Data preprocessing:

What is Data Preprocessing ?

Data preprocessing is a process of preparing the raw data and making it suitable for a machine learning model. It is the first and crucial step while creating a machine learning model.

When creating a machine learning project, it is not always a case that we come across the clean and formatted data. And while doing any operation with data, it is mandatory to clean it and put in a formatted way. So for this, we use data preprocessing task.

Importance of data preprocessing

- The need for data preprocessing is there because good data is more important than good models and for which the quality of the data is of paramount importance.
- Therefore, companies and individuals invest a lot of their time in cleaning and preparing the data for modeling.
- The data present in the real world contains a lot of quality issues, noise, inaccurate, and not complete.
- It may not contain relevant, specific attributes and could have missing values, even incorrect and spurious values.
- To improve the quality of the data preprocessing is essential.
- The preprocessing helps to make the data consistent by eliminating any duplicates, irregularities in the data, normalizing the data to compare, and improving the accuracy of the results.
- The machines understand the language of numbers, primarily binary numbers 1s and 0s.
- Nowadays, most of the generated and available data is unstructured, meaning not in tabular form, nor having any fixed structure to the data.
- The most consumable form of unstructured data is text, which comes in the form of tweets, posts, comments.
- We also get data in the format of images, audio and as we can see, such data is not present in the format that can be readily ingested into a model.
- So, for the parsing, we need to convert or transform the data so that the machine can interpret it.
- Again to reiterate, data preprocessing is a crucial step in the Data Science process.

Data Preprocessing Steps:

1. Data Cleaning
2. Data Integration
3. Data Transformation
4. Data Reduction

5. Data Splitting

1. Data Cleaning

What is Data Cleaning?

Data cleaning is the process of fixing or removing incorrect, corrupted, incorrectly formatted, duplicate, or incomplete data within a dataset.

Assess Data quality

There are various measures of data quality such as accuracy, consistency, completeness, timeliness, and validity. The most important measure of data quality is accuracy because it tells us how close our results are to reality. Common measures during the data quality assessment are:

1- Completeness: This measures how much of the data is present. For example, if you are tracking the number of visitors to a website, completeness would be the percentage of visitors that are correctly recorded.

2- Accuracy: Accuracy is one of the most important measure in data quality assessment as it identifies how close the data is to the true value. For example, if you are recording the temperature outside, accuracy would be how close your reading is to the actual temperature.

3- Timeliness: This measures how up-to-date the data is. For example, if you are tracking the stock price of a company, timeliness would be how close your data is to the current stock price.

4- Consistency: This measures how consistent the data is. For example, if you are tracking the number of employees at a company, consistency would be how often the data changes.

Data anomalies:

Anomaly detection (aka outlier analysis) is a step in data mining that identifies data points, events, and/or observations that deviate from a dataset's normal behavior. An unexpected change within these data patterns, or an event that does not conform to the expected data pattern, is considered an anomaly.

Detect missing values with pandas dataframe functions: .info() and .isna():

- Pandas **df.isna()** function is used to detect missing values.
- It return a boolean same-sized object indicating if the values are NA. NA values, such as None or numpy.NaN, gets mapped to True values.
- Everything else gets mapped to False values.
- Syntax: df.isna()
- Pandas **df.info()** function is used to get a concise summary of the dataframe.
- It comes really handy when doing exploratory analysis of the data.
- To get a quick overview of the dataset we use the df.info() function.

- Syntax: DataFrame.info()

Example:

```
import pandas as pd
df = pd.read_csv('dataset/titanic.csv')
df.info()
df.isnull()
df.isnull().sum()
df.isna()
df.isna().sum()
df.notnull()
df.notnull().sum()
```

Approaches to deal with missing values

1. Keep the missing value as is
2. Remove data objects with missing values
3. Remove the attributes with missing values
4. Estimate and impute missing values

Remove data objects with missing values (Delete Rows with Missing Values):

One way of handling missing values is the deletion of the rows having null values. Rows can be dropped if having one or more row values as null. Before using this method one thing we have to keep in mind is that we should not be losing information. Because if the information we are deleting is contributing to the output value then we should not use this method because this will affect our output.

When to delete the rows in a dataset?

- If a certain row has many missing values then you can choose to drop the entire row.
- When you have a huge dataset. Deleting for e.g. 2-3 rows will not make much difference.
- Output results do not depend on the Deleted data.

Note: No doubt it is one of the quick techniques one can use to deal with missing values. But this approach is not recommended.

Remove the attributes with missing values (Delete columns with Missing Values):

One way of handling missing values is the deletion of the columns having null values. If any columns have more than half of the values as null then you can drop the entire column. Before using this method one thing we have to keep in mind is that we should not be losing information. Because if the information we are deleting is contributing to the output value then we should not use this method because this will affect our output.

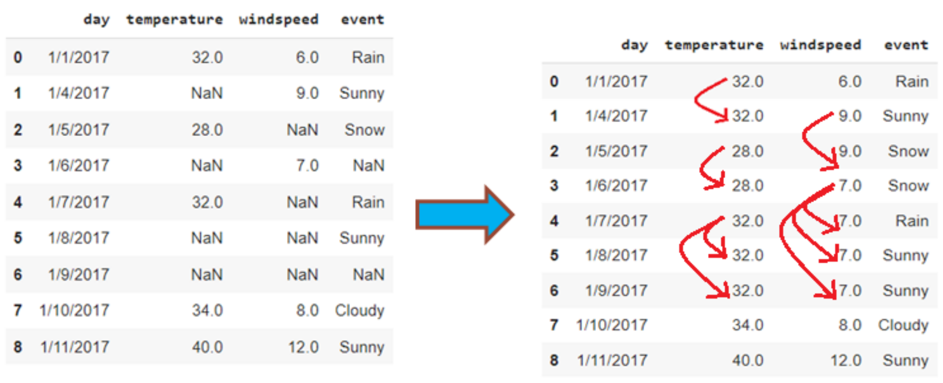
When to delete the column in a dataset?

- If a certain column has many missing values then you can choose to drop the entire column.
- When you have a huge dataset. Deleting for e.g. 2-3 rows/columns will not make much difference.
- Output results do not depend on the Deleted data.

Note: No doubt it is one of the quick techniques one can use to deal with missing values. But this approach is not recommended.

Estimate and impute missing values :
If you can replace the missing value with some arbitrary value using fillna().
Ex. In the below code, we are replacing the missing values with ‘0’.As well you can replace any particular column missing values with some arbitrary value also.

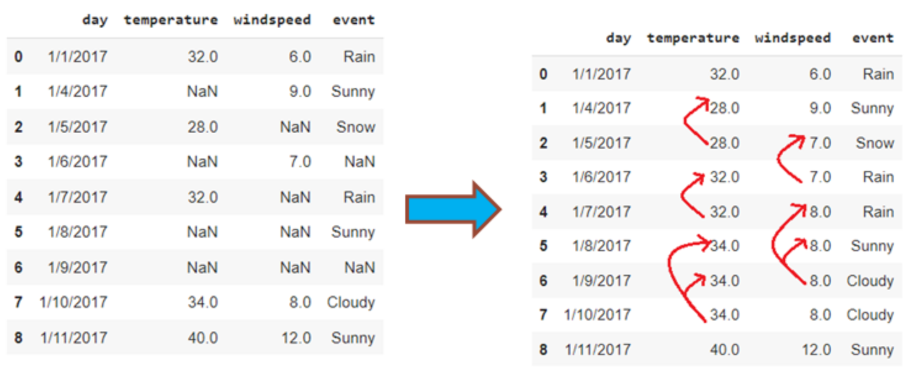
1. Replacing with previous value – Forward fill
We can impute the values with the previous value by using forward fill. It is mostly used in time series data.
Syntax: df.fillna(method=’ffill’)



	day	temperature	windspeed	event
0	1/1/2017	32.0	6.0	Rain
1	1/4/2017	NaN	9.0	Sunny
2	1/5/2017	28.0	NaN	Snow
3	1/6/2017	NaN	7.0	NaN
4	1/7/2017	32.0	NaN	Rain
5	1/8/2017	NaN	NaN	Sunny
6	1/9/2017	NaN	NaN	NaN
7	1/10/2017	34.0	8.0	Cloudy
8	1/11/2017	40.0	12.0	Sunny

	day	temperature	windspeed	event
0	1/1/2017	32.0	6.0	Rain
1	1/4/2017	32.0	9.0	Sunny
2	1/5/2017	28.0	9.0	Snow
3	1/6/2017	28.0	7.0	Snow
4	1/7/2017	32.0	7.0	Rain
5	1/8/2017	32.0	7.0	Sunny
6	1/9/2017	32.0	7.0	Sunny
7	1/10/2017	34.0	8.0	Cloudy
8	1/11/2017	40.0	12.0	Sunny

2. Replacing with next value – Backward fill
In backward fill, the missing value is imputed using the next value. It is mostly used in time series data.



	day	temperature	windspeed	event
0	1/1/2017	32.0	6.0	Rain
1	1/4/2017	NaN	9.0	Sunny
2	1/5/2017	28.0	NaN	Snow
3	1/6/2017	NaN	7.0	NaN
4	1/7/2017	32.0	NaN	Rain
5	1/8/2017	NaN	NaN	Sunny
6	1/9/2017	NaN	NaN	NaN
7	1/10/2017	34.0	8.0	Cloudy
8	1/11/2017	40.0	12.0	Sunny

	day	temperature	windspeed	event
0	1/1/2017	32.0	6.0	Rain
1	1/4/2017	28.0	9.0	Sunny
2	1/5/2017	28.0	7.0	Snow
3	1/6/2017	32.0	7.0	Rain
4	1/7/2017	32.0	8.0	Rain
5	1/8/2017	34.0	8.0	Sunny
6	1/9/2017	34.0	8.0	Cloudy
7	1/10/2017	34.0	8.0	Cloudy
8	1/11/2017	40.0	12.0	Sunny

3. SimpleInputer( ) : Replace missing values using a descriptive statistic (e.g. mean, median, or most frequent) along each column, or using a constant value.

4. Interpolation

Missing values can also be imputed using 'interpolation'. Pandas interpolate method can be used to replace the missing values with different interpolation methods like 'polynomial', 'linear', 'quadratic'. The default method is 'linear'.

Syntax: **df.interpolate(*method='linear'*)**

For the time-series dataset variable, it makes sense to use the interpolation of the variable before and after a timestamp for a missing value. Interpolation in most cases supposed to be the best technique to fill missing values.

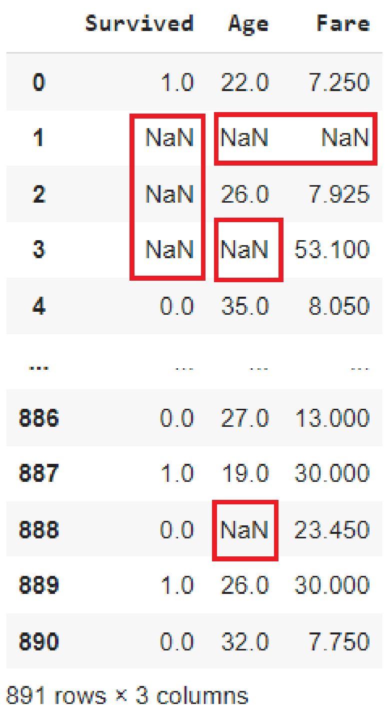
Example: Handling missing values: python code:

We have taken dataset **titanic.csv** which is freely available at kaggle.com.This dataset was taken as it has missing values.

1.**Reading the data**

```
import pandas as pd

df = pd.read_csv("train.csv",
usecols=['Age','Fare','Survived'])

df
```

The dataset is read and used three columns **'Age', 'Fare', 'Survived'.**



	Survived	Age	Fare
0	1.0	22.0	7.250
1	NaN	NaN	NaN
2	NaN	26.0	7.925
3	NaN	NaN	53.100
4	0.0	35.0	8.050
...	...	...	...
886	0.0	27.0	13.000
887	1.0	19.0	30.000
888	0.0	NaN	23.450
889	1.0	26.0	30.000
890	0.0	32.0	7.750

891 rows × 3 columns

2. **Checking if there are missing values**

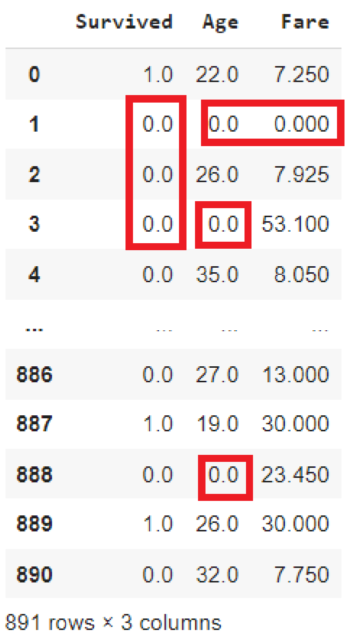
```
df.isnull().sum()
```

Output:

```
Survived      4
Age         179
Fare          2
dtype: int64
```

3.Filling missing values with 0.

```
new_df = df.fillna(0)
new_df
```

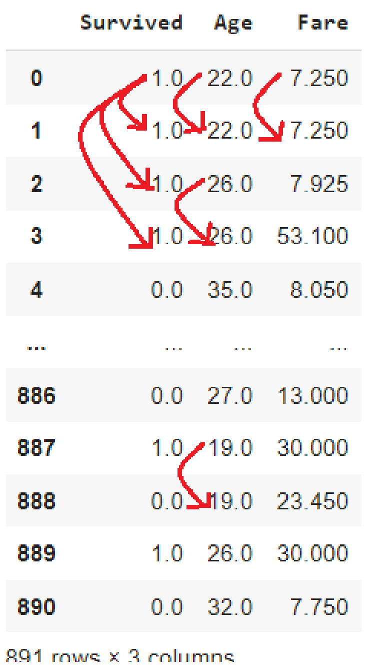


	Survived	Age	Fare
0	1.0	22.0	7.250
1	0.0	0.0	0.000
2	0.0	26.0	7.925
3	0.0	0.0	53.100
4	0.0	35.0	8.050
...	...	...	...
886	0.0	27.0	13.000
887	1.0	19.0	30.000
888	0.0	0.0	23.450
889	1.0	26.0	30.000
890	0.0	32.0	7.750

891 rows × 3 columns

4. Filling NaN values with forward fill value

```
new_df = df.fillna(method="ffill")
new_df
```



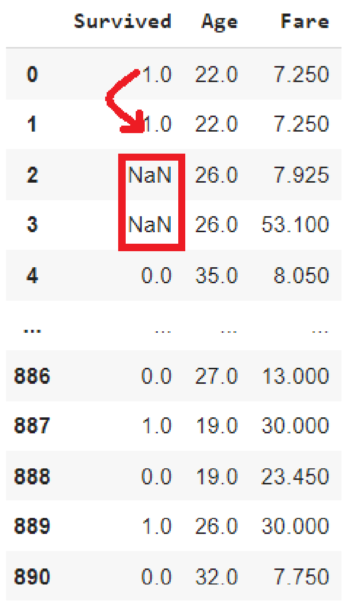
	Survived	Age	Fare
0	1.0	22.0	7.250
1	1.0	22.0	7.250
2	1.0	26.0	7.925
3	1.0	26.0	53.100
4	0.0	35.0	8.050
...	...	...	...
886	0.0	27.0	13.000
887	1.0	19.0	30.000
888	0.0	19.0	23.450
889	1.0	26.0	30.000
890	0.0	32.0	7.750

891 rows × 3 columns

If we use forward fill that simply means we are forwarding the previous value where ever we have NaN values.

5. Setting forward fill limit to 1

```
new_df = df.fillna(method="ffill",limit=1)
new_df
```

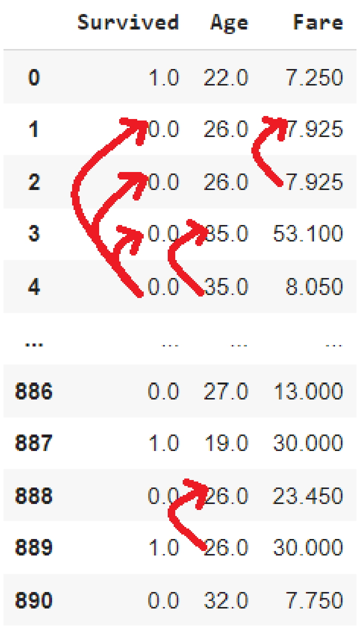


	Survived	Age	Fare
0	1.0	22.0	7.250
1	1.0	22.0	7.250
2	NaN	26.0	7.925
3	NaN	26.0	53.100
4	0.0	35.0	8.050
...	...	...	...
886	0.0	27.0	13.000
887	1.0	19.0	30.000
888	0.0	19.0	23.450
889	1.0	26.0	30.000
890	0.0	32.0	7.750

Now we have set the limit of forward fill to 1 which means that only once, the value will be copied below. Like in this case we had three NaN values consecutively in column Survived. But one NaN value was filled only as the limit is set to 1.

6. Filling NaN values in Backward Direction

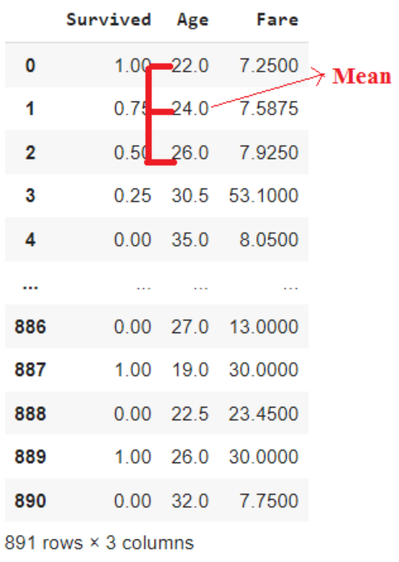
```
new_df = df.fillna(method="bfill")new_df
```



	Survived	Age	Fare
0	1.0	22.0	7.250
1	0.0	26.0	7.925
2	0.0	26.0	7.925
3	0.0	35.0	53.100
4	0.0	35.0	8.050
...	...	...	...
886	0.0	27.0	13.000
887	1.0	19.0	30.000
888	0.0	26.0	23.450
889	1.0	26.0	30.000
890	0.0	32.0	7.750

7. Interpolate of missing values

```
new_df = df.interpolate()

df
```



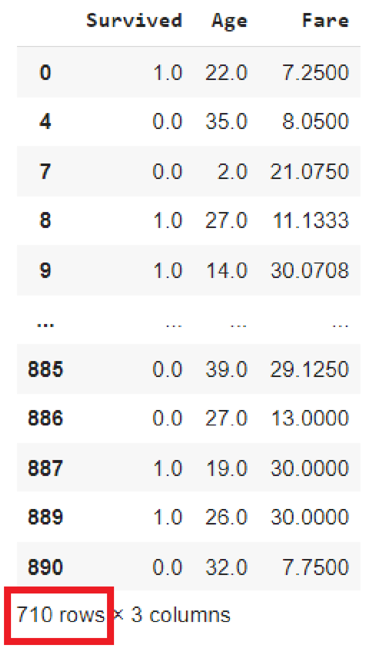
	Survived	Age	Fare
0	1.00	22.0	7.2500
1	0.75	24.0	7.5875
2	0.50	26.0	7.9250
3	0.25	30.5	53.1000
4	0.00	35.0	8.0500
...	...	...	...
886	0.00	27.0	13.0000
887	1.00	19.0	30.0000
888	0.00	22.5	23.4500
889	1.00	26.0	30.0000
890	0.00	32.0	7.7500

891 rows × 3 columns

In this, we were having two values 22 and 26. And in between value was a NaN value. So that NaN value is computed by getting the mean of 22 and 26 i.e. 24. In the same way, other NaN values were also computed.

8. Dropna()

```
new_df = df.dropna()
new_df
```



	Survived	Age	Fare
0	1.0	22.0	7.2500
4	0.0	35.0	8.0500
7	0.0	2.0	21.0750
8	1.0	27.0	11.1333
9	1.0	14.0	30.0708
...	...	...	...
885	0.0	39.0	29.1250
886	0.0	27.0	13.0000
887	1.0	19.0	30.0000
889	1.0	26.0	30.0000
890	0.0	32.0	7.7500

710 rows × 3 columns

Previously we were having 891 rows and after running this code we are left with 710 rows because some of the rows were continuing NaN values were dropped.

9. **Deleting the rows having all NaN values**

```
new_df = df.dropna(how='all')

new_df
```

Those rows in which **all the values are NaN** values will be deleted. If the row even has one value even then it will not be dropped.

Outliers

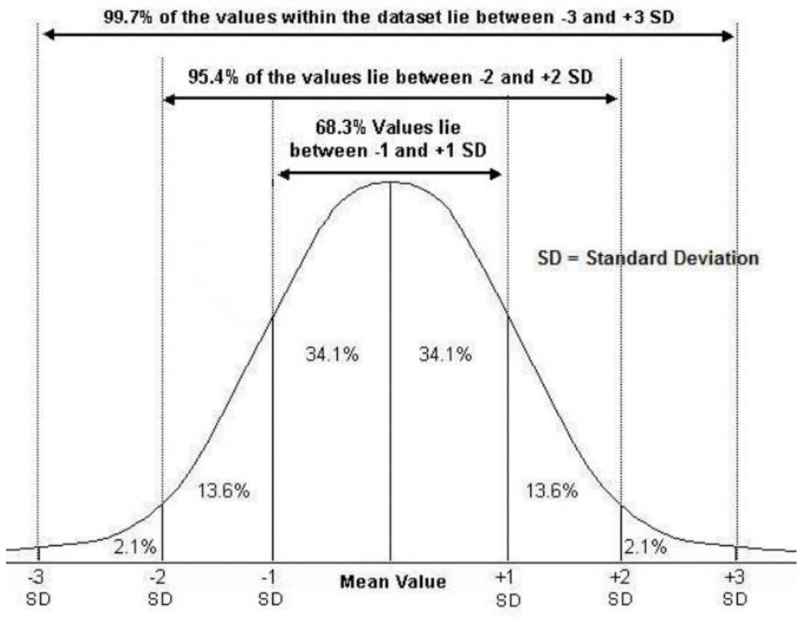
Outliers are those data points that are *significantly* different from the rest of the dataset. They are often abnormal observations that skew the data distribution, and arise due to inconsistent data entry, or erroneous observations.

Univariate outlier detection:

Univariate outliers are the data points whose values lie beyond the range of expected values based on one variable.

1. Using Standard Deviations Rule of Normal Distribution or Z-score method.

A z-score describes the position of a raw score in terms of its distance from the mean, when measured in standard deviation units. The z-score is positive if the value lies above the mean, and negative if it lies below the mean. It is also known as a standard score, because it allows comparison of scores on different kinds of variables by standardizing the distribution. The formula for calculating a z-score is is $z = (x-\mu)/\sigma$, where x is the raw score, μ is the population mean, and σ is the population standard deviation.
A limitation of this method is that it can only be used when the data is 'not' strongly skewed. It requires the data to be close to normal.



As in the figure above, if data is normally distributed, 99.7% of the values of the data should lie between +/- 3 standard deviations and 95.4% of the values within +/-2 standard deviations of the data. So, we may consider any data points away from +/- 2 or 3 standard deviations as outliers.
This can be achieved by calculating the *standard scores or z-scores* of the data points. Wikipedia states a z-score as:
*The **standard score** or **z-score** is the signed number of standard deviations by which the value of an observation or data point is above the mean value of what is being observed or measured.*
We calculate the z-score by subtracting the mean of all the data values from a data point and dividing by standard deviation.

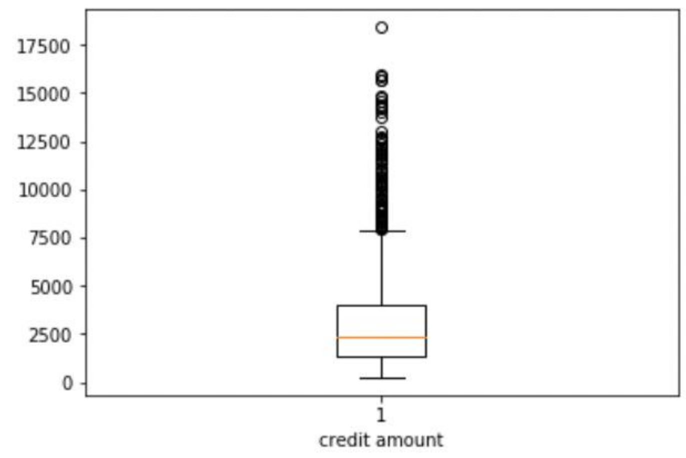
Example:

```
# Calculating the Zscore
df['cgpa_zscore'] = (df['cgpa'] -df['cgpa'].mean())/
df['cgpa'].std()
df[(df['cgpa_zscore'] > 3) | (df['cgpa_zscore'] < -3)]
```

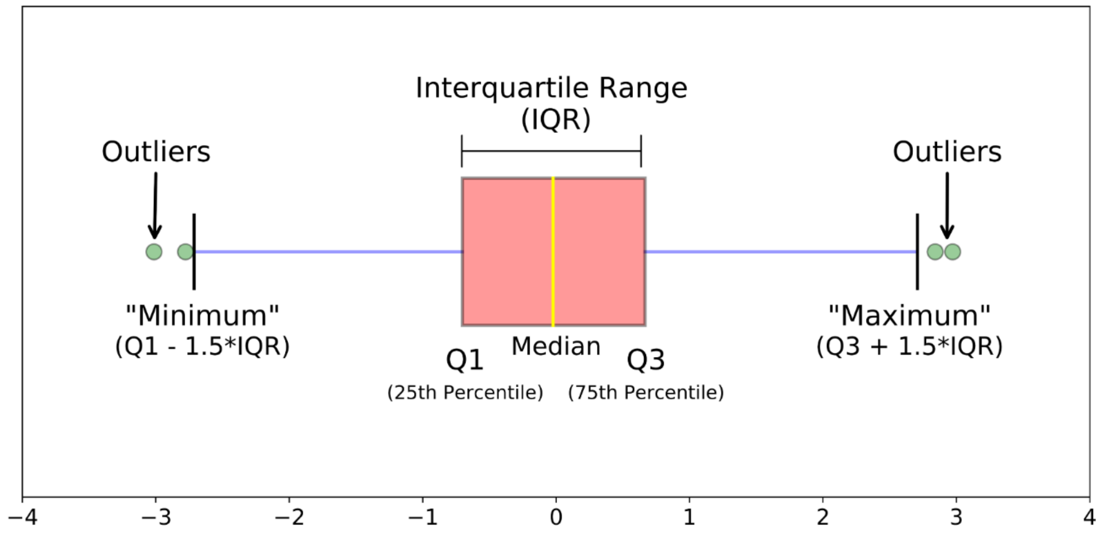
2.IQR and Box-and-Whisker's plot

A robust method for labeling outliers is the IQR (Inter Quartile Range) of exploratory data analysis. Box-and-Whiskers plot uses quartiles to plot the shape of a variable. The interquartile range is the range between the first and the third quartiles (the edges of the box). Any data point that falls outside of either 1.5 times the IQR below the first quartile or 1.5 times the IQR above the third quartile is considered an outlier.

Box- and-Whisker's plot for *Credit_Amount* variable



The box represents the 1st and 3rd quartiles, which are equal to the 25th and 75th percentiles. The line inside the box represents the 2nd quartile, which is the median.
If IQR = quartile_3 — quartile_1, then the lower is 'quartile_1 — (1.5 times IQR)' and the upper bound is 'quartile_3 + (1.5 times IQR)'. So, anything value below the lower bound and above the upper bound is considered as an outlier.



Example:

```
Q1 = df['placement_exam_marks'].quantile(0.25)
Q3 = df['placement_exam_marks'].quantile(0.75)
iqr = Q3 - Q1
upper_limit = Q3 + 1.5 * iqr
lower_limit = Q1 - 1.5 * iqr
df[df['placement_exam_marks'] > upper_limit]
df[df['placement_exam_marks'] < lower_limit]
```

Bivariate outlier detection

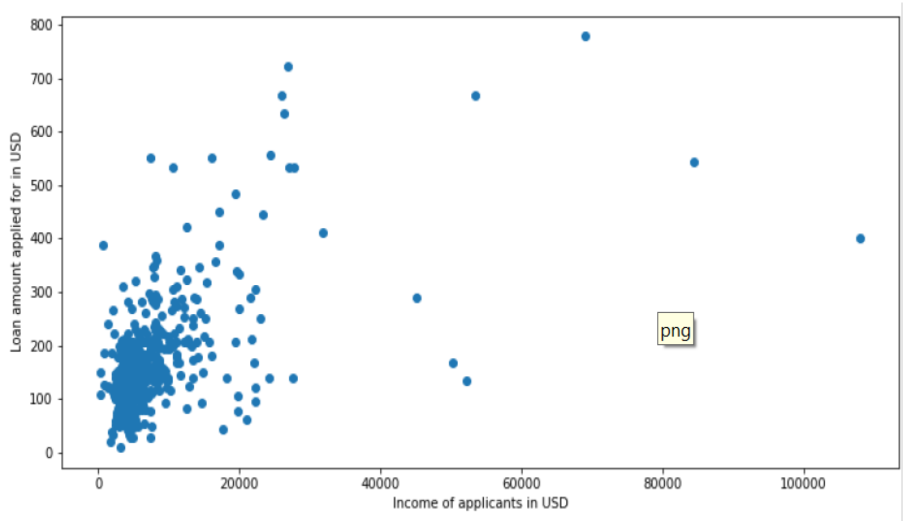
While plotting data, some values of one variable may not lie beyond the expected range, but when you plot the data with some other variable, these values may lie far from the expected value. These data point are bivariate outliers.

1. Scatterplot

A scatterplot visualizes the relationship between two quantitative variables. The data are displayed as a collection of points, and any points that fall outside the general clustering of the two variables may indicate outliers. The lines of code below generate a scatterplot between the variables 'Income' and 'Loan_amount'.

Example:

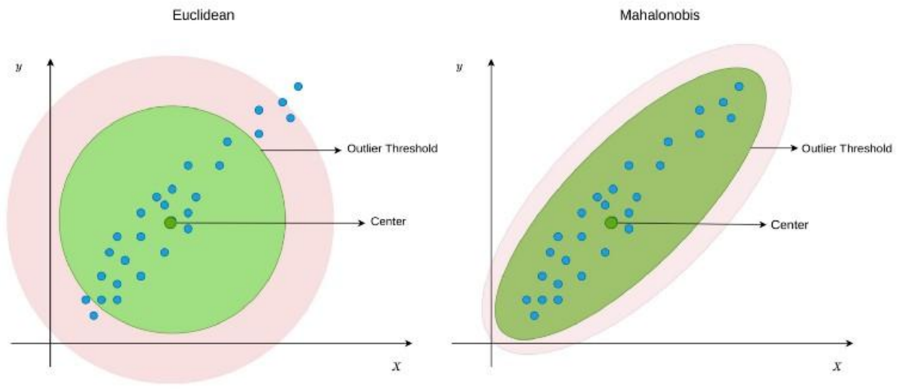
```
fig, ax = plt.subplots(figsize=(12,6))
ax.scatter(df['Income'], df['Loan_amount'])
ax.set_xlabel('Income of applicants in USD')
ax.set_ylabel('Loan amount applied for in USD')
plt.show()
```



2. Euclidean distance: **Euclidean distance** is one of the most known distance metrics to identify outliers based on their distance to the center point. There is also a Z-Score to define outliers for a single numeric variable. In some cases, clustering algorithms can be also preferred. All these methods consider outliers from different perspectives. The outliers are found based on one method may not be found by the others as outliers. Therefore, these methods and metrics should be chosen by considering the distribution of the variables. However, this brings out the needs of different metrics too. In this article, we will be discussing the distance metric called Mahalanobis Distance for detecting outliers in multivariable data.

3. Mahalanobis Distance

Mahalanobis Distance (MD) is an effective distance metric that finds the distance between the point and distribution. It works quite effectively on multivariate data because it uses a covariance matrix of variables to find the distance between data points and the center. This means that MD detects outliers based on the distribution pattern of data points, unlike the Euclidean distance. Please see Figure 1 to understand the difference.



Dealing with outliers

1. Do nothing: some times the number of outliers is very less then we can simply ignore the outliers.

2. Remove data objects with outliers (Trimming): In this method, we removed and completely drop all the rows which have outliers. Although it is not a good practice to follow.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
df = pd.read_csv('dataset/placement.csv')
df.head()
Q1 = df['placement_exam_marks'].quantile(0.25)
Q3 = df['placement_exam_marks'].quantile(0.75)
iqr = Q3 - Q1
upper_limit = Q3 + 1.5 * iqr
```

```
lower_limit = Q1 - 1.5 * iqr
df[df['placement_exam_marks'] > upper_limit]
df[df['placement_exam_marks'] < lower_limit]
index = df[(df['placement_exam_marks'] >= upper_limit)|
(df['placement_exam_marks'] <= lower_limit)].index
df.drop(index, inplace=True)
df['placement_exam_marks'].describe()
```

3. Replace with the upper cap or lower cap: In this quantile-based technique, we will do the flooring (e.g 10th percentile) for the lower values and capping (e.g for the 90th percentile) for the higher values. These percentile values will be used for the quantile-based flooring and capping.

```
import numpy as np
import pandas as pd
df = pd.read_csv('dataset/weight-height.csv')
df.shape
df['Height'].describe()
upper_limit = df['Height'].quantile(0.9)
upper_limit
lower_limit = df['Height'].quantile(0.1)
lower_limit
df['Height'] = np.where(df['Height'] >= upper_limit,
        upper_limit,
        np.where(df['Height'] <= lower_limit,
        lower_limit,
        df['Height']))
df.shape
```

4. Perform a log transformation

Transformation of the skewed variables may also help correct the distribution of the variables. These could be logarithmic, square root, or square transformations. The most common is the logarithmic transformation, which is done on the 'Loan_amount' variable in the first line of code below. The second and third lines of code print the skewness value before and after the transformation.

```
df["Log_Loanamt"] = df["Loan_amount"].map(lambda i: np.log(i)
if i > 0 else 0)
print(df['Loan_amount'].skew())
print(df['Log_Loanamt'].skew())
```

Output:

```
2.8146019248106815
```

-0.17792641310111373

Data integration

- Data Integration is a technique of integrating the data which resides in different sources. The goal is to provide the users with a holistic view of the data. It can be viewed more as a practice of consolidating data from various disparate sources. This is viewed as one of the most important steps in Data preprocessing.
- It is a strategy that integrates data from several sources to make it available to users in a single uniform view that shows their status. There are communication sources between systems that can include multiple databases, data cubes, or flat files. Data fusion merges data from various diverse sources to produce meaningful results. The consolidated findings must exclude inconsistencies, contradictions, redundancies, and inequities.
- Data integration is important because it gives a uniform view of scattered data while also maintaining data accuracy.

Example:

```
import pandas as pd
df1 = pd.read_csv('dataset/student.csv')
df1.info()
import pandas as pd
df2 = pd.read_csv('dataset/marks1.csv')
df2.info()
df_merge = pd.merge(df1,df2, on = 'Reg_no')
df_merge
```

Data integration challenges

1. Entity Identification Problem

As we know the data is unified from the heterogeneous sources then how can we 'match the real-world entities from the data'. For example, we have customer data from two different data source. An entity from one data source has customer_id and the entity from the other data source has customer_number. Now how does the data analyst or the system would understand that these two entities refer to the same attribute?

2. Redundancy and Correlation Analysis

Redundancy is one of the big issues during data integration. Redundant data is an unimportant data or the data that is no longer needed. It can also arise due to attributes that could be derived using another attribute in the data set.

For example, one data set has the customer age and other data set has the customers date of birth then age would be a redundant attribute as it could be derived using the date of birth.

Inconsistencies in the attribute also raise the level of redundancy. The redundancy can be discovered using correlation analysis. The attributes are analyzed to detect their interdependency on each other thereby detecting the correlation between them.

3. Tuple Duplication

Along with redundancies data integration has also deal with the duplicate rows. Duplicate rows may come in the resultant data if the denormalized table has been used as a source for data integration.

4. Data Conflict Detection and Resolution

Data conflict means the data merged from the different sources do not match. Like the attribute values may differ in different data sets. The difference maybe because they are represented differently in the different data sets. For suppose the price of a hotel room may be represented in different currencies in different cities. This kind of issues is detected and resolved during data integration.

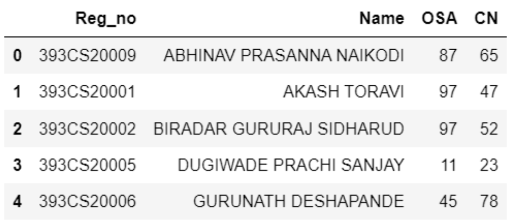
Approaches

1. Adding attributes:

Create a new column in Pandas DataFrame based on the existing columns

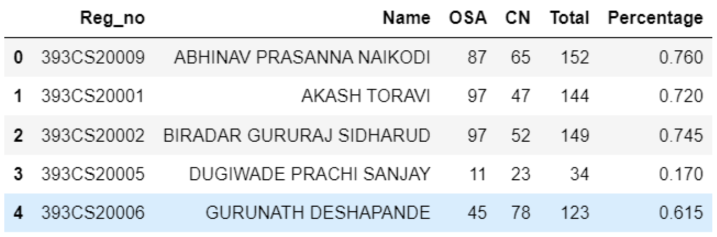
While working with data in Pandas, we perform a vast array of operations on the data to get the data in the desired form. One of these operations could be that we want to create new columns in the DataFrame based on the result of some operations on the existing columns in the DataFrame.

Example:



	Reg_no	Name	OSA	CN
0	393CS20009	ABHINAV PRASANNA NAIKODI	87	65
1	393CS20001	AKASH TORAVI	97	47
2	393CS20002	BIRADAR GURURAJ SIDHARUD	97	52
3	393CS20005	DUGIWADE PRACHI SANJAY	11	23
4	393CS20006	GURUNATH DESHAPANDE	45	78

```
import pandas as pd
df = pd.read_csv('dataset/details.csv')
df.head()
df['Total'] = df['OSA'] + df['CN']
df['Percentage'] = df['Total']/200
df.head()
```



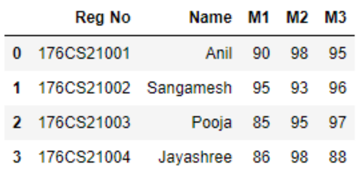
	Reg_no	Name	OSA	CN	Total	Percentage
0	393CS20009	ABHINAV PRASANNA NAIKODI	87	65	152	0.760
1	393CS20001	AKASH TORAVI	97	47	144	0.720
2	393CS20002	BIRADAR GURURAJ SIDHARUD	97	52	149	0.745
3	393CS20005	DUGIWADE PRACHI SANJAY	11	23	34	0.170
4	393CS20006	GURUNATH DESHAPANDE	45	78	123	0.615

2. Adding data objects

Add one row in an existing Pandas DataFrame

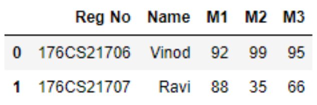
Let us see how to add a new row of values to an existing dataframe. This can be used when we want to insert a new entry in our data that we might have missed adding earlier. There are different methods to achieve this.

```
import pandas as pd
df=pd.read_csv('1.csv')
df
```



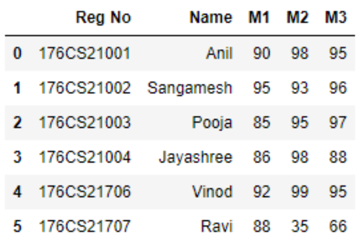
	Reg No	Name	M1	M2	M3
0	176CS21001	Anil	90	98	95
1	176CS21002	Sangamesh	95	93	96
2	176CS21003	Pooja	85	95	97
3	176CS21004	Jayashree	86	98	88

```
df1=pd.read_csv('2.csv')
df1
```



	Reg No	Name	M1	M2	M3
0	176CS21706	Vinod	92	99	95
1	176CS21707	Ravi	88	35	66

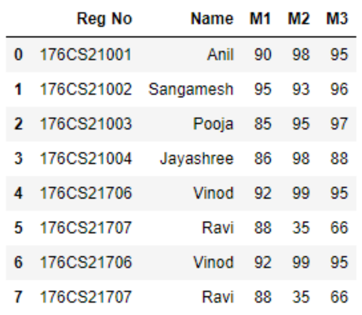
```
df=df.append(df1, ignore_index=True)
df
```



	Reg No	Name	M1	M2	M3
0	176CS21001	Anil	90	98	95
1	176CS21002	Sangamesh	95	93	96
2	176CS21003	Pooja	85	95	97
3	176CS21004	Jayashree	86	98	88
4	176CS21706	Vinod	92	99	95
5	176CS21707	Ravi	88	35	66

Using Concat Method:

```
df3 = pd.concat([df, df1], ignore_index = True)
df3
```



	Reg No	Name	M1	M2	M3
0	176CS21001	Anil	90	98	95
1	176CS21002	Sangamesh	95	93	96
2	176CS21003	Pooja	85	95	97
3	176CS21004	Jayashree	86	98	88
4	176CS21706	Vinod	92	99	95
5	176CS21707	Ravi	88	35	66
6	176CS21706	Vinod	92	99	95
7	176CS21707	Ravi	88	35	66

Data reduction

Data reduction is the transformation of numerical or alphabetical digital information derived empirically or experimentally into a corrected, ordered, and simplified form. The purpose of data reduction can be two-fold: reduce the number of data records by eliminating invalid data or produce summary data and statistics at different aggregation levels for various applications.

The data reduction approach decreases the amount of data while preserving its integrity. Data reduction has no effect on the outcome of data mining, which implies that the outcome of data mining before and after data reduction is the same (or almost the same)

Why Data Reduction?

Data reduction process reduces the size of data and makes it suitable and feasible for analysis. In the reduction process, integrity of the data must be preserved and data volume is reduced. There are many techniques that can be used for data reduction.

Difference between Data Redundancy and Data Inconsistency

Data Redundancy: It is defined as the redundancy means duplicate data and it is also stated that the same parts of data exist in multiple locations into the database. This condition is known as Data Redundancy.

Problems with Data Redundancy: Here, we will discuss the few problems with data redundancy as follows.

1. Wasted Storage Space.
2. More Difficult Database Update.
3. It will lead to Data Inconsistency.
4. Retrieval of data is slow and inefficient.

Example – **Let us take an example of a cricket player table.**

Step-1 : Consider cricket player table as follows.

Player Name	**Player Age**	**Team Name**	**Team ID**

Virat Kohli	32	India	1
Rohit Sharma	34	India	1
Ross Taylor	37	New Zealand	2
Shikhar Dhawan	35	India	1
Kane Williamson	30	New Zealand	2

Step-2 : We can clearly see that the Team Name and Team Id are repeated at multiple places. we can make a separate table to store this information and reduce data redundancy.

Player Name	**Player Age**	**Team Id**
Virat Kohli	32	1
Rohit Sharma	34	1
Ross Taylor	37	2
Shikhar Dhawan	35	1
Kane Williamson	30	2

Step-3 : This is known as Normalization used to reduce Data Redundancy.

Team Id	Team Name
1	India
2	New Zealand

Data Inconsistency:

When the same data exists in different formats in multiple tables. This condition is known as Data Inconsistency. It means that different files contain different information about a particular object or person. This can cause unreliable and meaningless information. Data Redundancy leads to Data Inconsistency.

Example – If we have an address of someone in many tables and when we change it in only one table and in another table it may not be updated so there is the problem of data inconsistency may occur.

Topic	**Data Redundancy**	**Data Inconsistency**
Condition	It will be applicable when the duplicate data exists in multiple places in the database.	It will be applicable when the duplicate data exists in different formats in multiple tables.
How to minimize it?	we can use normalization to minimize Data Redundancy.	we can use constraints on the database to minimize Data Inconsistency

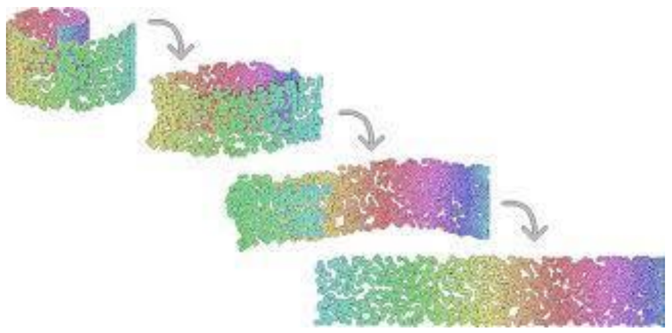
Objectives of Data Reduction

1. Increases Building Efficiency: Data analytics helps to reduce construction time and material-related costs by providing clear data and identifying potential structural errors before they happen.
2. Reduces Environmental Impact: The construction industry is one of the biggest polluters in the world. So we need to use eco-friendly building materials and do sustainable construction practices. Data analysis can help to solve this problem.
3. Promotes Collaboration: Lack of communication is one of the biggest problems construction professionals face while doing their duties. Fortunately, data analysis makes information accessible and easily shareable between crew members.
4. Improves Working Conditions: Data collected from health tracking software is used to improve working conditions on site.

Methods of Data Reduction

Dimensionality Reduction

When dimensionality increases, data becomes increasingly sparse while density and distance between points, critical to clustering and outlier analysis, becomes less meaningful. Dimensionality reduction helps reduce noise in the data and allows for easier visualization, such as the example below where 3-dimensional data is transformed into 2 dimensions to show hidden parts. One method of dimensionality reduction is wavelet transform, in which data is transformed to preserver relative distance between objects at different levels of resolution, and is often used for image compression.

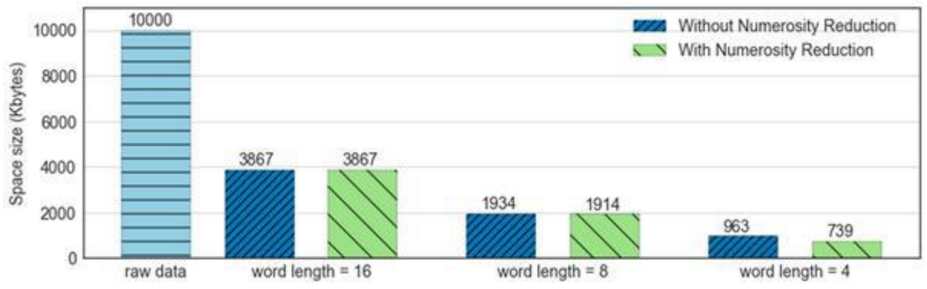


Numerosity Reduction

Numerosity Reduction is a data reduction technique which replaces the original data by smaller form of data representation. There are two techniques for numerosity reduction- Parametric and Non-Parametric methods.

This method of data reduction reduces the data volume by choosing alternate, smaller forms of data representation. Numerosity reduction can be split into 2 groups: parametric and non-parametric methods. Parametric methods (regression, for example) assume the data fits some model, estimate model parameters, store only the parameters, and discard the data. One example of this is in the image below, where the volume of data to be processed is reduced based on more specific criteria.

Another example would be a log-linear model, obtaining a value at a point in m-D space as the product on appropriate marginal subspaces. Non-parametric methods do not assume models, some examples being histograms, clustering, sampling, etc.



Data transformation

Data transformation is a technique used to convert the raw data into a suitable format that efficiently eases data mining and retrieves strategic information. Data transformation includes data cleaning techniques and a data reduction technique to convert the data into the appropriate form.

Data transformation changes the format, structure, or values of the data and converts them into clean, usable data.

Need for data transformation:

Organizations generate a huge amount of data daily. However, it is of no value unless it can be used to gather insights and drive business growth. Organizations utilize data transformation to convert data into formats that can then be used for several processes. There are a few reasons why organizations should transform their data.

- Transformation makes disparate sets of data compatible with each other, which makes it easier to aggregate data for a thorough analysis
- Migration of data is easier since the source format can be transformed into the target format
- Data transformation helps in consolidating data, structured and unstructured
- The process of transformation also allows for enrichment which enhances the quality of data

The ultimate goal is consistent, accessible data that provides organizations with accurate analytic insights and predictions.

Benefits of Data Transformation

Data holds the potential to directly affect an organization's efficiencies and its bottom line. It plays a crucial role in understanding customer behavior, internal processes, and industry trends. While every organization has the ability to collect an immense amount of data, the challenge is to ensure that this is usable. Data transformation processes empower organizations to reap the benefits offered by the data.

1.Data Utilization
If the data being collected isn't in an appropriate format, it often ends up not being utilized at all. With the help of data transformation tools, organizations can finally realize the true potential of the data they have amassed since the transformation process standardizes the data and improves its usability and accessibility.

2.Data Consistency
Data is continuously being collected from a range of sources which increases the inconsistencies in metadata. This makes organization and understanding data a huge challenge. Data transformation helps making it simpler to understand and organize data sets.

3.Better Quality Data
Transformation process also enhances the quality of data which can then be utilized to acquire business intelligence.

4.Compatibility Across Platforms
Data transformation also supports compatibility between types of data, applications and systems.
5.Faster Data Access
It is quicker and easier to retrieve data that has been transformed into a standardized format.

6.More Accurate Insights and Predictions
The transformation process generates data models which are then converted to metrics, dashboards and reports which enable organizations to achieve specific goals. The metrics and key performance indicators help businesses quantify their efforts and analyze their progress. After being transformed, data can be used for many use cases, including:

- Analytics which use metrics from one or many sources to gain deeper insights about the functions and operations of any organization. Transformation of data is required when the metric combines data from multiple sources.
- Machine learning which helps businesses with their profit and revenue projections, supports their decision making with predictive modeling, and automation of several business processes.
- Regulatory compliance which involves sensitive data that is vulnerable to malicious attacks

What is Normalization?

Normalization is a scaling technique in which values are shifted and rescaled so that they end up ranging between 0 and 1. It is also known as Min-Max scaling.

Here's the formula for normalization:

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Here, Xmax and Xmin are the maximum and the minimum values of the feature respectively.

- When the value of X is the minimum value in the column, the numerator will be 0, and hence X' is 0
- On the other hand, when the value of X is the maximum value in the column, the numerator is equal to the denominator and thus the value of X' is 1
- If the value of X is between the minimum and the maximum value, then the value of X' is between 0 and 1

What is Standardization?

Standardization is another scaling technique where the values are centered around the mean with a unit standard deviation. This means that the mean of the attribute becomes zero and the resultant distribution has a unit standard deviation.

Here's the formula for standardization:

$$X' = \frac{X - \mu}{\sigma}$$

μ is the mean of the feature values and σ is the standard deviation of the feature values. Note that in this case, the values are not restricted to a particular range.

Now, the big question in your mind must be when should we use normalization and when should we use standardization? Let's find out!

The Big Question – Normalize or Standardize?

Normalization vs. standardization is an eternal question among machine learning newcomers. Let me elaborate on the answer in this section.

- Normalization is good to use when you know that the distribution of your data does not follow a Gaussian distribution. This can be useful in algorithms that do not assume any distribution of the data like K-Nearest Neighbors and Neural Networks.
- Standardization, on the other hand, can be helpful in cases where the data follows a Gaussian distribution. However, this does not have to be necessarily true. Also, unlike normalization, standardization does not have a bounding range. So, even if you have outliers in your data, they will not be affected by standardization.

However, at the end of the day, the choice of using normalization or standardization will depend on your problem and the machine learning algorithm you are using. There is no hard and fast rule to tell you when to normalize or standardize your data. You can always start by fitting your model to raw, normalized and standardized data and compare the performance for best results.

Normalization or MinMax Scalar Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

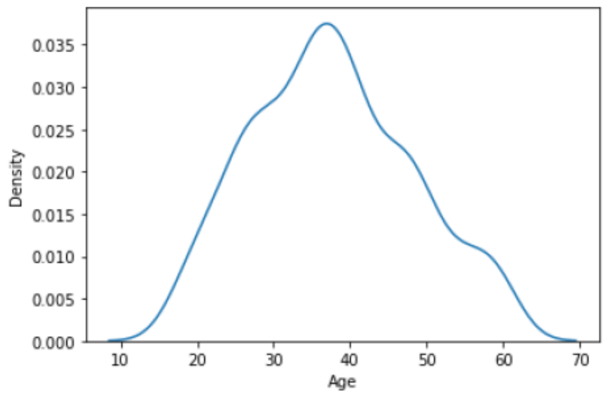
```
df = pd.read_csv('dataset/Social_Network_Ads.csv',usecols=[2,3,4])
```

```
df.head()
```

	Age	EstimatedSalary	Purchased
0	19	19000	0
1	35	20000	0
2	26	43000	0
3	27	57000	0
4	19	76000	0

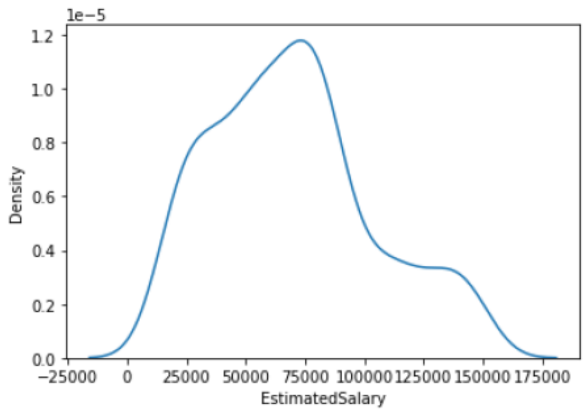
```
sns.kdeplot(df['Age'])
```

```
<AxesSubplot:xlabel='Age', ylabel='Density'>
```



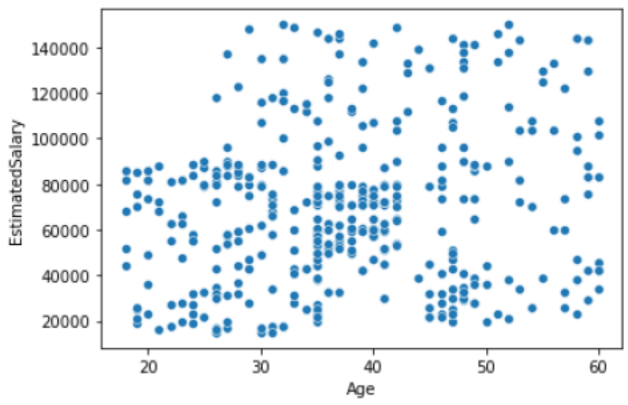
```
sns.kdeplot(df['EstimatedSalary'])
```

```
<AxesSubplot:xlabel='EstimatedSalary', ylabel='Density'>
```



```
color_dict={1:'red',3:'green',2:'blue'}
sns.scatterplot(df['Age'],df['EstimatedSalary'],palette=color_dict)
```

```
<AxesSubplot:xlabel='Age', ylabel='EstimatedSalary'>
```



```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
# fit the scaler to the dataset set, it will learn the parameters
scaler.fit(df)
# transform data sets
X_scaled = scaler.transform(df)
```

```
X_scaled = pd.DataFrame(X_scaled, columns=df.columns)
X_scaled.head()
```

	Age	EstimatedSalary	Purchased
0	0.023810	0.029630	0.0
1	0.404762	0.037037	0.0
2	0.190476	0.207407	0.0
3	0.214286	0.311111	0.0
4	0.023810	0.451852	0.0

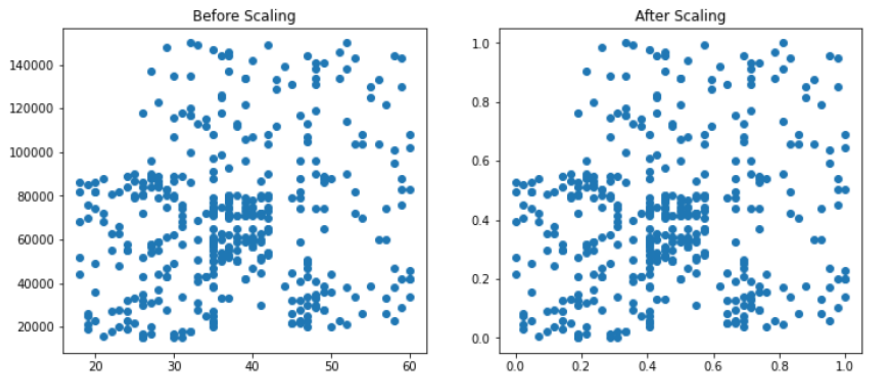
```
np.round(df.describe(), 1)
```

	Age	EstimatedSalary	Purchased
count	400.0	400.0	400.0
mean	37.7	69742.5	0.4
std	10.5	34097.0	0.5
min	18.0	15000.0	0.0
25%	29.8	43000.0	0.0
50%	37.0	70000.0	0.0
75%	46.0	88000.0	1.0
max	60.0	150000.0	1.0

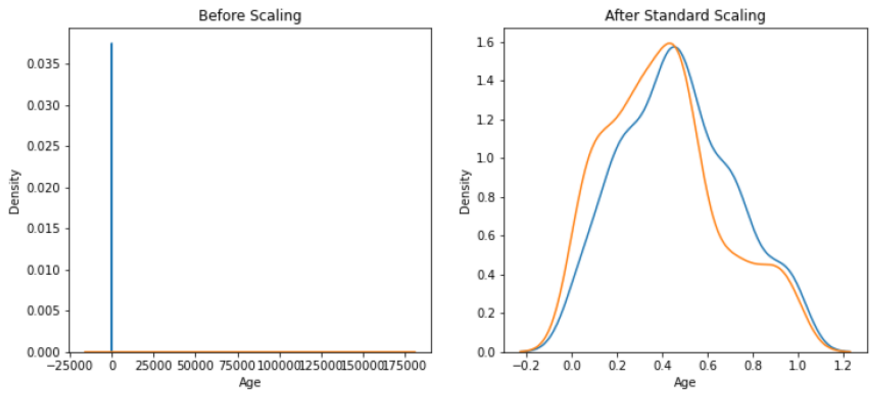
```
np.round(X_scaled.describe(), 1)
```

	Age	EstimatedSalary	Purchased
count	400.0	400.0	400.0
mean	0.5	0.4	0.4
std	0.2	0.3	0.5
min	0.0	0.0	0.0
25%	0.3	0.2	0.0
50%	0.5	0.4	0.0
75%	0.7	0.5	1.0
max	1.0	1.0	1.0

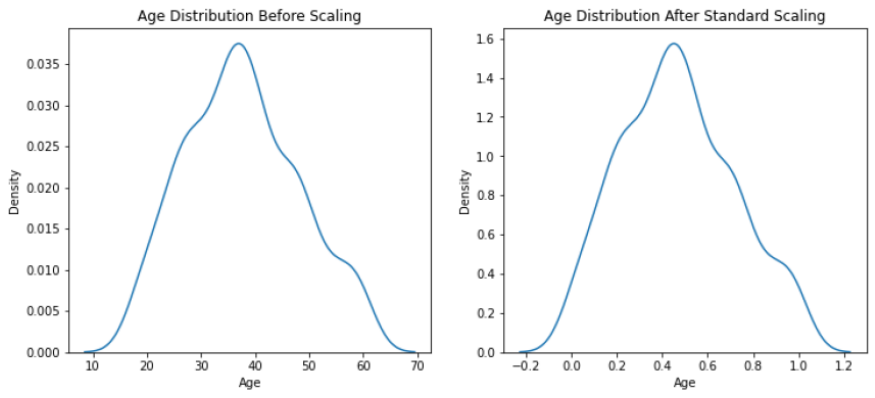
```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(12, 5))
ax1.scatter(df['Age'], df['EstimatedSalary'])
ax1.set_title("Before Scaling")
ax2.scatter(X_scaled['Age'], X_scaled['EstimatedSalary'])
ax2.set_title("After Scaling")
plt.show()
```



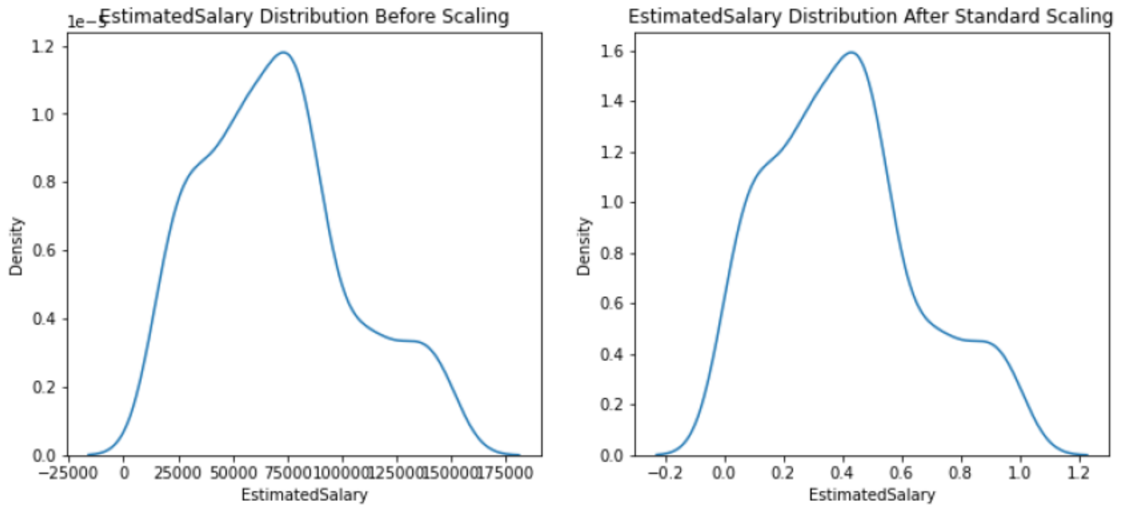
```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(12, 5))
# before scaling
ax1.set_title('Before Scaling')
sns.kdeplot(df['Age'], ax=ax1)
sns.kdeplot(df['EstimatedSalary'], ax=ax1)
# after scaling
ax2.set_title('After Standard Scaling')
sns.kdeplot(X_scaled['Age'], ax=ax2)
sns.kdeplot(X_scaled['EstimatedSalary'], ax=ax2)
plt.show()
```



```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(12, 5))
# before scaling
ax1.set_title('Age Distribution Before Scaling')
sns.kdeplot(df['Age'], ax=ax1)
# after scaling
ax2.set_title('Age Distribution After Standard Scaling')
sns.kdeplot(X_scaled['Age'], ax=ax2)
plt.show()
```



```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(12, 5))
# before scaling
ax1.set_title('EstimatedSalary Distribution Before Scaling')
sns.kdeplot(df['EstimatedSalary'], ax=ax1)
# after scaling
ax2.set_title('EstimatedSalary Distribution After Standard Scaling')
sns.kdeplot(X_scaled['EstimatedSalary'], ax=ax2)
plt.show()
```



```
df.head(5)
```

	Age	EstimatedSalary	Purchased
0	19	19000	0
1	35	20000	0
2	26	43000	0
3	27	57000	0
4	19	76000	0

```
X_scaled.head()
```

	Age	EstimatedSalary	Purchased
0	0.023810	0.029630	0.0
1	0.404762	0.037037	0.0
2	0.190476	0.207407	0.0
3	0.214286	0.311111	0.0
4	0.023810	0.451852	0.0

Standardization: standard Scalar code:

```
import numpy as np # linear algebra
import pandas as pd # data processing
import matplotlib.pyplot as plt
import seaborn as sns
```

```
df = pd.read_csv('dataset/Social_Network_Ads.csv')
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 5 columns):
 #   Column           Non-Null Count  Dtype 
---  ------           --------------  ----- 
 0   User ID          400 non-null    int64 
 1   Gender           400 non-null    object
 2   Age              400 non-null    int64 
 3   EstimatedSalary  400 non-null    int64 
 4   Purchased        400 non-null    int64 
dtypes: int64(4), object(1)
memory usage: 15.8+ KB
```

```
df=df.iloc[:,2:]
```

StandardScaler

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(df)
X_scaled = scaler.transform(df)
```

```
df.head()
```

	Age	EstimatedSalary	Purchased
0	19	19000	0
1	35	20000	0
2	26	43000	0
3	27	57000	0
4	19	76000	0

```
X_scaled
```

```
array([[-1.78179743, -1.49004624, -0.74593581],
       [-0.25358736, -1.46068138, -0.74593581],
       [-1.11320552, -0.78528968, -0.74593581],
       ...,
       [ 1.17910958, -1.46068138,  1.34059793],
       [-0.15807423, -1.07893824, -0.74593581],
       [ 1.08359645, -0.99084367,  1.34059793]])
```

```
X_scaled = pd.DataFrame(X_scaled, columns=df.columns)
```

```
np.round(df.describe(), 1)
```

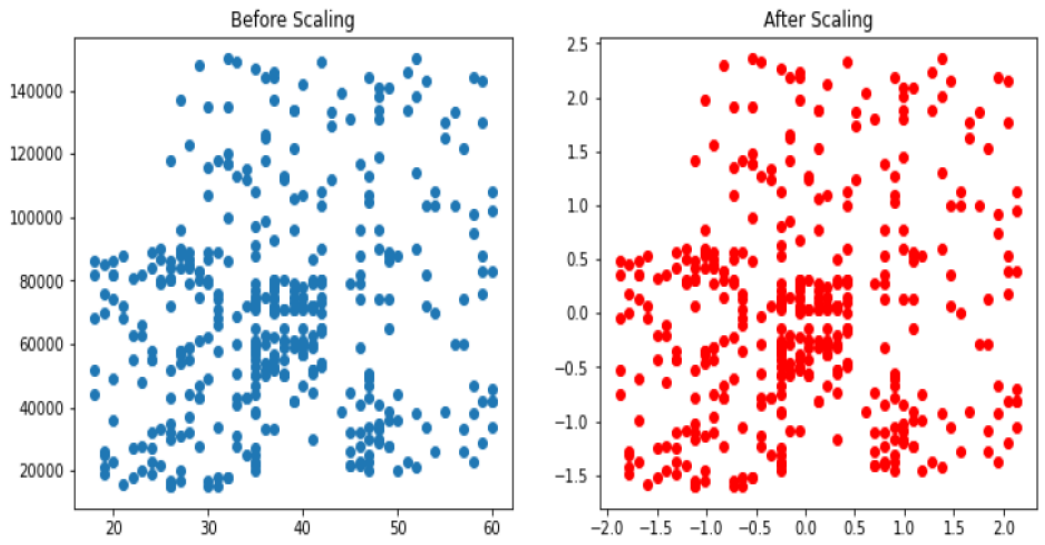
	Age	EstimatedSalary	Purchased
count	400.0	400.0	400.0
mean	37.7	69742.5	0.4
std	10.5	34097.0	0.5
min	18.0	15000.0	0.0
25%	29.8	43000.0	0.0
50%	37.0	70000.0	0.0
75%	46.0	88000.0	1.0
max	60.0	150000.0	1.0

```
np.round(X_scaled.describe(), 1)
```

	Age	EstimatedSalary	Purchased
count	400.0	400.0	400.0
mean	-0.0	-0.0	-0.0
std	1.0	1.0	1.0
min	-1.9	-1.6	-0.7
25%	-0.8	-0.8	-0.7
50%	-0.1	0.0	-0.7
75%	0.8	0.5	1.3
max	2.1	2.4	1.3

Effect of Scaling

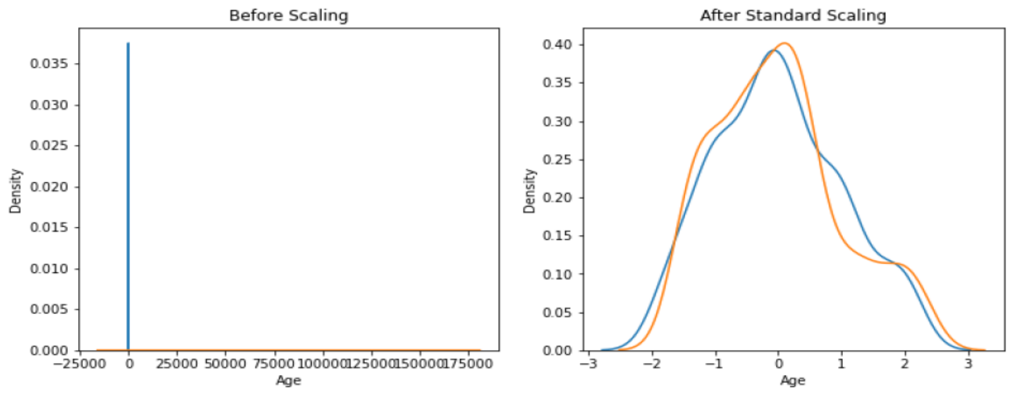
```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(12, 5))

ax1.scatter(df['Age'], df['EstimatedSalary'])
ax1.set_title("Before Scaling")
ax2.scatter(X_scaled['Age'], X_scaled['EstimatedSalary'],color='red')
ax2.set_title("After Scaling")
plt.show()
```



```
df.mean()
```

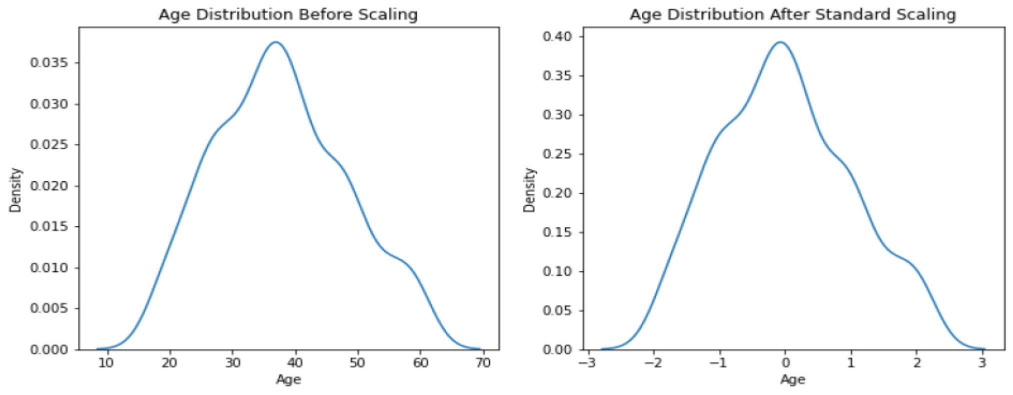
```
Age                  37.6550
EstimatedSalary   69742.5000
Purchased             0.3575
dtype: float64
```

```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(12, 5))

# before scaling
ax1.set_title('Before Scaling')
sns.kdeplot(df['Age'], ax=ax1)
sns.kdeplot(df['EstimatedSalary'], ax=ax1)

# after scaling
ax2.set_title('After Standard Scaling')
sns.kdeplot(X_scaled['Age'], ax=ax2)
sns.kdeplot(X_scaled['EstimatedSalary'], ax=ax2)
plt.show()
```

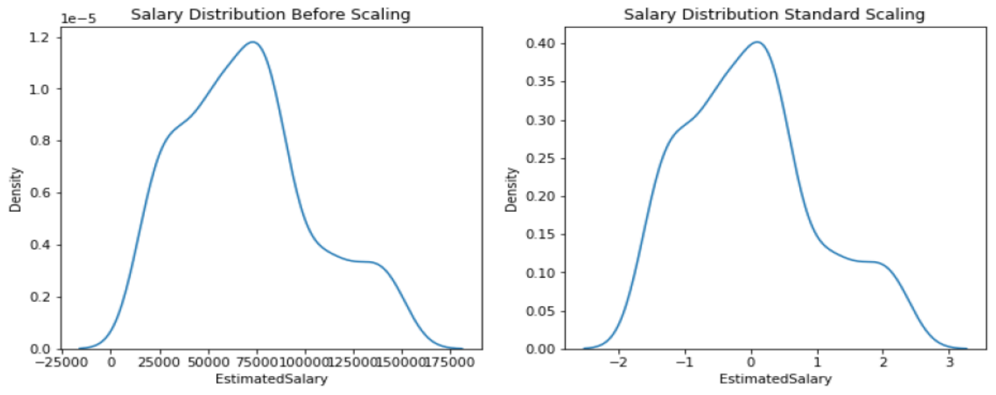


Comparison of Distributions

```
5]: fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(12, 5))

    # before scaling
    ax1.set_title('Age Distribution Before Scaling')
    sns.kdeplot(df['Age'], ax=ax1)

    # after scaling
    ax2.set_title('Age Distribution After Standard Scaling')
    sns.kdeplot(X_scaled['Age'], ax=ax2)
    plt.show()
```

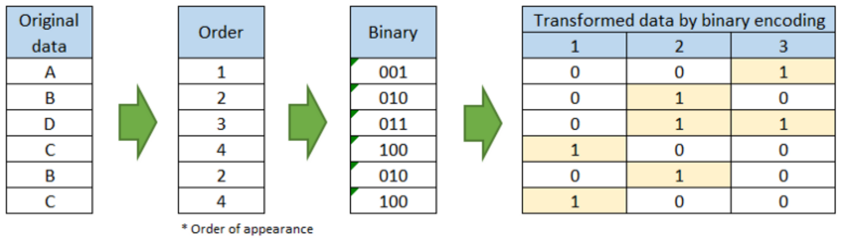


```
6]: fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(12, 5))

    # before scaling
    ax1.set_title('Salary Distribution Before Scaling')
    sns.kdeplot(df['EstimatedSalary'], ax=ax1)

    # after scaling
    ax2.set_title('Salary Distribution Standard Scaling')
    sns.kdeplot(X_scaled['EstimatedSalary'], ax=ax2)
    plt.show()
```



Data transformation with
Binary coding

Binary encoding is an approach to turn the categorical column to multiple binary columns while minimizing the number of new columns.
First, turn the categorical value to integers by some orders (e.g. alphabetical order or order of appearance for the top row). Next, turn it to binary digit such that 1 to 1, 2 to 10, 5 to 101, etc. Finally, split the binary digit into separate columns each of which has a single digit.



Original data
A
B
D
C
B
C

Order
1
2
3
4
2
4

* Order of appearance

Binary
001
010
011
100
010
100

Transformed data by binary encoding		
1	2	3
0	0	1
0	1	0
0	1	1
1	0	0
0	1	0
1	0	0

Illustration of binary encoding

Binary encoding can reduce the number of new columns to *log_2(number of levels)* order. As you can see in the example above, one of the new columns has 1 in different original level, which is **not a good thing** because the levels having 1 in the same column will be treated to share some property by the model while actually they have 1 in the same column just for a technical reason.

```
from category_encoders import BinaryEncoder
import pandas as pd

df_x = pd.DataFrame({'v1':['A','B','D','C','B','C'],
'v2':[10,11,2,0,30,50]})
be = BinaryEncoder()
df_be = be.fit_transform(df_x['v1'])
pd.concat([df_x,df_be],axis=1)
#        v1      v2      v1_0    v1_1    v1_2
# 0      A       10      0       0       1
# 1      B       11      0       1       0
# 2      D       2       0       1       1
# 3      C       0       1       0       0
# 4      B       30      0       1       0
# 5      C       50      1       0       0
```

Ranking transformation

Turning the raw numeric values to ranks of values. Ranks can be also turned to values between 0 and 1 with dividing by number of records, which makes the transformed amounts irrelevant to the number of records.
The book introduced an interesting example: when analyzing the number of customers in a shop, the numbers of customers on holidays are likely to be a lot more than the ones on weekdays. Then, using the raw number of customers may weight the holidays too highly. Turning the numbers to ranks can neutralize the impact of the absolute number of customers.
pandas rank function or numpy argsort function can handle this transformation.

Discretization
Data discretization refers to a method of converting a huge number of data values into smaller ones so that the evaluation and management of data become easy. In other words, data discretization is a method of converting attributes values of continuous data into a finite set of intervals with minimum data loss. There are two forms of data discretization first is supervised discretization, and the second is unsupervised discretization. Supervised discretization refers to a method in which the class data is used. Unsupervised discretization refers to a method depending upon the way which operation proceeds. It means it works on the top-down splitting strategy and bottom-up merging strategy.
Now, we can understand this concept with the help of an example
Suppose we have an attribute of Age with the given values

Age	1,5,9,4,7,11,14,17,13,18, 19,31,33,36,42,44,46,70,74,78,77

Table before Discretization

Attribute	Age	Age	Age	Age
	1,5,4,9,7	11,14,17,13,18,19	31,33,36,42,44,46	70,74,77,78
After Discretization	Child	Young	Mature	Old

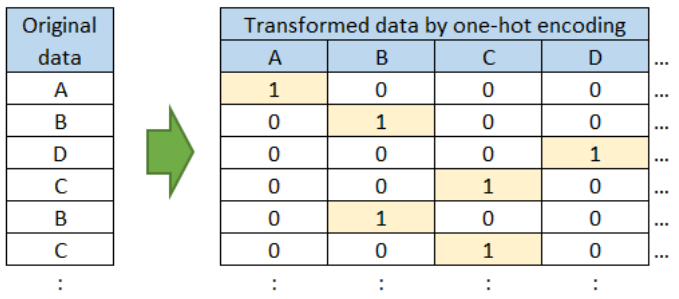
Another example is analytics, where we gather the static data of website visitors. For example, all visitors who visit the site with the IP address of India are shown under country level.

Some Famous techniques of data discretization
1.Histogram analysis
Histogram refers to a plot used to represent the underlying frequency distribution of a continuous data set. Histogram assists the data inspection for data distribution. For example, Outliers, skewness representation, normal distribution representation, etc.

2.Binning
Binning refers to a data smoothing technique that helps to group a huge number of continuous values into smaller values. For data discretization and the development of idea hierarchy, this technique can also be used.

3.Cluster Analysis
Cluster analysis is a form of data discretization. A clustering algorithm is executed by dividing the values of x numbers into clusters to isolate a computational feature of x.

One-hot encoding
One-hot encoding is an approach to convert one categorical column to multiple binary (0 or 1) columns as many as the number of distinct levels in the original column. If there are four levels on the categorical variable, one-hot encoding will create four new columns, each of which has 0 or 1 and represents if the original column has the level.



Original data
A
B
D
C
B
C
:

Transformed data by one-hot encoding				
A	B	C	D	…
1	0	0	0	…
0	1	0	0	…
0	0	0	1	…
0	0	1	0	…
0	1	0	0	…
0	0	1	0	…
:	:	:	:	

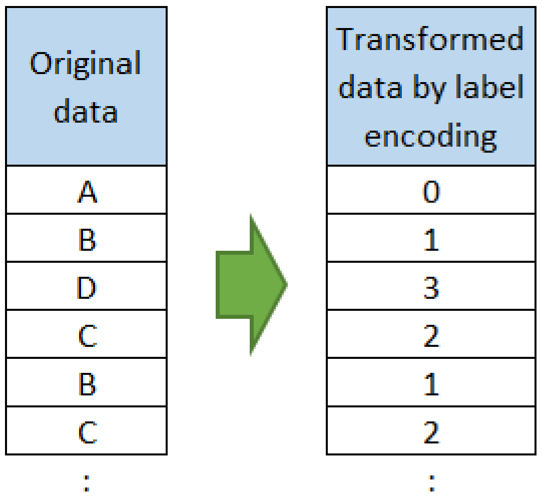
Illustration of one-hot encoding (four-level column (A-D) creates four new columns)

An apparent drawback of one-hot encoding is that the number of columns easily inflates with a lot of distinct levels. One possible solution is to group some levels based on domain knowledge or to group infrequent levels in "other" level.

You can do one-hot encoding with pandas get_dummies function or scikit learn preprocessing module's OneHotEncoder. The latter one can return sparse matrix with sparse=True parameter, which can save memory consumption when the original column has a lot of levels.

Label encoding

Label encoding is an approach to convert the levels to integers e.g. levels: ['A', 'B', 'C', …] to integers: [0, 1, 2, …].



Original data
A
B
D
C
B
C
:

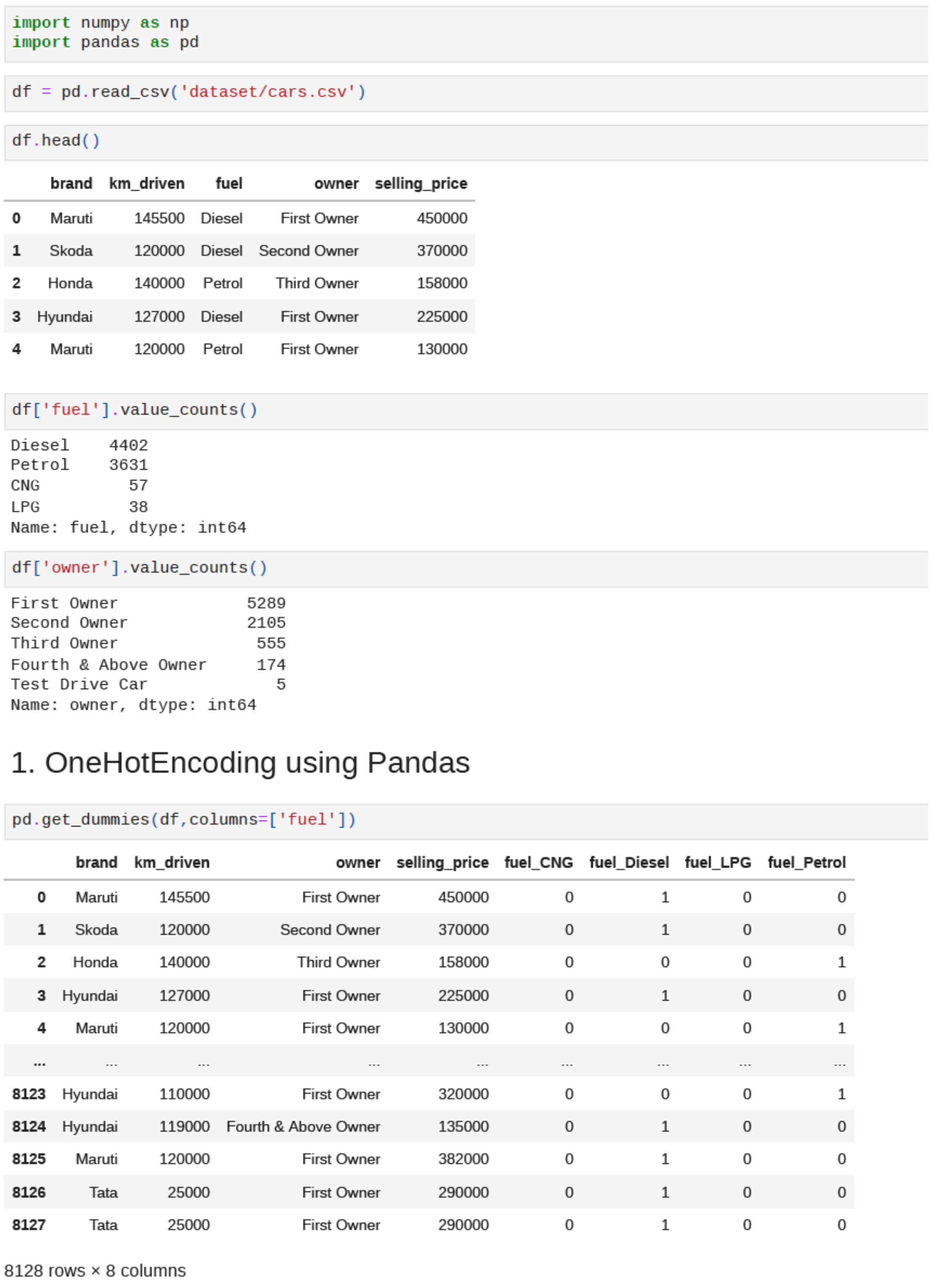
Transformed data by label encoding
0
1
3
2
1
2
:

Illustration of label encoding

This approach is not appropriate in most machine learning algorithms because the amount of transformed value actually has nothing to do with the target variable, **except decision-tree based models** that may be able to split the transformed numeric column multiple times with layering the tree node split. Also, **in case the categorical variable has 'ordinal' nature** e.g.

Cold<Warm<Hot<Very Hot, label encoding can potentially work better than other encoding techniques.

One-Hot-Encoding Code:



```
import numpy as np
import pandas as pd
```

```
df = pd.read_csv('dataset/cars.csv')
```

```
df.head()
```

	brand	km_driven	fuel	owner	selling_price
0	Maruti	145500	Diesel	First Owner	450000
1	Skoda	120000	Diesel	Second Owner	370000
2	Honda	140000	Petrol	Third Owner	158000
3	Hyundai	127000	Diesel	First Owner	225000
4	Maruti	120000	Petrol	First Owner	130000

```
df['fuel'].value_counts()
```

```
Diesel    4402
Petrol    3631
CNG         57
LPG         38
Name: fuel, dtype: int64
```

```
df['owner'].value_counts()
```

```
First Owner             5289
Second Owner            2105
Third Owner              555
Fourth & Above Owner     174
Test Drive Car             5
Name: owner, dtype: int64
```

1. OneHotEncoding using Pandas

```
pd.get_dummies(df,columns=['fuel'])
```

	brand	km_driven	owner	selling_price	fuel_CNG	fuel_Diesel	fuel_LPG	fuel_Petrol
0	Maruti	145500	First Owner	450000	0	1	0	0
1	Skoda	120000	Second Owner	370000	0	1	0	0
2	Honda	140000	Third Owner	158000	0	0	0	1
3	Hyundai	127000	First Owner	225000	0	1	0	0
4	Maruti	120000	First Owner	130000	0	0	0	1
...	...	...	...	...	...	...	...	...
8123	Hyundai	110000	First Owner	320000	0	0	0	1
8124	Hyundai	119000	Fourth & Above Owner	135000	0	1	0	0
8125	Maruti	120000	First Owner	382000	0	1	0	0
8126	Tata	25000	First Owner	290000	0	1	0	0
8127	Tata	25000	First Owner	290000	0	1	0	0

8128 rows × 8 columns

3. OneHotEncoding using Sklearn

```
# from sklearn.model_selection import train_test_split
# X_train,X_test,y_train,y_test = train_test_split(df.iloc[:,0:4],df.iloc[:,-1],test_siz
```

```
df.head()
```

	brand	km_driven	fuel	owner	selling_price
0	Maruti	145500	Diesel	First Owner	450000
1	Skoda	120000	Diesel	Second Owner	370000
2	Honda	140000	Petrol	Third Owner	158000
3	Hyundai	127000	Diesel	First Owner	225000
4	Maruti	120000	Petrol	First Owner	130000

```
from sklearn.preprocessing import OneHotEncoder
```

```
ohe = OneHotEncoder(drop='first', sparse=False)
```

```
df2 = ohe.fit_transform(df[['fuel','owner']])
```

```
df2
```

```
array([[1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 1., 0., 0.],
       [0., 0., 1., ..., 0., 0., 1.],
       ...,
       [1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.]])
```

```
df2.shape
```

```
(8128, 7)
```