Data Science Life Cycle

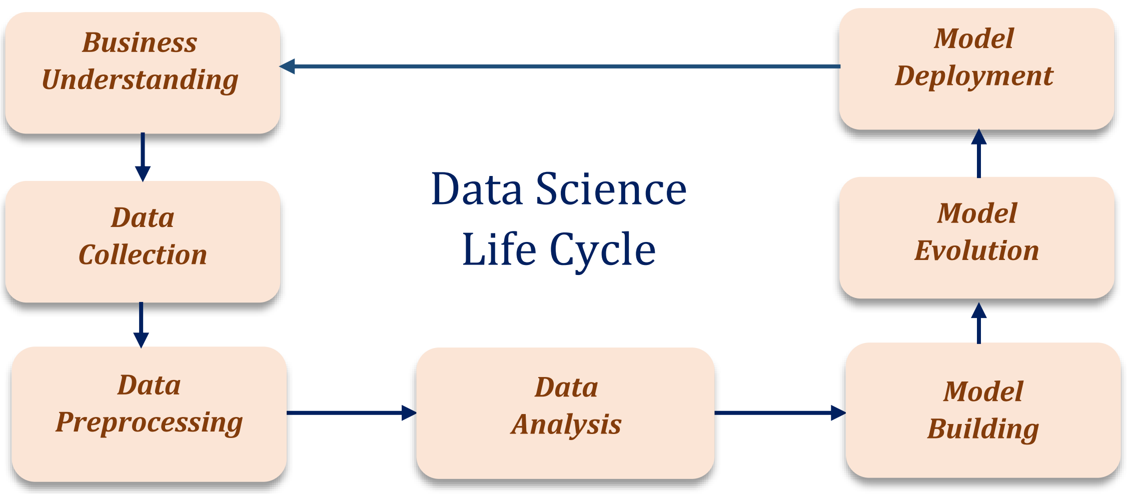
Learning Topics

✓ What is data science life cycle

1. What is Data Science Life Cycle:

Data science life cycle is a data science project flow planning that involves different types of steps in priority order.

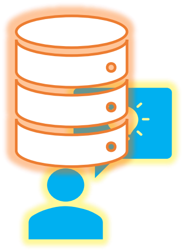
Data science life cycle is the first step in every data science projects. Not only for data science, any type of software industry's projects starts with project life cycle building by technical experts. It explains the different types of stages involved in data science project. Let's start learning about all stages one by one.



1.1. Business Understanding*:*

Understanding the business is one of the important roles in data science project. The business requirement and achieving goals are plays vital role in the project. In this stage, we collect all required information, data related to business and asking quarries about the data. Once all business-related information received then we will move to next step called data collection.

1.2. Data Collection:



Data collection is the second step in data-science life cycles. In this stage we collect all required data related to business requirement. This collected data is called **raw data**. Because this data coming from different resources and in different formats. This all data we can't use directly in ML models, because it contains many unsupportable formats and data impurities.

1.3. Data Preprocessing:

Next comes the data preprocessing stage. Right now, many preprocessing pipeline projects are going in the market. This consist of data cleaning, understand the outliers and unwanted data cleaning, removing useless columns, feature selection for correct training, feature encoding, feature scaling etc... The preprocessing is very important stage for all ML models training. A perfectly cleaned data make the perfect model. 90% of work in all ml projects is data preprocessing only remaining 10% is model building.

1.4. Data Analysis:

This concept includes the explanation of data, understanding the characteristics of data by presenting in graphical and numerical representation. The data analysis helps to select correct model. Analyzing information provides the hidden knowledge in the data and business-related information also.

1.5. Model Building:

Will select a suitable ML model based on the data analysis report. Once the model got selected, then will provide the training data to the model for training purpose.

1.6. Model Evolution:

Model evolution is nothing but testing the model. Once model got trained the next step is testing how accurate model got generated. If model gives expected accuracy, then model is ready for deployment.

1.7. Model Deployment:

Model deployment is nothing but Providing availability of the model for prediction future purpose.

CAPTER – 2

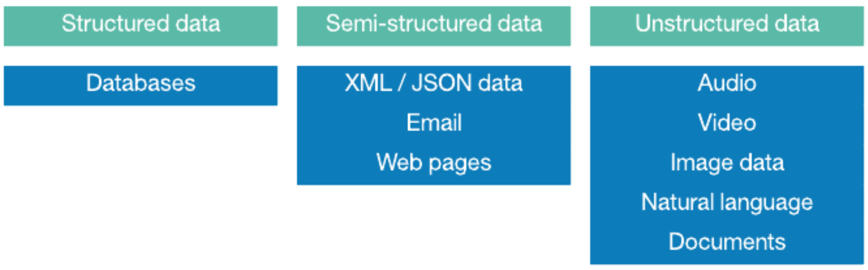
Types Of Data Based on Structure

Learning Topics

- ✓ Structured data
- ✓ Unstructured data
- ✓ Semi structured data

In data science, the data will be divided into three parts based on its structure. Will learn those three types in this chapter.

1. Structured Data:



Structured data	Semi-structured data	Unstructured data
Databases	XML / JSON data	Audio
	Email	Video
	Web pages	Image data
		Natural language
		Documents

- Structured data is usually stored with well-defined schemas such as Databases. It is generally in tabular format with column and rows that clearly define its attributes.
- It has consistent order and can be easily accessed and used by a person or a computer program.
- SQL (Structured Query language) is often used to manage structured data stored in databases.

1.1. Characteristics of Structured Data:

- Data has easily identifiable structure.
- Data is stored in the form of rows and columns.
- Data is well organized so, Definition, Format and Meaning of data is explicitly known
- Easy to access and query, so data can be easily used by other programs
- Easy to analyze and process

1.2. Sources of Structured Data:

- SQL Databases
- Spreadsheets such as Excel

1.3. Advantages of Structured Data:

- It has a well-defined structure that helps in easy storage and access of data.
- Each data record has index number and column name which helps in easy access.
- Data mining is easy i.e., knowledge can be easily extracted from data.
- Operations such as Updating and deleting is easy due to well-structured form of data.
- Business Intelligence operations such as Data warehousing can be easily undertaken.
- Easily scalable in case there is an increment of data.

- Ensuring security to data is easy

Note: Only 20% of structured data is available in world remain data is not in structured. This is the main challenge of every data scientist.

2. Unstructured Data:

Unstructured data is the data which does not identifiable structure such that it cannot be used by a computer program easily and not easily understandable. Unstructured data is can't organize in a pre-defined manner.

2.1. Characteristics:

- Data has any structure.
- Data cannot be stored in the form of rows and columns as in Databases.
- Data does not follow any semantic or rules
- Data don't have any particular format or sequence
- Very hard to identify the its structure.
- Due to lack of identifiable structure, it cannot use by computer programs easily

2.2. Sources of Unstructured Data:

- Images (JPEG, GIF, PNG, etc.), Videos
- Reports, Memos
- Word documents and PowerPoint presentations
- Web pages, Surveys

2.3. Advantages:

- We can give a required format or sequence to data.
- The data is not constrained by a fixed schema.
- Very Flexible due to absence of schema.
- Data is portable.
- It is very scalable.
- It can deal easily with the heterogeneity of sources.
- These types of data have a variety of business intelligence and analytics applications.

2.4. Disadvantages:

- It is difficult to store and manage due to lack of schema and structure.
- Indexing the data is difficult and not having pre-defined attributes.
- Search results are not very accurate.
- Ensuring security to data is difficult task.

2.5. Problems Faced in Storing Unstructured Data:

- It requires a lot of storage space.
- It is difficult to store videos, images, audios, etc.
- Due to unclear structure, operations like update, delete and search are very difficult.
- Storage cost is high as compared to structured data.
- Very difficult to indexing.

2.6. Possible Solution for Storing Unstructured Data:

- Unstructured data can be converted to easily manageable formats.
- Content addressable storage system (CAS) is used to store unstructured data.
- It stores data based on their metadata and a unique name is assigned to every object stored in it. The object is retrieved based on content not its location.

- Unstructured data can be stored in XML format.
- Unstructured data can be stored in RDBMS which supports BLOBs

2.7. Extracting Information from Unstructured Data:

Unstructured data do not have any structure. So, it cannot easily be interpreted by conventional algorithms. It is also difficult to tag and index unstructured data. So, extracting information from them is tough job. Here are possible solutions:

- Taxonomies or classification of data helps in organizing data in hierarchical structure. Which will make search process easy.
- Data can be stored in virtual repository and be automatically tagged. For example, Documentum.
- Use of application platforms like XOLAP.
- XOLAP helps in extracting information from e-mails and XML based documents
- Use of various data mining tools

3. Semi-structured Data:

Semi-structured data is data that does not conform to a data model but has some structure. It lacks a fixed or rigid schema. It is the data that does not reside in a rational database but that have some organizational properties that make it easier to analyze. With some processes, we can store them in the relational database.

3.1. Characteristics:

- Data does not conform to a data model but has some structure.
- Data cannot be stored in the form of rows and columns as in Databases
- Semi-structured data contains tags and elements (Metadata) which is used to group data and describe how the data is stored
- Similar entities are grouped together and organized in a hierarchy
- Entities in the same group may or may not have the same attributes or properties
- Does not contain sufficient metadata which makes automation and management of data difficult
- Size and type of the same attributes in a group may differ
- Due to lack of a well-defined structure, it cannot used by computer programs easily

3.2. Sources of Semi-Structured Data:

- E-mails
- JSON, XML, YML
- Binary executables
- TCP/IP packets
- Zipped files
- Integration of data from different sources
- Web pages

3.3. Advantages:

- The data is not constrained by a fixed schema
- Flexible i.e Schema can be easily changed.
- Data is portable
- It is possible to view structured data as semi-structured data
- Its supports users who cannot express their need in SQL
- It can deal easily with the heterogeneity of sources.

3.4. Disadvantages:

- Lack of fixed, rigid schema make it difficult in storage of the data
- Interpreting the relationship between data is difficult as there is no separation of the schema and the data.
- Queries are less efficient as compared to structured data.

3.5. Problems Faced in Storing Semi-Structured Data:

- Data usually has an irregular and partial structure. Some sources have implicit structure of data, which makes it difficult to interpret the relationship between data.
- Schema and data are usually tightly coupled i.e. they are not only linked together but are also dependent of each other. Same query may update both schema and data with the schema being updated frequently.
- Distinction between schema and data is very uncertain or unclear. This complicates the designing of structure of data
- Storage cost is high as compared to structured data

3.6. Possible Solution for Storing Semi-Structured Data:

- Data can be stored in DBMS specially designed to store semi-structured data
- XML is widely used to store and exchange semi-structured data. It allows its user to define tags and attributes to store the data in hierarchical form.
- Schema and Data are not tightly coupled in XML.
- Object Exchange Model (OEM) can be used to store and exchange semi-structured data. OEM structures data in form of graph.
- RDBMS can be used to store the data by mapping the data to relational schema and then mapping it to a table

3.7. Extracting Information from Semi-Structured Data:

Semi-structured data have different structure because of heterogeneity of the sources. Sometimes they do not contain any structure at all. This makes it difficult to tag and index. So, while extract information from them is tough job. Here are possible solutions –

- Graph based models (e.g. OEM) can be used to index semi-structured data
- Data modelling technique in OEM allows the data to be stored in graph-based model. The data in graph-based model is easier to search and index.
- XML allows data to be arranged in hierarchical order which enables the data to be indexed and searched
- Use of various data mining tools.

Read Data
From Different Data Files

Learning Topics

- ✓ Read CSV data
- ✓ Read excel data
- ✓ Read pickle data
- ✓ Read JSON data
- ✓ Read XML data

GitHub link: *Read_Data_From_Different_Data_Files*

In this chapter we will learn how to read different types of data files by using Pandas**.** The first step is reading the data from data files in every data science projects. But data is always available different types of file formats. Here the *pandas* library is help us to read all types of data files.

1. Read CSV Data:

```
import pandas as pd

data = pd.read_csv('CSV File Path',
                   usecols = [clm_1, clm_2],
                   sep=',',
                   names = [new_clm_name_1, new_clm_name_2],
                   index_col= 2,
                   header= 1)
print(data.head())
```

sep	We can mention our field separation based on the input CSV file structure.
header	This parameter is used to specify a particular row as header.
names	It is used to rename the columns.
index_col	To use a particular column as index.
usecols	Used to load some specific columns from data.

2. Read Pickle Data:

```
import pandas as pd

data = pd.read_pickle('Pickle File Path')
print(data.head())
```

3. Read Excel Data:

```
import pandas as pd

data = pd.read_excel('Excel File Path',
                     usecols = [clm_1, clm_2],
                     sheet_name ='sheet_1',
                     names = [new_clm_name_1, new_clm_name_2],
                     index_col= 2,
                     header= 1)
print(data.head())
```

We should mention renamed columns in the ***usecols*** list if we use ***usecols*** and ***names*** parameter together. When we use ***names*** parameter, it will rename the column names and if we mention old column names in the ***usecols*** parameter then it cannot be identified from the data file and it throws an error.

4. Read JSON Data:

```
import pandas as pd

data = pd.read_json('JSON File Path')
print(data.head())
```

5. Read XML Data:

```
import pandas as pd

data = pd.read_xml('XML File Path')
print(data.head())
```

Basic Data Analysis

Learning Topics

- ✓ Display data
- ✓ Understanding data size
- ✓ Handling rows and indexes
- ✓ Handling columns and it's data types
- ✓ Observe null values
- ✓ Duplicate values.
- ✓ Access subset of data
- ✓ Boolean indexing

GitHub link: *Basic_Data_Analysis*

Understand the nature and characteristics of the data is the main step in Data Science or in Machine Learning. Some types of analysis techniques are available in pandas to understand the data. we will learn one by one in this chapter.

First load all the data as a pandas DataFrame to start the analysis. The below code loads all the csv file as pandas DataFrame. Now the analyzing data is in ***df*** (pandas DataFrame) variable.

```
import pandas as pd

df = pd.read_csv('File path')
df.head()
```

1. Display Data

- There are three ways to display the data as follows.

Keyword / Function	***Use***
print(df)	*Display whole data*
df.head(10)	*Display staring rows*
df.tail(10)	*Display last rows*

- By default, head argument ***n = 5***, so by default it displays first 5 rows. If we want to display specific number of rows then pass needful number.
- head() function can take negative values also. Suppose if ***n = -3*** pass to head() then it will remove last 3 rows and display remaining rows.

- Vice versa tail. By default, tail also has n =5, so it displays last 5 rows.

2. Understanding Data Size

- Size is the basic characteristic of the data. There are different types of techniques available to understand the DataFrame size as follwes.

Keyword / Function	***Use***	***Return type***
`df.shape`	*Displays number of rows and columns.*	`<class 'tuple'>`
`df.size`	*Displays total number of labels / values*	`<class 'numpy.int32'>`
`df.ndim`	*Displays number of axes / array dimension.*	`<class 'int'>`
`df.info()`	*Displays total information of the data*	`<class 'NoneType'>`

3. Handling Rows and Indexes

- Indexes of data is one of the important topics in the data handling. Here will learn about how to see the index numbers and how to set a particular column as index.

Keyword / Function	***Use***
`df.index`	*To know dataset index numbers.*
`df.set_index("clm_1")`	*To set a specific column value as index numbers.*

`df.drop(index=2, inplace=True)`
 	
`df.drop(index=[2,3,4], inplace=True)`
	Drop some specific rows based on index numbers.

4. Handling Columns and It's Data Types

- Sometimes we need to change column names and data types of each column. In those cases, will use the below techniques.

Keyword / Function	***Use***
`df.columns`	*Provides all column names as series.*
`df.dtypes / df.info()`	*To know all columns data types*
`df['clm_1'].astype('dtype')`	*To change a particular column data type. (Column type casting)*
`df.astype({'clm_1': 'int', 'clm_2': 'str'})`	*Type caste the more than one column at a time.*

`df.drop(['clm_1', 'clm_2'], axis=1)`
`df.drop(columns=['clm_1', 'clm_2'])`
	To drop the columns

`df.info()`	*Displays all information of all columns.*

5. Observe Null Values

- Understanding of null values is one of the important step in data preprocessing. The below functions will helpful to understand the null values and its nature in data.

Keyword / Function	***Use***
`df.isna() / df.isnull()`	*Return the boolean values of data frame contains true where null values is present.*
`df.notna() / df.notnull()`	*Vise verse of* `isna()` *function*
`df.isna().sum() / df.notna().sum()`	*Return total number of null values in each column.*

6. Duplicate Values

- An important part of data analysis is analyzing *Duplicate Values* and removing them. Pandas *duplicated()* method helps in analyzing duplicate values. It returns a Boolean Series denoting duplicate rows.

Syntax: `DataFrame.duplicated(subset=None, keep='first')`

`subset:` optional
- Only consider certain columns for identifying duplicates, by default use all of the columns.

`Keep:` {'first', 'last', False} : default 'first'
- first : Mark duplicates as True except for the first occurrence.
- last : Mark duplicates as True except for the last occurrence.
- False : Mark all duplicates as True.

7. Access Subset of Data

- Pandas provides 2 interesting functions ***iloc*** and ***loc*** for access subset of data.

Syntax: **`DataFrame.iloc`**

- Iloc[] is purely integer index based selection by position. It takes rose and column index numbers and returns subset of data based on those index numbers
- .iloc[] is primarily integer position based, but may also be used with a Boolean array.
- ***Note*** : That contrary to usual python slices, the start included but the stop is excluded.

Keyword / Function	***Use***
`df.iloc[0]`	*Returns 0th row as pandas series.*
`df.iloc[[0]]`	*Returns 0th row as pandas data frame.*
`df.iloc[[rows_indexes], [clms_indexs]]`	*Return data frame with mentioned row and column indexes,*

Ex: `df.iloc[[1,2], [0,5]]`	
`df.iloc[1:3, 0:3]`	*1st row to 2nd row and 0th column to 2nd column*
`df.iloc[ :3 , 0 : 3]`	*Starting row to 2nd row and 0th column to 2nd column.*
`df.iloc[ 10 :, 0 : 3]`	*10th row to last row and 0th column to 2rd column*
`df.iloc[ :, 0 : 3]`	*First row to last row and 0th column to 2nd column.*

Syntax: **`DataFrame.loc`**

- .loc[] is purely integer index based selection by position. It takes rose and column index numbers and returns subset of data based on those index numbers
- .loc[] is primarily label based, but may also be used with a Boolean array.
- .loc[] is very helpful when data set has string type of indexes.
- ***Note*** : that contrary to usual python slices, **both** the start and the stop are included.

Keyword / Function	***Use***
`df.loc[['str_1', 'str_2']]`	*Returns data frame with selected string indexed rows only*
`df.loc['str_1':'str_5', 'clm_2':'cls_5']`	*Returns data frame with selected string indexed rows and columns only*

8. Boolean Indexing:

- Boolean indexing is the process of selecting the subset of data with boolean values based on some conditions.
- Comparison operators (*<, >, <=, >=, ==, ===, !=, ~*) are using to creating the boolean indexing.

  Ex: `df[df['column_name'] > value]`

- *`isin()`* function Is one of the way to make *boolean Indexing*. This function is used to check multiple value in a single column.

  Ex: `df[df['column_name'].isin([val_1, val_2, _ _ _, val_n])]`

- Applying boolean indexing on multiple columns is one of the scenarios for this requirement we need to separate each condition with parentheses "()".

  Ex: `df[(df['column_name'] > val ) & () | () & () ……]`

Working with Missing Data

Learning Topics

- ✓ Checking for missing values using isnull() and notnull()
- ✓ Checking for missing values in graphical format
- ✓ Filling null values

Code link: *Working_with_Missing_Data*

Missing Data can occur when no information is provided for one or more items or for a whole unit. It is a very big problem in real life scenario. Missing Data can also refer as NA (Not Available) values in pandas. Sometimes many datasets simply arrive with missing data, either because it exists and was not collected or it never existed. For Example, suppose different user being surveyed may choose not to share their income, some user may choose not to share the address in this way many datasets went missing.

In Pandas missing data is represented by two values:

- ***None***: None is a python single object that is often used for represents the missing data in python code.
- ***NaN***: NaN (Not a Number) is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation

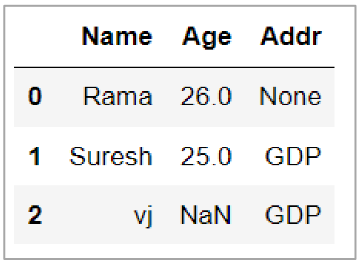
Pandas treat None and *NaN* as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful functions for detecting, removing, and replacing null values in Pandas DataFrame:

- `isnull()`
- `notnull()`

- `dropna()`
- `fillna()`
- `replace()`
- `interpolate()`

1. Checking for Missing Values Using isnull() and notnull()

The *isnull()* or *notnull()* function are helpful to check the missing values in Pandas DataFrame. Both function help in checking whether a value is *null* or not. These functions can also be used in Pandas Series in order to find null values in a series. The *isnull()* function returns a DataFrame with Boolean values which are True for *NaN* values.

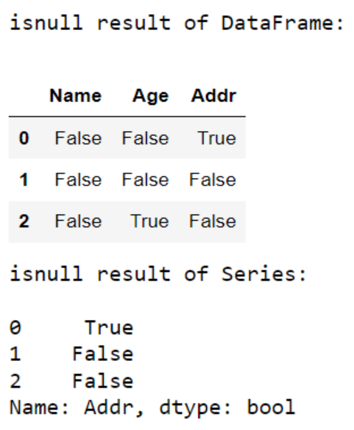


	Name	Age	Addr
0	Rama	26.0	None
1	Suresh	25.0	GDP
2	vj	NaN	GDP

1.1. Code:

```python
import pandas as pd

# Checking whether the total data set has null values or not
print("isnull result of DataFrame:")
display(data.isna())

# Checking whether a single column has null values or not
print(f"isnull result of Series: \n\n{data['Addr'].isnull()}")

###############SECOND WAY################

# Checking whether the total data set has null values or not
print("notnull result of DataFrame:")
display(data.notna())

# Checking whether a single column has null values or not
print(f"notnull result of Series: \n\n{data['Addr'].notnull()}")
```

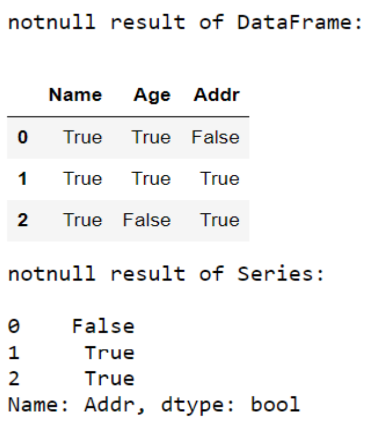
Output:



```
isnull result of DataFrame:
```

	Name	Age	Addr
0	False	False	True
1	False	False	False
2	False	True	False

```
isnull result of Series:

0     True
1    False
2    False
Name: Addr, dtype: bool
```



```
notnull result of DataFrame:
```

	Name	Age	Addr
0	True	True	False
1	True	True	True
2	True	False	True

```
notnull result of Series:

0    False
1     True
2     True
Name: Addr, dtype: bool
```

In above program, the second way output is reversing of first way output.

2. Checking for Missing Values in Graphical Format

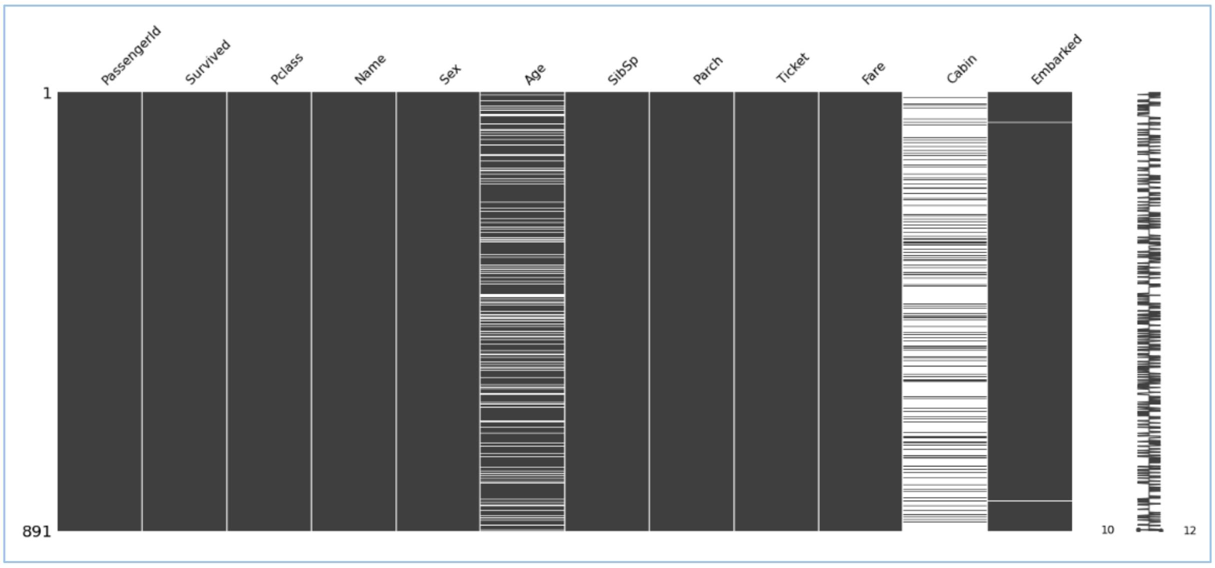
The `missingno` library helps to Identify the nature of null values in the data by using different types of graphs like matrix, bar graph, heatmap:

2.1. Code:

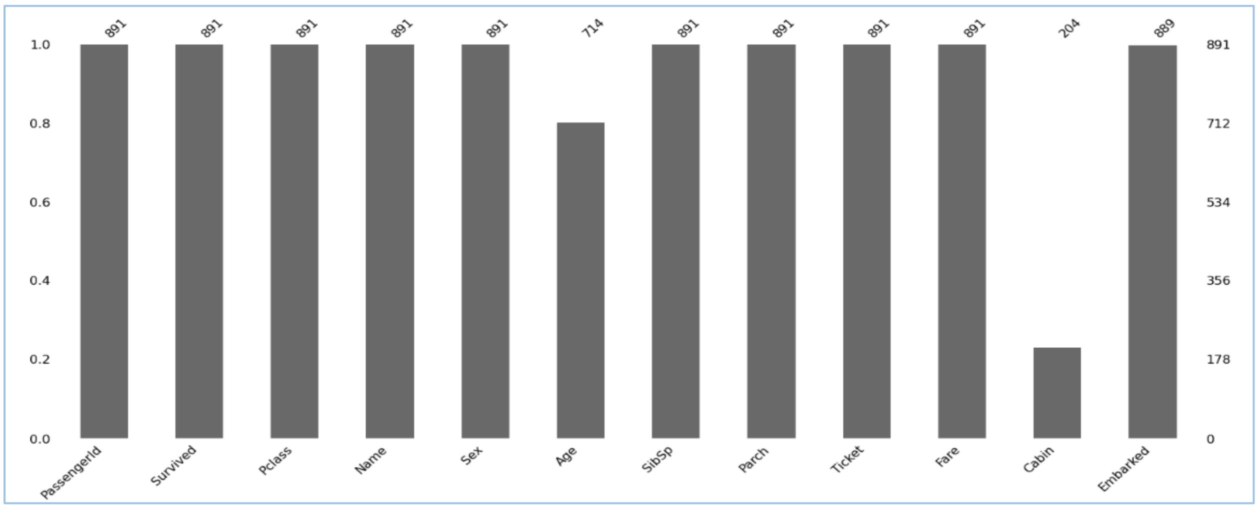
```
# Installing missingno package
pip install missingno

# Importing the package
import missingno as msno

# printing matrix graph.
msno.matrix(data)
# printing bar graph.
msno.bar(data)
# printing heatmap graph.
msno.heatmap(data)
```

Output:

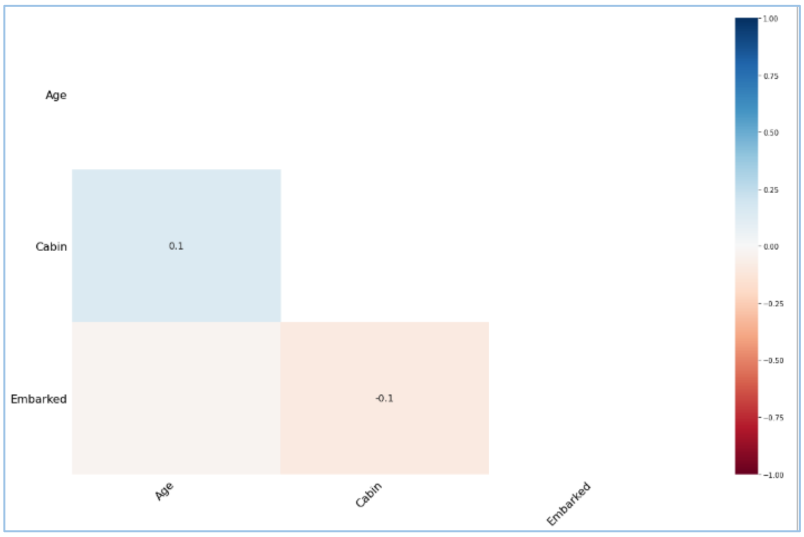
`Printing matrix graph:`



`Printing bar graph:`



Printing heatmap graph:



3. Filling Null Values

In order to fill null values we will use fillna*()*, *replace()* and *interpolate()* function. These functions replace *NaN* values with some other value of their own. The *interpolate()* function is basically uses in various interpolation technique to fill the missing values rather than hard-coding the value.

Keyword / Function	***Use***
`df.fillna(0)`	*Fill null values with a specific value (0)*
`df.fillna('str_val')`	*Filling null values in string type of data.*
`df.fillna(method='pad')`	*Fill null values with previous value.* *Note: It can't fill null value if first value is null in a column. Because first values don't have previous value.*
`df.fillna(method='bfill')`	*Fill null values with next value.* *Note: It can't fill null value if last value is null in a column. Because last values don't have next value.*
`df.dropna()`	*Dropping rows if any row has latest one null value.*
`df.dropna(axis = 1)`	*Dropping columns if any column has latest one null value.*
`df.dropna(how = 'all')`	*Drop rows, if all values are null in that row.*
`df.dropna( subset = ['clm_1', 'clm_2'])`	*Dropping rows if all subset column values are null.*

3.1. Filling Null Values with Mean:

- ***Continuous Numerical Data:*** Suitable for continuous numerical features where the distribution is approximately **Normal or Gaussian**
- ***Sample Size:*** If null values count is small, filling them with the mean might be reasonable.
- ***Outliers:*** Filling null values with the mean might not be the best choice when data contains outliers.
- ***Sequential data:*** If data is a time-series or sequential data, then filling with mean may not accurately capture the temporal dynamics. Techniques such as **forward filling, backward filling, or interpolation** might be more appropriate in such cases.

- ***Impact on Distribution:*** Filling null values with the mean can alter the distribution of your data, particularly if there are a large number of missing values. This can impact the performance of some machine learning models, especially those sensitive to distributional changes.

Calculating the mean without excluding NaN values can lead to inaccurate results.
Use the *skipna* parameter in the *mean()* method to exclude NaN values.

3.1.1. Code:

```
# Calculating mean value by exclude the existed nulls.
mean = df['clm_1'].mean(skipna=True)

# Filling null values with mean.
data['clm_1'].fillna(mean, inplace=True)
```

3.2. Filling Null Values with Median:

- ***Presence of Outliers:*** When your data contains outliers, using the median is more robust than the mean. Outliers can heavily influence the mean, making it less representative of the central tendency of the data. The median, being resistant to outliers, provides a better estimate in such cases.
- ***Skewed Distributions:*** If your data is heavily skewed or not normally distributed, the median can be a better measure of central tendency compared to the mean. In skewed distributions, the mean might be pulled towards the tail of the distribution, making it less representative of the central value.
- ***Ordinal or Categorical Data:*** When dealing with ordinal or categorical data, where the concept of average doesn't apply, using the median can be more meaningful than the mean. It provides a value that lies in the middle of the ordered data, which can be more interpretable.
- ***Limited Impact on Distribution:*** Filling null values with the median has a minimal impact on the distribution of your data compared to the mean. This makes it a suitable choice when you want to preserve the shape of the original distribution as much as possible.
- ***Preservation of Relationships:*** Using the median can help preserve the *overall distribution*, and *relationships between variables* within the data better than other methods.

3.2.1. Code:

```
# Calculating median value by exclude the existed nulls.
median_value = data['clm_1'].median(skipna=True)

# Filling null values with median.
data['clm_1'].fillna(median_value, inplace=True)
```

3.3. Filling Null Values with Mode:

- ***Categorical Data:*** Filling null values with the mode is perfect way when dataset contains categorical variables (e.g., colors, categories, labels). This helps maintain the integrity of the categorical distribution.
- ***Nominal Data:*** For nominal variables where there's no inherent order, using the mode is often the most suitable approach. It ensures that the imputed values align with the most common category in the dataset.
- ***Sparse Data:*** In datasets where certain categories are significantly more prevalent than others, filling null values with the mode can be advantageous. It helps maintain the relative frequencies of different categories and prevents imbalances in the data.

3.3.1. Code:

```python
# Calculating median value by exclude the existed nulls.
median_value = data['clm_1']. mode()[0]

# Filling null values with median.
data['clm_1'].fillna(median_value, inplace=True)
```

Encoding Techniques

Learning Topics

- ✓ What is encoding
- ✓ One-hot / dummy encoding
- ✓ Ordinal encoding
- ✓ Nominal and ordinal data
- ✓ Types of encoding technique
- ✓ Duplicate values
- ✓ Access subset of data
- ✓ Boolean indexing

GitHub link: *Encoding Techniques*

Getting started in applied machine learning can be difficult, especially when working with real-world data. Often, machine learning tutorials will recommend or require that you prepare your data in specific preprocessing ways before fitting a machine learning model. In this chapter will learn one of the preprocessing step called *Label Encoding*.

1. Encoding:

- Encoding is the process of converting string / labeled data (categorical data) into numerical data.

1.1. What is Categorical Data:

- Categorical data are variables that contain label / string values rather than numeric values.
- *Some examples include:*
  - A “*pet*” variable with the values: “*dog*” and “*cat*“.
  - A “*color*” variable with the values: “*red*“, “*green*” and “*blue*“.
  - A “*place*” variable with the values: *“first”,* “*second*” and “*third*“.
- Here each value represents a different category.

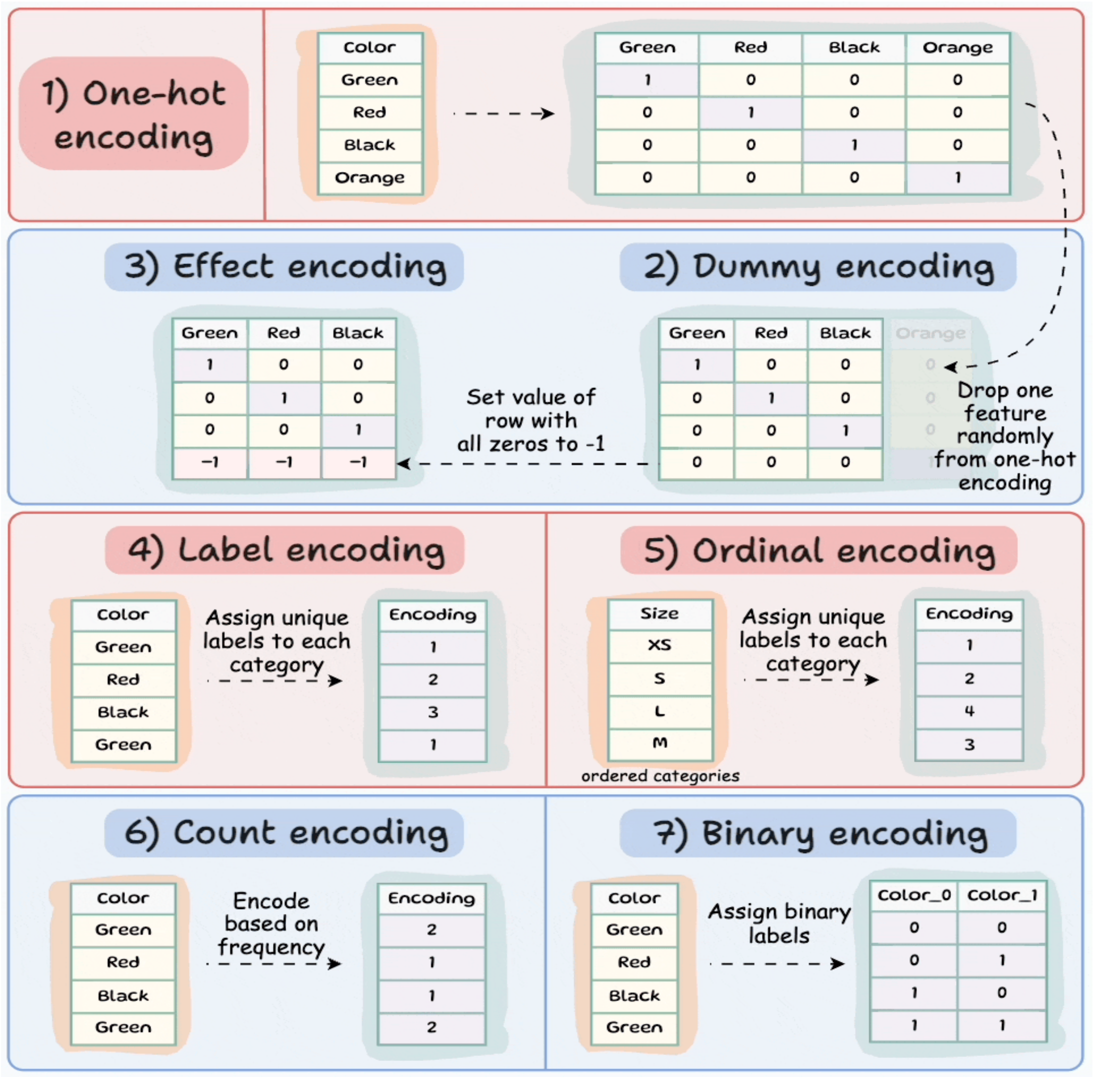
1.2. What is the Problem with Categorical Data:

- Some algorithms can work with categorical data directly. For example, a decision tree can be learned directly from categorical data, no data transform required.
- Many machine learning algorithms cannot operate on label data directly. They require all input variables and output variables to be numeric.
- This means that categorical data must be converted to a numerical form.

1.3. Types of Encoding Technique:

There are plenty of methods to encode categorical variables into numeric and each method comes with its own advantages and disadvantages. To discover them, we will see the following ways to encode categorical variables:

1. One-hot/dummy encoding
2. Binary encoding
3. Label encoding
4. Ordinal encoding
5. Frequency / count encoding
6. Target encoding / Mean Encoding
7. Feature Hashing
8. Weight of evidence encoding


1) One-hot encoding
3) Effect encoding
2) Dummy encoding
Set value of row with all zeros to -1
Drop one feature randomly from one-hot encoding
4) Label encoding
Assign unique labels to each category
5) Ordinal encoding
Assign unique labels to each category
ordered categories
6) Count encoding
Encode based on frequency
7) Binary encoding
Assign binary labels

Color
Green
Red
Black
Orange

Green	Red	Black	Orange
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Green	Red	Black
1	0	0
0	1	0
0	0	1
-1	-1	-1

Green	Red	Black	Orange
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	

Color
Green
Red
Black
Green

Encoding
1
2
3
1

Size
XS
S
L
M

Encoding
1
2
4
3

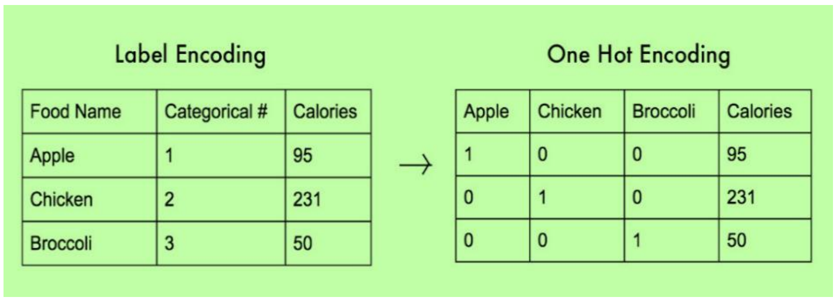
Color
Green
Red
Black
Green

Encoding
2
1
1
2

Color
Green
Red
Black
Green

Color_0	Color_1
0	0
0	1
1	0
1	1

2. One-Hot / Dummy Encoding:



Label Encoding

Food Name	Categorical #	Calories
Apple	1	95
Chicken	2	231
Broccoli	3	50

$\rightarrow$

One Hot Encoding

Apple	Chicken	Broccoli	Calories
1	0	0	95
0	1	0	231
0	0	1	50

In this technique, the categorical parameters will prepare as a separate column for each label and assign a value 1 or 0 based on its presence in that particular row.

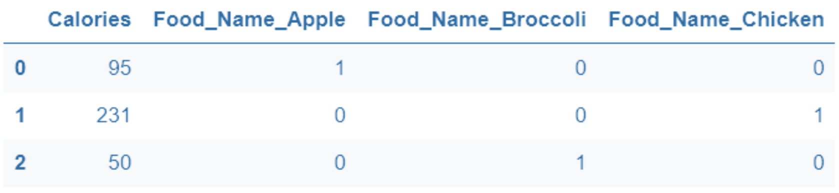
Let’s understand it with an example, consider the data where foods (apple, chicken, broccoli) and their corresponding calories are given. These food names need to convert into numerical by using one hot encoding.

- First it will take all unique labels from “*Food Name*” column and creates each column for each label.
- Now three labels Apple, chicken, and broccoli are three columns in the new data set and each column assigned with a value 1or 0 as shown in above image.

2.1. Code:

```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import OneHotEncoder

# One hot encoding
data_1 = pd.get_dummies(data, columns=['Food_Name'])
display(data_1)

######################## WAY -2 ##############################
## Code with sklearn package by using OneHotEncoder()
#Create an instance of One-hot-encoder
enc=OneHotEncoder()

'''NOTE: we have converted the enc.fit_transform() method to array because
the fit_transform method of OneHotEncoder returns SpiPy sparse matrix this
enables us to save space when we have huge  number of categorical
variables'''

columns = sorted(data['Food_Name'].unique())
print(f'Food_Name labels: {columns}')

enc_arr = enc.fit_transform(data[['Food_Name']]).toarray()
enc_data = pd.DataFrame(enc_arr, columns=columns).astype(int)
New_df=data.join(enc_data).drop(['Food_Name'], axis=1)

print(New_df)
```

Output:



```
   Calories  Food_Name_Apple  Food_Name_Broccoli  Food_Name_Chicken
0        95                1                   0                  0
1       231                0                   0                  1
2        50                0                   1                  0
```

2.2. Advantages:

- ***Simplicity:*** This method is easier to understand and use.
- ***Compatibility:*** One-hot encoding is compatible with most machine learning algorithms.
- ***Avoiding bias:*** One-hot encoding helps to avoid the bias that may be introduced by encoding categorical values as ordinal values.

- ***Better performance***: One-hot encoding often results in improved performance of machine learning models, particularly with decision trees and Random Forest algorithms.
- ***Efficient storage:*** One-hot encoding is space-efficient, as the encoded data can be stored as a sparse matrix, which uses less memory than dense matrices.

2.3. Disadvantages:

- It can lead to increased dimensionality, as a separate column for each label. This can make the model more complex and slower to train.
- It can lead to sparse data, as most observations will have a value of 0 in most of the one-hot encoded columns. These 0 values create negligible / infinity values in mathematical calculations in some models (ANN, Gradient Decent, etc...)
- It can lead to overfitting, especially if there are many categories in the variable and the sample size is relatively small.
- It can lead the multi-correlation problem due to high dependency between one-hot encoding columns.

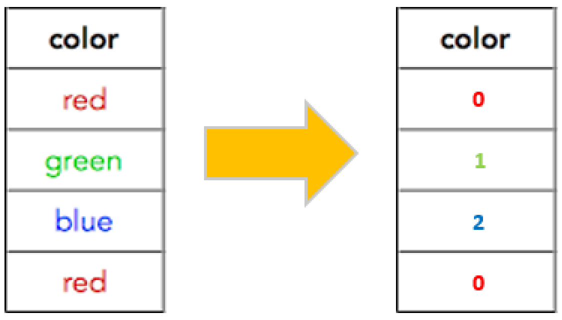
3. Label encoding:

Label encoding is a process of converting categorical data into numerical data by assigning a unique integer label to each category. but it does not add any additional information to the data.

The steps involved in label encoding are as follows:

1. Identify the categorical variable that needs to be encoded.
2. Assign a unique numerical label to each category of the variable, starting from 0 or 1.
3. Replace the categorical values in the dataset with their corresponding numerical labels.
4. Store the mapping of the original categorical values to their numerical labels, as it may be needed later for decoding the labels back to their original values.

For example, let's say we have a categorical variable "color" with three categories: "red", "green", and "blue". To encode this variable using label encoding, we would assign the labels 0, 1, and 2 to the categories respectively. The resulting encoded data would replace "red" with 0, "green" with 1, and "blue" with 2. We would also store the mapping of the original categories to their numerical labels, as follows:



{"red": 0, "green": 1, "blue": 2}

This mapping may be useful later for decoding the labels back to their original values.

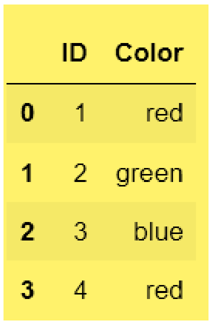
3.1. Code:

```
# Import label encoder
from sklearn import preprocessing

# label_encoder object knows
# how to understand word labels.
label_encoder = preprocessing.LabelEncoder()
```

```python
# Encode labels in column 'species'.
df['color']= label_encoder.fit_transform(df['color'])
print(df)
```

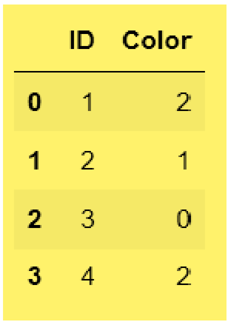
Output:

Before label encoder:



	ID	Color
0	1	red
1	2	green
2	3	blue
3	4	red

After label encoder:



	ID	Color
0	1	2
1	2	1
2	3	0
3	4	2

3.2. Advantages:

- ***Simplicity:*** Label encoding is a simple technique that is easy to implement and understand.
- ***Space efficiency:*** Label encoding typically requires less memory than one-hot encoding, another common technique for encoding categorical data.
- ***Maintains the range of the variable:*** Unlike one-hot encoding, which can expand the feature space substantially, label encoding keeps the range of the variable within a reasonable size.

3.3. Disadvantages:

- ***Arbitrary numerical assignments:*** Label encoding assigns *arbitrary numerical values* to categorical variables, which may not represent the true relationship between the categories. This can result in misleading results and reduced accuracy.
- ***Implies order:*** Label encoding implies an order between categories, even when there is none. For example, encoding "blue" as 2 and "green" as 1 implies that blue is somehow "better" or "higher" than green, which is not true.
- ***May not be suitable for nominal variables:*** Label encoding may not be suitable for nominal variables, which do not have any inherent order or hierarchy between the categories.
- ***Sensitive to initial assignment:*** The assigned numerical values are sensitive to the initial assignment and can lead to different results for the same data set if assigned differently.
- ***May lead to overfitting:*** Label encoding may lead to overfitting if the numerical values assigned to the categories are too similar or too different, as the algorithm may learn the encoding itself rather than the actual data.

4. Ordinal Encoding:

- Ordinal encoding technique using when categorical data is ordinal data.

Label	***Encoding value***
Worst	0
Good	1

Better	2
Best	3

- Whenever your categorical feature has ranks in between then the labels then that feature is called the ordinal data. The encoding values should be assigned based on the priority in this kind of data.
- For example, data have a feature called *performance* and it contains labels as *good, best, better, worst*. These all labels have some priority between them. Observe that priority in below.

4.1. Code:

```python
from sklearn.preprocessing import OrdinalEncoder
import pandas as pd
import numpy as np

# Creating distinct label values for mapping purpose.
encoders = {"best": 3, "better": 2, "good":1, "worst":0}
labels_order = ["worst", "good", "better", "best"]

# Way-1: Label Encoding Using map() Function
data["Performance_encoding_map"] = data['Performance'].map(encoders)

# Way-2: Label Encoding Using replace() Function
data["Performance_encoding_replace"] =
data['Performance'].replace(encoders)

# Way- 3: Label Encoding Using sklearn Librery
oe = OrdinalEncoder(categories=[labels_order], dtype=int)
data["Performance_encoding_algorithm"] =
oe.fit_transform(np.array(data["Performance"]).reshape(-1,1))
print(oe.categories_)
data
```

Output:

Performance	Performance_encoding_map	Performance_encoding_replace	Performance_encoding_algorithm
best	3	3	3
better	2	2	2
good	1	1	1
worst	0	0	0

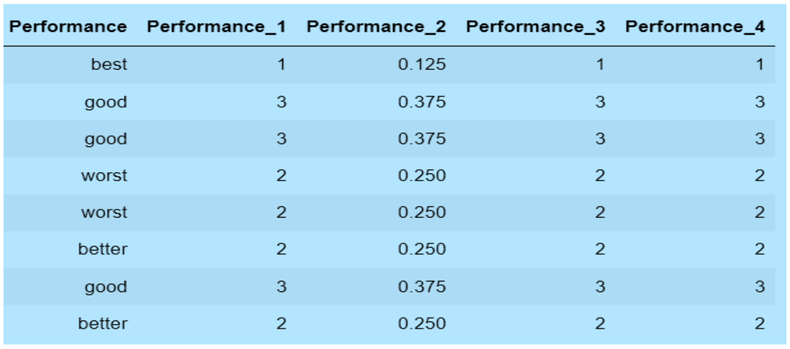
5. Frequency or Count Encoding:

- Frequency encoding Converts label to numerical based on the count of a label in a column.
- For example, if we have a column "*city_name*" and it contains three Labels are *India, America, South Korea*. Let's consider *India* appears 10 times, *America* appears 9 times, and *South Korea* appears 11 times. In this case the *India* will replace with 10 and *America* replaced with 9 and *South Korea* replaced with 11.

5.1. Code:

```python
from feature_engine.encoding import CountFrequencyEncoder
import category_encoders  as ce
import pandas as pd

# Way - 1: Using map()
performance_frq_dict = data['Performance'].value_counts().to_dict()
data['Performance_1'] = data['Performance'].map(performance_frq_dict)
data

# Way - 2 : Using groupby()
performance_frq_dict =
dict(data.groupby('Performance').size()/len(data))
data['Performance_2'] = data['Performance'].map(performance_frq_dict)
data

# Way - 3 : Using CountEncoder
count_encoder = ce.CountEncoder(cols=['Performance'])
data['Performance_3'] = count_encoder.fit_transform(data['Performance'])
data

# Way - 4 : Using CountFrequencyEncoder
count_frequency_encoder = CountFrequencyEncoder()
data['Performance_4'] =
count_frequency_encoder.fit_transform(data[['Performance']])
data
```

Output:



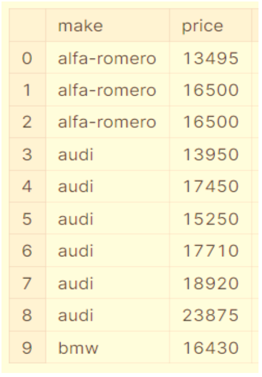
Performance	Performance_1	Performance_2	Performance_3	Performance_4
best	1	0.125	1	1
good	3	0.375	3	3
good	3	0.375	3	3
worst	2	0.250	2	2
worst	2	0.250	2	2
better	2	0.250	2	2
good	3	0.375	3	3
better	2	0.250	2	2

6. Target or Mean Encoding:

Def: A target encoding is a kind of encoding that replaces a feature's categories with some number derived from the target.

- The target encoding required two features. one is encodable feature called input feature, and second one is target feature.
- Here the target feature should affect the input feature. which means input feature should have dependency on target feature.
- Let's take a small data to understand the it better:

  Here *Make* = Input feature

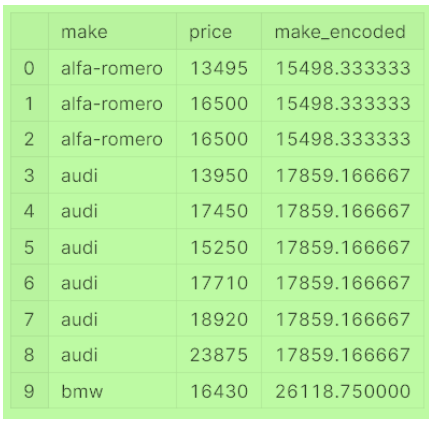


	make	price
0	alfa-romero	13495
1	alfa-romero	16500
2	alfa-romero	16500
3	audi	13950
4	audi	17450
5	audi	15250
6	audi	17710
7	audi	18920
8	audi	23875
9	bmw	16430

Price = Target feature

6.1. Target Encoding Using Mean Technique:

$$in\_feature\_mean = \frac{\sum target\ values\ of\ a\ single\ (ith)\ lable}{Num\ of\ taget\ values\ of\ a\ single\ label}$$



	make	price	make_encoded
0	alfa-romero	13495	15498.333333
1	alfa-romero	16500	15498.333333
2	alfa-romero	16500	15498.333333
3	audi	13950	17859.166667
4	audi	17450	17859.166667
5	audi	15250	17859.166667
6	audi	17710	17859.166667
7	audi	18920	17859.166667
8	audi	23875	17859.166667
9	bmw	16430	26118.750000

In feature mean calculation for *aifa-romero* label:

$$in\_feature\_mean = \frac{13495 + 16500 + 16500}{3}$$
$$= 15498.333$$

We can see all $in\_feature\_mean$ of each label in right table. These mean values are encoded values.

There is a problem involved in above mean technique, those are:

1. If anyone of the label appears very less time in that feature, then that label should have low mean value. But it generates accurate mean that can be high value if we compared with high frequency labels.

A solution to these problems is applying smoothing technique as explained below.

6.2. Target Encoding Using Smoothing Technique:

- The idea of smoothing technique is to blend the $in\_feature\_mean$ with the $overall\_mean$.
- Rare categories get less weight on their category average, and the missing categories just get the overall average.

Formula:

$$encoding = (w\ \times\ in\_feature\_mean)\ +\ \ (1 - w) \times\ overall\_mean$$

$$w = \frac{n}{(n + m)}$$

$overall\_mean$ = averages value of all target values
w = Weight (it is between 0 and 1)
n = Total number of times that category occurs in the data
m = "smoothing factor". Larger values of m put more weight on the overall estimate.

Importance of *m* value:

- The selection of *m* value is very important in above calculation.

- When the distance or difference between (noise of the target data) the target values is more of a particular label. Then, we required mode data to concluded the stable mean. In this condition, m values should be more.
- When the distance or difference (noise of the target data) between the target values is low of a particular label. Then, we concluded the stable mean very easily. In this condition, less m value is sufficient.

6.3. Code:

```python
from sklearn.preprocessing import OrdinalEncoder
import pandas as pd
import numpy as np

# Creating distinct label values for mapping purpose.
encoders = {"best": 3, "better": 2, "good":1, "worst":0}
labels_order = ["worst", "good", "better", "best"]

```

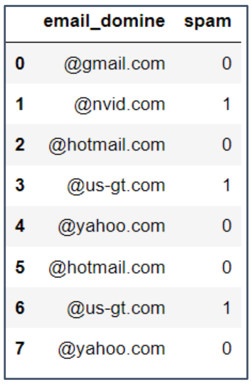
Output:

7. Feature Hashing Encoding:

Def: Feature hashing is a technique used in machine learning to transform categorical data into a numerical format that can be used in models.

7.1. Woking:

- Let's consider we have a dataset of emails as shown in below, and we want to predict the email is spam or not?
- Now we need to convert the "email_domine" column into numerical by using feature hashing encoding technique.
- In feature hashing, first we should select the bins value and based on bins value feature hashing will assign a unique number to each text. The result dataset looks like below.



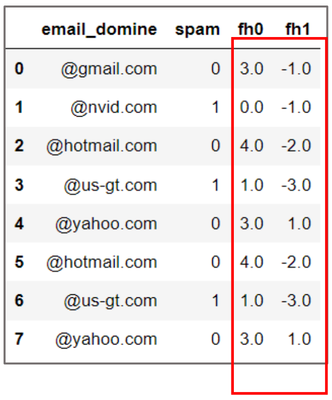
	email_domine	spam
0	@gmail.com	0
1	@nvid.com	1
2	@hotmail.com	0
3	@us-gt.com	1
4	@yahoo.com	0
5	@hotmail.com	0
6	@us-gt.com	1
7	@yahoo.com	0

Note: If the bins value is less then number of unique labels in categorical data leads collisions issue i.e., different categories hashing to the same value.

7.2. Code:

```python
from sklearn.feature_extraction import FeatureHasher

n_features = 2
features = [f"fh{n}" for n in range(0, n_features)]
fh = FeatureHasher(n_features=n_features, input_type='string')
features_array = fh.fit_transform(data['email_domine'])

hasher_df = pd.DataFrame(r.toarray(), columns= features)
data = pd.concat([data, hasher_df], axis=1)
```

Output:



	email_domine	spam	fh0	fh1
0	@gmail.com	0	3.0	-1.0
1	@nvid.com	1	0.0	-1.0
2	@hotmail.com	0	4.0	-2.0
3	@us-gt.com	1	1.0	-3.0
4	@yahoo.com	0	3.0	1.0
5	@hotmail.com	0	4.0	-2.0
6	@us-gt.com	1	1.0	-3.0
7	@yahoo.com	0	3.0	1.0

8. Weight of Evidence (WOE) Encoding:

Def: The WOE value quantifies the relationship between a category and the target variable. It measures how well the category predicts the positive (1) or negative (0) class of the target variable.

In this technique each label will be replaced with WOE value calculated by using below formula:

$$WOE = \log\left(\frac{Distribution\ of\ positive\ events\ in\ the\ category}{Distribution\ of\ negative\ events\ in\ the\ category}\right) \times 100$$

$$WOE = \log\left(\frac{n_{+ve}/N_{+ve}}{n_{-ve}/N_{-ve}}\right)$$

Here n_{+ve} = Number of nth labels belongs to +ve (1) class.
n_{-ve} = Number of nth labels belongs to -ve (0) class.
N_{+ve} = Number of +ve (1) classes in whole dataset.
N_{-ve} = Number of -ve (0) classes in whole dataset.

The WOE value can be positive or negative:

- If WOE>0, it indicates that the category is associated with a higher likelihood of the positive event.
- If WOE<0, it indicates that the category is associated with a higher likelihood of the negative event.
- If WOE=0, it suggests that the category has no discriminatory power between the positive and negative events.

When to Use WOE:

1. Binary Classification Problems: WOE is most commonly used in binary classification problems where you have a binary target variable (0 or 1) and you want to assess the predictive power of categorical independent variables (features) on this binary target.
2. Categorical Variables: WOE is beneficial when dealing with categorical variables with multiple levels or categories. It helps transform these variables into a numeric form that can be directly used in machine learning models like logistic regression.
3. Feature Selection: WOE can be used as a feature engineering technique to select the most informative categories within a categorical variable. This helps reduce dimensionality and improve model performance.

4. Handling Missing Values: WOE can be used to handle missing values within categorical variables. You can create a separate category or bin for missing values and calculate its WOE.
5. Addressing Class Imbalance: When dealing with imbalanced datasets, especially in credit scoring or fraud detection, WOE can help capture the characteristics of the minority class effectively.
6. Collinearity: WOE can be a useful technique to address collinearity issues within categorical variables by grouping similar categories together based on their impact on the target.

Feature Scaling

Learning Topics

- ✓ What is feature scaling
- ✓ Min-Max scaling
- ✓ Z-score normalization
- ✓ log transformation

GitHub link: *Feature_Scaling*

Def: Feature scaling is a technique used in machine learning to standardize (*converting large range values into small range values*) the range of features or variables of a dataset.

- Usually, the scaling range between 0 and 1 by using normal distribution, or a standard distribution with a mean of 0 and standard deviation of 1.
- The primary objective of feature scaling is to normalize the data, to remove any biases that may arise the ranges of the input features.
- Some machine learning algorithms, such as k-nearest neighbours (KNN) and support vector machines (SVM), are sensitive to the scale of input features, and the performance of the algorithm may be improved by scaling the features.
- Feature scaling required especially when the features have different scales or units in the dataset.

The choice of scaling technique depends on the nature of the data and the requirements of the machine learning algorithm being used.

Some of the problems that can be reduced by feature scaling include:

- *Gradient Descent Convergence*: In gradient descent algorithms, feature scaling can help the algorithm converge faster by avoiding oscillation or overshooting during the weight update steps.
- *Distance based algorithms*: Distance-based algorithms such as K-Nearest Neighbours, Support Vector Machines (SVMs), and Principal Component Analysis (PCA) are highly sensitive to the scale of the input features. Feature scaling can help to ensure that these algorithms work correctly by bringing all the features to the same scale.
- *Regularization*: Regularization techniques such as L1 and L2 regularization assume that all features have equal importance. In case of unequal feature scales, regularization techniques may be more biased towards features with larger scales. Feature scaling can help prevent this bias.
- *Neural Networks*: In deep learning, feature scaling can help the network learn faster and more effectively by reducing the chance of vanishing gradients or exploding gradients.

Advantages:

- *Helps with feature selection:* The MinMaxScaler can help with feature selection by reducing the impact of features with large values. This can be important when using techniques such as linear regression or logistic regression, where large values can have a disproportionate impact on the model.

Disadvantages:

- *May not be appropriate for all algorithms:* The *Feature Scaling* may not be appropriate for all machine learning algorithms. For example, some algorithms, such as tree-based models, are not sensitive to the scale of the data and may not benefit from scaling.
- *Can result in loss of information:* The *Feature Scaling* can result in a loss of information, especially when the scaling range is small. This can be important when the original values of the data are important for interpretation or analysis.

1. Min-Max Scaler:

- The *MinMaxScaler* is a data normalization technique, that works by transforming each feature (column) values between 0 and 1, or any other specified range.
- This is achieved by subtracting the minimum value of the feature and dividing by the range (i.e., the difference between the maximum and minimum values). The resulting values will fall between 0 and 1.

The formula for the *MinMaxScaler* is:

$$X_{min-max} = \frac{X_i - X_{min}}{X_{max} - X_{min}}$$

where X_i = original value of the feature,
X_{min} = minimum value of the feature,
X_{max} = maximum value of the feature,
$X_{min-max}$ = scaled value of the feature.

By using *MinMaxScaler*, we can arrange all features are on the same scale, which can be important for many machine learning algorithms. It can also help to avoid the issues that the domination of a feature with a large scale over other features with smaller scales.

1.1. Working:

- ***Identify the range of the data:*** The first step is to determine the minimum and maximum values of scaling feature in the dataset.
- ***Scale the data:*** Once the minimum and maximum values are identified, then scale the data to the desired range by using *MinMaxScaler*.
- The formula calculates the scaled value by subtracting the minimum value from the original value, and then dividing the result by the range (i.e., the difference between the maximum and minimum values).

1.2. Mathematical Intuition Behind MinMaxScaler:

- The mathematical intuition behind MinMaxScaler is based on the concept of linear transformation.
- Linear transformation is a type of transformation that preserves the structure (i.e. relationships between variables) of the data, while scaling the data to a different range.

- In the case of MinMaxScaler, the linear transformation is used to scale the data to a specified range, typically between 0 and 1.

The formula for MinMaxScaler is:

$$X_{min-max} = \frac{X_i - X_{min}}{X_{max} - X_{min}}$$

The formula works by first subtracting the minimum value (X_{min}) from the original value (X_i) to obtain the "distance" of the value from the minimum (X_{min}). This "distance" is then divided by the range, which is the difference between the maximum (X_{max}) and minimum (X_{min}) values of the feature. This gives a scaled value that is proportional to the original value, but scaled to the range between 0 and 1.

If $X_i = X_{min}$ then $X_{min-max} = 0$

If $X_i = X_{max}$ then $X_{min-max} = 1$

1.3. Code:

```
from sklearn.preprocessing import MinMaxScaler
import pandas as pd

# create a sample dataframe
df = pd.DataFrame({'A': [1, 4, 7], 'B': [2, 5, 8], 'C': [3, 6, 9]})
scaling_clms = ['A', 'B']
# create a MinMax scaler object
scaler = MinMaxScaler()

# fit and transform a single column using the scaler
scaled_column = scaler.fit_transform(df[scaling_clms])

clm = [clm for clm in df.columns if clm not in scaling_clms]

# concatenate the scaled column with the remaining columns
scaled_df = pd.concat([pd.DataFrame(scaled_column, columns=scaling_clms), df[clm]], axis=1)

# print the original and scaled dataframes
print("Original dataframe:\n", df)
print("Scaled dataframe:\n", scaled_df)
```

1.4. Advantages:

- *Simple and easy to use:* The MinMaxScaler is a simple and easy-to-use technique for scaling data. It is easy to understand and implement, even for beginners.
- *Preserves the shape of the distribution:* The MinMaxScaler preserves the shape of the distribution of the data. It scales the data to a common range, but does not change the shape of the distribution. This can be important for some applications, such as when the data has a non-normal distribution.
- *Can improve model accuracy:* The *MinMaxScaler* can improve the accuracy of machine learning models, especially when the data has a clear minimum and maximum value. It can help to reduce the impact of outliers and ensure that all features are on a similar scale.

1.5. Disadvantages:

- *Sensitive to outliers:* The MinMaxScaler is sensitive to outliers, which can affect the scaling of the data. Outliers misleading the minimum and maximum values which means if

column has outlier at max value the it will take that outlier as max value, which can reduce the effectiveness of the scaler.
- *May not work well with some distributions:* The MinMaxScaler may not work well with some distributions, such as those with a very narrow range or those with a lot of zero values. In these cases, other scaling techniques may be more appropriate.

2. Standard Scaler

- Standard Scaler is a pre-processing technique used to *standardize* the features of a dataset.
- It is widely used method in data pre-processing and machine learning, particularly for algorithms that assume normal distribution of the input variables.
- Standard Scaler transforms the data with mean of each feature is zero and the standard deviation is 1. This is the main mathematical logic behind the Standard Scaler.

2.1. Working:

The StandardScaler works by subtracting the mean from each feature and then dividing by the standard deviation.

The formula for standardizing a feature is:

$$z = \frac{x_i - \mu}{\sigma}$$

where z = standardized value
x = original value of the feature
μ = mean of the feature
σ = standard deviation of the feature.

- Compute the mean and standard deviation of scaling feature in the dataset.
- Subtract the mean from each value in the column. This centres the data around zero.
- Divide the centre values by the standard deviation. This scales the data so that it has a standard deviation of 1.

The resulting dataset has a mean of zero and a standard deviation of one.

2.2. Mathematical Intuition Behind StandardScaler:

The mathematical intuition behind the StandardScaler is based on the concept of the z-score, which measures the distance of a data point from the mean in units of the standard deviation.

When we standardize a feature using the StandardScaler, we calculate the z-score for each value in the feature by subtracting the mean of the feature from the value and then dividing the result by the standard deviation of the feature. The resulting z-score represents the number of standard deviations that the value is away from the mean.

The mean and standard deviation formulas are:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\sigma = \sqrt{\frac{\sum(X - \mu)^2}{N}}$$

The formula for standardizing a feature is:

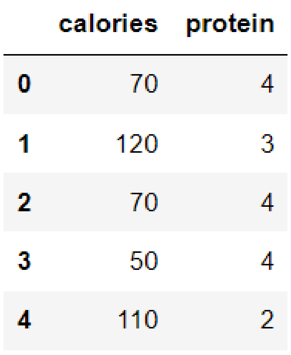
$$z = \frac{x_i - \mu}{\sigma}$$

2.3. Code:

```python
from sklearn.preprocessing import StandardScaler
std_sclr = StandardScaler()
data['calories'] = std_sclr.fit_transform(data.loc[:, ['calories']])
display(data)
```

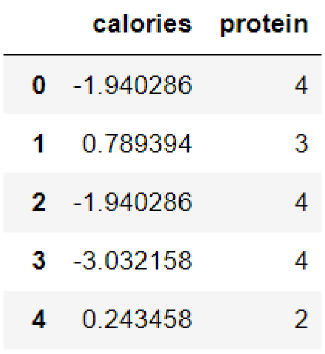
Output:

Before standard scalar:



	calories	protein
0	70	4
1	120	3
2	70	4
3	50	4
4	110	2

After standard encoder:



	calories	protein
0	-1.940286	4
1	0.789394	3
2	-1.940286	4
3	-3.032158	4
4	0.243458	2

2.4. Advantages:

- StandardScaler makes it easier to compare features that are measured in different units or have different scales.
- StandardScaler helps machine learning algorithms to converge more quickly, particularly gradient descent-based algorithms, by making the cost surface smoother.
- StandardScaler can help to reduce the impact of outliers, by scaling the values to be within a more reasonable range.
- StandardScaler is a simple and fast technique that can be easily applied to any dataset.

2.5. Disadvantages:

- StandardScaler assumes that the data is normally distributed, which may not be true in all cases. In such cases, other scaling techniques may be more appropriate.
- StandardScaler can be sensitive to outliers if the number of outliers is very large or if the outliers are very extreme. In such cases, robust scaling techniques may be more appropriate.
- StandardScaler can increase the risk of overfitting in some cases, particularly if the number of features is very large. In such cases, feature selection or regularization may be more appropriate.
- StandardScaler can lead to loss of interpretability, as the transformed features no longer have the same units as the original features.

3. Max Absolute Scaler:

- It rescales the feature values to the range [-1, 1] by dividing through the largest maximum absolute value in each feature.
- It is similar to MinMaxScaler, but instead of scaling the data to a fixed range [0 -1], MaxAbsScaler scales based on the absolute maximum value of the feature.
- This scaler is useful when the data contains large outliers or when the data is not normally distributed.
- MaxAbsScaler can be applied to both dense and sparse input data. It is commonly used in feature scaling for linear models, clustering algorithms, and neural networks.
- MaxAbsScaler transforms the data by making feature's absolute maximum value is 1.
- It does not shift/center the data and thus does not destroy any sparsity.

3.1. Working of MaxAbsScaler

$$X_{maxabsscal} = \frac{x_i}{\max(|X|)}$$

1. First identify the maximum absolute value of the feature.
2. Divide each value of the feature with maximum absolute value. The resultant is the scaled value.

3.2. Mathematical Intuition

This scaling is important because many machine learning algorithms are sensitive to the scale of the input features, and can perform poorly if the input features are not scaled correctly.

It is a simple linear transformation that divides each value in a feature by its absolute maximum value.

$$X_{maxabsscal} = \frac{x_i}{\max(|X|)}$$

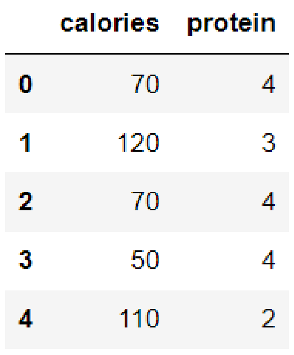
The largest absolute value in the feature becomes 1 when $x_i = \max(|X|)$. In this case both numerator and denominator are same in above formula.

3.3. Code:

```python
from sklearn.preprocessing import MaxAbsScaler

# Create a MaxAbsScaler object
scaler = MaxAbsScaler()

# Scale the data
data['calories'] = scaler.fit_transform(data.loc[:,['calories']])

# Print the scaled data
print("Scaled data:\n\n", data.head())
```

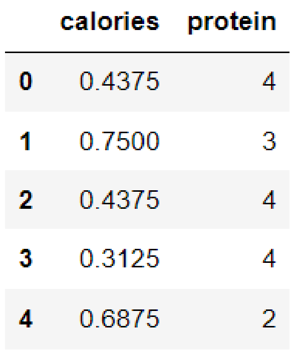
Output:

Before MaxAbsScaler:



	calories	protein
0	70	4
1	120	3
2	70	4
3	50	4
4	110	2

After MaxAbsScaler:



	calories	protein
0	0.4375	4
1	0.7500	3
2	0.4375	4
3	0.3125	4
4	0.6875	2

3.4. Advantages:

- *Maintains the sign of the data:* Unlike MinMaxScaler, which shifts all the data to a positive range, MaxAbsScaler preserves the sign of the data. This is useful when the data contains negative values, as it maintains the relative distances between different values within a feature.
- *Suitable for sparse data:* MaxAbsScaler can be applied to sparse data, which is useful when working with large datasets. It preserves the sparsity of the data, and ensures that the scaling is applied only to the non-zero elements of the data.
- *Easy to interpret:* MaxAbsScaler is a simple linear transformation that is easy to understand and interpret. It is based on the absolute maximum value of each feature, which is an intuitive scaling parameter.

3.5. Disadvantages:

- *Sensitive to outliers:* Like all scaling methods, MaxAbsScaler is sensitive to outliers. Large outliers in the data can affect the scaling and distort the relative distances between the other values.
- *Limited scaling range:* MaxAbsScaler scales the data to a range of -1 to 1, which may not be suitable for all datasets. If the data has a wider range of values, the scaling may not be sufficient to bring all the features to the same scale.
- *Not suitable for non-linear data:* MaxAbsScaler is a linear transformation, and may not be suitable for datasets with non-linear relationships between the input features and the target variable. In such cases, non-linear scaling methods such as logarithmic or exponential scaling may be more appropriate.

4. Robust Scaler

- The Robust scaler is similar to Standard scaler, but Standard scaler uses ***mean and standard deviation***. Robust scaler uses ***median and interquartile range (IQR)***.
- The median is a measure of central tendency that is less sensitive to outliers than the mean, while the IQR is a measure of the spread of the data that is also less sensitive to outliers than the standard deviation.
- This formula centers the data around the median and scales it by the IQR. The resulting dataset has a similar scale to the standardization (mean = 0, standard deviation = 1) but is more robust to outliers.
- The meaning of Robust scaling is "***distance of a data point from the median in units of the IQR***". Which means ***How many IQRs required to get the distance from a data point to median***?.
- This scaler working internally by setting the median as 0 and IQR as 1.

- The Robust Scaler doesn't explicitly set predefined minimum and maximum values like some other scaling methods such as Min-Max Scaling. Instead, it focuses on centering the data by subtracting the median and scaling it by the interquartile range (IQR).

4.1. Working:

$$robust\ scale\ value = \frac{x_i - median}{IQR}$$

1. First calculate Median of scaling feature.
2. Calculate the interquartile range of the feature.
3. Substitute median and IQR in above formula for each and every value of the feature then the resultant is the robust scaled value.

4.2. Mathematical Intuition:

Internally, the Robust Scaler achieves robustness by centering the data based on the median and scaling it based on the interquartile range (IQR). Here's how it works:

1. Centering using Median (location shift):

- The median is a measure of central tendency that represents the middle value of a dataset when it is sorted in ascending order.
- The Robust Scaler subtracts the median of the feature from each data point in that feature.
- This operation effectively centres the data, making the median of the feature become 0 after scaling.

2. Scaling using IQR (scale shift):

- The interquartile range (IQR) is a measure of statistical dispersion, representing the range between the first quartile (25th percentile) and the third quartile (75th percentile) of the data.
- The Robust Scaler then divides each data point in the feature by the IQR.
- This operation scales the data by the spread of the middle 50% of the data, making the IQR of the feature equal to 1 after scaling.

The formula for the Robust Scaler transformation for a given feature X is:

$$robust\ scale\ value = \frac{x_i - median}{IQR}$$

3. Outlier Effect:

The robust scalar is not sensitive to outliers but standard scalar is more sensitive to outliers. To understand this better by understand the mean and median calculations as follow:

Consider the list of input numbers below.

X = {1,2,3,4, 5, 500} Here 500 is the outlier.

Mean:

The mean of X is:

$$\mu_x = \frac{1+2+3+4+5+500}{6}$$

$$= 85.83$$

Now the X mean is 85.83, this mean very far from first 5 values and it is affected by outlier 500.

Median:

- To find the median, we first sort the list. Median is the middle value that splits the list in half.
- The list above is already sorted. Thus its median is 3.5

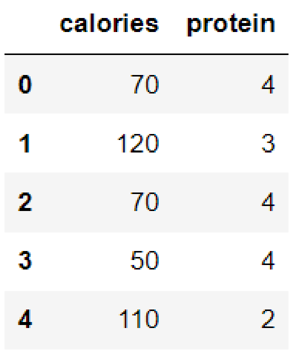
The outlier doesn't affect the median. That's because the median doesn't depend on every value in the list. The last value could have been 500 or even 10000. And it wouldn't change the median at all.

4.3. Code:

```python
from sklearn.preprocessing import RobustScaler

# create a RobustScaler object
scaler = RobustScaler()

# fit the scaler to the data and transform it
data['calories'] = scaler.fit_transform(data.loc[:,['calories']])

# print the original and scaled data
print("Scaled data:\n\n", data.head())
```

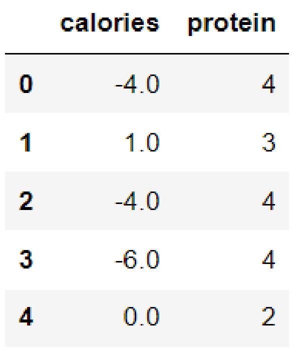
Output:

Before RobustScaler:



	calories	protein
0	70	4
1	120	3
2	70	4
3	50	4
4	110	2

After RobustScaler:



	calories	protein
0	-4.0	4
1	1.0	3
2	-4.0	4
3	-6.0	4
4	0.0	2

4.3. Advantages:

- *Robustness to outliers:* The Robust Scaler is less sensitive to outliers than other scaling methods, such as standardization, which makes it useful in cases where there are extreme values in the data.
- *Preserves the distribution shape:* The Robust Scaler preserves the distribution shape of the original data, unlike methods like standardization that can change the distribution shape.
- *Not affected by the range of values:* The Robust Scaler is not affected by the range of values in the data, making it suitable for use with datasets that have a wide range of values.

4.4. Disadvantages:

- *Data transformation:* The Robust Scaler transforms the data, which can make it difficult to interpret the original values after scaling. This can be a disadvantage in some applications where the original values need to be preserved.
- *May not work well with small datasets:* The Robust Scaler uses the median and interquartile range, which may not be as reliable with small datasets. In such cases, the standardization method may be more suitable.
- *Sensitivity to the shape of the distribution:* The Robust Scaler is sensitive to the shape of the distribution of the data, and may not work as well if the distribution is not symmetric. In such cases, other scaling methods such as normalization may be more suitable.

5. Log transformation scale

- Log transformation is a mathematical transformation that is used to reduce the magnitude of values and to make highly skewed data more suitable for analysis or modelling.
- The log transformation involves taking the logarithm of each value in a dataset, which compresses the range of the data and can make patterns more visible.
- There are two common types of log transformation: natural logarithm (log base e) and base-10 logarithm. The natural logarithm is commonly used because it has certain mathematical properties that make it convenient for analysis.

5.1. Working

The log transformation is a mathematical transformation that involves taking the logarithm of each value in a dataset. In the case of the natural logarithm (log base e), the formula for the transformation is:

$$y = \log(x) \qquad \text{or} \qquad y = \ln(x)$$

1. Substitute each value of a feature into above formula and get the scaled value "y".

The log transformation is useful in situations where the data is highly skewed, such as in financial or economic data, where the range of values can vary widely and outliers can skew the data. By taking the logarithm of each value, the range of values is compressed, and the distribution of the data can be brought closer to a normal distribution.

5.2. Mathematical Intuition

The mathematical intuition behind the log transformation is that it compresses the range of values in a dataset by applying a logarithmic function to each value.

The natural logarithm (log base e) is commonly used in the log transformation because it has certain mathematical properties that make it useful for analysis. For example, the natural logarithm is a continuous, increasing function that maps positive values to negative values, and it is able to handle both large and small values.

One important property of the log transformation is that it has the effect of reducing the impact of extreme values, such as outliers.

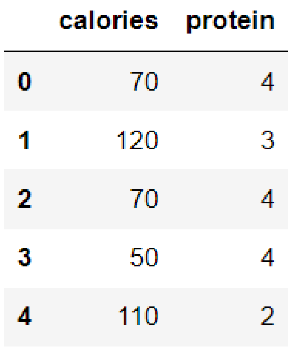
Another important property of the log transformation is that it can be used to linearize relationships between variables. For example, if there is a non-linear relationship between two variables in a regression model, applying a log transformation to one or both of the variables can help to linearize the relationship and make it more suitable for analysis.

5.3. Code:

```python
import numpy as np
data['calories'] = np.log(data.loc[:,['calories']])
data.head()
```

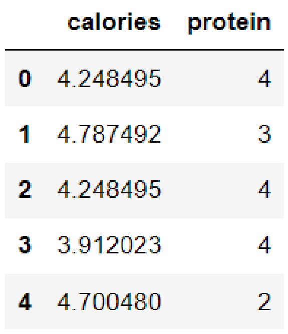
Output:

Before RobustScaler:



```
   calories  protein
0        70        4
1       120        3
2        70        4
3        50        4
4       110        2
```

After RobustScaler:



```
   calories  protein
0  4.248495        4
1  4.787492        3
2  4.248495        4
3  3.912023        4
4  4.700480        2
```

5.3. Advantages:

- Normalization: Log transformation can help to normalize the data by reducing the skewness in the data distribution, which is especially useful when dealing with data with long tails and extreme values.
- Data Compression: Log transformation can compress the data, making it easier to interpret and visualize trends and patterns.
- Multiplicative Effects: Log transformation can help to handle multiplicative effects in data. For example, in finance, a log transformation can be used to convert percentage changes to additive changes.
- Homogeneity of Variance: Log transformation can help to stabilize the variance in the data, making it more homogeneous across different groups.

5.4. Disadvantages:

- Interpretation: It can be challenging to interpret the results of data that has been transformed logarithmically, as the units of measurement are no longer easily interpretable.
- Data Loss: Depending on the nature of the data, some information can be lost during log transformation. For example, negative values cannot be transformed logarithmically.
- Outliers: Log transformation can be highly affected by outliers, which can distort the data and its interpretation.
- Zero Values: Zero values can be problematic when using a logarithmic transformation, as it is impossible to take the logarithm of zero.
- Negative Values: if the data negative values, the log transformation may not be applicable.