Data Preprocessing

Data preprocessing is an integral step in Machine Learning as the quality of data and the useful information that can be derived from it directly affects the ability of our model to learn; therefore, it is extremely important that we preprocess our data before feeding it into our model. The concepts that I will cover in this article are-

1. Handling Null Values
2. Standardization
3. Handling Categorical Variables
4. One-Hot Encoding
5. Multicollinearity

Handling Null Values —

In any real-world dataset, there are always few null values. It doesn't really matter whether it is a regression, classification or any other kind of problem, no model can handle these NULL or NaN values on its own, so we need to intervene.

In python NULL is represented with NaN. So don't get confused between these two, they can be used interchangeably.

First of all, we need to check whether we have null values in our dataset or not. We can do that using the isnull() method.

```
df.isnull()
# Returns a boolean matrix, if the value is NaN then True otherwise
Falsedf.isnull().sum()
# Returns the column names along with the number of NaN values in that particular
column
```

There are various ways for us to handle this problem. The easiest way to solve this problem is by dropping the rows or columns that contain null values.

df.dropna()

dropna() takes various parameters like —

1. axis — We can specify axis=0 if we want to remove the rows and axis=1 if we want to remove the columns.
2. how — If we specify how = ‘all’ then the rows and columns will only be dropped if all the values are NaN. By default how is set to ‘any’.
3. thresh — It determines the threshold value so if we specify thresh=5 then the rows having less than 5 real values will be dropped.
4. subset —If we have 4 columns A, B, C and D then if we specify subset=[‘C’] then only the rows that have their C value as NaN will be removed.
5. inplace — By default, no changes will be made to your dataframe. So if you want these changes to reflect onto your dataframe then you need to use inplace = True.

However, it is not the best option to remove the rows and columns from our dataset as it can result in significant information loss. If you have 300K data points then removing 2–3 rows won’t affect your dataset much but if you only have 100 data points and out of which 20 have NaN values for a particular field then you can’t simply drop those rows. In real-world datasets, it can happen quite often that you have a large number of NaN values for a particular field.

Ex — Suppose we are collecting the data from a survey, then it is possible that there could be an optional field which let’s say 20% of people left blank. So, when we get the dataset then we need to understand that the remaining 80% of data is still useful, so rather than dropping these values we need to somehow substitute the missing 20% values. We can do this with the help of **Imputation.**

Imputation — Imputation is simply the process of substituting the missing values of our dataset. We can do this by defining our own customized function or we can simply perform imputation by using the **SimpleImputer** class provided by sklearn.

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
imputer = imputer.fit(df[['Weight']])
df['Weight'] = imputer.transform(df[['Weight']])
```

.values used here return a NumPy representation of the data frame.
Only the values in the data frame will be returned, the axes labels will be removed.

Standardization —

It is another integral preprocessing step. In Standardization, we transform our values such that the mean of the values is 0 and the standard deviation is 1.

	Country	Age	Salary	Purchased
0	France	44.0	72000.000000	No
1	Spain	27.0	48000.000000	Yes
2	Germany	30.0	54000.000000	No
3	Spain	38.0	61000.000000	No
4	Germany	40.0	63777.777778	Yes

Consider the above data frame, here we have 2 numerical values: **Age** and **Weight**. They are not on the same scale as Age is in years and Weight is in Kg and since Weight is more likely to greater than Age; therefore, our model will give more weightage to Weight, which is not the ideal scenario as Age is also an integral factor here. To avoid this issue, we perform Standardization.

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

So, in simple terms, we just calculate the mean and standard deviation of the values and then for each data point we just subtract the mean and divide it by standard deviation.

Example — Consider the column Age from Dataframe 1. In order to standardize this column, we need to calculate the mean and standard deviation and then we will transform each value of age using the above formula. We don't need to do this process manually as sklearn provides a function called **StandardScaler.**

```
from sklearn.preprocessing import StandardScaler
std = StandardScaler()
X = std.fit_transform(df[['Age','Weight']])
```

The important thing to note here is that we need to standardize both training and testing data.

- fit_transform is equivalent to using fit and then transform.
- fit function calculates the mean and standard deviation and the transform function actually standardizes the dataset and we can do this process in a single line of code using the fit_transform function.

Another important thing to note here is that we will use only the transform method when dealing with the test data.

Handling Categorical Variables —

Handling categorical variables is another integral aspect of Machine Learning. Categorical variables are basically the variables that are discrete and not continuous. Ex — color of an item is a discrete variable whereas its price is a continuous variable. Categorical variables are further divided into 2 types —

- **Ordinal categorical variables** — These variables can be ordered. Ex — Size of a T-shirt. We can say that M<L<XL.
- **Nominal categorical variables** — These variables can't be ordered. Ex — Color of a T-shirt. We can't say that Blue<Green as it doesn't make any sense to compare the colors as they don't have any relationship.

The important thing to note here is that we need to preprocess ordinal and nominal categorical variables differently.

Handling Ordinal Categorical Variables — First of all, we need to create a dataframe.

```
df_cat = pd.DataFrame(data =
                [['green','M',10.1,'class1'],
                 ['blue','L',20.1,'class2'],
                 ['white','M',30.1,'class1']])
df_cat.columns = ['color','size','price','classlabel']
```

Here the columns ‘size’ and ‘classlabel’ are ordinal categorical variables whereas ‘color’ is a nominal categorical variable.

There are 2 simple and neat techniques to transform ordinal CVs.

1. Using map() function —

```
size_mapping = {'M':1,'L':2}
df_cat['size'] = df_cat['size'].map(size_mapping)
```

Here M will be replaced with 1 and L with 2.

2. Using Label Encoder —

```
from sklearn.preprocessing import LabelEncoder
class_le = LabelEncoder()
df_cat['classlabel'] =
class_le.fit_transform(df_cat['classlabel'].values)
```

Here class1 will be represented with 0 and class2 with 1.

Incorrect way of handling Nominal Categorical Variables —

The **biggest mistake** that most people make is that they are not able to differentiate between ordinal and nominal CVs. So if you use the same map() function or LabelEncoder with nominal variables then the model will think that there is some sort of relationship between the nominal CVs.

So if we use map() to map the colors like -

```
col_mapping = {'Blue':1,'Green':2}
```

Then according to the model, Green > Blue, which is a senseless assumption, and the model will give you results considering this relationship. So, although you will get the results using this method, they won't be optimal.

Correct way of handling Nominal Categorical Variables —

The correct way of handling nominal CVs is to use One-Hot Encoding. The easiest way to use One-Hot Encoding is to use the get_dummies() function.

```
df_cat = pd.get_dummies(df_cat[['color','size','price']])
```

Here we have passed 'size' and 'price' along with 'color' but the get_dummies() function is pretty smart and will consider only the string variables. So, it will just transform the 'color' variable. Now, you must be wondering what the hell is this One-Hot Encoding. So, let's try and understand it.

One-Hot Encoding — So in One-Hot Encoding what we essentially do is that we create 'n' columns where n is the number of unique values that the nominal variable can take.

Ex — Here if color can take Blue,Green and White then we will just create three new columns namely — color_blue,color_green and color_white and if the color is green then the values of color_blue and color_white column will be 0 and value of color_green column will be 1 . So out of the n columns, only one column can have value = 1 and the rest all will have value = 0.

One-Hot Encoding is a cool and neat hack but there is only one problem associated with it and that is **Multicollinearity.** As you all must have assumed that it is a heavy word so it must be difficult to understand, so let me just validate your newly formed belief. Multicollinearity is indeed a slightly tricky but extremely important concept of Statistics. The good thing here is that we don't really need to understand all the nitty-gritty details of multicollinearity, rather we just need to focus on how it will impact our model. So, let's dive into this concept of Multicollinearity and how it will impact our model.

Multicollinearity and its impact — Multicollinearity occurs in our dataset when we have features that are strongly dependent on each other. Ex- In this case we have features -

color_blue,color_green and color_white which are all dependent on each other and it can impact our model.

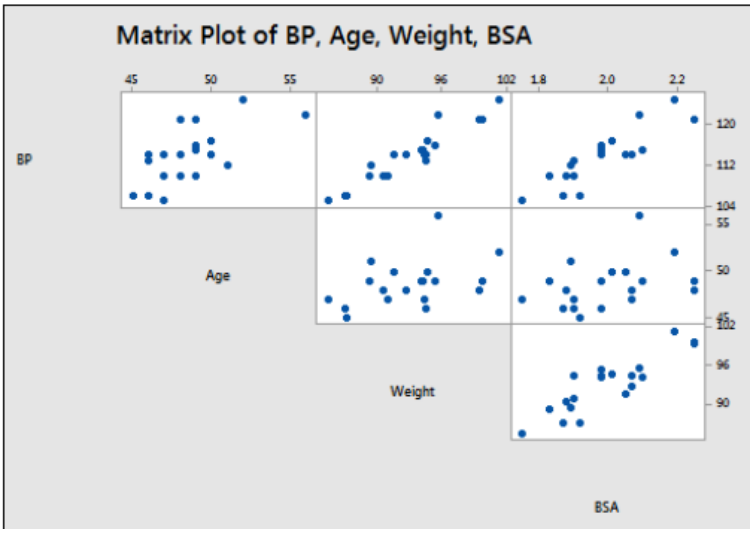
If we have multicollinearity in our dataset then we won't be able to use our weight vector to calculate the feature importance. Multicollinearity impacts the interpretability of our model.

12.1 - What is Multicollinearity?

multicollinearity exists whenever two or more of the predictors in a regression model…

Now that we have understood what Multicollinearity is, let's now try to understand how to identify it.

- The easiest method to identify Multicollinearity is to just plot a pair plot and you can observe the relationships between different features. If you get a linear relationship between 2 features, then they are strongly correlated with each other and there is multicollinearity in your dataset.



Here (Weight, BP) and (BSA, BP) are closely related. You can also use the correlation matrix to check how closely related the features are.

Correlation: BP, Age, Weight, BSA, Dur, Pulse, Stress

	BP	Age	Weight	BSA	Dur	Pulse
Age	0.659					
Weight	0.950	0.407				
BSA	0.866	0.378	0.875			
Dur	0.293	0.344	0.201	0.131		
Pulse	0.721	0.619	0.659	0.465	0.402	
Stress	0.164	0.368	0.034	0.018	0.312	0.506

We can observe that there is a strong co-relation (0.950) between Weight and BP and between BSA and BP (0.875).

Simple hack to avoid Multicollinearity-

We can use drop_first=True in order to avoid the problem of Multicollinearity.

```
df_cat = pd.get_dummies(df_cat[['color','size','price']],drop_first=True)
```

Here drop_first will drop the first column of color. So here color_blue will be dropped and we will only have color_green and color_white.The important thing to note here is that we don’t lose any information because if color_green and color_white are both 0 then it implies that the color must have been blue. So we can infer the whole information with the help of only these 2 columns, hence the strong correlation between these three columns is broken.