What to Do When we have a Data that is too big for memory.

Possible solutions:

Money-costing solution:

One possible solution is to buy a new computer with a more robust CPU and larger RAM that is capable of handling the entire dataset. On the other hand, rent a cloud or a virtual memory and then create some clustering arrangement to handle the workload.

Time-costing solution:

Your RAM might be too small to handle your data, but often, your hard drive is much larger than your RAM. Therefore, use the hard drive to deal with your date will make the processing of it much slower because even an SSD hard drive is slower than a RAM.

There are some techniques that you can use to handle big data that do not require spending any money or having to deal with long loading times.

Technique 1: Compression

The first technique we will cover is compressing the data. Compression here does not mean putting the data in a ZIP file; it instead means storing the data in the memory in a compressed format.

In other words, compressing the data is finding a way to represent the data in a different way that will use less memory. There are two types of data compression: lossless compression and lossy one. Both these types only affect the loading of your data and will not cause any changes in the processing section of your code.

Lossless compression:

Lossless compression does not cause any losses in the data. That is, the original data and the compressed ones are semantically identical. You can perform lossless compression on your data frames in 3 ways:

Load specific columns:

Loading the entire dataset takes huge memory space!

However, when we really only need two columns of the dataset, so why would we load the entire dataset? Loading only the needed columns, I need which will require less memory usage. We can do this using pandas as shown here:

```
Df= pd.read_csv(dataPath, usecols= [col1, col2,…])
```

Manipulate datatypes

Another way to decrease the memory usage of our data is to truncate numerical items in the data. For example, whenever we load a CSV into a column in a data frame, if the file contains numbers, it will store it as which takes 64 bytes to store one numerical value. However, we can truncate that and use other int formats to save some memory.

If we know that the numbers in a particular column will never be higher than 32767, we can use an int16 or int32 and reduce the memory usage of that column by 75%. So, assume that the number of cases in each county cannot exceed 32767 — which is not true in real-life — then, we can truncate that column to int16 instead of int64.

Sparse columns:

If the data has a column or more with lots of empty values stored as NaN, you save memory by using a sparse column representation so you will not waste memory storing all those empty values.

Assume the county column has some NaN values and I just want to skip the rows containing NaN, I can do that easily using sparse series.

Lossy compression:

What if performing lossless compression was not enough. What if we need to compress the data even more? In this case, we can use lossy compression, so we sacrifice 100% accuracy in your data for the sake of memory usage.

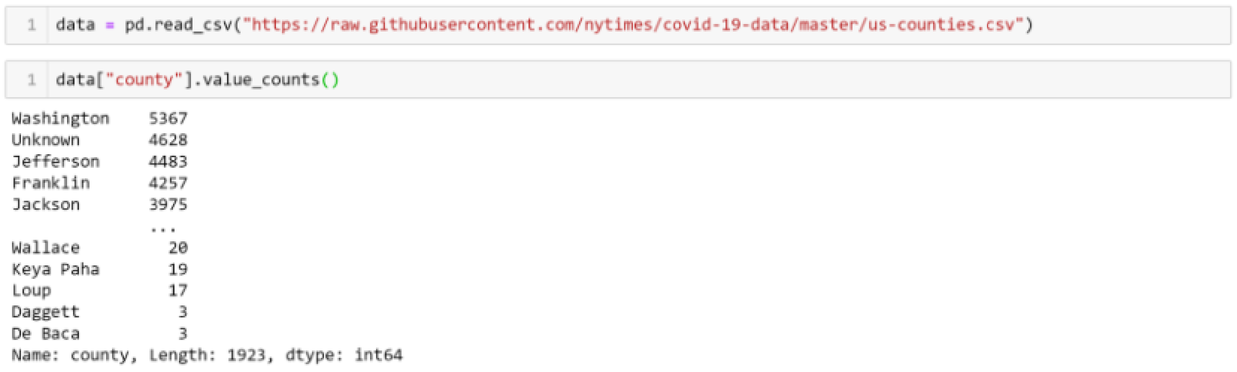
We can perform lossy compression in two ways: modify numeric values and sampling.

Modifying numeric values: Sometimes, you do not need full accuracy in your numeric data so that you can truncate them from int64 to int32 or int16.

Sampling: Maybe we want to prove that some states have higher COVID cases than others do, so we take a sample of some counties to see which states have more cases. Doing that is considered lossy compression because we are not considering all rows.

Technique 2: Chunking

Another way to handle large datasets is by chunking them. That is cutting a large dataset into smaller chunks and then processing those chunks individually. After all the chunks have been processed, you can compare the results and calculate the final findings.



```python
data = pd.read_csv("https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv")
```

```python
data["county"].value_counts()
```

```
Washington    5367
Unknown       4628
Jefferson     4483
Franklin      4257
Jackson       3975
              ...
Wallace         20
Keya Paha       19
Loup            17
Daggett          3
De Baca          3
Name: county, Length: 1923, dtype: int64
```

Let’s assume we want to find the country with the most number of cases. We can divide my dataset into chunks of 100 rows, process each of them individually, and then get the maximum of the smaller results

Technique 3: Indexing

Chunking is excellent if we need to load our dataset only once, but if we want to load multiple datasets, then indexing is the way to go.

Think of indexing as the index of a book; we can know the necessary information about an aspect without needing to read the entire book.