UNCERTAINTY

> by Dr Juan H Klopper

- Research Fellow
- School for Data Science and Computational Thinking
- Stellenbosch University

INTRODUCTION

One the most important aspects of Data Science is the ability to express uncertainty in our data and in our results.

The genration of random variables is not precise. Take a simple example such as height measurement. We only measure up to a set precision. If the measurement is done by hand, we cannot guarantee accuracy.

We also only work with sample of a population. Most often, there is a large difference in the population size and the sample size. We therefor don't approach the population parameters with our test statisics. There is uncertainty in our results.

In this notebook we learning to understand uncertainty and how to calculate and express the uncertainty in our results. This will be done by investigating the method of **bootstrapping**. Later we will learn how to calculate **confidence intervals**.

PACKAGES USED IN THIS NOTEBOOK

We see all the familiar, industry standard package imported below.

```
import numpy as np # Numerical analysis
import numpy as np # Numerical analysis
from scipy import stats # Statistical module
from pandas import DataFrame # Importing only the DataFrame function from pandas
```

```
# Data visualisation
import plotly.express as px
import plotly.graph_objects as go
import plotly.figure_factory as ff
import plotly.io as pio
```

```
# Setting a different plotting theme
pio.templates.default = 'ggplot2'
```

BOOTSTRAPPING

Bootstrapping is the technique of multiple resampling for our given sample. *New* samples are generated by drawning, at random, from the original sample set, with replacement.

In the code below, we generate random values for a variable in a population and then take a random sample from the population. Since we designed the population, we know the parameters of the variable. The variable is named `population` and the values are taken from a normal distribution with a mean of 100 and a standard deviation of 10.

```
np.random.seed(10)
population = stats.norm.rvs(
    loc=100, # Mean 100
    scale = 10, # Standard deviation 10
    size=20000 # Population size
)
```

We can calculate the exact mean and standard deviation of the variable in the population (both are parameters).

```
np.mean(population) # Mean parameter
```

```
99.99365915541331
```

```
np.std(population) # Standard deviation parameter
```

```
9.983005340618837
```

For our study, we randomly select 50 individuals from the population. Note the use of the `replace` argument and its value `False`. We do not want to select the same subject twice.

```
np.random.seed(1) # Reproducible results
np.random.seed(1) # Reproducible results
sample = np.random.choice(
    population,
    size=50,
    replace=False # Fifty seperate individuals
)
```

We calculate the mean and standard deviation of the sample (two statistics).

```
np.mean(sample) # Mean statistic
```

```
97.80136620289275
```

```
np.std(sample) # Standard deviation parameter
```

```
10.840027370000964
```

The statistics are not the same as the parameters.

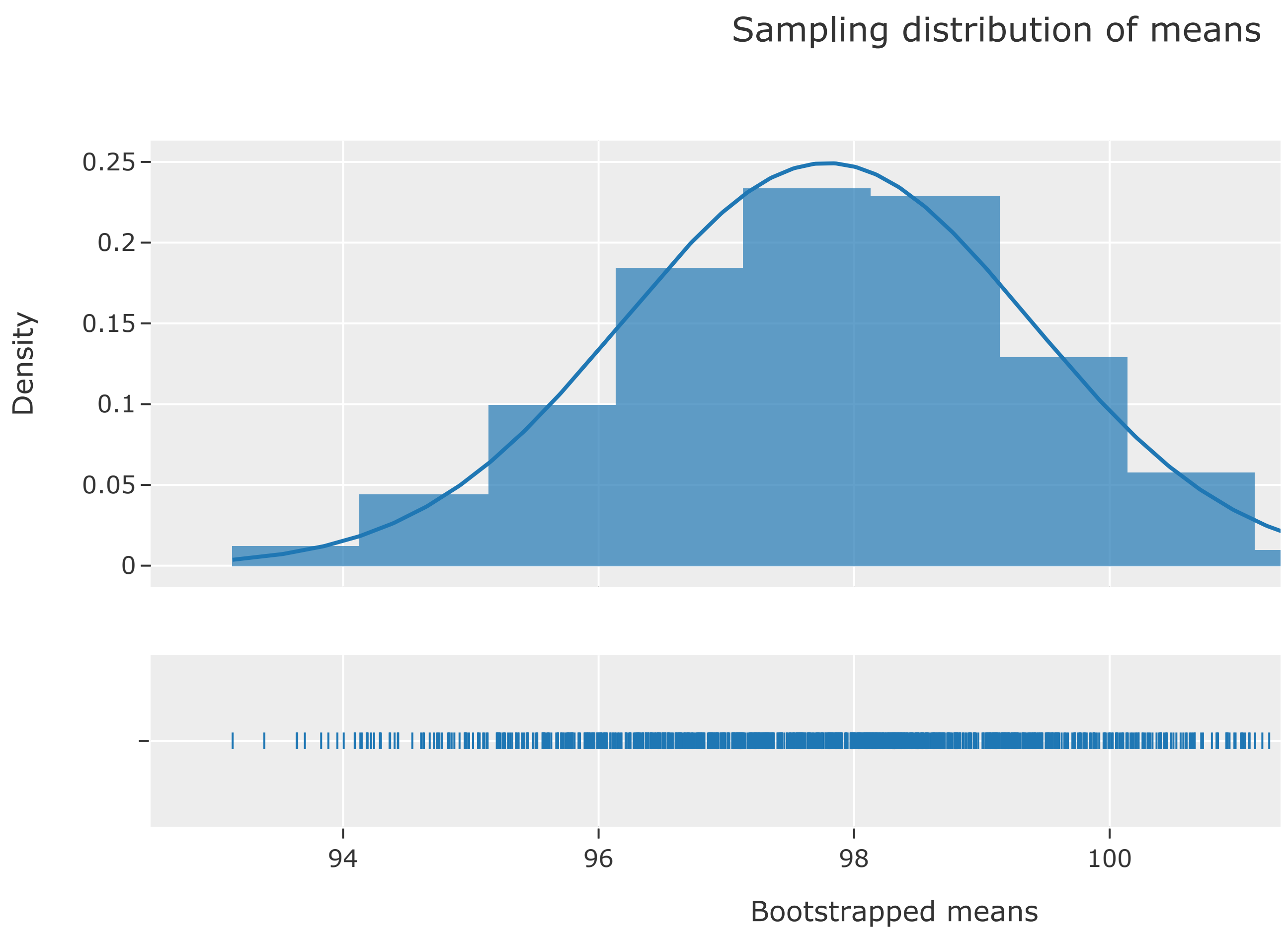
The process of boostrapping resamples multiple times from the sample and records the statistic each time. This gives us a distribution of the statistic. Each resample must have the same sample size as the original sample. This is done with replacement, else we would simply return the same sample each time. **Replacement** then simply refers to the fact that we return a subject for possible reselection every time. Since we want the same sample size in each new bootstrapped sample, we will have some subjects occur more than once in each new sample.

We use list comprehesnion below to build a list of 1000 resampled means. Note below that we set the `replace` argument to `True` and the sample size in the `choice` function is the same as the original sample.

```
means = [np.mean(np.random.choice(sample, 50, replace=True)) for i in range(100
```

The `create_displot` function from the figure_factory module of the plotly package provides us with a histogram and a *normal* curve based on the data. We note that the plot includes the population mean.

```
ff.create_distplot(
    [means],
    ['Resampled means'],
    curve_type='normal'
).update_layout(
    title='Sampling distribution of means',
    xaxis=dict(title='Bootstrapped means'),
    yaxis=dict({'title':'Density'})
)
```



Now we can use percentiles to consider the middle 95% of values. We ask what value would represent a percentile of 2.5% and what value would represent a percentile of 97.5%. We have to be careful when working with percentiles, though. We have an infinite number of percentile values (if we use decimal values as we do here). We do not have an inifinite number of means. There can also be ties (means with the same value). For this reason, we view the percentiles as the following steps. As as example, we use a percentile of 2.5% and a sample size of n.

1. Sort the collection in ascending order
2. Calculate k, which is 2.5% of n (shown in (1) below)
3. If k is a whole number then the k-th value in the ordered collection represents the 2.5% percentile.
4. Else, round k up to the nearest whole number and select that value from the ordered collection.

$$k = \frac{2.5}{100} n \tag{1}$$

We are interested in the values representing the 2.5% and the 97.5% percentiles. These are calculated below and assigned to appropriately named computer variables.

```python
k_2_5 = 2.5 / 100 * 1000
k_2_5
```

```
25.0
```

```python
k_97_5 = 97.5 / 100 * 1000
k_97_5
```

```
975.0
```

Since Python is 0-indexed, we want the 24-th and 974-th values in the ordered array of resampled means.

```python
sorted_means = np.sort(means)
```

```python
sorted_means[24]
```

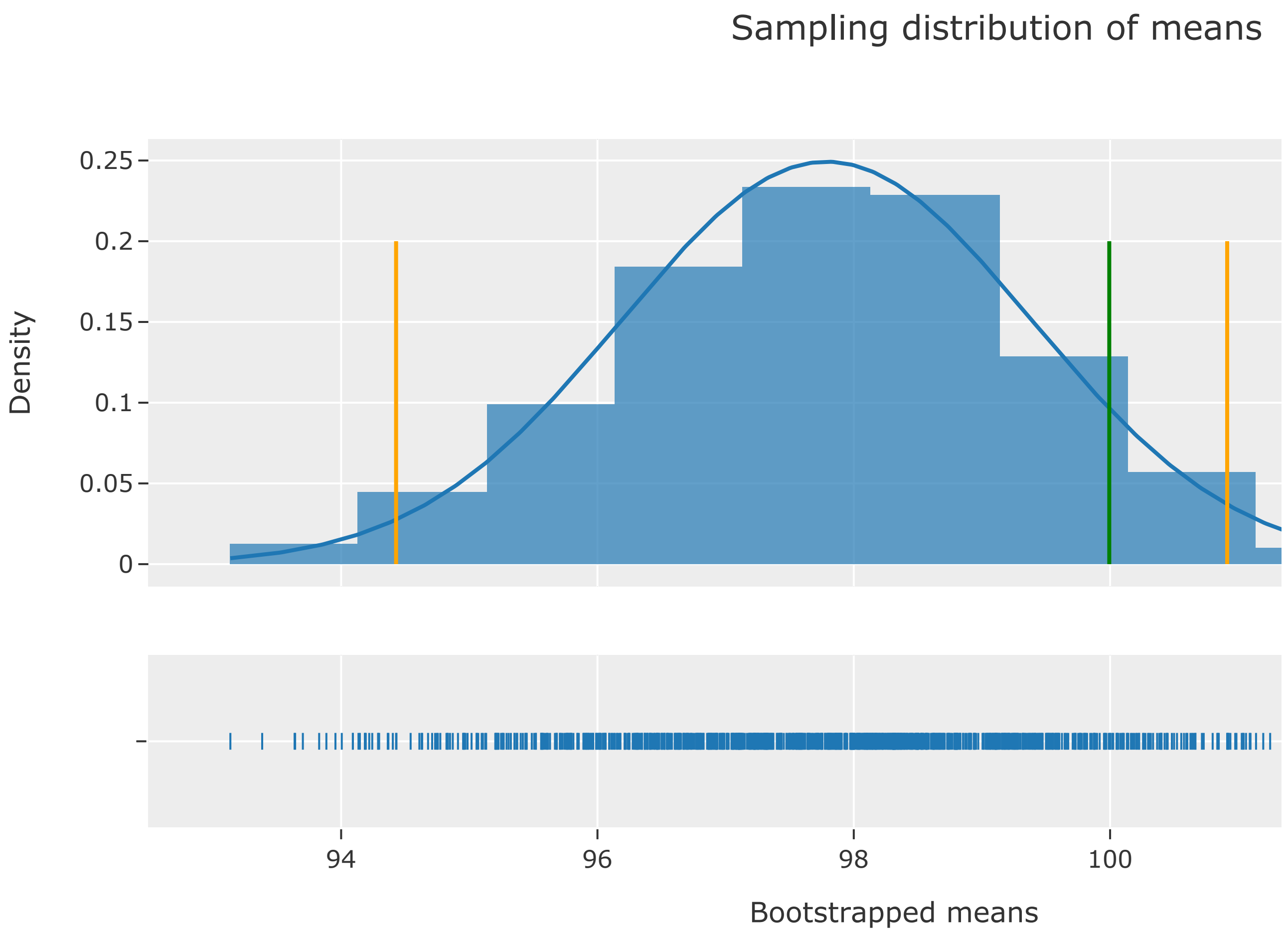
```
94.42864993563255
```

```python
sorted_means[974]
```

```
100.91367042455282
```

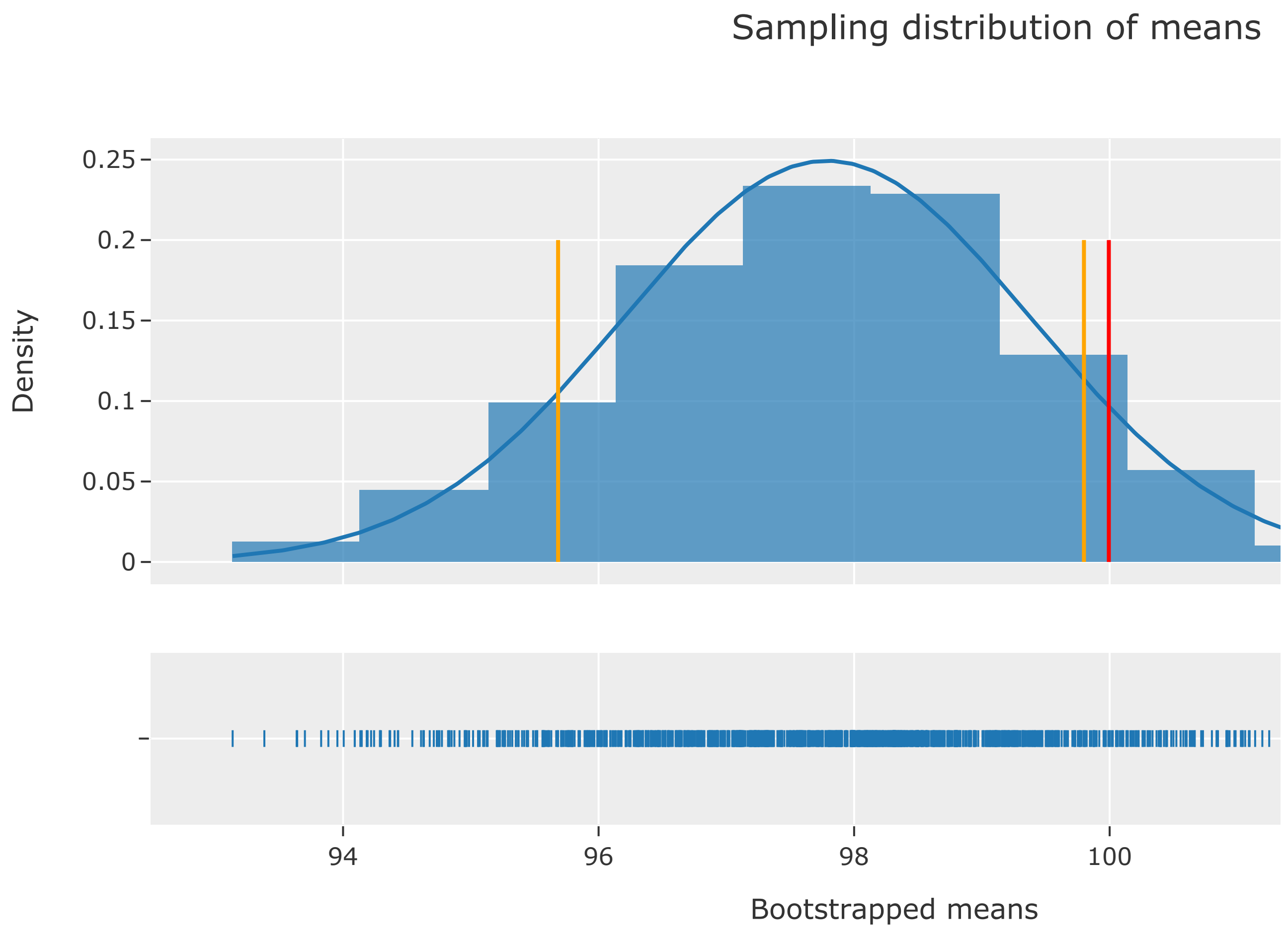
Below, we create another distribution plot. It indicates the percentile values and the population mean.

```python
ff.create_distplot(
    [means],
    ['Resampled means'],
    curve_type='normal'
).add_trace(
    go.Scatter(
        x=[sorted_means[24], sorted_means[24]],
        y=[0, 0.2],
        name='2.5%',
        mode='lines',
        marker=dict({'color':'orange'})
    )
).add_trace(
    go.Scatter(
        x=[sorted_means[974], sorted_means[974]],
        y=[0, 0.2],
        name='97.5%',
        mode='lines',
        marker=dict({'color':'orange'})
    )
).add_trace(
    go.Scatter(
        x=[np.mean(population), np.mean(population)],
        y=[0, 0.2]
        y=[0, 0.2],
        name='Population mean',
        mode='lines',
        marker=dict({'color':'green'})
    )
).update_layout(
    title='Sampling distribution of means',
    xaxis=dict(title='Bootstrapped means'),
    yaxis=dict({'title':'Density'})
)
```



We note that the area between the orange lines represent 95% of the area under the curve. We also note that the population parameters falls within these bounds. We could have chosen a different area under the curve. At 80%, the population mean would be outside of the bounds. For 80% we have 10% on either side.

```
ff.create_distplot(
    [means],
    ['Resampled means'],
    curve_type='normal'
).add_trace(
    go.Scatter(
        x=[sorted_means[99], sorted_means[99]],
        y=[0, 0.2],
        name='10%'
        name='10%',
        mode='lines',
        marker=dict({'color':'orange'})
    )
).add_trace(
    go.Scatter(
        x=[sorted_means[899], sorted_means[899]],
        y=[0, 0.2],
        name='90%',
        mode='lines',
        marker=dict({'color':'orange'})
    )
).add_trace(
    go.Scatter(
        x=[np.mean(population), np.mean(population)],
        y=[0, 0.2],
        name='Population mean',
        mode='lines',
        marker=dict({'color':'red'})
    )
).update_layout(
    title='Sampling distribution of means',
    xaxis=dict(title='Bootstrapped means'),
    yaxis=dict({'title':'Density'})
)
```



The (percentage) bounds are termed **confidence levels** and the actual values at those bounds are the **confidence intervals**. At first then, we calculated for a 95% confidence level and returned the 95% confidence interval values.

Below, we revisit our sample static and 95% confidence intervals.

```
np.mean(sample) # Sample mean
```

```
97.80136620289275
```

```
sorted_means[24] # Lower bound
```

```
94.42864993563255
```

```
sorted_means[974] # Upper bound
```

```
100.91367042455282
```

When expressing the uncertainty in our sample statistic we would then state: *The variable mean in the sample was 97.8 (95% confidence interval 94.4 - 100.9).*

Does this mean that we are 95% confident that the population parameter is within the confidence interval? NO. The confidence intervals state that if we were to repeat the experiment 100 times, we would find the population parameter in 95 of the repeat experiments.

CONFIDENCE INTERVALS USING SCIPY

We have seen the use of the *t* distribution. It is commonly used in statistical test, especially when we do not know the population parameters. The `interval` function in the stats module of the scipy package can be used to calculate the confidence intervals given a confidence level.

We use the `t.interval` function below for the *t* distribution. The `alpha` argument is the confidence level (in percentage). The `df` argument is the degrees of freedom, which is the sample size minus the number of groups (we only had a single sample group). We also need the mean (`loc` argument) and the standard error of the sample (`scale` argument with the value calculated by the `sem` function). The latter is the standard deviation divided by the square root of the sample size.

```
stats.t.interval(
    alpha=0.95,
    df=len(sample)-1,
    loc=np.mean(sample)
    loc=np.mean(sample),
    scale=stats.sem(sample)
)
```

```
(94.68938755375859, 100.91334485202691)
```

These values are very near our bootstrapped values. As the sample size increases and the number of repeated sampling increases, the confidence intervals will be closer to these values.

EXAMPLE USING THE MEDIAN

We are not stuck to the mean as sample statistic. In this example, we consider the confidence intervals for the median.

Below, we genarate random values taken from the χ^2 distribution. We imagine that the array that we create is for a continuous numerical variable in a sample. There are 200 observations in the sample.

```
np.random.seed(12)
var = np.random.chisquare(df=10, size=200)
```

We calculate the sample median below.

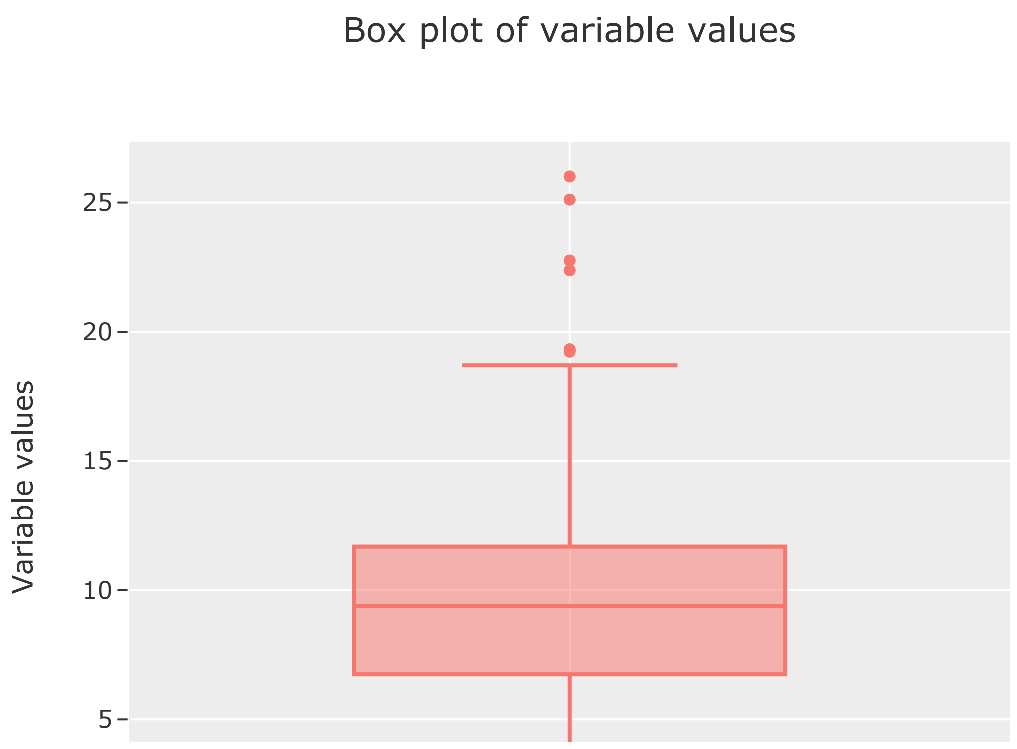
```
np.median(var)
```

```
9.378925104862498
```

We see a median of 9.38.

Next, we visulise the data.

```
px.box(
    y=var,
    title='Box plot of variable values',
    labels={'y':'Variable values'},
    width=600
)
```

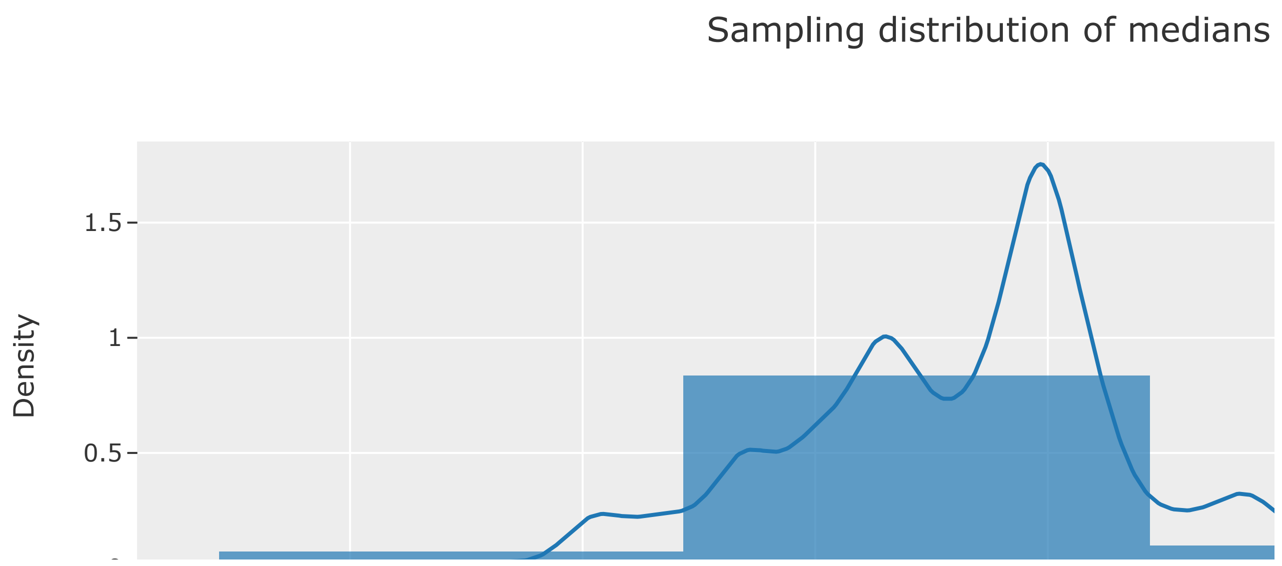


Now we use bootstrap resamples to generate a distribution of medians. Here we have 10000 bootstrapped samples

```
medians = [np.median(np.random.choice(var, 200, replace=True)) for i in range(1(
```

We use again to view a distribution plot of the medians. We use the `kde` (kernel density estimate) value for the `curve_type` argument, as the distribution is not normal.

```
ff.create_distplot(
    [medians],
    ['Resampled medians'],
    curve_type='kde'
).update_layout(
    title='Sampling distribution of medians',
    xaxis=dict(title='Bootstrapped medians'),
    yaxis=dict({'title':'Density'})
)
```



For a 95% confidence level, we still use equation (1).

```
k_2_5 = 2.5 / 100 * 10000
k_97_5 = 97.5 / 100 * 10000
```

```
k_2_5
```

```
250.0
```

```
k_97_5
```

```
9750.0
```

We need the medians sorted and to remember to use 0 indexing.

```
sorted_medians = np.sort(medians)
```

Now we can calculate the 95% confidence intervals.

```
# Lower bound
sorted_medians[249] # The median that represents a 2.5% percentile
```

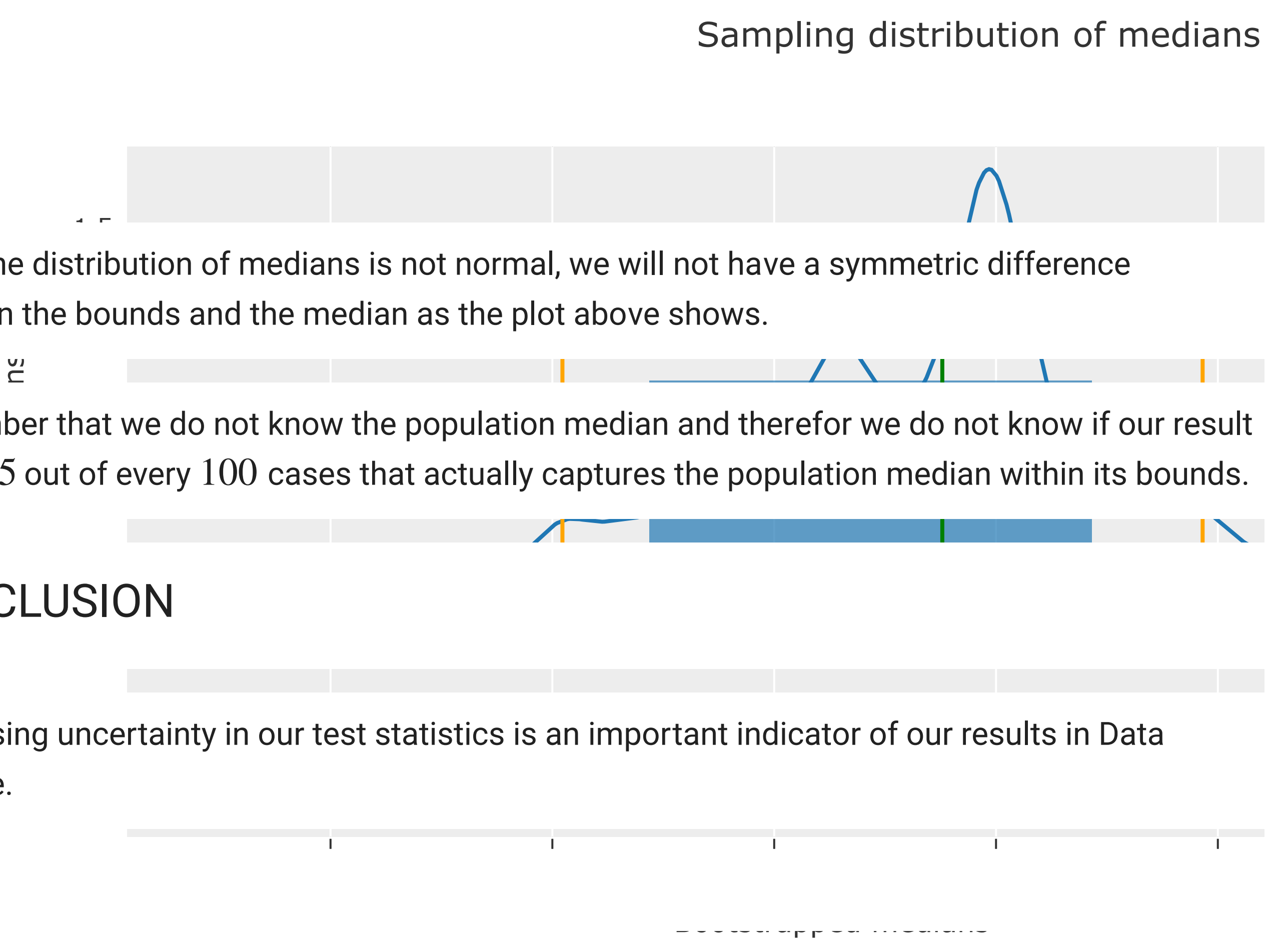
```
8.522691490434514
```

```
# Upper bound
sorted_medians[9749] # The median that represents a 97.5% percentile
```

```
9.965635026054326
```

We can now plot these confidence interval values and our original median.

```
ff.create_distplot(
    [medians],
    ['Resampled medians'],
    curve_type='kde'
).add_trace(
    go.Scatter(
        x=[sorted_medians[249], sorted_medians[249]],
        y=[0, 1.5],
        name='2.5%',
        mode='lines',
        marker=dict({'color':'orange'})
    )
).add_trace(
    go.Scatter(
        x=[sorted_medians[9749], sorted_medians[9749]],
        y=[0, 1.5],
        name='97.5%',
        mode='lines',
        marker=dict({'color':'orange'})
    )
).add_trace(
    go.Scatter(
        x=[np.median(var), np.median(var)],
        y=[0, 1.5],
        name='Original sample median%',
        mode='lines',
        marker=dict({'color':'green'})
    )
).update_layout(
    title='Sampling distribution of medians',
    xaxis=dict(title='Bootstrapped medians'),
    yaxis=dict({'title':'Density'})
)
```



Since the distribution of medians is not normal, we will not have a symmetric difference between the bounds and the median as the plot above shows.

Remember that we do not know the population median and therefor we do not know if our result is the 95 out of every 100 cases that actually captures the population median within its bounds.

CONCLUSION

Expressing uncertainty in our test statistics is an important indicator of our results in Data Science.

1

0s completed at 14:03