Suicide Rate Prediction with Machine Learning

PROJECT REPORT

Group Number : 3

Group Members :

- **Shubham Deshmukh**
- **Rameshwari Jadhav**
- **Samarasinha Baisani**

Date:
12/16/2024

Semester:
Fall 2024

Course Name:
Machine Learning

Table of Contents

1. PROJECT OBJECTIVES

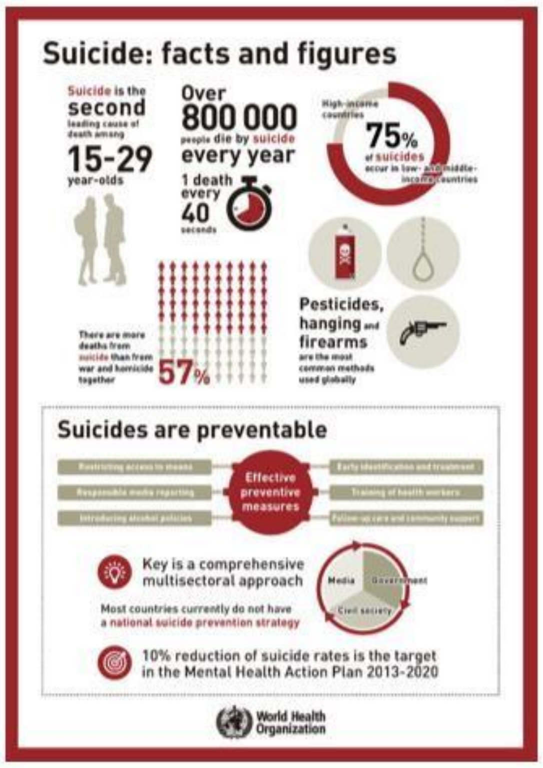

Suicide: facts and figures
Suicide is the second leading cause of death among 15-29 year-olds
Over 800 000 people die by suicide every year
1 death every 40 seconds
High-income countries
75% of suicides occur in low- and middle-income countries
There are more deaths from suicide than from war and homicide together
57%
Pesticides, hanging and firearms are the most common methods used globally
Suicides are preventable
Restricting access to means
Responsible media reporting
Introducing alcohol policies
Effective preventive measures
Early identification and treatment
Training of health workers
Follow-up care and community support
Key is a comprehensive multisectoral approach
Media
Government
Civil society
Most countries currently do not have a national suicide prevention strategy
10% reduction of suicide rates is the target in the Mental Health Action Plan 2013-2020
World Health Organization

Figure 1: Suicide Facts & Figures

Suicide is a serious public health problem. The World Health Organization (WHO) estimates that every year close to 800 000 people take their own life, which is one person every 40 seconds and there are many more people who attempt suicide. Suicide occurs throughout the lifespan and was the second leading cause of death among 15-29-year-olds globally in 2016.

Suicide does not just occur in high-income countries but is a global phenomenon in all regions of the world. In fact, over 79% of global suicides occurred in low- and middle-income countries in 2016. On average, in US there are 129 suicides per day.

The objective of this project is to predict the suicide rates using Machine Learning algorithms and to analyzing significant patterns features that result in increase of suicide rates globally. This project is done in the Google Collaboratory.

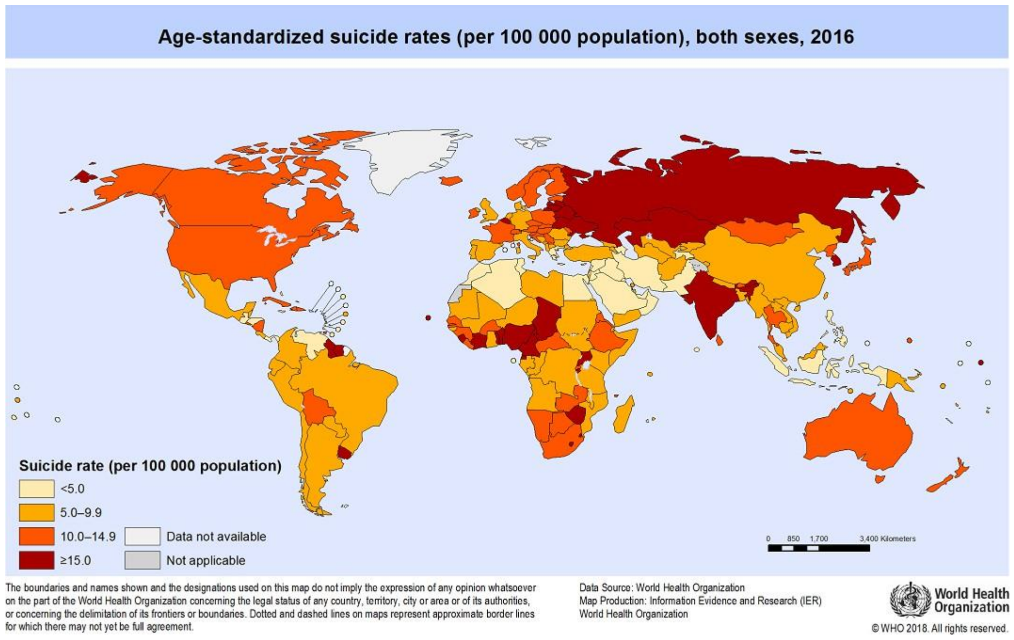

Age-standardized suicide rates (per 100 000 population), both sexes, 2016
Suicide rate (per 100 000 population)
<5.0
5.0–9.9
10.0–14.9
≥15.0
Data not available
Not applicable
0 850 1,700 3,400 Kilometers
The boundaries and names shown and the designations used on this map do not imply the expression of any opinion whatsoever on the part of the World Health Organization concerning the legal status of any country, territory, city or area or of its authorities, or concerning the delimitation of its frontiers or boundaries. Dotted and dashed lines on maps represent approximate border lines for which there may not yet be full agreement.
Data Source: World Health Organization
Map Production: Information Evidence and Research (IER)
World Health Organization
World Health Organization
© WHO 2018. All rights reserved.

Figure 2: Age – Suicide rate Distribution Worldwide

2. DATASET DETAILS

The dataset is borrowed from Kaggle. This is a compiled dataset pulled from four other datasets linked by time and place from year 1985 to 2016. The source of those datasets is WHO, World Bank, UNDP and a dataset published in Kaggle. The details of the dataset are:

- **Number of Instances:** 27820
- **Number of Attributes:** 12

The below table defines attributes in the dataset:

No.	Attribute Name	Description
1	country	Name of country
2	year	Year of the incident: 1985 to 2016
3	sex	Gender: male or female
4	age	Range of age in years
5	suicides_no	Number of incidents
6	population	Corresponding population of the country
7	country-year	Combination of country and year
8	HDI for year	Human development index (HDI) for year
9	gdp_for_year ($)	GDP of the country for the year
10	gdp_per_capita ($)	GDP per capita of the country for the year
11	generation	Generation of the person
12	suicides/100k pop	Number of suicides for 100k population

The generation feature includes the following categories:

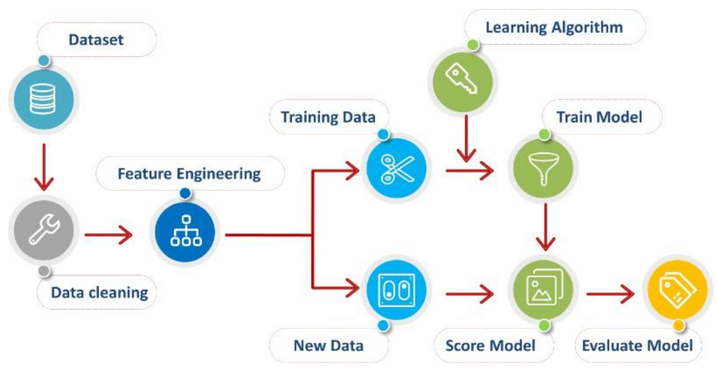
- Generation Z: Born 1996 – TBD
- Millennials: Born 1977 – 1995
- Generation X: Born 1965 – 1976
- Baby Boomers: Born 1946 – 1964
- Silent: Born 1927 - 1945
- G.I. Generation: Born 1901 – 1926

The age of a person is categorized into 6 age groups shown below:

- 75+ years
- 55-74 years
- 35-54 years
- 25-34 years
- 15-24 years
- 5-14 years

3. APPROACH

The following steps are implemented to build a required supervised machine learning model to predict the suicide rate of a country:



1. Data Loading from the CSV file.
2. Understanding the data.
3. Visualizing the data.
4. Preparing the data for the model.
5. Splitting the data.
6. Modeling & training.
7. Model Evaluation

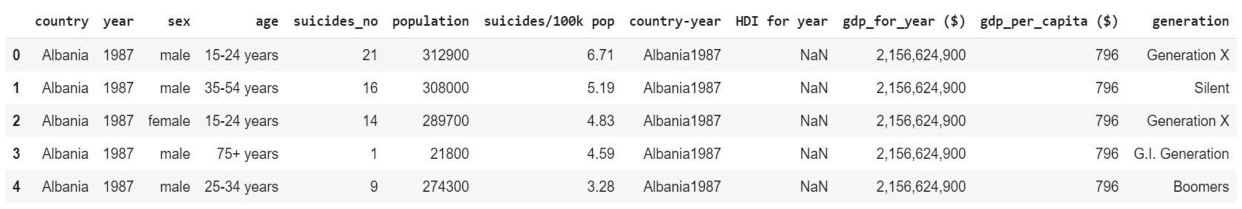
Figure 3: Machine Learning Approach

Working & results of each of these steps are elucidated in detail along with the required code snippets and other details.

3.1. LOADING DATASET

Initially, all the basic necessary libraries like Pandas, Numpy, Scikit-learn, pyplot, Seaborn etc, are imported into Jupyter Notebook. These are the main required libraries for building and training machine learning models. If any other libraries are required in the future, they can be imported accordingly.

The ~30k samples of data in the CSV file are loaded into Pandas dataframe using read_csv() function. This function returns the data in the CSV file as a two-dimensional data structure with labeled axes, called dataframe as shown below:

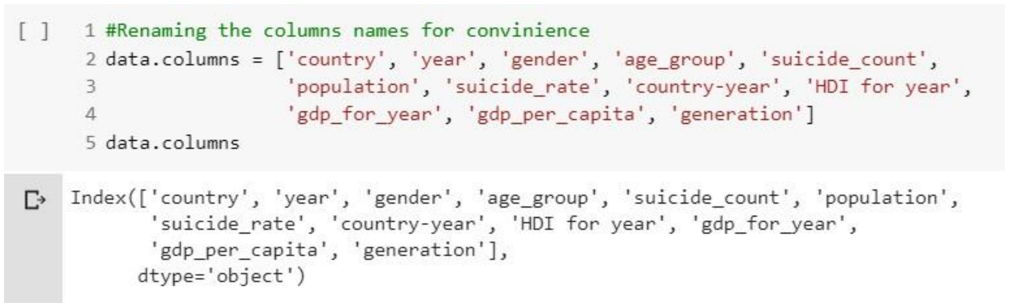


	country	year	sex	age	suicides_no	population	suicides/100k pop	country-year	HDI for year	gdp_for_year ($)	gdp_per_capita ($)	generation
0	Albania	1987	male	15-24 years	21	312900	6.71	Albania1987	NaN	2,156,624,900	796	Generation X
1	Albania	1987	male	35-54 years	16	308000	5.19	Albania1987	NaN	2,156,624,900	796	Silent
2	Albania	1987	female	15-24 years	14	289700	4.83	Albania1987	NaN	2,156,624,900	796	Generation X
3	Albania	1987	male	75+ years	1	21800	4.59	Albania1987	NaN	2,156,624,900	796	G.I. Generation
4	Albania	1987	male	25-34 years	9	274300	3.28	Albania1987	NaN	2,156,624,900	796	Boomers

Figure 4: Dataframe showing few samples of data from CSV file

3.2. FAMILIARIZING WITH DATA

After successful storage of data in the dataframe, one can view the data in the tabular format and can access each part of data easily. The dataframe is of the shape *(27820, 12)*, which indicated that the number of samples and features in the dataset. Few of the give column names are renames for the convenient usage of them in this project. And the new list of the column names of the dataframe are as follows:

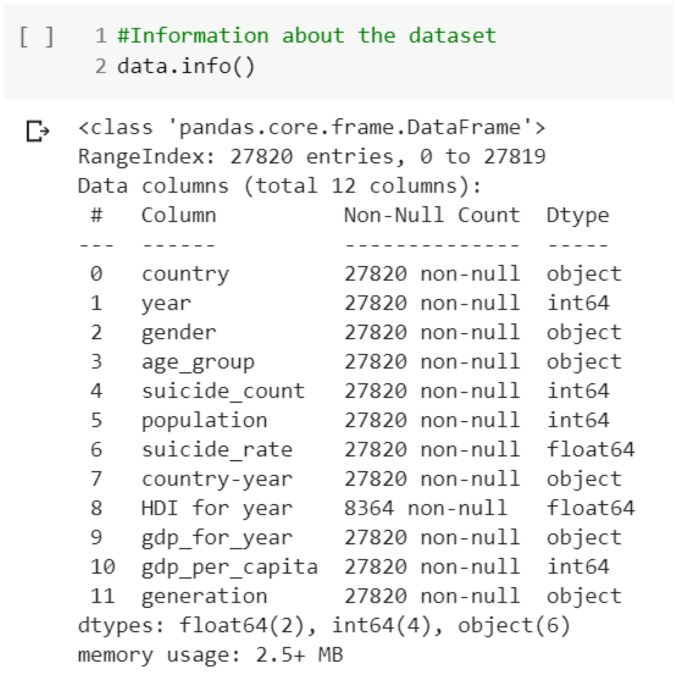


```
#Renaming the columns names for convinience
data.columns = ['country', 'year', 'gender', 'age_group', 'suicide_count',
                'population', 'suicide_rate', 'country-year', 'HDI for year',
                'gdp_for_year', 'gdp_per_capita', 'generation']
data.columns
```

```
Index(['country', 'year', 'gender', 'age_group', 'suicide_count', 'population',
       'suicide_rate', 'country-year', 'HDI for year', 'gdp_for_year',
       'gdp_per_capita', 'generation'],
      dtype='object')
```

Figure 5: Renaming dataset columns

The detailed information about the dataset along with the code snippet is shown below:



```
#Information about the dataset
data.info()
```

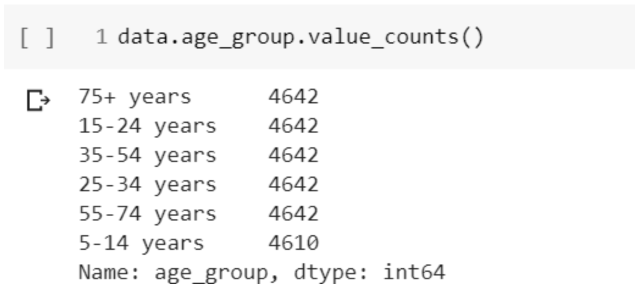
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 27820 entries, 0 to 27819
Data columns (total 12 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   country         27820 non-null  object 
 1   year            27820 non-null  int64  
 2   gender          27820 non-null  object 
 3   age_group       27820 non-null  object 
 4   suicide_count   27820 non-null  int64  
 5   population      27820 non-null  int64  
 6   suicide_rate    27820 non-null  float64
 7   country-year    27820 non-null  object 
 8   HDI for year    8364 non-null   float64
 9   gdp_for_year    27820 non-null  object 
 10  gdp_per_capita  27820 non-null  int64  
 11  generation      27820 non-null  object 
dtypes: float64(2), int64(4), object(6)
memory usage: 2.5+ MB
```

Figure 6: Dataset Information & corresponding code snippet

The following observations are made after seeing the data in the dataframe shown in Figure 3:

- Categorical features are country, year, sex, age group, country-year, generation (based on age grouping average).

- Numerical features are count of suicides, population, suicide rate, HDI for year, gdp_for_year, gdp_per_capita.
- *'HDI for year'* column has missing values. None of the other columns have any missing values. (Interpreted from Figure 6.)
- The age feature has 6 unique age group. Age is grouped into year buckets as categorical format which needs to be encoded.

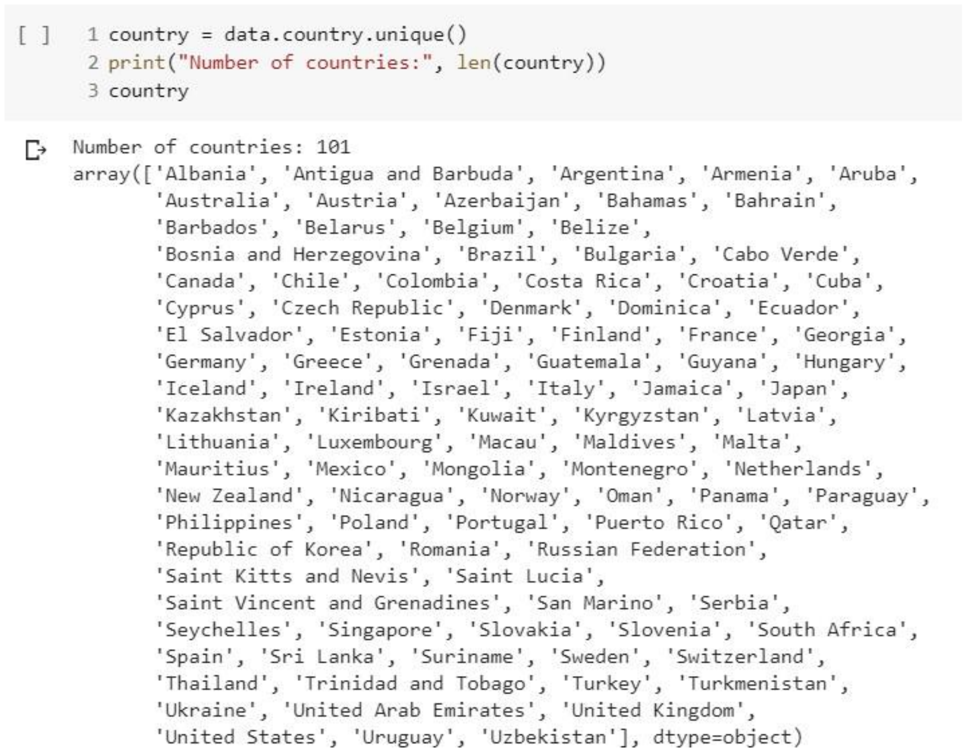


```
[ ]   data.age_group.value_counts()
```

```
75+ years      4642
15-24 years    4642
35-54 years    4642
25-34 years    4642
55-74 years    4642
5-14 years     4610
Name: age_group, dtype: int64
```

Figure 7: Number of Samples in each Age Group

- Similarly, few of the categorical values needs to be encoded.
- The total number of countries in the dataset are 101 and are shown below:



```
[ ]   country = data.country.unique()
      print("Number of countries:", len(country))
      country
```

```
Number of countries: 101
array(['Albania', 'Antigua and Barbuda', 'Argentina', 'Armenia', 'Aruba',
       'Australia', 'Austria', 'Azerbaijan', 'Bahamas', 'Bahrain',
       'Barbados', 'Belarus', 'Belgium', 'Belize',
       'Bosnia and Herzegovina', 'Brazil', 'Bulgaria', 'Cabo Verde',
       'Canada', 'Chile', 'Colombia', 'Costa Rica', 'Croatia', 'Cuba',
       'Cyprus', 'Czech Republic', 'Denmark', 'Dominica', 'Ecuador',
       'El Salvador', 'Estonia', 'Fiji', 'Finland', 'France', 'Georgia',
       'Germany', 'Greece', 'Grenada', 'Guatemala', 'Guyana', 'Hungary',
       'Iceland', 'Ireland', 'Israel', 'Italy', 'Jamaica', 'Japan',
       'Kazakhstan', 'Kiribati', 'Kuwait', 'Kyrgyzstan', 'Latvia',
       'Lithuania', 'Luxembourg', 'Macau', 'Maldives', 'Malta',
       'Mauritius', 'Mexico', 'Mongolia', 'Montenegro', 'Netherlands',
       'New Zealand', 'Nicaragua', 'Norway', 'Oman', 'Panama', 'Paraguay',
       'Philippines', 'Poland', 'Portugal', 'Puerto Rico', 'Qatar',
       'Republic of Korea', 'Romania', 'Russian Federation',
       'Saint Kitts and Nevis', 'Saint Lucia',
       'Saint Vincent and Grenadines', 'San Marino', 'Serbia',
       'Seychelles', 'Singapore', 'Slovakia', 'Slovenia', 'South Africa',
       'Spain', 'Sri Lanka', 'Suriname', 'Sweden', 'Switzerland',
       'Thailand', 'Trinidad and Tobago', 'Turkey', 'Turkmenistan',
       'Ukraine', 'United Arab Emirates', 'United Kingdom',
       'United States', 'Uruguay', 'Uzbekistan'], dtype=object)
```

Figure 8: Countries in the dataset

Till now we loaded the data and understood what the data is about and the details of it. Now let's go into next step and see how the data is distributed with some plots and graphs.

3.3. VISUALIZING THE DATA

The dataset is visualized by plotting few graphs/plots using famous matplotlib and seaborn libraries. And the plots are shown below. To understand the distribution of all attributes in the given dataset, individual bar graphs are generated. The distribution graphs are shown below:

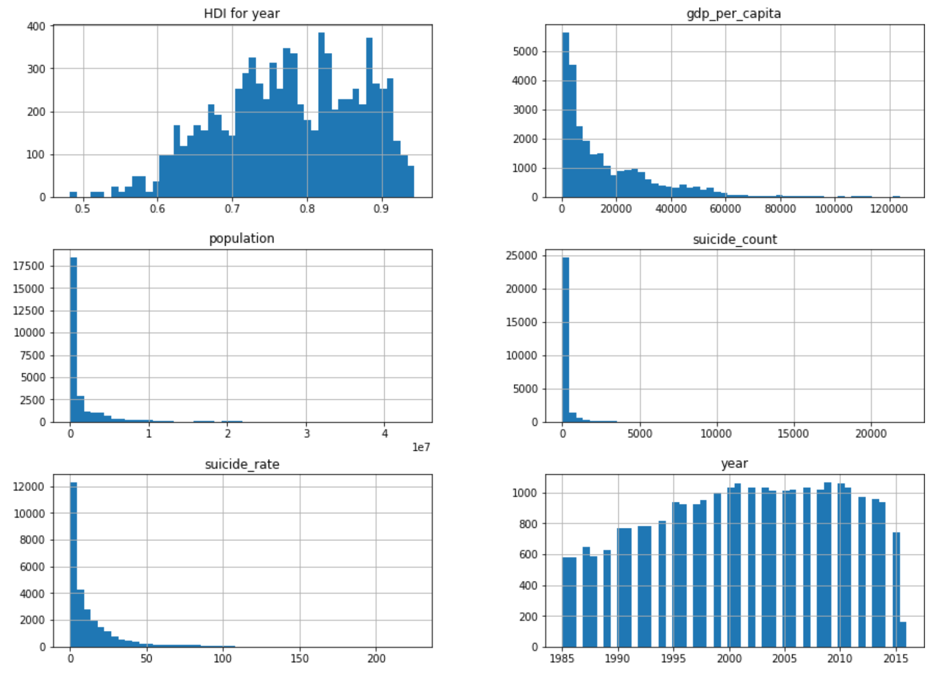


Figure 9: Distribution graphs of features in the dataset

To observe the relation between each attribute of the dataset, a correlation heatmap is generated and is shown below:

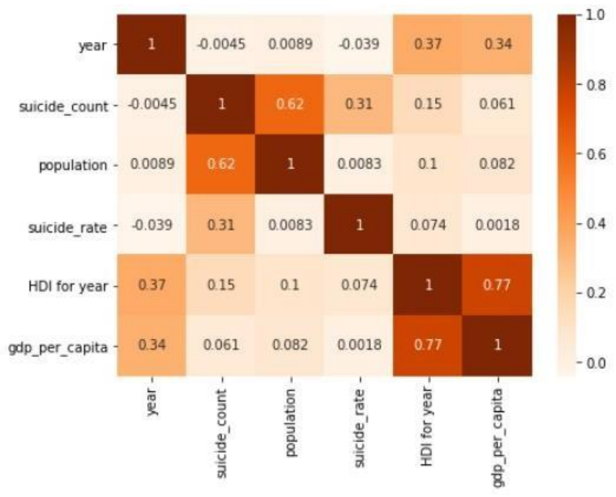


Figure 10: Heatmap of the dataset

The below bar plot shows the number of suicides in male and female population and we can interpret that the male population are more prone to suicide than the female.

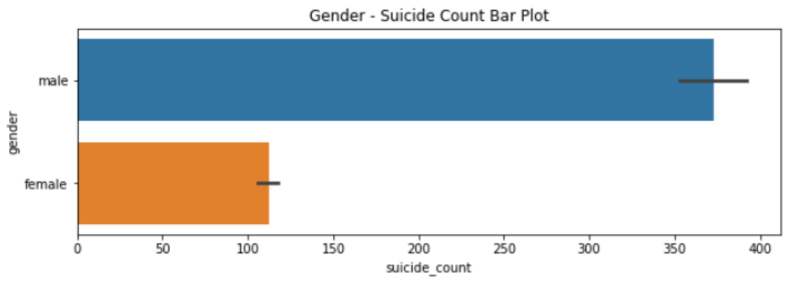


Figure 11: Bar plot Gender – Suicide count

Now, let's check the suicide cases based on the age group and generation separately. And the corresponding bar plots are shown below:

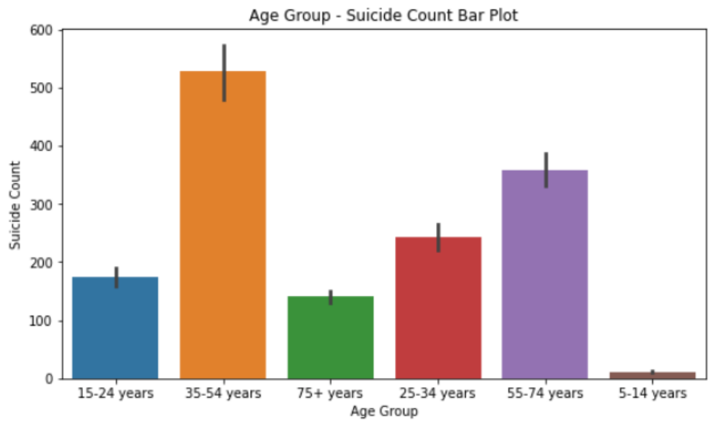


Figure 12: Bar plot of Generation & suicide count grouped by gender

The above boxplot shows that the suicide cases are more in the age group of 35-54 years followed by 55- 74 years. The surprising part is that the suicide cases in 5-14 year age group even though they are very less, mostly in tens. And lets see suicide count distribution in generation.

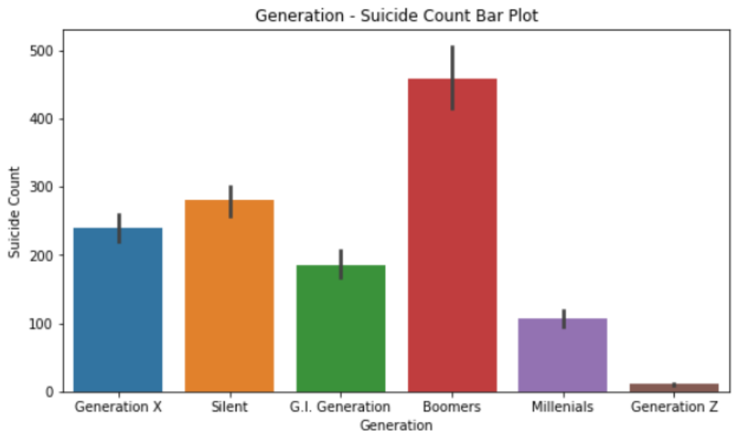


Figure 13: Bar plot of Generation & suicide count grouped by gender

Observation from the above plot are as below:

- The cases are more in the boomers, silent and X generations. These generations are made up of people born until 1976 based on the details provided.
- On further observation, these generations are the ones were most of them are in the age group where most suicides occur.

Now, let's see if all the above mentioned pattern exists in all the age groups, generations and also considering gender. So, the required bar plots are as show below:

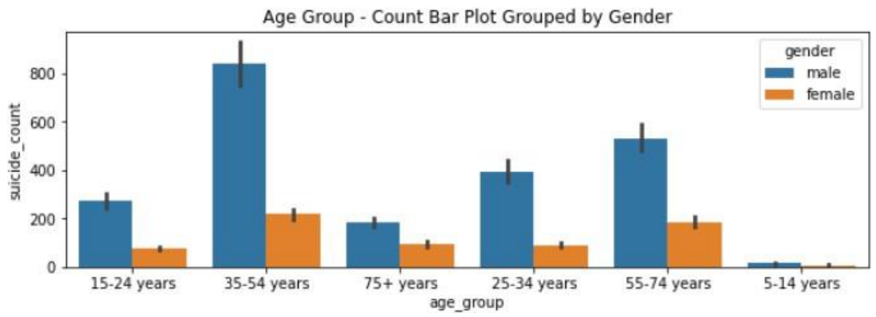


Figure 14: Bar plot of Age group & suicide count grouped by gender

The above bar plot stated that the 35-54 years age group is more prone to suicides irrespective of the gender followed by 55-74 years age group irrespective of the gender.

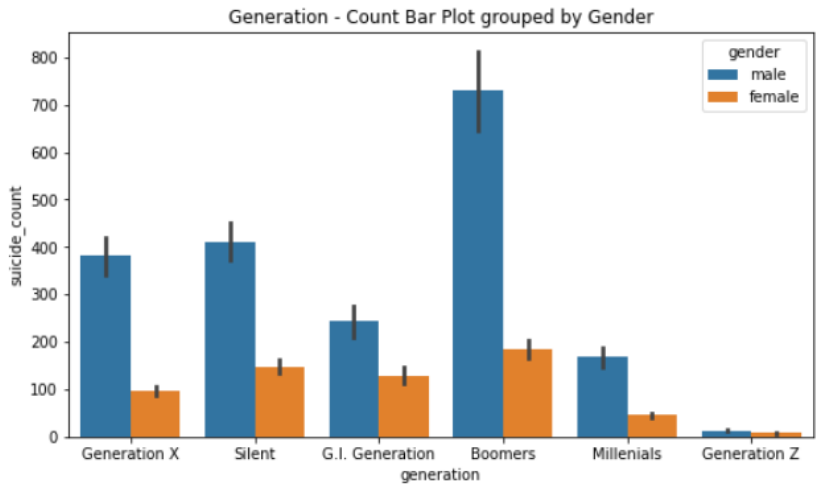


Figure 15: Bar plot of Generation & suicide count grouped by gender

The above bar plot stated that the Boomers generation has more suicide cases followed by Silent generation irrespective of the gender.

From the above four bar plots, it is clear that men commit suicide considerably more than women irrespective of age group and generation they belong to.

The next plot is about the countries and their suicide rate which is shown below:

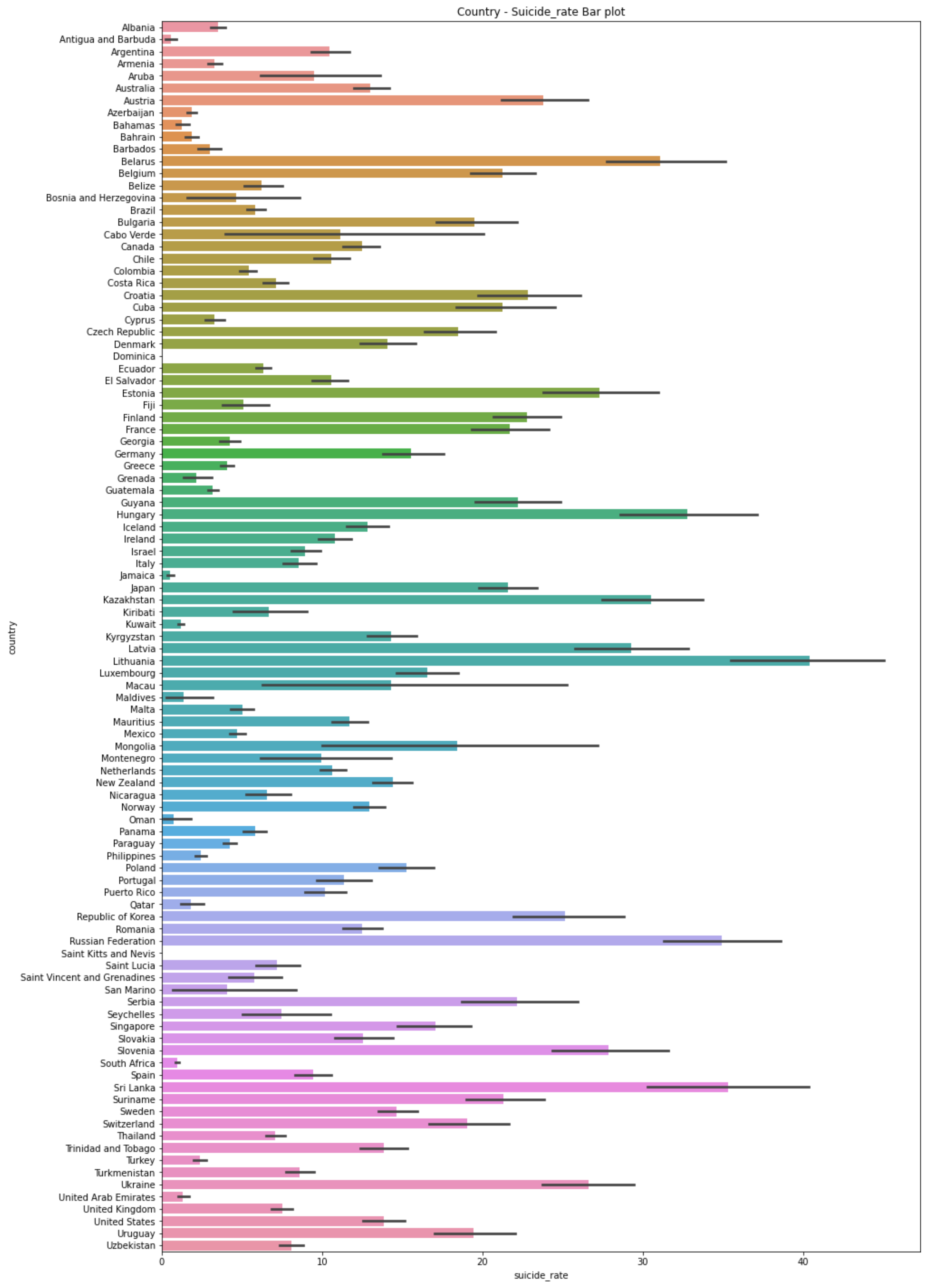


Figure 16: Bar plot of Countries & Suicide rate

The above bar plot shows that the highest suicide rate country is Lithuania followed by Sri Lanka.

We saw the suicide cases distribution across the countries so, lets even see it across the years. The corresponding plot is as shown below:

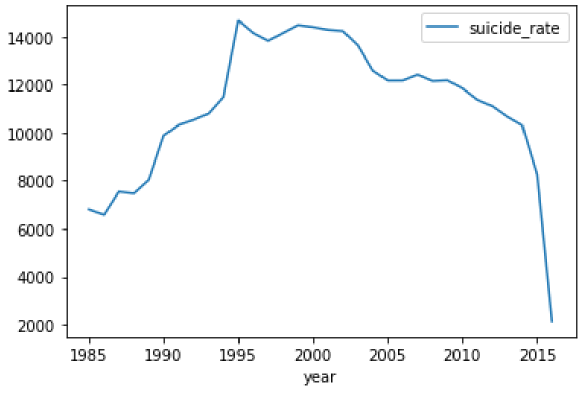


Figure 17: Line plot of Years & Suicide rate

The observation from the above plot are that the suicide rate had grown rapidly from year 1990 & the rate of suicide has drastically reduced in year 2016. The dataset was collected during early 2016. So, all the suicide cases of 2016 are not recorded in the dataset.

The final visualization of the dataset is its scatter matrix. This plot helps in to have a look as the outlies in the features of the data and also their distribution. The scatter matrix is shown below:

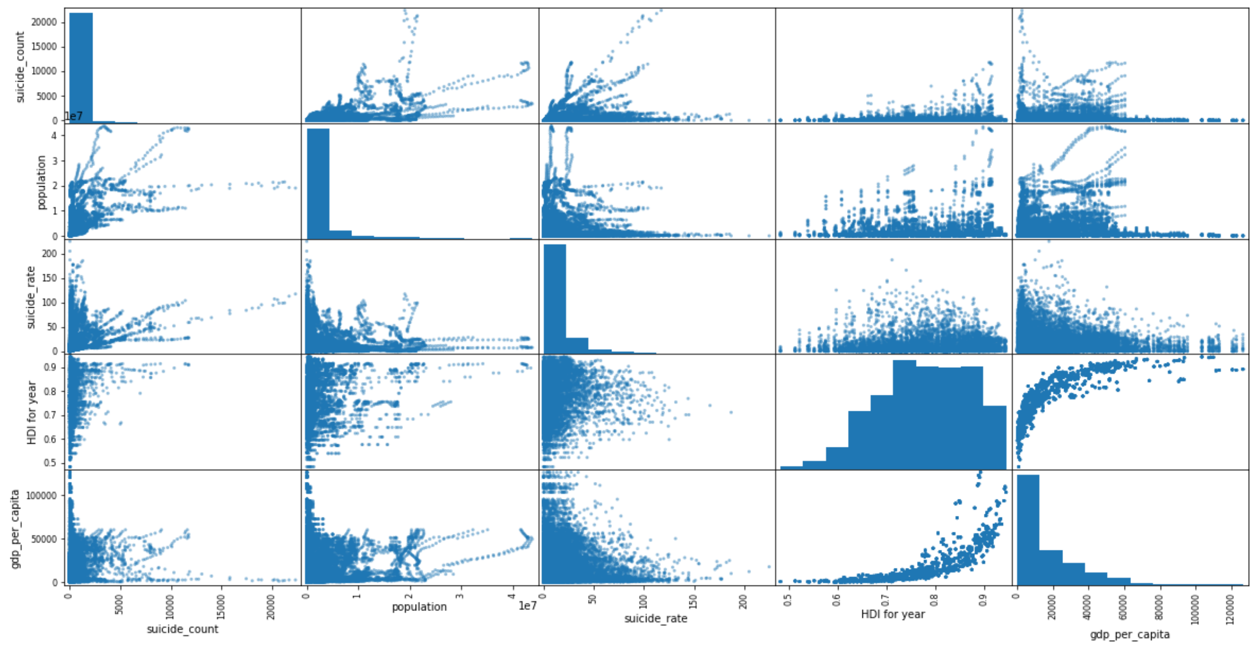


Figure 18: Scatter Matrix of Dataset

From the scatter matrix, it is obvious that the data has outlier & these are addressed during data preprocessing by scaling and encoding the features.

3.4. DATA EXPLORATION & PREPROCESSING

In the data preprocessing step of Machine Learning, the data gets transformed, or encoded, to bring it to such a state that now the machine can easily parse it. In other words, the features of the data can now be easily interpreted by the algorithm. The steps involved in the data preprocessing are shown below:

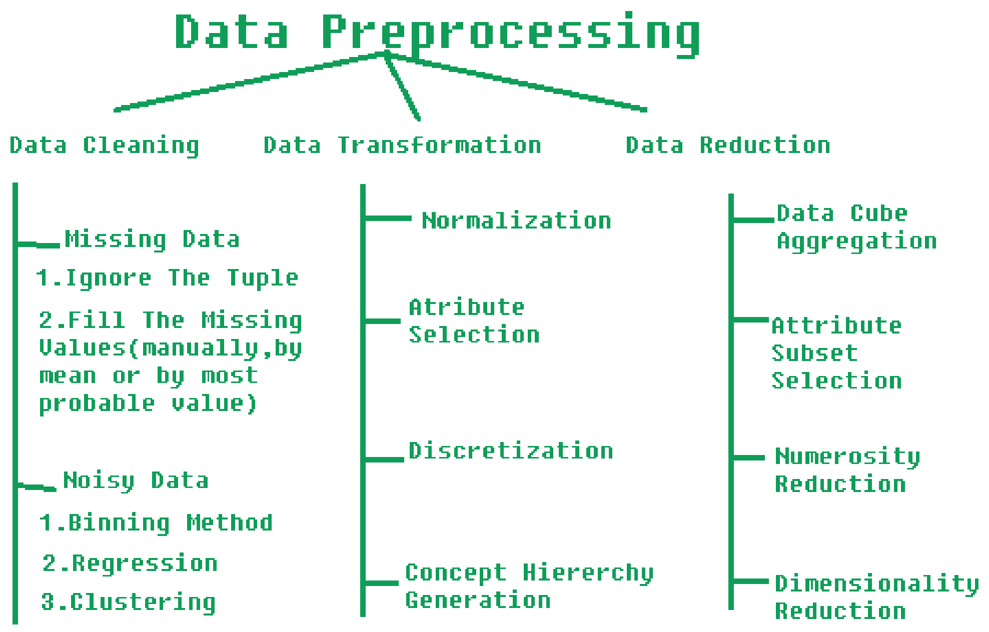
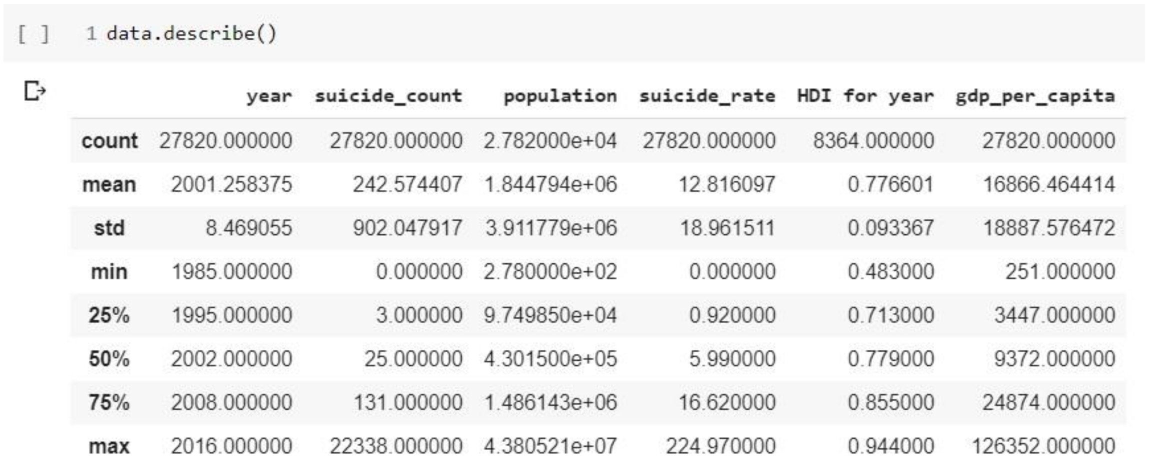


Figure 19: Data Preprocessing Steps

Out of these steps, the steps specific to our dataset are performed. Initially, lets see the numerical distribution of the dataset. The *'describe()'* function of dataframes gives the following statistics:

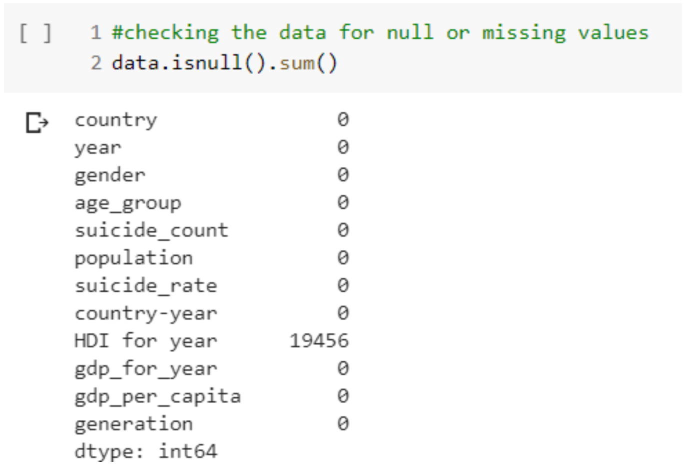


```
data.describe()
```

	year	suicide_count	population	suicide_rate	HDI for year	gdp_per_capita
count	27820.000000	27820.000000	2.782000e+04	27820.000000	8364.000000	27820.000000
mean	2001.258375	242.574407	1.844794e+06	12.816097	0.776601	16866.464414
std	8.469055	902.047917	3.911779e+06	18.961511	0.093367	18887.576472
min	1985.000000	0.000000	2.780000e+02	0.000000	0.483000	251.000000
25%	1995.000000	3.000000	9.749850e+04	0.920000	0.713000	3447.000000
50%	2002.000000	25.000000	4.301500e+05	5.990000	0.779000	9372.000000
75%	2008.000000	131.000000	1.486143e+06	16.620000	0.855000	24874.000000
max	2016.000000	22338.000000	4.380521e+07	224.970000	0.944000	126352.000000

Figure 20: Statistics of Numerical Columns

Let’s check each column of the dataset any null or missing values by using the one of the two functions, isnull() or notnull(). Upon using these functions on the dataframe object, we got the values of 0 and 20,000 respectively for each attribute as shown below:

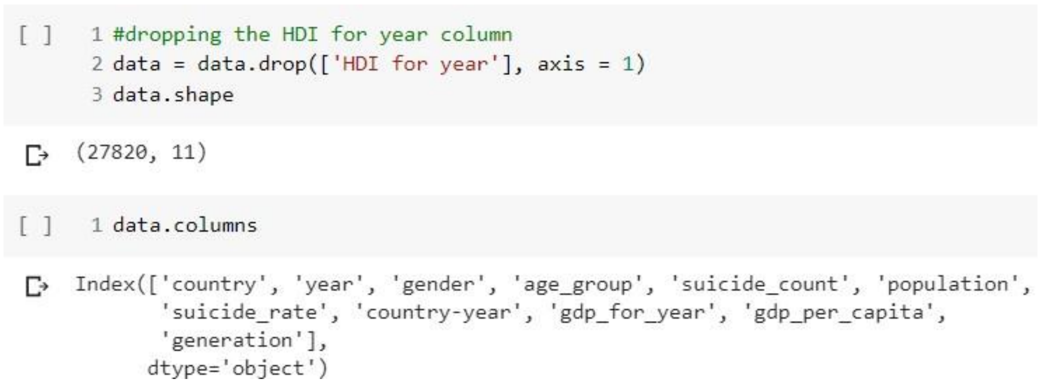


```
[ ]  #checking the data for null or missing values
     data.isnull().sum()
```

```
country              0
year                 0
gender               0
age_group            0
suicide_count        0
population           0
suicide_rate         0
country-year         0
HDI for year     19456
gdp_for_year         0
gdp_per_capita       0
generation           0
dtype: int64
```

Figure 21: Null / Missing value Check Result

From the above stats, it is clear that the column, *‘HDI for year’* has 19456 null values out of 27820 samples which is approximately 70% of the column data. This may tamper the model performance so, dropping the HDI for year column from the dataset as shown below:



```
[ ]  #dropping the HDI for year column
     data = data.drop(['HDI for year'], axis = 1)
     data.shape
```

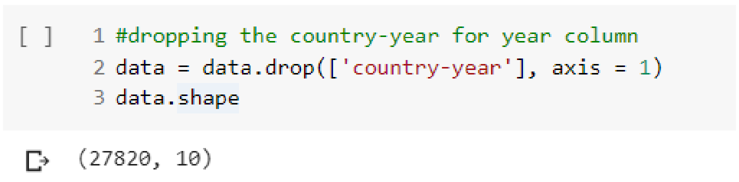
```
(27820, 11)
```

```
[ ]  data.columns
```

```
Index(['country', 'year', 'gender', 'age_group', 'suicide_count', 'population',
       'suicide_rate', 'country-year', 'gdp_for_year', 'gdp_per_capita',
       'generation'],
      dtype='object')
```

Figure 22: Dropping *HDI for year* column

The other redundant column I observed in the dataset is *‘country-year’* column. This is just a combination of country and year columns which doesn’t have a significance to the model. So, dropping the *‘country-year’* column as shown below:

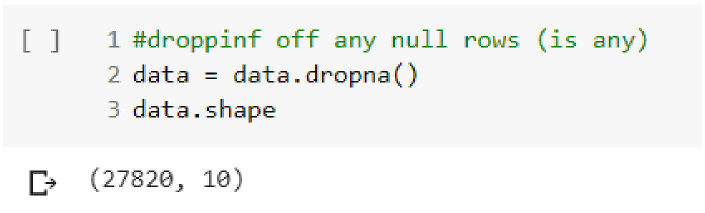


```
[ ]  #dropping the country-year for year column
     data = data.drop(['country-year'], axis = 1)
     data.shape
```

```
(27820, 10)
```

Figure 23: Dropping *country-year* column

From now on, we are going further with 10 features which also include the target column. For further assurance, let's use *dropna()* function on the dataset to drop all the null rows from it as shown below:



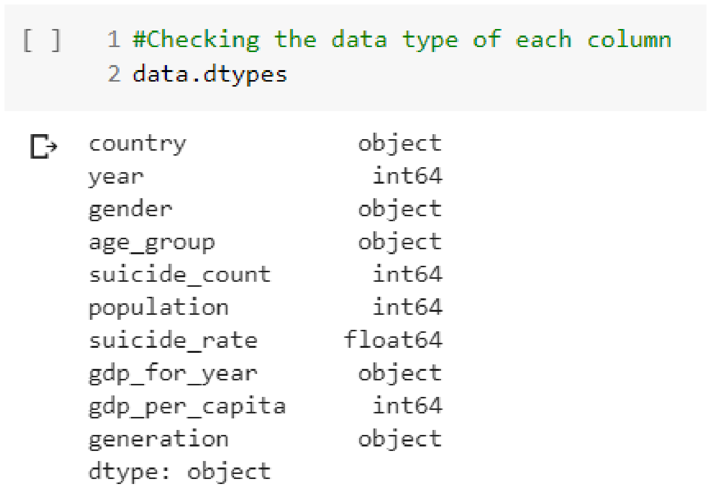
```
[ ]  #droppinf off any null rows (is any)
     data = data.dropna()
     data.shape

(27820, 10)
```

Figure 24: Dropping off null rows

After dropping the HDI for year column, from the above execution it is clear that the dataset doesn't have any null or missing values. Moving on to standardization and encoding steps of data preprocessing.

Standardization of numerical columns in the dataset is a common requirement for many machine learning estimators: they might behave badly if the individual features do not more or less look like standard normally distributed data. Apart from this, the dataset also has man outliers which can be suppressed after standardization.

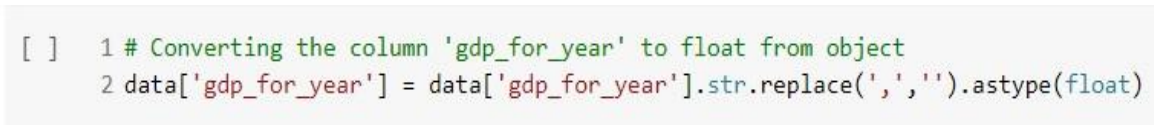
Before starting the standardization, let's check the data type of each column in the dataset to ensure that the type of numerical data is either int or float.



```
[ ]  #Checking the data type of each column
     data.dtypes

country            object
year                int64
gender             object
age_group          object
suicide_count       int64
population          int64
suicide_rate      float64
gdp_for_year       object
gdp_per_capita      int64
generation         object
dtype: object
```

Figure 25: Checking data type of each column

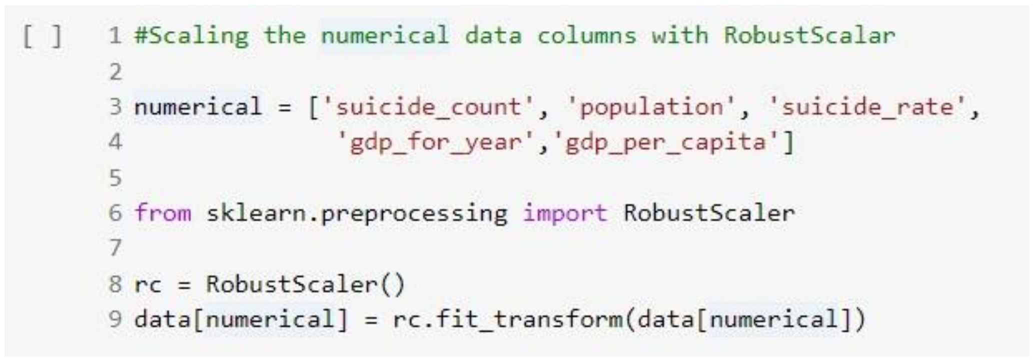
The above code snippet shows that the column *'gdp_for_year'* is of type object. On checking the data in that column, it is clear that the numerical values have comma (,) in them. So, clearning the data in the *'gdp_for_year'* column as follows:



```
[ ]   # Converting the column 'gdp_for_year' to float from object
      data['gdp_for_year'] = data['gdp_for_year'].str.replace(',','').astype(float)
```

Figure 26: Cleaning data in *gdp*_for_year column & converting it to float

Now, the numerical columns, population, gdp_for_year & gdp_per_capita are ready toe be standardized. Scikit-learn library has many standardizing methods like StandardScalar(), MinMaxScalar() and others. Out of all of the, I chose Scikit-learn's RobustScalar as it works well on data with outliers and use more robust estimates for the center and range of the data. The code snippet for this task is shown below:

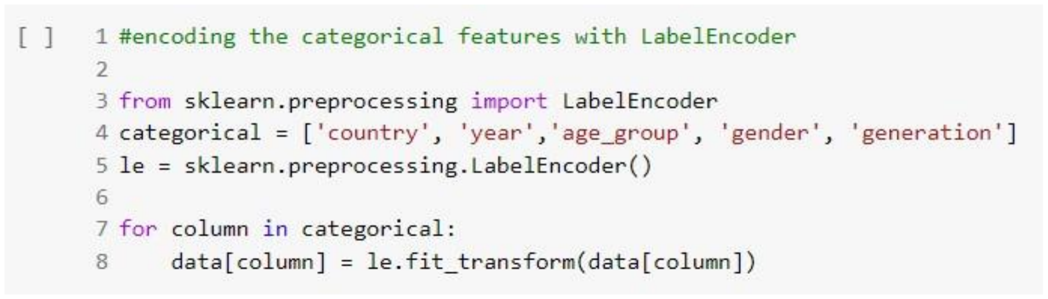


```
[ ]   #Scaling the numerical data columns with RobustScalar

      numerical = ['suicide_count', 'population', 'suicide_rate',
                   'gdp_for_year','gdp_per_capita']

      from sklearn.preprocessing import RobustScaler

      rc = RobustScaler()
      data[numerical] = rc.fit_transform(data[numerical])
```

Figure 27: Standardizing data with RobustScalar

We dealt with numerical columns of the dataset. Now let's see the categorical columns of the dataset. We know the categorical columns of the dataset are country, year, gender, age_group and generation. Similar to standardization functions, Scikit-learn has many encoding functions like OneHotEncoder, OriginalEncoder and others.

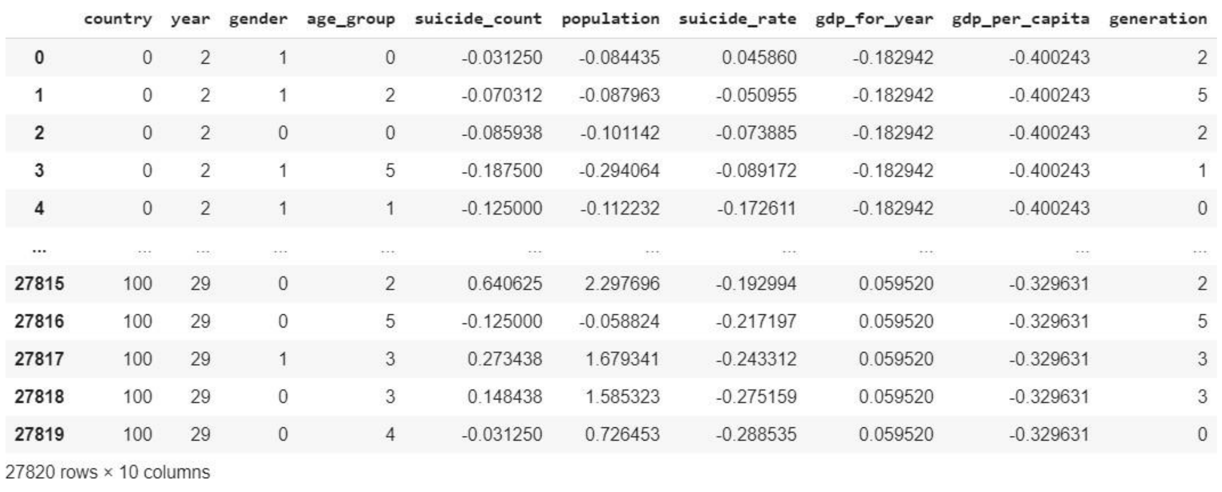
For this dataset, I choose LabelEncoder to encode the dataset. Label Encoding refers to converting the labels into numeric form so as to convert it into the machine-readable form. The encoding coding snippet is as shown below:



```
[ ]   #encoding the categorical features with LabelEncoder

      from sklearn.preprocessing import LabelEncoder
      categorical = ['country', 'year','age_group', 'gender', 'generation']
      le = sklearn.preprocessing.LabelEncoder()

      for column in categorical:
          data[column] = le.fit_transform(data[column])
```

Figure 28: Encoding data with LabelEncoder

After the required data preprocessing steps are done, the data is transformed into as shown below:



	country	year	gender	age_group	suicide_count	population	suicide_rate	gdp_for_year	gdp_per_capita	generation
0	0	2	1	0	-0.031250	-0.084435	0.045860	-0.182942	-0.400243	2
1	0	2	1	2	-0.070312	-0.087963	-0.050955	-0.182942	-0.400243	5
2	0	2	0	0	-0.085938	-0.101142	-0.073885	-0.182942	-0.400243	2
3	0	2	1	5	-0.187500	-0.294064	-0.089172	-0.182942	-0.400243	1
4	0	2	1	1	-0.125000	-0.112232	-0.172611	-0.182942	-0.400243	0
...	...	...	...	...	...	...	...	...	...	...
27815	100	29	0	2	0.640625	2.297696	-0.192994	0.059520	-0.329631	2
27816	100	29	0	5	-0.125000	-0.058824	-0.217197	0.059520	-0.329631	5
27817	100	29	1	3	0.273438	1.679341	-0.243312	0.059520	-0.329631	3
27818	100	29	0	3	0.148438	1.585323	-0.275159	0.059520	-0.329631	3
27819	100	29	0	4	-0.031250	0.726453	-0.288535	0.059520	-0.329631	0

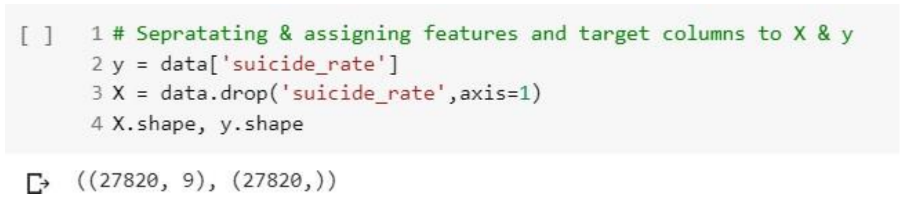
27820 rows × 10 columns

Figure 29: Data after Scaling and Encoding

The data is now thoroughly processed and is ready for the next step i.e., splitting the data.

3.5. SPLITTING THE DATA

Before getting into data splitting to train and test data, we have to create feature (X) and target (y) variables from the dataset. In the suicide dataset, I considered suicide_rate of country as the target variable and the rest of them are considered as the features. The code snippet of this task is as follows:



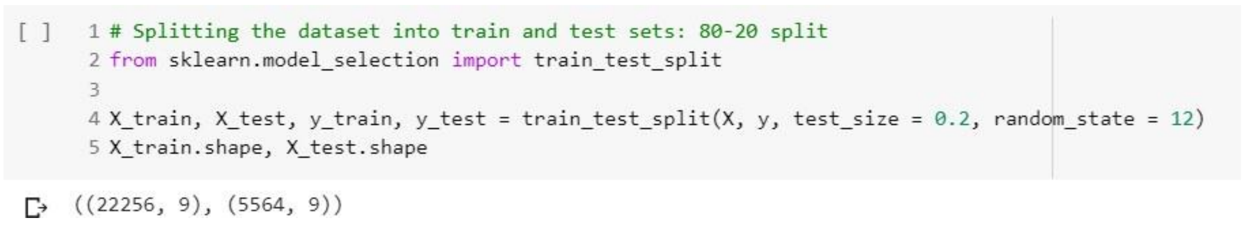
```
[ ]  # Sepratating & assigning features and target columns to X & y
     y = data['suicide_rate']
     X = data.drop('suicide_rate',axis=1)
     X.shape, y.shape
```

```
((27820, 9), (27820,))
```

Figure 30: Creating X & y

After forming the input (X) and target (y) variables, the entire dataset needs to be split into train and test datasets. We train our model on the train data and test the accuracy of built model prediction or classification on the test data. By splitting the dataset, we are not doing any changes to the test data which gives unaltered results of our model efficiency.

The dataset splitting is done by using predefined 'train_test_split()' function from scikit-learn as shown below. And the dataset is split into 80% of train data and 20% of test data, basically an 80-20 split.



```
[ ]
# Splitting the dataset into train and test sets: 80-20 split
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 12)
X_train.shape, X_test.shape
```

```
((22256, 9), (5564, 9))
```

Figure 31: Splitting the dataset

The input and target variables of train data are denoted by X_train & y_train and for test data, X_test & y_test respectively. After splitting, the training dataset has 22,256 samples and the test dataset has 5,564 samples. Now, the supervised machine learning models are trained with the training dataset in the next step.

3.6. MODEL BUILDING & TRAINING

Machine Learning algorithms are off two types: Supervised and Unsupervised algorithms. The type that is focused for this project is Supervised algorithms. Supervised machine learning is one of the most commonly used and successful types of machine learning. Supervised learning is used whenever we want to predict a certain outcome/label from a given set of features, and we have examples of features-label pairs.

We build a machine learning model from these features-label pairs, which comprise our training set. Our goal is to make accurate predictions for new, never-before-seen data. Further, there are two major types of supervised machine learning problems, called classification and regression. Basically, in classification, the goal is to predict a class label, which is a choice from a predefined list of possibilities.

For regression tasks, the goal is to predict a continuous number, or a floating-point number in programming terms (or real number in mathematical terms). The suicide data set comes under regression problem, as the prediction of suicide rate is a continuous number. There are wide variety of supervised machine learning algorithms or models. Among them the below mentioned models (regression) are parameter tuned, if needed and trained on the dataset.

1. k-Nearest Neighbors
2. Linear Regression
3. Decision Tree
4. Random Forest
5. Gradient Boosting
6. Multilayer Perceptrons (MLP)
7. XGBoost

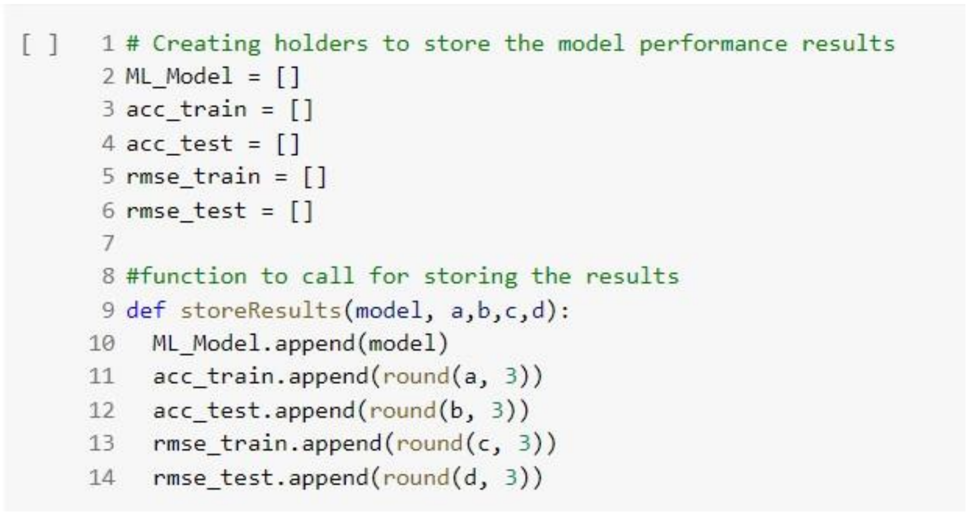
8. Bagging Regressor
9. Custom Ensemble

The above mentioned parameter tune is nothing but the model hyperparameters are tuned in such a way that the model shows optimum performance on the test dataset. The beat parameters for the model can be obtained by using Scikit Learn's GridSearchCV method. This method is applied on all the algorithms.

For the evaluation of the performance of these models, the metrics considered are Accuracy & Root Mean Squared Error (RMSE). The generic process of training a model and evaluating is as follows:

- Import the model from the Scikit-Learn (if the library can be used).
- Instantiate the model and tune if the parameters if required.
- Fit the training data to the model to train it.
- The model is ready for the predictions on the test data.
- Calculate the model performance evaluation metrics.

The above mentioned five steps are applied on each model and the detailed execution is mentioned below. Before jumping into the models, I created a function to store the evaluation results of each model to a list. This step is done to make the comparison of the models easy. The code for this is as follows:



```
[ ]
# Creating holders to store the model performance results
ML_Model = []
acc_train = []
acc_test = []
rmse_train = []
rmse_test = []

#function to call for storing the results
def storeResults(model, a,b,c,d):
  ML_Model.append(model)
  acc_train.append(round(a, 3))
  acc_test.append(round(b, 3))
  rmse_train.append(round(c, 3))
  rmse_test.append(round(d, 3))
```

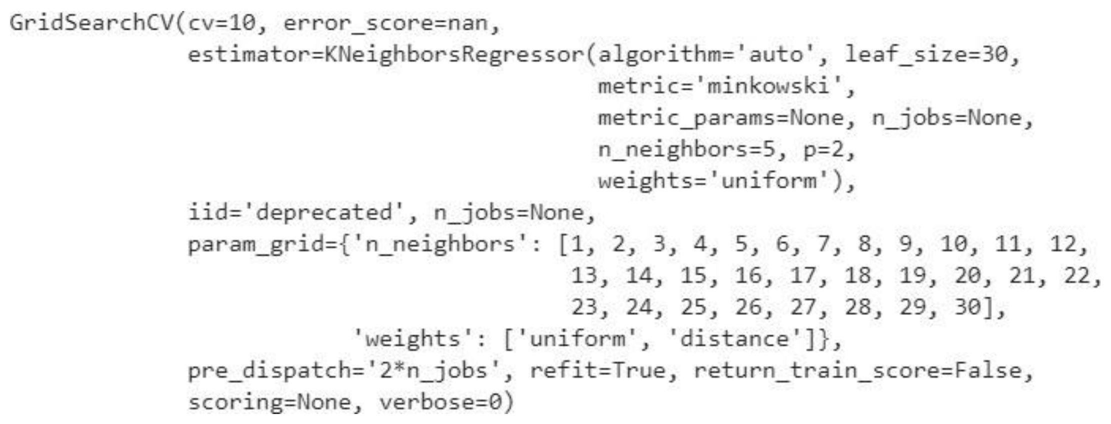
Figure 32: To Store the Model Results

After tuning the parameters and calculating the performance of every model, the above function is called to store the training & test data accuracy & RMSE. Later this stored data is used to compare the models and determine the suitable model with set hyperparameters to the suicide dataset.

3.6.1. k-Nearest Neighbors (KNN):

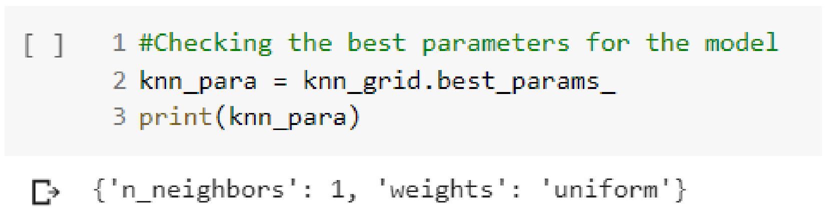
K nearest neighbors is a simple algorithm that stores all available cases and predict the numerical target based on a similarity measure (e.g., distance functions). It can be performed on both classification and regression problems. A simple implementation of KNN regression is to calculate the average of the numerical target of the k nearest neighbors.

The most frequently tuned parameter of KNN regression are *‘n_neighbors’* & *‘weight’*. Without tuning it, the model gave me the accuracy on test data of 69.1%. So, hyperparameter tuning is done for the model with the GridSearchCV on the mentioned parameter. The model and the obtained best parameters for the model are:



```
GridSearchCV(cv=10, error_score=nan,
             estimator=KNeighborsRegressor(algorithm='auto', leaf_size=30,
                                           metric='minkowski',
                                           metric_params=None, n_jobs=None,
                                           n_neighbors=5, p=2,
                                           weights='uniform'),
             iid='deprecated', n_jobs=None,
             param_grid={'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                         13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
                                         23, 24, 25, 26, 27, 28, 29, 30],
                         'weights': ['uniform', 'distance']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

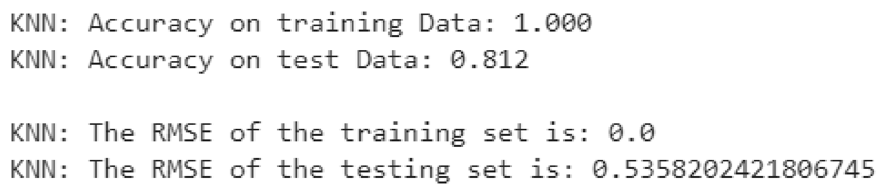
Figure 33: KNN Regression Model



```
[ ]  #Checking the best parameters for the model
     knn_para = knn_grid.best_params_
     print(knn_para)

{'n_neighbors': 1, 'weights': 'uniform'}
```

Figure 34: KNN Best Parameters

For these parameters, the train and test dataset accuracy are shown below:



```
KNN: Accuracy on training Data: 1.000
KNN: Accuracy on test Data: 0.812

KNN: The RMSE of the training set is: 0.0
KNN: The RMSE of the testing set is: 0.5358202421806745
```

Figure 35: KNN Model Results

Evaluating training and testing set performance with different numbers of neighbors from 1 to 30 and weight as uniform. The plot shows the training and test set accuracy on the y-axis against the setting of n_neighbors on the x-axis:

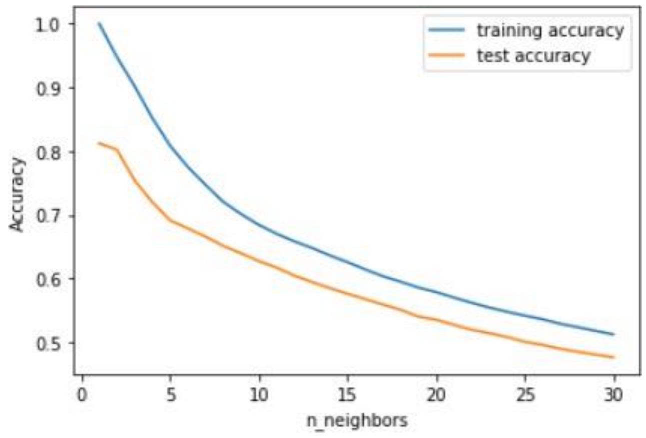
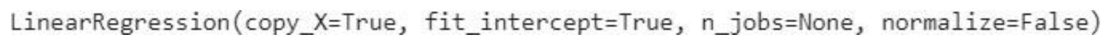


Figure 36: KNN Model Results for multiple n_neighbors

This discrepancy between performance on the training set and the testing set fro n_neighbors < 5 is a clear sign of overfitting. After that, the performance is not soo great so, moving on to the other models.

3.6.2. Linear Regression

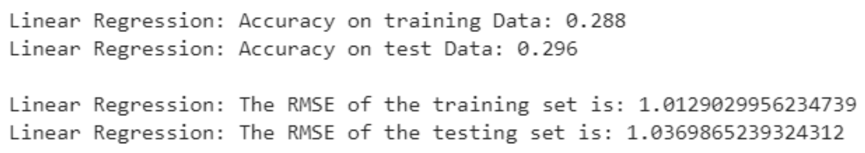
Linear regression, or ordinary least squares (OLS), is the simplest and most classic linear method for regression. Linear regression finds the parameters w and b that minimize the mean squared error between predictions and the true regression targets, y, on the training set. The parameter tuning is not done on this model as the model has no parameters to be tuned for. The model is as shown below:



```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

Figure 37: Linear Regression Model

The linear regression model with hyperparameter tuning gave the following accuracy:



```
Linear Regression: Accuracy on training Data: 0.288
Linear Regression: Accuracy on test Data: 0.296

Linear Regression: The RMSE of the training set is: 1.0129029956234739
Linear Regression: The RMSE of the testing set is: 1.0369865239324312
```

Figure 38: Linear Regression Model Results

The model performance is not very good, but we can see that the scores on the training and test sets are very close together. This means we are likely underfitting, not overfitting.

Note to Remember: *The most common ML algorithms, logistic regression and linear support vector machines (linear SVMs), are supervised classification algorithms, can't be applied on regression problems. Despite its name, Logistic Regression is a classification algorithm and not a regression algorithm, and it should not be confused with Linear Regression.*

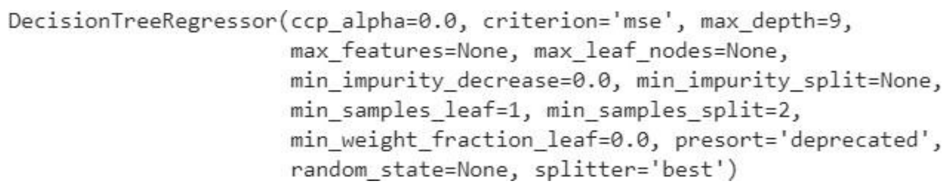
3.6.3. Decision Trees: Regression

Decision trees are widely used models for classification and regression tasks. Essentially, they learn a hierarchy of if/else questions, leading to a decision. Learning a decision tree means learning the sequence of if/else questions that gets us to the true answer most quickly.

In the machine learning setting, these questions are called tests (not to be confused with the test set, which is the data we use to test to see how generalizable our model is). To build a tree, the algorithm searches over all possible tests and finds the one that is most informative about the target variable.

On training the Decision Tree Regression model on the dataset without tweaking the parameters, the obtained training & test data accuracies are 100% & 98.9% respectively. As expected, the accuracy on the training set is 100%—because the leaves are pure, the tree was grown deep enough that it could perfectly memorize all the labels on the training data.

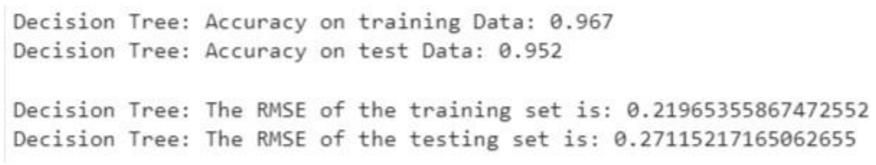
To avoid this overfitting, the parameters should be tuned so that pre-pruning is applied on the tree. To do this, the *'max_depth'* parameter of the model should be restricted. Limiting the depth of the tree decreases overfitting. So multiple values of the parameter are tried on the model and the finalized model is as follows:



```
DecisionTreeRegressor(ccp_alpha=0.0, criterion='mse', max_depth=9,
                      max_features=None, max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, presort='deprecated',
                      random_state=None, splitter='best')
```

Figure 39: Decision Tree Regression Model

On setting the *max_depth* to 9 gives the following results:



```
Decision Tree: Accuracy on training Data: 0.967
Decision Tree: Accuracy on test Data: 0.952

Decision Tree: The RMSE of the training set is: 0.21965355867472552
Decision Tree: The RMSE of the testing set is: 0.27115217165062655
```

Figure 40: Decision Tree Regression Model Results

Evaluating training and testing set performance with different numbers of max_depth from 1 to 30. The plot shows the training and test set accuracy on the y-axis against the setting of max_depth on the x-axis:

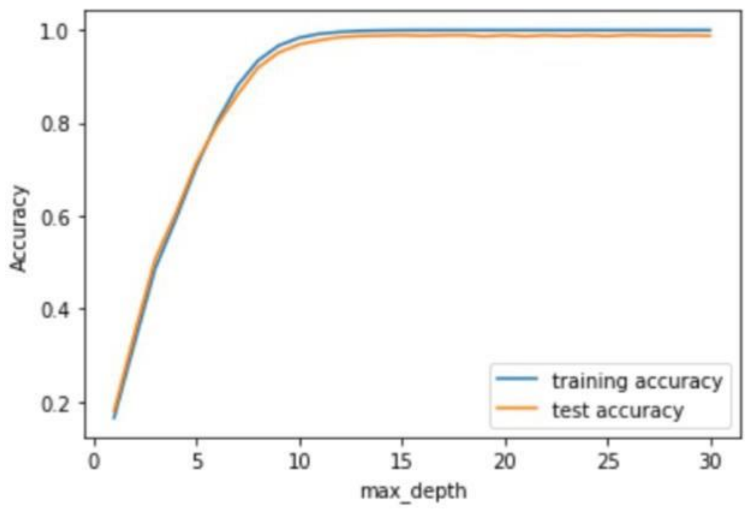


Figure 41: Decision Tree Model Results for multiple max_depth

The model preformance is gradually increased on incresing the max_depth parameter. But after max_depth = 9, the model overfits. So, the model is considered with max_depth = 9 which has an accuracy of 95.2%. The importance of the features for this model is shown below:

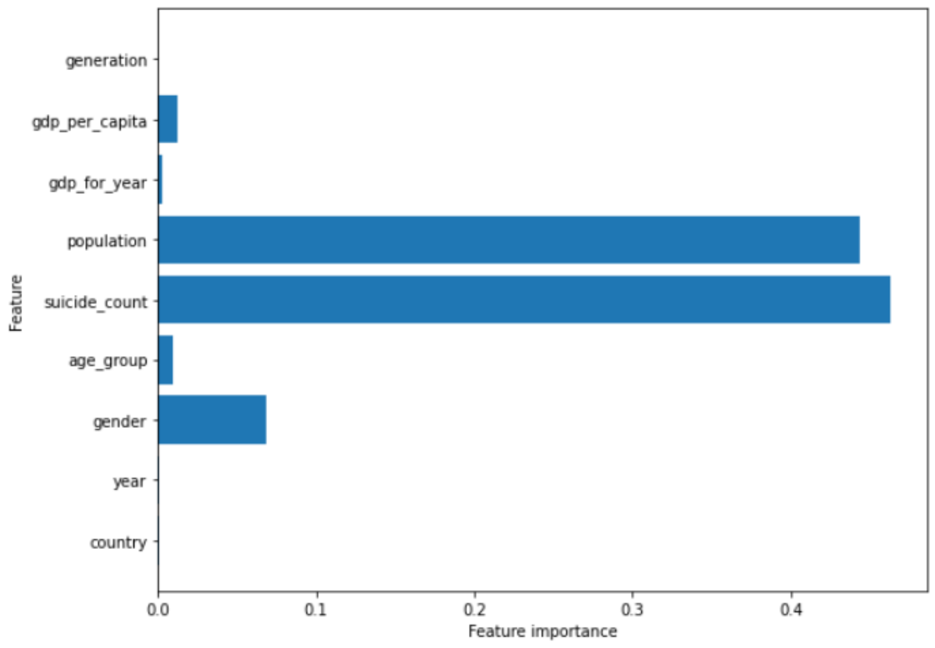


Figure 42: Decision Tree Model Feature Importance

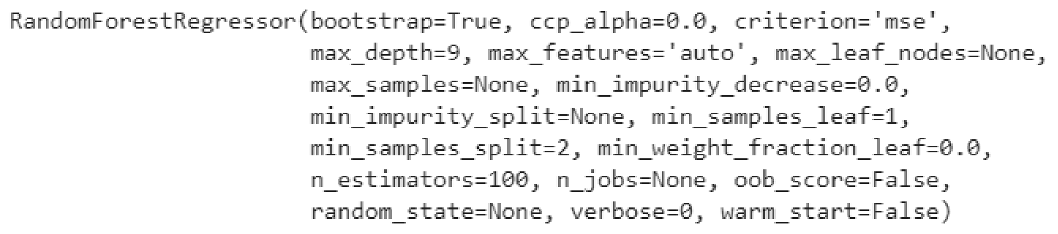
3.6.4. Random Forest: Ensemble of Decision Trees

Random forests for regression and classification are currently among the most widely used machine learning methods. A random forest is essentially a collection of decision trees, where each tree is slightly different from the others. The idea behind random forests is that each tree might do a relatively good job of predicting but will likely overfit on part of the data.

If we build many trees, all of which work well and overfit in different ways, we can reduce the amount of overfitting by averaging their results. To build a random forest model, you need to decide on the number of trees to build (the n_estimators parameter of RandomForestRegressor or RandomForestClassifier). They are very powerful, often work well without heavy tuning of the parameters, and don't require scaling of the data.

On training the Random Forest Regression model on the dataset without tweaking the parameters, the obtained training & test data accuracies are 99.9% & 99.4% respectively. As expected, the accuracy on the training set is almost 100% and this might be the case of overfitting.

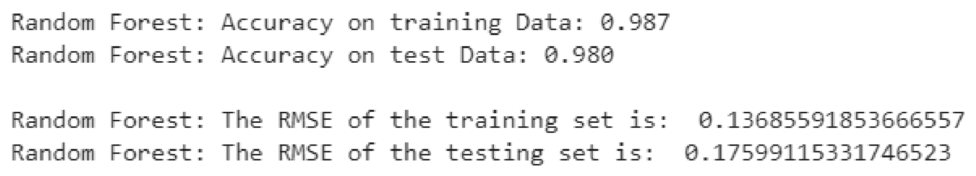
To avoid this overfitting, the parameters should be tuned so that pre-pruning is applied on the tree. To do this, the *'max_depth'* parameter of the model should be restricted by keeping *'n_estimators'* to 100. Limiting the depth of the tree decreases overfitting. So multiple values of the parameter are tried on the model and the finalized model is as follows:



```
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=9, max_features='auto', max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=100, n_jobs=None, oob_score=False,
                      random_state=None, verbose=0, warm_start=False)
```

Figure 43: Random Forest Regression Model

On setting the *max_depth* to 9 gives the following results:



```
Random Forest: Accuracy on training Data: 0.987
Random Forest: Accuracy on test Data: 0.980

Random Forest: The RMSE of the training set is:  0.13685591853666557
Random Forest: The RMSE of the testing set is:  0.17599115331746523
```

Figure 44: Random Forest Regression Model Results

Evaluating training and testing set performance with different numbers of max_depth from 1 to 30. The plot shows the training and test set accuracy on the y-axis against the setting of max_depth on the x-axis:

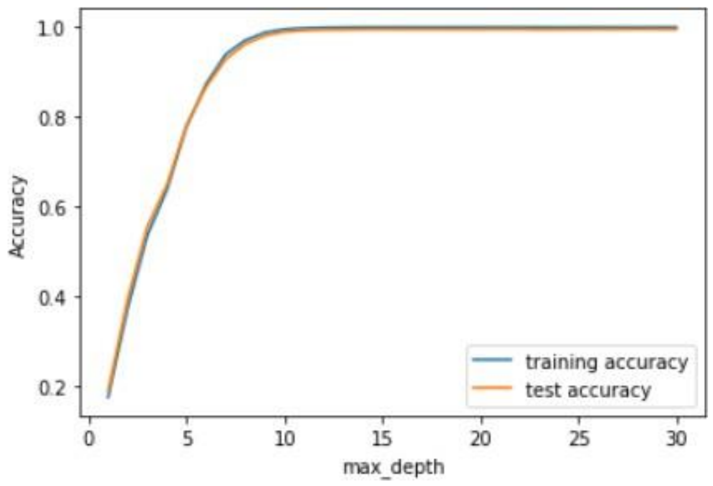


Figure 45: Random Forest Model Results for multiple max_depth

The random forest gives us an accuracy of 99.4%, better than the linear models or a single decision tree, without tuning any parameters. But this might also be a case of overfitting. So, the prarameter are tuned and the finalized model has an accuracy of 98.0% which is better than the linear & decision tree models.

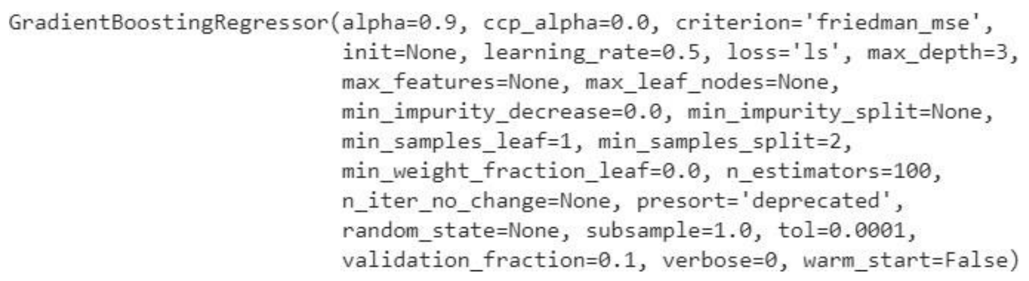
3.6.5. Gradient Boosting: Ensemble of Decision Trees

The gradient boosted regression tree is another ensemble method that combines multiple decision trees to create a more powerful model. Despite the “regression” in the name, these models can be used for regression and classification. In contrast to the random forest approach, gradient boosting works by building trees in a serial manner, where each tree tries to correct the mistakes of the previous one.

On training the Gradient Boost Regression model on the dataset without tweaking the parameters, the obtained training & test data accuracies are 95% & 94.3% respectively. The tunable parameters are learning_rate, n_estimators & max_depth.

In this model, learning_rate parameter controls how strongly each tree tries to correct the mistakes of the previous trees. A higher learning rate means each tree can make stronger corrections, allowing for more complex models. Adding more trees to the ensemble, which can be accomplished by increasing n_estimators, also increases the model complexity, as the model has more chances to correct mistakes on the training set.

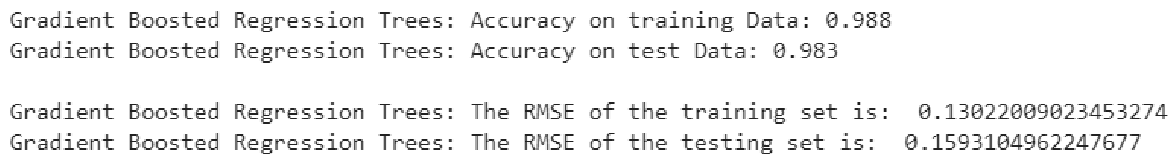
Even upon changing the n_estimators with the combination of other two, there is no change in the model performance. So, the default value of n_estimators is taken and the learning_rate parameter is tuned. Multiple values of the parameter are tried on the model and the finalized model is as follows:



```
GradientBoostingRegressor(alpha=0.9, ccp_alpha=0.0, criterion='friedman_mse',
                          init=None, learning_rate=0.5, loss='ls', max_depth=3,
                          max_features=None, max_leaf_nodes=None,
                          min_impurity_decrease=0.0, min_impurity_split=None,
                          min_samples_leaf=1, min_samples_split=2,
                          min_weight_fraction_leaf=0.0, n_estimators=100,
                          n_iter_no_change=None, presort='deprecated',
                          random_state=None, subsample=1.0, tol=0.0001,
                          validation_fraction=0.1, verbose=0, warm_start=False)
```

Figure 46: Gradient Boosting Regression Model

The model performance after tuning the parameters is as follows:



```
Gradient Boosted Regression Trees: Accuracy on training Data: 0.988
Gradient Boosted Regression Trees: Accuracy on test Data: 0.983

Gradient Boosted Regression Trees: The RMSE of the training set is:  0.13022009023453274
Gradient Boosted Regression Trees: The RMSE of the testing set is:  0.1593104962247677
```

Figure 47: Gradient Boosting Regression Model Results

Evaluating training and testing set performance with different numbers of learning_rate from 0.1 to 0.9. The plot shows the training and test set accuracy on the y-axis against the setting of learning_rate on the x-axis:

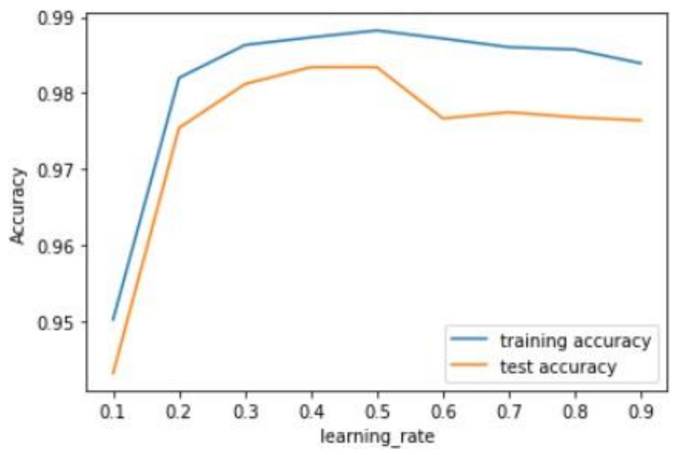


Figure 48: Gradient Boosting Model Results for multiple learning_rate

For the plot we can see that when learning_rate > 0.5, the model performance decreases slightly. The optimized Gradient Boosted model gives us an accuracy of 98.3%, with parameter tuning which is the best performance so far.

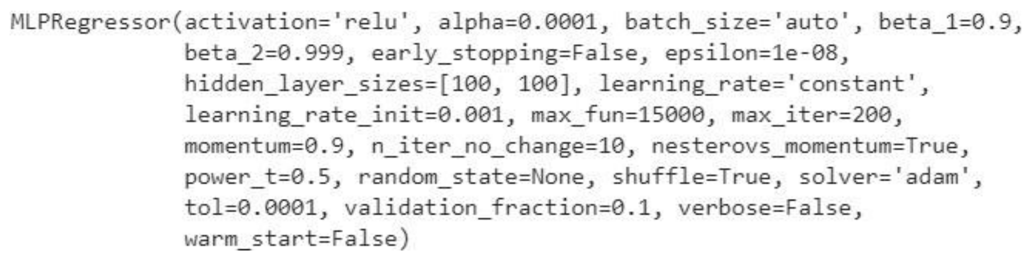
3.6.6. Multilayer Perceptrons (MLPs): Deep Learning

Deep learning algorithms shows great promise in many machine learning applications. Multilayer perceptrons (MLPs) are also known as (vanilla) feed-forward neural networks, or sometimes just neural networks. Multilayer perceptrons can be applied for both classification and regression problems.

MLPs can be viewed as generalizations of linear models that perform multiple stages of processing to come to a decision. In an MLP the process of computing weighted sums is repeated multiple times, first computing hidden units that represent an intermediate processing step, which are again combined using weighted sums to yield the final result

On training the MLP Regression model on the dataset without tweaking the parameters, the obtained training & test data accuracies are 89.8% & 90.5% respectively. The tunable parameters are number of hidden layes, hidden_units in each layer & alpha.

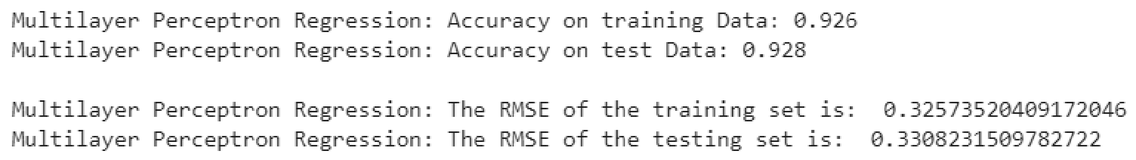
The hidden layers are increased to 2 and the hidden units of each layer is set to 100 with a default alpha value. The resulting model is as follows:



```
MLPRegressor(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,
             beta_2=0.999, early_stopping=False, epsilon=1e-08,
             hidden_layer_sizes=[100, 100], learning_rate='constant',
             learning_rate_init=0.001, max_fun=15000, max_iter=200,
             momentum=0.9, n_iter_no_change=10, nesterovs_momentum=True,
             power_t=0.5, random_state=None, shuffle=True, solver='adam',
             tol=0.0001, validation_fraction=0.1, verbose=False,
             warm_start=False)
```

Figure 49: MLP Regression Model

The tuned multilayer perceptron regression model gives the following results:



```
Multilayer Perceptron Regression: Accuracy on training Data: 0.926
Multilayer Perceptron Regression: Accuracy on test Data: 0.928

Multilayer Perceptron Regression: The RMSE of the training set is:  0.32573520409172046
Multilayer Perceptron Regression: The RMSE of the testing set is:  0.3308231509782722
```

Figure 50: MLP Regression Model Results

The hyperparameter tuning, the model performance increased. And the resulting model has the accuracy of 92.8%.

3.6.7. XGBoost Regression

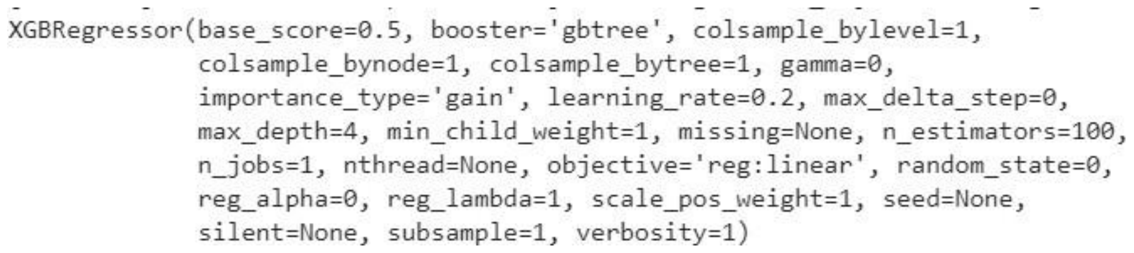
XGBoost is one of the most popular machine learning algorithms these days. XGBoost stands for eXtreme Gradient Boosting. Regardless of the type of prediction task at hand; regression or classification. XGBoost is an implementation of gradient boosted decision trees designed for speed and performance.

XGBoost is an ensemble tree method that apply the principle of boosting weak learners (CARTs generally) using the gradient descent architecture. XGBoost improves upon the base

Gradient Boosting Machines (GBM) framework through systems optimization and algorithmic enhancements.

On training the XGBoost Regression model on the dataset without tweaking the parameters, the obtained training & test data accuracies are 94.9% & 94.3% respectively. The tunable parameters are learning_rate, n_estimators & max_depth.

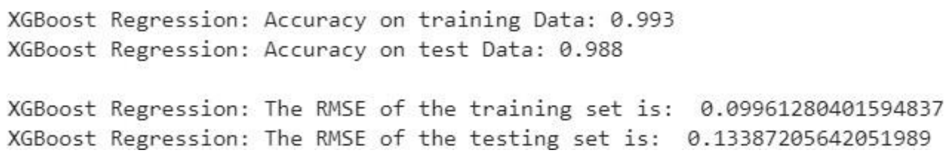
The default value of n_estimators is considered and incremented the learning_rate & max_depth by 1 unit. The resulting model is as follows:



```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=1, gamma=0,
             importance_type='gain', learning_rate=0.2, max_delta_step=0,
             max_depth=4, min_child_weight=1, missing=None, n_estimators=100,
             n_jobs=1, nthread=None, objective='reg:linear', random_state=0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
             silent=None, subsample=1, verbosity=1)
```

Figure 51: XGBoost Regression Model

The tuned XGBoost regression model gives the following results:



```
XGBoost Regression: Accuracy on training Data: 0.993
XGBoost Regression: Accuracy on test Data: 0.988

XGBoost Regression: The RMSE of the training set is:  0.09961280401594837
XGBoost Regression: The RMSE of the testing set is:  0.13387205642051989
```

Figure 52: XGBoost Regression Model Results

Upon tuning the hyperparameter, the model performance increased, and the resulting model performance is 98.8%.

3.6.8. Bagging Regression

Bagging Regressor is an ensemble estimator which fits base estimator on each random subset of the Train dataset and then aggregates their individual predictions to form a final prediction using voting or averaging method. Here the base estimator is Decision Trees.

On training the Bagging Regression model on the dataset without tweaking the parameters, the obtained training & test data accuracies are 99.8% & 98.3% respectively. This is the case of overfitting. So, parameter tuning is performed on the n_estimators.

Multiple values of n_estimators are checked by keeping other parameters to default values. The resulting model is as follows:

```
BaggingRegressor(base_estimator=None, bootstrap=True, bootstrap_features=False,
                 max_features=1.0, max_samples=1.0, n_estimators=1, n_jobs=None,
                 oob_score=False, random_state=None, verbose=0,
                 warm_start=False)
```

Figure 53: Bagging Regression Model

The tuned Bagging regression model gives the following results:

```
Bagging Regression: Accuracy on training Data: 0.994
Bagging Regression: Accuracy on test Data: 0.982

Bagging Regression: The RMSE of the training set is:  0.09617435482332859
Bagging Regression: The RMSE of the testing set is:  0.16567700949308659
```

Figure 54: Bagging Regression Model Results

Evaluating training and testing set performance with different numbers n_estimators from 1 to 30. The plot shows the training and test set accuracy on the y-axis against the setting of n_estimators on the x-axis:

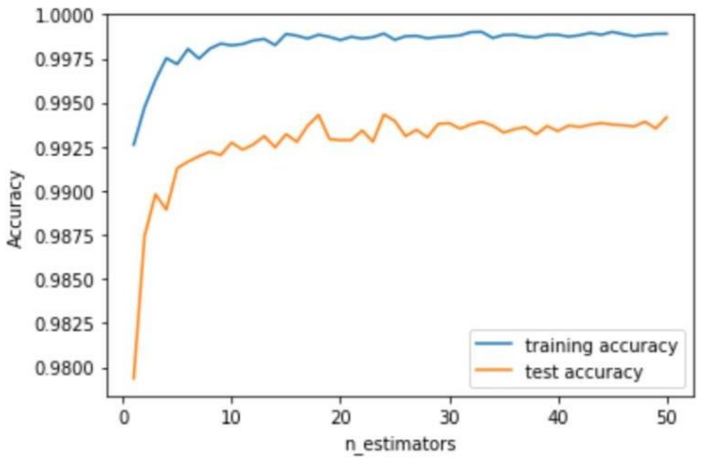


Figure 55: Bagging Regression Model Results for multiple n_estimators

From the above plot, it is clear that the model performs very well on this dataset. Even with tuning of n_estimators parameters, the training accuracy always stayed above 99% & the test data accuracy is always above 97%. This may or may not be the case of overfitting.

3.6.9. Custom Ensemble

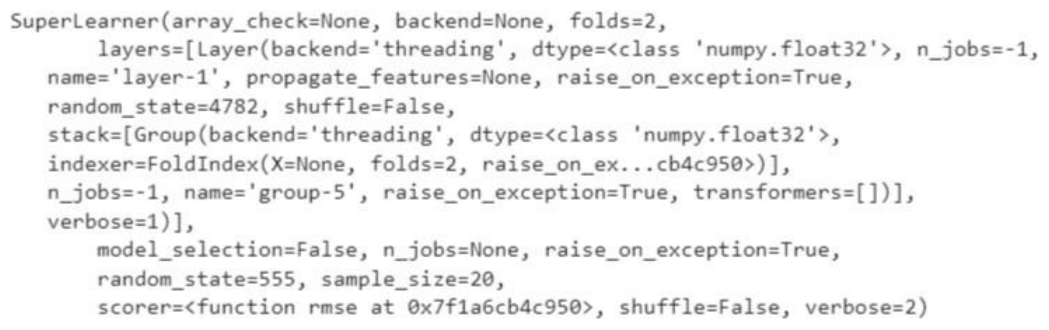
To build a custom ensemble, a Python library called mlens is used. mlens is short of ML-Ensemble used for memory efficient parallelized ensemble learning. ML-Ensemble is a library for building Scikit-learn compatible ensemble estimator. Ensembles are built as a feed-forward network, with a set of layers stacked on each other.

Layers are stacked sequentially with each layer taking the previous layer's output as input. Ensembles are built by adding layers to an instance object: layers in their turn are comprised of a list of estimators. Because base learners in an ensemble are independent of each other, ensembles benefit greatly from parallel processing. ML-Ensemble is designed to maximize parallelization at minimum memory footprint.

The custom ensemble model was build using the three least accuracy models that we saw earlier. In this case, the ensemble consists of MLP Regression, KNN Regression and Linear Regression models. The ensemble is off three layers:

- Layer 1: MLP Regression
- Layer 2: KNN Regression
- Layer 3: Linear Regression (Final layer)

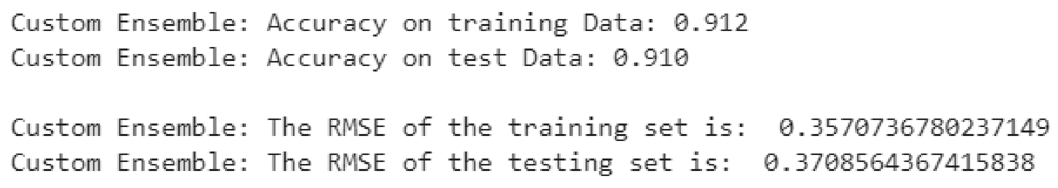
The ensemble model is shown below:



```
SuperLearner(array_check=None, backend=None, folds=2,
       layers=[Layer(backend='threading', dtype=<class 'numpy.float32'>, n_jobs=-1,
   name='layer-1', propagate_features=None, raise_on_exception=True,
   random_state=4782, shuffle=False,
   stack=[Group(backend='threading', dtype=<class 'numpy.float32'>,
   indexer=FoldIndex(X=None, folds=2, raise_on_ex...cb4c950>)],
   n_jobs=-1, name='group-5', raise_on_exception=True, transformers=[])],
   verbose=1)],
       model_selection=False, n_jobs=None, raise_on_exception=True,
       random_state=555, sample_size=20,
       scorer=<function rmse at 0x7f1a6cb4c950>, shuffle=False, verbose=2)
```

Figure 56: Custom Ensemble Model with SuperLearner

As usual, after building the model, the train data is fit to the model and the target values are predicted. Later the model is evaluated, and the results are as follows:



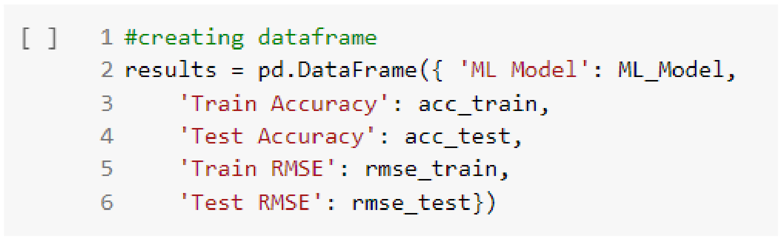
```
Custom Ensemble: Accuracy on training Data: 0.912
Custom Ensemble: Accuracy on test Data: 0.910

Custom Ensemble: The RMSE of the training set is:  0.3570736780237149
Custom Ensemble: The RMSE of the testing set is:  0.3708564367415838
```

Figure 57: Custom Ensemble Model Results

The above results show that the ensemble model performance is much better than linear regression model. In a way this model is an improved model of the models used in this.

4. MODEL RESULTS COMPARISON

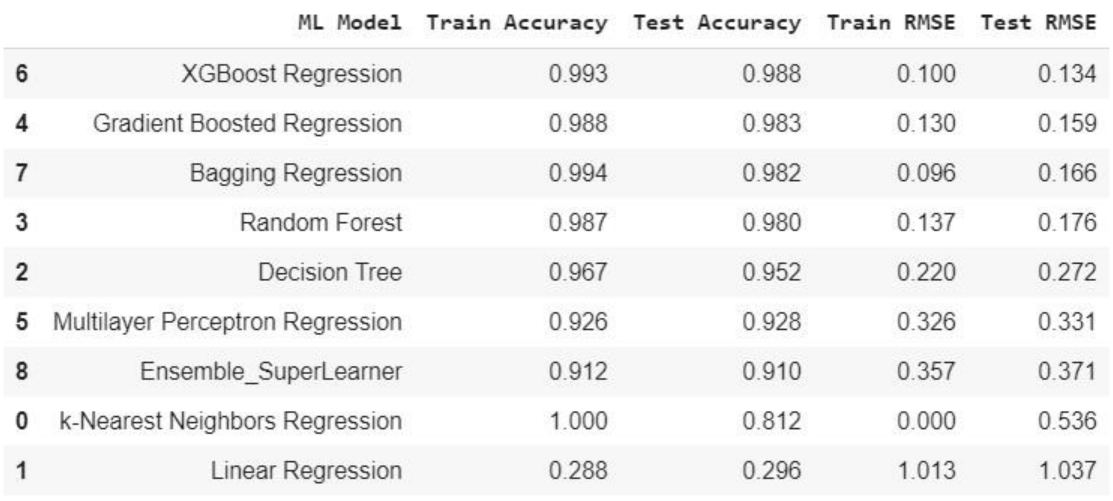
The results from the models are stored in four different lists which are created before stating the model training. These lists are gathered together to form a dataframe and the code snippet is as follows:



```
[ ]    #creating dataframe
       results = pd.DataFrame({ 'ML Model': ML_Model,
           'Train Accuracy': acc_train,
           'Test Accuracy': acc_test,
           'Train RMSE': rmse_train,
           'Test RMSE': rmse_test})
```

Figure 58: Custom Ensemble Model Results

The resulting dataframe of the above execution in sorted order is as follows:



	ML Model	Train Accuracy	Test Accuracy	Train RMSE	Test RMSE
6	XGBoost Regression	0.993	0.988	0.100	0.134
4	Gradient Boosted Regression	0.988	0.983	0.130	0.159
7	Bagging Regression	0.994	0.982	0.096	0.166
3	Random Forest	0.987	0.980	0.137	0.176
2	Decision Tree	0.967	0.952	0.220	0.272
5	Multilayer Perceptron Regression	0.926	0.928	0.326	0.331
8	Ensemble_SuperLearner	0.912	0.910	0.357	0.371
0	k-Nearest Neighbors Regression	1.000	0.812	0.000	0.536
1	Linear Regression	0.288	0.296	1.013	1.037

Figure 59: Models Performance Results (descending order)

Among all the trained modesl, XGBoost performance is better. It is understandable because this model is very good in execution Speed & model performance.

5. STATISTICAL TESTS

Statistical tests are used in hypothesis testing. They can be used to:

- determine whether a predictor variable has a statistically significant relationship with an outcome variable.
- estimate the difference between two or more groups.

Statistical tests assume a null hypothesis of no relationship or no difference between groups. Then they determine whether the observed data fall outside of the range of values predicted by the null hypothesis. The following are the test on this dataset:

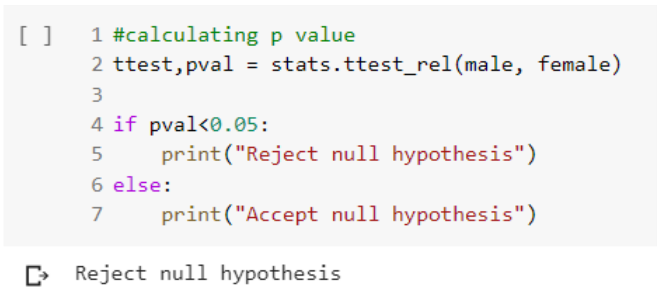
5.1. T-Test: Male & Female Suicide Rates

Test 1: To check the difference in suicide rates between male and female

Using independent sample t-test to check the difference in suicide rates between male and female. The hypothesis statements for this test are:

H0: There is no difference in the suicide rates among male and female (Null).
H1: There is difference in the suicide rates among male and female (Alternate).

Initially, 'stats' module is imported from 'SciPy' library. Then the library module, *'ttest_rel()'* is used to calculate t-statistic and pvalue. If the obtained pvalue is less than significance level (0.05) then, we can reject the null hypothesis & consider the alternate hypothesis. The code snippet for this calculation is as follows:



```
[ ]  #calculating p value
     ttest,pval = stats.ttest_rel(male, female)

     if pval<0.05:
         print("Reject null hypothesis")
     else:
         print("Accept null hypothesis")
```

```
Reject null hypothesis
```

Figure 60: Models Performance Results (descending order)

The obtained T-test result is to reject the null hypothesis. The final conclusion of this test is that there is different in suicide rates of male & female.

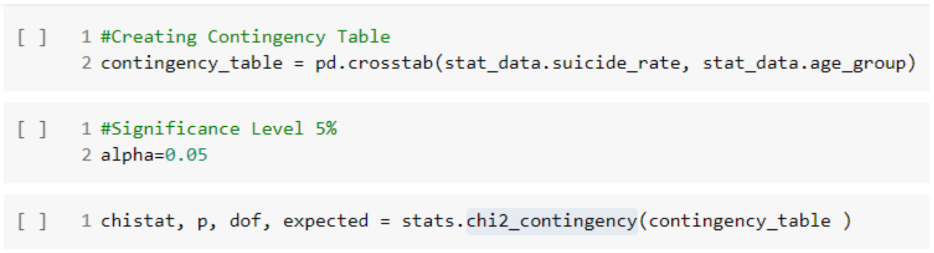
5.2. Chi-Squared Test: Age & Suicide Rate

Test 2: To find out the dependence of suicide rate on the age.

Finding out whether there is a dependence of suicide rate on the age using the Chi- Square test. The hypothesis statements for this test are:

H_0: Suicide rate and age are independent (Null).
H_1: Suicide rate and age are dependent (Alternate).

Initially, 'stats' module is imported from 'SciPy' library. The contingency table should be created for the testing features by using *'crosstab()'* module of Pandas. The significane level is set to 0.05. Then from the library, *'chi2_contingency()'* module is used to calculate chi test-statistic, pvalue, degrees of freedom and expected frequencies. The following shows the code for this task:



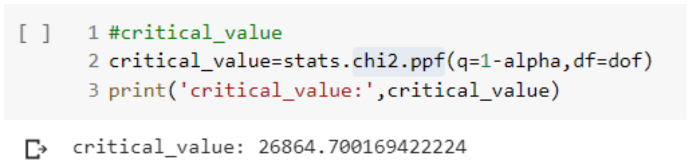
```
[ ]  #Creating Contingency Table
     contingency_table = pd.crosstab(stat_data.suicide_rate, stat_data.age_group)
```

```
[ ]  #Significance Level 5%
     alpha=0.05
```

```
[ ]  chistat, p, dof, expected = stats.chi2_contingency(contingency_table )
```

Figure 61: Chi-Square Test

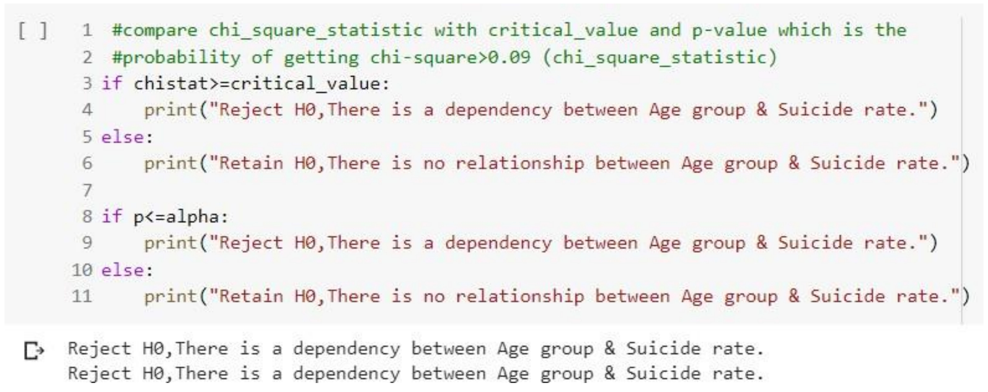
Then critical value for the test is calculated by using the module *'chi2.ppf()'*. The obtained critical value is shown below:



```
[ ]  #critical_value
     critical_value=stats.chi2.ppf(q=1-alpha,df=dof)
     print('critical_value:',critical_value)

critical_value: 26864.700169422224
```

Figure 62: Critical value for Chi-Square Test

The final step is to compare the calculated values as shown below:



```
[ ]  #compare chi_square_statistic with critical_value and p-value which is the
     #probability of getting chi-square>0.09 (chi_square_statistic)
     if chistat>=critical_value:
         print("Reject H0,There is a dependency between Age group & Suicide rate.")
     else:
         print("Retain H0,There is no relationship between Age group & Suicide rate.")

     if p<=alpha:
         print("Reject H0,There is a dependency between Age group & Suicide rate.")
     else:
         print("Retain H0,There is no relationship between Age group & Suicide rate.")

Reject H0,There is a dependency between Age group & Suicide rate.
Reject H0,There is a dependency between Age group & Suicide rate.
```

Figure 63: Chi-Square Test Result

The obtained result of the Chi- Square test is the is to reject the null hypothesis. The final conclusion of this test is dependency between Age group & Suicide rate. That means the suicide rate varies with the age group.

6. SUMMARY OF LEARNINGS

The final take away from this project is the working of different machine learning models on a dataset and understanding their parameters. Creating this notebook helped me to learn a lot about the parameters of the models, how to tune them and how they affect the model performance. The final conclusion on the suicide dataset are that the irrespective of age group and generation, male population are more prone to commit suicide than female.

7. REFERENCES

[1] Introduction to Machine Learning with Python by Andreas C. Müller and Sarah Guido

[2] https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/ - 2016

[3] https://www.ritchieng.com/machine-learning-efficiently-search-tuning-param/ - 2017

[4] https://machinelearningmastery.com/xgboost-python-mini-course/ - 2016

[5] https://www.datavedas.com/regression-problems-in-python/ - 2020

[6] https://machinelearningmastery.com/super-learner-ensemble-in-python/ - 2017

[7] https://www.scribbr.com/statistics/statistical-tests/

[8] https://medium.com/@nhan.tran/the-chi-square-statistic-p-1-37a8eb2f27bb