In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import pandas as pd
import numpy as np

adult = pd.read_csv("adult_with_pii.csv")
def laplace_mech(v, sensitivity, epsilon):
    return v + np.random.laplace(loc=0, scale=sensitivity / epsilon)
def pct_error(orig, priv):
    return np.abs(orig - priv)/orig * 100.0
```

Local Differential Privacy

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 (https://github.com/google/rappor) and Apple (https://www.apple.com/privacy/docs/Differential_Privacy_Overview.pdf).

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 (https://en.wikipedia.org/wiki/Randomized_response) is a mechanism for local differential privacy which was first proposed in a 1965 paper by S. L. Warner (https://www.jstor.org/stable/2283137?seq=1#metadata_info_tab_contents). 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.

In [184]:
```
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.

In [185]:
```
pd.Series([rand_resp_sales('Sales') for i in range(200)]).value_counts()
```

Out[185]:
```
True     155
False     45
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.

In [186]:
```
responses = [rand_resp_sales(r) for r in adult['Occupation']]
```

In [187]:
```
pd.Series(responses).value_counts()
```

Out[187]:
```
False    22633
True      9928
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:

In [188]:
```
len(adult[adult['Occupation'] == 'Sales'])
```

Out[188]:
```
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."

In [189]:
```
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 y
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.

In [190]:
```
# 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
```

Out[190]:
```
3721.5
```

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

In [192]:
```
true_result = np.sum(adult['Occupation'] == 'Sales')
true_result
```

Out[192]:
```
3650
```

In [193]:
```
pct_error(true_result, rr_result)
```

Out[193]:
```
1.9589041095890412
```

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:

In [169]:
```
pct_error(true_result, laplace_mech(true_result, 1, 1))
```

Out[169]:
```
0.011423124062500005
```

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. (https://arxiv.org/abs/1705.04421) 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 (https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42852.pdf) (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.

In [194]:
```
domain = adult['Occupation'].dropna().unique()
domain
```

Out[194]:
```
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-ser
v'], 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.

In [195]:
```
def encode(response):
    return [1 if d == response else 0 for d in domain]

encode('Sales')
```

Out[195]:
```
[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).

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

In [196]:
```
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'))
```

Out[196]:
```
[1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0]
```

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.

$$\epsilon = \log\left(\frac{p(1-q)}{(1-p)q}\right)$$

In [207]:
```
def unary_epsilon(p, q):
    return np.log((p*(1-q)) / ((1-p)*q))

unary_epsilon(.75, .25)
```

Out[207]:
```
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:

In [203]:
```
counts = np.sum([encode(r) for r in adult['Occupation']], axis=0)
list(zip(domain, counts))
```

Out[203]:
```
[('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:

In [208]:
```
counts = np.sum([perturb(encode(r)) for r in adult['Occupation']], axis=0)
list(zip(domain, counts))
```

Out[208]:
```
[('Adm-clerical', 10042),
 ('Exec-managerial', 10204),
 ('Handlers-cleaners', 9006),
 ('Prof-specialty', 10238),
 ('Other-service', 9635),
 ('Sales', 9844),
 ('Craft-repair', 10233),
 ('Transport-moving', 8863),
 ('Farming-fishing', 8721),
 ('Machine-op-inspct', 9122),
 ('Tech-support', 8753),
 ('Protective-serv', 8523),
 ('Armed-Forces', 8157),
 ('Priv-house-serv', 8042)]
```

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:

$$A[i] = \frac{\sum_j B'_j[i] - nq}{p - q}$$

In [205]:
```
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]
```

In [206]:
```
responses = [perturb(encode(r)) for r in adult['Occupation']]
counts = aggregate(responses)
list(zip(domain, counts))
```

Out[206]:
```
[('Adm-clerical', 3865.5),
 ('Exec-managerial', 4047.5),
 ('Handlers-cleaners', 989.5),
 ('Prof-specialty', 4001.5),
 ('Other-service', 2993.5),
 ('Sales', 3699.5),
 ('Craft-repair', 4093.5),
 ('Transport-moving', 1613.5),
 ('Farming-fishing', 715.5),
 ('Machine-op-inspct', 2119.5),
 ('Tech-support', 947.5),
 ('Protective-serv', 821.5),
 ('Armed-Forces', -92.5),
 ('Priv-house-serv', 387.5)]
```

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 (https://arxiv.org/abs/1705.04421) 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.