Assignment_18BCS6033_CO2_Emission

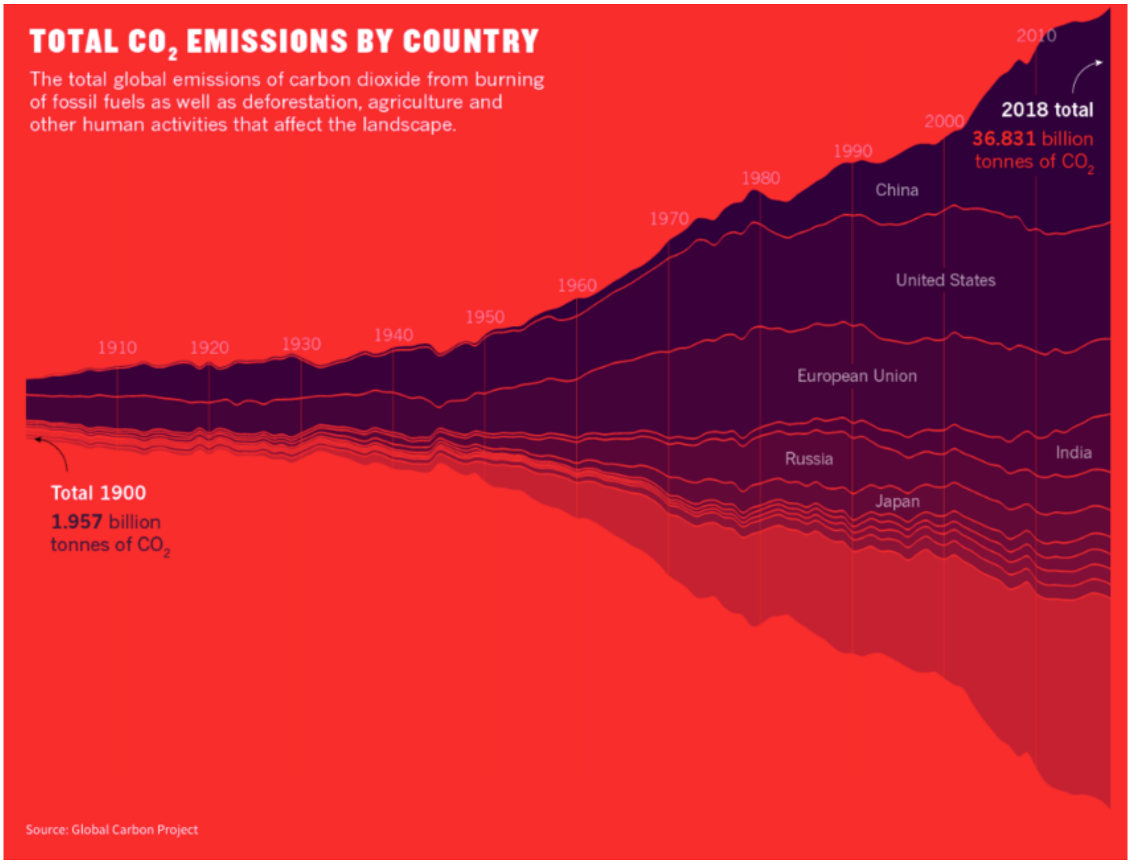
September 12, 2020

1 Assignment on detecting which macroeconomic factors are the biggest predictors of Global Warming, i.e. Global CO2 Emissions.

1.1 Karan Trehan

1.1.1 18BCS6033

1.1.2 18AITAIML1 - Group B




```
[1]: # hide warnings
import warnings
warnings.filterwarnings('ignore')
```

2 Importing the Required Libraries

[2]:
```
#For Data Handling
import pandas as pd
import numpy as np

#For Statistical Calculations
import scipy.stats as st

#For Regression and Feature Selection
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import Lasso
from sklearn.model_selection import GridSearchCV

#For Data Visualization/Exploratory Analysis
from matplotlib import pyplot as plt
import seaborn as sns
```

3 Reading and Understanding the Data

Let's start with the following steps:

1. Importing data using the pandas library
2. Understanding the structure of the data

[3]:
```
#Reading all the Dataframes and storig them into respective Variable Names.

cars_trucks_buses = pd.read_csv('.\\co2_prediction␣
↪dataset\\co2_prediction\\cars_trucks_and_buses_per_1000_persons.csv')
co2_emissions = pd.read_csv('.\\co2_prediction␣
↪dataset\\co2_prediction\\co2_emissions_tonnes_per_person.csv')
coal_consumption = pd.read_csv('.\\co2_prediction␣
↪dataset\\co2_prediction\\coal_consumption_per_cap.csv')
elec_gen = pd.read_csv('.\\co2_prediction␣
↪dataset\\co2_prediction\\electricity_generation_per_person.csv')
elec_use = pd.read_csv('.\\co2_prediction␣
↪dataset\\co2_prediction\\electricity_use_per_person.csv')
forest_coverage = pd.read_csv('.\\co2_prediction␣
↪dataset\\co2_prediction\\forest_coverage_percent.csv')
hydro_power_gen = pd.read_csv('.\\co2_prediction␣
↪dataset\\co2_prediction\\hydro_power_generation_per_person.csv')
income_pp = pd.read_csv('.\\co2_prediction␣
↪dataset\\co2_prediction\\income_per_person_gdppercapita_ppp_inflation_adjusted.
↪csv')
```

```
industry_gdp = pd.read_csv('.\\co2_prediction␣
 ↪dataset\\co2_prediction\\industry_percent_of_gdp.csv')
natgas_prod = pd.read_csv('.\\co2_prediction␣
 ↪dataset\\co2_prediction\\natural_gas_production_per_person.csv')
oil_consum = pd.read_csv('.\\co2_prediction␣
 ↪dataset\\co2_prediction\\oil_consumption_per_cap.csv')
oil_prod = pd.read_csv('.\\co2_prediction␣
 ↪dataset\\co2_prediction\\oil_production_per_person.csv')
yearly_co2_emission = pd.read_csv('.\\co2_prediction␣
 ↪dataset\\co2_prediction\\yearly_co2_emissions_1000_tonnes.csv')
```

[4]:
```
#Defining a Dictionary for all the Dataframes, as it will be easy to access them␣
 ↪when they will be used to combine into one
#Main Dataframe

columns = {'cars_trucks_buses' : cars_trucks_buses ,
              'co2_emissions' : co2_emissions,
              'coal_consumption' : coal_consumption,
              'elec_gen' : elec_gen,
              'elec_use' : elec_use,
              'forest_coverage' : forest_coverage,
              'hydro_power_gen' : hydro_power_gen,
              'income_pp' : income_pp,
              'industry_gdp' : industry_gdp,
              'natgas_prod' : natgas_prod,
              'oil_consum' : oil_consum,
              'oil_prod' : oil_prod,
              'yearly_co2_emission' : yearly_co2_emission
          }
```

[5]:
```
#Printing the shape of every Dataframe to check which one has the highest number␣
 ↪of row count,
#in other words, having data of max number of countries.

for i in columns.keys():
    print(i , ' = ' , columns.get(i).shape)
```

```
cars_trucks_buses  =  (157, 7)
co2_emissions  =  (192, 216)
coal_consumption  =  (65, 53)
elec_gen  =  (65, 33)
elec_use  =  (138, 56)
forest_coverage  =  (192, 27)
hydro_power_gen  =  (118, 53)
income_pp  =  (193, 220)
industry_gdp  =  (189, 59)
natgas_prod  =  (49, 48)
```

```
oil_consum  =  (65, 53)
oil_prod  =  (49, 53)
yearly_co2_emission  =  (192, 265)
```

- income_pp has the highest number of rows implying that it contains data of almost all the countries.

[6]:
```
#Creating the main DataFrame 'df' and inserting the first column as 'country' of␣
↪the Series "income_pp['geo']"
df = pd.DataFrame(columns.get('income_pp')['geo'])

for i in columns.keys() :
    if '2014' in columns.get(i).columns :
        df =pd.merge(df , columns.get(i)[['geo','2014']] , how='outer' ,␣
↪on='geo')
    else:
        print(i)
```

```
cars_trucks_buses
hydro_power_gen
```

[7]:
```
#Creating a list of new Column Names using the dictionary `columns`
new_columns = ['country']
new_columns.extend(list(columns.keys()))
new_columns.remove('cars_trucks_buses')
new_columns.remove('hydro_power_gen')

#Renaming the columns of the main dataframe
df.columns = new_columns
```

[8]:
```
df = df.sort_values('country')
df = df.reset_index(drop=True)
```

[9]:
```
df.head()
```

[9]:
```
       country  co2_emissions  coal_consumption  elec_gen  elec_use  \
0  Afghanistan          0.299               NaN       NaN       NaN
1      Albania          1.960               NaN       NaN    2310.0
2      Algeria          3.720           0.00458    1640.0    1360.0
3      Andorra          5.830               NaN       NaN       NaN
4       Angola          1.290               NaN       NaN     312.0

   forest_coverage  income_pp  industry_gdp  natgas_prod  oil_consum  \
0             2.07     1780.0         21.10          NaN         NaN
1            28.20    10700.0         21.50          NaN         NaN
2             0.82    13500.0         42.30         1.92       0.452
3            34.00    44900.0          9.91          NaN         NaN
4            46.50     6260.0           NaN          NaN         NaN
```

```
   oil_prod  yearly_co2_emission
0       NaN               9810.0
1       NaN               5720.0
2      1.76             145000.0
3       NaN                462.0
4      3.08              34800.0
```

[10]:
```
df.tail()
```

[10]:
```
       country  co2_emissions  coal_consumption  elec_gen  elec_use  \
189  Venezuela          6.030           0.00641    3590.0    2660.0
190    Vietnam          1.800           0.20500    1540.0    1410.0
191      Yemen          0.865               NaN       NaN     216.0
192     Zambia          0.288               NaN       NaN     707.0
193   Zimbabwe          0.780               NaN       NaN     537.0

     forest_coverage  income_pp  industry_gdp  natgas_prod  oil_consum  \
189            53.10    16700.0          37.2       0.8390       1.090
190            47.20     5370.0          33.2       0.0993       0.195
191             1.04     3770.0          44.0       0.3200         NaN
192            65.70     3630.0          32.9          NaN         NaN
193            37.20     1910.0          22.5          NaN         NaN

     oil_prod  yearly_co2_emission
189     4.510             185000.0
190     0.195             167000.0
191     0.256              22700.0
192       NaN               4500.0
193       NaN              12000.0
```

[11]:
```
#Checking information about the Dataset
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 194 entries, 0 to 193
Data columns (total 12 columns):
country                 194 non-null object
co2_emissions           192 non-null float64
coal_consumption        65 non-null float64
elec_gen                65 non-null float64
elec_use                137 non-null float64
forest_coverage         191 non-null float64
income_pp               193 non-null float64
industry_gdp            183 non-null float64
natgas_prod             49 non-null float64
oil_consum              65 non-null float64
```

```
oil_prod               49 non-null float64
yearly_co2_emission    192 non-null float64
dtypes: float64(11), object(1)
memory usage: 18.3+ KB
```

4 Data Preparation and Pre-processing

[12]:
```
df['country'].is_unique
```

[12]: True

We can observe that the `country` column has Unique Values and it will be redundant to create Dummy Variables for it (it will increase the complexity of the Model), so we will drop the column.

[13]:
```
#Dropping the 'country' column
df.drop(columns=['country'],axis=1,inplace=True)
```

[14]:
```
#Viewing the Statistical Measures/Details of the Dataset
df.describe()
```

[14]:

	co2_emissions	coal_consumption	elec_gen	elec_use
count	192.000000	65.00000	65.000000	137.000000
mean	4.440085	0.44212	6188.215385	4253.621898
std	6.065368	0.53450	5046.927099	6024.002485
min	0.044500	0.00000	350.000000	39.000000
25%	0.659000	0.04000	2890.000000	812.000000
50%	2.265000	0.25000	4750.000000	2580.000000
75%	5.695000	0.59400	8110.000000	5360.000000
max	45.400000	2.34000	27600.000000	53800.000000

	forest_coverage	income_pp	industry_gdp	natgas_prod	oil_consum
count	191.000000	193.000000	183.000000	49.000000	65.000000
mean	31.907068	17210.398964	26.761093	4.421976	1.410500
std	23.783266	18911.747174	13.365268	10.830251	1.756355
min	0.000000	602.000000	2.530000	0.021200	0.036100
25%	11.000000	3270.000000	18.600000	0.230000	0.495000
50%	32.000000	10800.000000	24.700000	0.742000	1.060000
75%	47.650000	24000.000000	31.400000	2.990000	1.490000
max	98.300000	121000.000000	70.500000	66.000000	12.100000

	oil_prod	yearly_co2_emission
count	49.000000	1.920000e+02
mean	4.747659	1.759925e+05
std	8.236582	8.607430e+05
min	0.032200	1.100000e+01
25%	0.301000	2.190000e+03
50%	1.440000	1.130000e+04

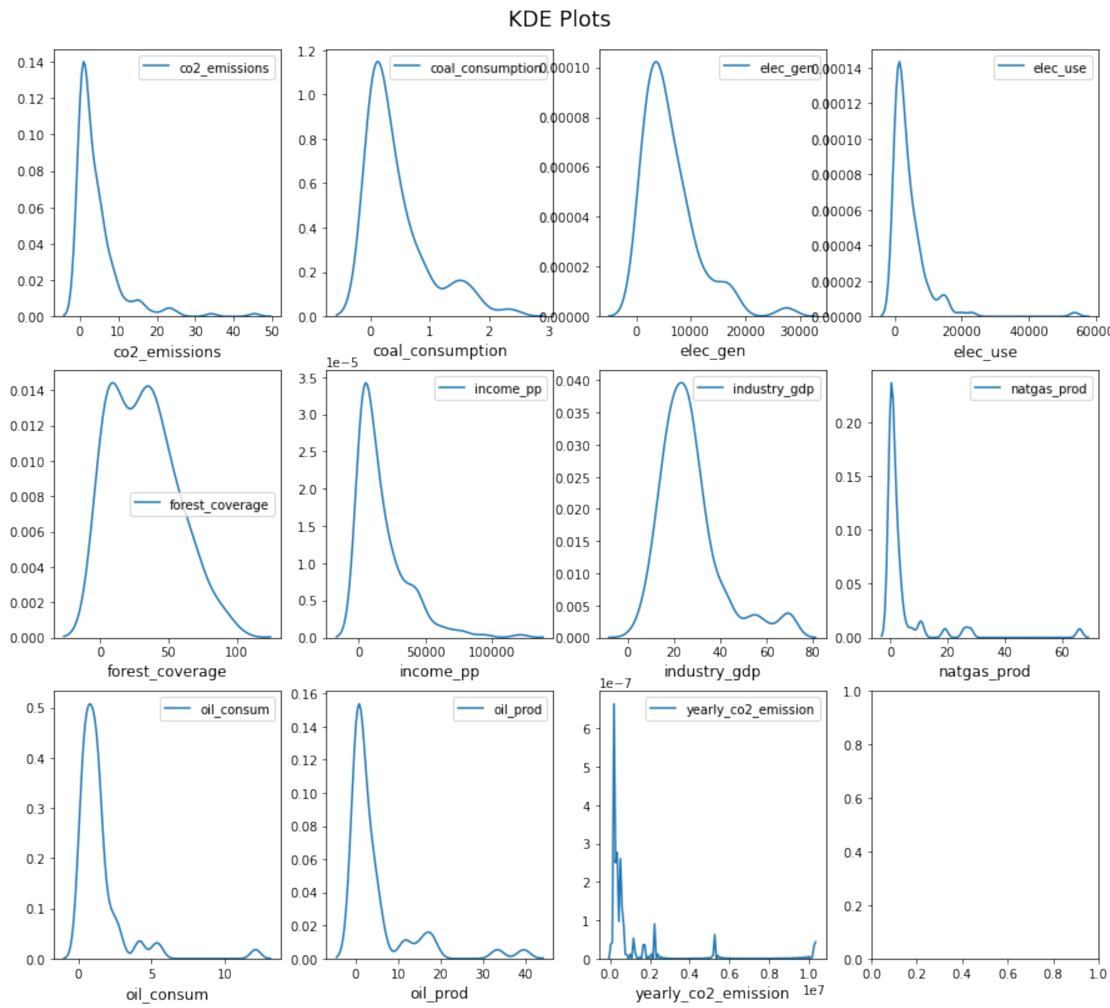
```
75%      4.510000          6.377500e+04
max     39.700000          1.030000e+07
```

5 Data Visualization

5.1 Univariate Analysis

[15]:
```
#Defining a function 'kdePlot()' which can be used for plotting the kde plots␣
↪for all columns of the Dataframe
#passed as an argument

def kdePlot(df,rows,cols,Title):
    # 'n' will store the number of Columns
    n = df.shape[1]

    #Creating subplots
    fig, axs = plt.subplots(rows,cols, figsize = (15, 15))
    fig.subplots_adjust(top=0.8)

    #Looping through the DataFrame and plotting for each Column
    k=0
    j=0
    for i, var in enumerate(df.columns.values):

        if (j%cols==0 and j!=0):
            k+=1
        if (j%8==0 and j!=0):
            j=0

        sns.kdeplot(df[var],ax=axs[k, i-int(k*cols)])
        axs[k, i-int(k*cols)].set_xlabel(var, fontsize = 'large')
        j+=1

    #Providing Tiltle for the Plot
    plt.suptitle(Title, fontsize = 'xx-large',y=0.83)
    plt.show()
```

[16]:
```
#Plotting KDE Plots
kdePlot(df ,rows = 3,cols = 4,Title = "KDE Plots")
```



Data is Skewed

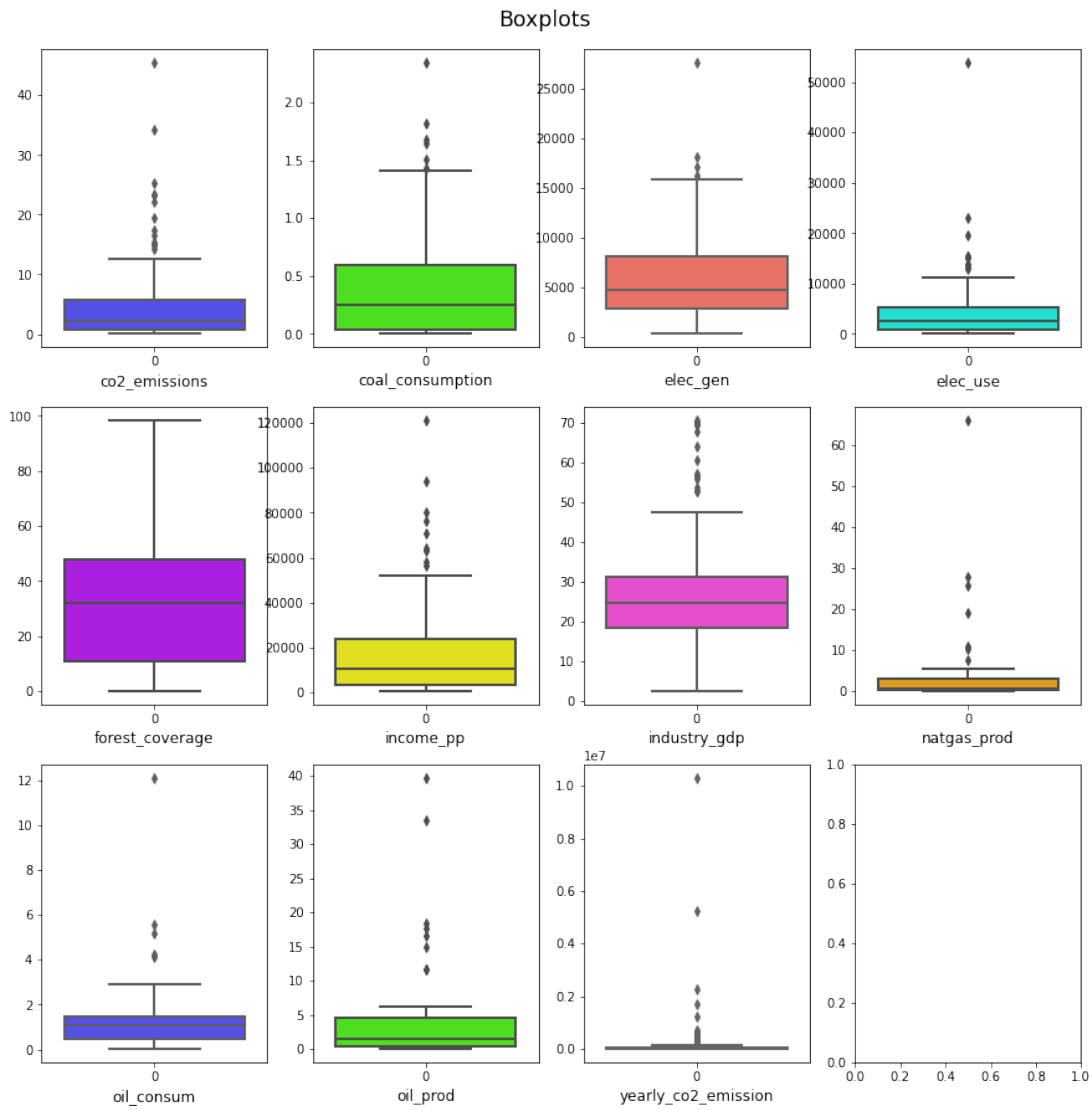
[17]:
```python
#Defining a function 'distributionPlot()' which can be used for plotting the␣
↪distribution plots for all columns of the Dataframe
#passed as an argument

def distributionPlot(df,rows,cols,Title):
    # 'n' will store the number of Columns
    n = df.shape[1]

    #Creating subplots
    fig, axs = plt.subplots(rows,cols, figsize = (15, 8))
    fig.subplots_adjust(top=0.8)

    #Defining the color schemes
```

```
    colors = ['#3E37FF', '#3BFF00', '#FF6050', '#00FFEA', '#BA00FF', '#FFFE00',␣
↪'#FF36DD','orange'] # to set color

    #Looping through the DataFrame and plotting for each Column
    k=0
    j=0
    for i, var in enumerate(df.columns.values):

        if (j%cols==0 and j!=0):
            k+=1
        if (j%8==0 and j!=0):
            j=0

        sns.distplot(df[var],ax=axs[k,␣
↪i-int(k*cols)],color=colors[j],kde_kws=dict(linewidth=cols),hist=True)
        axs[k, i-int(k*cols)].set_xlabel(var, fontsize = 'large')
        j+=1

    #Providing Tiltle for the Plot
    plt.suptitle(Title, fontsize = 'xx-large',y=0.85)
    plt.show()
```

[18]:
```
#Defining a function 'boxPlot()' which can be used for plotting the box-plots␣
↪for all columns of the Dataframe passed as
#an argument
def boxPlot(df,rows,cols,Title):
    # 'n' will store the number of Columns
    n = df.shape[1]

    #Creating subplots
    fig, axs = plt.subplots(rows,cols, figsize = (15, 15))

    #Defining the color schemes
    colors = ['#3E37FF', '#3BFF00', '#FF6050', '#00FFEA', '#BA00FF', '#FFFE00',␣
↪'#FF36DD','orange'] # to set color

    #Looping through the DataFrame and plotting for each Column
    k=0
    j=0
    for i, var in enumerate(df.columns.values):
        if (j%cols==0 and j!=0):
            k+=1
        if (j%8==0 and j!=0):
            j=0

        sns.boxplot(data = df[var] , ax = axs[k, i-int(k*cols)], color=␣
↪colors[j],linewidth=2)
```

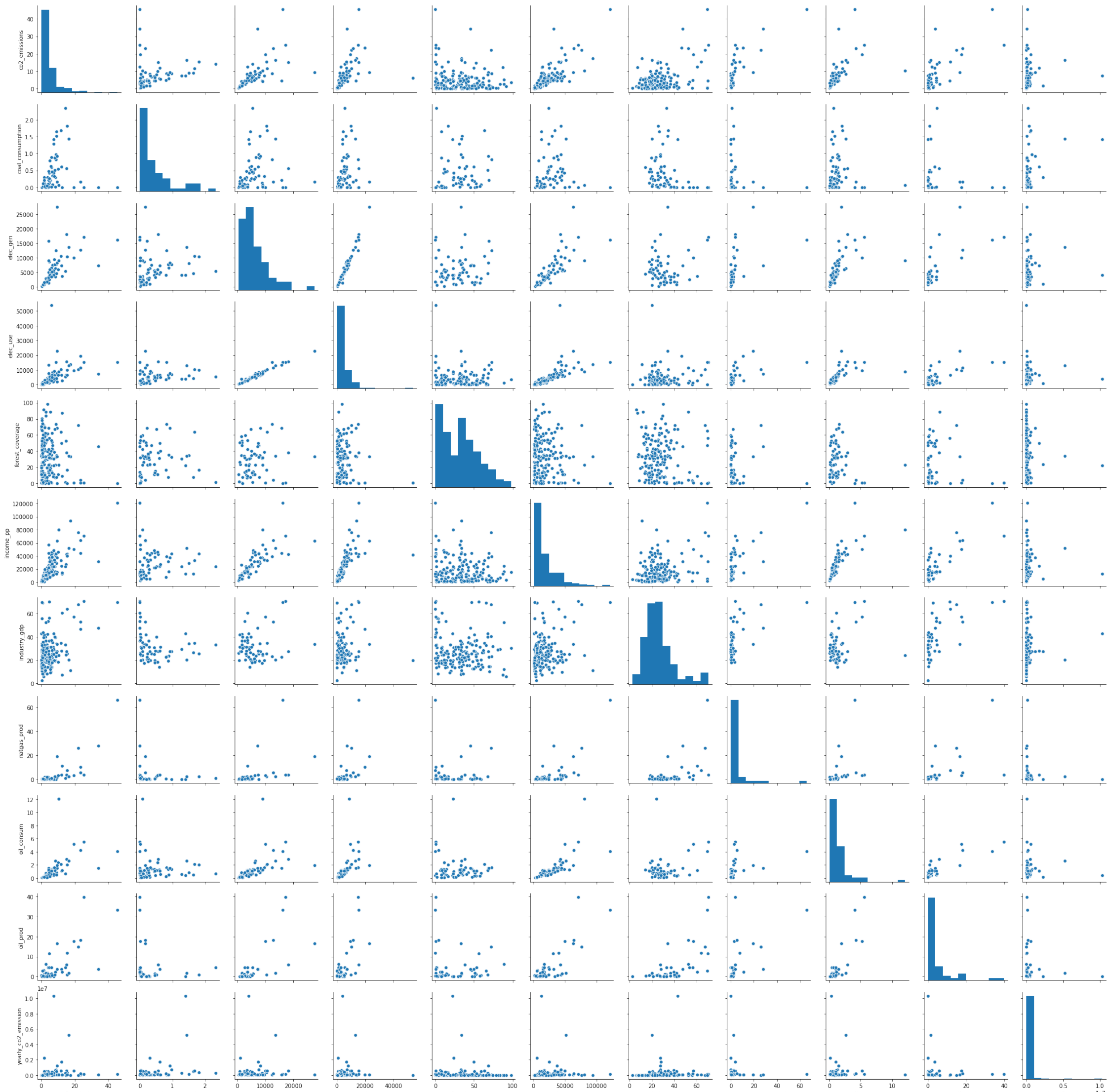
```
        axs[k, i-int(k*cols)].set_xlabel(var, fontsize = 'large')
        j+=1

    #Providing Tiltle for the Plot
    plt.suptitle(Title, fontsize = 'xx-large',y=0.91)
    plt.show()
```

[19]:
```
#Plotting Boxplots Plots
boxPlot(df ,rows = 3,cols = 4,Title = "Boxplots")
```



5.2 Bi-Variate Analysis

[20]:
```
sns.pairplot(df)
plt.show()
```

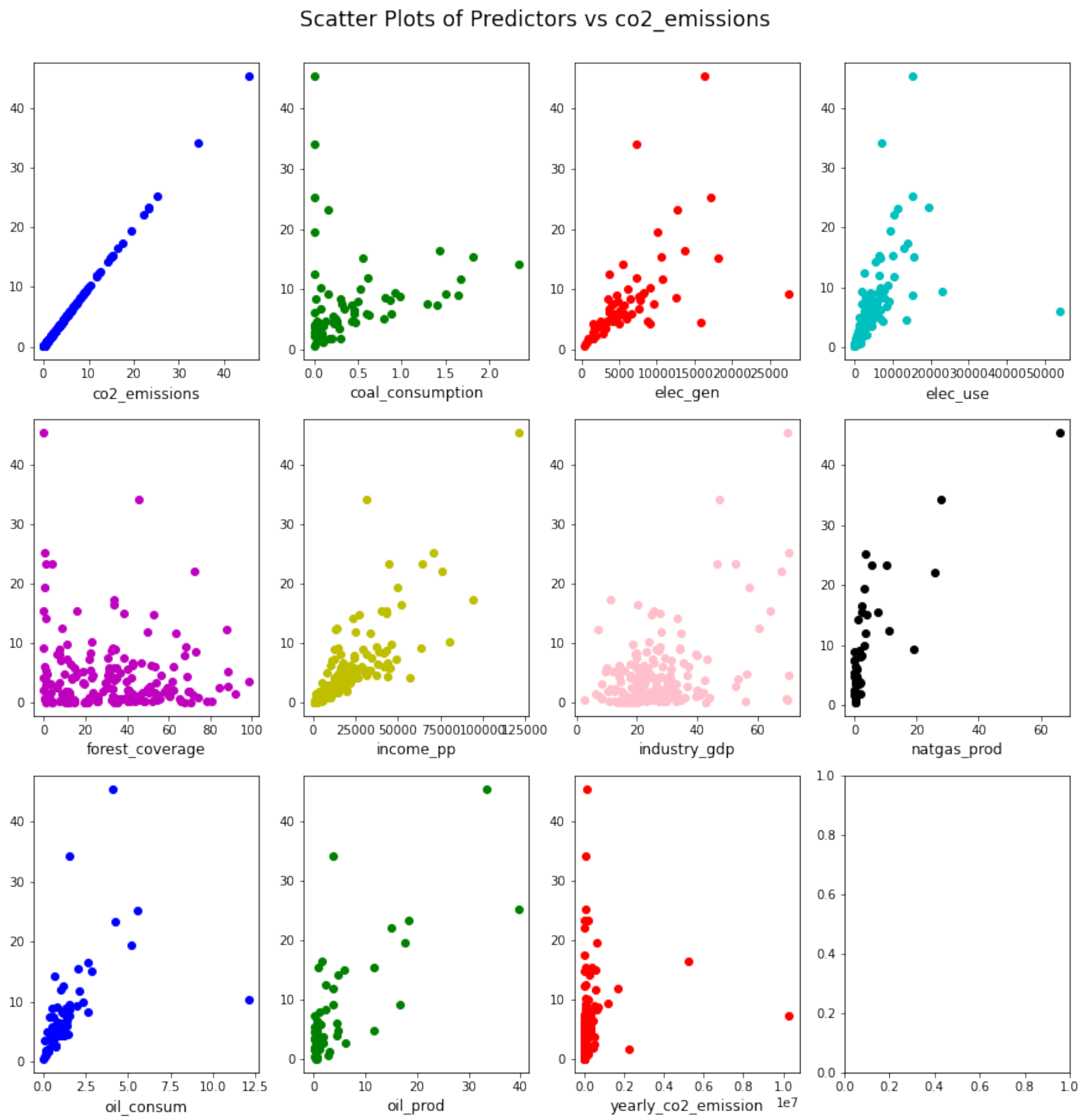


Since Much information cannot be obtained from the Pairplot, we'll plot a Correlation Heat map for a better understanding

[21]:
```
#Defining a function 'scatterPlot()' which can be used for plotting the
↪scatter-plots for all columns v/s 'co2_emissions'
#(dependent Variable) of the Dataframe passed as an argument

def scatterPlot(df,rows,cols,Title):
    # 'n' will store the number of Columns
```

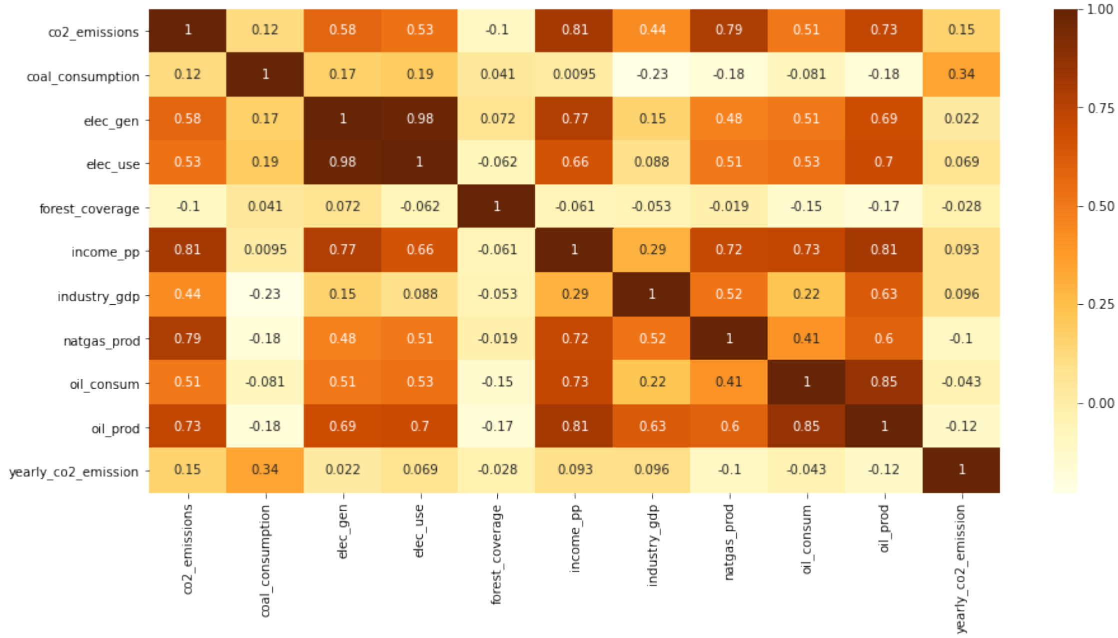
```
    n = df.shape[1]

    #Creating subplots
    fig, axs = plt.subplots(rows,cols, figsize = (15, 15))

    #Defining the color schemes
    colors = ['b', 'g', 'r', 'c', 'm', 'y', 'pink','k'] # to set color

    #Looping through the DataFrame and plotting for each Column
    k=0
    j=0
    for i, var in enumerate(df.columns.values):
        if (j%cols==0 and j!=0):
            k+=1
        if (j%8==0 and j!=0):
            j=0

        axs[k, i-int(k*cols)].scatter(df[var], df["co2_emissions"], color =␣
 ↪colors[j])
        axs[k, i-int(k*cols)].set_xlabel(var, fontsize = 'large')
        j+=1

    #Providing Tiltle for the Plot
    plt.suptitle(Title, fontsize = 'xx-large',y=0.92)
    plt.show()
```

[22]:
```
#Plotting Scatter Plots v/s 'co2_emissions' (dependent Variable)
scatterPlot(df,3,4,'Scatter Plots of Predictors vs co2_emissions')
```



[23]:

```
#Plotting the Correlation Map to check correlation between the columns since␣
 ↪pairplot is a bit difficult to interpret
fig, axs = plt.subplots(figsize = (15,7))
sns.heatmap(df.corr(),cmap='YlOrBr',annot=True)
plt.show()
```



Correlation of `co2_emmisions` with independent variables: * `co2_emmisions` is highly correlated to `income_pp` , `natgas_prod` , `oil_prod`

Correlation among independent variables: * `elec_gen` and `elec_use` are highly correlated, depicts that the electricity generated and electricity used are quite dependent on each other. (`elec_gen` has more correlation with `co2_emissions` than `elec_use`) * `oil_consum` and `oil_prod` are highy correlated, depicts that the oil consumed and oil produced are also quite dependent on each other.

6 Checking for Missing and Duplicated Values

[24]:
```
#Checking for duplicacy in the DataFrame using '.duplicated()' method and then␣
↪checking the number of rows using
# '.shape[0]'
print("Number of Duplicate Rows in DataFrame:" , df[df.duplicated()].shape[0])
```

```
Number of Duplicate Rows in DataFrame: 0
```

[25]:
```
#Checking the Percentage of Columns having Missing Values in both the DataFrames
print('-+-'*10)
print(round(df.isnull().sum()/len(df)*100,2))
print('-+-'*10)
```

```
-+--+--+--+--+--+--+--+--+--+-
co2_emissions          1.03
coal_consumption      66.49
elec_gen              66.49
```

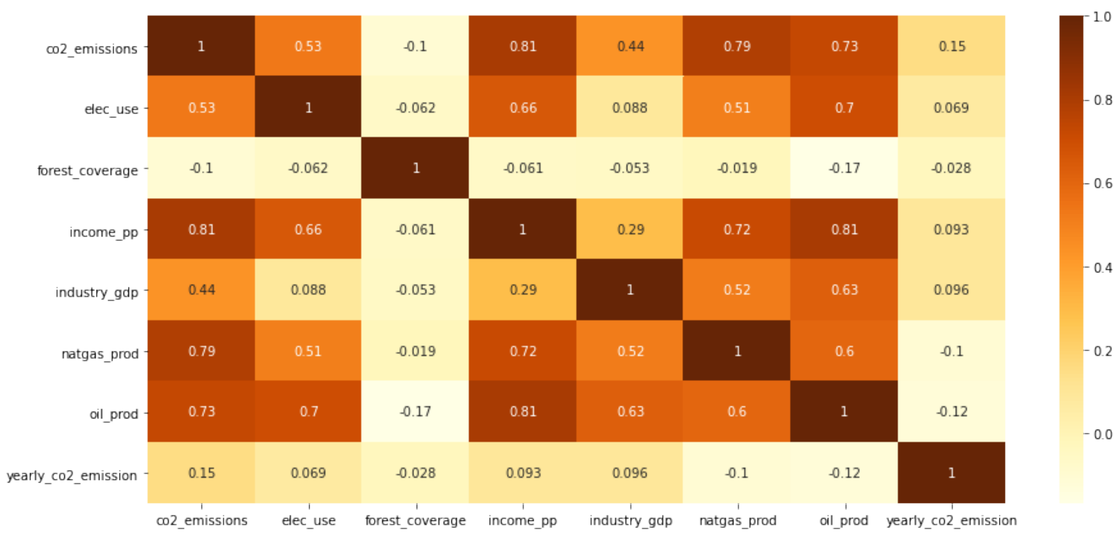
```
elec_use                29.38
forest_coverage          1.55
income_pp                0.52
industry_gdp             5.67
natgas_prod             74.74
oil_consum              66.49
oil_prod                74.74
yearly_co2_emission      1.03
dtype: float64
-+--+--+--+--+--+--+--+--+--+-
```

There are a lot of missing values in the Dataset.

[26]:
```
#Dropping Columns
df.drop(columns=['coal_consumption','elec_gen','oil_consum'],axis=1,inplace=True)
```

- Dropping Column `coal_consumption` because it has 66.49% missing values and is not correlated with the Target Variable.
- Dropping Columns `elec_gen` and `oil_consum` because they are highly correlated with `elec_use` and `oil_prod` and also both contain 66.49% missing values respectively.

[27]:
```
#Again Plotting the Correlation Map for further Analysis
fig, axs = plt.subplots(figsize = (15,7))
sns.heatmap(df.corr(),cmap='YlOrBr',annot=True)
plt.show()
```



- `natgas_prod` and `oil_prod` are highly correlated with `income_pp`.

[28]:
```
#Dropping Columns
df.drop(columns=['natgas_prod', 'oil_prod'],axis=1,inplace=True)
```

To prevent multicollinearity problem and also considering the 74.74% missing values in `natgas_prod` and `oil_prod`, they both are dropped.

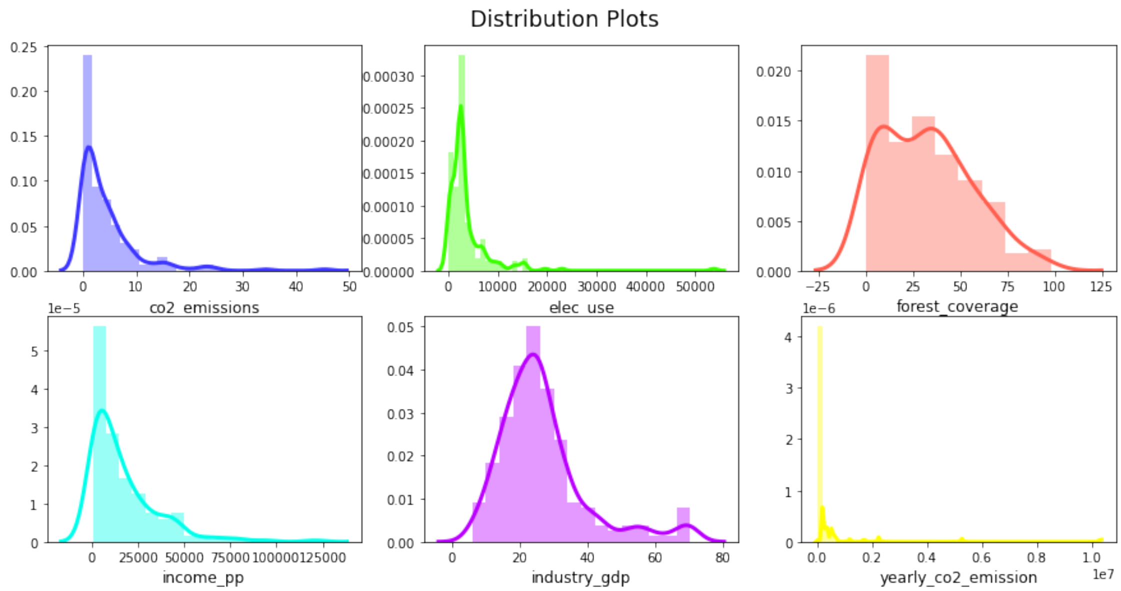
[29]:
```
#Dropping the Rows in which the features are having 0.52% to 1.55% misssing␣
↪values
df = df[df['co2_emissions'].notna()]
df = df[df['forest_coverage'].notna()]
df = df[df['income_pp'].notna()]
df = df[df['yearly_co2_emission'].notna()]
```

[30]:
```
#Imputing the Null Values by median for the rest of the cases of the respective␣
↪columns
df['elec_use'] = df['elec_use'].fillna(df['elec_use'].median())
df['industry_gdp'] = df['industry_gdp'].fillna(df['industry_gdp'].median())
```

[31]:
```
#Again hecking information about the Dataset
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 189 entries, 0 to 193
Data columns (total 6 columns):
co2_emissions          189 non-null float64
elec_use               189 non-null float64
forest_coverage        189 non-null float64
income_pp              189 non-null float64
industry_gdp           189 non-null float64
yearly_co2_emission    189 non-null float64
dtypes: float64(6)
memory usage: 10.3 KB
```

[32]:
```
#Plotting Distribution Plots after Missing Value Treatment
distributionPlot(df ,rows = 2,cols = 3,Title = "Distribution Plots")
```



7 Capping and Dropping the Outliers

[33]:
```
#Capping the Outliers
for x in df.columns:
    print("Capping The",x)
    ten = df[x].quantile(0.10)
    nin = df[x].quantile(0.90)
    df[x] = np.where(df[x] < ten, ten,df[x])
    df[x] = np.where(df[x] > nin, nin,df[x])
```

```
Capping The co2_emissions
Capping The elec_use
Capping The forest_coverage
Capping The income_pp
Capping The industry_gdp
Capping The yearly_co2_emission
```

[34]:
```
#Viewing the Statistical Measures/Details of the Dataset
df.describe()
```

[34]:

	co2_emissions	elec_use	forest_coverage	income_pp	\
count	189.000000	189.000000	189.000000	189.000000	
mean	3.607596	3118.738624	30.994148	15408.338624	
std	3.287092	2266.598876	21.375526	13799.018398	
min	0.219400	346.400000	1.836000	1644.000000	
25%	0.780000	1480.000000	11.000000	3270.000000	

```
50%          2.310000  2600.000000        32.000000  10800.000000
75%          5.830000  3970.000000        48.100000  24000.000000
max          9.988000  7732.000000        66.040000  43000.000000

       industry_gdp  yearly_co2_emission
count    189.000000           189.000000
mean      25.515132         63777.167196
std        8.770000         99805.487346
min       13.900000           433.400000
25%       18.800000          2350.000000
50%       24.700000         11600.000000
75%       30.900000         64600.000000
max       42.440000        305600.000000
```

[35]:

```
#Finding out the 25% and 75% quartile to further use them in calculating the␣
↪Interquartile Range.
q1 = df.quantile(0.25)
q3 = df.quantile(0.75)
IQR = q3 - q1
print(IQR)
```

```
co2_emissions              5.05
elec_use                2490.00
forest_coverage           37.10
income_pp              20730.00
industry_gdp              12.10
yearly_co2_emission    62250.00
dtype: float64
```

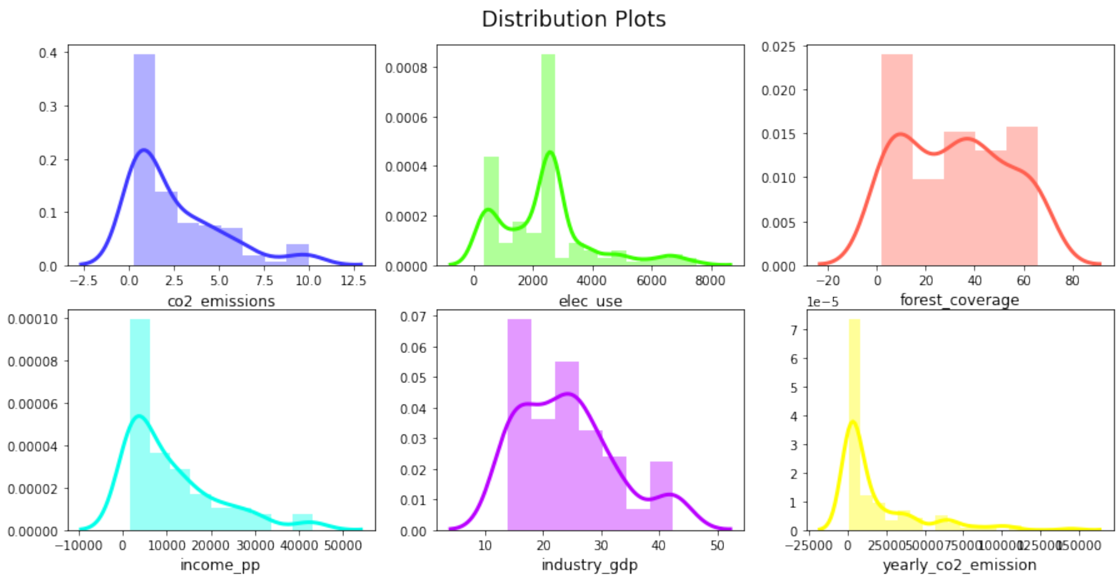
[36]:

```
#Dropping rows on the basis of IQR Range
df = df[~((df < (q1 - 1.5 * IQR)) |(df > (q3 + 1.5 * IQR))).any(axis=1)]
print(df.shape)
```

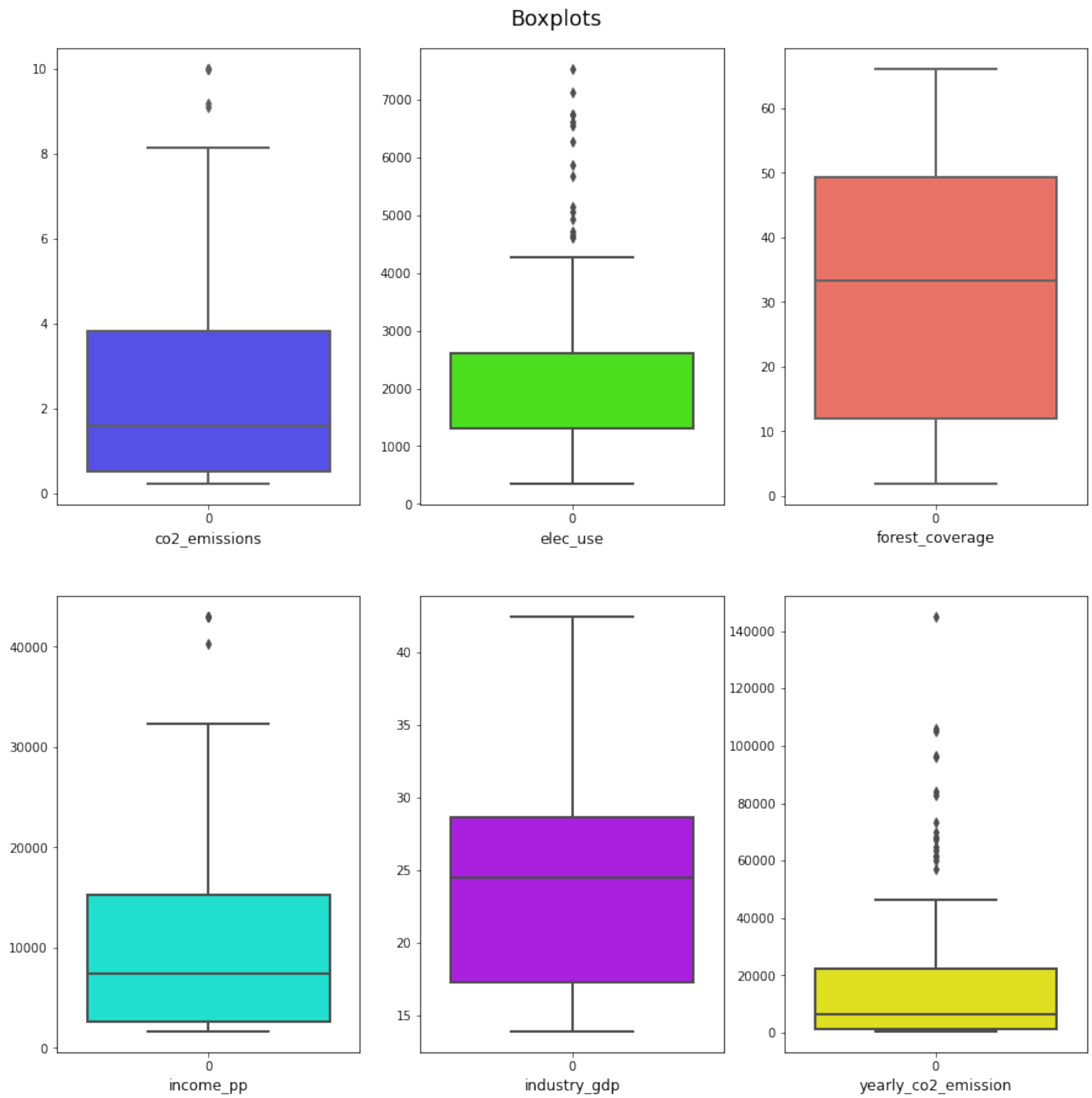
```
(143, 6)
```

[37]:

```
#Plotting Distribution Plots after Outlier Treatment
distributionPlot(df ,rows = 2,cols = 3,Title = "Distribution Plots")
```



[38]:
```
#Plotting Boxplots Plots after Outlier Treatment
boxPlot(df,rows = 2,cols = 3,Title = "Boxplots")
```



8 Splitting the Dataset into Train and Test Datasets

[39]:
```
#Splitting Data into Independent and Dependent variables
df2 = df.copy()
y = df2.pop('co2_emissions')
X = df2
```

[40]:
```
#Splitting further into train and test datasets
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    train_size=0.7,
```

```
                                                    test_size = 0.3,␣
↪random_state=100)
```

9 Model building and Feature Selection using Lasso Regression

[41]:
```
#List of Alphas to tune
params = {'alpha': [0.0001, 0.001, 0.01, 0.05, 0.1,
 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 2.0, 3.0,
 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 20, 50, 100, 500, 1000 ]}

#Cross Validation Folds
folds = 5
```

[42]:
```
#Creating an object of Lasso Class.
lasso = Lasso()

#Creating an object of GridSearchCV Class.
model_cv = GridSearchCV(estimator = lasso,
                        param_grid = params,
                        scoring= 'neg_mean_absolute_error',
                        cv = folds,
                        return_train_score=True,
                        verbose = 1)

#Fitting Model with the help of GridSearchCV
model_cv.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 28 candidates, totalling 140 fits
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 140 out of 140 | elapsed:    1.0s finished
```

[42]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
             estimator=Lasso(alpha=1.0, copy_X=True, fit_intercept=True,
                             max_iter=1000, normalize=False, positive=False,
                             precompute=False, random_state=None,
                             selection='cyclic', tol=0.0001, warm_start=False),
             iid='warn', n_jobs=None,
             param_grid={'alpha': [0.0001, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3,
                                   0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 2.0, 3.0,
                                   4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 20, 50,
                                   100, 500, 1000]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='neg_mean_absolute_error', verbose=1)
```

[43]:

```python
#Converting the results of the Model to a DataFrame
cv_results = pd.DataFrame(model_cv.cv_results_)

#Checking the DataFrame for the Rank of Alpha Values
cv_results.head(28)
```

[43]:

```
    mean_fit_time  std_fit_time  mean_score_time  std_score_time param_alpha  \
0        0.006160      0.003306         0.003428        0.001824      0.0001
1        0.001603      0.001981         0.002289        0.003241       0.001
2        0.001811      0.001573         0.003833        0.004015        0.01
3        0.000972      0.001494         0.000169        0.000338        0.05
4        0.003881      0.001433         0.001031        0.000929         0.1
5        0.006365      0.002330         0.001204        0.000679         0.2
6        0.002501      0.001251         0.001994        0.002526         0.3
7        0.002641      0.002191         0.003581        0.002352         0.4
8        0.003168      0.002996         0.003133        0.003064         0.5
9        0.002207      0.002117         0.003148        0.002576         0.6
10       0.001732      0.001592         0.003626        0.001842         0.7
11       0.002222      0.001121         0.002514        0.002740         0.8
12       0.002415      0.002071         0.003419        0.002566         0.9
13       0.003928      0.002093         0.001321        0.001624           1
14       0.005237      0.003189         0.000200        0.000400           2
15       0.000000      0.000000         0.006261        0.007669           3
16       0.005304      0.006036         0.000000        0.000000           4
17       0.003124      0.006249         0.003124        0.006249           5
18       0.003337      0.006187         0.000000        0.000000           6
19       0.006248      0.007653         0.000000        0.000000           7
20       0.006292      0.007706         0.000399        0.000798           8
21       0.000000      0.000000         0.006482        0.007946           9
22       0.003137      0.006273         0.003124        0.006248          10
23       0.006108      0.005999         0.000638        0.000867          20
24       0.002532      0.000820         0.001192        0.000957          50
25       0.002210      0.001168         0.002211        0.001991         100
26       0.005157      0.002673         0.000869        0.001737         500
27       0.001275      0.001265         0.001342        0.001964        1000

                params  split0_test_score  split1_test_score  \
0   {'alpha': 0.0001}          -0.868840          -0.811754
1    {'alpha': 0.001}          -0.868816          -0.811726
2     {'alpha': 0.01}          -0.868575          -0.811441
3     {'alpha': 0.05}          -0.867505          -0.810173
4      {'alpha': 0.1}          -0.866167          -0.808589
5      {'alpha': 0.2}          -0.863492          -0.805420
6      {'alpha': 0.3}          -0.860816          -0.802251
7      {'alpha': 0.4}          -0.858142          -0.799082
8      {'alpha': 0.5}          -0.855468          -0.795913
9      {'alpha': 0.6}          -0.852793          -0.792745
```

```
10     {'alpha': 0.7}          -0.850116          -0.789576
11     {'alpha': 0.8}          -0.847442          -0.786406
12     {'alpha': 0.9}          -0.844767          -0.783237
13     {'alpha': 1.0}          -0.842093          -0.780068
14     {'alpha': 2.0}          -0.827811          -0.752254
15     {'alpha': 3.0}          -0.828065          -0.749306
16     {'alpha': 4.0}          -0.828319          -0.749193
17     {'alpha': 5.0}          -0.828573          -0.749080
18     {'alpha': 6.0}          -0.828827          -0.748968
19     {'alpha': 7.0}          -0.829082          -0.748855
20     {'alpha': 8.0}          -0.829336          -0.748743
21     {'alpha': 9.0}          -0.829590          -0.748630
22    {'alpha': 10.0}          -0.829844          -0.748518
23      {'alpha': 20}          -0.832386          -0.747393
24      {'alpha': 50}          -0.842350          -0.744020
25     {'alpha': 100}          -0.862167          -0.738397
26     {'alpha': 500}          -1.003408          -0.715717
27    {'alpha': 1000}          -1.023215          -0.726245

    split2_test_score  split3_test_score  ...  mean_test_score  \
0           -1.267548          -0.872434  ...        -0.934904
1           -1.267577          -0.872466  ...        -0.934898
2           -1.267866          -0.872779  ...        -0.934836
3           -1.269150          -0.874181  ...        -0.934563
4           -1.270755          -0.875933  ...        -0.934221
5           -1.273967          -0.879438  ...        -0.933538
6           -1.278544          -0.882943  ...        -0.933128
7           -1.283160          -0.886448  ...        -0.932726
8           -1.286936          -0.889953  ...        -0.932156
9           -1.284746          -0.891255  ...        -0.929952
10          -1.282571          -0.890606  ...        -0.927360
11          -1.280381          -0.889957  ...        -0.924766
12          -1.278191          -0.889307  ...        -0.922172
13          -1.276001          -0.888658  ...        -0.919578
14          -1.254103          -0.888565  ...        -0.903647
15          -1.249555          -0.888703  ...        -0.902694
16          -1.249045          -0.888841  ...        -0.902683
17          -1.248536          -0.888978  ...        -0.902672
18          -1.248027          -0.889116  ...        -0.902660
19          -1.247518          -0.889253  ...        -0.902650
20          -1.247008          -0.889391  ...        -0.902639
21          -1.246499          -0.889529  ...        -0.902628
22          -1.245990          -0.889666  ...        -0.902617
23          -1.241388          -0.891042  ...        -0.902605
24          -1.227953          -0.895171  ...        -0.903112
25          -1.208358          -0.902052  ...        -0.905158
26          -1.120154          -0.995714  ...        -0.956560
```

27	-1.072703	-1.019317	...	-0.966285

	std_test_score	rank_test_score	split0_train_score	split1_train_score	\
0	0.167713	26	-0.879834	-0.879026	
1	0.167732	25	-0.879815	-0.878999	
2	0.167922	24	-0.879626	-0.878730	
3	0.168768	23	-0.878786	-0.877537	
4	0.169834	22	-0.877735	-0.876046	
5	0.171996	21	-0.875767	-0.873064	
6	0.174734	20	-0.874118	-0.870082	
7	0.177523	19	-0.872638	-0.867099	
8	0.180013	18	-0.871242	-0.864117	
9	0.180280	17	-0.870409	-0.861653	
10	0.180651	16	-0.870015	-0.859716	
11	0.181023	15	-0.869620	-0.859043	
12	0.181402	14	-0.869226	-0.858369	
13	0.181788	13	-0.868832	-0.857696	
14	0.180784	11	-0.870531	-0.869942	
15	0.179220	9	-0.870545	-0.873176	
16	0.178999	8	-0.870558	-0.873047	
17	0.178778	7	-0.870572	-0.872919	
18	0.178556	6	-0.870585	-0.872792	
19	0.178335	5	-0.870599	-0.872664	
20	0.178114	4	-0.870612	-0.872537	
21	0.177893	3	-0.870626	-0.872409	
22	0.177673	2	-0.870640	-0.872282	
23	0.175665	1	-0.870775	-0.871007	
24	0.169709	10	-0.872237	-0.867538	
25	0.161096	12	-0.874676	-0.863296	
26	0.133096	27	-0.913021	-0.890933	
27	0.122929	28	-0.922500	-0.929215	

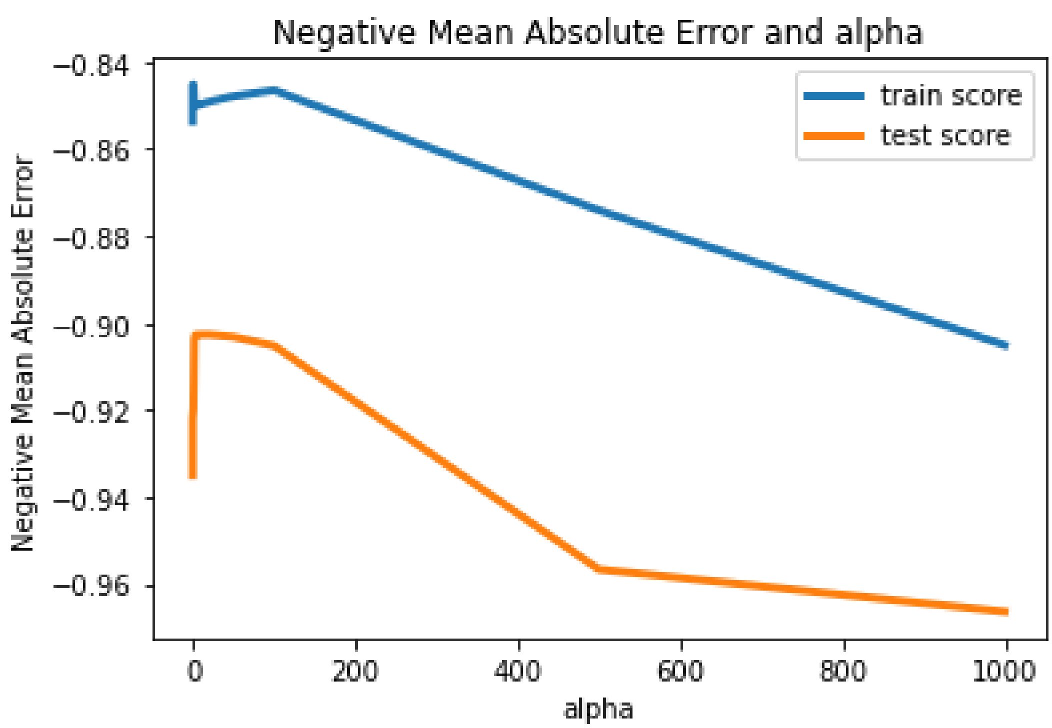
	split2_train_score	split3_train_score	split4_train_score	\
0	-0.777477	-0.850198	-0.881758	
1	-0.777492	-0.850184	-0.881732	
2	-0.777636	-0.850049	-0.881473	
3	-0.778278	-0.849449	-0.880319	
4	-0.779081	-0.848697	-0.878877	
5	-0.780685	-0.847195	-0.875993	
6	-0.782290	-0.846467	-0.873107	
7	-0.783895	-0.846590	-0.870376	
8	-0.785318	-0.847724	-0.867977	
9	-0.785444	-0.848564	-0.866222	
10	-0.785573	-0.848668	-0.864995	
11	-0.785699	-0.848892	-0.863770	
12	-0.785825	-0.849158	-0.863760	
13	-0.785952	-0.849424	-0.864034	

14	-0.788132	-0.849521	-0.867761
15	-0.789090	-0.849527	-0.868141
16	-0.788963	-0.849532	-0.868088
17	-0.788837	-0.849538	-0.868036
18	-0.788710	-0.849544	-0.867984
19	-0.788584	-0.849550	-0.867932
20	-0.788457	-0.849555	-0.867879
21	-0.788331	-0.849561	-0.867827
22	-0.788204	-0.849567	-0.867775
23	-0.786939	-0.849625	-0.867253
24	-0.783466	-0.850152	-0.866062
25	-0.778349	-0.851508	-0.864710
26	-0.766654	-0.906743	-0.893080
27	-0.833684	-0.924511	-0.915242

	mean_train_score	std_train_score
0	-0.853659	0.039834
1	-0.853644	0.039819
2	-0.853503	0.039671
3	-0.852874	0.039011
4	-0.852087	0.038186
5	-0.850541	0.036558
6	-0.849213	0.034966
7	-0.848120	0.033418
8	-0.847276	0.032017
9	-0.846458	0.031376
10	-0.845793	0.030932
11	-0.845405	0.030616
12	-0.845268	0.030450
13	-0.845188	0.030324
14	-0.849178	0.031493
15	-0.850095	0.031618
16	-0.850038	0.031644
17	-0.849980	0.031670
18	-0.849923	0.031696
19	-0.849866	0.031723
20	-0.849808	0.031749
21	-0.849751	0.031775
22	-0.849693	0.031802
23	-0.849120	0.032074
24	-0.847891	0.033059
25	-0.846508	0.034863
26	-0.874086	0.054348
27	-0.905030	0.035956

[28 rows x 21 columns]

[44]:
```
#Plotting Mean test and train scores with alpha
cv_results['param_alpha'] = cv_results['param_alpha'].astype('int32')

#Plotting the Graph
plt.plot(cv_results['param_alpha'], cv_results['mean_train_score'] , linewidth=3)
plt.plot(cv_results['param_alpha'], cv_results['mean_test_score'] , linewidth=3)
plt.xlabel('alpha')
plt.ylabel('Negative Mean Absolute Error')
plt.title("Negative Mean Absolute Error and alpha")
plt.legend(['train score', 'test score'], loc='upper right')
plt.show()
```



[45]:
```
#Choosing the Value of Alpha = 20 from the Rank Column in the above DataFrame
alpha = 20

#Creating another Lasso() object with alpha value = 20 as an initialized value
lasso = Lasso(alpha=alpha)

#Fitting the Model
lasso.fit(X_train, y_train)
```

[45]:
```
Lasso(alpha=20, copy_X=True, fit_intercept=True, max_iter=1000, normalize=False,
      positive=False, precompute=False, random_state=None, selection='cyclic',
```

```
        tol=0.0001, warm_start=False)
```

[46]:
```
#Checking the Value of Coefficients
lasso.coef_
```

[46]:
```
array([ 3.93785762e-04, -0.00000000e+00,  1.42500857e-04,  0.00000000e+00,
        1.85464339e-05])
```

[47]:
```
#Checking the Accuracy Score of the Lasso Model
lasso.score(X_train, y_train)
```

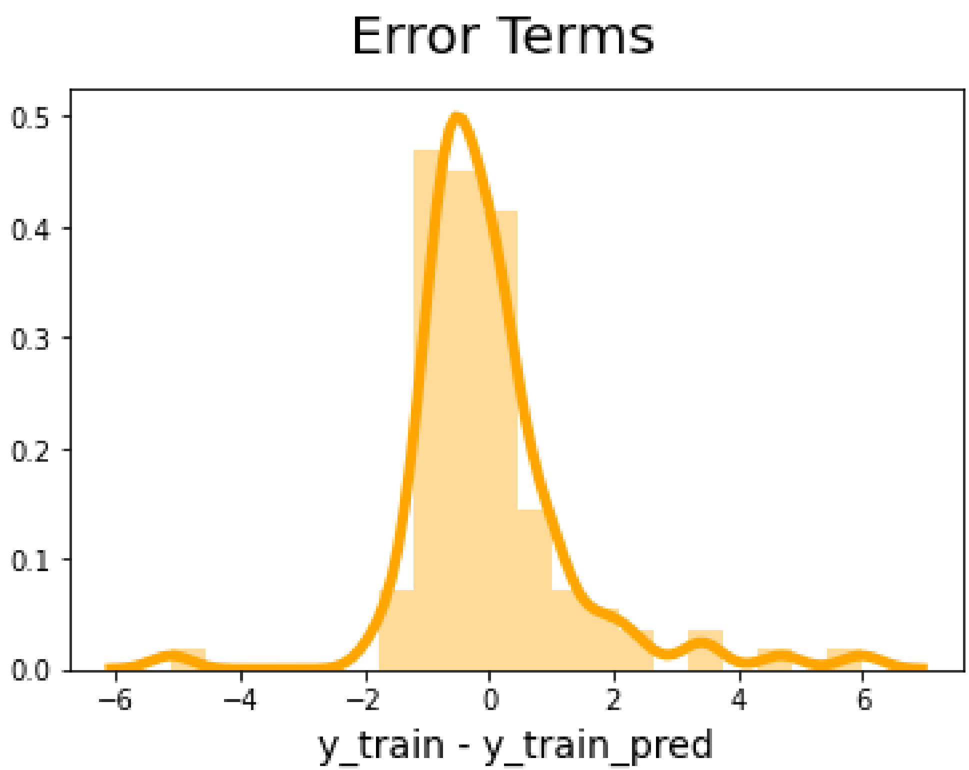
[47]:
```
0.7306500828024272
```

[48]:
```
y_train_pred = lasso.predict(X_train)
```

9.1 Residual Analysis of Training Data of the Model

[49]:
```
fig = plt.figure()
sns.distplot((y_train - y_train_pred), bins = 20 ,␣
 ↪color='orange',kde_kws=dict(linewidth=4))

#labelling the graph
fig.suptitle('Error Terms', fontsize = 20)
plt.xlabel('y_train - y_train_pred', fontsize = 14)
```

[49]:
```
Text(0.5, 0, 'y_train - y_train_pred')
```



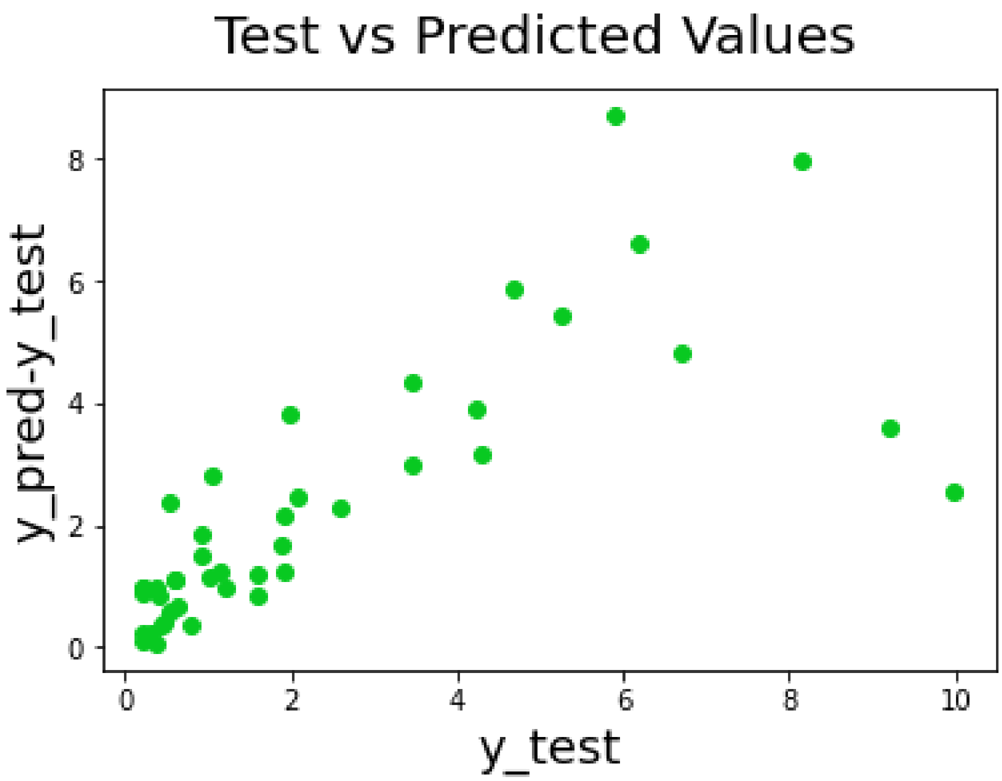
9.2 Making Predictions using Test Data

[50]:
```
y_pred = lasso.predict(X_test)
```

9.3 Evaluation of Model

[51]:
```
# Plotting y_test and y_pred to understand the spread.
fig = plt.figure()
plt.scatter(y_test, y_pred , color = '#08C921')
fig.suptitle('Test vs Predicted Values', fontsize = 20)
plt.xlabel('y_test', fontsize = 18)
plt.ylabel('y_pred-y_test', fontsize = 18)
```

[51]:
```
Text(0, 0.5, 'y_pred-y_test')
```



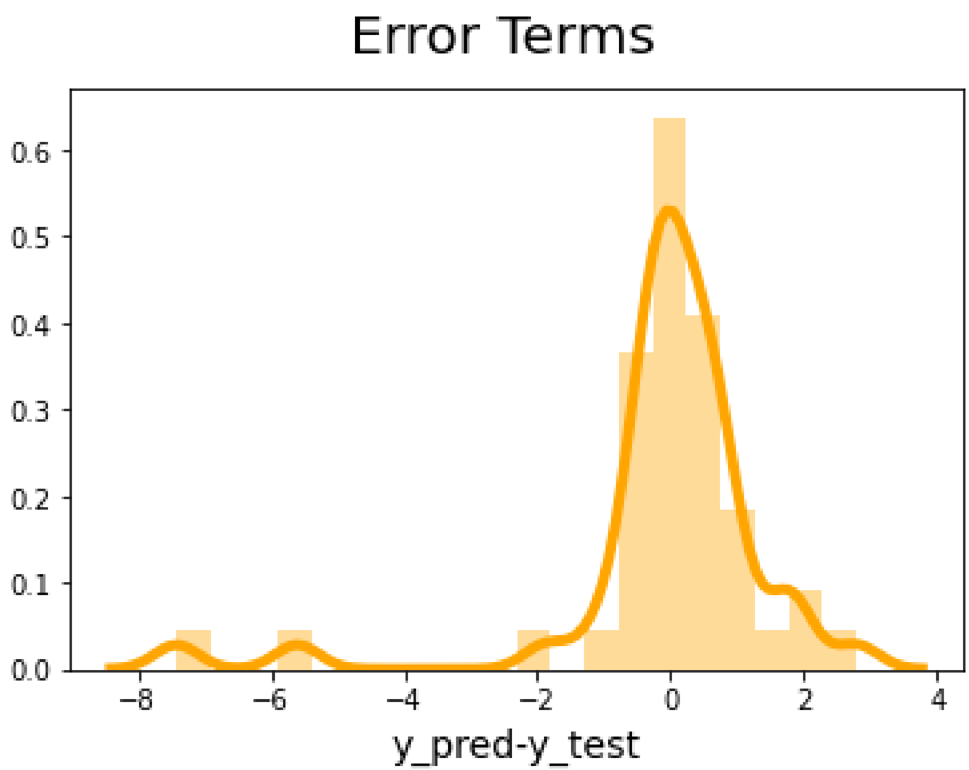
[52]:
```
lasso.score(X_test, y_test)
```

[52]: 0.5874630359593458

9.4 Residual Analysis of Testing Data of the Model

[53]:
```
fig = plt.figure()
sns.distplot((y_pred-y_test), bins = 20,␣
 ↪color='orange',kde_kws=dict(linewidth=4))

#labelling the graph
fig.suptitle('Error Terms', fontsize = 20)
plt.xlabel('y_pred-y_test', fontsize = 14)
```

[53]: Text(0.5, 0, 'y_pred-y_test')



9.5 Checking Error Metrics of the Model

[54]:
```
#Defining the Funtion 'errorMetrics' which will Calculate various␣
↪paramets(Squared error, Mean Squared error,
#Root Mean Squared error and R-Squared Value)
def errorMetrics(y_pred,y):
    error = y_pred - y

    SE = np.square(error) # squared errors
    MSE = np.mean(SE) # mean squared errors
    RMSE = np.sqrt(MSE) # Root Mean Squared Error, RMSE
    Rsquared = 1.0 - (np.var(error) / np.var(y))
    print('Squared Error',round(sum(SE),3) ,
          '\nMean Squared Error' , round(MSE,3),
          '\nRoot Mean Squared Error' , round(RMSE,3),
          '\nR Squared' , round(Rsquared,3),'\n\n')
```

[55]:
```
#Passing the pred_variables to the function
pred_variables=[[y_train_pred,y_train],
                [y_pred,y_test]]
```

```
j=1
for i in pred_variables:
    print('*'*40)
    if(j%2==0):
        print('---------Test Error of Model',int(j/2),'---------')
    else:
        print('---------Train Error of Model',int(j-(j/2)+1),'--------')

    #Calculating R2 value and appending them to Test R2 and Train R2 respectively
    errorMetrics(i[0],i[1])
    j+=1
print('*'*40)
```

```
****************************************
---------Train Error of Model 1 --------
Squared Error 171.425
Mean Squared Error 1.714
Root Mean Squared Error 1.309
R Squared 0.731


****************************************
---------Test Error of Model 1 ---------
Squared Error 117.328
Mean Squared Error 2.729
Root Mean Squared Error 1.652
R Squared 0.588


****************************************
```

9.6 Final Equation

[56]:
```
coeff = list(lasso.coef_)
col_name = list(X_train.columns)

print('co2_emissions = ' , end = ' ')
for i,j in zip(coeff, col_name):
    if i == 0:
        continue
    print(round(i,8),' * ',j , ' + ' , end='')
```

```
co2_emissions =  0.00039379  *  elec_use  + 0.0001425  *  income_pp  + 1.855e-05
*  yearly_co2_emission  +
```

- Final Equation is : `co2_emissions` **= 0.00039379 x** `elec_use` **+ 0.0001425 x** `income_pp` **+ 1.855e-05 x** `yearly_co2_emission`

9.7 Conclusion

The final model is the best possible Lasso Regression Model which could be developed. The dataset was not that great for this problem, faced problems while cleaning it. Eventhough this model has 73% Accuracy, an even better model can be developed.