5)
```

	Education
HS-grad	10501.561849
Some-college	7291.583605
Bachelors	5354.602625
Masters	1722.822477
Assoc-voc	1382.470028

Contingency Tables

A *contingency table* or *cross tabulation* (often shortened to *crosstab*) is like a multi-dimensional histogram. It counts the frequency of rows in the dataset with particular values for more than one attribute at a time. Contingency tables are frequently used to show the relationship between two variables when analyzing data. For example, we might want to see counts based on both education level and gender:

```
pd.crosstab(adult['Education'], adult['Sex']).head(5)
```

Sex	Female	Male
Education		
10th	295	638
11th	432	743
12th	144	289
1st-4th	46	122
5th-6th	84	249

Like the histogram we saw earlier, each individual in the dataset participates in exactly *one* count appearing in this table. It's impossible for any single row to have multiple values simultaneously, for any set of data attributes considered in building the contingency table. As a result, it's safe to use parallel composition here, too.

```
ct = pd.crosstab(adult['Education'], adult['Sex'])
f = lambda x: x + np.random.laplace(loc=0, scale=1/epsilon)
ct.applymap(f).head(5)
```

Sex	Female	Male
Education		
10th	293.817738	639.004865
11th	430.937092	741.357329
12th	144.516689	290.177689
1st-4th	46.450268	119.861570
5th-6th	85.488219	248.197719

It's also possible to generate contingency tables of more than 2 variables. Consider what happens each time we add a variable, though: each of the counts tends to get smaller. Intuitively, as we split the dataset into more chunks, each chunk has fewer rows in it, so all of the counts get smaller.

These shrinking counts can have a significant affect on the accuracy of the differentially private results we calculate from them. If we think of things in terms of signal and noise, a large count represents a strong *signal* - it's unlikely to be disrupted too much by relatively weak noise (like the noise we add above), and therefore the results are likely to be

useful even after the noise is added. However, a small count represents a weak *signal* - potentially *as weak* as the noise itself - and after we add the noise, we won't be able to infer anything useful from the results.

So while it may seem that parallel composition gives us something "for free" (more results for the same privacy cost), that's not really the case. Parallel composition simply moves the tradeoff between accuracy and privacy along a different axis - as we split the dataset into more chunks and release more results, each result contains a weaker signal, and so it's less accurate.

Post-processing

The third property of differential privacy we will discuss here is called *post-processing*. The idea is simple: it's impossible to reverse the privacy protection provided by differential privacy by post-processing the data in some way. Formally:

- If $F(X)$ satisfies ϵ-differential privacy
- Then for any (deterministic or randomized) function g, $g(F(X))$ satisfies ϵ-differential privacy

The post-processing property means that it's always safe to perform arbitrary computations on the output of a differentially private mechanism - there's no danger of reversing the privacy protection the mechanism has provided. In particular, it's fine to perform post-processing that might reduce the noise or improve the signal in the mechanism's output (e.g. replacing negative results with zeros, for queries that shouldn't return negative results). In fact, many sophisticated differentially private algorithms make use of post-processing to reduce noise and improve the accuracy of their results.

The other implication of the post-processing property is that differential privacy provides resistance against privacy attacks based on auxiliary information. For example, the function g might contain auxiliary information about elements of the dataset, and attempt to perform a linking attack using this information. The post-processing property says that such an attack is limited in its effectiveness by the privacy parameter ϵ, regardless of the auxiliary information contained in g.

Sensitivity

> **Learning Objectives**
>
> After reading this chapter, you will be able to:
>
> - Define sensitivity
> - Find the sensitivity of counting queries
> - Find the sensitivity of summation queries
> - Decompose average queries into counting and summation queries
> - Use clipping to bound the sensitivity of summation queries

As we mentioned when we dicussed the Laplace mechanism, the amount of noise necessary to ensure differential privacy for a given query depends on the *sensitivity* of the query. Roughly speaking, the sensitivity of a function reflects the amount the function's output will change when its input changes. Recall that the Laplace mechanism defines a mechanism $F(x)$ as follows:

\begin{align} F(x) = f(x) + \textsf{Lap}(\frac{s}{\epsilon}) \end{align}

where $f(x)$ is a deterministic function (the query), ϵ is the privacy parameter, and s is the sensitivity of f.

For a function $f : \mathcal{D} \rightarrow \mathbb{R}$ mapping datasets ($\mathcal{D}$) to real numbers, the *global sensitivity* of f is defined as follows:

\begin{align} GS(f) = \max_{x, x': d(x,x') <= 1} |f(x) - f(x')| \end{align}

Here, $d(x, x')$ represents the *distance* between two datasets x and x', and we say that two datasets are *neighbors* if their distance is 1 or less. How this distance is defined has a huge effect on the definition of privacy we obtain, and we'll discuss the distance metric on datasets in detail later on.

The definition of global sensitivity says that for *any two* neighboring datasets x and x', the difference between $f(x)$ and $f(x')$ is at most $GS(f)$. This measure of sensitivity is called "global" because it is independent of the actual dataset being queried (it holds for *any* choice of neighboring x and x'). Another measure of sensitivity, called *local*

sensitivity, fixes one of the datasets to be the one being queried; we will consider this measure in a later section. For now, when we say "sensitivity," we mean global sensitivity.

Distance

The distance metric $d(x, x')$ described earlier can be defined in many different ways. Intuitively, the distance between two datasets should be equal to 1 (i.e. the datasets are neighbors) if they differ in the data of exactly one individual. This idea is easy to formalize in some contexts (e.g. in the US Census, each individual submits a single response containing their data) but extremely challenging in others (e.g. location trajectories, social networks, and time-series data).

A common formal definition for datasets containing rows is to consider the number of rows which differ between the two. When each individual's data is contained in a single row, this definition often makes sense. Formally, this definition of distance is encoded as a *symmetric difference* between the two datasets:

\begin{align} d(x, x') = | x - x' \cup x' - x | \end{align}

This particular definition has several interesting and important implications:

- If x' is constructed from x by *adding one row*, then $d(x, x') = 1$
- If x' is constructed from x by *removing one row*, then $d(x, x') = 1$
- If x' is constructed from x by *modifying one row*, then $d(x, x') = 2$

In other words, adding or removing a row results in a neighboring dataset; *modifying* a row results in a dataset at distance *2*.

This particular definition of distance results in what is typically called *unbounded differential privacy*. Many other definitions are possible, including one called *bounded differential privacy* in which modifying a single row in a dataset *does* result in a neighboring dataset.

For now, we will stick to the formal definition defined above in terms of symmetric difference. We will discuss alternative definitions in a later section.

Calculating Sensitivity

How do we determine the sensitivity of a particular function of interest? For some simple functions on real numbers, the answer is obvious.

- The global sensitivity of $f(x) = x$ is 1, since changing x by 1 changes $f(x)$ by 1
- The global sensitivity of $f(x) = x + x$ is 2, since changing x by 1 changes $f(x)$ by 2
- The global sensitivity of $f(x) = 5 * x$ is 5, since changing x by 1 changes $f(x)$ by 5
- The global sensitivity of $f(x) = x * x$ is unbounded, since the change in $f(x)$ depends on the value of x

For functions that map datasets to real numbers, we can perform a similar analysis. We will consider the functions which represent common aggregate database queries: counts, sums, and averages.

Counting Queries

Counting queries (`COUNT` in SQL) count the number of rows in the dataset which satisfy a specific property. As a rule of thumb, **counting queries always have a sensitivity of 1**. This is because adding a row to the dataset can increase the output of the query by at most 1: either the new row has the desired property, and the count increases by 1, or it does not, and the count stays the same (the count may correspondingly decrease when a row is removed).

Example: "How many people are in the dataset?" (sensitivity: 1 - counting rows with a property)

```
adult.shape[0]
```

```
32561
```

Example: "How many people have an educational status above 10?" (sensitivity: 1 - counting rows with a property)

```
adult[adult['Education-Num'] > 10].shape[0]
```

```
10516
```

Example: "How many people have an educational status equal to or below 10?" (sensitivity: 1 - counting rows with a property)

```
adult[adult['Education-Num'] <= 10].shape[0]
```

```
22045
```

Example: "How many people are named Joe Near?" (sensitivity: 1 - counting rows with a property)

```
adult[adult['Name'] == 'Joe Near'].shape[0]
```

```
0
```

Summation Queries

Summation queries (`SUM` in SQL) sum up the *attribute values* of dataset rows.

Example: "What is the sum of the ages of people with an educational status above 10?"

```
adult[adult['Education-Num'] > 10]['Age'].sum()
```

```
422876
```

Sensitivity for these queries is not as simple as it is for counting queries. Adding a new row to the dataset will increase the result of our example query by the *age of the new person*. That means the sensitivity of the query depends on the *contents* of the row we add.

We'd like to come up with a concrete number to represent the sensitivity of the query. Unfortunately, no number really exists. We could claim, for example, that the sensitivity is 125 - but it may turn out that the row we add to the database corresponds to a person who is over 125 years old, which would violate our claim. For any number we come up with, it's possible for the added row to violate our claim.

You might (rightly) be skeptical of this point. Say we claim the sensitivity is 1000 - it's very unlikely that we'll find a person who is 1000 years old to violate this claim. In this specific domain - ages - there's a very reasonable upper bound on how old someone can be. The oldest person ever lived to be 122 years old, so an upper bound of 125 seems reasonable.

But this is not a *proof* that nobody will ever live to be 126. And in other domains (e.g. income), it can be much harder to come up with a reasonable upper bound.

As a rule of thumb, summation queries have **unbounded sensitivity** when no lower and upper bounds exist on the value of the attribute being summed. When lower and upper bounds do exist, the sensitivity of a summation query is equal to the **difference between them**. In the next section, we will see a technique called *clipping* for enforcing bounds when none exist, so that summation queries with unbounded sensitivity can be converted into queries with bounded sensitivity.

Average Queries

Average queries (`AVG` in SQL) calculate the mean of attribute values in a particular column.

Example: "What is the average age of people with an educational status above 10?"

```
adult[adult['Education-Num'] > 10]['Age'].mean()
```

```
40.21262837580829
```

The easiest way to answer an average query with differential privacy is by re-phrasing it as two queries: a summation query divided by a counting query. For the above example:

```
adult[adult['Education-Num'] > 10]['Age'].sum() / adult[adult['Education-Num'] > 10]
['Age'].shape[0]
```

```
40.21262837580829
```

The sensitivities of both queries can be calculated as described above. Noisy answers for each can be calculated (e.g. using the Laplace mechanism) and the noisy answers can be divided to obtain a differentially private mean. The total privacy cost of both queries can be calculated by sequential composition.

Clipping

Queries with unbounded sensitivity cannot be directly answered with differential privacy using the Laplace mechanism. Fortunately, we can often transform such queries into equivalent queries with *bounded* sensitivity, via a process called *clipping*.

The basic idea beind clipping is to *enforce* upper and lower bounds on attribute values. For example, ages above 125 can be "clipped" to exactly 125. After clipping has been performed, we are *guaranteed* that all ages will be 125 or below. As a result, the senstivity of a summation query on clipped data is equal to the difference between the upper and lower bounds used in clipping: $upper - lower$. For example, the following query has a sensitivity of 125:

```
adult['Age'].clip(lower=0, upper=125).sum()
```

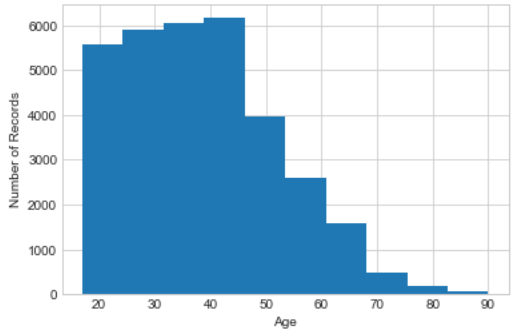
```
1256257
```

The primary challenge in performing clipping is to determine the upper and lower bounds. For ages, this is simple - nobody can have an age less than 0, and probably nobody will be older than 125. In other domains, as mentioned earlier, it's much more difficult.

Furthermore, there is a tradeoff between the amount of information lost in clipping and the amount of noise needed to ensure differential privacy. When the upper and lower clipping bounds are closer together, then the sensitivity is lower, and less noise is needed to ensure differential privacy. However, aggressive clipping often removes a lot of information from the data; this information loss tends to cause a *loss* of accuracy which outweighs the improvement in noise resulting from smaller sensitivity.

As a rule of thumb, **try to set the clipping bounds to include 100% of the dataset**, or get as close as possible. This is harder in some domains (e.g. graph queries, which we will study later) than others.

It's tempting to determine the clipping bounds by looking at the data. For example, we can look at the histogram of ages in our dataset to determine an appropriate upper bound:

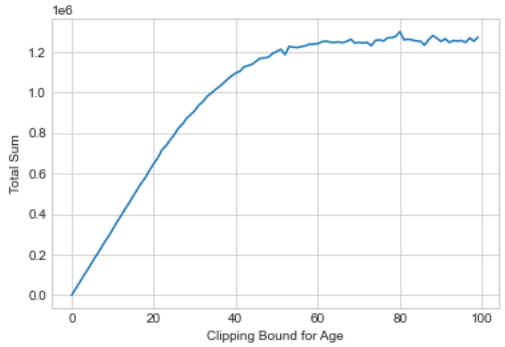


It's clear from this histogram that nobody in this particular dataset is over 90, so an upper bound of 90 would suffice.

However, it's important to note that **this approach does not satisfy differential privacy**. If we pick our clipping bounds by looking at the data, then the bounds themselves might reveal something about the data.

Typically, clipping bounds are decided either by using a property of the dataset that can be known without looking at the data (e.g. that the dataset contains ages, which are likely to lie between 0 and 125), or by performing differentially private queries to evaluate different choices for the clipping bounds.

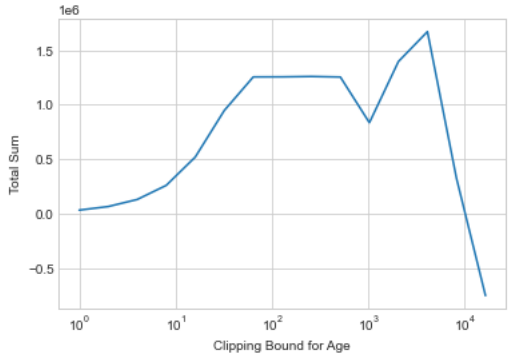
To use the second approach, we typically set the lower bound to 0 and slowly increase the upper bound until the query's output stops changing (meaning we haven't included any new data by increasing the bound). For example, let's try computing the sum of ages for clipping bounds from 0 to 100, using the Laplace mechanism for each one to ensure differential privacy:



The total privacy cost for building this plot is $\epsilon = 1$ by sequential composition, since we do 100 queries each with $\epsilon_i = 0.01$. It's clear that the results level off around a value of `upper = 80`, so this is a good choice for the clipping bound.

We can use the same approach for data attributes from any numerical domain, but it helps to know something about the scale of the data in advance. For example, trying clipping values between 0 and 100 for yearly incomes would not work very well - we wouldn't even come close to finding a reasonable upper bound.

One refinement that can work well when the scale of the data is not known is to test upper bounds according to a logarithmic scale.



This approach allows us to test a huge range of possible bounds with a small number of queries, but at the expense of less precision in determining the perfect bound. As the upper bound gets really large, the noise will start to overwhelm the signal - notice how the sum fluctuates wildly for the largest clipping parameters! The key is to look for a region of the graph which is relatively smooth (meaning low noise) and also not increasing (meaning the clipping bound is sufficient). Here, this occurs at roughly $2^8 = 256$, which is a reasonable approximation of the upper bound we derived earlier.

Approximate Differential Privacy

> **Learning Objectives**
>
> After reading this chapter, you will be able to:
>
> - Define approximate differential privacy
> - Explain the differences between approximate and pure differential privacy
> - Describe the advantages and disadvantages of approximate differential privacy
> - Describe and calculate L1 and L2 sensitivity of vector-valued queries
> - Define and apply the Gaussian mechanism
> - Apply advanced composition

Approximate differential privacy, also called (ϵ, δ)-differential privacy, has the following definition:

\begin{align} \mathsf{Pr}[F(x) = S] \leq e^\epsilon \mathsf{Pr}[F(x') = s] + \delta \end{align}

The new privacy parameter, δ, represents a "failure probability" for the definition. With probability $1 - \delta$, we will get the same guarantee as pure differential privacy; with probability δ, we get no guarantee. In other words:

- With probability $1 - \delta$, $\frac{\Pr[F(x)=S]}{\Pr[F(x')=s]} \leq e^{\epsilon}$
- With probability δ, we get no guarantee at all

This definition should seem a little bit scary! With probability δ, anything at all could happen - including a release of the entire sensitive dataset! For this reason, we typically require δ to be very small - usually $\frac{1}{n^2}$ or less, where n is the size of the dataset. In addition, we'll see that the (ϵ, δ)-differentially private mechanisms in practial use don't fail catastrophically, as allowed by the definition - instead, they fail *gracefully*, and don't do terrible things like releasing the entire dataset.

Such mechanisms *are* possible, however, and they do satisfy the definition of (ϵ, δ)-differential privacy. We'll see an example of such a mechanism later in this section.

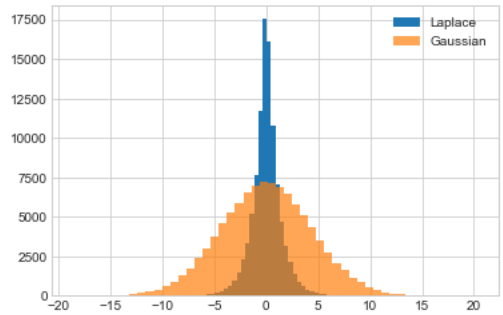
The Gaussian Mechanism

The Gaussian mechanism is an alternative to the Laplace mechanism, which adds Gaussian noise instead of Laplacian noise. The Gaussian mechanism does *not* satisfy pure ϵ-differential privacy, but does satisfy (ϵ, δ)-differential privacy. According to the Gaussian mechanism, for a function $f(x)$ which returns a number, the following definition of $F(x)$ satisfies (ϵ, δ)-differential privacy:

\begin{align} F(x) = f(x) + \mathcal{N}(\sigma^2)\ \text{where } \sigma^2 = \frac{2s^2 \log(1.25/\delta)}{\epsilon^2} \end{align}

where s is the sensitivity of f, and $\mathcal{N}(\sigma^2)$ denotes sampling from the Gaussian (normal) distribution with center 0 and variance σ^2. Note that here (and elsewhere in these notes), $\log$ denotes the natural logarithm.

For real-valued functions $f : D \to \mathbb{R}$, we can use the Gaussian mechanism in exactly the same way as we do the Laplace mechanism, and it's easy to compare what happens under both mechanisms for a given value of ϵ.



Here, we graph the empirical probability density function of the Laplace and Gaussian mechanisms for $\epsilon = 1$, with $\delta = 10^{-5}$ for the Gaussian mechanism.

Compared to the Laplace mechanism, the plot for the Gaussian mechanism looks "squished." Differentially private outputs which are far from the true answer are much more likely using the Gaussian mechanism than they are under the Laplace mechanism (which, by comparison, looks extremely "pointy").

So the Gaussian mechanism has two major drawbacks - it requires the use of the the relaxed (ϵ, δ)-differential privacy definition, *and* it's less accurate than the Laplace mechanism. Why would we want to use it?

Vector-Valued Functions and their Sensitivities

So far, we have only considered real-valued functions (i.e. the function's output is always a single real number). Such functions are of the form $f : D \to \mathbb{R}$. Both the Laplace and Gaussian mechanism, however, can be extended to *vector-valued* functions of the form $f : D \to \mathbb{R}^k$, which return vectors of real numbers. We can think of histograms as vector-valued functions, which return a vector whose elements consist of histogram bin counts.

We saw earlier that the *sensitivity* of a function is:

\begin{align} GS(f) = \max_{d(x,x') \leq 1} \lvert f(x) - f(x') \rvert \end{align}

How do we define sensitivity for vector-valued functions?

Consider the expression $f(x) - f(x')$. If f is a vector-valued function, then this expression represents the difference between two vectors, which can be computed as the difference between their corresponding elements (the difference of two length-k vectors is thus a new length-k vector). This new vector is the distance between $f(x)$ and $f(x')$, represented as a vector.

The magnitude of this vector is the sensitivity of f. There are several ways to compute the magnitude of a vector; we'll use two of them: the $L1$ norm and the $L2$ norm.

L1 and L2 Norms

The $L1$ norm of a vector V of length k is defined as $\|V\|_1 = \sum_{i=1}^{k} |V_i|$ (i.e. it's the sum of the vector's elements). In 2-dimensional space, the $L1$ norm of the difference between two vectors yields the "manhattan distance" between them.

The $L2$ norm of a vector V of length k is defined as $\|V\|_2 = \sqrt{\sum_{i=1}^{k} V_i^2}$ (i.e. the square root of the sum of the squares). In 2-dimensional space, this is the "Euclidian distance," and it's always less than or equal to the $L1$ distance.

L1 and L2 Sensitivities

The $L1$ sensitivity of a vector-valued function f is:

\begin{align} GS(f) = \max_{d(x,x') \leq 1} \lVert f(x) - f(x') \rVert_1 \end{align}

This is equal to the sum of the *elementwise* sensitivities. For example, if we define a vector-valued function f that returns a length-k vector of 1-sensitive results, then the $L1$ sensitivity of f is k.

Similarly, the $L2$ sensitivity of a vector-valued function f is:

\begin{align} GS_2(f) = \max_{d(x,x') \leq 1} \lVert f(x) - f(x') \rVert_2 \end{align}

Using the same example as above, a vector-valued function f returning a length-k vector of 1-sensitive results has $L2$ sensitivity of $\sqrt{k}$. For long vectors, the $L2$ sensitivity will obviously be much lower than the $L1$ sensitivity! For some applications, like machine learning algorithms (which sometimes return vectors with thousands of elements), $L2$ sensitivity is *significantly* lower than $L1$ sensitivity.

Choosing Between L1 and L2

As mentioned earlier, both the Laplace and Gaussian mechanisms can be extended to vector-valued functions. However, there's a key difference between these two extensions: the vector-valued Laplace mechanism **requires** the use of $L1$ sensitivity, while the vector-valued Gaussian mechanism allows the use of either $L1$ or $L2$ sensitivity. This is a major strength of the Gaussian mechanism. For applications in which $L2$ sensitivity is much lower than $L1$ sensitivity, the Gaussian mechansim allows adding *much* less noise.

- The **vector-valued Laplace mechanism** releases $f(x) + (Y_1, \ldots, Y_k)$, where Y_i are drawn i.i.d. from the Laplace distribution with scale $\frac{s}{\epsilon}$ and s is the $L1$ sensitivity of f
- The **vector-valued Gaussian mechanism** releases $f(x) + (Y_1, \ldots, Y_k)$, where Y_i are drawn i.i.d. from the Gaussian distribution with $\sigma^2 = \frac{2s^2 \log(1.25/\delta)}{\epsilon^2}$ and s is the $L2$ sensitivity of f

The Catastrophe Mechanism

The definition of (ϵ, δ)-differential privacy says that a mechanism which satisfies the definition must "behave well" with probability $1 - \delta$. That means that with probability δ, the mechanism can do anything at all. This "failure probability" is concerning, because mechanisms which satisfy the relaxed definition may (with low probability) result in very bad outcomes.

Consider the following mechanism, which we will call the *catastrophe mechansim*:

\begin{align} F(q, x) =;& \text{Sample a number } r \text{ from the uniform distribution between 0 and 1}\ &\text{If } r < \delta, \text{return } x\ &\text{Otherwise, return } q(x) + \text{Lap}(s/\epsilon), \text{where } s \text{ is the sensitivity of } q\ \end{align}

With probability $1 - \delta$, the catastrophe mechanism satisfies ϵ-differential privacy. With probability δ, it *releases the whole dataset with no noise*. This mechanism satisfies the definition of approximate differential privacy, but we probably wouldn't want to use it in practice.

Fortunately, most (ϵ, δ)-differentially private mechanisms don't have such a catastrophic failure mode. The Gaussian mechanism, for example, doesn't ever release the whole dataset. Instead, with probability δ, the Gaussian mechanism doesn't *quite* satisfy ϵ-differential privacy - it satisfies $c\epsilon$-differential privacy instead, for some value c.

The Gaussian mechanism thus fails *gracefully*, rather than catistrophically, so it's reasonable to have far more confidence in the Gaussian mechanism than in the catastrophe mechanism. Later, we will see alternative relaxations of the definition of differential privacy which distinguish between mechanisms that fail gracefully (like the Gaussian mechanism) and ones that fail catastropically (like the catastrophe mechanism).

Advanced Composition

We have already seen two ways of combining differentially private mechanisms: sequential composition and parallel composition. It turns out that (ϵ, δ)-differential privacy admits a new way of analyzing the sequential composition of differentially private mechanisms, which can result in a lower privacy cost.

The advanced composition theorem is usually stated in terms of mechanisms which are instances of k*-fold adaptive composition*. A k-fold adaptive composition is a sequence of mechanisms $m_1, \ldots, m_k$ such that:

- Each mechanism m_i may be chosen based on the outputs of all previous mechanisms $m_1, \ldots, m_{i-1}$ (hence *adaptive*)
- The input to each mechanism m_i is both the private dataset and all of the outputs of previous mechanisms (hence *composition*)

Iterative programs (i.e. loops or recursive functions) are nearly always instances of k-fold adaptive composition. A `for` loop that runs 1000 iterations, for example, is a 1000-fold adaptive composition. As a more specific example, an averaging attack is a k-fold adaptive composition:

```
# works for sensitivity-1 queries
def avg_attack(query, epsilon, k):
    results = [query + np.random.laplace(loc=0, scale=1/epsilon) for i in range(k)]
    return np.mean(results)

avg_attack(10, 1, 500)
```

```
10.007228493453594
```

In this example, the sequence of mechanisms is fixed ahead of time (we use the same mechanism each time), and $k = 500$.

The standard sequential composition theorem says that the total privacy cost of this mechanism is $k\epsilon$ (in this case, 500ϵ).

The advanced composition theorem says:

- If each mechanism m_i in a k-fold adaptive composition $m_1, \ldots, m_k$ satisfies ϵ-differential privacy
- Then for $\epsilon', \delta \geq 0$, the total privacy cost of the entire k-fold adaptive composition is equal to ϵ', δ, where:

\begin{align} \epsilon' = 2\epsilon \sqrt{2k \log(1/\delta)} \end{align}

Plugging in $\epsilon = 1$ from the example above, and setting $\delta = 10^{-5}$, we get:

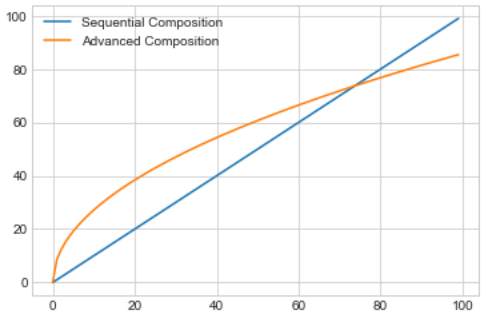
\begin{align} \epsilon' =& 2 \sqrt{1000 \log(100000)}\ \approx& 214.59 \end{align}

So advanced composition derives a much lower bound on ϵ' than sequential composition, *for the same mechanism*. What does this mean? It means that the bounds given by sequential composition are *loose* - they don't tightly bound the *actual* privacy cost of the computation. In fact, advanced composition also gives loose bounds - they're just

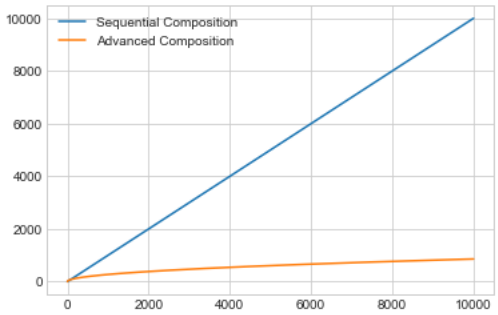
slightly *less* loose than the ones given by sequential composition.

It's important to note that the two bounds are technically incomparable, since advanced composition introduces a δ. When δ is small, however, we will often compare the ϵs given by both methods.

So, should we *always* use advanced composition? It turns out that we should *not*. Let's try the experiment above for different values of k, and graph the *total privacy cost* under both sequential compsosition and advanced composition.



Standard sequential composition, it turns out, beats advanced composition for k smaller than about 70. Thus, advanced composition is only really useful when k is large (e.g. more than 100). When k is very large, though, advanced composition can make a *big* difference.



Local Sensitivity

> **Learning Objectives**
>
> After reading this chapter, you will be able to:
>
> - Define local sensitivity and explain how it differs from global sensitivity
> - Describe how local sensitivity can leak information about the data
> - Use propose-test-release to safely apply local sensitivity
> - Describe the smooth sensitivity framework
> - Use the sample-and-aggregate framework to answer queries with arbitrary sensitivity

So far, we have seen only one measure of sensitivity: global sensitivity. Our definition for global sensitivity considers *any* two neighboring datasets. This seems pessimistic, since we're going to run our differentially private mechanisms on an *actual* dataset - shouldn't we consider neighbors of *that* dataset?

This is the intuition behind *local sensitivity*: fix one of the two datasets to be the *actual* dataset being queried, and consider all of its neighbors. Formally, the local sensitivity of a function $f : \mathcal{D} \rightarrow \mathbb{R}$ at $x : \mathcal{D}$ is defined as:

\begin{align} LS(f, x) = \max_{x': d(x,x') \leq 1} \lvert f(x) - f(x') \rvert \end{align}

Notice that local sensitivity is a function of both the query (f) and the *actual* dataset (x). Unlike in the case of global sensitivity, we can't talk about the local sensitivity of a function without also considering the dataset *at which* that local sensitivity occurs.

Local Sensitivity of the Mean

Local sensitivity allows us to place finite bounds on the sensitivity of some functions whose global sensitivity is difficult to bound. The mean function is one example. So far, we've calculated differentially private means by splitting the query into two queries: a differentially private sum (the numerator) and a differentially private count (the denominator). By sequential composition and post-processing, the quotient of these two results satisfies differential privacy.

Why do we do it this way? Because the amount the output of a mean query might change when a row is added to or removed from the dataset *depends on the size of the dataset*. If we want to bound the global sensitivity of a mean query, we have to assume the worst: a dataset of size 1. In this case, if the data attribute values lie between upper and lower bounds u and l, the global sensitivity of the mean is just $|u - l|$. For large datasets, this is *extremely* pessimistic, and the "noisy sum over noisy count" approach is much better.

The situation is different for local sensitivity. In the worst case, we can add a new row to the dataset which contains the maximum value (u). Let $n = |x|$ (i.e. the size of the dataset). We start with the value of the mean:

\begin{align} f(x) =& \frac{\sum_{i=1}^{n} x_i}{n} \end{align}

Now we consider what happens when we add a row:

\begin{align} \lvert f(x') - f(x) \rvert = & \bigg\lvert \frac{\sum_{i=1}^{n} x_i + u}{n+1} - \frac{\sum_{i=1}^{n} x_i}{n} \bigg\rvert \ \leq& \bigg\lvert \frac{\sum_{i=1}^{n} x_i + u}{n+1} - \frac{\sum_{i=1}^{n} x_i}{n+1} \bigg\rvert \ =& \bigg\lvert \frac{\sum_{i=1}^{n} x_i + u - \sum_{i=1}^{n} x_i}{n+1}\bigg\rvert \ =& \bigg\lvert \frac{u}{n+1} \bigg\rvert \ \end{align}

This local sensitivity measure is defined in terms of the actual dataset's size, which is not possible under global sensitivity.

Achieving Differential Privacy via Local Sensitivity?

We have defined an alternative measure of sensitivity - but how do we use it? Can we just use the Laplace mechanism, in the same way as we did with global sensitivity? Does the following definition of F satisfy ϵ-differential privacy?

\begin{align} F(x) = f(x) + \mathsf{Lap}(\frac{LS(f,x)}{\epsilon}) \end{align}

No! Unfortunately not, since $LS(f, x)$ itself depends on the dataset. If the analyst knows the local sensitivity of a query *at a particular dataset*, then the analyst may be able to infer some information about the dataset. It's therefore *not possible* to use local sensitivity directly to achieve differential privacy. For example, consider the bound on local sensitivity for the mean, defined above. If we know the local sensitivity at a particular x, we can infer the exact size of x with *no noise*:

\begin{align} \lvert x \rvert = \frac{b}{LS(f, x)} - 1 \end{align}

Moreover, keeping the local sensitivity secret from the analyst *doesn't help either*. It's possible to determine the scale of the noise from just a few query answers, and the analyst can use this value to infer the local sensitivity. Differential privacy is designed to protect the output of $f(x)$ - *not* of the sensitivity measure used in its definition.

Several approaches have been proposed for safely using local sensitivity. We'll explore these in the rest of this section.

With auxiliary data, this can tell us something really sensitive. What if our query is: "Average score of people named Joe in the dataset with a 98% on the exam"? Then the size of the thing being averaged is sensitive!!

Propose-test-release

The primary problem with local sensitivity is that the sensitivity itself reveals something about the data. What if we make the *sensitivity itself* differentially private? This is challenging to do directly, as there's often no finite bound on the global sensitivity of a function's local sensitivity. However, we can ask a differentially private question that gets at this value indirectly.

The *propose-test-release* framework takes this approach. The framework first asks the analyst to *propose* an upper bound on the local sensitivity of the function being applied. Then, the framework runs a differentially private *test* to check that the dataset being queried is "far from" a dataset where local sensitivity is higher than the proposed bound. If the test passes, the framework *releases* a noisy result, with the noise calibrated to the proposed bound.

In order to answer the question of whether a dataset is "far from" one with high local sensitivity, we define the notion of *local sensitivity at distance* k. We write $A(f, x, k)$ to denote the maximum local sensitivity achievable for f by taking k steps away from the dataset x. Formally:

\begin{align} A(f,x,k) = \max_{y: d(x,y) \leq k} LS(f, y) \end{align}

Now we're ready to define a query to answer the question: "how many steps are needed to achieve a local sensitivity greater than a given upper bound b?"

\begin{align} D(f, x, b) = \text{argmin}_k A(f, x, k) > b \end{align}

Finally, we define the propose-test-release framework (see Barthe et al., Figure 10), which satisfies (ϵ, δ)-differential privacy:

1. Propose a target bound b on local sensitivity.
2. If $D(f, x, b) + \text{Lap}(\frac{1}{\epsilon}) < \frac{\log(2/\delta)}{2\epsilon}$, return $\perp$.
3. Return $f(x) + Lap(\frac{b}{\epsilon})$

Notice that $D(f, x, b)$ has a *global* sensitivity of 1: adding or removing a row in x might change the distance to a "high" local sensitivity by 1. Thus, adding Laplace noise scaled to $\frac{1}{\epsilon}$ yields a differentially private way to measure local sensitivity.

Why does this approach satisfy (ϵ, δ)-differential privacy (and not pure ϵ-differential privacy)? It's because there's a non-zero chance of *passing the test by accident*. The noise added in step 2 might be large enough to pass the test, even though the value of $D(f, x, b)$ is actually *less* than the minimum distance required to satisfy differential privacy.

This failure mode is much closer to the catastrophic failure we saw from the "catastrophe mechanism" - with non-zero probability, the propose-test-release framework allows releasing a query answer with *far* too little noise to satisfy differential privacy. On the other hand, it's not nearly as bad as the catastrophe mechanism, since it never releases the answer with *no* noise.

Also note that the privacy cost of the framework is (ϵ, δ) *even if* it returns $\perp$ (i.e. the privacy budget is consumed whether or not the analyst receives an answer).

Let's implement propose-test-release for our mean query. Recall that the local sensitivity for this query is $\left|\frac{u}{n+1}\right|$; the best way to increase this value is to make n smaller. If we take k steps from the dataset x, we can arrive at a local sensitivity of $\left|\frac{u}{(n-k)+1}\right|$. We can implement the framework in Python using the following code.

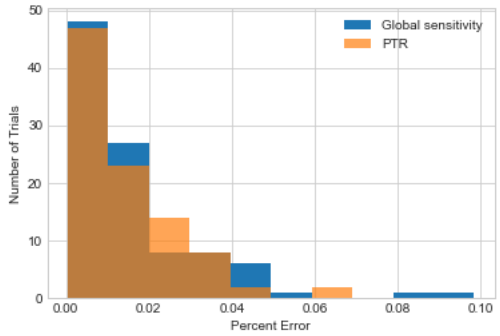
```
df = adult['Age']
u = 100                     # set the upper bound on age to 100
epsilon = 1                 # set epsilon = 1
delta = 1/(len(df)**2)      # set delta = 1/n^2
b = 0.005                   # propose a sensitivity of 0.005

ptr_avg(df, u, b, epsilon, delta, logging=True)
```

```
Noisy distance is 12562.135267153066 and threshold is 10.73744412245554
```

```
38.58250564003604
```

Keep in mind that local sensitivity isn't always better. For mean queries, our old strategy of splitting the query into two separate queries (a sum and a count), both with bounded global sensitivity, often works much better. We can implement the same mean query with global sensitivity:

```
38.585343734556254
```



We might do slightly better with propose-test-release, but it's not a huge difference. Moreover, to use propose-test-release, the analyst has to propose a bound on sensitivity - and we've cheated by "magically" picking a decent value (0.005). In practice, the analyst would need to perform several queries to explore which values work - which will consume additional privacy budget.

Smooth Sensitivity

Our second approach for leveraging local sensitivity is called *smooth sensitivity*, and is due to Nissim, Raskhodnikova, and Smith. The *smooth sensitivity framework*, instantiated with Laplace noise, provides (ϵ, δ)-differential privacy:

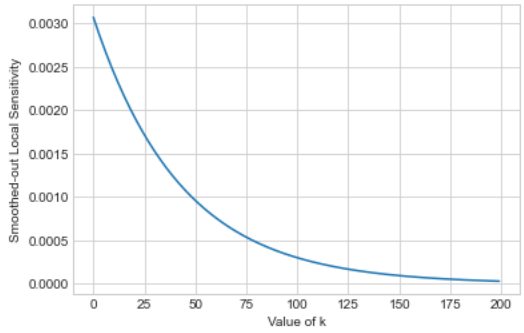
1. Set $\beta = \frac{\epsilon}{2\log(2/\delta)}$
2. Let $S = \max_{k=1,\ldots,n} e^{-\beta k} A(f, x, k)$
3. Release $f(x) + \text{Lap}(\frac{2S}{\epsilon})$

The idea behind smooth sensitivity is to use a "smooth" approximation of local sensitivity, rather than local sensitivity itself, to calibrate the noise. The amount of smoothing is designed to prevent the unintentional release of information about the dataset that can happen when local sensitivity is used directly. Step 2 above performs the smoothing: it scales the local sensitivity of nearby datasets by an exponential function of their distance from the actual dataset, then takes the maximum scaled local sensitivity. The effect is that if a spike in local sensitivity exists in the neighborhood of x, that spike will be reflected in the smooth sensitivity of x (and therefore the spike itself is "smoothed out," and doesn't reveal anything about the dataset).

Smooth sensitivity has a significant advantage over propose-test-release: it doesn't require the analyst to propose a bound on sensitivity. For the analyst, using smooth sensitivity is just as easy as using global sensitivity. However, smooth sensitivity has two major drawbacks. First, smooth sensitivity is always larger than local sensitivity (by at least a factor of 2 - see step 3), so it may require adding quite a bit more noise than alternative frameworks like propose-test-release (or even global sensitivity). Second, calculating smooth sensitivity requires finding the maximum smoothed-out sensitivity over *all* possible values for k, which can be extremely challenging computationally. In many cases, it's possible to prove that considering a small number of values for k is sufficient (for many functions, the exponentially decaying $e^{-\beta k}$ quickly overwhelms the growing value of $A(f, x, k)$), but such a property has to be proven for *each* function we want to use with smooth sensitivity.

As an example, let's consider the smooth sensitivity of the mean query we defined earlier.

```
Final sensitivity: 0.006142128861863522
```



There are two things to notice here. First, even though we consider only values of k less than 200, it's pretty clear that the smoothed-out local sensitivity of our mean query approaches 0 as k grows. In fact, for this case, the maximum occurs at $k = 0$. This is true in many cases, but if we want to use smooth sensitivity, we have to *prove* it (which we won't do here). Second, notice that the final sensitivity we'll use for adding noise to the query's answer is *higher* than the sensitivity we proposed earlier (under propose-test-release). It's not a big difference, but it shows that it's sometimes *possible* to achieve a lower sensitivity with propose-test-release than with smooth sensitivity.

Sample and Aggregate

We'll consider one last framework related to local sensitivity, called *sample and aggregate* (also due to Nissim, Raskhodnikova, and Smith). For any function $f : D \to \mathbb{R}$ and upper and lower clipping bounds u and l, the following framework satisfies ϵ-differential privacy:

1. Split the dataset $X \in D$ into k disjoint chunks $x_1, \ldots, x_k$
2. Compute a clipped answer for each chunk: $a_i = \max(l, \min(u, f(x_i)))$
3. Compute a noisy average of the answers: $A = (\frac{1}{k} \sum_{i=1}^{k} a_i) + \text{Lap}(\frac{u-l}{k\epsilon})$

Note that this framework satisfies pure ϵ-differential privacy, and it actually works *without* the use of local sensitivity. In fact, we don't need to know *anything* about the sensitivity of f (global or local). We also don't need to know anything about the chunks x_i, except that they're disjoint. Often, they're chosen randomly ("good" samples tend to result in higher accuracy), but they don't need to be.

The framework can be shown to satisfy differential privacy just by global sensitivity and parallel composition. We split the dataset into k chunks, so each individual appears in exactly one chunk. We don't know the sensitivity of f, but we clip its output to lie between u and l, so the sensitivity of each clipped answer $f(x_i)$ is $u - l$. Since we take the mean of k invocations of f, the global sensitivity of the mean is $\frac{u-l}{k}$.

Note that we're claiming a bound on the global sensitivity of a mean *directly*, rather than splitting it into sum and count queries. We weren't able to do this for "regular" mean queries, because the number of things being averaged in a "regular" mean query depends on the dataset. In this case, however, the number of items being averaged is *fixed* by the analyst, via the choice of k - it's *independent* of the dataset. Mean queries like this one - where the number of things being averaged is fixed, and can be made public - can leverage this improved bound on global sensitivity.

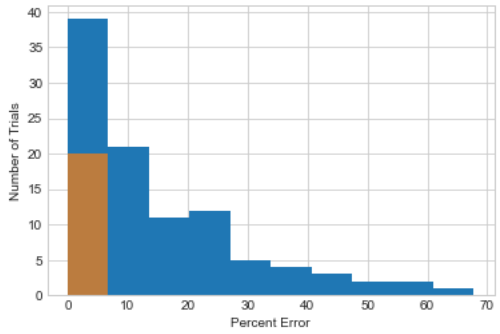
In this simple instantation of the sample and aggregate framework, we ask the analyst to provide the upper and lower bounds u and l on the *output* of each $f(x_i)$. Depending on the definition of f, this might be *extremely* difficult to do well. In a counting query, for example, f's output will depend directly on the dataset.

More advanced instantiations have been proposed (Nissim, Raskhodnikova, and Smith discuss some of these) which leverage local sensitivity to avoid asking the analyst for u and l. For some functions, however, bounding f's output is easy - so this framework suffices. We'll consider our example from above - the mean of ages within a dataset - with this property. The mean age of a population is highly likely to fall between 20 and 80, so it's reasonable to set $l = 20$ and $u = 80$. As long as our chunks x_i are each representative of the population, we're not likely to lose much information with this setting.

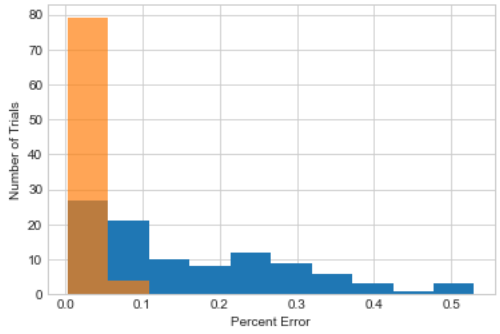
The key parameter in this framework is the number of chunks, k. As k goes up, the sensitivity of the final noisy mean goes *down* - so more chunks means less noise. On the other hand, as k goes up, each chunk gets *smaller*, so each answer $f(x_i)$ is less likely to be close to the "true" answer $f(X)$. In our example above, we'd like the average age within each chunk to be close to the average age of the whole dataset - and this is less likely to happen if each chunk contains only a handful of people.

How should we set k? It depends on f and on the dataset, which makes it tricky. Let's try various values of k for our mean query.

```
Text(0, 0.5, 'Number of Trials')
```




```
Text(0, 0.5, 'Number of Trials')
```



```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-13-9f43a347f3ee> in <module>
      1 # k = 6000; sample and aggregate is getting close!
----> 2 plot_results(6000)
      3 plt.xlabel('Percent Error')
      4 plt.ylabel('Number of Trials')

<ipython-input-10-c588e46b05ac> in plot_results(k)
      1 def plot_results(k):
      2     df = adult['Age']
----> 3     _, bins, _ = plt.hist([pct_error(np.mean(df), saa_avg_age(k, epsilon))
for i in range(100)]);
      4     plt.hist([pct_error(np.mean(df), gs_avg(df, u, epsilon)) for i in
range(100)], alpha=.7, bins=bins);

<ipython-input-10-c588e46b05ac> in <listcomp>(.0)
      1 def plot_results(k):
      2     df = adult['Age']
----> 3     _, bins, _ = plt.hist([pct_error(np.mean(df), saa_avg_age(k, epsilon))
for i in range(100)]);
      4     plt.hist([pct_error(np.mean(df), gs_avg(df, u, epsilon)) for i in
range(100)], alpha=.7, bins=bins);

<ipython-input-9-b62d5128b2e1> in saa_avg_age(k, epsilon, logging)
     15
     16     # Step 2: run f on each x_i and clip its output
---> 17     answers = [f(x_i) for x_i in xs]
     18
     19     u = 80

<ipython-input-9-b62d5128b2e1> in <listcomp>(.0)
     15
     16     # Step 2: run f on each x_i and clip its output
---> 17     answers = [f(x_i) for x_i in xs]
     18
     19     u = 80

<ipython-input-9-b62d5128b2e1> in f(df)
      1 def f(df):
----> 2     return df.mean()
      3
      4 def saa_avg_age(k, epsilon, logging=False):
      5     df = adult['Age']

/usr/local/lib/python3.9/site-packages/pandas/core/generic.py in stat_func(self,
axis, skipna, level, numeric_only, **kwargs)
  11459             nv.validate_median(tuple(), kwargs)
  11460         else:
> 11461             nv.validate_stat_func(tuple(), kwargs, fname=name)
  11462         if skipna is None:
  11463             skipna = True

KeyboardInterrupt:
```

So - sample and aggregate isn't able to beat our global sensitivity-based approach, but it can get pretty close if you choose the right value for k. The big advantage is that sample and aggregate works for *any* function f, regardless of its sensitivity; if f is well-behaved, then it's possible to obtain good accuracy from the framework. On the other hand, using sample and aggregate requires the analyst to set the clipping bounds u and l, and the number of chunks k.

Variants of Differential Privacy

Learning Objectives

After reading this chapter, you will be able to:

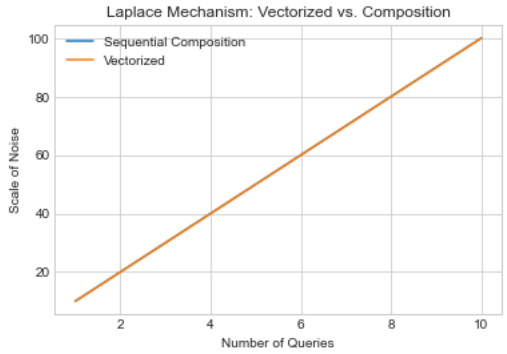
- Define Rényi differential privacy and zero-concentrated differential privacy
- Describe the advantages of these variants over (ϵ, δ)-differential privacy
- Convert privacy costs from these variants into (ϵ, δ)-differential privacy

Recall that most of the bounds on privacy cost we have shown are *upper* bounds, but they sometimes represent very loose upper bounds - the true privacy cost is much less than the upper bound says. The primary motivation in developing new variants of differential privacy is to enable tighter bounds on privacy cost - especially for iterative algorithms - while maintaining privacy definitions which are useful in practice. For example, the catastrophic failure mode of (ϵ, δ)-differential privacy is not desirable; the variants we'll see in this section enable even tighter composition for some kinds of queries, while at the same time *eliminating* the catastropic failure mode.

Let's take a quick look at the tools we have already seen; we'll look first at sequential composition for ϵ-differential privacy. It turns out that sequential composition for ϵ-differential privacy is *tight*. What does that mean? It means there's a counterexample that would fail to satisfy any lower bound:

- A mechanism F exists which satisfies ϵ-differential privacy
- When composed k times, F satisfies $k\epsilon$-differential privacy
- But F does *not* satisfy $c\epsilon$-differential privacy for any $c < k$

A neat way to visualize this is to look at what happens to privacy cost when we "vectorize" a query: that is, we merge lots of queries into a single query which returns a vector of the individual answers. Because the answer is a vector, we can use the vector-valued Laplace mechanism just once, and avoid composition altogether. Below, we'll graph how much noise is needed for k queries, first under sequential composition, and then using the "vectorized" form. In the sequential composition case, each query has a sensitivity of 1, so the scale of the noise for each one is $\frac{1}{\epsilon_i}$. If we want a total privacy cost of ϵ, then the ϵ_is must add up to ϵ, so $\epsilon_i = \frac{\epsilon}{k}$. This means that each query gets Laplace noise with scale $\frac{k}{\epsilon}$. In the "vectorized" case, there's just one query, but it has an L1 sensitivity of $\sum_{i=1}^{k} 1 = k$, so the scale of the noise is $\frac{k}{\epsilon}$ in this case too.



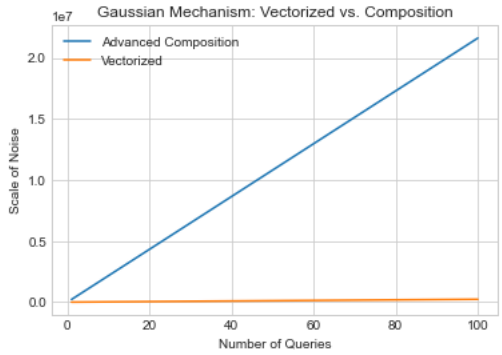
The two lines overlap *completely*. This means that no matter how many queries we're running, under ϵ-differential privacy, we can't do any better than sequential composition. That's because sequential composition is just as good as vectorizing the query, and we can't do any better than that.

What about (ϵ, δ)-differential privacy? The story is a little different there. In the sequential composition case, we can use advanced composition; we have to be a little careful to ensure that the total privacy cost is exactly (ϵ, δ). Specifically, we set $\epsilon_i = \frac{\epsilon}{2\sqrt{2k \log(1/\delta')}}$, $\delta_i = \frac{\delta}{2k}$, and $\delta' = \frac{\delta}{2}$ (splitting δ to go 50% towards the queries, and 50% towards advanced composition). By advanced composition, the total privacy cost for all k queries is (ϵ, δ). The scale of the noise, by the Gaussian mechanism, is:

\begin{align} \sigma^2 =& \frac{2 \log(\frac{1.25}{\delta_i})}{\epsilon_i^2}\ =& \frac{16 k \log(\frac{1}{\delta'}) \log(\frac{1.25}{\delta_i})}{\epsilon^2}\ =& \frac{16 k \log(\frac{2}{\delta}) \log(\frac{2.5 k}{\delta})}{\epsilon^2}\ \end{align}

In the "vectorized" case, we have just one query, with an L2 sensitivity of $\sqrt{k}$. The scale of the noise, by the Gaussian mechanism, is $\sigma^2 = \frac{2k \log(1.25/\delta)}{\epsilon^2}$.

What does this difference mean in practice? The two behave the same asymptotically in k, but have different constants, and the advanced composition case has an additional logarithmic factor in δ. All this adds up to a much looser bound in the case of advanced composition. Let's graph the two as we did before.



It's not even close - the "vectorized" version grows *much* slower. What does this mean? We should be able to do *much* better for sequential composition!

Max Divergence and Rényi Divergence

It turns out that the definition of differential privacy can be stated directly in terms of something called *max divergence*. In statistics, a *divergence* is a way of measuring the distance between two probability distributions - which is exactly what we want to do for differential privacy. The *max divergence* is the worst-case analog of the Kullback–Leibler divergence, one of the most common such measures. The max divergence between two probability distributions Y and Z is defined to be:

\begin{align} D_\infty(Y \Vert Z) = \max_{S \subseteq \text{Supp}(Y)} \Big[\log \frac{Pr[Y \in S]}{Pr[Z \in S]} \Big] \end{align}

This already looks a lot like the condition for ϵ-differential privacy! In particular, it turns out that F satisfies ϵ-differential privacy if:

\begin{align} D_\infty(F(x) \Vert F(x') \leq \epsilon \end{align}

An interesting direction for research in differential privacy is the exploration of alternative privacy definitions in terms of other divergences. Of these, the Rényi divergence is particularly interesting, since it also (like max divergence) allows us to recover the original definition of differential privacy. The Rényi divergence of order α between probability distributions P and Q is defined as (where $P(x)$ denotes the probability density of P at point x):

\begin{align} D_\alpha(P \Vert Q) = \frac{1}{\alpha - 1} \log E_{x \sim Q} \Big(\frac{P(x)}{Q(x)}\Big)^\alpha \end{align}

If we set $\alpha = \infty$, then we immediately recover the definition of ϵ-differential privacy! The obvious question arises: what happens if we set α to something else? As we'll see, it's possible to use the Rényi divergence to derive really interesting relaxations of differential privacy that allow better composition theorems while at the same time avoiding the possibility of "catastrophe" which is possible under (ϵ, δ)-differential privacy.

Rényi Differential Privacy

In 2017, Ilya Mironov proposed Rényi differential privacy (RDP). A randomized mechanism F satisfies $(\alpha, \bar{\epsilon})$-RDP if for all neighboring datasets x and x'

\begin{align} D_\alpha(F(x) \Vert F(x')) \leq \bar{\epsilon} \end{align}

In other words, RDP requires that the Rényi divergence of order α between $F(x)$ and $F(x')$ to be bounded by $\bar{\epsilon}$. Note that we'll use $\bar{\epsilon}$ to denote the ϵ parameter of RDP, in order to distinguish it from the ϵ in pure ϵ-differential privacy and (ϵ, δ)-differential privacy.

A key property of Rényi differential privacy is that a mechanism which satisfies RDP also satisfies (ϵ, δ)-differential privacy. Specifically, if F satisfies $(\alpha, \bar{\epsilon})$-RDP, then for $\delta > 0$, F satisfies (ϵ, δ)-differential privacy for $\epsilon = \bar{\epsilon} + \frac{\log(1/\delta)}{\alpha - 1}$. The analyst is free to pick any value of δ; a meaningful value (e.g. $\delta \leq \frac{1}{n^2}$) should be picked in practice.

The basic mechanism for achieving Rényi differential privacy is the Gaussian mechanism. Specifically, for a function $f : \mathcal{D} \to \mathbb{R}^k$ with $L2$ sensitivity Δf, the following mechanism satisfies $(\alpha, \bar{\epsilon})$-RDP:

\begin{align} F(x) = f(x) + \mathcal{N}(\sigma^2) \text{ where } \sigma^2 = \frac{\Delta f^2 \alpha}{2\epsilon} \end{align}

We can implement the Gaussian mechanism for Rényi differential privacy as follows:

```
def gaussian_mech_RDP_vec(vec, sensitivity, alpha, epsilon):
    sigma = np.sqrt((sensitivity**2 * alpha) / (2 * epsilon))
    return [v + np.random.normal(loc=0, scale=sigma) for v in vec]
```

The major advantage of Rényi differential privacy is *tight composition* for the Gaussian mechanism - and this advantage in composition comes without the need for a special advanced composition theorem. The sequential composition theorem of Rényi differential privacy states that:

- If F_1 satisfies $(\alpha, \bar{\epsilon_1})$-RDP
- And F_2 satisfies $(\alpha, \bar{\epsilon_2})$-RDP
- Then their composition satisfies $(\alpha, \bar{\epsilon_1} + \bar{\epsilon_2})$-RDP

Based on this sequential composition theorem, running an $(\alpha, \bar{\epsilon})$-RDP mechanism k times results in $(\alpha, k\bar{\epsilon})$-RDP. For a given level of noise (i.e. a given value for σ^2), bounding the privacy cost of repeated applications of the Gaussian mechanism using RDP's sequential composition, and *then* converting to (ϵ, δ)-differential privacy, will usually yield a *much* lower privacy cost than performing the composition directly in (ϵ, δ) world (even with advanced composition).

As a result, the ideas behind Rényi differential privacy have been used to greatly improve the privacy cost accounting in a number of recent iterative algorithms, including Google's differentially private version of Tensorflow.

Finally, like other variants of differential privacy, RDP provides a post-processing property.

Zero-Concentrated Differential Privacy

In concurrent work released in 2016, Mark Bun and Thomas Steinke proposed zero-concentrated differential privacy (zCDP). Like RDP, zCDP is defined in terms of the Rényi divergence, but it includes only a single privacy parameter (ρ). A randomized mechanism F satisfies ρ-zCDP if for all neighboring datasets x and x', and all $\alpha \in (1, \infty)$:

\begin{align} D_\alpha (F(x) \Vert F(x')) \leq \rho\alpha \end{align}

This is a stronger requirement than RDP, because it restricts the Rényi divergence of many orders; however, the bound becomes more relaxed as α grows. Like RDP, zCDP can be converted to (ϵ, δ)-differential privacy: if F satisfies ρ-zCDP, then for $\delta > 0$, F satisfies (ϵ, δ)-differential privacy for $\epsilon = \rho + 2\sqrt{\rho \log(1/\delta)}$.

zCDP is also similar to RDP in that the Gaussian mechanism can be used as a basic mechanism. Specifically, for a function $f : \mathcal{D} \to \mathbb{R}^k$ with $L2$ sensitivity Δf, the following mechanism satisfies ρ-zCDP:

\begin{align} F(x) = f(x) + \mathcal{N}(\sigma^2) \text{ where } \sigma^2 = \frac{\Delta f^2}{2\rho} \end{align}

As with RDP, this mechanism is easy to implement:

```
def gaussian_mech_zCDP_vec(vec, sensitivity, rho):
    sigma = np.sqrt((sensitivity**2) / (2 * rho))
    return [v + np.random.normal(loc=0, scale=sigma) for v in vec]
```

In another similarity with RDP, zCDP's sequential composition is also asymptotically tight for repeated applications of the Gaussian mechanism. It's also very simple: the ρs add up. Specifically:

Sequential composition:

- If F_1 satisfies ρ_1-zCDP
- And F_2 satisfies ρ_2-zCDP
- Then their composition satisfies $\rho_1 + \rho_2$-zCDP

Finally, zCDP also provides a post-processing property.

Truncated Concentrated Differential Privacy

Coming soon

Composition under Variants of Differential Privacy

Which variant should we use, and when?

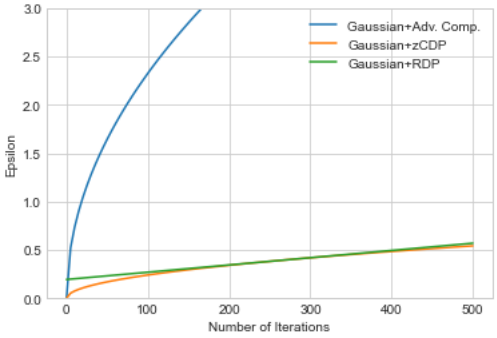
The recently-developed variants will yield *significantly* tighter bounds on privacy cost when:

- The Gaussian mechanism is used (especially on high-dimensional vectors)
- The algorithm in question applies the mechanism many times (e.g. hundreds or thousands of times)

To use RDP and zCDP, we typically implement an algorithm in terms of the variant we want to use, and then convert the total privacy cost of running the algorithm back to (ϵ, δ)-differential privacy so that we can compare it to other algorithms.

To see the effect of this strategy, let's imagine an algorithm that applies the Gaussian mechanism k times. We'll fix values for σ (i.e. the amount of noise added with the Gaussian mechanism in each of the k iterations) and δ, and then compare the final ϵs achieved for each variant.

We'll see that composition under RDP and zCDP result in *smaller values of ϵ for the same amount of noise added*. The algorithm is identical under all variants (i.e. it adds the same amount of noise in each case) - so this means that RDP and zCDP are providing *significantly tighter bounds on privacy cost* for the *same algorithm*.



The first thing to note is that using sequential composition under either zCDP or RDP is *much* better than using advanced composition with (ϵ, δ)-differential privacy. When building iterative algorithms with the Gaussian mechanism, these variants should always be used.

The second thing to note is the difference between zCDP (in orange) and RDP (in green). The ϵ for RDP grows linearly in k, because we have fixed a value for α. The ϵ for zCDP is sublinear in k, since zCDP effectively considers many αs. The two lines touch at some value of k, depending on the α chosen for RDP (for $\alpha = 20$, they touch at roughly $k = 300$).

The practial effect of this difference is that α must be chosen carefully when using RDP in order to bound privacy cost as tightly as possible. This is usually easy to do, since algorithms are usually parameterized by α; as a result, we can simply test multiple values of α to see which one results in the smallest corresponding ϵ. Since this test is *independent* of the data (it depends mainly on the privacy parameters we pick, and the number of iterations we want to run), we can test as many values of α as we want without paying additional privacy cost. We only need to test a small range of values for α - typically in the range between 2 and 100 - to find a minimum. This is the approach taken in most practical implementations, including Google's privacy-preserving version of Tensorflow.

The Exponential Mechanism

Learning Objectives

After reading this chapter, you will be able to:

- Define, implement, and apply the Exponential and Report Noisy Max mechanisms
- Describe the challenges of applying the Exponential mechanism in practice
- Describe the advantages of these mechanisms

The fundamental mechanisms we have seen so far (Laplace and Gaussian) are focused on numerical answers, and add noise directly to the answer itself. What if we want to return a precise answer (i.e. no added noise), but still preserve differential privacy? One solution is the exponential mechanism, which allows selecting the "best" element from a set while preserving differential privacy. The analyst defines which element is the "best" by specifying a *scoring function* that outputs a score for each element in the set, and also defines the set of things to pick from. The mechanism provides differential privacy by *approximately* maximizing the score of the element it returns - in other words, to satisfy differential privacy, the exponential mechanism sometimes returns an element from the set which does *not* have the highest score.

The exponential mechanism satisfies ϵ-differential privacy:

1. The analyst selects a set $\mathcal{R}$ of possible outputs
2. The analyst specifies a scoring function $u : \mathcal{D} \times \mathcal{R} \rightarrow \mathbb{R}$ with global sensitivity Δu
3. The exponential mechanism outputs $r \in \mathcal{R}$ with probability proportional to:

\begin{align} \exp \Big(\frac{\epsilon u(x, r)}{2 \Delta u} \Big) \end{align}

The biggest practical difference between the exponential mechanism and the previous mechanisms we've seen (e.g. the Laplace mechanism) is that the output of the exponential mechanism is *always* a member of the set $\mathcal{R}$. This is extremely useful when selecting an item from a finite set, when a noisy answer would not make sense. For example, we might want to pick a date for a big meeting, which uses each participant's personal calendar to maximize the number of participants without a conflict, while providing differential privacy for the calendars. Adding noise to a date doesn't make much sense: it might turn a Friday into a Saturday, and *increase* the number of conflicts significantly. The exponential mechanism is perfect for problems like this one: it selects a date *without noise*.

The exponential mechanism is interesting for several reasons:

- The privacy cost of the mechanism is just ϵ, regardless of the size of $\mathcal{R}$ - more on this next.
- It works for both finite and infinite sets $\mathcal{R}$, but it can be really challenging to build a practical implementation which samples from the appropriate probability distribution when $\mathcal{R}$ is infinite.
- It represents a "fundamental mechanism" of ϵ-differential privacy: all other ϵ-differentially private mechanisms can be defined in terms of the exponential mechanism with the appropriate definition of the scoring function u.

The Exponential Mechanism for Finite Sets

```
options = adult['Marital Status'].unique()

def score(data, option):
    return data.value_counts()[option]/1000

score(adult['Marital Status'], 'Never-married')
```

```
10.683
```

```
def exponential(x, R, u, sensitivity, epsilon):
    # Calculate the score for each element of R
    scores = [u(x, r) for r in R]

    # Calculate the probability for each element, based on its score
    probabilities = [np.exp(epsilon * score / (2 * sensitivity)) for score in scores]

    # Normalize the probabilties so they sum to 1
    probabilities = probabilities / np.linalg.norm(probabilities, ord=1)

    # Choose an element from R based on the probabilities
    return np.random.choice(R, 1, p=probabilities)[0]

exponential(adult['Marital Status'], options, score, 1, 1)
```

```
'Married-civ-spouse'
```

```
r = [exponential(adult['Marital Status'], options, score, 1, 1) for i in range(200)]
pd.Series(r).value_counts()
```

```
Married-civ-spouse    177
Never-married          23
dtype: int64
```

Report Noisy Max

Can we recover the exponential mechanism using the Laplace mechanism? In the case of a finite set $\mathcal{R}$, the basic idea of the exponential mechanism - to select from a set with differential privacy - suggests a naive implementation in terms of the Laplace mechanism:

1. For each $r \in \mathcal{R}$, calculate a *noisy score* $u(x, r) + \text{Lap}(\frac{\Delta u}{\epsilon})$
2. Output the element $r \in \mathcal{R}$ with the maximum noisy score

Since the scoring function u is Δu sensitive in x, each "query" in step 1 satisfies ϵ-differential privacy. Thus if $\mathcal{R}$ contains n elements, the above algorithm satisfies $n\epsilon$-differential privacy by sequential composition.

However, if we used the exponential mechanism, the total cost would be just ϵ instead! Why is the exponential mechanism so much better? Because *it releases less information*.

Our analysis of the Laplace-based approach defined above is very pessimistic. The whole set of noisy scores computed in step 1 actually satisfies $n\epsilon$-differential privacy, and we could release the whole thing. That the output in step 2 satisfies $n\epsilon$-differential privacy follows from the post-processing property.

But the exponential mechanism releases *only* the identity of the element with the maximum noisy score - *not* the score itself, or the scores of any other element.

The algorithm defined above is often called the *report noisy max* algorithm, and it actually satisfies ϵ-differential privacy, no matter how large the set $\mathcal{R}$ is - specifically because it releases *only* the identity of the element with the largest noisy count. The proof can be found in Dwork and Roth, Claim 3.9.

Report noisy max is easy to implement, and it's easy to see that it produces very similar results to our earlier implementation of the exponential mechanism for finite sets.

```
def report_noisy_max(x, R, u, sensitivity, epsilon):
    # Calculate the score for each element of R
    scores = [u(x, r) for r in R]

    # Add noise to each score
    noisy_scores = [laplace_mech(score, sensitivity, epsilon) for score in scores]

    # Find the index of the maximum score
    max_idx = np.argmax(noisy_scores)

    # Return the element corresponding to that index
    return R[max_idx]

report_noisy_max(adult['Marital Status'], options, score, 1, 1)
```

```
'Married-civ-spouse'
```

```
r = [report_noisy_max(adult['Marital Status'], options, score, 1, 1) for i in
range(200)]
pd.Series(r).value_counts()
```

```
Married-civ-spouse    194
Never-married           6
dtype: int64
```

So the exponential mechanism can be replaced with report noisy max when the set $\mathcal{R}$ is finite, but what about when it's infinite? We can't easily add Laplace noise to an infinite set of scores. In this context, we have to use the actual exponential mechanism.

In practice, however, using the exponential mechanism for infinite sets is often challenging or impossible. While it's easy to write down the probability density function defined by the mechanism, it's often the case that no efficient algorithm exists for sampling from it. As a result, numerous theoretical papers appeal to the exponential mechanism to show that a differentially private algorithm "exists" with certain desirable properties, but many of these algorithms are impossible to use in practice.

The Exponential Mechanism as Fundamental Mechanism for ϵ-Differential Privacy

We've seen that it's not possible to recover the exponential mechanism using the Laplace mechanism plus sequential composition, because we can't capture the fact that the algorithm we designed doesn't release all of the noisy scores. What about the reverse - can we recover the Laplace mechanism from the exponential mechanism? It turns out that we can!

Consider a query $q(x) : \mathcal{D} \rightarrow \mathbb{R}$ with sensitivity Δq. We can release an ϵ-differentially private answer by adding Laplace noise: $F(x) = q(x) + \text{Lap}(\Delta q/\epsilon)$. The probability density function for this differentially private version of q is:

\begin{align} \mathsf{Pr}[F(x) = r] =& \frac{1}{2b} \exp\Big(- \frac{\lvert r - \mu \rvert}{b}\Big)\ =& \frac{\epsilon}{2 \Delta q} \exp\Big(- \frac{\epsilon \lvert r - q(x) \rvert}{\Delta q}\Big) \end{align}

Consider what happens when we set the scoring function for the exponential mechanism to $u(x, r) = -2|q(x) - r|$. The exponential mechanism says that we should sample from the probability distribution proportional to:

\begin{align} \mathsf{Pr}[F(x) = r] =&; \exp \Big(\frac{\epsilon u(x, r)}{2 \Delta u} \Big)\ &= \exp \Big(\frac{\epsilon (-2 \lvert q(x) - r \rvert)}{2 \Delta q} \Big)\ &= \exp \Big(- \frac{\epsilon \lvert r - q(x) \rvert}{\Delta q} \Big)\ \end{align}

So it's possible to recover the Laplace mechanism from the exponential mechanism, and we get the same results (up to constant factors - the general analysis for the exponential mechanism is not tight in all cases).

The exponential mechanism is extremely general - it's generally possible to re-define any ϵ-differentially private mechanism in terms of a carefully chosen definition of the scoring function u. If we can analyze the sensitivity of this scoring function, then the proof of differential privacy comes for free.

On the other hand, applying the general analysis of the exponential mechanism sometimes comes at the cost of looser bounds (as in the Laplace example above), and mechanisms defined in terms of the exponential mechanism are often very difficult to implement. The exponential mechanism is often used to prove theoretical lower bounds (by showing that a differentially private algorithm *exists*), but practical algorithms often replicate the same behavior using some other approach (as in the case of report noisy max above).

The Sparse Vector Technique

> **Learning Objectives**
>
> After reading this chapter, you will be able to:
>
> - Describe the Sparse Vector Technique and the reasons to use it
> - Define and implement Above Threshold
> - Apply the Sparse Vector Technique in iterative algorithms

We've already seen one example of a mechanism - the exponential mechanism - which achieves a lower-than-expected privacy cost by withholding some information. Are there others?

There are, and one that turns out to be extremely useful in practical algorithms is the *sparse vector technique* (SVT). The sparse vector technique operates on a stream of sensitivity-1 queries over a dataset; it releases the *identity* of the first query in the stream which passes a test, and nothing else. The advantage of SVT is that it incurs a fixed total privacy cost, no matter *how many queries it considers*.

Above Threshold

The most basic instantiation of the sparse vector technique is an algorithm called `AboveThreshold` (see Dwork and Roth, Algorithm 1). The inputs to the algorithm are a stream of sensitivity-1 queries, a dataset D, a *threshold* T, and the privacy parameter ϵ; the algorithm preserves ϵ-differential privacy. A Python implementation of the algorithm appears below.

```
# preserves epsilon-differential privacy
def above_threshold(queries, df, T, epsilon):
    T_hat = T + np.random.laplace(loc=0, scale = 2/epsilon)

    for idx, q in enumerate(queries):
        nu_i = np.random.laplace(loc=0, scale = 4/epsilon)
        if q(df) + nu_i >= T_hat:
            return idx
    return -1 # the index of the last element
```

The `AboveThreshold` algorithm returns (approximately) the index of the first query in `queries` whose result exceeds the threshold. The algorithm preserves differential privacy by sometimes returning the *wrong* index; sometimes, the index returned may be for a query whose result does *not* exceed the threshold, and sometimes, the index may not be the *first* whose query result exceeds the threshold.

The algorithm works by generating a *noisy threshold* `T_hat`, then comparing noisy query answers (`q(i) + nu_i`) against the noisy threshold. The algorithm returns the index of the first comparison that succeeds.

It's a little bit surprising that the privacy cost of this algorithm is just ϵ, because it may compute the answers to *many* queries. In particular, a naive version of this algorithm might compute noisy answers to all of the queries first, then select the index of the first one whose value is above the threshold:

```
# preserves |queries|*epsilon-differential privacy
def naive_above_threshold(queries, df, T, epsilon):
    for idx, q in enumerate(queries):
        nu_i = np.random.laplace(loc=0, scale = 1/epsilon)
        if q(df) + nu_i >= T:
            return idx
    return None
```

For a list of queries of length n, this version preserves $n\epsilon$-differential privacy by sequential composition.

Why does `AboveThreshold` do so much better? As we saw with the exponential mechanism, sequential composition would allow `AboveThreshold` to release *more information* than it actually does. In particular, our naive version of the algorithm could release the indices of *every* query exceeding the threshold (not just the first one), *plus* the noisy query answers themselves, and it would still preserve $n\epsilon$-differential privacy. The fact that `AboveThreshold` withholds all this information allows for a tighter analysis of privacy cost.

Applying the Sparse Vector Technique

The sparse vector technique is extremely useful when we want to run many different queries, but we only care about the answer for one of them (or a small subset of them). In fact, this application gives the technique its name: it's most useful when the *vector* of queries is *sparse* - i.e. most of the answers don't exceed the threshold.

We've already seen a perfect example of such a scenario: selecting a clipping bound for summation queries. Earlier, we took an approach like the naive version of `AboveThreshold` defined above: compute noisy answers under many different clipping bounds, then select the lowest one for which the answer doesn't change much.

We can do much better with the sparse vector technique. Consider a query which clips the ages of everyone in the dataset, then sums them up:

```
def age_sum_query(df, b):
    return df['Age'].clip(lower=0, upper=b).sum()

age_sum_query(adult, 30)
```

```
913809
```

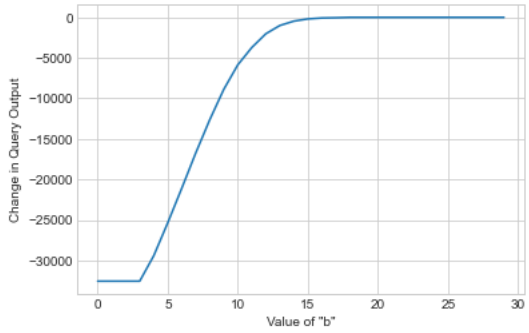
The naive algorithm for selecting a good value for `b` is to obtain differentially private answers for many values of `b`, returning the smallest one where the value stops increasing:

```
def naive_select_b(query, df, epsilon):
    bs = range(1, 1000, 10)
    best = 0
    threshold = 10
    epsilon_i = epsilon / len(bs)

    for b in bs:
        r = laplace_mech(query(df, b), b, epsilon_i)

        # if the new answer is pretty close to the old answer, stop
        if r - best <= threshold:
            return b
        # otherwise update the "best" answer to be the current one
        else:
            best = r

    return bs[-1]

naive_select_b(age_sum_query, adult, 1)
```

```
91
```

Can we use SVT here? We only care about one thing: the value of `b` where the value of `age_sum_query(df, b)` stops increasing. However, the sensitivity of `age_sum_query(df, b)` is `b`, because adding or removing a row in `df` could change the sum by at most `b`; to use SVT, we need to build a stream of 1-sensitive queries.

The value we actually care about, though, is whether or not the query's answer is *changing* at a specific value of b (i.e. `age_sum_query(df, b) - age_sum_query(df, b + 1)`). Consider what happens when we add a row to `df`: the answer to the first query goes up by b, but the answer to the second query *also* goes up - by $b + 1$. The sensitivity is therefore $|b - (b + 1)| = 1$ - so each query will be 1-sensitive, as desired! As the value of b approaches the optimal one, the value of the difference we defined above will approach 0:

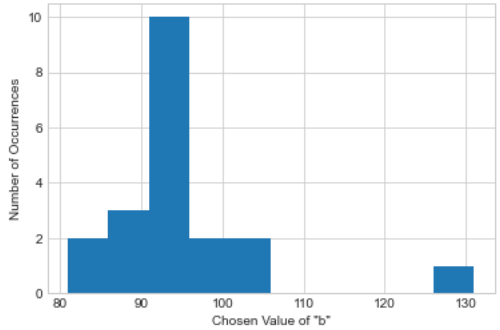


Let's define a stream of difference queries, and use `AboveThreshold` to determine the index of the best value of `b` using the sparse vector technique.

```
def create_query(b):
    return lambda df: age_sum_query(df, b) - age_sum_query(df, b + 1)

bs = range(1,150,5)
queries = [create_query(b) for b in bs]
epsilon = .1

bs[above_threshold(queries, adult, 0, epsilon)]
```

```
91
```

Note that it *doesn't matter* how long the list `bs` is - we'll get accurate results (and pay the same privacy cost) no matter its length. The really powerful effect of SVT is to eliminate the dependence of privacy cost on the number of queries we perform. Try changing the range for `bs` above and re-running the plot below. You'll see that the output doesn't depend on the number of values for `b` we try - even if the list has *thousands* of elements!



We can use SVT to build an algorithm for summation queries (and using this, for average queries) that automatically computes the clipping parameter.

```
def auto_avg(df, epsilon):
    def create_query(b):
        return lambda df: df.clip(lower=0, upper=b).sum() - df.clip(lower=0,
upper=b+1).sum()

    # Construct the stream of queries
    bs = range(1,150000,5)
    queries = [create_query(b) for b in bs]

    # Run AboveThreshold, using 1/3 of the privacy budget, to find a good clipping
parameter
    epsilon_svt = epsilon / 3
    final_b = bs[above_threshold(queries, df, 0, epsilon_svt)]

    # Compute the noisy sum and noisy count, using 1/3 of the privacy budget for each
    epsilon_sum = epsilon / 3
    epsilon_count = epsilon / 3

    noisy_sum = laplace_mech(df.clip(lower=0, upper=final_b).sum(), final_b,
epsilon_sum)
    noisy_count = laplace_mech(len(df), 1, epsilon_count)

    return noisy_sum/noisy_count

auto_avg(adult['Age'], 1)
```

```
38.571558786811565
```

This algorithm invokes three differentially private mechanisms: `AboveThreshold` once, and the Laplace mechanism twice, each with $\frac{1}{3}$ of the privacy budget. By sequential composition, it satisfies ϵ-differential privacy. Because we are free to test a really wide range of possible values for `b`, we're able to use the same `auto_avg` function for data on many different scales! For example, we can also use it on the capital gain column, even though it has a very different scale than the age column.

```
auto_avg(adult['Capital Gain'], 1)
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-10-5b72967cbeb8> in <module>
----> 1 auto_avg(adult['Capital Gain'], 1)

<ipython-input-9-d75c5d3477ec> in auto_avg(df, epsilon)
      9     # Run AboveThreshold, using 1/3 of the privacy budget, to find a good
clipping parameter
     10     epsilon_svt = epsilon / 3
---> 11     final_b = bs[above_threshold(queries, df, 0, epsilon_svt)]
     12
     13     # Compute the noisy sum and noisy count, using 1/3 of the privacy budget
for each

<ipython-input-2-bb0d36153548> in above_threshold(queries, df, T, epsilon)
      5     for idx, q in enumerate(queries):
      6         nu_i = np.random.laplace(loc=0, scale = 4/epsilon)
----> 7         if q(df) + nu_i >= T_hat:
      8             return idx
      9     return -1 # the index of the last element

<ipython-input-9-d75c5d3477ec> in <lambda>(df)
      1 def auto_avg(df, epsilon):
      2     def create_query(b):
----> 3         return lambda df: df.clip(lower=0, upper=b).sum() - df.clip(lower=0,
upper=b+1).sum()
      4
      5     # Construct the stream of queries

/usr/local/lib/python3.9/site-packages/pandas/core/generic.py in clip(self, lower,
upper, axis, inplace, *args, **kwargs)
   7350             upper is None or (is_scalar(upper) and is_number(upper))
   7351         ):
-> 7352             return self._clip_with_scalar(lower, upper, inplace=inplace)
   7353
   7354         result = self

/usr/local/lib/python3.9/site-packages/pandas/core/generic.py in
_clip_with_scalar(self, lower, upper, inplace)
   7207             if lower is not None:
   7208                 subset = self.to_numpy() >= lower
-> 7209                 result = result.where(subset, lower, axis=None,
inplace=False)
   7210
   7211         if np.any(mask):

/usr/local/lib/python3.9/site-packages/pandas/core/generic.py in where(self, cond,
other, inplace, axis, level, errors, try_cast)
   9002         """
   9003         other = com.apply_if_callable(other, self)
-> 9004         return self._where(
   9005             cond, other, inplace, axis, level, errors=errors,
try_cast=try_cast
   9006         )

/usr/local/lib/python3.9/site-packages/pandas/core/generic.py in _where(self, cond,
other, inplace, axis, level, errors, try_cast)
   8867                 axis=block_axis,
   8868             )
-> 8869             result = self._constructor(new_data)
   8870             return result.__finalize__(self)
   8871

/usr/local/lib/python3.9/site-packages/pandas/core/series.py in __init__(self, data,
index, dtype, name, copy, fastpath)
    210         ):
    211             # GH#33357 called with just the SingleBlockManager
--> 212             NDFrame.__init__(self, data)
    213             self.name = name
    214             return

/usr/local/lib/python3.9/site-packages/pandas/core/generic.py in __init__(self, data,
copy, attrs)
    203         object.__setattr__(self, "_is_copy", None)
    204         object.__setattr__(self, "_mgr", data)
--> 205         object.__setattr__(self, "_item_cache", {})
    206         if attrs is None:
    207             attrs = {}

KeyboardInterrupt:
```

Note that this takes a long time to run! That's because we have to try a lot more values for `b` before finding a good one, since the capital gain column has a much larger scale. We can reduce this cost by increasing the step size (5, in our implementation above) or by constructing `bs` with an exponential scale.

Returning Multiple Values

In the above application, we only needed the index of the *first* query which exceeded the threshold, but in many other applications we would like to find the indices of *all* such queries.

We can use SVT to do this, but we'll have to pay a higher privacy cost. We can implement an algorithm called `Sparse` (see Dwork and Roth, Algorithm 2) to accomplish the task, using a very simple approach:

1. Start with a stream $qs = \{q_1, \ldots, q_k\}$ of queries
2. Run `AboveThreshold` on qs to learn the index i of the first query which exceeds the threshold
3. Restart the algorithm (go to (1)) with $qs = \{q_{i+1}, \ldots, q_k\}$ (i.e. the *remaining* queries)

If the algorithm invokes `AboveThreshold` n times, with a privacy parameter of ϵ for each invocation, then it satisfies $n\epsilon$-differential privacy by sequential composition. If we want to specify an upper bound on total privacy cost, we need to bound n - so the `Sparse` algorithm asks the analyst to specify an upper bound c on the number of times `AboveThreshold` will be invoked.

```
def sparse(queries, df, c, T, epsilon):
    idxs = []
    pos = 0
    epsilon_i = epsilon / c

    # stop if we reach the end of the stream of queries, or if we find c queries above
the threshold
    while pos < len(queries) and len(idxs) < c:
        # run AboveThreshold to find the next query above the threshold
        next_idx = above_threshold(queries[pos:], df, T, epsilon_i)

        # if AboveThreshold reaches the end, return
        if next_idx == -1:
            return idxs

        # otherwise, update pos to point to the rest of the queries
        pos = next_idx+pos
        # update the indices to return to include the index found by AboveThreshold
        idxs.append(pos)
        # and move to the next query in the stream
        pos = pos + 1

    return idxs
```

```
epsilon = 1
sparse(queries, adult, 3, 0, epsilon)
```

```
[19, 22, 23]
```

By sequential composition, the `sparse` algorithm satisfies ϵ-differential privacy (it uses $\epsilon_i = \frac{\epsilon}{c}$ for each invocation of `AboveThreshold`). The version described in Dwork and Roth uses advanced composition, setting the ϵ value for each invocation of `AboveThreshold` so that the total privacy cost is ϵ (zCDP or RDP could also be used to perform the composition).

Application: Range Queries

A *range query* asks: "how many rows exist in the dataset whose values lie in the range (a, b)?" Range queries are counting queries, so they have sensitivity 1; we can't use parallel composition on a set of range queries, however, since the rows they examine might overlap.

Consider a set of range queries over ages (i.e. queries of the form "how many people have ages between a and b?"). We can generate many such queries at random:

```
def age_range_query(df, lower, upper):
    df1 = df[df['Age'] > lower]
    return len(df1[df1['Age'] < upper])

def create_age_range_query():
    lower = np.random.randint(30, 50)
    upper = np.random.randint(lower, 70)
    return lambda df: age_range_query(df, lower, upper)


range_queries = [create_age_range_query() for i in range(10)]
results = [q(adult) for q in range_queries]
results
```

```
[6163, 1602, 3967, 3745, 14271, 898, 0, 808, 6657, 17655]
```

The answers to such range queries vary widely - some ranges create tiny (or even empty) groups, with small counts, while others create large groups with high counts. In many cases, we know that the small groups will have inaccurate answers under differential privacy, so there's not much point in even running the query. What we'd like to do is learn which queries are worth answering, and then pay privacy cost for *just* those queries.

We can use the sparse vector technique to do this. First, we'll determine the indices of the range queries in the stream which exceed a threshold for "goodness" that we decide on. Then, we'll use the Laplace mechanism to find differentially private answers for *just* those queries. The total privacy cost will be proportional to the number of queries *above* the threshold - not the total number of queries. In cases where we expect just a few queries to be above the threshold, this can result in a much smaller privacy cost.

```
def range_query_svt(queries, df, c, T, epsilon):
    # first, run Sparse to get the indices of the "good" queries
    sparse_epsilon = epsilon / 2
    indices = sparse(queries, adult, c, T, sparse_epsilon)

    # then, run the Laplace mechanism on each "good" query
    laplace_epsilon = epsilon / (2*c)
    results = [laplace_mech(queries[i](df), 1, laplace_epsilon) for i in indices]
    return results
```

```
range_query_svt(range_queries, adult, 5, 10000, 1)
```

```
[14272.137097167237, 17697.903735723932]
```

Using this algorithm, we pay half of the privacy budget to determine the first c queries which lie above the threshold of 10000, then the other half of the budget to obtain noisy answers to *just* those queries. If the number of queries exceeding the threshold is tiny compared to the total number, then we're able to obtain much more accurate answers using this approach.

Exercises in Algorithm Design

Issues to Consider

- How many queries are required, and what kind of composition can we use?
    - Is parallel composition possible?
    - Should we use sequential composition, advanced composition, or a variant of differential privacy?
- Can we use the sparse vector technique?
- Can we use the exponential mechanism?
- How should we distribute the privacy budget?
- If there are unbounded sensitivities, how can we bound them?
- Would synthetic data help?
- Would post-processing to "de-noise" help?

1. Generalized Sample and Aggregate

Design a variant of sample and aggregate which does *not* require the analyst to specify the output range of the query function f.

Ideas: use SVT to find good upper and lower bounds on $f(x)$ for the whole dataset first. The result of $clip(f(x), lower, upper)$ has bounded sensitivity, so we can use this query with SVT. Then use sample and aggregate with these upper and lower bounds.

2. Summary Statistics

Design an algorithm to produce differentially private versions of the following statistics:

- Mean: $\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$
- Variance: $var = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$
- Standard deviation: $\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$

Ideas:

Mean

1. Use SVT to find upper and lower clipping bounds
2. Compute noisy sum and count, and derive mean by post-processing

Variance

1. Split it into a count query ($\frac{1}{n}$ - we have the answer from above) and a sum query
2. What's the sensitivity of $\sum_{i=1}^{n}(x_i - \mu)^2$? It's b^2; we can clip and compute $\sum_{i=1}^{n}(x_i - \mu)^2$, then multiply by (1) by post processing

Standard Deviation

1. Just take the square root of variance

Total queries:

- Lower clipping bound (SVT)
- Upper clipping bound (SVT)
- Noisy sum (mean)
- Noisy count
- Noisy sum (variance)

3. Heavy Hitters

Google's RAPPOR system is designed to find the most popular settings for Chrome's home page. Design an algorithm which:

- Given a list of the 10,000 most popular web pages by traffic,
- Determines the top 10 most-popular home pages out of the 10,000 most popular web pages

Ideas: Use parallel composition and take the noisy top 10

4. Hierarchical Queries

Design an algorithm to produce summary statistics for the U.S. Census. Your algorithm should produce total population counts at the following levels:

- Census tract
- City / town
- ZIP Code
- County
- State
- USA

Ideas:

Idea 1: *Only* compute the bottom level (census tract), using parallel composition. Add up all the tract counts to get the city counts, and so on up the hierarchy. Advantage: lowers privacy budget.

Idea 2: Compute counts at all levels, using parallel composition for each level. Tune the budget split using real data; probably we need more accuracy for the smaller levels of the hierarchy.

Idea 3: As (2), but also use post-processing to re-scale lower levels of the hierarchy based on higher ones; truncate counts to whole numbers; move negative counts to 0.

5. Workloads of Range Queries

Design an algorithm to accurately answer a workload of *range queries*. Range queries are queries on a single table of the form: "how many rows have a value for c between a and b?" (i.e. the count of rows which lie in a specific range).

Part 1

The whole workload is pre-specified as a finite sequence of ranges: $\{(a_1, b_1), \ldots, (a_k, b_k)\}$, and

Part 2

The length of the workload k is pre-specified, but queries arrive in a streaming fashion and must be answered as they arrive.

Part 3

The workload may be infinite.

Ideas:

Just run each query with sequential composition.

For part 1, combine them so we can use L2 sensitivity. When k is large, this will work well with Gaussian noise.

Or, build synthetic data:

- For each range $(i, i+1)$, find a count (parallel composition). This is a synthetic data representation! We can answer infinitely many queries by adding up the counts of all the segments in this histogram which are contained in the desired interval.
- For part 2, use SVT

For SVT: for each query in the stream, ask how far the real answer is from the synthetic data answer. If it's far, query the real answer's range (as a histogram, using parallel composition) and update the synthetic data. Otherwise just give the synthetic data answer. This way you *ONLY* pay for updates to the synthetic data.

Machine Learning

> **Learning Objectives**
>
> After reading this chapter, you will be able to:
>
> - Describe and implement the basic algorithm for gradient descent
> - Use the Gaussian mechanism to implement differentially private gradient descent
> - Clip gradients to enforce differential privacy for arbitrary loss functions
> - Describe the effect of noise on the training process

In this chapter, we're going to explore building differentially private machine learning classifiers. We'll focus on a kind of *supervised learning* problem: given a set of *labeled training examples* $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, in which x_i is called the *feature vector* and y_i is called the *label*, train a *model* θ which can *predict* the label for a new feature vector which was not present in the training set. Each x_i is typically a vector of real numbers which describe the features of a training example, and the y_is are drawn from a predefined set of *classes* (usually expressed as integers) that examples can be drawn from. A *binary* classifier has two classes (usually either 1 and 0, or 1 and -1).

Logistic Regression with Scikit-Learn

To train a model, we will use some of the data we have available to build a set of training examples (as described earlier), but we'll also set aside some of the data as *test examples*. Once we have trained the model, we want to know how well it works on examples that are *not* present in the training set. A model which works well on new examples it hasn't seen before is said to *generalize* well. One which does *not* generalize well is said to have *overfitted* the training data.

To test generalization, we'll use the test examples - we have labels for them, so we can test the generalization accuracy of the model by asking the model to classify each one, and then comparing the predicted class against the actual label from our dataset. We'll split our data into a training set containing 80% of the examples, and a testing set containing 20% of the examples.

A simple way to build a binary classifier is with *logistic regression*. The scikit-learn library has a built-in module for performing logistic regression, called `LogisticRegression`, and it's easy to use to build a model using our data.

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression().fit(X_train,y_train)
model
```

```
/usr/local/lib/python3.9/site-packages/sklearn/linear_model/_logistic.py:763:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

```
LogisticRegression()
```

Next, we can use the model's `predict` method to predict labels for the test set.

```
model.predict(X_test)
```

```
array([-1., -1., -1., ..., -1., -1., -1.])
```

So, how many test examples does our model get correct? We can compare the predicted labels against the actual labels from the dataset; if we divide the number of correctly predicted labels by the total number of test examples, we can measure the percent of the examples which are correctly classified.

```
np.sum(model.predict(X_test) == y_test)/X_test.shape[0]
```

```
0.8447589562140646
```

Our model predicts the correct label for 84% of the examples in our test set. For this dataset, that's a pretty decent result.

What is a Model?

What exactly *is* a model? How does it encode the information it uses to make predictions?

There are many different kinds of models, but the ones we'll explore here are *linear models*. For an unlabeled example with a k-dimensional feature vector $x_1, \ldots, x_k$, a linear model predicts a label by first calculating the quantity:

\begin{align} w_1 x_1 + \dots + w_k x_k + bias \end{align}

and then taking the sign of it (i.e. if the quantity above is negative, we predict the label -1; if it's positive, we predict 1).

The model itself, then, can be represented by a vector containing the values $w_1, \ldots, w_k$ and the value for $bias$. The model is said to be linear because the quantity we calculate in predicting a label is a polynomial of degree 1 (i.e. linear). The values $w_1, \ldots, w_k$ are often called the *weights* or *coefficients* of the model, and $bias$ is often called the *bias term* or *intercept*.

This is actually how scikit-learn represents its logistic regression model, too! We can check out the weights of our trained model using the `coef_` attribute of the model:

```
model.intercept_[0], model.coef_[0]
```

Note that we'll always have exactly the same number of weights w_i as we have features x_i, since we have to multiply each feature by its corresponding weight. That means our model has exactly the same dimensionality as our feature vectors.

Now that we have a way to get the weights and bias term, we can implement our own function to perform prediction:

```
# Prediction: take a model (theta) and a single example (xi) and return its predicted
label
def predict(xi, theta, bias=0):
    label = np.sign(xi @ theta + bias)
    return label

np.sum(predict(X_test, model.coef_[0], model.intercept_[0]) == y_test)/X_test.shape[0]
```

```
0.8447589562140646
```

We've made the bias term optional here, because in many cases it's possible to do just as well without it. To make things simpler, we won't bother to train a bias term in our own algorithm.

Training a Model with Gradient Descent

How does the training process actually work? The scikit-learn library has some pretty sophisticated algorithms, but we can do just about as well by implementing a simple one called *gradient descent*.

Most training algorithms for machine learning are defined in terms of a *loss function*, which specifies a way to measure how "bad" a model is at prediction. The goal of the training algorithm is to minimize the output of the loss function - a model with low loss will be *good* at prediction.

The machine learning community has developed many different commonly-used loss functions. A simple loss function might return 0 for each correctly predicted example, and 1 for each incorrectly predicted example; when the loss becomes 0, that means we've predicted each example's label correctly. A more commonly used loss function for binary classification is called the *logistic loss*; the logistic loss gives us a measure of "how far" we are from predicting the correct label (which is more informative than the simple 0 vs 1 approach).

The logistic loss is implemented by the following Python function:

```
# The loss function measures how good our model is. The training goal is to minimize
the loss.
# This is the logistic loss function.
def loss(theta, xi, yi):
    exponent = - yi * (xi.dot(theta))
    return np.log(1 + np.exp(exponent))
```

We can use the loss function to measure how good a particular model is. Let's try it out with a model whose weights are all zeros. This model isn't likely to work very well, but it's a starting point from which we can train a better one.

```
theta = np.zeros(X_train.shape[1])
loss(theta, X_train[0], y_train[0])
```

```
0.6931471805599453
```

We typically measure how good our model is over our entire training set by simply averaging the loss over all of the examples in the training data. In this case, we get *every* example wrong, so the average loss on the whole training set is exactly equal to the loss we calculated above for just one example.

```
np.mean([loss(theta, x_i, y_i) for x_i, y_i in zip(X_train, y_train)])
```

```
0.6931471805599453
```

Our goal in *training* the model is to *minimize* the loss. So the key question is: how do we modify the model to make the loss smaller?

Gradient descent is an approach that makes the loss smaller by updating the model according to the *gradient* of the loss. The gradient is like a multi-dimensional derivative: for a function with multi-dimensional inputs (like our loss function above), the gradient tells you how fast the function's output is changing with respect to *each* dimension of the input. If the gradient is positive in a particular dimension, that means the function's value will *increase* if we increase the model's weight for that dimension; we want the loss to *decrease*, so we should modify our model by *negating* the gradient - i.e. do the *opposite* of what the gradient says. Since we move the model in the opposite direction of the gradient, this is called *descending* the gradient.

When we iteratively perform many steps of this descent process, we slowly get closer and closer to the model which minimizes the loss. This algorithm is called *gradient descent*. Let's see how this looks in Python; first, we'll define the gradient function.

```
# This is the gradient of the logistic loss
# The gradient is a vector that indicates the rate of change of the loss in each
direction
def gradient(theta, xi, yi):
    exponent = yi * (xi.dot(theta))
    return - (yi*xi) / (1+np.exp(exponent))
```

A Single Step of Gradient Descent

Next, let's perform a single step of gradient descent. We can apply the `gradient` function to a single example from our training data, which should give us enough information to improve the model for that example. We "descend" the gradient by subtracting it from our current model `theta`.

```
# If we take a step in the *opposite* direction from the gradient (by negating it), we
should
# move theta in a direction that makes the loss *lower*
# This is one step of gradient descent - in each step, we're trying to "descend" the
gradient
# In this example, we're taking the gradient on just a single training example (the
first one)
theta = theta - gradient(theta, X_train[0], y_train[0])
theta
```

Now, if we call `predict` on the same example from the training data, its label is predicted correctly! That means our update did indeed improve the model, since it's now capable of classifying this example.

```
y_train[0], predict(theta, X_train[0])
```

```
(-1.0, -1.0)
```

We'll be measuring the accuracy of our model many times, so let's define a helper function for measuring accuracy. It works in the same way as the accuracy measurement for the sklearn model above. We can use it on the `theta` we've built by descending the gradient for one example, to see how good our model is on the test set.

```
def accuracy(theta):
    return np.sum(predict(X_test, theta) == y_test)/X_test.shape[0]

accuracy(theta)
```

```
0.7585139318885449
```

Our improved model now predicts 75% of the labels for the test set correctly! That's good progress - we've improved the model considerably.

A Gradient Descent Algorithm

We need to make two changes to arrive at a basic gradient descent algorithm. First, our single step above used only a single example from the training data; we want to consider the *whole* training set when updating the model, so that we improve the model for *all* examples. Second, we need to perform multiple iterations, to get as close as possible to

minimizing the loss.

We can solve the first problem by calculating the *average gradient* over all of the training examples, and using it for the descent step in place of the single-example gradient we used before. Our `avg_grad` function calculates the average gradient over a whole array of training examples and the corresponding labels.

```
def avg_grad(theta, X, y):
    grads = [gradient(theta, xi, yi) for xi, yi in zip(X, y)]
    return np.mean(grads, axis=0)

avg_grad(theta, X_train, y_train)
```

To solve the second problem, we'll define an iterative algorithm that descends the gradient multiple times.

```
def gradient_descent(iterations):
    # Start by "guessing" what the model should be (all zeros)
    theta = np.zeros(X_train.shape[1])

    # Perform `iterations` steps of gradient descent using training data
    for i in range(iterations):
        theta = theta - avg_grad(theta, X_train, y_train)

    return theta
```

```
theta = gradient_descent(10)
accuracy(theta)
```

```
0.7787483414418399
```

After 10 iterations, our model reaches nearly 78% accuracy - not bad! Our gradient descent algorithm looks simple (and it is!) but don't let its simplicity fool you - this basic approach is behind many of the recent successes in large-scale deep learning, and our algorithm is *very* close in its design to the ones implemented in popular frameworks for machine learning like Tensorflow.

Notice that we didn't quite make it to the 84% accuracy of the sklearn model we trained earlier. Don't worry - our algorithm is definitely capable of this! We just need more iterations, to get closer to the minimum of the loss.

With 100 iterations, we get closer - 82% accuracy. However, the algorithm takes a long time to run when we ask for so many iterations. Even worse, the closer we get to minimizing the loss, the more difficult it is to improve - so we might get to 82% accuracy after 100 iterations, but it might take 1000 iterations to get to 84%. This points to a fundamental tension in machine learning - generally speaking, more iterations of training can improve accuracy, but more iterations requires more computation time. Most of the "tricks" used to make large-scale deep learning practical are actually aimed at speeding up each iteration of gradient descent, so that more iterations can be performed in the same amount of time.

One more thing that's interesting to note: the value of the loss function does indeed go down with each iteration of gradient descent we perform - so as we perform more iterations, we slowly get closer to minimizing the loss. Also note that the training and testing loss are very close to one another, suggesting that our model is not *overfitting* to the training data.

Gradient Descent with Differential Privacy

How can we make the above algorithm differentially private? We'd like to design an algorithm that ensures differential privacy for the training data, so that the final model doesnt reveal anything about individual training examples.

The only part of the algorithm which uses the training data is the gradient calculation. One way to make the algorithm differentially private is to add noise to the gradient itself at each iteration before updating the model. This approach is usually called *noisy gradient descent*, since we add noise directly to the gradient.

Our gradient function is a vector valued function, so we can use `gaussian_mech_vec` to add noise to its output:

```
def noisy_gradient_descent(iterations, epsilon, delta):
    theta = np.zeros(X_train.shape[1])
    sensitivity = '???'

    for i in range(iterations):
        grad = avg_grad(theta, X_train, y_train)
        noisy_grad = gaussian_mech_vec(grad, sensitivity, epsilon, delta)
        theta = theta - noisy_grad

    return theta
```

There's just one piece of the puzzle missing - **what is the sensitivity of the gradient function**? Answering this question is the central difficulty in making the algorithm work.

There are two major challenges here. First, the gradient is the result of an *average query* - it's the mean of many per-example gradients. As we've seen previously, it's best to split queries like this up into a sum query and a count query. This isn't difficult to do - we can compute the sum of the per-example gradients, rather than their average, and divide by a noisy count later. Second, we need to bound the sensitivity of each per-example gradient. There are two basic approaches for this: we can either analyze the gradient function itself (as we have done with previous queries) to determine its worst-case global sensitivity, or we can *enforce* a sensitivity by clipping the output of the gradient function (as we did in sample and aggregate).

We'll start with the second approach - often called *gradient clipping* - because it's simpler conceptually and more general in its applications.

Gradient Clipping

Recall that when we implemented sample and aggregate, we enforced a desired sensitivity on a function f with unknown sensitivity by clipping its output. The sensitivity of f was:

\begin{align} \lvert f(x) - f(x') \rvert \end{align}

After clipping with parameter b, this becomes:

\begin{align} \lvert \mathsf{clip}(f(x), b) - \mathsf{clip}(f(x'),b) \rvert \end{align}

In the worst case, $\mathsf{clip}(f(x), b) = b$, and $\mathsf{clip}(f(x'), b) = 0$, so the sensitivity of the clipped result is exactly b (the value of the clipping parameter).

We can use the same trick to bound the L2 sensitivity of our gradient function. We'll need to define a function which "clips" a vector so that it has L2 norm within a desired range. We can accomplish this by *scaling* the vector: if we divide the vector elementwise by its L2 norm, then the resulting vector will have an L2 norm of 1. If we want to target a particular clipping parameter b, we can multiply the scaled vector by b to scale it back up to have L2 norm b. We want to avoid modifying vectors that already have L2 norm below b; in that case, we just return the original vector. We can use `np.linalg.norm` with the parameter `ord=2` to calculate the L2 norm of a vector.

```
def L2_clip(v, b):
    norm = np.linalg.norm(v, ord=2)

    if norm > b:
        return b * (v / norm)
    else:
        return v
```

Now we're ready to analyze the sensitivity of the clipped gradient. We denote the gradient as $\nabla(\theta; X, y)$ (corresponding to `gradient` in our Python code):

\begin{align} \lVert \mathsf{L2_clip}(\nabla (\theta; X, y), b) - \mathsf{L2_clip}(\nabla (\theta; X', y)) \rVert_2 \end{align}

In the worst case, $\mathsf{L2\_clip}(\nabla(\theta; X, y), b)$ has L2 norm of b, and $\mathsf{L2\_clip}(\nabla(\theta; X', y))$ is all zeros - so that the L2 norm of the difference is equal to b. Thus, the L2 sensitivity of the clipped gradient is bounded by the clipping parameter b!

Now we can proceed to compute the sum of clipped gradients, and add noise based on the L2 sensitivity b that we've enforced by clipping.

```
def gradient_sum(theta, X, y, b):
    gradients = [L2_clip(gradient(theta, x_i, y_i), b) for x_i, y_i in zip(X,y)]

    # sum query
    # L2 sensitivity is b (by clipping performed above)
    return np.sum(gradients, axis=0)
```

Now we're ready to complete our noisy gradient descent algorithm. To compute the noisy average gradient, we need to:

1. Add noise to the sum of the gradients based on its sensitivity b
2. Compute a noisy count of the number of training examples (sensitivity 1)
3. Divide the noisy sum from (1) by the noisy count from (2)

```
def noisy_gradient_descent(iterations, epsilon, delta):
    theta = np.zeros(X_train.shape[1])
    sensitivity = 5.0

    noisy_count = laplace_mech(X_train.shape[0], 1, epsilon)

    for i in range(iterations):
        grad_sum        = gradient_sum(theta, X_train, y_train, sensitivity)
        noisy_grad_sum  = gaussian_mech_vec(grad_sum, sensitivity, epsilon, delta)
        noisy_avg_grad  = noisy_grad_sum / noisy_count
        theta           = theta - noisy_avg_grad

    return theta
```

```
theta = noisy_gradient_descent(10, 0.1, 1e-5)
accuracy(theta)
```

```
0.7810703228659885
```

Each iteration of this algorithm satisfies (ϵ, δ)-differential privacy, and we perform one additional query to determine the noisy count which satisfies ϵ-differential privacy. If we perform k iterations, then by sequential composition, the algorithm satisfies $(k\epsilon + \epsilon, k\delta)$-differential privacy. We can also use advanced composition to analyze the total privacy cost; even better, we could convert the algorithm to Rényi differential privacy or zero-concentrated differential privacy, and obtain tight bounds on composition.

Sensitivity of the Gradient

Our previous approach is very general, since it makes no assumptions about the behavior of the gradient. Sometimes, however, we *do* know something about the behavior of the gradient. In particular, a large class of useful gradient functions (including the gradient of the logistic loss, which we're using here) are *Lipschitz continuous* - meaning they have bounded global sensitivity. Formally, it is possible to show that:

\begin{align} \text{If}; \lVert x_i \rVert_2 \leq b; \text{then}; \lVert \nabla(\theta; x_i, y_i) \rVert_2 \leq b \end{align}

This fact allows us to clip the values of the *training examples* (i.e. the *inputs* to the gradient function), instead of the *output* of the gradient function, and obtain a bound on the L2 sensitivity of the gradient.

Clipping the training examples instead of the gradients has two advantages. First, it's often easier to estimate the scale of the training data (and thus to pick a good clipping parameter) than it is to estimate the scale of the gradients you'll compute during training. Second, it's computationally more efficient: we can clip the training examples *once*, and re-use the clipped training data every time we train a model; with gradient clipping, we need to clip each gradient during training. Furthermore, we're no longer forced to compute per-example gradients so that we can clip them; instead, we can compute all of the gradients at once, which can be done very efficiently (this is a commonly used trick in machine learning, but we won't discuss it here).

Note, however, that many useful loss functions - in particular, those derived from neural networks in deep learning - do *not* have bounded global sensitivity. For these loss functions, we're forced to use gradient clipping.

We can clip the training examples instead of the gradients with a couple of simple modifications to our algorithm. First, we clip the training examples using `L2_clip` before we start training. Second, we simply delete the code for clipping the gradients.

```
def gradient_sum(theta, X, y, b):
    gradients = [gradient(theta, x_i, y_i) for x_i, y_i in zip(X,y)]

    # sum query
    # L2 sensitivity is b (by sensitivity of the gradient)
    return np.sum(gradients, axis=0)
```

```
def noisy_gradient_descent(iterations, epsilon, delta):
    theta = np.zeros(X_train.shape[1])
    sensitivity = 5.0

    noisy_count = laplace_mech(X_train.shape[0], 1, epsilon)
    clipped_X = [L2_clip(x_i, sensitivity) for x_i in X_train]

    for i in range(iterations):
        grad_sum        = gradient_sum(theta, clipped_X, y_train, sensitivity)
        noisy_grad_sum  = gaussian_mech_vec(grad_sum, sensitivity, epsilon, delta)
        noisy_avg_grad  = noisy_grad_sum / noisy_count
        theta           = theta - noisy_avg_grad

    return theta
```

```
theta = noisy_gradient_descent(10, 0.1, 1e-5)
accuracy(theta)
```

```
0.7801857585139319
```

Many improvements to this algorithm are possible, which can improve privacy cost and accuracy. Many are drawn from the machine learning literature. Some examples include:

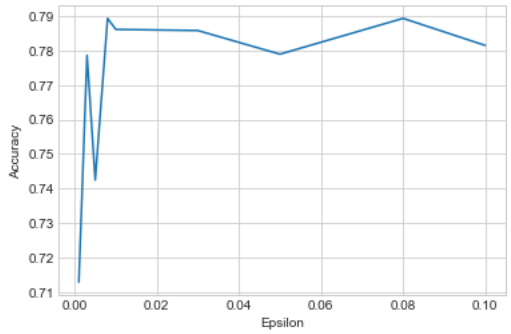
- Bounding the *total* privacy cost by ϵ by calculating a per-iteration ϵ_i as part of the algorithm.
- Better composition for large numbers of iterations via advanced composition, RDP, or zCDP.
- Minibatching: calculating the gradient for each iteration using a small chunk of the training data, rather than the whole training set (this reduces the computation needed to calculate the gradient).
- Parallel composition in conjunction with minibatching.
- Random sampling of batches in conjunction with minibatching.
- Other hyperparameters, like a learning rate η.

Effect of Noise on Training

So far, we've seen that the number of iterations has a big effect on the accuracy of the model we get, since more iterations can get you closer to the minimum of the loss. Since our differentially private algorithm adds noise to the gradient, this can also affect accuracy - the noise can cause our algorithm to move in *the wrong direction* during training, and actually make the model *worse*.

It's reasonable to expect that smaller values of ϵ will result in less accurate models (since this has been the trend in every differentially private algorithm we have seen so far). This is true, but there's also a slightly more subtle tradeoff which occurs because of the composition we need to consider when we perform many iterations of the algorithm: more iterations means a larger privacy cost. In the standard gradient descent algorithm, more iterations generally result in a better model. In our differentially private version, more iterations can make the model *worse*, since we have to use a smaller ϵ for each iteration, and so the scale of the noise goes up. In differentially private machine learning, it's important (and sometimes, very challenging) to strike the right balance between the number of iterations used and the scale of the noise added.

Let's do a small experiment to see how the setting of ϵ effects the accuracy of our model. We'll train a model for several values of ϵ, using 20 iterations each time, and graph the accuracy of each model against the ϵ value used in training it.



The plot shows that very small values of ϵ result in far less accurate models. Keep in mind that the ϵ we specify in the plot is a *per-iteration* ϵ, so the privacy cost is much higher after composition.

Local Differential Privacy

> **Learning Objectives**
>
> After reading this chapter, you will be able to:
>
> - Define the local model of differential privacy and contrast it with the central model
> - Define and implement the randomized response and unary encoding mechanisms
> - Describe the accuracy implications of these mechanisms and the challenges of the local model

So far, we have only considered the *central model* of differential privacy, in which the sensitive data is collected centrally in a single dataset. In this setting, we assume that the *analyst* is malicious, but that there is a *trusted data curator* who holds the dataset and correctly executes the differentially private mechanisms the analyst specifies.

This setting is often not realistic. In many cases, the data curator and the analyst are *the same*, and no trusted third party actually exists to hold the data and execute mechanisms. In fact, the organizations which collect the most sensitive data tend to be exactly the ones we *don't* trust; such organizations certainly can't function as trusted data curators.

An alternative to the central model of differential privacy is the *local model of differential privacy*, in which data is made differentially private before it leaves the control of the data subject. For example, you might add noise to your data *on your device* before sending it to the data curator. In the local model, the data curator does not need to be trusted, since the data they collect *already* satisfies differential privacy.

The local model thus has one huge advantage over the central model: data subjects don't need to trust anyone else but themselves. This advantage has made it popular in real-world deployments, including the ones by Google and Apple.

Unfortunately, the local model also has a significant drawback: the accuracy of query results in the local model is typically *orders of magnitude lower* for the same privacy cost as the same query under central differential privacy. This huge loss in accuracy means that only a small handful of query types are suitable for local differential privacy, and even for these, a large number of participants is required.

In this section, we'll see two mechanisms for local differential privacy. The first is called *randomized response*, and the second is called *unary encoding*.

Randomized Response

Randomized response is a mechanism for local differential privacy which was first proposed in a 1965 paper by S. L. Warner. At the time, the technique was intended to improve bias in survey responses about sensitive issues, and it was not originally proposed as a mechanism for differential privacy (which wouldn't be invented for another 40 years). After differential privacy was developed, statisticians realized that this existing technique *already* satisfied the definition.

Dwork and Roth present a variant of randomized response, in which the data subject answers a "yes" or "no" question as follows:

1. Flip a coin

2. If the coin is heads, answer the question truthfully
3. If the coin is tails, flip another coin
4. If the second coin is heads, answer "yes"; if it is tails, answer "no"

The randomization in this algorithm comes from the two coin flips. As in all other differentially private algorithms, this randomization creates uncertainty about the true answer, which is the source of privacy.

As it turns out, this randomized response algorithm satisfies ϵ-differential privacy for $\epsilon = \log(3) = 1.09$.

Let's implement the algorithm for a simple "yes" or "no" question: "is your occupation 'Sales'?" We can flip a coin in Python using `np.random.randint(0, 2)`; the result is either a 0 or a 1.

```
def rand_resp_sales(response):
    truthful_response = response == 'Sales'

    # first coin flip
    if np.random.randint(0, 2) == 0:
        # answer truthfully
        return truthful_response
    else:
        # answer randomly (second coin flip)
        return np.random.randint(0, 2) == 0
```

Let's ask 200 people who *do* work in sales to respond using randomized response, and look at the results.

```
pd.Series([rand_resp_sales('Sales') for i in range(200)]).value_counts()
```

```
True     144
False     56
dtype: int64
```

What we see is that we get both "yesses" and "nos" - but that the "yesses" outweigh the "nos." This output demonstrates both features of the differentially private algorithms we've already seen - it includes uncertainty, which creates privacy, but also displays enough signal to allow us to infer something about the population.

Let's try the same thing on some actual data. We'll take all of the occupations in the US Census data set we've been using, and encode responses for the question "is your occupation 'Sales'?" for each one. In an actual deployed system, we wouldn't collect this data set centrally at all - instead, each respondant would run `rand_resp_sales` locally, and submit their randomized response to the data curator. For our experiment, we'll run `rand_resp_sales` on the existing data set.

```
responses = [rand_resp_sales(r) for r in adult['Occupation']]
```

```
pd.Series(responses).value_counts()
```

```
False    22576
True      9985
dtype: int64
```

This time, we get many more "nos" than "yesses." This makes a lot of sense, with a little thought, because the majority of the participants in the data set are *not* in sales.

The key question now is: how do we estimate the *acutal* number of salespeople in the data set, based on these responses? The number of "yesses" is not a good estimate for the number of salespeople:

```
len(adult[adult['Occupation'] == 'Sales'])
```

```
3650
```

And this is not a surprise, since many of the "yesses" come from the random coin flips of the algorithm.

In order to get an estimate of the true number of salespeople, we need to analyze the randomness in the randomized response algorithm and estimate how many of the "yes" responses are from actual salespeople, and how many are "fake" yesses which resulted from random coin flips. We know that:

- With probability $\frac{1}{2}$, each respondant responds randomly

- With probability $\frac{1}{2}$, each random response is a "yes"

So, the probability that a respondant responds "yes" by random chance (rather than because they're a salesperson) is $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$. This means we can expect one-quarter of our *total* responses to be "fake yesses."

```
responses = [rand_resp_sales(r) for r in adult['Occupation']]

# we expect 1/4 of the responses to be "yes" based entirely on the coin flip
# these are "fake" yesses
fake_yesses = len(responses)/4

# the total number of yesses recorded
num_yesses = np.sum([1 if r else 0 for r in responses])

# the number of "real" yesses is the total number of yesses minus the fake yesses
true_yesses = num_yesses - fake_yesses
```

The other factor we need to consider is that half of the respondants answer randomly, but *some of the random respondants might actually be salespeople*. How many of them are salespeople? We have no data on that, since they answered randomly!

But, since we split the respondants into "truth" and "random" groups randomly (by the first coin flip), we can hope that there are roughly the same number of salespeople in both groups. Therefore, if we can estimate the number of salespeople in the "truth" group, we can double this number to get the number of salespeople in total.

```
# true_yesses estimates the total number of yesses in the "truth" group
# we estimate the total number of yesses for both groups by doubling
rr_result = true_yesses*2
rr_result
```

```
3791.5
```

How close is that to the true number of salespeople? Let's compare!

```
true_result = np.sum(adult['Occupation'] == 'Sales')
true_result
```

```
3650
```

```
pct_error(true_result, rr_result)
```

```
3.8767123287671232
```

With this approach, and fairly large counts (e.g. more than 3000, in this case), we generally get "acceptable" error - something below 5%. If your goal is to determine the most popular occupation, this approach is likely to work. However, when counts are smaller, the error will quickly get larger.

Furthermore, randomized response is *orders of magnitude* worse than the Laplace mechanism in the central model. Let's compare the two for this example:

```
pct_error(true_result, laplace_mech(true_result, 1, 1))
```

```
0.009618675924036936
```

Here, we get an error of about 0.01%, even though our ϵ value for the central model is slightly lower than the ϵ we used for randomized response.

There *are* better algorithms for the local model, but the inherent limitations of having to add noise before submitting your data mean that local model algorithms will *always* have worse accuracy than the best central model algorithms.

Unary Encoding

Randomized response allows us to ask a yes/no question with local differential privacy. What if we want to build a histogram?

A number of different algorithms for solving this problem in the local model of differential privacy have been proposed. A 2017 paper by Wang et al. provides a good summary of some optimal approaches. Here, we'll examine the simplest of these, called *unary encoding*. This approach is the basis for Google's RAPPOR system (with a number of modifications to make it work better for large domains and multiple responses over time).

The first step is to define the domain for responses - the labels of the histogram bins we care about. For our example, we want to know how many participants are associated with each occupation, so our domain is the set of occupations.

```
domain = adult['Occupation'].dropna().unique()
domain
```

```
array(['Adm-clerical', 'Exec-managerial', 'Handlers-cleaners',
       'Prof-specialty', 'Other-service', 'Sales', 'Craft-repair',
       'Transport-moving', 'Farming-fishing', 'Machine-op-inspct',
       'Tech-support', 'Protective-serv', 'Armed-Forces',
       'Priv-house-serv'], dtype=object)
```

We're going to define three functions, which together implement the unary encoding mechanism:

1. `encode`, which encodes the response
2. `perturb`, which perturbs the encoded response
3. `aggregate`, which reconstructs final results from the perturbed responses

The name of this technique comes from the encoding method used: for a domain of size k, each responses is encoded as a length-k vector of bits, with all positions 0 except the one corresponding to the occupation of the respondant. In machine learning, this representation is called a "one-hot encoding."

For example, 'Sales' is the 6th element of the domain, so the 'Sales' occupation is encoded with a vector whose 6th element is a 1.

```
def encode(response):
    return [1 if d == response else 0 for d in domain]

encode('Sales')
```

```
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
```

The next step is `perturb`, which flips bits in the response vector to ensure differential privacy. The probability that a bit gets flipped is based on two parameters p and q, which together determine the privacy parameter ϵ (based on a formula we will see in a moment).

$$\Pr[B'[i] = 1] = \begin{cases} p & \text{if } B[i] = 1 \\ q & \text{if } B[i] = 0 \end{cases}$$

```
def perturb(encoded_response):
    return [perturb_bit(b) for b in encoded_response]

def perturb_bit(bit):
    p = .75
    q = .25

    sample = np.random.random()
    if bit == 1:
        if sample <= p:
            return 1
        else:
            return 0
    elif bit == 0:
        if sample <= q:
            return 1
        else:
            return 0

perturb(encode('Sales'))
```

```
[0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1]
```

Based on the values of p and q, we can calculate the value of the privacy parameter ϵ. For $p = .75$ and $q = .25$, we will see an ϵ of slightly more than 2.

\begin{align} \epsilon = \log{\Big(\frac{p (1-q)}{(1-p) q}\Big)} \end{align}

```
def unary_epsilon(p, q):
    return np.log((p*(1-q)) / ((1-p)*q))

unary_epsilon(.75, .25)
```

```
2.1972245773362196
```

The final piece is aggregation. If we hadn't done any perturbation, then we could simply take the set of response vectors and add them element-wise to get counts for each element in the domain:

```
counts = np.sum([encode(r) for r in adult['Occupation']], axis=0)
list(zip(domain, counts))
```

```
[('Adm-clerical', 3770),
 ('Exec-managerial', 4066),
 ('Handlers-cleaners', 1370),
 ('Prof-specialty', 4140),
 ('Other-service', 3295),
 ('Sales', 3650),
 ('Craft-repair', 4099),
 ('Transport-moving', 1597),
 ('Farming-fishing', 994),
 ('Machine-op-inspct', 2002),
 ('Tech-support', 928),
 ('Protective-serv', 649),
 ('Armed-Forces', 9),
 ('Priv-house-serv', 149)]
```

But as we saw with randomized response, the "fake" responses caused by flipped bits cause the results to be difficult to interpret. If we perform the same procedure with the perturbed responses, the counts are all wrong:

```
counts = np.sum([perturb(encode(r)) for r in adult['Occupation']], axis=0)
list(zip(domain, counts))
```

```
[('Adm-clerical', 10073),
 ('Exec-managerial', 10197),
 ('Handlers-cleaners', 8891),
 ('Prof-specialty', 10209),
 ('Other-service', 9984),
 ('Sales', 9933),
 ('Craft-repair', 10160),
 ('Transport-moving', 8936),
 ('Farming-fishing', 8695),
 ('Machine-op-inspct', 9174),
 ('Tech-support', 8588),
 ('Protective-serv', 8529),
 ('Armed-Forces', 8190),
 ('Priv-house-serv', 8069)]
```

The aggregate step of the unary encoding algorithm takes into account the number of "fake" responses in each category, which is a function of both p and q, and the number of responses n:

\begin{align} A[i] = \frac{\sum_j B'_j[i] - n q}{p - q} \end{align}

```
def aggregate(responses):
    p = .75
    q = .25

    sums = np.sum(responses, axis=0)
    n = len(responses)

    return [(v - n*q) / (p-q) for v in sums]
```

```
responses = [perturb(encode(r)) for r in adult['Occupation']]
counts = aggregate(responses)
list(zip(domain, counts))
```

```
[('Adm-clerical', 3521.5),
 ('Exec-managerial', 4007.5),
 ('Handlers-cleaners', 1747.5),
 ('Prof-specialty', 4103.5),
 ('Other-service', 3129.5),
 ('Sales', 3457.5),
 ('Craft-repair', 3965.5),
 ('Transport-moving', 1781.5),
 ('Farming-fishing', 1025.5),
 ('Machine-op-inspct', 2157.5),
 ('Tech-support', 865.5),
 ('Protective-serv', 817.5),
 ('Armed-Forces', 13.5),
 ('Priv-house-serv', -128.5)]
```

As we saw with randomized response, these results are accurate enough to obtain a rough ordering of the domain elements (at least the most popular ones), but orders of magnitude less accurate than we could obtain with the Laplace mechanism in the central model of differential privacy.

Other methods have been proposed for performing histogram queries in the local model, including some detailed in the paper linked earlier. These can improve accuracy somewhat, but the fundamental limitations of having to ensure differential privacy for *each sample individually* in the local model mean that even the most complex technique can't match the accuracy of the mechanisms we've seen in the central model.

Synthetic Data

> **Learning Objectives**
>
> After reading this chapter, you will be able to:
>
> - Describe the idea of differentially private synthetic data and explain why it is useful
> - Define simple synthetic representations used in generating synthetic data
> - Define *marginals* and implement code to calculate them
> - Implement simple differentially private algorithms to generate low-dimensional synthetic data
> - Describe the challenges of generating high-dimensional synthetic data

In this section, we'll examine the problem of generating *synthetic data* using differentially private algorithms. Strictly speaking, the input of such an algorithm is an *original dataset*, and its output is a *synthetic dataset* with the same shape (i.e. same set of columns and same number of rows); in addition, we would like the *values* in the synthetic dataset to have the same properties as the corresponding values in the original dataset. For example, if we take our US Census dataset to be the original data, then we'd like our synthetic data to have a similar distribution of ages for the participants as the original data, and to preserve correlations between columns (e.g. a link between age and occupation).

Most algorithms for generating such synthetic data rely on a *synthetic representation* of the original dataset, which does *not* have the same shape as the original data, but which *does* allow answering queries about the original data. For example, if we *only* care about range queries over ages, then we could generate an age histogram - a count of how many participants in the original data had each possible age - and use the histogram to answer the queries. This histogram is a *synthetic representation* which is suitable for answering some queries, but it does not have the same shape as the original data, so it's not *synthetic data*.

Some algorithms simply use the synthetic representation to answer queries. Others use the synthetic representation to generate synthetic data. We'll look at one kind of synthetic representation - a histogram - and several methods of generating synthetic data from it.

Synthetic Representation: a Histogram

We've already seen many histograms - they're a staple of differentially private analyses, since parallel composition can be immediately applied. We've also seen the concept of a *range query*, though we haven't used that name very much. As a first step towards synthetic data, we're going to design a synthetic representation for one column of the original dataset which is capable of answering range queries.

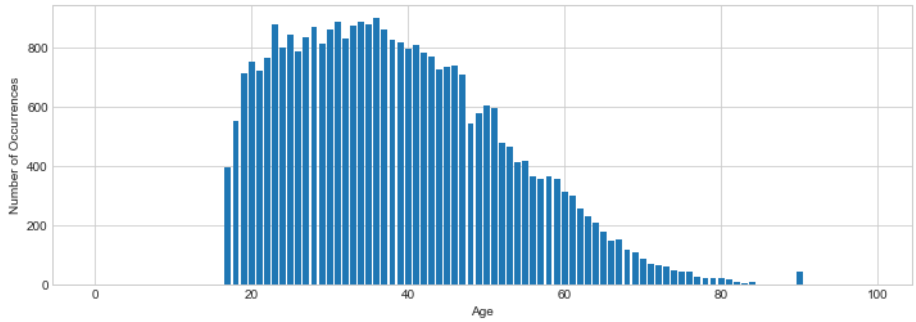
A *range query* counts the number of rows in the dataset which have a value lying in a given range. For example, "how many participants are between the ages of 21 and 33?" is a range query.

```
def range_query(df, col, a, b):
    return len(df[(df[col] >= a) & (adult[col] < b)])

range_query(adult, 'Age', 21, 33)
```

```
9878
```

We can define a histogram query which defines a histogram bin for each age between 0 and 100, and count the number of people in each bin using range queries. The result looks very much like the output of calling `plt.hist` on the data - because we've essentially computed the same result manually.

```
bins = list(range(0, 100))
counts = [range_query(adult, 'Age', b, b+1) for b in bins]
plt.xlabel('Age')
plt.ylabel('Number of Occurrences')
plt.bar(bins, counts);
```



We can use these histogram results as a synthetic representation for the original data! To answer a range query, we can add up all the counts of the bins which fall into the range.

```
def range_query_synth(syn_rep, a, b):
    total = 0
    for i in range(a, b):
        total += syn_rep[i]
    return total
```

```
range_query_synth(counts, 21, 33)
```

```
9878
```

Notice that we get *exactly* the same result, whether we issue the range query on the original data or our synthetic representation. We haven't lost any information from the original dataset (at least for the purposes of answering range queries over ages).

Adding Differential Privacy

We can easily make our synthetic representation differentially private. We can add Laplace noise separately to each count in the histogram; by parallel composition, this satisfies ϵ-differential privacy.

```
epsilon = 1
dp_syn_rep = [laplace_mech(c, 1, epsilon) for c in counts]
```

We can use the same function as before to answer range queries using our differentially private synthetic representation. By post-processing, these results also satisfy ϵ-differential privacy; furthermore, since we're relying on post-processing, we can answer as many queries as we want without incurring additional privacy cost.

```
range_query_synth(dp_syn_rep, 21, 33)
```

```
9874.13862308987
```

How accurate are the results? For small ranges, the results we get from our synthetic representation have very similar accuracy to the results we could get by applying the Laplace mechanism directly to the result of the range query we want to answer. For example:

```
Synthetic representation error: 0.05365687227828565
Laplace mechanism error: 0.029420674021466017
```

As the range gets bigger, the count gets larger, so we would expect error to improve. We have seen this over and over again - larger groups means a stronger signal, which leads to lower relative error. With the Laplace mechanism, we see exactly this behavior. With our synthetic representation, however, we're adding together noisy results from many *smaller* groups - so as the signal grows, so does the noise! As a result, we see roughly the same magnitude of relative error when using the synthetic representation, *regardless of the size of the range* - precisely the opposite of the Laplace mechanism!

```
Synthetic representation error: 0.010456588921896881
Laplace mechanism error: 0.007413220478904351
```

This difference demonstrates the drawback of our synthetic representation: it can answer any range query over the range it covers, but it might not offer the same accuracy as the Laplace mechanism. The major advantage of our synthetic representation is the ability to answer infinitely many queries without additional privacy budget; the major disadvantage is the loss in accuracy.

Generating Tabular Data

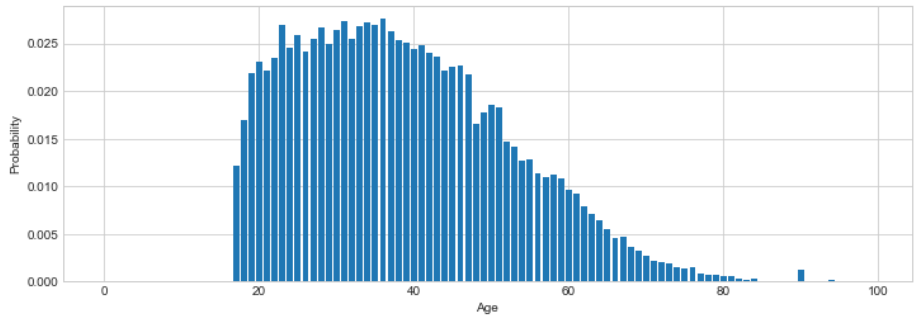
The next step is to go from our synthetic representation to *synthetic data*. To do this, we want to treat our synthetic representation as a probability distribution that estimates the underlying distribution from which the original data was drawn, and sample from it. Because we're considering just a single column, and ignoring all the others, this is called a *marginal distribution* (specifically, a *1-way marginal*).

Our strategy here will be simple: we have counts for each histogram bin; we'll normalize these counts so that they sum to 1, and then treat them as probabilities. Once we have these probabilities, we can sample from the distribution it represents by randomly selecting a bin of the histogram, weighted by the probabilities. Our first step is to prepare the counts, by ensuring that none is negative and by normalizing them to sum to 1:

```
dp_syn_rep_nn = np.clip(dp_syn_rep, 0, None)
syn_normalized = dp_syn_rep_nn / np.sum(dp_syn_rep_nn)
np.sum(syn_normalized)
```

```
1.0
```

Notice that if we plot the normalized counts - which we can now treat as probabilities for each corresponding histogram bin, since they sum to 1 - we see a shape that looks very much like the original histogram (which, in turn, looks a lot like the shape of the original data). This is all to be expected - except for their scale, these probabilities *are* simply the counts.

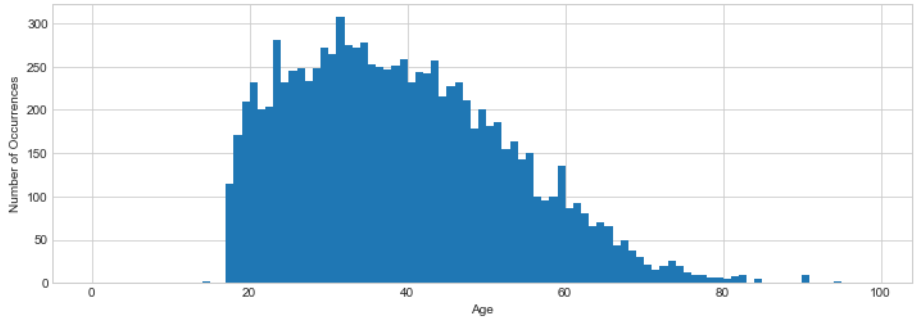


The final step is to generate new samples based on these probabilities. We can use `np.random.choice`, which allows passing in a list of probabilities (in the `p` parameter) associated with the choices given in the first parameter. It implements exactly the weighted random selection that we need for our sampling task. We can generate as many samples as we want without additional privacy cost, since we've already made our counts differentially private.

```
def gen_samples(n):
    return np.random.choice(bins, n, p=syn_normalized)

syn_data = pd.DataFrame(gen_samples(5), columns=['Age'])
syn_data
```

	Age
0	30
1	38
2	21
3	18
4	38

The samples we generate this way will be roughly distributed - we hope - according to the same underlying distribution as the original data. That means we can use the generated synthetic data to answer the same queries we could answer using the original data. In particular, if we plot the histogram of ages in a large synthetic data set, we'll see the same shape as we did in the original data.



We can also answer other queries we've seen in the past, like averages and range queries:

```
Mean age, synthetic: 38.8058
Mean age, true answer: 38.58164675532078
Percent error: 0.5776282016585534
```

```
Mean age, synthetic: 9139
Mean age, true answer: 29568
Percent error: 69.0915854978355
```

```
<ipython-input-2-bef15ba552d4>:2: UserWarning: Boolean Series key will be reindexed
to match DataFrame index.
  return len(df[(df[col] >= a) & (adult[col] < b)])
<ipython-input-2-bef15ba552d4>:2: UserWarning: Boolean Series key will be reindexed
to match DataFrame index.
  return len(df[(df[col] >= a) & (adult[col] < b)])
```

Our mean query has fairly low error (though still much larger than we would achieve by applying the Laplace mechanism directly). Our range query, however, has very large error! This is simply because we haven't quite matched the shape of the original data - we only generated 10,000 samples, and the original data set has more than 30,000 rows. We can perform an additional differentially private query to determine the number of rows in the original data, and then generate a new synthetic data set with the same number of rows, and this will improve our range query results.

```
Mean age, synthetic: 29671
Mean age, true answer: 29568
Percent error: 0.3483495670995671
```

```
<ipython-input-2-bef15ba552d4>:2: UserWarning: Boolean Series key will be reindexed
to match DataFrame index.
  return len(df[(df[col] >= a) & (adult[col] < b)])
<ipython-input-2-bef15ba552d4>:2: UserWarning: Boolean Series key will be reindexed
to match DataFrame index.
  return len(df[(df[col] >= a) & (adult[col] < b)])
```

Now we see the lower error we expect.

Generating More Columns

So far we've generated synthetic data that matches the number of rows of the original data set, and is useful for answering queries about the original data, but it has only a single column! How do we generate more columns?

There are two basic approaches. We could repeat the process we followed above for each of k columns (generating k 1-way marginals), and arrive at k separate synthetic data sets, each with a single column. Then, we could smash these data sets together to construct a single data set with k columns. This approach is straightforward, but since we consider each column in isolation, we'll lose correlations *between* columns that existed in the original data. For example, it might be the case that age and occupation are correlated in the data (e.g. managers are more likely to be old than they are to be young); if we consider each column in isolation, we'll get the *number* of 18-year-olds and the *number* of managers correct, but we may be very wrong about the number of 18-year-old managers.

The other approach is to consider multiple columns together. For example, we can consider both age and occupation at the same time, and count how many 18-year-old managers there are, how many 19-year-old managers there are, and so on. The result of this modified process is a 2-way marginal distribution. We'll end up considering all possible combinations of age and occupation - which is exactly what we did when we built contingency tables earlier! For example:

```
ct = pd.crosstab(adult['Age'], adult['Occupation'])
ct.head()
```

Now we can do exactly what we did before - add noise to these counts, then normalize them and treat them as probabilities! Each count now corresponds to a *pair* of values - both an age and an occupation - so when we sample from the distribution we have constructed, we'll get both values at once.

```
dp_ct = ct.applymap(lambda x: max(laplace_mech(x, 1, 1), 0))
dp_vals = dp_ct.stack().reset_index().values.tolist()
probs = [p for _,_,p in dp_vals]
vals = [(a,b) for a,b,_ in dp_vals]
probs_norm = probs / np.sum(probs)
list(zip(vals, probs_norm))[0]
```

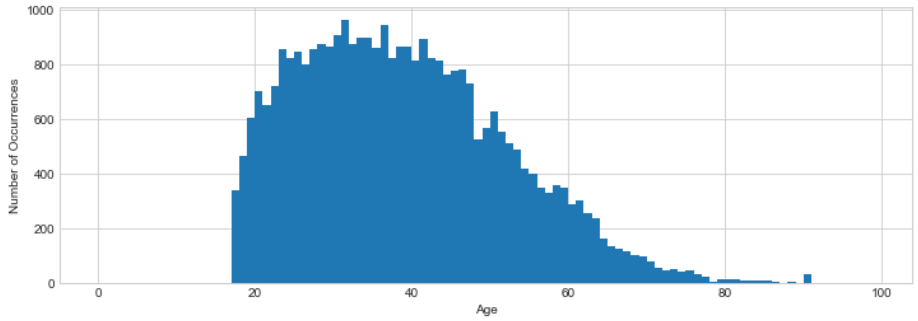
```
((17, 'Adm-clerical'), 0.0007989936755012214)
```

Examining the first element of the probabilities, we find that we'll have a 0.07% chance of generating a row representing a 17-year-old clerical worker. Now we're ready to generate some rows! We'll first generate a list of indices into the `vals` list, then generate rows by indexing into `vals`; we have to do this because `np.random.choice` won't accept a list of tuples in the first argument.

```
indices = range(0, len(vals))
n = laplace_mech(len(adult), 1, 1.0)
gen_indices = np.random.choice(indices, int(n), p=probs_norm)
syn_data = [vals[i] for i in gen_indices]

syn_df = pd.DataFrame(syn_data, columns=['Age', 'Occupation'])
syn_df.head()
```

	Age	Occupation
0	38	Handlers-cleaners
1	36	Sales
2	39	Prof-specialty
3	28	Tech-support
4	25	Other-service

The downside of considering two columns at once is that our accuracy will be lower. As we add more columns to the set we're considering (i.e., build an n-way marginal, with increasing values of n), we see the same effect we did with contingency tables - each count gets smaller, so the signal gets smaller relative to the noise, and our results are not as accurate. We can see this effect by plotting the histogram of ages in our new synthetic data set; notice that it has approximately the right shape, but it's less smooth than either the original data or the differentially private counts we used for the age column by itself.



We see the same loss in accuracy when we try specific queries on just the age column:

```
Percent error using synthetic data: 15.420908865160168
```

```
<ipython-input-2-bef15ba552d4>:2: UserWarning: Boolean Series key will be reindexed
to match DataFrame index.
  return len(df[(df[col] >= a) & (adult[col] < b)])
```

Summary

- A *synthetic representation* of a data set allows answering queries about the original data
- One common example of a synthetic representation is a histogram, which can be made differentially private by adding noise to its counts
- A histogram representation can be used to generate *synthetic data* with the same shape as the original data by treating its counts as probabilities: normalize the counts to sum to 1, then sample from the histogram bins using the corresponding normalized counts as probabilities
- The normalized histogram is a representation of a *1-way marginal distribution*, which captures the information in a single column in isolation
- A 1-way marginal does *not* capture correlations between columns
- To generate multiple columns, we can use multiple 1-way marginals, or we can construct a representation of a n-way marginal where $n > 1$
- Differentially private n-way marginals become increasingly noisy as n grows, since a larger n implies a smaller count for each bin of the resulting histogram
- The challenging tradeoff in generating synthetic data is thus:
    - Using multiple 1-way marginals loses correlation between columns
    - Using a single n-way marginal tends to be very inaccurate
- In many cases, generating synthetic data which is both accurate and captures the important correlations between columns is likely to be impossible

---

By Joseph P. Near