MODULE: (2024) 5DATA002W.2

Machine Learning and Data Mining

Python Lab

Seminars 1 & 2

Explore, Understand, & Manipulate Your Data Like A Pro!

Seminar1

Part (A) Navigating Google Colab Environment
Part (B) Coding in Google Colab Environment
Part (C) Data Manipulations in Google Colab

Seminar 2

Part (D) Data Exploration and Visualisation
Part (E) Finding Outliers & Extreme Values
Part (F) Mitigating Missing Data
Part (G) Data Scaling
Part (H) Saving Your Colab Notebook

Python Lab Seminar 1 & 2

Explore, Understand & Manipulate Data Like a Pro!

Part (A) Navigating Google Colab Environment

What is Google Collaboratory and Jupyter Notebook?

Google Colab is a convenient and easy-to-use way to run Jupyter notebooks on the cloud, and their free version comes with some access to GPUs. Jupyter Notebook (formerly known as IPython Notebook or ipynb) is used to create interactive notebook documents that can contain live code, equations, visualisations, media and other computational outputs. Jupyter Notebook is often used by data scientists and students to document and demonstrate coding workflows or simply experiment with code.

Why use Colab Notebook (.ipynb files) and not Python Integrated Development Environment (.py files)?

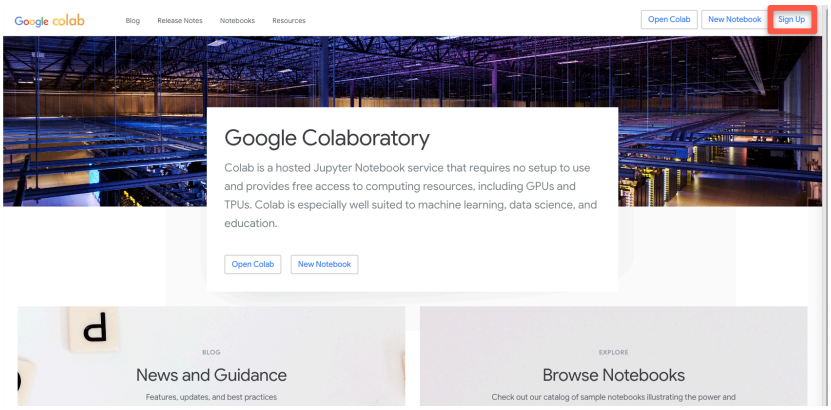
Colab notebooks allow you to combine **executable code** and **rich text** in a single document, along with images, HTML, and more. When you create your own Colab notebooks, they are stored in your Google Drive account. You can easily share your Colab notebooks with co-workers or friends, allowing them to comment on your notebooks, edit them, and, more importantly, see your results and replicate your work. This is called *reproducibility*. Machine learning-related methods and processes often use randomisation functions when applied, which hinders the reproduction of the same exact results when Python code is shared and run on different machines among scientists. Using Colab notebooks allows for reproducibility of results on different machines.

Colab notebooks are similar to Jupyter Notebook, is an open-source web application that allows users to create and share documents that contain live code and results (code outputs), including equations, visualisations, and narrative text. On the other hand, Python IDLE is an integrated development environment (IDE) that provides a basic interface for writing and executing Python code. (.py) files saved in Python IDLE are suitable for reproducing your machine-learning results.

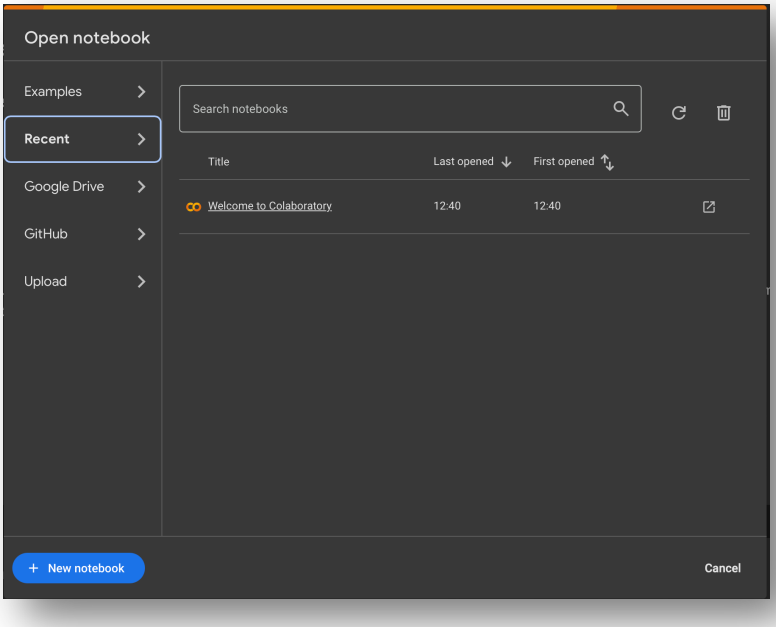
Therefore, from now on, you will save your Python Machine Learning experiments as Colab Notebook (.ipynb). We can use **Google Collaboratory Cloud Environment** as a convenient platform because Google Colab hosts these notebooks, so we don't use our computer resources to run the notebook.

Starting with Google Colab Notebooks Requires a Google Account

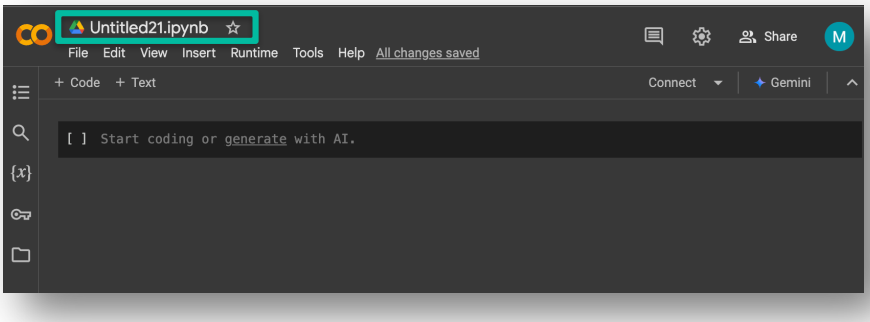
1- To launch Google Colab https://colab.google/
2- Log in with your Google account or sign up for one.

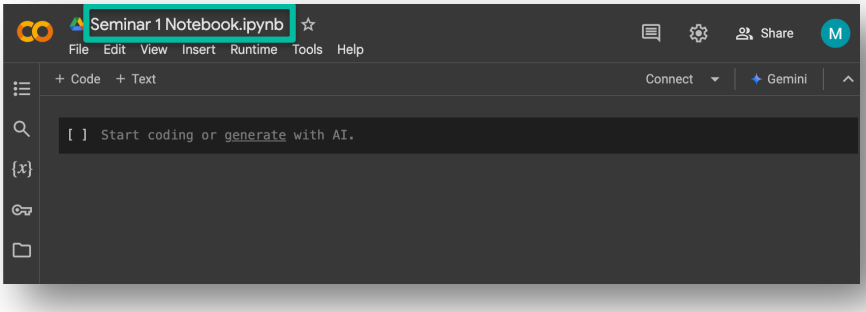


3- Select **New Notebook** to create a new Python Colab Notebook. If you are not signed into Google, you will be asked to do so. The other presented option, **Recent,** shows you the last Python notebooks you worked with. **Google Drive** and **GitHub** allow you to load your Python Notebooks from your Google Drive and GitHub; The **Upload** option allows you to load your Python Notebooks from your PC hard drive filing system.



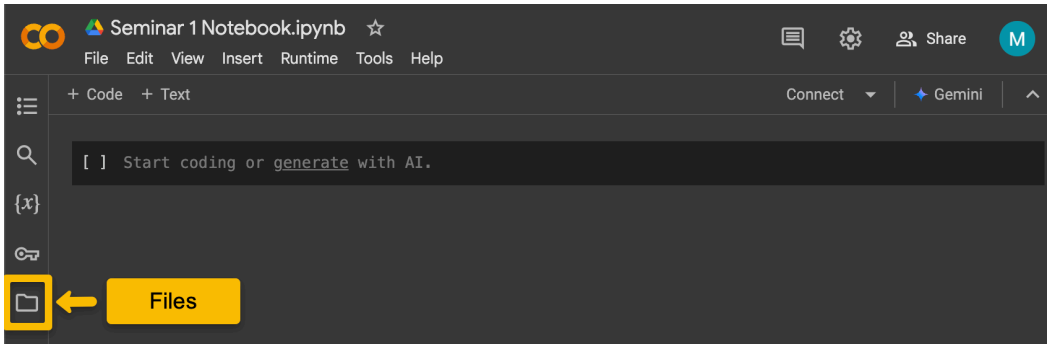
4- Once you start a new notebook, you can modify the notebook name according to your preference by editing **Untitled.ipynb**. Ensure you keep the file extension **.ipynb**

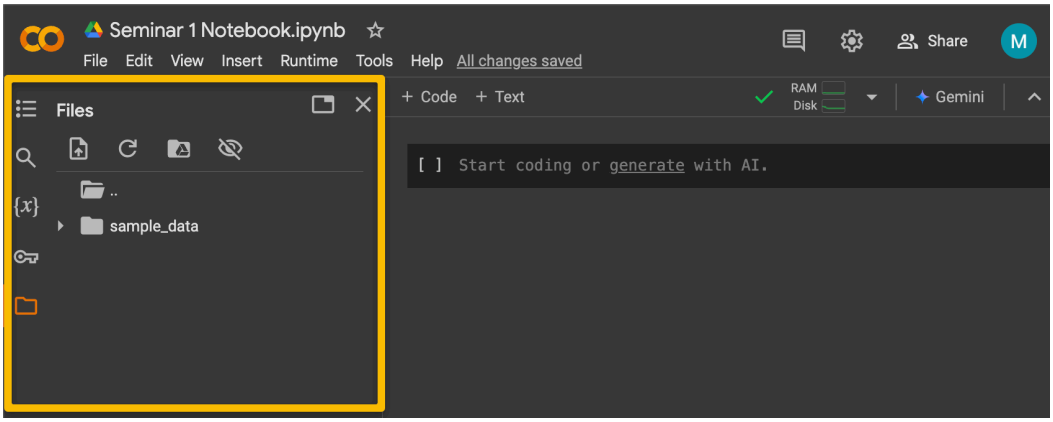




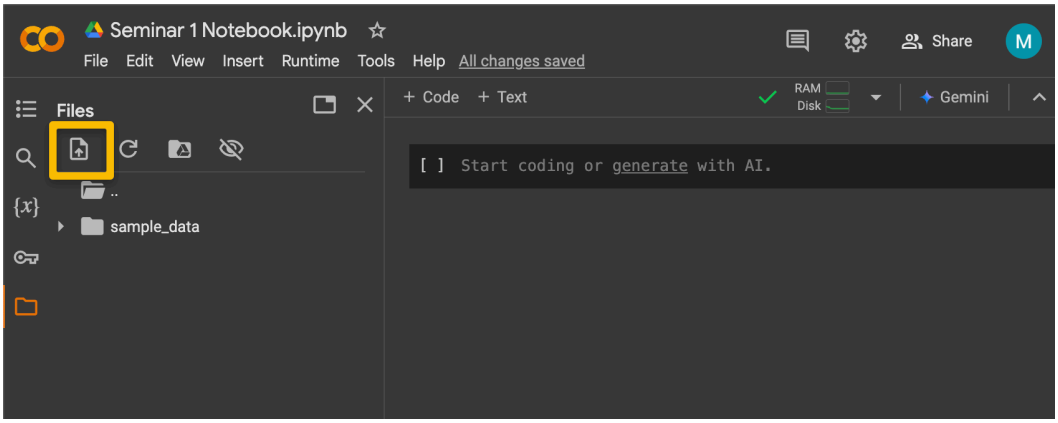
5- Download your **Mall Customers Dataset** from your Blackboard seminar folder. This will be downloaded to your PC hard drive in your download folder. Remember, we are not working on the PC environment right now; we are working on a cloud environment; therefore, we need to transfer the dataset from your PC to the cloud hard drive ☺ , This is called **"Upload".**

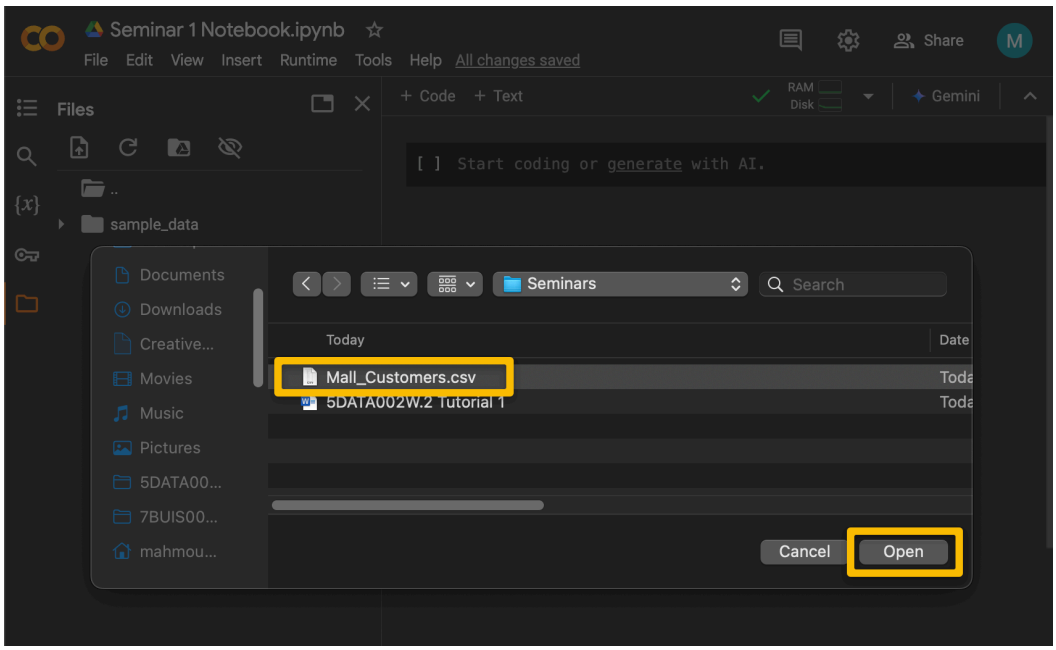
6- In Colab, Select the **"Files"** icon to expand the files side bar.



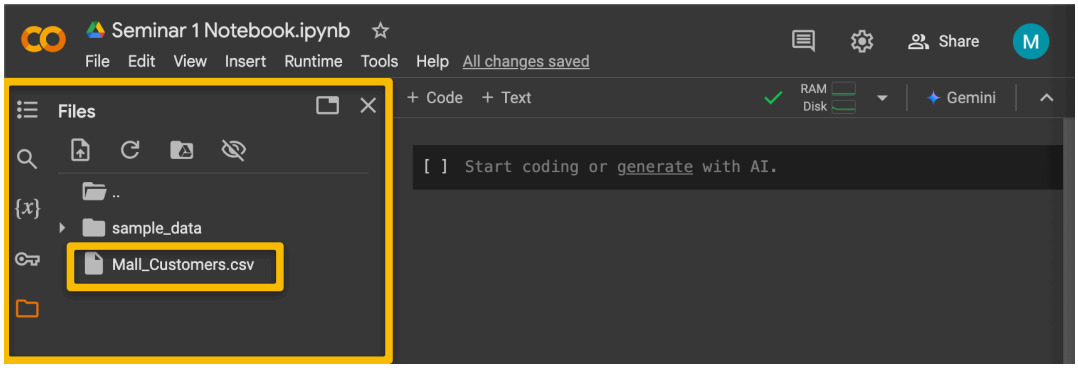


7- To upload your Dataset from your PC (local machine) storage drive to the Colab Cloud storage, select the **“Upload to session storage”** icon. Select you Mall Customer Dataset from your directory then click “**Open**”.

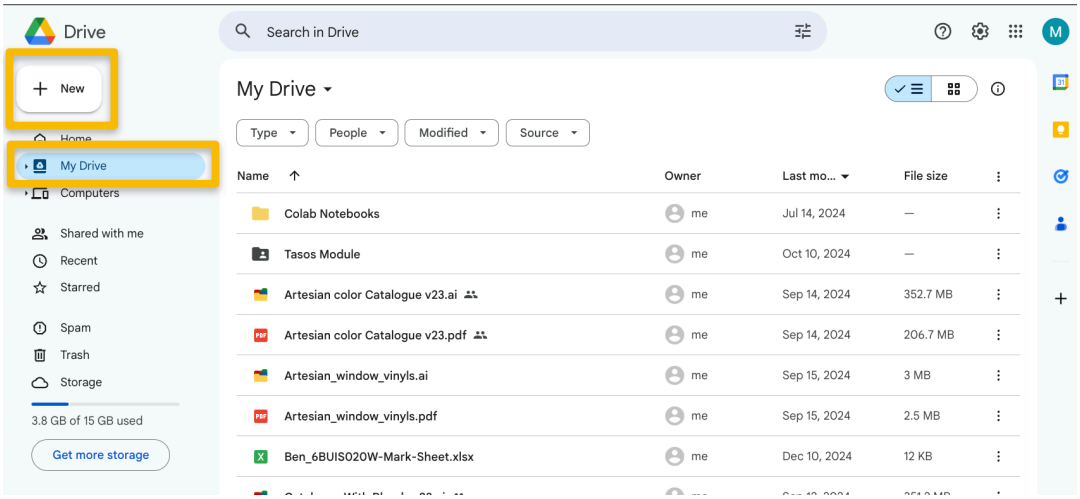


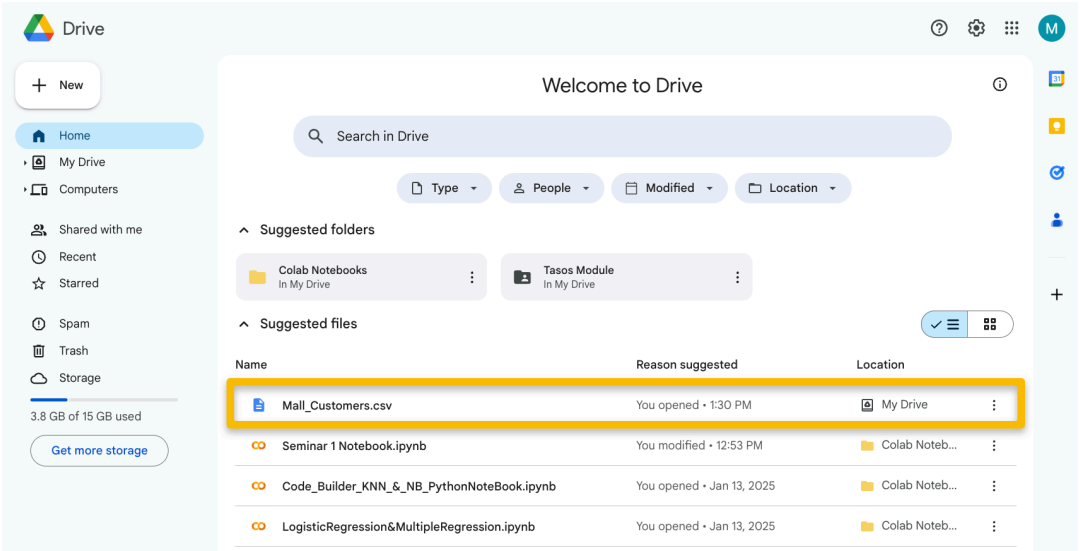


8- Wait for a second, and you shall see your dataset file in the Files sidebar.

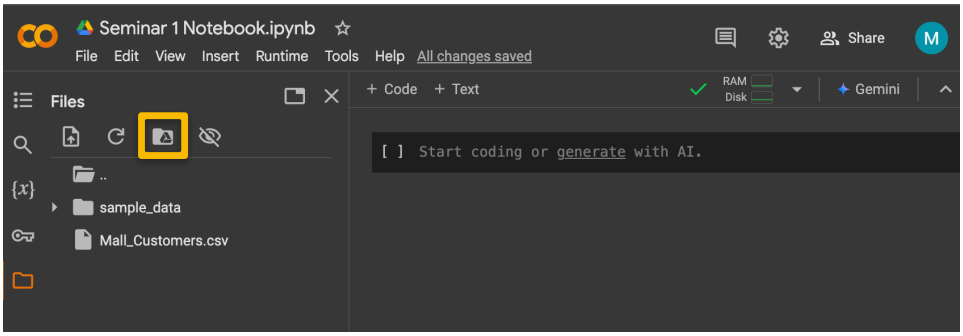


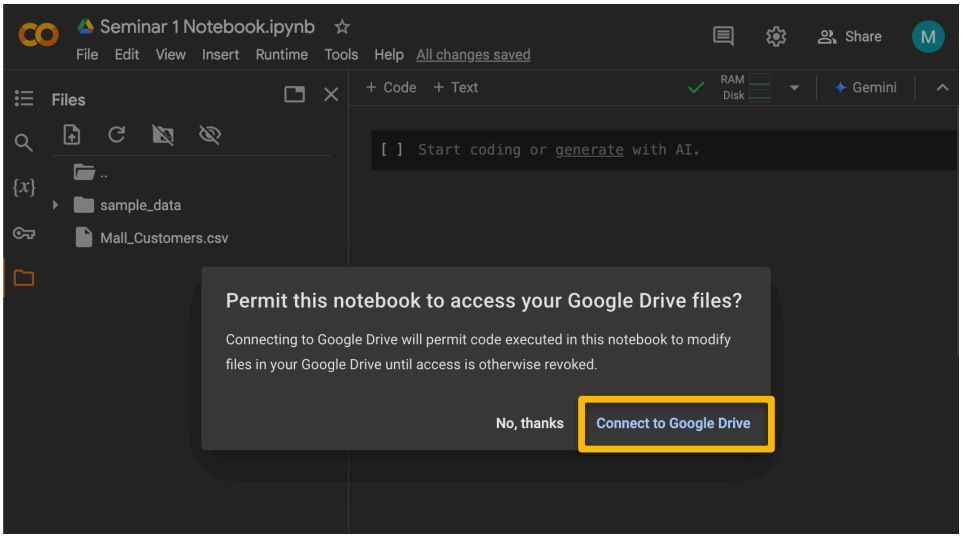
9- Sometimes, your dataset is not on your local machine. Instead, it is in your Google Drive. Therefore, another way to upload your dataset is to mount your Google Drive first! To test this functionality, go to your Google Drive and upload your Customers Mall Dataset to your Google Drive. Select "**+New**" then "**Upload file**" to upload your dataset to google drive.



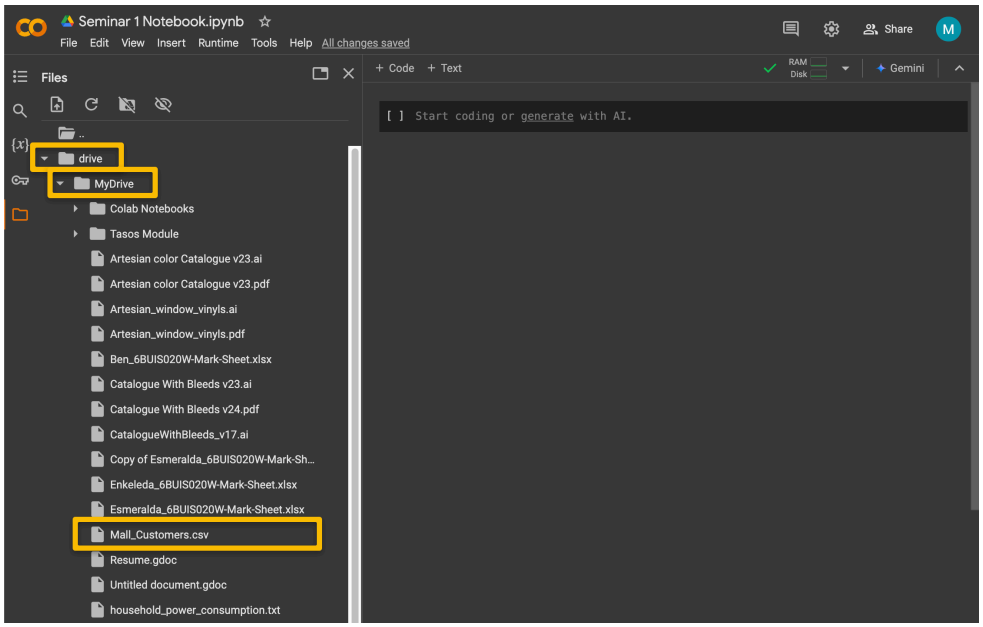


10- Once your dataset is saved to your Google Drive, from the Files sidebar, In Colab, select "**Mount Drive**". Grant Colab permissions to access your Google Drive "**Connect to Google Drive**", this will point the Colab cloud to use your Google Drive as its storage drive ☺.





11- Wait for a few moments, and you will see a new directory in the Files sidebar, "**drive**" by clicking on it, you can expand it to see all your files on Google Drive, one of which is your Mall Customer Dataset.

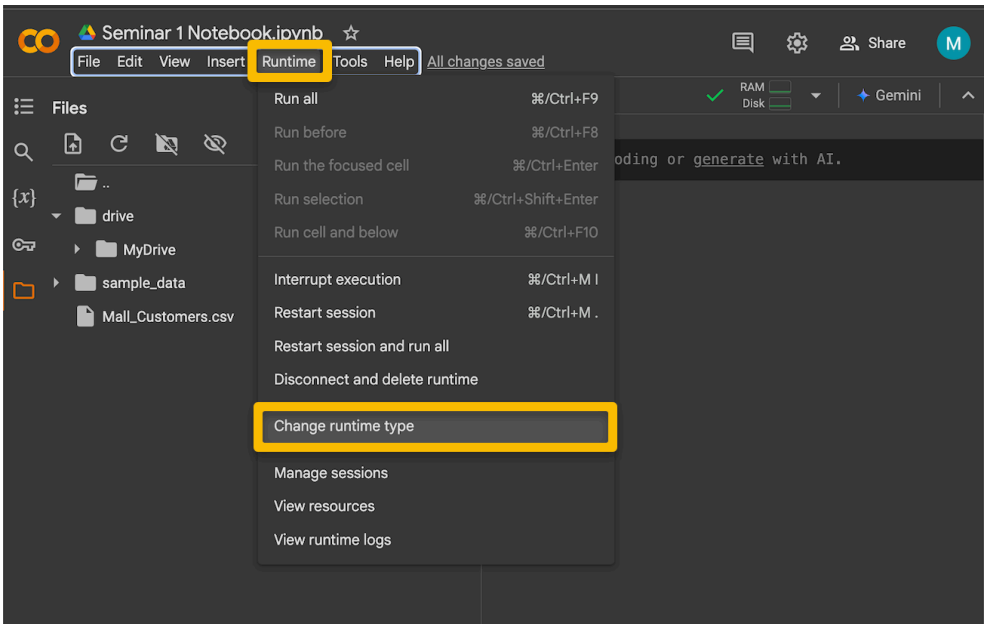


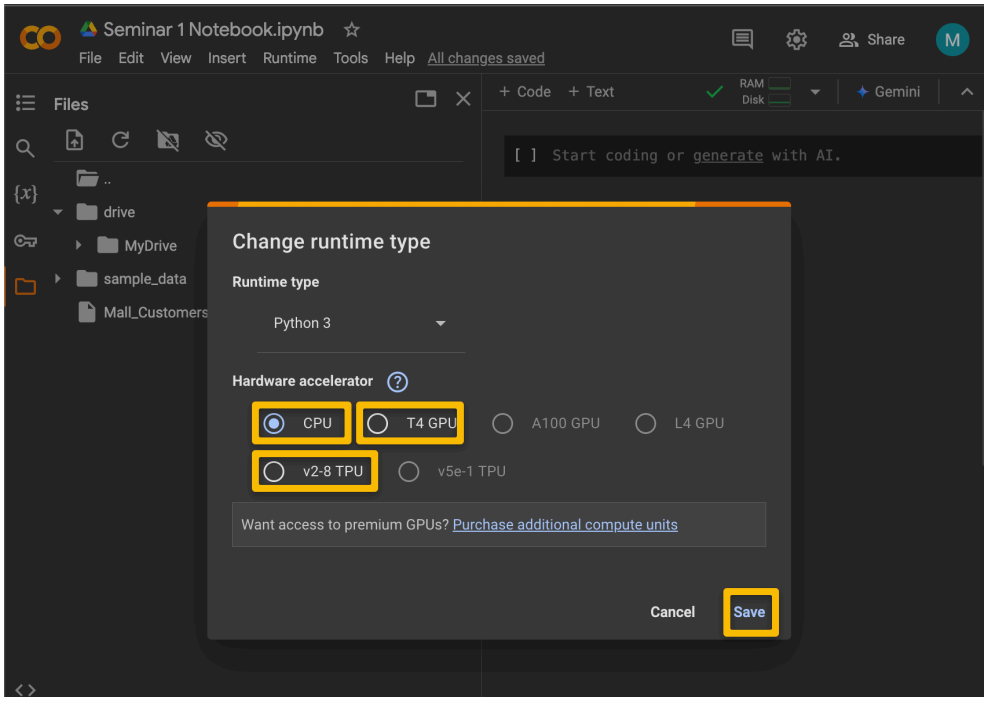
12- Remember that mounting and retrieving files from Google Drive in Colab is much faster than manually uploading files into Colab; this is very useful when using large datasets like image datasets, which are multiple Giga Bytes in size. Using the "Upload" function in Colab for these large files means a significant amount of waiting time.

13- Like a PC, the Colab environment runs hardware on a CPU; you can accelerate your hardware using a GPU or even a TPU. CPUs, Graphics Processing Units (GPUs), and Tensor Processing Units (TPUs) are all processors that perform computing tasks. CPUs are general-purpose chips, GPUs are specialised for accelerated computing tasks like graphic rendering and AI workloads, and TPUs are Google's custom Application-Specific

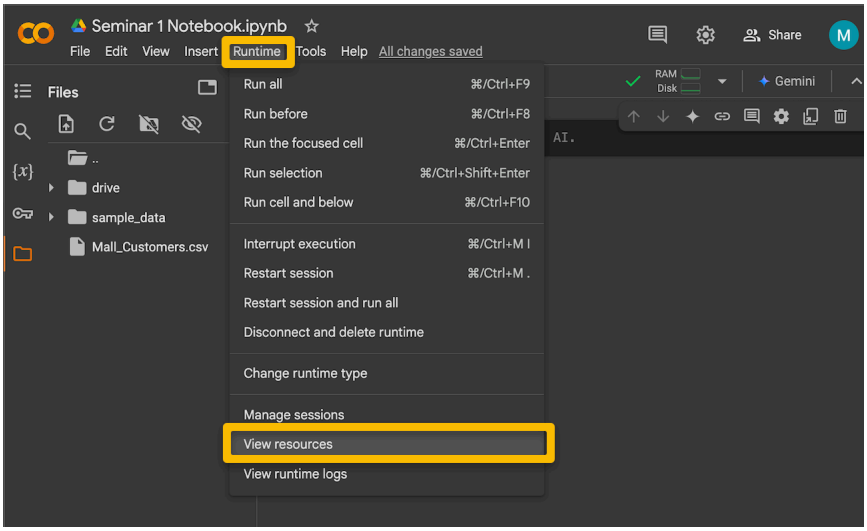
Integrated Circuit (ASICs) designed specifically for AI-based computing tasks. GPUs have the ability to break complex problems into thousands or millions of separate tasks and work them out all at once, while TPUs were designed specifically for neural network loads and have the ability to work quicker than GPUs while also using fewer resources. Depending on your data types and size, use these to your advantage; machine learning modelling is a resource-hungry operation and requires faster processing power.

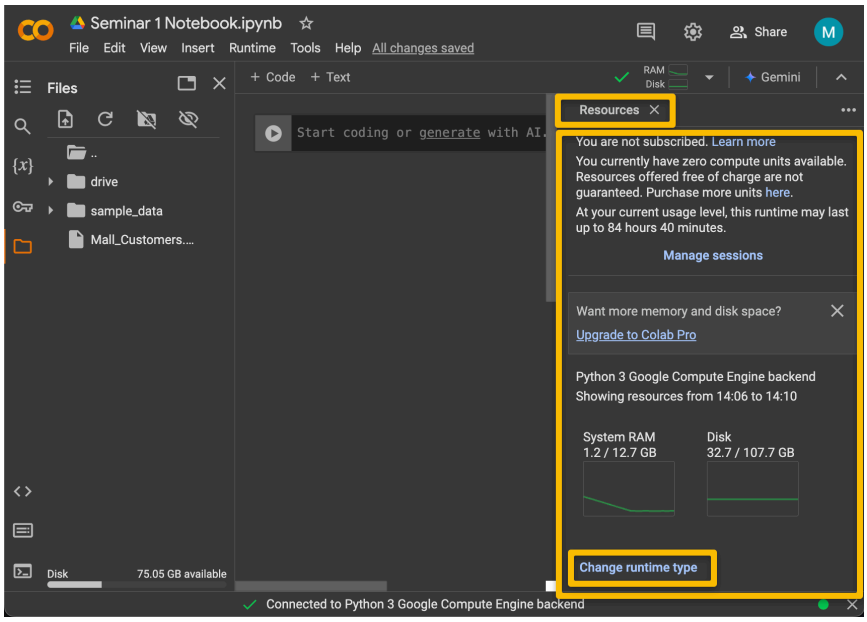
14- To use a GPU or a TPU for your Colab environment, go to the "**Runtime**" dropdown menu, select "**Change runtime type**", and select your preferred processing unit. Some are freely available on the free Colab version, while others require a subscription. Our Mall customer Dataset is a small .csv file, so for now, stay on "**CPU**".





15- When creating machine learning models, it is important to keep an eye on the environment computing resources that you are using, especially if they are expected to run for a long period of time (i.e., hours); you need to ensure sufficient resources are there to complete the machine learning task and avoid runtime interruptions and termination. To view your runtime resources from the "**Runtime**" dropdown, select "**View resources**". If you are straining/depleting your environmental resources, consider "**Change runtime type**". Additional computing resources may cost additional subscription payment, but you do not need this for our module.

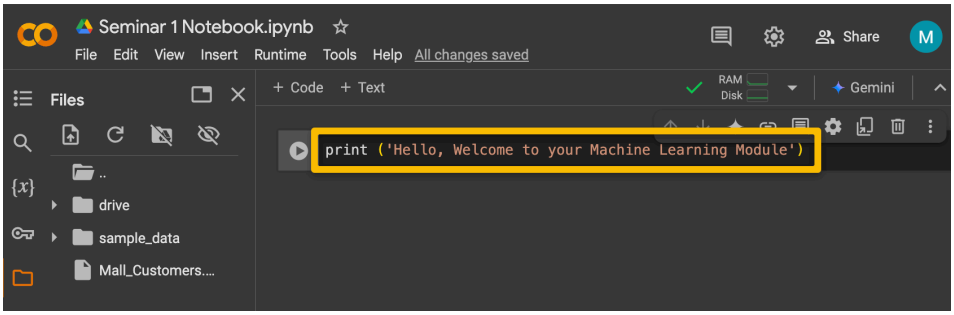




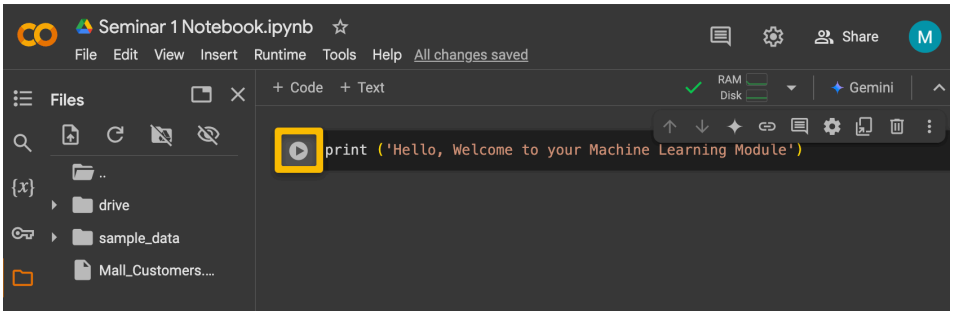
Part (B) Coding in Google Colab Environment

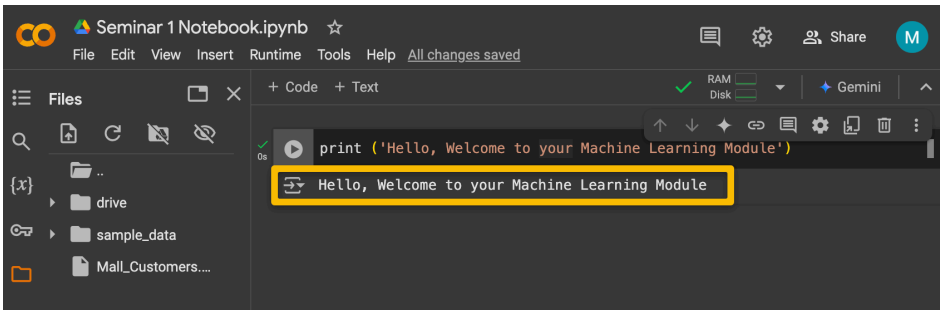
1- To code in Colab, type your code in the black coding cell. Try it, copy and paste the following python code into the coding cell.

```
print ('Hello, Welcome to your Machine Learning Module')
```

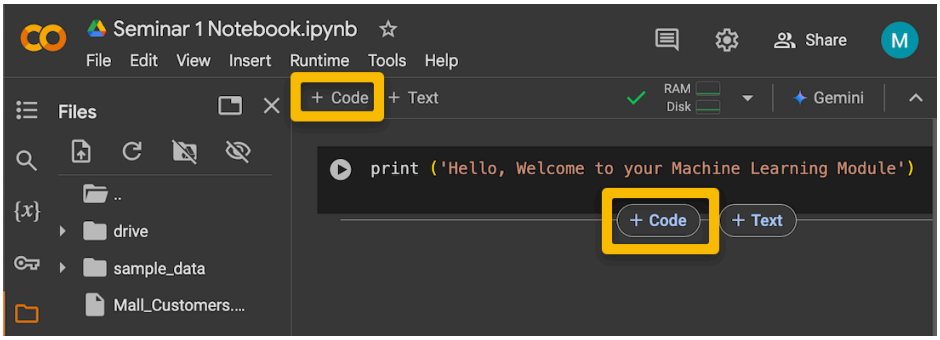


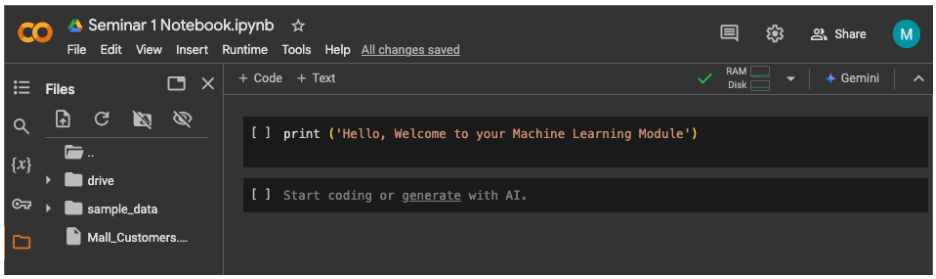
2- To run your code, you can select the "**Run cell**" button or press "**Ctrl + Enter**" from your Windows keyboard. From Mac OS, press "**Command + Enter**" and you will see the output below the code cell.



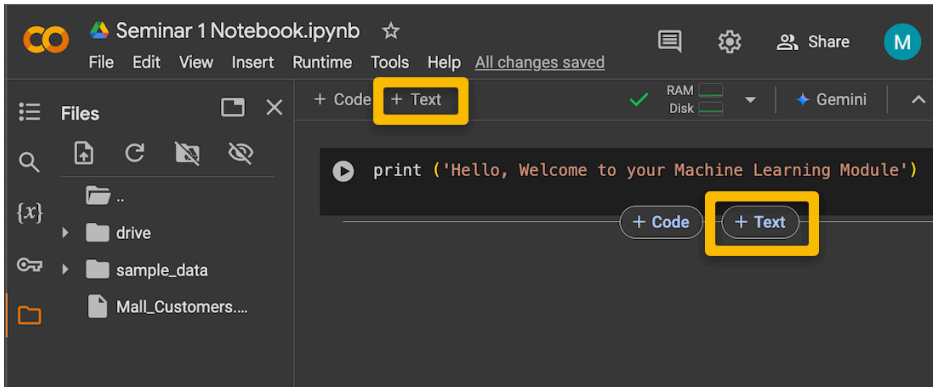


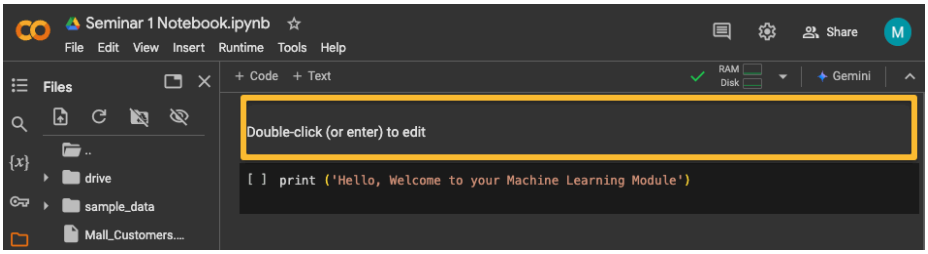
3- To add additional code cells, you can click "**Insert code cell below**" or you can click the "**Add code cell button**" after hovering on the cell itself.



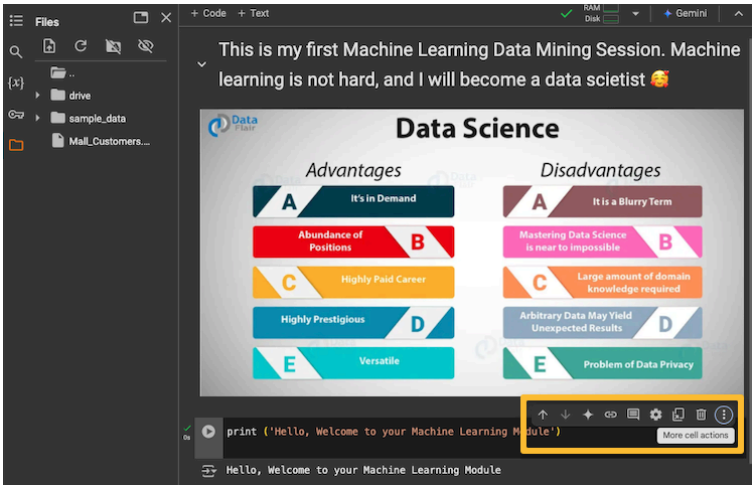


4- You may want to write notes, comments, or further explanations about the code cell you created. To add a text cell above or below the code cell, you can simply use "**Add text cell**". You add and edit text and attach images, links, and emojis, too. This organises your notebook. Simply double-click in the text box, edit, and then click the **ESC** button on your keyboard.



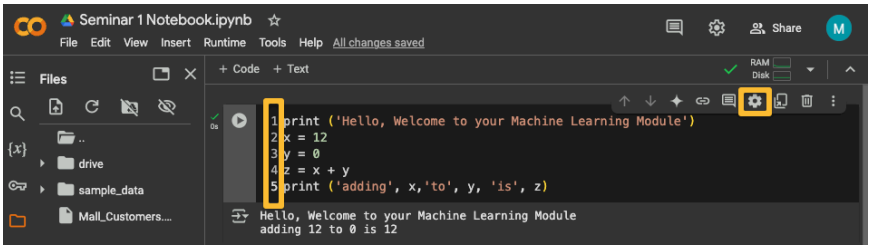


5- To delete code cells or text cells, you can click on the cell, and immediately you shall see the "**Cell menu**", from the cell menu, you can perform so many functions on the cell, including delete, copy, paste, move cells and others in "**More cell actions**" including customising your Google Colab workspace view.



6- Paste the following code in a cell code and explore the "**Open editor settings**" option from the "**cell menu"** to add line numbers to each line of code in the code cell.

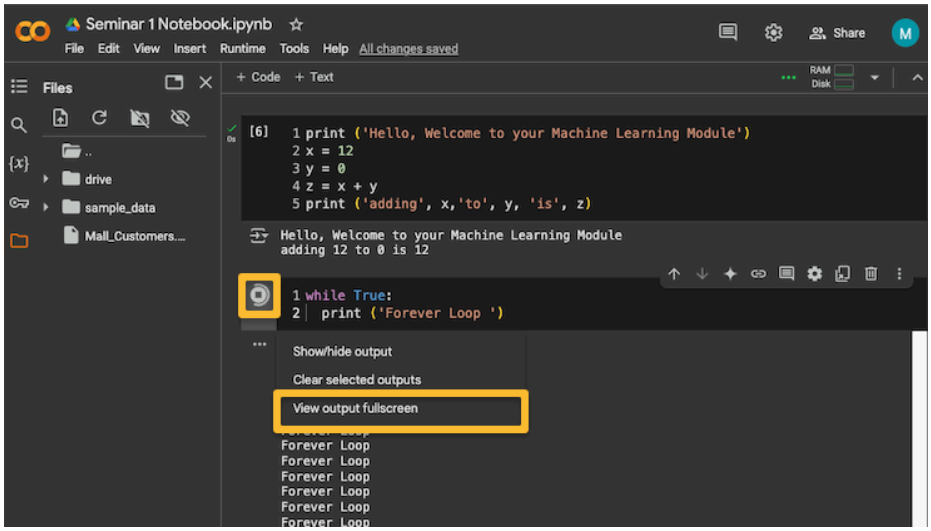
```
print ('Hello, Welcome to your Machine Learning Module')
x = 12
y = 0
z = x + y
print ('adding', x,' to', y, 'is', z)
```



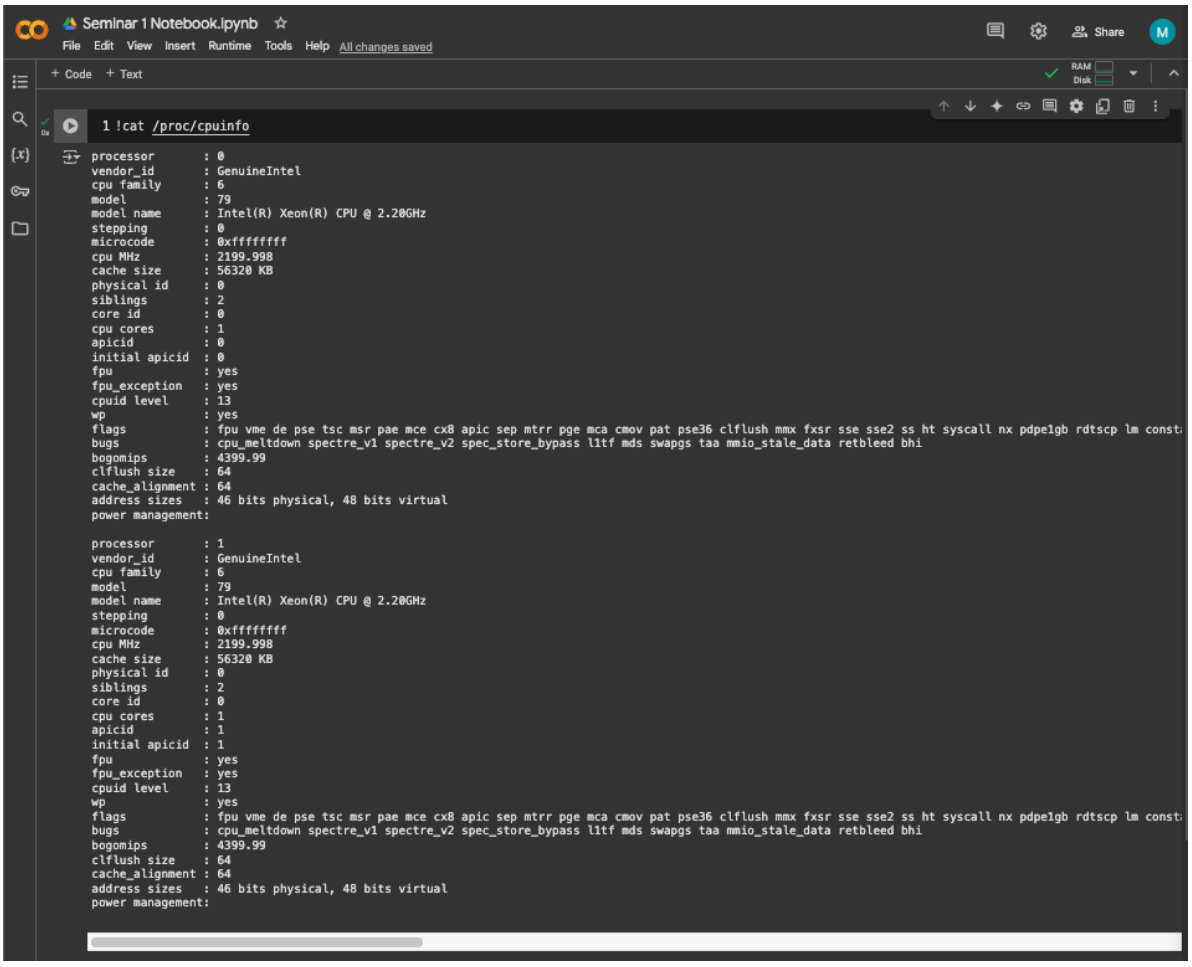
7- There are many operations from "**Code cell output actions**" you can perform on your code output display, such as "**hide/show**", "**Clear selected output**", and "**View output in full screen**" Try this code:

```
while True:
  print ('Forever Loop ')
```

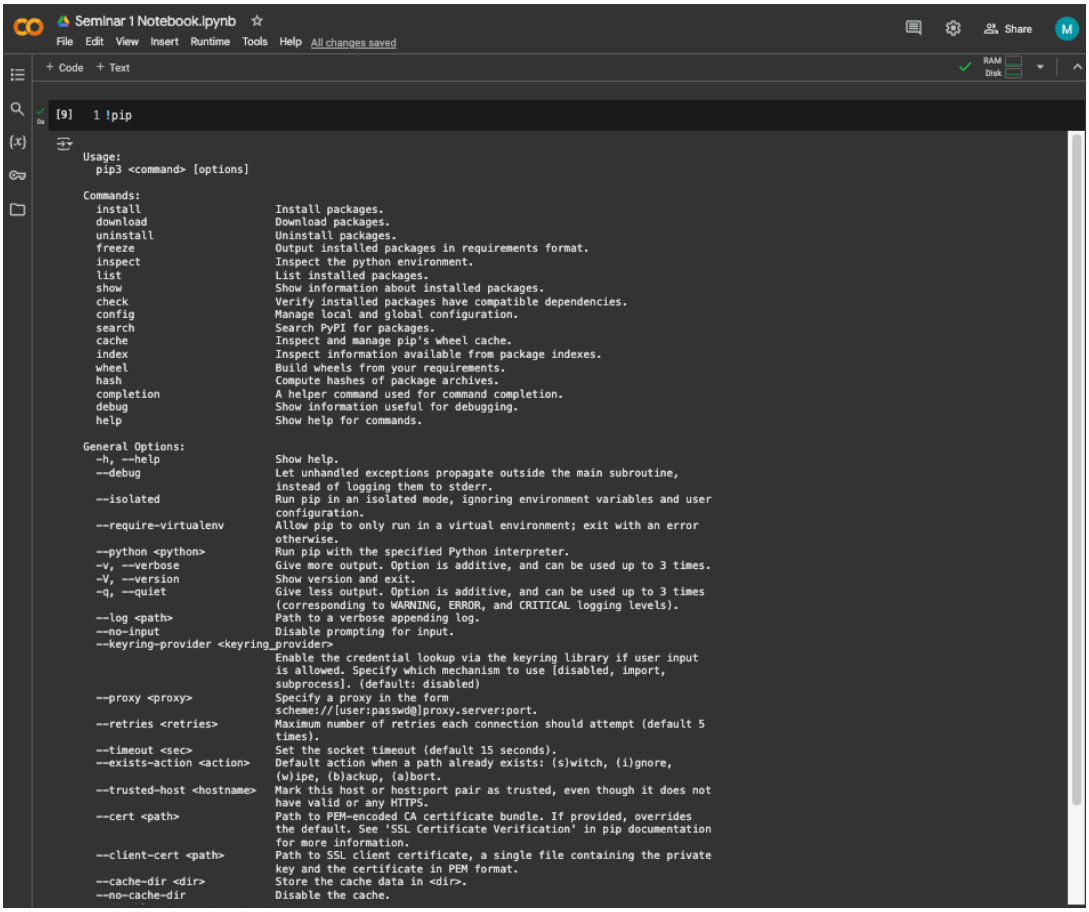
The above code will create an infinite loop due to the absence of a break. You can interrupt the execution if your program takes too long to run by clicking "**Interrupt**" or using the keyboard shortcut **Ctrl + M + I.** Alternatively, you can view the output on the full screen. See **Appendix (A)** for more useful keyboard shortcuts for Colab notebooks if you want to try them.



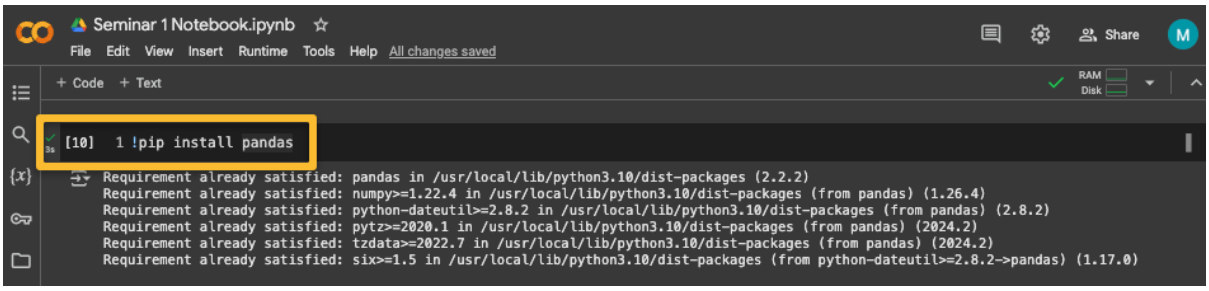
8- For environment **configurations and package installation**, we can run some shell commands with **!** to tell the configuration of the Colab environment.
!cat /proc/cpuinfo



More importantly, **!** is used for installing new libraries into the Google Colab environment with **!pip**



Now, try installing the following data manipulations library by running the command **!pip install pandas**



Note for your coursework: When the task asks for the code output, paste only the output from the output cell, NOT the code cell. If the coursework task asks for the code block or line, this is copied from the code cell.

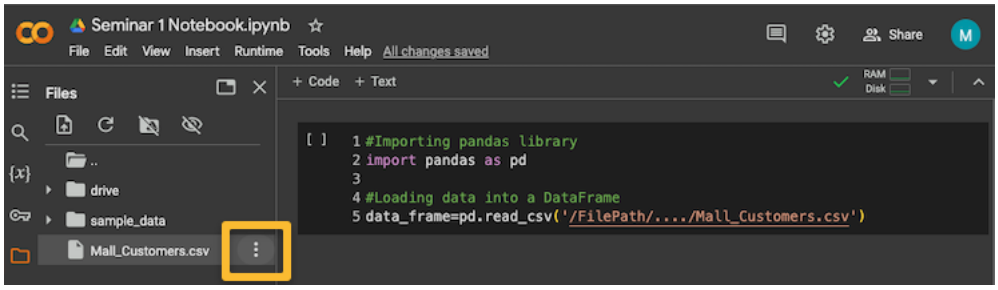
Part (C) Data Manipulations in Google Colab

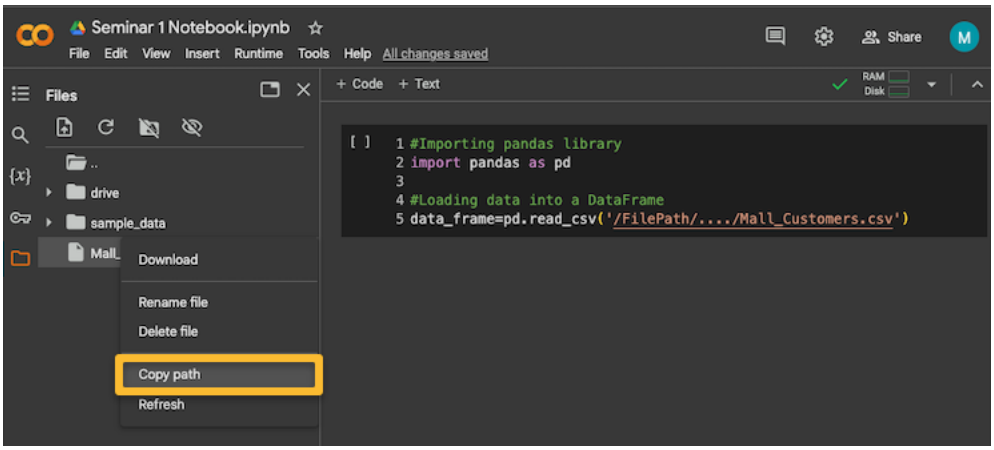
Data Manipulation is changing data into a more organised format according to one's requirements. Thus, Data Manipulation involves the processing of data into useful information. Through the **Pandas library,** data manipulation becomes easy. Hence, let's understand Data Manipulation with Pandas in more detail. We will also use the **Mall_Customers dataset** to show the syntax of these functions in the work.

Pandas is a powerful, fast, and open-source library built on NumPy. It is used for data manipulation and real-world data analysis in Python. Easy handling of missing data, Flexible reshaping and pivoting of data sets, and size mutability make pandas a great tool for performing data manipulation and handling the data efficiently.
1- Load your dataset in Colab with Pandas library: Reading CSV file using **pd.read_csv** and loading data. You must *import pandas as* using *pd* for the shorthand.

```
#Importing pandas library
import pandas as pd
#Loading data into a DataFrame
data_frame=pd.read_csv('/FilePath/..../Mall_Customers.csv')
```

Remember, we are using the dataset in the Colab storage, not your local machine, so we need to find its file path in the Colab environment. Hover over your data file in the expanded file sidebar, click on the "**Kebab menu**" icon, then select "**Copy path**", then paste it in the **read_csv function argument**. Depending on the directory where your Dataset file is stored on Colab, the file dataset path may vary. In my case, the file path **'/content/Mall_Customers.csv'.** Then, run the cell.






```
#Importing pandas library
import pandas as pd
#Loading data into a DataFrame
data_frame=pd.read_csv('/content/Mall_Customers.csv')
```

2- **Accessing your dataset values:** As part of data exploration, you can print the rows and columns of your dataset on the screen to see their values. Copy and paste this cell into your Colab. By default, **data_frame.head()** displays the first five rows and **data_frame.tail()** displays the last five rows. If we want to get the first 'n' number of rows, then we use **data_frame.head(n)** is similar to the syntax used to print the last n rows of the data frame, **data_frame.tail(n)**

Code cell:

```
#displaying first five rows
data_frame.head()
```

Output cell:

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19.0	15.0	NaN
1	2	Male	NaN	15.0	81.0
2	3	Female	20.0	16.0	6.0
3	4	Female	23.0	NaN	77.0
4	5	Female	NaN	NaN	40.0

3- You can display the column names of the data_frame by applying the **list function**

Code cell:

```
# Program to print all the column names of the data_frame
list(data_frame.columns)
```

Output cell:

```
['CustomerID', 'Gender', 'Age', 'Annual Income (k$)', 'Spending Score (1-100)']
```

4- Data formatting issues: having **correct data types is critical to machine learning**. Every machine learning algorithm processes a range of variables with specific data types; without the correct data types, the machine learning algorithm throws an error. Exploring data types tells you if there are any formatting problems in values that your dataset holds. The functions **info()** prints the summary of a data_frame that includes the data type of each column.

Code cell:

```
data_frame.info()
```

Output cell:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 5 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   CustomerID              200 non-null    int64
 1   Gender                  200 non-null    object
 2   Age                     197 non-null    float64
 3   Annual Income (k$)      198 non-null    float64
 4   Spending Score (1-100)  199 non-null    float64
dtypes: float64(3), int64(1), object(1)
memory usage: 7.9+ KB
```

5- Recorded data value errors: Formatting errors are not the only issue you may experience in your dataset. **There can be value errors**. Value error can render your machine learning entity's interpretation untrustworthy. One way to find errors in your dataset is by looking at its descriptive stats. The **describe() function** outputs **descriptive statistics,** which include those that summarise the central tendency, dispersion, and shape of a dataset's distribution, excluding NaN values. By default, for numeric data, the result's index will include **count, mean, std, min, and max, as well as lower, 50, and upper percentiles**. For object data (e.g. categorical values), the result's index will include **count, unique, top, and freq**. We will talk more about variable types in your first lecture. Copy, paste and run the following code cell in you Colab notebook.

Code cell:

```
data_frame.describe()
```

Output cell:

	CustomerID	Age	Annual Income (k$)	Spending Score (1-100)
count	200.000000	197.000000	198.000000	199.000000
mean	100.500000	39.000000	61.005051	50.256281
std	57.879185	14.002915	26.017857	25.876350
min	1.000000	18.000000	15.000000	1.000000
25%	50.750000	29.000000	42.250000	34.500000
50%	100.500000	36.000000	62.000000	50.000000
75%	150.250000	49.000000	78.000000	73.000000
max	200.000000	70.000000	137.000000	99.000000

Code cell:

```
data_frame.describe(include='object')
```

Output cell:

	Gender
count	200
unique	2
top	Female
freq	112

6- Dataset dimensions: when asked about your data dimension, you are asked about the number of rows and the number of columns. You can use the **shape** method to obtain only that information. **In machine learning, we call columns features and the rows instances.**

Code cell:

```
data_frame.shape
```

Output cell:

```
(200, 5)
```

7- Missing data values: one common issue you will find in datasets is missing data values. Depending on the portion and cause of missing values, scientists can decide on a suitable method to mitigate their missingness. Using suitable methods will likely produce reliable analysis. Therefore, to find the number of missing values in the dataset, use **data_frame.isnull( ).sum( )**. In the below example, if the dataset doesn't contain any null values, each column's output is 0.

Code cell:

```
data_frame.isnull().sum()
```

Output cell:

```
CustomerID                0
Gender                    0
Age                       3
Annual Income (k$)        2
Spending Score (1-100)    1
dtype: int64
```

To find the percentage of missing data values per variable, we understand that the length of the dataset is the number of rows, the number of customers. Therefore, missing values in a column occupy a percentage of the variable length

Code cell:

```
data_frame.isna().sum()/len(data_frame)*100
```

Output cell:

```
CustomerID                0.0
Gender                    0.0
Age                       1.5
Annual Income (k$)        1.0
Spending Score (1-100)    0.5
dtype: float64
```

7- Removing Rows (instances): By using the **drop(index)** function, we can drop the row at a particular index. If we want to replace the data_frame with the row removed.

Code cell:

```
#Removing 4th indexed value from the data_frame
data_frame.drop(4, inplace = True)
data_frame.head()
```

Output cell:

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19.0	15.0	NaN
1	2	Male	NaN	15.0	81.0
2	3	Female	20.0	16.0	6.0
3	4	Female	23.0	NaN	77.0
5	6	Female	22.0	17.0	76.0

To remove multiple rows (instances), we can include a list of indices for those instances. For example, remove instances 1 and 3.

Code cell:

```
data_frame.drop(data_frame.index[[1,3]], inplace=True)
```

8- Removing unnecessary variables from your analysis: Not all variables are useful for analysis; there are unnecessary variables; we will discuss these in our lecture. This function can also be used to remove the columns of a data frame by adding the attribute **axis =1** and providing the list of columns we would like to remove. **data_frame = data_frame.drop('column_name', axis=1)**

Code cell:

```
data_frame.drop('CustomerID',axis=1, inplace=True)
data_frame.head()
```

Output cell:

	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	Male	19.0	15.0	NaN
1	Male	NaN	15.0	81.0
2	Female	20.0	16.0	6.0
3	Female	23.0	NaN	77.0
5	Female	22.0	17.0	76.0

To remove multiple unnecessary variables, we use a list within the drop function that contains a list of the column names to be removed. Assume you want to remove two more variables: **Gender and Age**, try the following

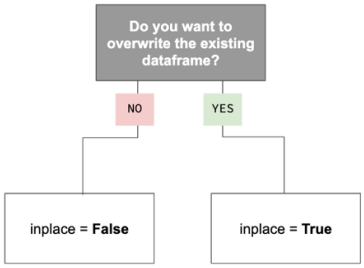
Code cell:

```
data_frame.drop(['Gender', 'Age'],axis=1, inplace=True)
data_frame.head()
```

Output cell:

	Annual Income (k$)	Spending Score (1-100)
0	15.0	NaN
1	15.0	81.0
2	16.0	6.0
3	NaN	77.0
5	17.0	76.0

When trying to make changes to a Pandas data frame using a function, we use 'inplace=True' if we want to commit the changes to the data frame. The use of inplace=True can be summarised in the following condition:



9- Renaming columns: sometimes, you may be required to change the names of your columns. To rename the columns, we list the original names of the columns to change and then apply the **rename()** function using attribute **axis=1**

Code cell:

```
#Importing pandas library
import pandas as pd
#Loading data into a DataFrame
data_frame=pd.read_csv('/content/Mall_Customers.csv')
data_frame.rename({'CustomerID':"ID", 'Gender':"Sex", 'Annual Income
(k$)':"Salary"}, axis=1, inplace=True)
data_frame.head()
```

Output cell:

	ID	Sex	Age	Salary	Spending Score (1-100)
0	1	Male	19.0	15.0	NaN
1	2	Male	NaN	15.0	81.0
2	3	Female	20.0	16.0	6.0
3	4	Female	23.0	NaN	77.0
4	5	Female	NaN	NaN	40.0

10- Renaming categorical values (labels) in variables: Some machine learning algorithms cannot process categorical variables in a string format; thus, they can throw an error. For that, we convert the (map) or (fil) each category to a numeric type of value. For example, Sex has two categorical values: Male and Female, we can map them to 1 and 2 respectively in that variable.

Code cell:

```
data_frame['Sex'] = data_frame['Sex'].map({'Male': 1 , 'Female': 2})
data_frame.head()
```

Output cell:

	ID	Sex	Age	Salary	Spending Score (1-100)
0	1	1	19.0	15.0	NaN
1	2	1	NaN	15.0	81.0
2	3	2	20.0	16.0	6.0
3	4	2	23.0	NaN	77.0
4	5	2	NaN	NaN	40.0

In the case of having a variable with a large number of categories (labels), mapping them individually can become cumbersome (difficult to handle). Therefore, you can use a **LabelEncoder()** function to automatically map all categories to numeric labels based on their alphabetical order.
Let's reload our original dataset in its original form and rename the variables again. To understand the difference between **map()** and **LabelEncoder()** functions.

Code cell:

```
import pandas as pd
data_frame=pd.read_csv('/content/Mall_Customers.csv')

data_frame.rename({'CustomerID':"ID", 'Gender':"Sex", 'Annual Income
(k$)':"Salary"}, axis=1, inplace=True)

data_frame.head()
```

Output cell:

	ID	Sex	Age	Salary	Spending Score (1-100)
0	1	Male	19.0	15.0	NaN
1	2	Male	NaN	15.0	81.0
2	3	Female	20.0	16.0	6.0
3	4	Female	23.0	NaN	77.0
4	5	Female	NaN	NaN	40.0

Let's check the unique values in the variable **"Sex"**; for that, you can use the **unique()** function.

Code cell:

```
data_frame['Sex'].unique()
```

Output cell:

```
array(['Male', 'Female'], dtype=object)
```

Now let's import the library **sklearn,** which has the **preprocessing package** with the **LableEncoder()** function to encode the categorical values for the variable **"Sex"**

Code cell:

```
from sklearn import preprocessing
label_encoder = preprocessing.LabelEncoder()

label_encoder = preprocessing.LabelEncoder()
data_frame['Sex']= label_encoder.fit_transform(data_frame['Sex'])

data_frame.head()
```

Output cell:

	ID	Sex	Age	Salary	Spending Score (1-100)
0	1	1	19.0	15.0	NaN
1	2	1	NaN	15.0	81.0
2	3	0	20.0	16.0	6.0
3	4	0	23.0	NaN	77.0
4	5	0	NaN	NaN	40.0

When comparing **data.head()** results output from the **map()** and **LabelEncoder()** functions, you notice that in the map(), Males and Females were encoded to 1 and 2, respectively. However, the **LabelEncoder()** encoded Males as 1 and Females as 0; this numeric ranking is due to the alphabetical order of the categories, F before M.

11- Filtering: this is another way of dropping instances. However, we can use filtering if we know the exact values or range of values whose instances we want to remove from the data. This can be useful if we identify a range of values that are errors or not required for the analysis. We can exclude them from the analysis by filtering them out from the dataset. For example, in our data, we are interested in analysing low-spending customers; we may want to use machine learning to find similarities among them (this will be done later during the semester).
A common operation in data analysis is to filter values based on a condition or multiple conditions. Pandas provides a variety of ways to filter data points (i.e. rows). First, Let's remove the spaces in the **Spending Score (1-100)** variable by renaming it **Spending_Score**.

Code cell:

```
data_frame.rename({'Spending Score (1-100)':"Spending_Score"}, axis=1,
inplace=True)
data_frame.head()
```

Output cell:

	ID	Sex	Age	Salary	Spending_Score
0	1	1	19.0	15.0	NaN
1	2	1	NaN	15.0	81.0
2	3	0	20.0	16.0	6.0
3	4	0	23.0	NaN	77.0
4	5	0	NaN	NaN	40.0

Now, we can use the logical operators in the column **Spending_Score** values to filter all customers whose spending score was below 70. Let's call their dataset **low_spenders_data**. Once they are filtered, check the minimum and maximum values for the **Spending_Score** variable in their descriptive stats.

Code cell:

```
low_spenders_data= data_frame[data_frame.Spending_Score < 70]
low_spenders_data.describe()
```

Output cell:

	ID	Sex	Age	Salary	Spending_Score
count	145.000000	145.000000	143.000000	144.000000	145.000000
mean	97.193103	0.448276	42.321678	59.763889	38.020690
std	52.330188	0.499041	14.760340	23.184268	18.394772
min	3.000000	0.000000	18.000000	16.000000	1.000000
25%	57.000000	0.000000	31.000000	44.000000	22.000000
50%	93.000000	0.000000	43.000000	60.000000	42.000000
75%	135.000000	1.000000	52.500000	73.000000	52.000000
max	199.000000	1.000000	70.000000	137.000000	69.000000

Pandas allow for combining multiple logical operators. For instance, we can apply conditions on both the **Salary** and **Spending_Score** columns. Perhaps you want to target high earners (above the average salary) with low spending scores (below 70). Let's call their dataset **high_earners_low_spenders_data**

Code cell:

```
high_earners_low_spenders_data =  data_frame[(data_frame.Salary > 59.76) &
(data_frame.Spending_Score < 70)]
high_earners_low_spenders_data.describe()
```

Output cell:

	ID	Sex	Age	Salary	Spending_Score
count	73.000000	73.000000	73.000000	73.000000	73.000000
mean	140.164384	0.493151	40.068493	77.287671	34.027397
std	32.965476	0.503413	13.849718	17.164627	18.828888
min	93.000000	0.000000	18.000000	60.000000	1.000000
25%	111.000000	0.000000	28.000000	63.000000	16.000000
50%	135.000000	0.000000	40.000000	73.000000	39.000000
75%	170.000000	1.000000	49.000000	87.000000	49.000000
max	199.000000	1.000000	68.000000	137.000000	69.000000

12- Variable Construction: Data Scientists usually use values in other variables to construct a new variable that holds new values. We can add new columns to our dataset. Let's call it **"New Column"** holding a single value of 1 for all rows.

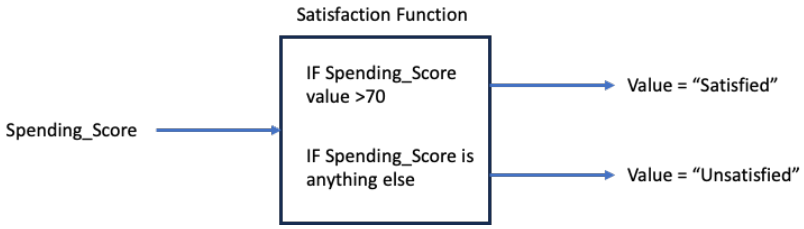
Code cell:

```
#Creates a new column with all the values equal to 1
data_frame['NewColumn'] = 1
data_frame.head()
```

Output cell:

	ID	Sex	Age	Salary	Spending_Score	NewColumn
0	1	1	19.0	15.0	NaN	1
1	2	1	NaN	15.0	81.0	1
2	3	0	20.0	16.0	6.0	1
3	4	0	23.0	NaN	77.0	1
4	5	0	NaN	NaN	40.0	1

However, having a variable with values that do not vary is considered useless and unnecessary. We can create a function that converts values from another column to fill in the values for the newly constructed variable. For example, let's construct a new variable, **"Customer Satisfaction",** and assume that customers with a spending score above 70 are "satisfied" while anyone else is "unsatisfied".



To build this box, we create a function in Python. A function in Python is defined by **df** followed by the name of the function. In our case, our function box is called "**Satisfaction**", and the argument it will be modifying or testing is the **"value"** of the **Spending_Score** variable for customers. Let's code this satisfaction function:

Code cell:

```
def satisfaction(value):
    if value > 70:
        return "Satisfied"
    else:
        return "Unsatisfied"
```

Now, we know that the **"value"** comes from **the Spending_Score** variable; therefore, we point (apply) the function to the **Spending_Score** variable from our dataset **data_frame**. This will convert the values in the **Spending_Score** variable to either **"Satisfied"** or **"Unsatisfied".**

Code cell:

```
data_frame['Spending_Score'].apply(satisfaction)
```

Output cell:

	Spending_Score
0	Unsatisfied
1	Satisfied
2	Unsatisfied
3	Satisfied
4	Unsatisfied
...	...
195	Satisfied
196	Unsatisfied
197	Satisfied
198	Unsatisfied
199	Satisfied

Note: Pandas truncated the output view of the rows; you are not able to see the full column values. You can undo this by applying the **set_options** function to pandas to display more rows.

Code cell:

```
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 150)
```

Now let's assign you're the new values of **Spending_ Score** to a newly constructed column "**Customer_Satisfaction**"

Code cell:

```
data_frame['Customer_Satisfaction'] = data_frame['Spending_Score'].apply(satisfaction)
data_frame.head()
```

Output cell:

	ID	Sex	Age	Salary	Spending_Score	NewColumn	Customer_Satisfaction
0	1	1	19.0	15.0	NaN	1	Unsatisfied
1	2	1	NaN	15.0	81.0	1	Satisfied
2	3	0	20.0	16.0	6.0	1	Unsatisfied
3	4	0	23.0	NaN	77.0	1	Satisfied
4	5	0	NaN	NaN	40.0	1	Unsatisfied

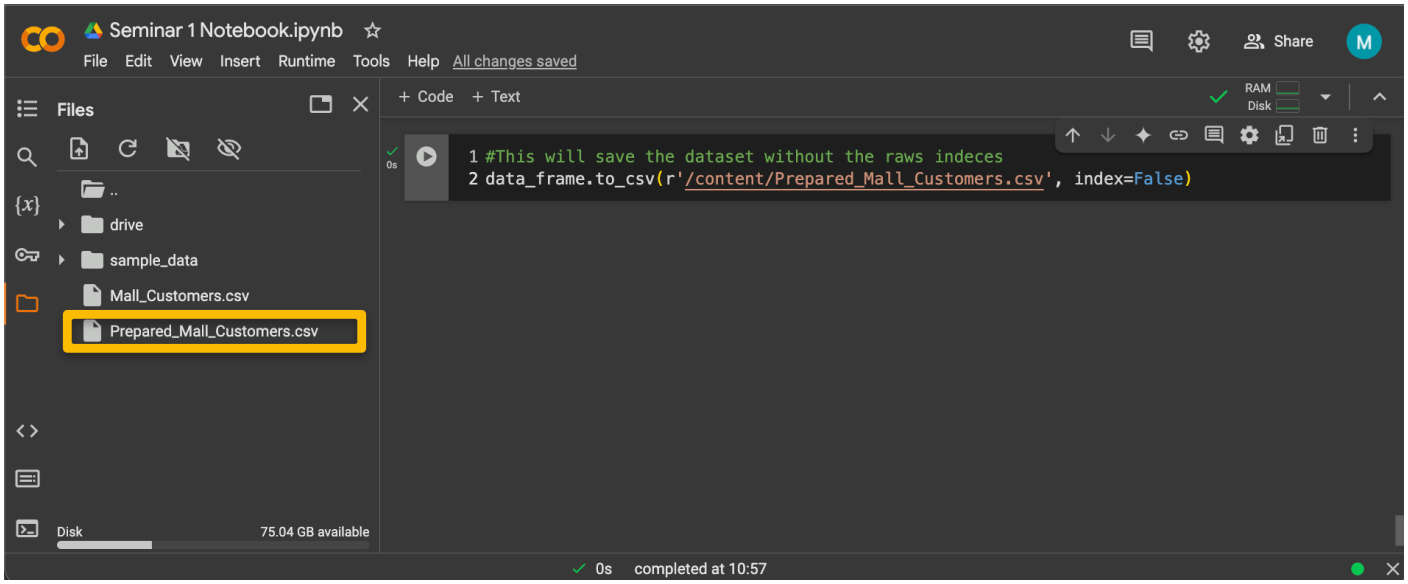
13- Saving you prepared dataset: Once you have prepared and cleaned your dataset, you should save it so that you don't need to rerun the code again. This is productive, so you can apply machine learning. To export your Pandas data_frame to a .csv file, use the template:
data_frame.to_csv(r '/Exported NewFile Path/NewFileName.csv', index=False)

You can use the same path as the original dataset, but ensure you change the file name so that you don't overwrite the original dataset with the prepared one. Let's call the clean dataset, **Prepared_Mall_Customers. Also note, that** you can save your clean data with or without the instance indexes.

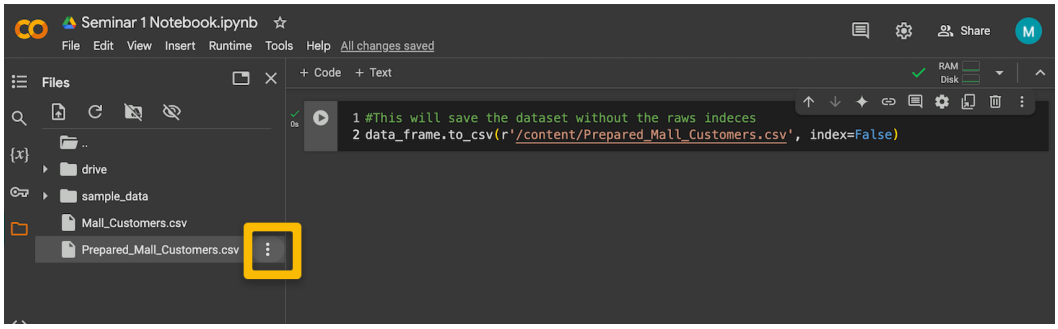
Code cell:

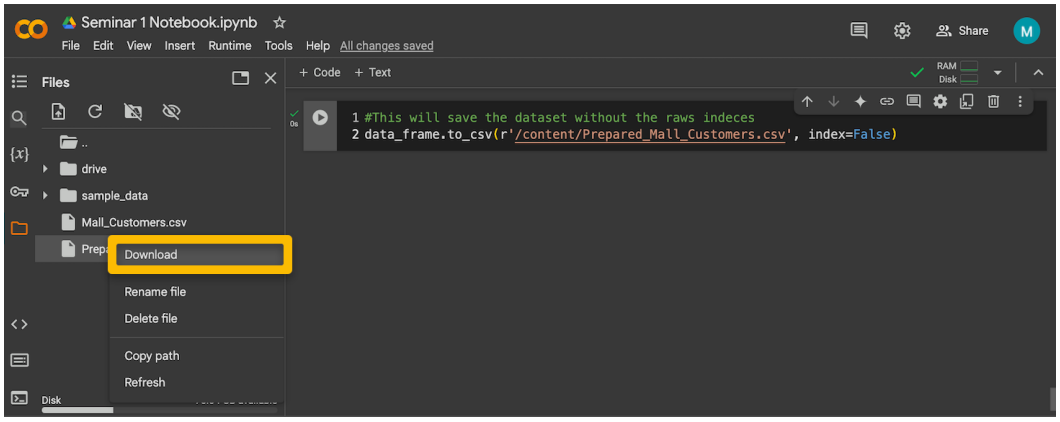
```
#This will save the dataset without the raws indeces
data_frame.to_csv(r'/content/Prepared_Mall_Customers.csv', index=False)
```

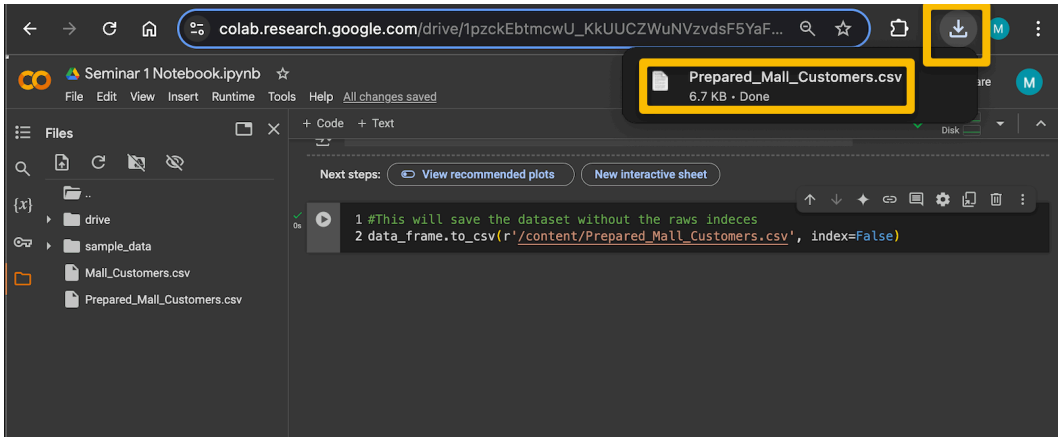
Output cell:

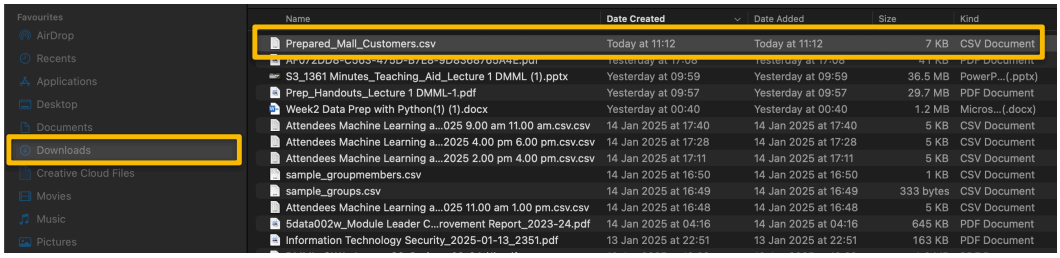


The job is not complete yet; remember, you saved your clean dataset on the Colab cloud; you must download it now onto your local machine storage; otherwise, if you terminate Colab, it won't retain any dataset; for **privacy and legality**, Google Colab deletes all used datasets upon exit. To save the clean dataset onto your local machine, from the **expanded File sidebar,** drop the **kebab menu next to your saved clean dataset,** then select **"Download"** and select the directory **path to your preferred folder** in your local machine storage. In many cases it will be downloaded to your download folder directly via your browser.









Part (D) Data Exploration and Visualisation in Google Colab

Exploring your data with visualisation: visualising your variables is a great way to spot issues in your data straight away. You can visualise your variables (columns or features) in your dataset in multiple ways:

1- Univariate data visualisations: in these plots, a single variable is visualised only; hence, the name **"uni" means just "one"**; examples of this are frequency distribution plots like histograms and bar plots. To plot a histogram, you can use a library called **plotly** and its **express** package.

Code cell:

```
import plotly.express as px
```

let's load your prepared dataset and plot a univariate histogram for the "**Spending_Score**" variable. To get an interactive histogram plot.

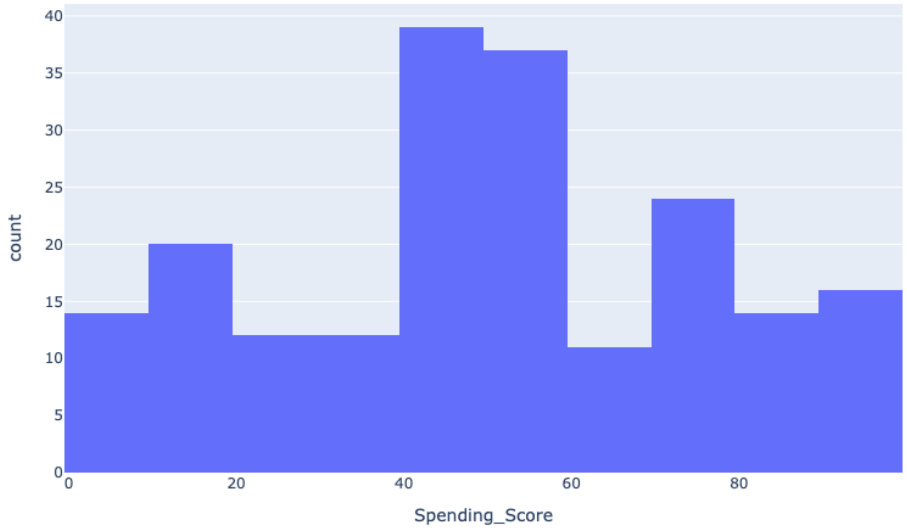
Code cell:

```
#lets load your prepared dataset
data = pd.read_csv('/content/Prepared_Mall_Customers.csv')

# Construct the histogram plot for the Spending_Score histogarm
Spending_Score_fig = px.histogram(data, x='Spending_Score')

# Display the plot
Spending_Score_fig.show()
```

Output cell:



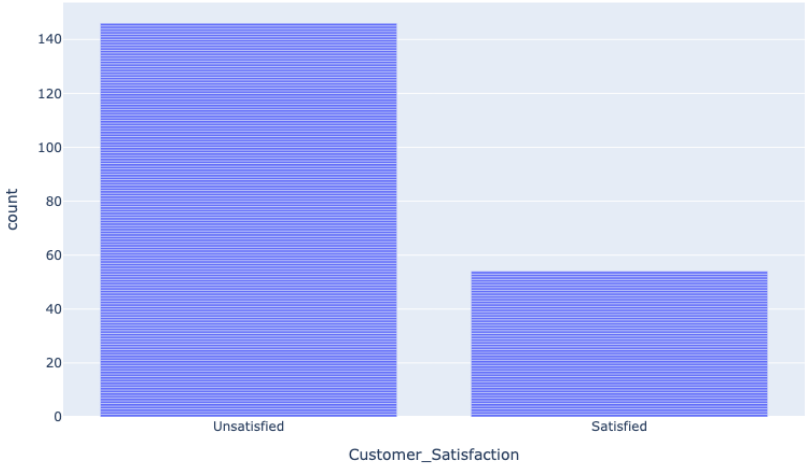
Bar plots are similar to histograms; they both visualise the frequency distribution of the variables. However, histograms are used for plotting variables whose values are numeric (continuous) like integers and floats. Bar plot are for categorical variables (Object data type variables). Let's pltestot a Bar plot for the **"Customer_Satisfaction"** variable.

Code cell:

```
# Construct the histogram plot for the Spending_Score histogarm
Customer_Satisfaction_fig = px.bar(data, x='Customer_Satisfaction')

# Display the plot
Customer_Satisfaction_fig.show()
```

Output cell:

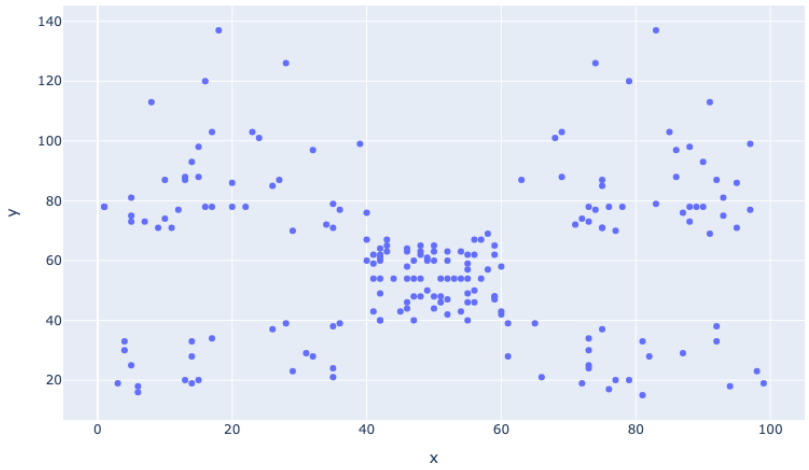


2- Bivariate data visualisations: in these plots, TWO variables are visualised only; hence, the name **"Bi" means just "TWO".** Bivariate plots combine two variables in the plot to see if there is any association between them**.** An example of bivariate plots are scatter plots. Let's see if there is an association between Customer's **"Spending_Score"** and their **"Salary"**
Do you notice any interesting observations in the plot?

Code cell:

```
Age_Salary_Association_fig = px.scatter(x=data['Spending_Score'],
y=data['Salary'])
Age_Salary_Association_fig.show()
```

Output cell:

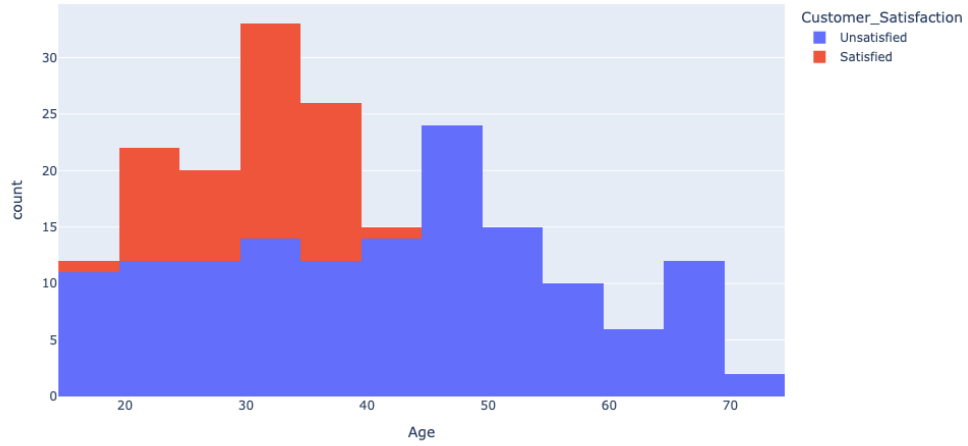


You can also create **bivariate histograms** to compare two distributions with histograms. These are called "**Stacked Histograms**". Let's see the distribution of satisfaction compared to the customer's age.

Code cell:

```
Spending_Score_Satisfaction_fig = px.histogram(data, x='Age',
color='Customer_Satisfaction')
Spending_Score_Satisfaction_fig.show()
```

Output cell:

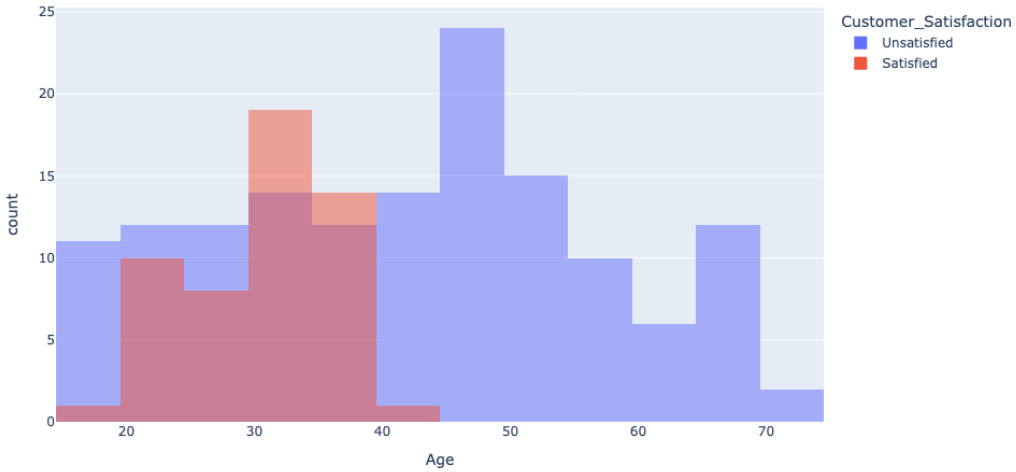


Due to overlapping distributions, it is not possible to spot interestingness between them, but what if we made the colours translucent? That would offer us a better chance to see the separability between both variables. For that, we use barmode='overlay' to create an **overlaid histogram. Can you spot any interestingness here?**

Code cell:

```
Spending_Score_Satisfaction_fig = px.histogram(data, x='Age',
color='Customer_Satisfaction', barmode='overlay')
Spending_Score_Satisfaction_fig.show()
```

Output cell:

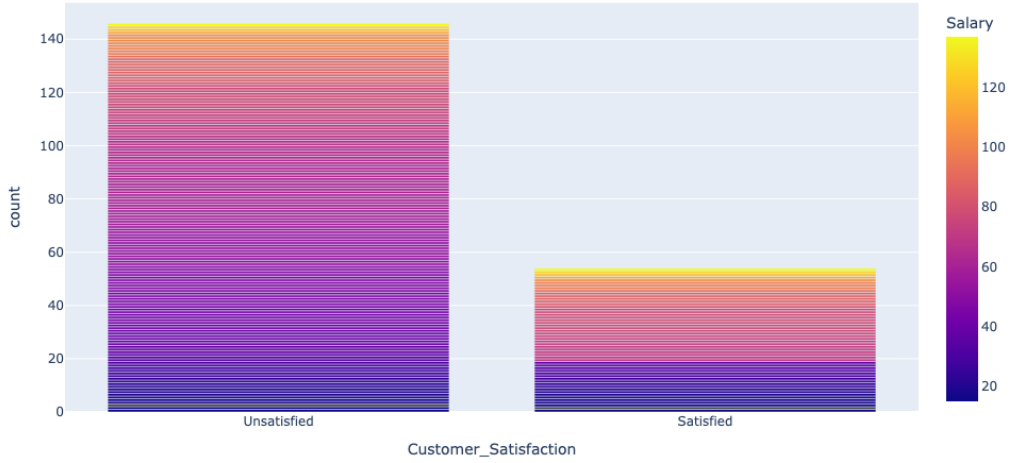


You can also create **bivariate bar plots** to compare two mixed distributions; one is for a numeric variable, and the other for a categorical variable. These are also called "**Stacked Bar Charts**" Let's see the distribution of "**Customer_Satisfaction**" compared to "**Salary**". Can you interpret what you see in this plot? What does the heat legend indicate?

Code cell:

```
Spending_Score_Satisfaction_fig = px.bar(data, x='Customer_Satisfaction',
color='Salary')
Spending_Score_Satisfaction_fig.show()
```

Output cell:



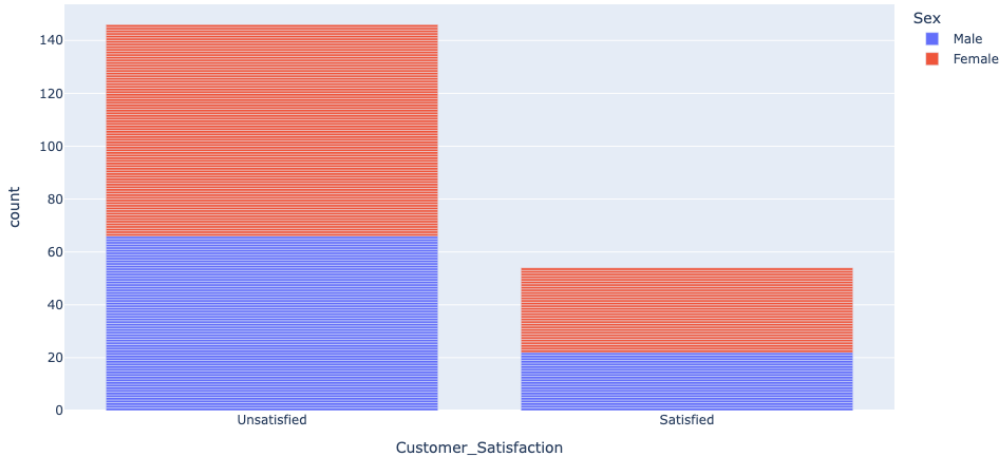
You can also create **bivariate bar plots** to compare two categorical distributions, each for a categorical (object-data type) variable. These are also called **Stacked Bar Charts.** Let's see the distribution of "**Customer_Satisfaction**" compared to "**Sex**". Why did we manipulate the "Sex" variable? What was done to it? **Can you interpret what you see in this plot? What does the legend indicate?**

Code cell:

```
data['Sex'] = data['Sex'].map({1:'Male', 0:'Female'})
```

```
Spending_Score_Satisfaction_fig = px.bar(data, x='Customer_Satisfaction',
color='Sex')
Spending_Score_Satisfaction_fig.show()
```

Output cell:

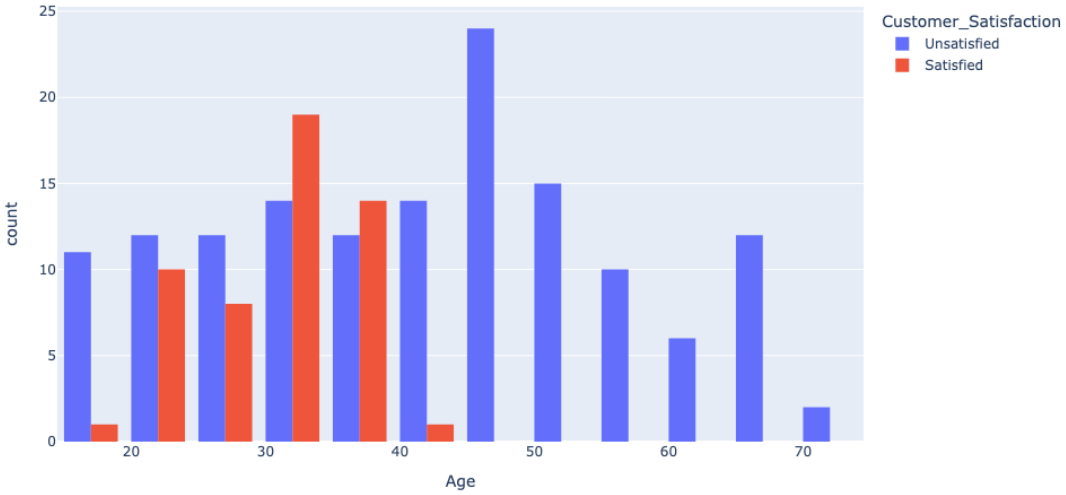


If interpreting the previous **Stacked Bar Charts** was difficult, let's try **Clustered Bar Charts.** Let's see if you can find any interestingness when visualising the distribution of "**Age**" compared to "**Customer_Satisfaction**". **Can you interpret what you see in this plot? Anything you want to flag to the marketing department? If you are to shop there, are you likely to be satisfied or unsatisfied?**

Code cell:

```
data['Sex'] = data['Sex'].map({1:'Male', 0:'Female'})
Spending_Score_Satisfaction_fig = px.histogram(data, x='Age',
color='Customer_Satisfaction', barmode="group")
Spending_Score_Satisfaction_fig.show()
```

Output cell:

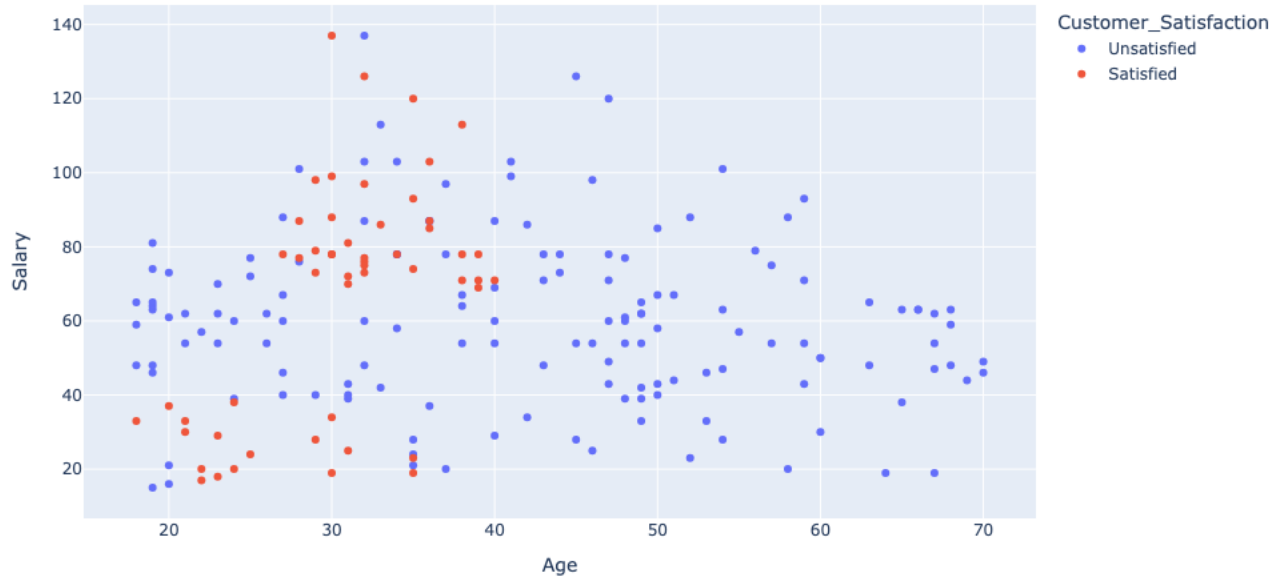


3- Multivariate plots combine more than two variables, hence the name. In these plots, you can try to find association/interestingness between all of them. Let's combine three variables, "**Customer_Satisfaction**", "**Age**", and "**Salary**" in one scatterplot! What are the salary groups for unsatisfied customers? In which age group are they? I which income range?

Code cell:

```
Age_Salary_Satisfaction_fig = px.scatter(data, x="Age", y="Salary",
color="Customer_Satisfaction")
Age_Salary_Satisfaction_fig.show()
```

Output cell:



Part (E) Finding Outliers & Extreme Values

Identifying and dealing with outliers can be tough, but it is an essential part of the data analytics process, as well as feature engineering for machine learning. So, how do we find outliers? Luckily, there are several methods for identifying outliers that are easy to execute in Python using only a few lines of code.

Outliers and Extreme values are the inconsistent values within the dataset. That means the outlier data points vary greatly from the expected values—either being much larger or significantly smaller. Outliers and Extreme values can be the result of various issues like human error in data entry or collection, faulty equipment, poor data sampling or simply these values can indicate a true anomaly or phenomenon.
A data scientist should use various techniques to visualise and identify outliers before deciding whether they should be dropped, kept, or modified. Let's load our prepared dataset. Drop any unnecessary variables from your data and display the basic stats for the RETAINED variables. In this case, **"ID"** and **"NewColumn"** are unnecessary variables.

Code cell:

```
#let's load your prepared dataset
data = pd.read_csv('/content/Prepared_Mall_Customers.csv')
data.describe().transpose()
#We used transpose to make the columns rows and the rows columns to twist
the table
```

Output cell:

	ID	Sex	Age	Salary	Spending_Score	NewColumn	Customer_Satisfaction
0	1	1	19.0	15.0	NaN	1	Unsatisfied
1	2	1	NaN	15.0	81.0	1	Satisfied
2	3	0	20.0	16.0	6.0	1	Unsatisfied
3	4	0	23.0	NaN	77.0	1	Satisfied
4	5	0	NaN	NaN	40.0	1	Unsatisfied

1- Using pandas describe() function to find outliers: After checking the data and dropping the columns, use .describe() to generate some summary statistics. Generating summary statistics is a quick way to help us determine whether or not the dataset has outliers. By looking at the stats, we know that we dropped two variables: **"ID"** and **"NewColumn"**. **But why did the "Customer_Satisfaction" variable disappear?**

Code cell:

```
# Drop unnecessary variables and rename your dataset
df = data.drop(columns=(['ID', 'NewColumn']))
df.describe()
```

Output cell:

	Sex	Age	Salary	Spending_Score
count	200.000000	197.000000	198.000000	199.000000
mean	0.440000	39.000000	61.005051	50.256281
std	0.497633	14.002915	26.017857	25.876350
min	0.000000	18.000000	15.000000	1.000000
25%	0.000000	29.000000	42.250000	34.500000
50%	0.000000	36.000000	62.000000	50.000000
75%	1.000000	49.000000	78.000000	73.000000
max	1.000000	70.000000	137.000000	99.000000

As we can see, the "Salary" columns have outliers. For example, the max "Salary" is $137k, more than twice its mean, which is $61K. The mean is sensitive to outliers. In this dataset, finding outliers by looking at basic stats is not straightforward, but in other datasets, if the mean is so small compared to the max value, this is an indication that the max value is an outlier. As we explore the data using additional methods, we can decide how to handle the outliers.

2- Using data visualisation to find outliers to find outliers: Several different visualisations will help us understand the data and the outliers. The type of plot you pick will depend on the number of variables you're analysing. These are a few of the most popular visualization methods for finding outliers in data:

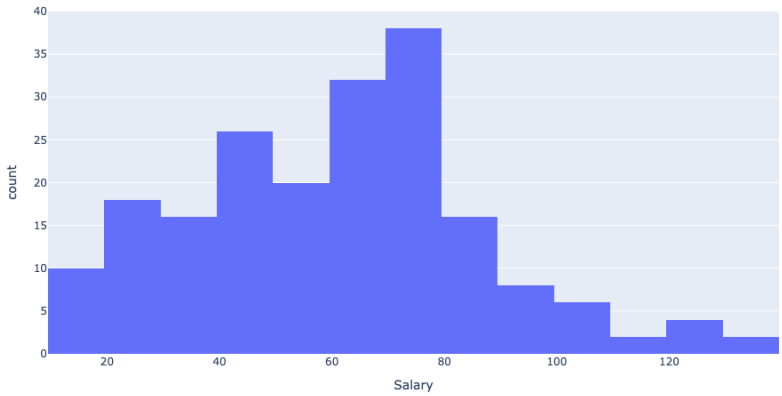
- **Histogram**
- **Box plot**
- **Scatter plot**

a) Using a histogram, we can see how the data is distributed. Having data that follows a normal distribution is necessary for some of the statistical techniques used to detect outliers.

Code cell:

```
Salary_fig = px.histogram(df, x='Salary')
Salary_fig.show()
```

Output cell:



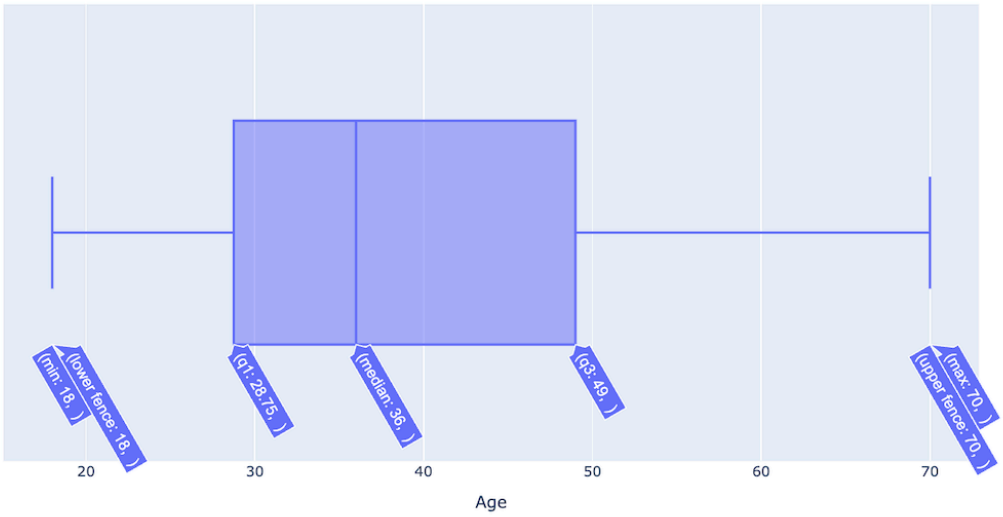
Do you think that outliers could be found at the higher end of the tail of the distribution of Salary?

B) Using a box plot, a box plot allows us to identify the univariate outliers, or outliers for one variable. Box plots are useful because they show minimum and maximum values, the median, and the interquartile range of the data. In the chart, the outliers are shown as points, which makes them easy to see. Let's plot the box plot for **"Age "** and **"Salary".**

Code cell:

```
Age_fig = px.box(df, x='Age')
Age_fig.show()
```

Output cell:



Code cell:

```
Salary_fig = px.box(df, x='Salary')
Salary_fig.show()
```

Output cell:

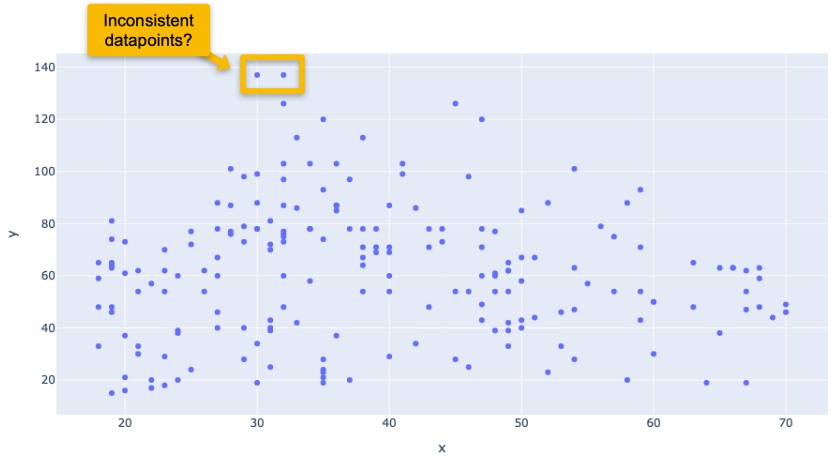
As we can see, there is an outlier value in **"Salary"** but not in **"Age"**. Above the box and upper fence is a point showing outliers. Since the chart is interactive, we can zoom in to get a better view of the box and points, and we can hover the mouse on the box to view the box plot values.

C) Using a scatter plot, using a Scatter plot, it is possible to review multivariate outliers, or the outliers that exist in two or more variables.

Code cell:

```
Age_Salary_Scatter_fig = px.scatter(x=df['Age'], y=df['Salary'])
Age_Salary_Scatter_fig.show()
```

Output cell:



Although by looking at these two data points, they are almost far away from other data points, you cannot be sure these are outliers. Keep in mind they follow the box plot results each has a salary of $137K.

3- Using IQR statistical method to find outliers to find outliers: Since the data doesn't follow a normal distribution, we will calculate the outlier data points using the statistical method called interquartile range (IQR). Using the IQR, the outlier data points are the ones falling below Q1–1.5 IQR or above Q3 + 1.5 IQR. Q1 is the 25th percentile, Q3 is the 75th percentile of the dataset, and IQR represents the interquartile range calculated by Q3 minus Q1 (Q3–Q1).

Using the convenient **pandas .quantile() function**, we can create a **simple Python function** that takes in our column from the data frame and outputs the outliers:

Code cell:

```
def find_outliers_IQR(df):
  q1=df.quantile(0.25)
  q3=df.quantile(0.75)
  IQR=q3-q1
  outliers = df[((df<(q1-1.5*IQR))|(df>(q3+1.5*IQR)))]
  return outliers
```

Notice using .quantile() we can define Q1 and Q3. Next, we calculate IQR, and then we use the values to find the outliers in the data frame. Since it takes a data frame, we can input one or multiple columns at a time. Let's find the outlier customers in **"Salary"**

Code cell:

```
outliers = find_outliers_IQR(df['Salary'])
print("number of outliers: "+ str(len(outliers)))
outliers
```

Output cell:

	Salary
198	137.0
199	137.0

dtype: float64

Using the IQR method, we find 2 Salary outliers in the dataset. Their indexes are 198 and 199. If you decide to drop these two outliers, you can use the drop function to drop the two outlier customers.

Code cell:

```
df.drop(df.index[[199,198]], inplace=True)
```

To verify the successful removal of outliers, check for outliers again in the same variable or you can check the maximum Salary value in the summary stats again to make sure we no longer see $137k as a maximum value.

Code cell:

```
outliers = find_outliers_IQR(df['Salary'])
print("number of outliers: "+ str(len(outliers)))
outliers
```

Output cell:

```
number of outliers: 0
```

Salary

dtype: float64

Code cell:

```
df.describe().transpose()
```

Output cell:

	count	mean	std	min	25%	50%	75%	max
Sex	198.0	0.434343	0.496927	0.0	0.0	0.0	1.0	1.0
Age	195.0	39.082051	14.050844	18.0	28.5	36.0	49.0	70.0
Salary	196.0	60.229592	24.980676	15.0	42.0	61.5	78.0	126.0
Spending_Score	197.0	50.253807	25.799987	1.0	35.0	50.0	73.0	99.0

Part (F) Mitigating Missing Data

1- Find missing values in the dataset: The **isnull( )** detects the missing values and returns a Boolean object indicating if the values are NA. The values which are none or empty get mapped to **True** values and not null values get mapped to false values

Code cell:

```
df.isnull( )
```

Output cell:

	Sex	Age	Salary	Spending_Score	Customer_Satisfaction
0	False	False	False	True	False
1	False	True	False	False	False
2	False	False	False	False	False
3	False	False	True	False	False
4	False	True	True	False	False
5	False	False	False	False	False
6	False	True	False	False	False
7	False	False	False	False	False
8	False	False	False	False	False
9	False	False	False	False	False
10	False	False	False	False	False
11	False	False	False	False	False
12	False	False	False	False	False
13	False	False	False	False	False

2- To find out the number of missing values in the dataset or the portion of them, use data_frame.isnull( ).sum( ). In the below example, the dataset doesn't contain any null values. Hence, each column's output is 0.

Code cell:

```
#To find the percentage of missing data per variable
df.isna().sum()/len(data_frame)*100
```

Output cell:

Sex	0.000000
Age	1.515152
Salary	1.010101
Spending_Score	0.505051
Customer_Satisfaction	0.000000

dtype: float64

3- One way to deal with missing values is to delete records containing missing values, also known as **complete case analysis**, but one must pay attention to the portion of missing not to be excessive, excessive removal of missing data can bias your analysis. Typically, the removal of 5% of data points with missing data values or less is acceptable for complete case analysis. To remove instances with missing data, we can use the **dropna()** function

Code cell:

```
df_Complete_Case = data_frame.dropna()
df_Complete_Case
```

Output cell:

	ID	Sex	Age	Salary	Spending_Score	NewColumn	Customer_Satisfaction
2	3	0	20.0	16.0	6.0	1	Unsatisfied
5	6	0	22.0	17.0	76.0	1	Satisfied
7	8	0	23.0	18.0	94.0	1	Satisfied
8	9	1	64.0	19.0	3.0	1	Unsatisfied
9	10	0	30.0	19.0	72.0	1	Satisfied
10	11	1	67.0	19.0	14.0	1	Unsatisfied
11	12	0	35.0	19.0	99.0	1	Satisfied
12	13	0	58.0	20.0	15.0	1	Unsatisfied
13	14	0	24.0	20.0	77.0	1	Satisfied
14	15	1	37.0	20.0	13.0	1	Unsatisfied

By checking the index for the remaining instances, you notice some missing indexes due to the removal of rows with missing data.

4- Simple imputation with the mean, median or mode for missing values can also be done following the same 5% condition for complete case analysis. To perform a simple imputation with the mean for a variable, you have to calculate the mean for that variable. You can impute a single variable at a time or multiple at once.

Mean_VariableName = data['Variable Name'].mean()
data['Variable Name'].fillna(Mean_VariableName, inplace=True)

Code cell:

```
Mean_Salary = df['Salary'].mean()
Mean_Spending_Score = df['Spending_Score'].mean()
Mean_Age = df['Age'].mean()

df['Salary'].fillna(Mean_Salary, inplace=True)
df['Spending_Score'].fillna(Mean_Spending_Score, inplace=True)
df['Age'].fillna(Mean_Age, inplace=True)
```

Code cell:

```
#To find the percentage of missing data per variable
df.isna().sum()/len(df)*100
```

Output cell:

Sex	0.0
Age	0.0
Salary	0.0
Spending_Score	0.0
Customer_Satisfaction	0.0

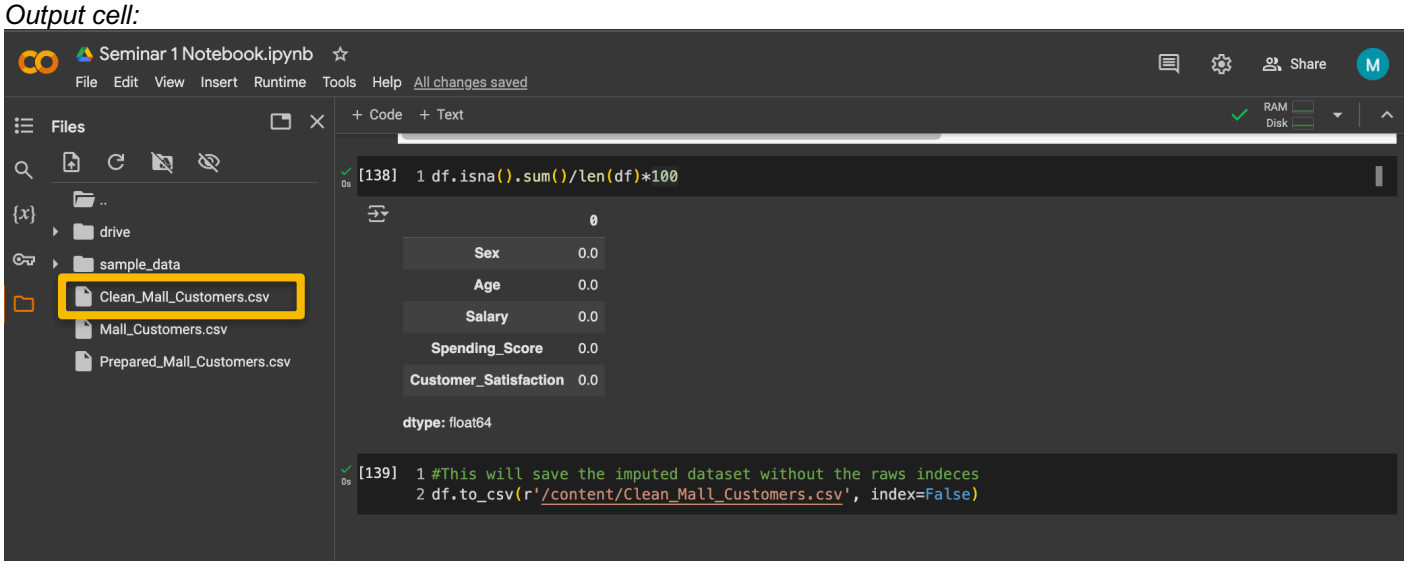
dtype: float64

Now save your clean dataset ready for processing and remember to download it to your local machine.

Code cell:

```
#This will save the imputed dataset without the row index
df.to_csv(r'/content/Clean_Mall_Customers.csv', index=False)
```

Output cell:



Part (G) Data Scaling

In this tutorial, we'll study several data scaling and normalisation techniques in Python using both sklearn and conventional programming, and we'll share lots of examples. Here are the data scaling techniques we're going to learn in this tutorial:

Standard Scaling (Standardization or Z Score)
Minimum – Maximum Scaling (Normalization).
However, there are many other methods of scaling you should consider exploring at your own time, including, Mean Scaling, Maximum Absolute Scaling, Median and Quantile Scaling, Robust Scaler and Log Scaling

1- Data Merge: Load your **Mall_Customers.csv** dataset. The Mall management team contacted their customers and obtained additional data about the same customers, stored in a different CSV file, **Mall_Customers_Additional.csv**. Now, you need to merge both files for analysis. Start by loading both files.

Code cell:

```
import pandas as pd
#let's load your prepared dataset
df1 = pd.read_csv('/content/Mall_Customers.csv')
df1.head()
```

Output cell:

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19.0	15.0	NaN
1	2	Male	NaN	15.0	81.0
2	3	Female	20.0	16.0	6.0
3	4	Female	23.0	NaN	77.0
4	5	Female	NaN	NaN	40.0

Code cell:

```
import pandas as pd
#let's load your prepared dataset
df2 = pd.read_csv('/content/Mall_Customers_Additional.csv')
df2.head()
```

Output cell:

	CustomerID	Travel_distance_meters	Dependents
0	1	5040	5
1	2	3492	3
2	3	2795	0
3	4	6315	0
4	5	1978	3

We can join both data frames on **the "CustomerID"** column to have one new dataset **"Merged_Mall_df"**

Code cell:

```
Merged_Mall_df = df1.merge(df2, on='CustomerID')
Merged_Mall_df.head()
```

Output cell:

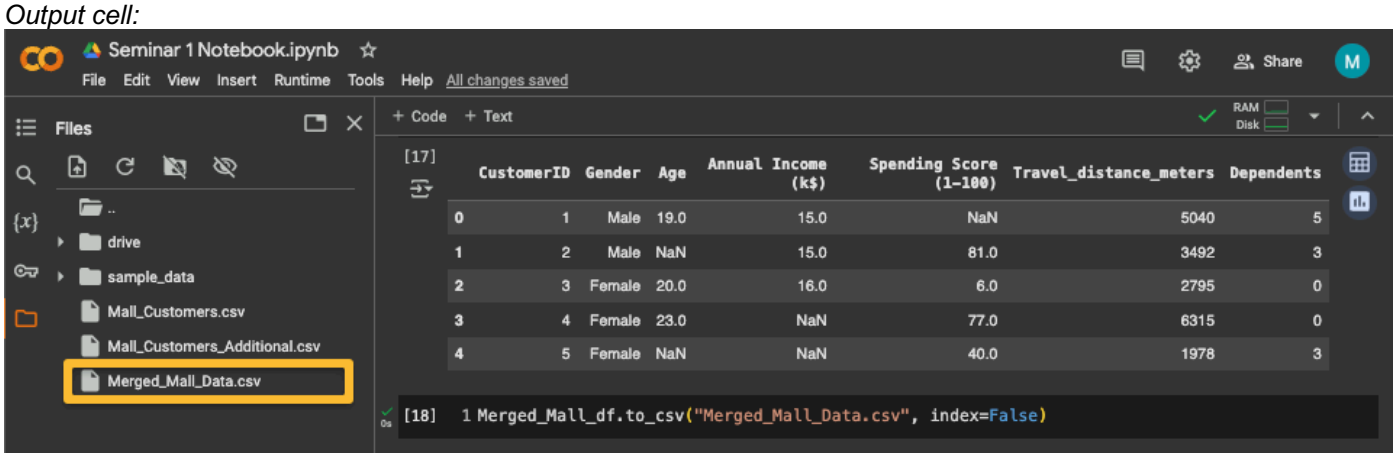
	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)	Travel_distance_meters	Dependents
0	1	Male	19.0	15.0	NaN	5040	5
1	2	Male	NaN	15.0	81.0	3492	3
2	3	Female	20.0	16.0	6.0	2795	0
3	4	Female	23.0	NaN	77.0	6315	0
4	5	Female	NaN	NaN	40.0	1978	3

You can save the newly merged dataset as a new .csv file, name it **Merged_Mall_Data.csv,** then download it on your local machine.

Code cell:

```
Merged_Mall_df.to_csv("Merged_Mall_Data.csv", index=False)
```

Output cell:



2- Data Magnitude Variations: Load your new merged dataset and examine its basic stats using the describe() method. You can see from the above output that our dataset now contains just FIVE numeric columns, excluding the **CustomerID** columns. Notice the magnitude (scale) of data for the column is very different among the **"Age"**, **"Travel_Distance_meters",** and **"Dependents"** columns. You can easily notice that by looking at the **mean value** for each column.

Code cell:

```
import pandas as pd
#let's load your prepared dataset
df = pd.read_csv('/content/Merged_Mall_Data.csv')
df.describe()
```

Output cell:

	CustomerID	Age	Annual Income (k$)	Spending Score (1-100)	Travel_distance_meters	Dependents
count	200.000000	197.000000	198.000000	199.000000	200.000000	200.000000
mean	100.500000	39.000000	61.005051	50.256281	4915.625000	2.525000
std	57.879185	14.002915	26.017857	25.876350	2796.210913	1.820976
min	1.000000	18.000000	15.000000	1.000000	174.000000	0.000000
25%	50.750000	29.000000	42.250000	34.500000	2324.000000	1.000000
50%	100.500000	36.000000	62.000000	50.000000	4958.000000	3.000000
75%	150.250000	49.000000	78.000000	73.000000	7243.750000	4.000000
max	200.000000	70.000000	137.000000	99.000000	9994.000000	5.000000

When observing the magnitude variations, **"Age"** values vary in double digits, **"Travel_Distance_meters"** values vary in four digits, and **"Dependents"** values vary in a single digit. columns. You can easily notice that by looking at the **mean value** for each column. This can affect the performance of certain machine-learning algorithms.

The above output confirms our three columns are not scaled. The mean, minimum and maximum values, and even the standard deviation values for all three columns are very different.
This unscaled dataset is not suitable for processing by some statistical algorithms. We need to scale this data so that's exactly what we'll do. We'll show you different types of data scaling techniques in action.

3- Standard Scaling (Standardisation AKA Z-Score): Several machine learning algorithms, like linear regression support vector machines (SVMs), assume all the features in a dataset are centred around 0 and have unit variances. It's a common practice to apply standard scaling to your data before training these machine learning algorithms on your dataset.

a) In standard scaling, a feature is scaled by subtracting the mean from all the data points and dividing the resultant values by the standard deviation of the data. To perform this in Python, you must import the relevant library and packages. Then, to apply the **StandardScaler function** to the data, we use the **fit_transform** method. Check the transformed values for the scaled data to observe the value becoming ratios of the same magnitude.

Formula for the z-score:

$$z = \frac{\text{value} - \text{mean}}{\text{standard deviation}}$$

Code cell:

```
from sklearn.preprocessing import StandardScaler
#drop unnecessary numeric and non-numeric variables
df_numeric = df.drop(columns=(['CustomerID', 'Gender']))
ss = StandardScaler()
df_scaled = ss.fit_transform(df_numeric)
df_scaled
```

Output cell:

```
array([[-1.43191295e+00, -1.77269273e+00,            nan,
         4.45914533e-02,  1.36257170e+00],
       [           nan, -1.77269273e+00,  1.19109751e+00,
        -5.10404082e-01,  2.61503659e-01],
       [-1.36031730e+00, -1.73416016e+00, -1.71461193e+00,
        -7.60295483e-01, -1.39009840e+00],
       [-1.14553036e+00,            nan,  1.03612634e+00,
         5.01709869e-01, -1.39009840e+00],
       [           nan,            nan, -3.97356984e-01,
        -1.05320979e+00,  2.61503659e-01],
       [-1.21712600e+00, -1.69562759e+00,  9.97383551e-01,
         1.05316498e-02, -1.39009840e+00],
       [           nan, -1.65709503e+00, -1.71461193e+00,
        -1.77693580e-01, -1.39009840e+00],
       [-1.14553036e+00, -1.65709503e+00,  1.69475382e+00,
         8.02870237e-01, -2.89030360e-01],
       [ 1.78989118e+00, -1.61856246e+00, -1.83084031e+00,
        -9.20062851e-02, -1.39009840e+00],
       [-6.44360826e-01, -1.61856246e+00,  8.42412380e-01,
         1.53582825e-01,  1.36257170e+00],
       [ 2.00467813e+00, -1.61856246e+00, -1.40466959e+00,
         5.48318021e-01,  2.61503659e-01],
       [-2.86382589e-01, -1.61856246e+00,  1.88846778e+00,
        -3.83845023e-01,  1.36257170e+00],
       [ 1.36031730e+00, -1.58002989e+00, -1.36592680e+00,
        -9.95352943e-02,  2.61503659e-01],
       [-1.07393471e+00, -1.58002989e+00,  1.03612634e+00,
        -6.62776887e-01,  1.36257170e+00],
       [-1.43191295e-01, -1.58002989e+00, -1.44341238e+00,
         8.09323674e-01, -8.39564379e-01],
       [-1.21712600e+00, -1.58002989e+00,  1.11361193e+00,
         8.34061847e-01, -2.89030360e-01],
       [-2.86382589e-01, -1.54149732e+00, -5.91070947e-01,
        -9.99431156e-01,  8.12037678e-01],
       [-1.36031730e+00, -1.54149732e+00,  6.09955624e-01,
```

b) The **fit_transform()** method returns a NumPy array, which you can convert to a Pandas Data frame by passing the array to the Data frame class constructor in pandas. The following script makes the conversion and prints the header for our newly scaled dataset.

Code cell:

```
df_scaled = pd.DataFrame(df_scaled,columns = df_numeric.columns)
df_scaled.head()
```

Output cell:

	Age	Annual Income (k$)	Spending Score (1-100)	Travel_distance_meters	Dependents
0	-1.431913	-1.772693	NaN	0.044591	1.362572
1	NaN	-1.772693	1.191098	-0.510404	0.261504
2	-1.360317	-1.734160	-1.714612	-0.760295	-1.390098
3	-1.145530	NaN	1.036126	0.501710	-1.390098
4	NaN	NaN	-0.397357	-1.053210	0.261504

c) Check the basic statistics now for the scaled (standardised) data variables.

Code cell:

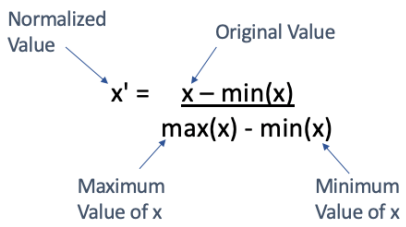
```
df_scaled.describe()
```

Output cell:

	Age	Annual Income (k$)	Spending Score (1-100)	Travel_distance_meters	Dependents
count	1.970000e+02	198.000000	1.990000e+02	2.000000e+02	2.000000e+02
mean	9.017040e-18	0.000000	5.802171e-17	-8.881784e-18	7.105427e-17
std	1.002548e+00	1.002535	1.002522e+00	1.002509e+00	1.002509e+00
min	-1.503509e+00	-1.772693	-1.908326e+00	-1.699988e+00	-1.390098e+00
25%	-7.159565e-01	-0.722680	-6.104423e-01	-9.291604e-01	-8.395644e-01
50%	-2.147869e-01	0.038338	-9.929057e-03	1.519246e-02	2.615037e-01
75%	7.159565e-01	0.654859	8.811552e-01	8.346893e-01	8.120377e-01
max	2.219465e+00	2.928281	1.888468e+00	1.820721e+00	1.362572e+00

4- Minimum-maximum Scaling (Normalisation): Min/Max scaling normalises the data between 0 and 1 by subtracting the overall minimum value from each data point and dividing the result by the difference between the minimum and maximum values.
The Min/Max scaler is commonly used for data scaling when the maximum and minimum values for data points are known. For instance, you can use the min/max scaler to normalise image pixels having values between 0 and 255.

a) You'll want to use the **MinMaxScaler** class from the **sklearn.preprocessing** module to perform min/max scaling. The fit_transform method of the class performs the min/max scaling on the input Pandas Data frame, as shown below:



Code cell:

```
from sklearn.preprocessing import MinMaxScaler
mms = MinMaxScaler()
df_mms = mms.fit_transform(df_numeric)
df_mms
```

Output cell:

```
[1.92307692e-01, 5.90163934e-01, 7.55102041e-01, 1.23217923e-02,
 1.00000000e+00],
[3.46153846e-01, 5.90163934e-01, 9.18367347e-02, 5.24134420e-01,
 1.00000000e+00],
[3.46153846e-01, 5.90163934e-01, 9.28571429e-01, 3.80244399e-01,
 1.00000000e+00],
[6.53846154e-01, 5.98360656e-01, 1.22448980e-01, 5.49389002e-01,
 6.00000000e-01],
[2.30769231e-01, 5.98360656e-01, 8.67346939e-01, 8.01629328e-01,
 1.00000000e+00],
[7.69230769e-01, 5.98360656e-01, 1.42857143e-01, 3.47352342e-01,
 2.00000000e-01],
[1.73076923e-01, 5.98360656e-01, 6.93877551e-01, 2.77189409e-01,
 1.00000000e+00],
[7.88461538e-01, 6.39344262e-01, 1.32653061e-01, 3.10590631e-01,
 0.00000000e+00],
[3.26923077e-01, 6.39344262e-01, 9.08163265e-01, 7.63136456e-01,
 8.00000000e-01],
[3.65384615e-01, 6.72131148e-01, 3.16326531e-01, 2.45010183e-01,
 6.00000000e-01],
[2.69230769e-01, 6.72131148e-01, 8.67346939e-01, 1.89409369e-01,
 8.00000000e-01],
[5.38461538e-01, 6.80327869e-01, 1.42857143e-01, 6.06924644e-02,
 1.00000000e+00],
[2.11538462e-01, 6.80327869e-01, 8.87755102e-01, 5.31466395e-01,
 2.00000000e-01],
[4.42307692e-01, 6.88524590e-01, 3.87755102e-01, 1.61812627e-01,
 1.00000000e+00],
[2.30769231e-01, 6.88524590e-01, 9.79591837e-01, 3.47556008e-01,
```

b) The **fit_transform()** method returns a NumPy array, which you can convert to a Pandas Data frame by passing the array to the Data frame class constructor in Pandas. The following script makes the conversion and prints the header for our newly scaled dataset.

Code cell:

```
df_mms = pd.DataFrame(df_mms,columns = df_numeric.columns)
df_mms.head()
```

Output cell:

	Age	Annual Income (k$)	Spending Score (1-100)	Travel_distance_meters	Dependents
0	0.019231	0.000000	NaN	0.495519	1.0
1	NaN	0.000000	0.816327	0.337882	0.6
2	0.038462	0.008197	0.051020	0.266904	0.0
3	0.096154	NaN	0.775510	0.625356	0.0
4	NaN	NaN	0.397959	0.183707	0.6

c) Check the basic statistics now for the scaled (standardised) data variables. Notice that all the variables have a minimum value of 0 and a maximum of 1

Code cell:

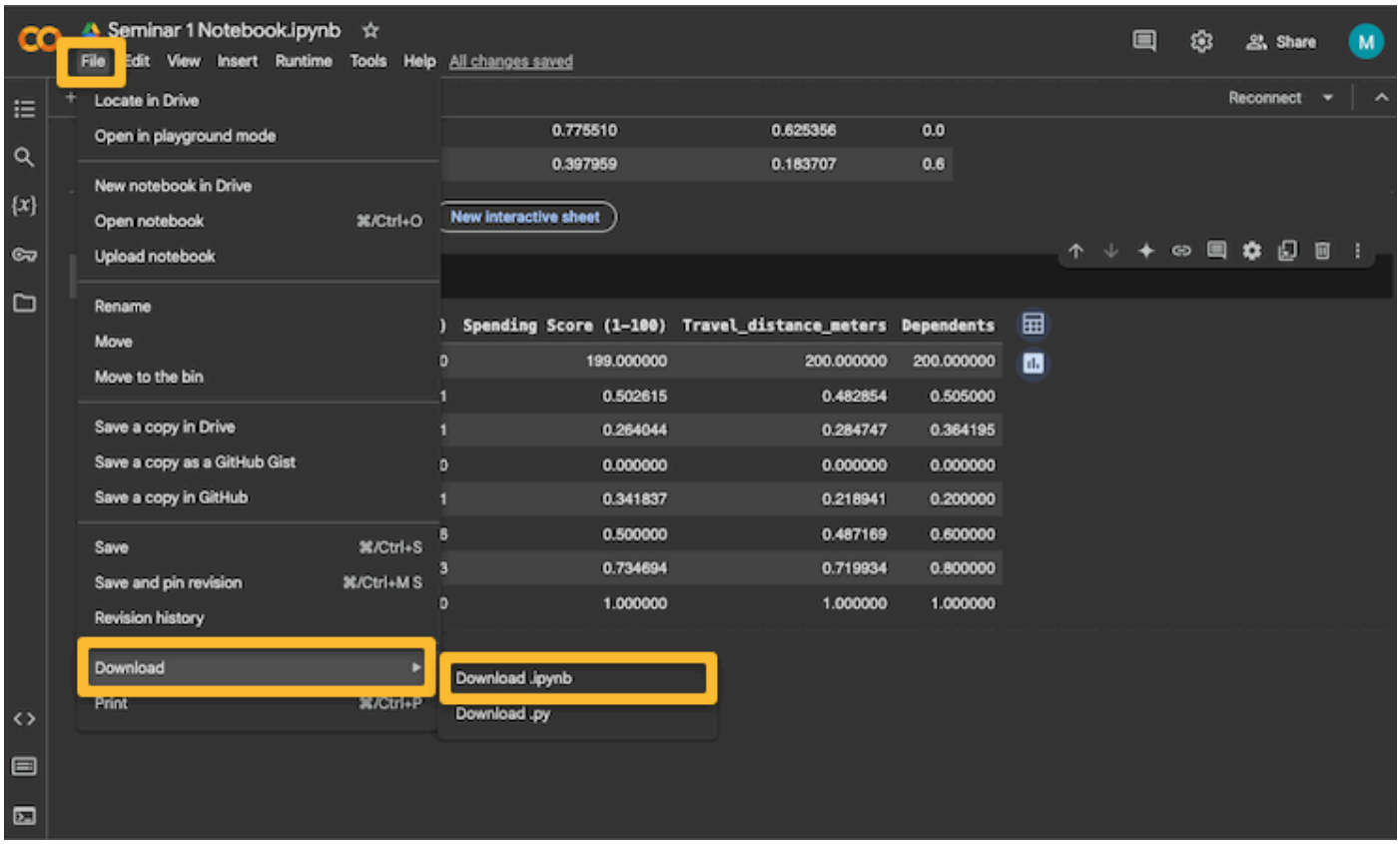
```
df_mms.describe()
```

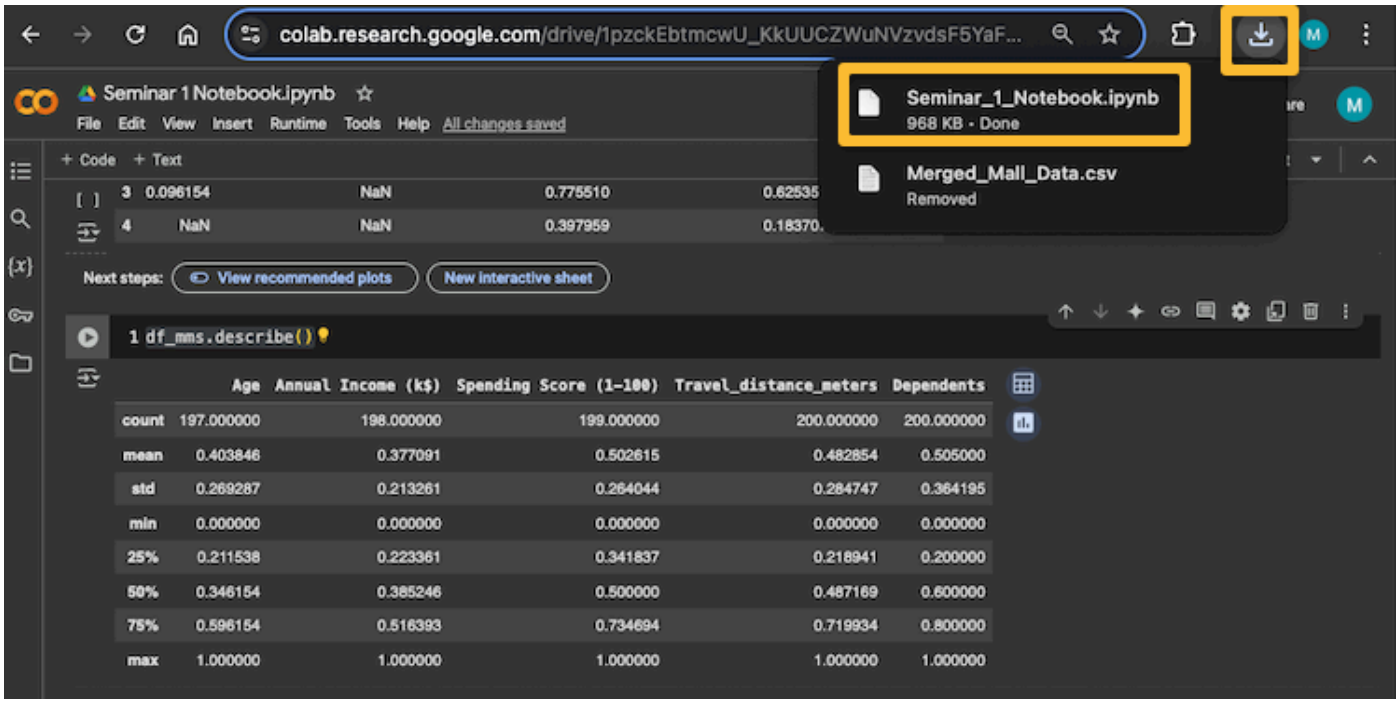
Output cell:

	Age	Annual Income (k$)	Spending Score (1-100)	Travel_distance_meters	Dependents
count	197.000000	198.000000	199.000000	200.000000	200.000000
mean	0.403846	0.377091	0.502615	0.482854	0.505000
std	0.269287	0.213261	0.264044	0.284747	0.364195
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.211538	0.223361	0.341837	0.218941	0.200000
50%	0.346154	0.385246	0.500000	0.487169	0.600000
75%	0.596154	0.516393	0.734694	0.719934	0.800000
max	1.000000	1.000000	1.000000	1.000000	1.000000

Part (H) Saving Your Colab Notebook

Saving your Colab notebook is one of the most important tasks you must do. This is to share your experimental results with other scientists and code with others. Also, it is important to save your coursework Python work as a Colab Notebook. Follow the following steps in the screenshots to Download your Colab Notebook .ipynb.





Appendix (A)

Google Colab Keyboard Shortcuts

- To move the cell up ctrl+m K.
- To move the cell down ctrl+m J.
- To create a new cell below ctrl+m b.
- To create a new cell above ctrl+m a.
- To delete a cell ctrl+m d.
- To convert a text cell to a code cell ctrl + m + y.
- To convert a code cell to text cell ctrl + m + m (double tap m)
- To replace within cell ctrl + shift + h
- To replace within entire notebook ctrl + h
- Ctrl + Shift + p command palette
- Ctrl + M + C: Copy the selected cell.
- Ctrl + M + X: Cut the selected cell.
- Ctrl + M + V: Paste the copied/cut cell below the selected cell.
- Ctrl + M + D: Delete the selected cell.
- Ctrl + M + Z: Undo the last cell deletion (very handy if you accidentally delete something important).
- Ctrl + m + i to interrupt m
- Ctrl + m + l to toggle line numbers
- Ctrl + m + o to toggle output