Machine Learning -1

Data Preprocessing

Agenda

- Missing Values (Standard Missing Values, Non-Standard Missing Values)
- Handling Non-Numeric Data (One-Hot Encoding, Label Encoding, Ordinal Encoding)
- Normalization and Transformation
- Outliers (Based on Boxplot, IQR, Z-score, Scatter plot)
- Feature Engineering - Introduction
- Train Test Split

Missing Values

Business problem

Problem statement: Goal is to predict the sales of products in BigMart outlets based on various attributes such as item type, price, and outlet size etc. The independent variables (features) include, the product's MRP, the category it belongs to, its type, etc., and the dependent variable (target) is the sales of that product. The objective is to build a model that can effectively predict the sales of a product based on its attributes, which can then be used by the company to make data-driven decisions and improve their sales.

Read the data

We use the ‘bigmartsales’ dataset.

```
import pandas as pd

# read the data
df_sales = pd.read_csv("bigmartsales.csv")

# check first five rows of data
df_sales.head()
```

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establis
0	FDA15	9.30	Low Fat	0.016047	Dairy	249.8	OUT049	
1	DRC01	5.92	Regular	0.019278	Soft Drinks	48.3	OUT018	
2	FDN15	17.50	Low Fat	0.016760	Meat	141.6	OUT049	
3	FDX07	19.20	Regular	0.000000	Fruits and Vegetables	182.1	OUT010	
4	NCD19	8.93	Low Fat	0.000000	Household	53.9	OUT013	

Data information

We shall use the `bigmartsales` dataset. The data description is as follows:

Item_Identifier: Unique product ID assigned to every distinct item

Item_Weight: Weight of the product

Item_Fat_Content: Describes whether the product is low in fat or not

Item_Visibility: Total display area allocated to the particular product (in %)

Item_Type: Describes the food category to which the item belongs

Item_MRP: Maximum Retail Price (list price) of the product

Outlet_Identifier: Unique store ID assigned. It consists of an alphanumeric string of length 6

Data information

Outlet_Establishment_Year: The establishment year of the store

Outlet_Size: Size of the store in terms of ground area covered

Outlet_Location_Type: Size of the city in which the store is located

Outlet_Type: Is the outlet just a grocery store or a supermarket

Profit: Profit of the item sold (in %)

Item_Outlet_Sales: Sales of the product (target variable)

Variable type

Check the data type of each column using the dtypes().

```
df_sales.dtypes
```

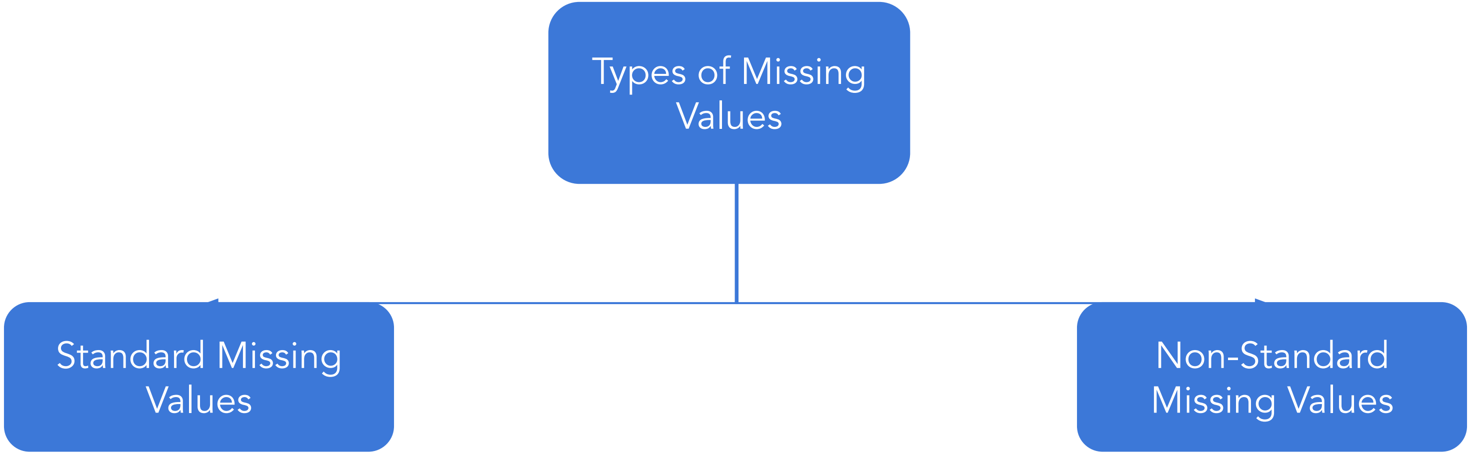
```
Item_Identifier               object
Item_Weight                  float64
Item_Fat_Content              object
Item_Visibility              float64
Item_Type                     object
Item_MRP                     float64
Outlet_Identifier             object
Outlet_Establishment_Year      int64
Outlet_Size                   object
Outlet_Location_Type          object
Outlet_Type                   object
Item_Outlet_Sales            float64
Profit                       float64
dtype: object
```

What are missing values?

- In real data, there are some variables where a particular element is missing because of various reasons, such as corrupt data, failure to load the information

- We can not use the data with missing values for model building

Identify missing values



Standard missing values

Standard missing values are the values that pandas can detect. Let's take a look at the missing values.

```python
# use isnull() to check for missing values
# sum(): gives the sum of missing values in each column
missing_values = df_sales.isnull().sum()

# print the missing values
missing_values
```

```
Item_Identifier                 0
Item_Weight                   749
Item_Fat_Content                0
Item_Visibility                 0
Item_Type                       0
Item_MRP                        0
Outlet_Identifier               0
Outlet_Establishment_Year       0
Outlet_Size                  2410
Outlet_Location_Type         2050
Outlet_Type                     0
Item_Outlet_Sales               0
Profit                          0
dtype: int64
```

Standard missing values

```
# use isnull() to check for missing values
# sum(): gives the sum of missing values in each column
missing_values = df_sales.isnull().sum()

# print the missing values
missing_values
```

```
Item_Identifier                 0
Item_Weight                   749
Item_Fat_Content                0
Item_Visibility                 0
Item_Type                       0
Item_MRP                        0
Outlet_Identifier               0
Outlet_Establishment_Year       0
Outlet_Size                  2410
Outlet_Location_Type         2050
Outlet_Type                     0
Item_Outlet_Sales               0
Profit                          0
dtype: int64
```

We see the variables ‘Item_weight’, ‘Outlet_size’ and ‘outlet_Location_Type’ has missing values detected by pandas.

Non-standard missing values

- Sometimes the missing values have the different formats

```
# check the count of the categories
df_sales.Outlet_Location_Type.value_counts()
```

```
Tier 2     2793
Tier1      2388
Tier 3      932
?           120
  --        109
  -          67
na           48
NAN          16
Name: Outlet_Location_Type, dtype: int64
```

Non-standard missing values

```
# replace "?" with NaN
# to_replace: value that will be replaced
# value: value to replace values matching `to_replace` with
df_sales.Outlet_Location_Type.replace(to_replace = "?", value = np.NaN, inplace = True)

# replace " --" with NaN
# to_replace: value that will be replaced
# value: value to replace values matching `to_replace` with
df_sales.Outlet_Location_Type.replace(to_replace = "  --", value = np.NaN, inplace = True)

# replace " -" with NaN
# to_replace: value that will be replaced
# value: value to replace values matching `to_replace` with
df_sales.Outlet_Location_Type.replace(to_replace = "  -", value = np.NaN, inplace = True)

# replace "na" with NaN
# to_replace: value that will be replaced
# value: value to replace values matching `to_replace` with
df_sales.Outlet_Location_Type.replace(to_replace = "na", value = np.NaN, inplace = True)

# replace "NAN" with NaN
# to_replace: value that will be replaced
# value: value to replace values matching `to_replace` with
df_sales.Outlet_Location_Type.replace(to_replace = "NAN", value = np.NaN, inplace = True)
```

Non-standard missing values

To see the variables with missing values:

```
missing_values=df_sales.isnull().sum()[df_sales.isnull().sum()>0]
missing_values
```

```
Item_Weight              749
Outlet_Size             2410
Outlet_Location_Type    2410
dtype: int64
```

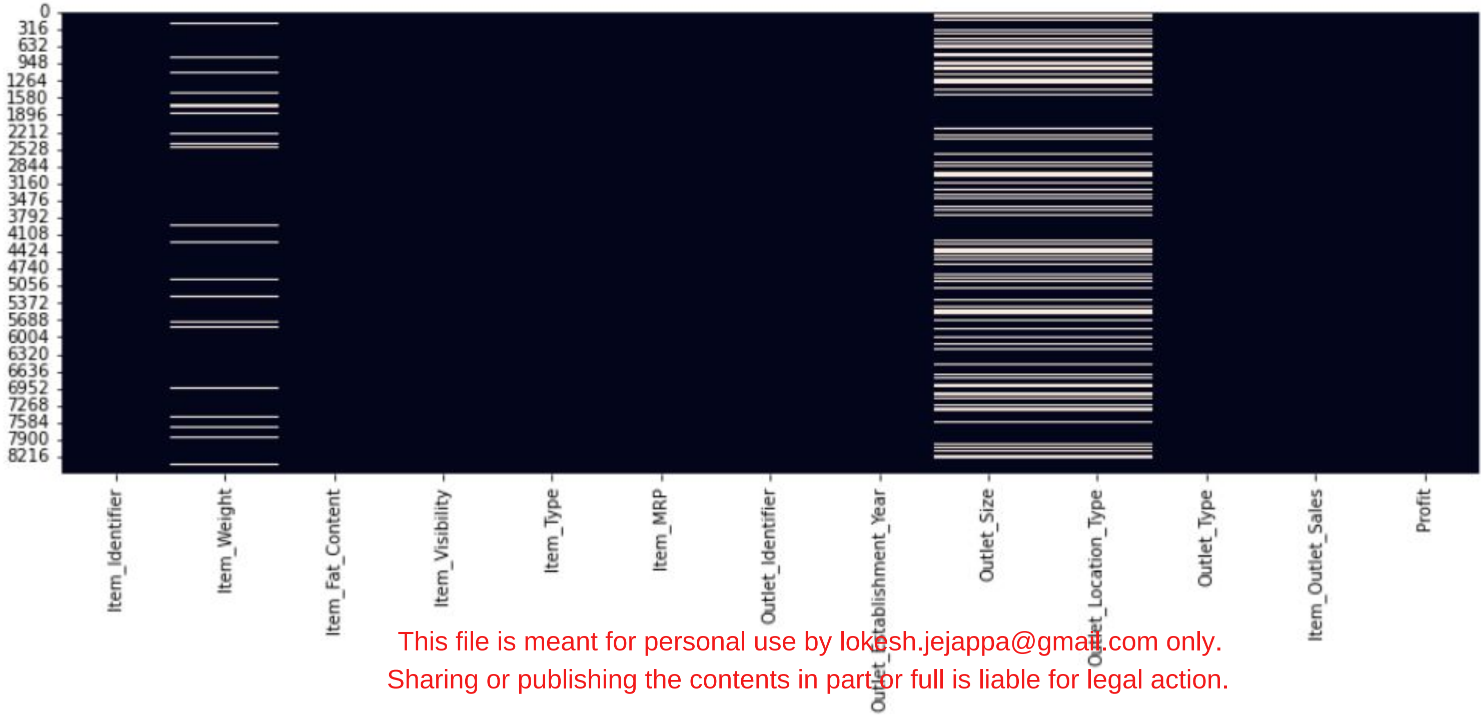
Missing value plot

Let us visualize the missing values using heatmap.

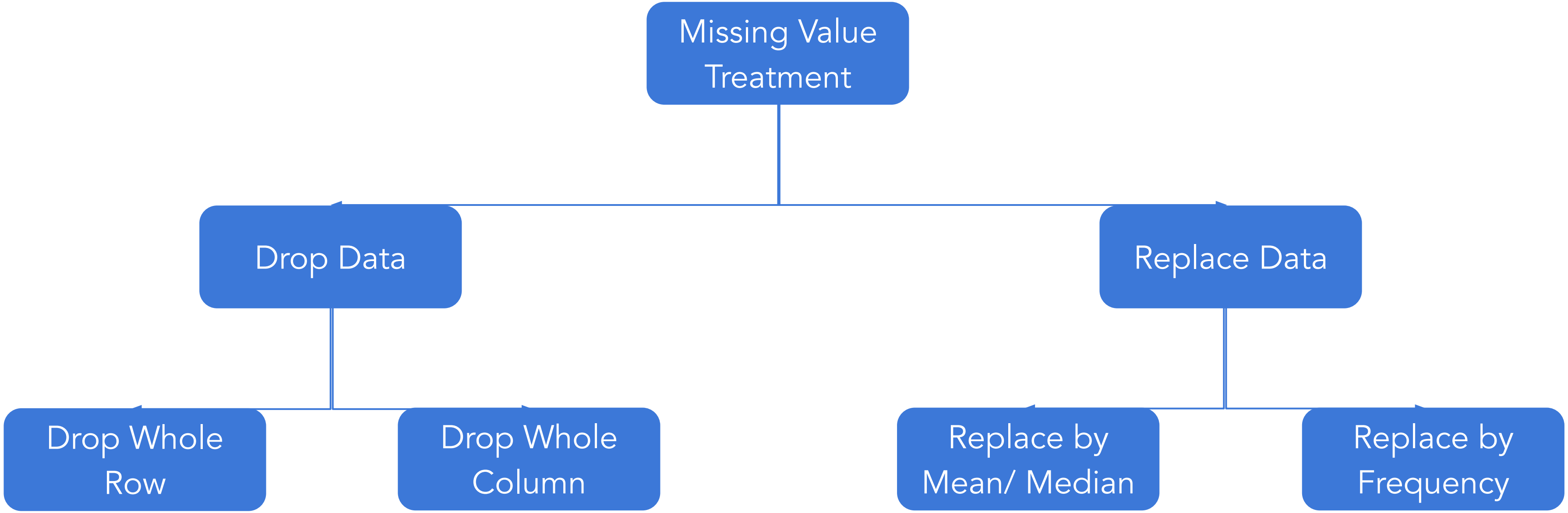
```
# let us plot a heatmap of the missing values

# import the required libraries
# import the library seaborn and matplotlib
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

# set the figure size
plt.rcParams["figure.figsize"]=[15,5]

# plot a heatmap of the missing values in the data
# cbar: specify whether to display the color index or not
sns.heatmap(df_sales.isnull(), cbar = False)

# display the plot
plt.show()
```

Missing value plot

Take a quick look at the variables with the missing values.



Deal with missing values



Drop the rows or columns

- There are cases when a variable has a lot of missing values. In that case, we can drop the variable, if the variable is not a very important predictor for the target variable

- As a rule of thumb, for a variable, if the data has 60-70 percent missing values we should consider the dropping the variable

- Removing the row with the missing values will lead to loss of information

Drop the rows or columns

In our dataset, none of the columns are empty enough to drop entirely. We have some freedom to choose the method to replace the missing values.

Replace by frequency

- For the categorical variable, the missing values can be replaced by the most frequent class of the variable

```
# check the count of the data
df_sales.Outlet_Size.value_counts()
```

```
Medium    2793
Small     2388
High       932
Name: Outlet_Size, dtype: int64
```

- We see that cars having four doors are common

Replace by frequency

Replace missing values by most frequent class

Replace by ‘four’ (most frequent class)

Perform operations on the original data

```
# import the library numpy as np
import numpy as np

# replace all the missing values with 'Medium'
df_sales.Outlet_Size.replace(np.NaN,"Medium" ,inplace = True)
```

Missing values

Replace by mean

- For the numeric variable, missing values can be replaced by the mean
- Median can also be used instead of mean if outliers are present in the features

```python
# fill the missing values with mean
# inplace: makes permanent changes in the dataframe
df_sales["Item_Weight"].fillna(me, inplace = True)
```

Perform operations on the original data

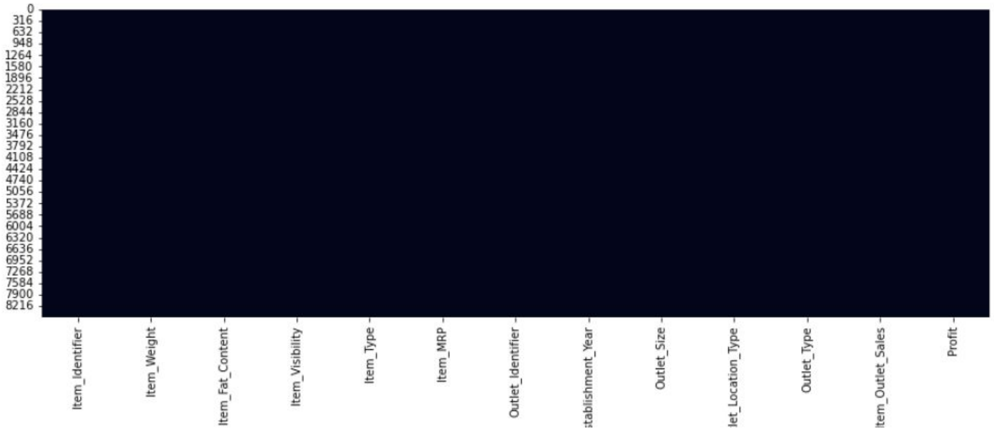
Missing values

Replace by average

Missing value plot

Let us see the heatmap of the missing value once again.

```python
# let us plot a heatmap of the missing values

# import the required libraries
# import the library seaborn and matplotlib
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

# set the figure size
plt.rcParams["figure.figsize"]=[15,5]

# plot a heatmap of the missing values in the data
# cbar: specify whether to display the color index or not
sns.heatmap(df_sales.isnull(), cbar = False)

# display the plot
plt.show()
```



Now, there are no missing values in the data.

Handling Non-numeric Data

Handling non-numeric data

- The dataset may contain numerical and/or categorical variables. Most of the algorithms are designed to work on numeric data

- There are several methods (label encoding, dummy encoding) to convert the categorical data into numerical data

Type of encoding

There are many ways to encode the categorical variables:

- N-1 Dummy encoding
- One-hot encoding
- Label encoding
- Ordinal encoding
- Frequency encoding
- Target encoding

Type of encoding

There are many ways to encode the categorical variables:

- N-1 Dummy encoding
- One-hot encoding
- Label encoding
- Ordinal encoding
- Frequency encoding
- Target encoding

(N-1) dummy encoding (using pandas)

- It is used to create dummy variables from a categorical variable
- For a categorical variable that can take k values, k-1 dummy variables are created

Product
Vegetables
Dairy
Fruits
Vegetables

→

Dairy_Product	Fruits_Product
0	0
1	0
0	1
0	0

(N-1) dummy encoding (using pandas)

Perform N-1 dummy encoding on variable origin.

```
# create dummy variables for 'Item_Type'
# 'drop_first = True' creates (n-1 = 15) dummy variables from (n = 16) categories
pd.get_dummies(df_sales, columns= ['Item_Type'], drop_first = True).head()
```

lousehold	Item_Type_Meat	Item_Type_Others	Item_Type_Seafood	Item_Type_Snack Foods	Item_Type_Soft Drinks	Item_Type_Starchy Foods
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	1	0	0	0	0	0
0	0	0	0	0	0	0
1	0	0	0	0	0	0

value '1' in column 'Item_Type_soft Drinks' denotes the Item_Type is of soft Drinks

Type of encoding

There are many ways to encode the categorical variables:

- N-1 Dummy encoding
- One-hot encoding
- Label encoding
- Ordinal encoding
- Frequency encoding
- Target encoding

One hot encoding

- It is used to create dummy variables from a categorical variable
- For a categorical variable that can take k values, k dummy variables are created
- Each category is converted into one column with values ‘0’ and ‘1’, depending on the presence or absence of the category in the corresponding observation

Product
Vegetables
Dairy
Fruits
Vegetables

→

Vegetables_Product	Dairy_Product	Fruits_Product
1	0	0
0	1	0
0	0	1
1	0	0

One hot encoding (using pandas)

There are 16 unique categories in the variable 'Item_Type'.

```
# check the categories of the variable 'Item_Type'
print('Categories in Item_Type:', df_sales.Item_Type.unique())
```

```
Categories in Item_Type: ['Dairy' 'Soft Drinks' 'Meat' 'Fruits and Vegetables' 'Household'
 'Baking Goods' 'Snack Foods' 'Frozen Foods' 'Breakfast'
 'Health and Hygiene' 'Hard Drinks' 'Canned' 'Breads' 'Starchy Foods'
 'Others' 'Seafood']
```

Let us use one-hot encoding to create 16 variables corresponding to each level in the variable, Item_Type.

One hot encoding (using pandas)

Perform one-hot encoding on variable Item_Type.

```
# create dummy variables for 'Item_Type'
# It creates 16 dummy variables from 16 categories
pd.get_dummies(df_sales, columns= ['Item_Type']).head()
```

Item_Type_Health and Hygiene	Item_Type_Household	Item_Type_Meat	Item_Type_Others	Item_Type_Seafood	Item_Type_Snack Foods
0	0	0	0	0	0
0	0	0	0	0	0
0	0	1	0	0	0
0	0	0	0	0	0
0	1	0	0	0	0

One hot encoding (using sklearn)

The sklearn library provides a function to convert the categorical variable into one-hot encoded variables.

```python
# import the OneHotEncoder
from sklearn.preprocessing import OneHotEncoder

# creating instance of one-hot-encoder
encode = OneHotEncoder()

# fit_transform: It returns a sparse array
# .toarray(): It returns a numpy array
df_encode = pd.DataFrame(encode.fit_transform(df_sales[['Outlet_Type']]).toarray(),
                         columns = ['Grocery Store', 'Supermarket Type1',
                                    'Supermarket Type2', 'Supermarket Type3'])

# merge with main dataframe (df_sales)
df_encode = pd.concat([df_sales, df_encode], axis=1)

# print 20 rows of the data
df_encode.head()
```

let_Size	Outlet_Location_Type	Outlet_Type	Item_Outlet_Sales	Profit	Grocery Store	Supermarket Type1	Supermarket Type2	Supermarket Type3
Medium	Tier 2	Supermarket Type1	3735.1380	11.5	0.0	1.0	0.0	0.0

Type of encoding

There are many ways to encode the categorical variables:

- N-1 Dummy encoding
- One-hot encoding
- $\boxed{\text{Label encoding}}$
- Ordinal encoding
- Frequency encoding
- Target encoding

Label encoding

- The LabelEncoder considers the levels in a categorical variable by alphabetical order for encoding

- LabelEncoder labels different levels of the categorical variable with values between 0 and n-1, where 'n' is number of distinct categories

Label encoding (using sklearn)

```
# import the LabelEncoder
from sklearn.preprocessing import LabelEncoder

# instantiate the encoder
labelencoder = LabelEncoder()

# fit the encoder on 'Outlet_Size'
df_sales['Label_Encoded_Outlet_Size'] = labelencoder.fit_transform(df_sales.Outlet_Size)

# display first 5 observations
df_sales.head()
```

stablishment_Year	Outlet_Size	Outlet_Location_Type	Outlet_Type	Item_Outlet_Sales	Profit	Label_Encoded_Outlet_Size
1999	Medium	Tier 2	Supermarket Type1	3735.1380	11.5	1
2009	Medium	Tier 2	Supermarket Type2	443.4228	14.3	1

Type of encoding

There are many ways to encode the categorical variables:

- N-1 Dummy encoding
- One-hot encoding
- Label encoding
- Ordinal encoding
- Frequency encoding
- Target encoding

Ordinal encoding

The ordinal encoder from sklearn encodes the categorical variable with values between 0 and (n-1). We can pass the order of the categories to preserve the order present in the ordinal categorical variable.

Ordinal encoding (using sklearn)

```
# import the OrdinalEncoder
from sklearn.preprocessing import OrdinalEncoder

# instantiate the encoder with the list of categories in the required order
orderencoding = OrdinalEncoder(categories = [["Small", "Medium", "High", "nan"]])

# add a column of ordered labels
# reshape(-1, 1): to rearrange the data
df_sales['Order_Outlet_Size'] = orderencoding.fit_transform(df_sales['Outlet_Size'].values.
                                                              reshape(-1,1))

# display the data
df_sales.head()
```

Outlet_Size	Outlet_Location_Type	Outlet_Type	Item_Outlet_Sales	Profit	Label_Encoded_Outlet_Size	Order_Outlet_Size
Medium	Tier 2	Supermarket Type1	3735.1380	11.5	1	1.0
Medium	Tier 2	Supermarket Type2	443.4228	14.3	1	1.0

Type of encoding

There are many ways to encode the categorical variables:

- N-1 Dummy encoding
- One-hot encoding
- Label encoding
- Ordinal encoding
- Frequency encoding
- Target encoding

Frequency encoding

- If a categorical variable contains too many levels, then using one-hot encoding will increase the number of features drastically
- Frequency encoding replaces each label of the categorical variable by the percentage of observations within that category

Example of frequency encoding

Skill	Encoded Feature
Python	0.44
Python	0.44
Python	0.44
Python	0.44
R	0.33
R	0.33
R	0.33
SQL	0.22
SQL	0.22

Python	0.44 (4 out of 9)
R	0.33 (3 out of 9)
SQL	0.22 (2 out of 9)

Frequency encoding

```
# frequency encoding on 'Item_Type'
# size of each category
encoding = df_sales.groupby('Item_Type').size()

# get frequency of each category
encoding = encoding/len(df_sales)

# encode the column
# map(): apply encoding to each item in the variable
# and multiply by 100
df_sales['Freq_Encoded_Item_Type'] = df_sales.Item_Type.map(encoding)*100

# print first five rows of the data
df_sales.head()
```

on_Type	Outlet_Type	Item_Outlet_Sales	Profit	Label_Encoded_Outlet_Size	Order_Outlet_Size	Freq_Encoded_Item_Type
Tier 2	Supermarket Type1	3735.1380	11.5	1	1.0	8.001877
Tier 2	Supermarket Type2	443.4228	14.3	1	1.0	5.221166

Frequency encoding

- The method fails when two or more categories have the same number of observations. In such a scenario, different labels will have the same frequency
- Likewise, if all the categories have same frequency in the data, as the encoded column will contain a single value. (i.e. we get a column of a zero-variance)

Skill	Encoded Feature
C#	0.4
R	0.3
R	0.3
R	0.3
C++	0.3
C++	0.3
C++	0.3
C#	0.4
C#	0.4
C#	0.4

R	0.3 (3 out of 10)
C++	0.3 (3 out of 10)
C#	0.4 (4 out of 10)

Type of encoding

There are many ways to encode the categorical variables:

- N-1 Dummy encoding
- One-hot encoding
- Label encoding
- Ordinal encoding
- Frequency encoding
- Target encoding

Target encoding

- Target encoding is a technique used in a classification problem to convert a categorical variable to a numeric variable

- Encodes each level of categorical variable with its corresponding target mean

- The dimensionality of the data remains the same as that of not encoded data

- It is also known as mean encoding

Target encoding

- Consider the given data where we have a categorical variable ‘smoker’ explaining the smoking habits of each individual

- Target is a binary variable with levels (0 = No heart attack, 1 = Heart attack)

Smoker	Target
yes	1
yes	0
no	1
no	0
yes	0
no	0
yes	1
yes	0
no	0
yes	1

Target encoding

The target encoded values are computed as follows:

Smoker	Target		Total values	Mean of target variable
	1	0		
yes	3	3	6	(1+0+0+1+0+1)/6 = 3/6 = 0.5
no	1	3	4	(1+0+0+0)/4 =1/4 = 0.25

Smoker	Target
yes	1
yes	0
no	1
no	0
yes	0
no	0
yes	1
yes	0
no	0
yes	1

Target encoding

The target encoding:

Smoker	Encode
yes	0.5
no	0.25

Smoker	Target		Total values	Mean of target variable
	1	0		
yes	3	3	6	(1+0+0+1+0+1)/6 = 3/6 = 0.5
no	1	3	4	(1+0+0+0) =1/4 = 0.25

Target encoding

Create a dataframe of the given data

```
# create the dataframe
df_smoker = pd.DataFrame({
    'Smoker': ['yes', 'yes', 'no', 'no', 'yes', 'no','yes','yes', 'no', 'yes'],
    'Target': [1, 0, 1, 0, 0, 0, 1, 0, 0, 1]
    })

# print the dataframe
print(df_smoker)
```

```
  Smoker  Target
0    yes       1
1    yes       0
2     no       1
3     no       0
4    yes       0
5     no       0
6    yes       1
7    yes       0
8     no       0
9    yes       1
```

Target encoding

Calculate the average of the target for each category in the feature.

```
# group the target variable by smoking habit using groupby()
# mean(): gives the mean
# we obtain the target mean for each level of categorical variable
df_smoker.groupby('Smoker')['Target'].mean()
```

```
Smoker
no       0.25
yes      0.50
Name: Target, dtype: float64
```

These values will replace the 'yes' and 'no' values in the variable

Target encoding

```
# create a new variable "Smoker_encoded" which has the encoded values
# map(): maps an output of a function to a pandas dataframe column
df_smoker["Smoker_encoded"] = df_smoker['Smoker'].map(mean)

# print the dataframe
df_smoker
```

	Smoker	Target	Smoker_encoded
0	yes	1	0.50
1	yes	0	0.50
2	no	1	0.25
3	no	0	0.25
4	yes	0	0.50
5	no	0	0.25
6	yes	1	0.50
7	yes	0	0.50
8	no	0	0.25
9	yes	1	0.50

Smoker = yes is encoded as 0.5

Smoker = no is encoded as 0.25

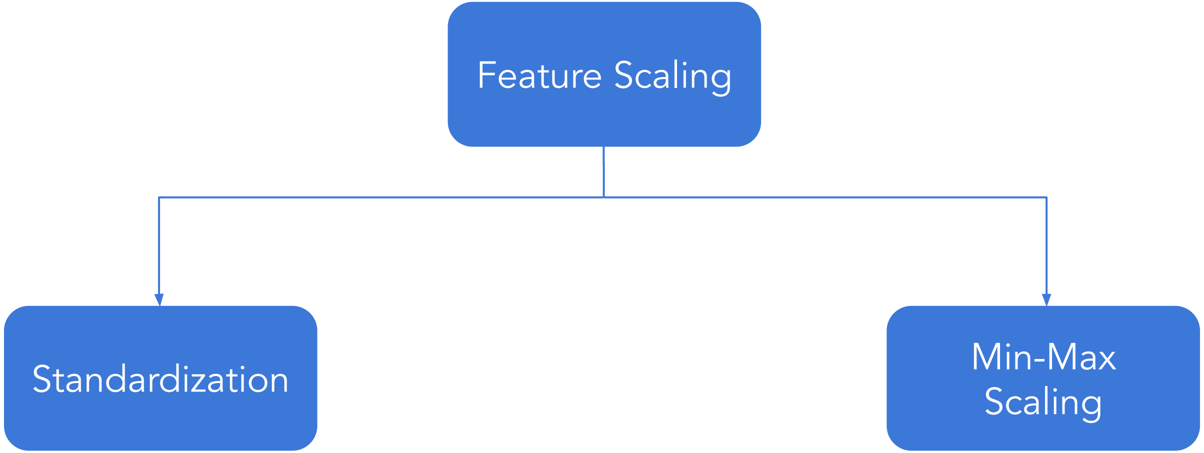
Feature Scaling

Feature scaling

- Feature scaling is also known as data normalization

- It is a technique used to transform the data into a common scale. Since the features have various ranges, it becomes a necessary step in data preprocessing while using machine learning algorithms

Methods to perform feature scaling



Standardization or Z-score normalization

- Standardization transforms the data such that the data has mean 0 and unit variance

- The procedure involves subtracting the mean from observation and then dividing by the standard deviation

$$\boxed{x_{new} = \frac{x - \mu}{\sigma}}$$

Standardization or Z-score normalization

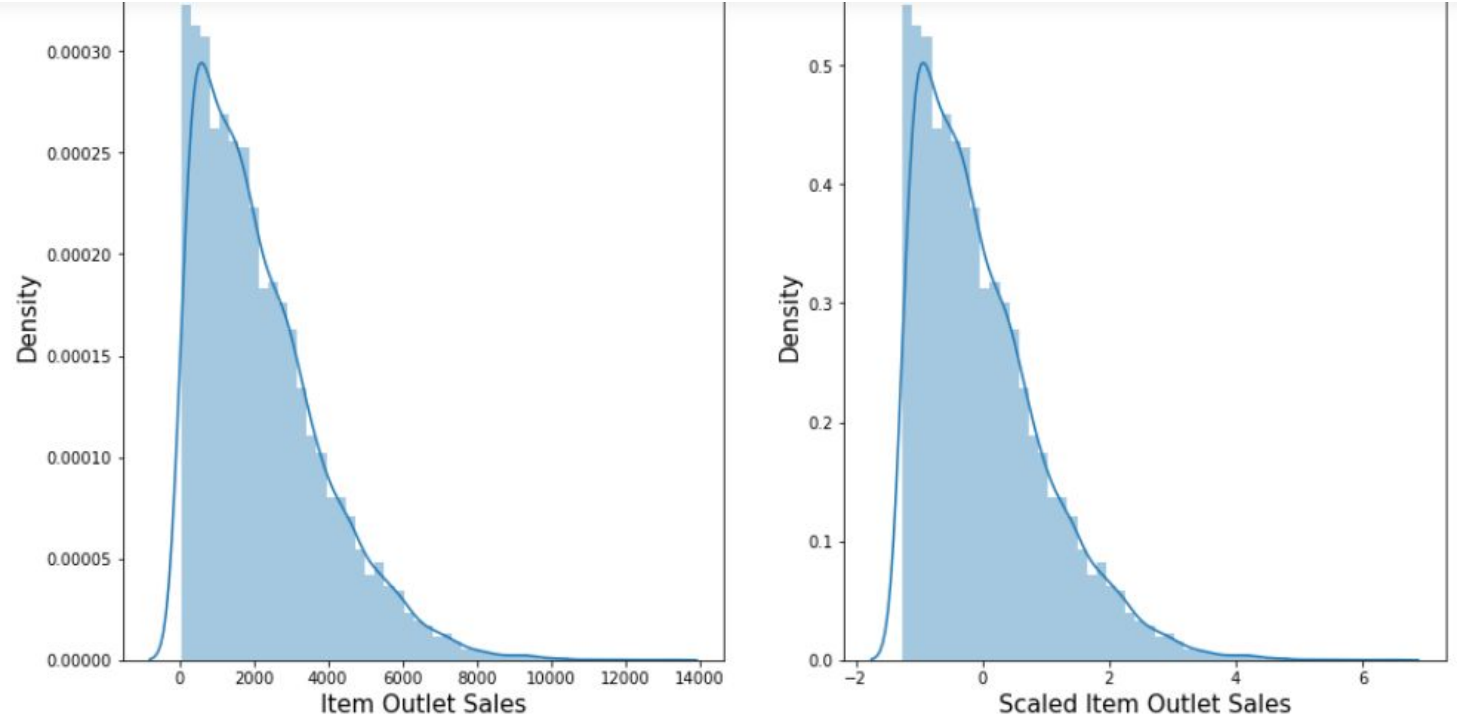
```
# calculate the minimum and maximum values of the variable
print(" The minimum value of the sales:",df_sales['Item_Outlet_Sales'].min(),"\n",
      "The maximum value of the sales:", df_sales['Item_Outlet_Sales'].max())
```

```
 The minimum value of the sales: 33.29
 The maximum value of the sales: 13086.9648
```

```
# import StandardScaler
from sklearn.preprocessing import StandardScaler

# instantiate the standardscaler
standard_scale = StandardScaler()

# fit the StandardScaler
df_sales['Scaled_Item_Outlet_Sales'] = standard_scale.fit_transform(df_sales[['Item_Outlet_Sales']])

# calculate the minimum and maximum values of the variable
print(" The minimum value of the sales:",df_sales['Scaled_Item_Outlet_Sales'].min(),"\n",
      "The maximum value of the sales:", df_sales['Scaled_Item_Outlet_Sales'].max())
```

```
 The minimum value of the sales: -1.2587901671720854
 The maximum value of the sales: 6.391044932769205
```

Standardization or Z-score normalization



Min-max normalization

- Performs linear transformation on the original data
- The min-max normalization is given as:

$$\boxed{X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}}$$

- After normalization, all values will be between 0 and 1

PLEASE!! NOTE

A Min-Max normalization preserves linear relationships between variables. Therefore, the correlation between the two variables will not change after a linear transformation.

Min-max normalization

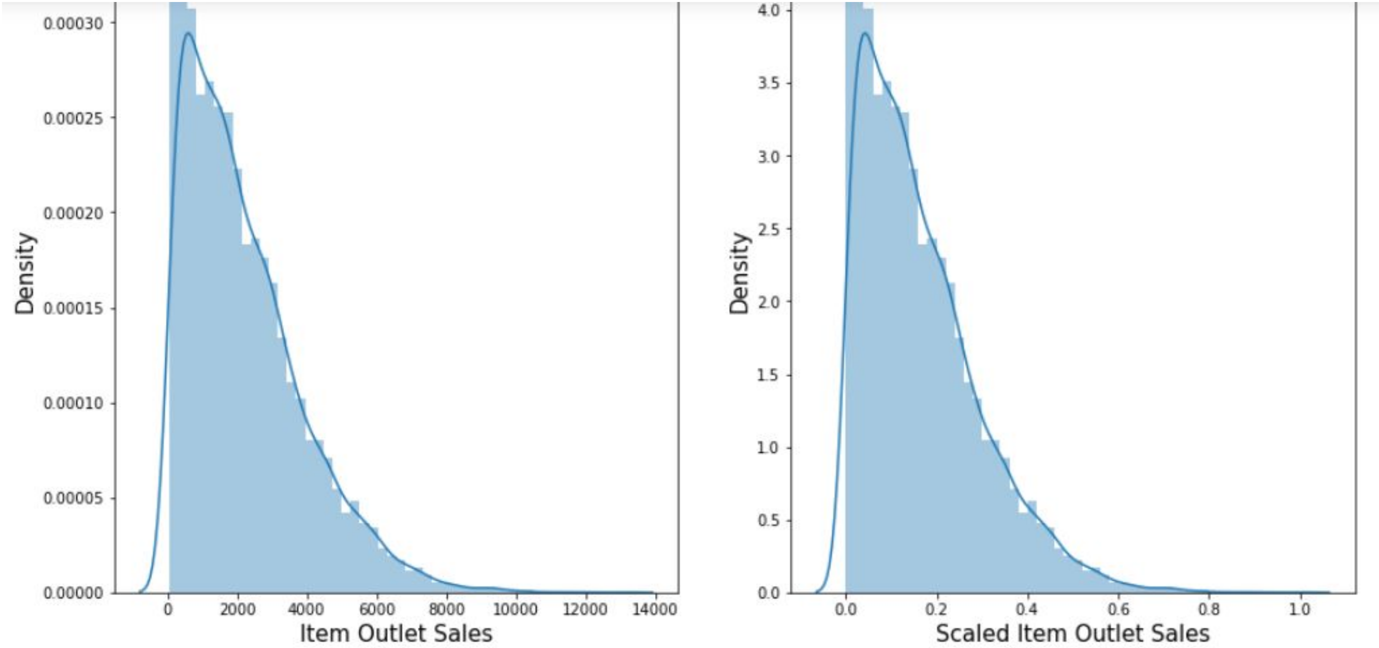
Perform min-max normalization on *Item_outlet_sales*

```
# import MinMaxScaler
from sklearn.preprocessing import MinMaxScaler

# instantiate the MinMaxScaler
min_max = MinMaxScaler()

# fit the MinMaxScaler
df_sales['minmax_Item_Outlet_Sales'] = min_max.fit_transform(df_sales[['Item_Outlet_Sales']])

# minimum and maximum value of the normalized variable
df_sales['minmax_Item_Outlet_Sales'].min(), df_sales['minmax_Item_Outlet_Sales'].max()
```

```
(0.0, 1.0)
```

Min-max normalization

Perform min-max normalization on *Item_outlet_sales*



Data Transformation

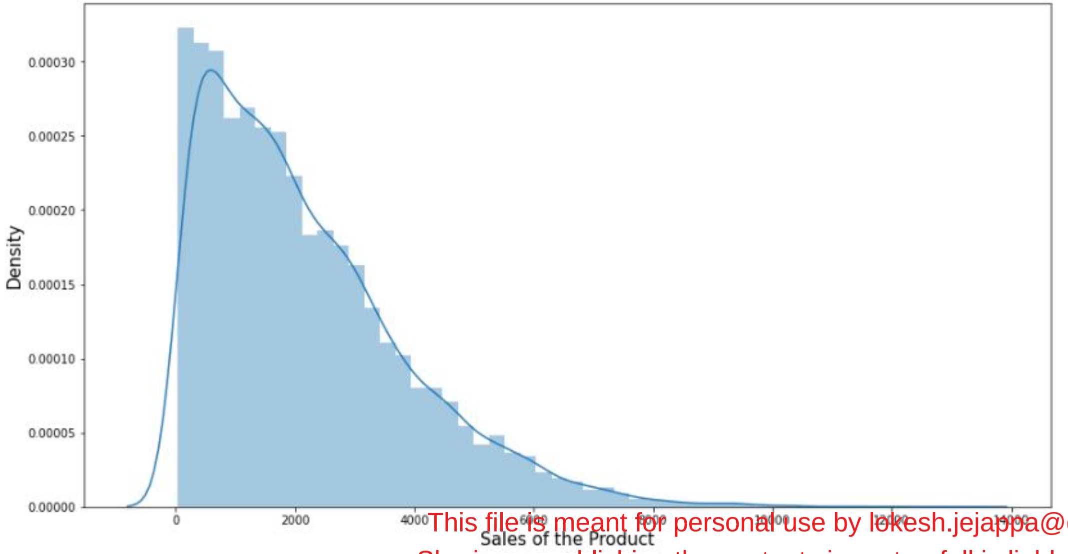
Log transformation

- Reduces the skewness in the distribution of the original data
- Makes the data more interpretable
- The arithmetic mean of the log-transformed data is the geometric mean of the original data
- It converts the exponential growth to a linear growth

Log transformation

- Check the distribution of the variable ‘Sales of the Product’
- The variable ‘Item_Outlet_Sales’ is positively skewed. `Skewness: 1.177530`

Skewness: 1.1775306028542796

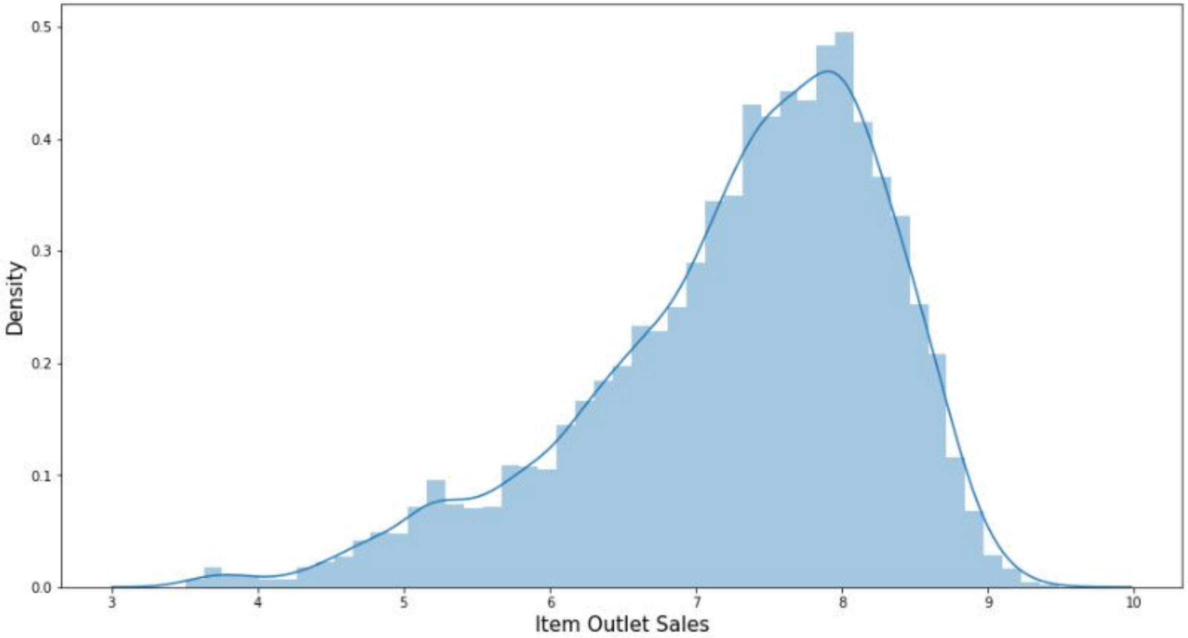


Log transformation

Apply log transformation using log() from numpy.

Note the reduction in skewness.

`Skewness:0.88775334320`

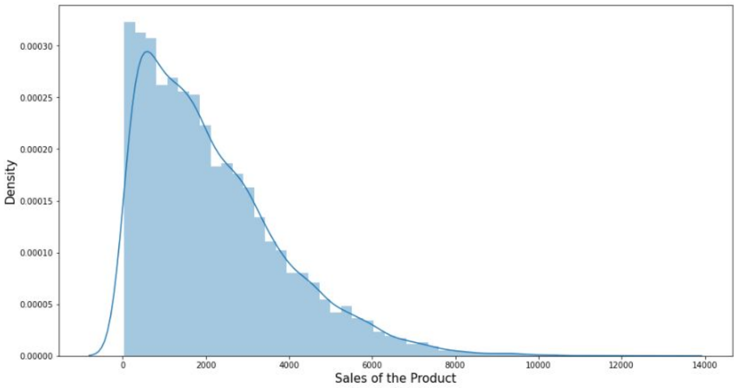


Log transformation

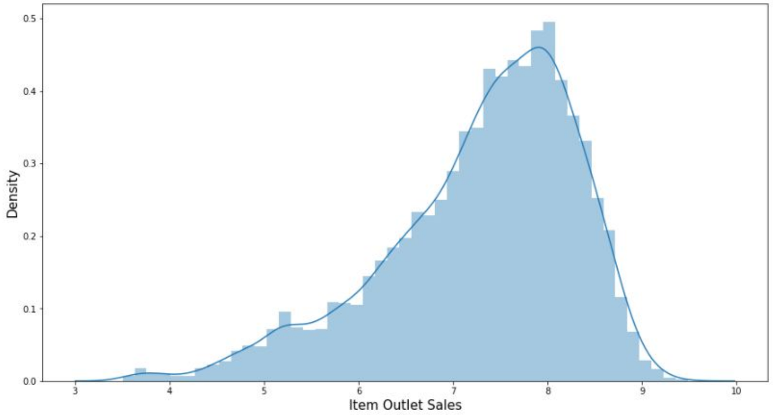
Before log transformation

After log transformation

Skewness: 1.1775306028542796



The skewness of the variable is reduced

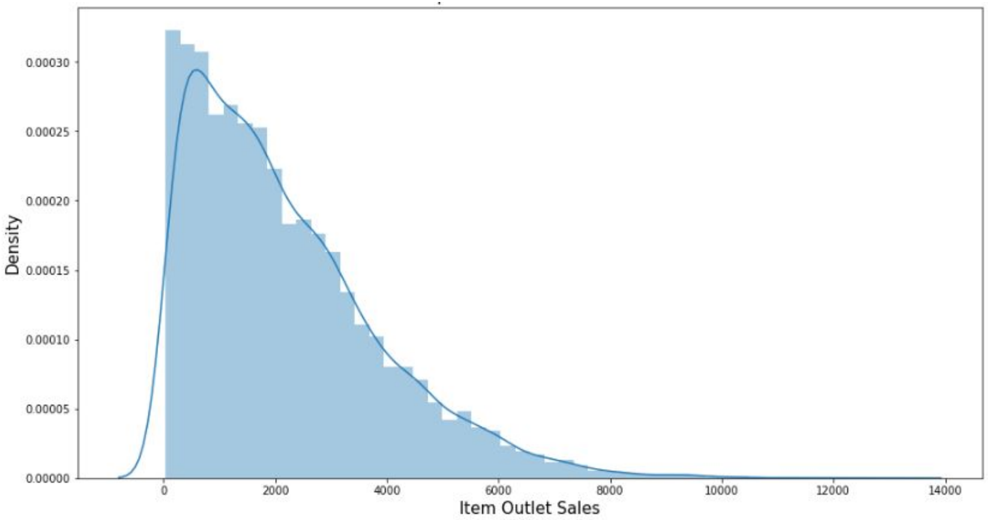


Exponential transformation

- It is the inverse transformation of the log transformation
- It is used to convert the log-transformed values to their original units

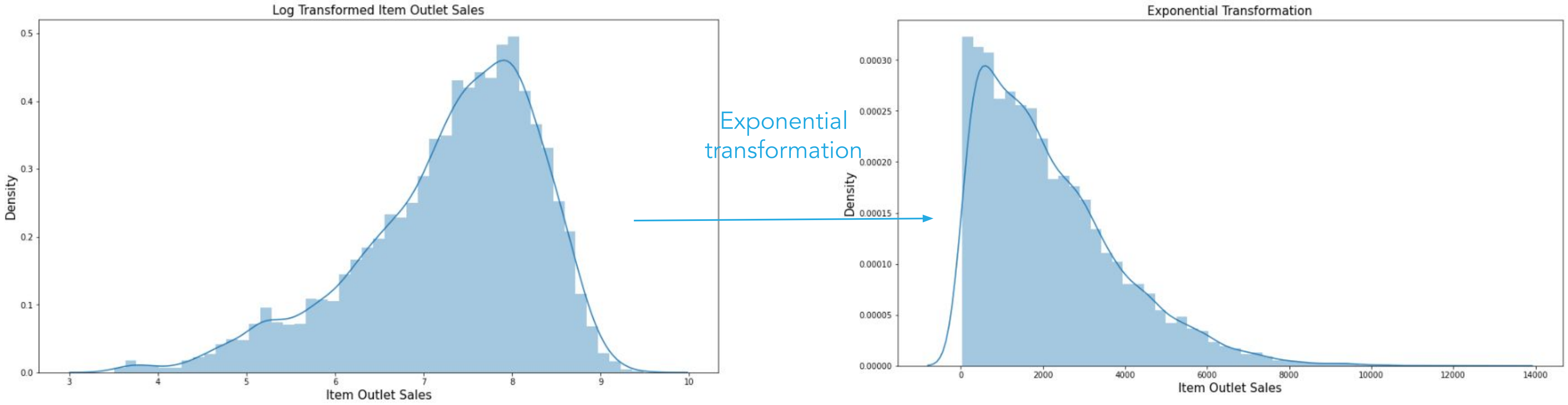
Exponential transformation

Interpretation: Applying exponentia
transformation on the log-transform
values of the variable 'Item outlet sa
we get the original 'Item outlet sales



Exponential transformation

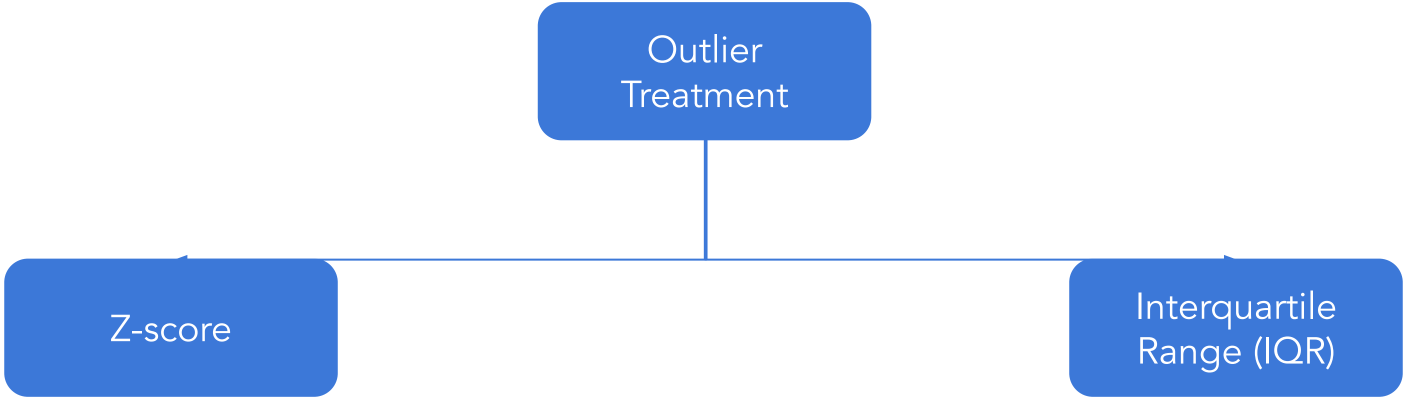
Log-transformed values of displacement are in their original scale after applying the exponential transformation.



Outlier treatment

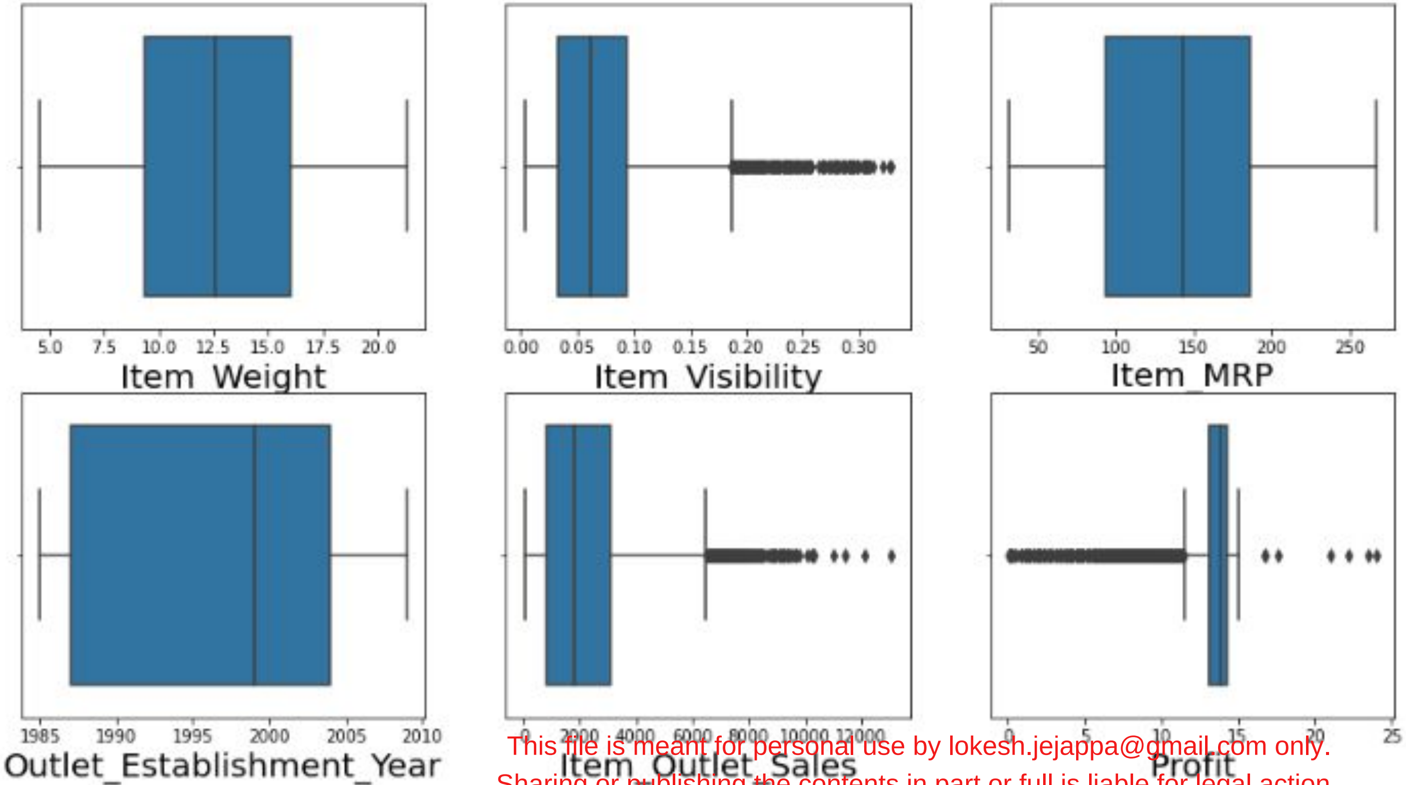
Discover outliers

- In the last session, we learnt the outlier detection using boxplot and z-score
- Now, we will see how to deal with the outliers



Discover outliers using BoxPlot

- Consider the variables and create a boxplot



Z-score

- Z-scores can quantify the unusualness of observation when data follow the normal distribution
- Mathematical formula for z-score for the variable X is given as:

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

Where,

μ: Mean of the variable

σ: Standard deviation

Z-score

- Z-scores are the number of standard deviations above or below the mean
- For example, a z-score of 3 indicates that observation is three standard deviations above the average while a z-score of -3 signifies it is three standard deviations below the mean
- A z-score of zero represents a value that equals the mean

Z-score

Use the zscore() from scipy library to detect the outliers in the variable ‘profit’.

```
# import library scipy
import scipy
# from scipy import the module stats
from scipy import stats

# z-scores are defined for each observation in a variable
# compute the z-scores using the method zscore from the scipy library
z_scores_Profit = scipy.stats.zscore(df_num["Profit"])

# display the z-scores
z_scores_Profit
```

```
0       -1.125033
1        0.520342
2        0.637869
3        0.108998
4        0.402815
           ...
8518     0.402815
8519     0.461578
8520    -2.300301
8521     0.461578
8522     0.696632
```

Z-score

- A standard cut-off value for finding outliers are z-scores of +/-3
- Let us identify the outliers

```python
# print the rows where z-score is less than -3
row_index_less = np.where(z_scores_Profit < -3)

# print the values
print(row_index_less)
```

```
(array([  41,   50,  144,  217,  320,  324,  406,  425,  432,  457,  546,
         602,  607,  611,  641,  716,  761,  816,  892,  921,  926,  957,
         960,  986, 1066, 1133, 1178, 1212, 1237, 1304, 1409, 1459, 1533,
        1551, 1561, 1582, 1623, 1645, 1649, 1659, 1669, 1715, 1784, 1788,
        1805, 1816, 1835, 1867, 1893, 1900, 1916, 1956, 1974, 1984, 1989,
        2000, 2019, 2060, 2062, 2130, 2201, 2224, 2246, 2289, 2332, 2338,
        2425, 2449, 2477, 2510, 2515, 2526, 2537, 2543, 2572, 2636, 2792,
        2836, 2919, 2925, 2969, 2987, 2992, 2996, 3025, 3035, 3098, 3105,
        3223, 3254, 3296, 3356, 3415, 3416, 3476, 3513, 3619, 3668, 3675,
        3759, 3785, 3849, 3852, 3904, 3917, 3932, 3936, 3990, 4073, 4091,
        4146, 4197, 4213, 4407, 4487, 4493, 4556, 4571, 4705, 4721, 4724,
        4764, 4800, 5049, 5127, 5231, 5274, 5286, 5313, 5336, 5378, 5516,
        5540, 5657, 5658, 5728, 5730, 5750, 5788, 5883, 5891, 5906, 5971,
        5987, 6093, 6137, 6211, 6247, 6271, 6372, 6390, 6419, 6507, 6534,
        6541, 6558, 6606, 6610, 6696, 6759, 6760, 6776, 6778, 6897, 6931,
        6972, 7042, 7180, 7242, 7267, 7304, 7351, 7384, 7432, 7629, 7684,
        7751, 7779, 7797, 7822, 7851, 7893, 7919, 7948, 8074, 8099, 8101,
        8237, 8273, 8366, 8376, 8395, 8429, 8433, 8456], dtype=int64),)
```

```python
# print the rows where z-score is more than 3
row_index_more = np.where(z_scores_Profit > 3)

# print the values more than the
print(row_index_more)
```

```
(array([3026, 4386, 5089, 8369], dtype=int64),)
```

- The array contains list of row index, which has a z-score greater than 3

Z-score

- Now, remove the outliers and get the clean data

```
# filter out the outlier values
# ~ : selects all rows which do not satisfy the condition
df_sales_zscore = df_sales["Profit"][~(( z_scores_Profit < -3) |(z_scores_Profit > 3))]
```

```
# check the shape
df_sales_zscore.shape
```

```
(8324,)
```

- However, if our data does not follow the normal distribution, this approach might not be right

Z-score

- In case the variable has Gaussian or Gaussian-like distribution, considering 3 standard deviations from the mean is a standard practice for identifying outliers

- However, for smaller samples of data, you may consider 2 standard deviations (95%) from the mean, and for larger samples, you may consider 4 standard deviations (99.9%) from the mean to detect outliers

Interquartile range

- The interquartile range is the middle 50% of the dataset
- It ranges between the third and the first quartile
- We use the interquartile range, first quartile, and third quartile to identify the outliers

Interquartile range

Let us calculate the IQR by calculating Q1 and Q3.

```
# obtain the first quartile
Q1 = df_num.quantile(0.25)

# obtain the third quartile
Q3 = df_num.quantile(0.75)

# obtain the IQR
IQR = Q3 - Q1

# print the IQR
print(IQR)
```

```
Item_Weight                      6.690000
Item_Visibility                  0.061500
Item_MRP                        91.850000
Outlet_Establishment_Year       17.000000
Item_Outlet_Sales             2267.049000
Profit                           1.150000
Label_Encoded_Outlet_Size        1.000000
Order_Outlet_Size                1.000000
Freq_Encoded_Item_Type           7.978411
Scaled_Item_Outlet_Sales         1.328557
minmax_Item_Outlet_Sales         0.173671
dtype: float64
```

Interquartile range

The outlier is a point which falls below Q1 – 1.5×IQR or above Q3 + 1.5×IQR.

```
# filter out the outlier values
# ~ : selects all rows which do not satisfy the condition
# any() : returns whether any element is True over the columns
# axis : "1" indicates columns should be altered (0 for 'index')
df_sales_iqr = df_sales[~((df_sales < (Q1 - 1.5 * IQR)) |(df_sales > (Q3 + 1.5 * IQR))).any(axis=1)]
```

Interquartile range

Check the shape of the data.

```
# check the shape of the data
df_sales_iqr.shape
```

```
(7487, 18)
```

Introduction to Feature Engineering

Feature engineering

Let us perform feature engineering on the variable 'Outlet_Establishmeny_Year'

```
# import the required libraries
import datetime
from datetime import date

# get the current year
# today(): gives today's date
# year: gives the current year
current_year = date.today().year

df_sales["Age_Outlet"] = current_year - df_sales["Outlet_Establishment_Year"]
```

```
# display head of the data
df_sales.head(3)
```

ales	Profit	Label_Encoded_Outlet_Size	Order_Outlet_Size	Freq_Encoded_Item_Type	Scaled_Item_Outlet_Sales	minmax_Item_Outlet_Sales	Age_Outlet
380	11.5	1	1.0	8.001877	0.910601	0.283587	23
228	14.3	1	1.0	5.221166	-1.018440	0.031419	13
700	14.5	1	1.0	4.986507	-0.049238	0.158115	23

Feature engineering

- It is the process of using domain knowledge of the data to create new features that make the machine learning model perform better

- Feature engineering is the essential art in machine learning, which creates a massive difference between a good model and a bad model

Feature engineering

```
# extract the first two letter of 'Item_Identifier'
# apply: applies a function along an axis of the DataFrame
# lambda: creates a small, one-time, anonymous function to extract the first two letters
# indexing in python begins with 0
df_sales['Product_type'] = df_sales['Item_Identifier'].apply(lambda x: x[0:2])

# rename the categories
# map: maps the category names to the identifier codes
df_sales['Product_type'] = df_sales['Product_type'].map({'FD':'Food','NC':'Non-Consumable', 'DR':'Drinks'})

# display the counts
df_sales['Product_type'].value_counts()
```

```
Food              6125
Non-Consumable    1599
Drinks             799
Name: Product_type, dtype: int64
```

Train-Test Split

Train-test split

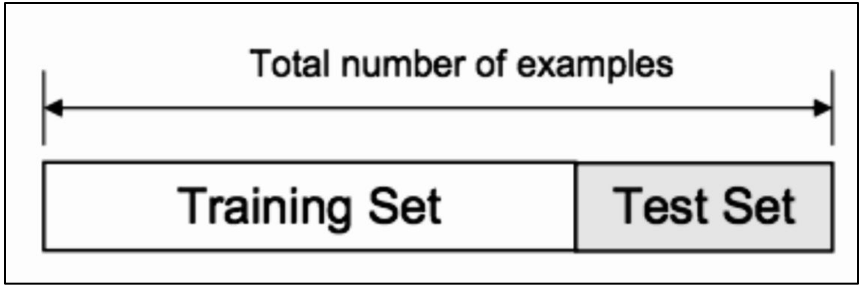
- The most straightforward technique that is used to evaluate the performance of a machine learning algorithm is to use different subsets of a dataset

- We can split our original dataset into two parts (train and test)

- Build the model on the training dataset, make predictions on the test dataset and evaluate the forecasts against the expected results

Train-test split

The size of the split can depend on the size of the dataset, although it is common to use 70% of the data for training and the remaining 30% for testing.



Train-test split

```
#import the sklearn library
import sklearn

# import the train_test_split module from sklearn
from sklearn.model_selection import train_test_split
```

```
# select the target variable
Y = df_sales['Item_Outlet_Sales']

# select all the independent variables
# by dropping the target variable
X = df_sales.drop(['Item_Outlet_Sales'], axis = 1)
```

Train-test split

```
# let us now split the dataset into train & test
# test_size: the proportion of data to be included in the testing set
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.25, random_state=100)

# print the shape of 'x_train'
print("X_train ",X_train.shape)

# print the shape of 'x_test'
print("X_test ",X_test.shape)

# print the shape of 'y_train'
print("Y_train ",Y_train.shape)

# print the shape of 'y_test'
print("Y_test ",Y_test.shape)
```

```
X_train  (6392, 19)
X_test  (2131, 19)
Y_train  (6392,)
Y_test  (2131,)
```

Proportion of the dataset considered as the test set

To get the specific set of samples over multiple function calls

Train-test split

- The number of samples in train and test set changes as we alter the percentage of the test set using the parameter, ‘test_size’
- By default, the split function returns 75% of the data for training and remaining 25% for testing
- Each integer value of the parameter, ‘random_state’ produces a specific set of train and test samples
- If no random state is specified, the function produces different set of train and test samples for multiple function calls