National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Shreeya Namboori
Student ID: x18128947

School of Computing
National College of Ireland

Supervisor: Dr. Muhammad Iqbal

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Shreeya Namboori
Student ID:	x18128947
Programme:	Data Analytics
Year:	2018
Module:	MSc Research Project
Supervisor:	Dr. Muhammad Iqbal
Submission Due Date:	20/12/2018
Project Title:	Configuration Manual
Word Count:	1694
Page Count:	15

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Shreeya Namboori
Date:	12th December 2019

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Shreeya Namboori
x18128947

1 Introduction

The presented configuration manual document states the hardware and software requirements or tools used in the MSc research project "Forecasting Carbon Dioxide Emissions in the United States using Machine Learning". It also presents the code that was used in modeling.

2 Required System Configuration

2.1 Hardware Required

Processor : Inter(R) Core(TM) i5-6200U CPU @ 2.30GHz 2.40GHz
RAM : 8 GB
Storage Capacity : 1 TB (Terabyte) HDD (Hard Disk Drive)
System Type : 64-bit Operating System , x64- based processor
GPU : NVIDIA GEFORCE
Operating System : Windows 10 (64-bit operating system)

2.2 Software Required

Data Cleaning : RStudio
Programming and forecasting graphs :Jupyter Notebook by Anaconda
Visualization : PowerBI
Flow chart and diagrams : Draw.io (Online software for making charts and process flow diagrams)
Other tools: Microsoft Word (for making tables), Snipping tool (for taking screenshots of diagrams or tables), Microsoft Excel (for reading csv, and creating separate csv file for each sector using MS Excel filter).

3 Research Project Development

This section gives a detailed description of the steps followed from the starting of the project till the end to achieve the research objectives.

3.1 Data Collection and Pre-processing

The data for monthly CO_2 emissions from the United States (U.S.) has been collected from the EIA (U.S. Energy Information Administration) website [1]. The data contains CO_2 emissions of nine sectors from January 1973 to July 2019 and it has 5445 rows and six columns. RStudio software was used for cleaning this dataset.

	A	B	C	D	E	F	G	H	I
1	MSN	YYYYMM	Value	Column_C	Description	Unit			
2	CLEIEUS	197301	72.076	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide			
3	CLEIEUS	197302	64.442	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide			
4	CLEIEUS	197303	64.084	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide			
5	CLEIEUS	197304	60.842	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide			
6	CLEIEUS	197305	61.798	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide			
7	CLEIEUS	197306	66.538	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide			
8	CLEIEUS	197307	72.626	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide			
9	CLEIEUS	197308	75.181	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide			
10	CLEIEUS	197309	68.397	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide			
11	CLEIEUS	197310	67.668	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide			
12	CLEIEUS	197311	67.021	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide			
13	CLEIEUS	197312	71.118	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide			
14	CLEIEUS	197313	811.791	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide			
15	CLEIEUS	197401	70.55	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide			
16	CLEIEUS	197402	62.929	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide			

Figure 1: Raw CSV Data

```
#loading the readr library to read the data
library(readr)
#reading the downloaded data
ghg <- read_csv("MER_T11_06 (1).csv")
head(ghg)
#removing column 4
ghg<-ghg[c(1:3,5)]
tail(ghg)
#column names modified
names(ghg)[names(ghg) == "YYYYMM"] <- "YEAR"
names(ghg)[names(ghg) == "MSN"] <- "SECTOR"
names(ghg)[names(ghg) == "Description"] <- "SECTOR DESCRIPTION"
colSums(is.na(ghg))
#####################################################################
# 416 NA values are ommited as they are the missing values for two
#sectors whose emissions actually starts from Jan 1989
#####################################################################
ghg<-na.omit(ghg)
head(ghg)
ghg1<-ghg
```

Figure 2: RStudio Code : Removing columns and NA values and modifying column names

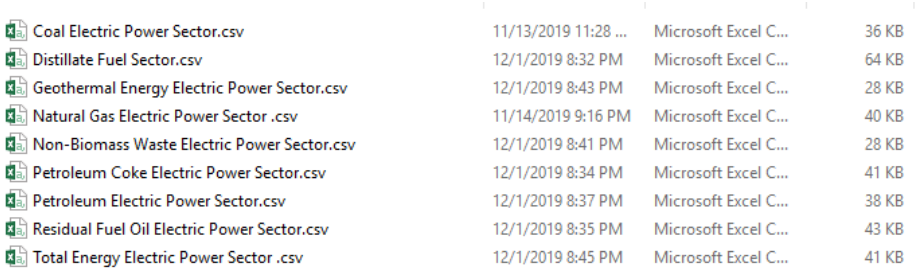
[1] https://www.eia.gov/totalenergy/data/browser/?tbl=T11.06#/?f=M

After reading the CSV file in ghg variable the fourth column is dropped off as it just contains the index number of the sectors. The column names 'YYYYMM', 'MSN' and 'Description' are modified to 'YEAR', 'SECTOR' and 'SECTOR DESCRIPTION'. The colSums(is.na(ghg)) function checks for the number of NA values in each column and 416 NA values are found in the Value column of ghg. These values come from the Geothermal Energy Electric Power Sector and the Non-Biomass Waste Electric Power Sector as their emissions have been getting recorded from January 1989, therefore, the emissions between the year January 1973 to December 1988 are given as NA values. These values are removed from ghg using na.omit(ghg) function (Figure 2).

```
ghg1<-ghg
# using substring to separate dates, month and year by "/"
ghg1$YEAR<- paste(substr(ghg$YEAR, 1, 4), sep = '/', substr(ghg$YEAR,5,6))
head(ghg1$YEAR,15)
# removing row with month mentioned as 13 by converting it to NA
ghg1$YEAR<-gsub("????*.[/]13", "NA", ghg1$YEAR)
head(ghg1,15)
ghg1$YEAR<-gsub("*?.NA", "NA", ghg1$YEAR)
head(ghg1,15)
ghg1$YEAR<-gsub("*?.NA", "NA", ghg1$YEAR)
head(ghg1,15)
ghg1$YEAR<-gsub("*?.NA", "NA", ghg1$YEAR)
head(ghg1,15)
colSums(is.na(ghg1))
write.csv(ghg1,file="uscarbon.csv",row.names=FALSE)
ghg <- read_csv("uscarbon.csv")
head(ghg,15)
colSums(is.na(ghg))
# 382 NA values are ommited
ghg<-na.omit(ghg)
head(ghg,13)
write.csv(ghg,file="uscarbon.csv",row.names=FALSE)
```

Figure 3: RStudio Code : Separating year and month, converting the 13th month of the year to NA values and omitting NA values

The 'YEAR' column contains the year and month joined together without a separator, therefore, a slash separator '/' is used for separating the year from the month. There is a 13th month each year which contains the total sum of emission of the 12 months. Since this value is an anomaly in the normal monthly emission data it is removed from the observations by converting the 'YEAR' column of the 13th month to NA. Since colSums do not recognize this data as NA it is written as CSV and then again loaded into the RStudio to check for NA values using colSums. This time the colSums show 382 NA values which are then omitted by na.omit function and this cleaned data is finally written down as CSV for modeling purpose (Figure 3). A separate CSV file was created for each sector using MS Excel filter by SECTOR to make it easier for the models to use data.



Coal Electric Power Sector.csv	11/13/2019 11:28 ...	Microsoft Excel C...	36 KB
Distillate Fuel Sector.csv	12/1/2019 8:32 PM	Microsoft Excel C...	64 KB
Geothermal Energy Electric Power Sector.csv	12/1/2019 8:43 PM	Microsoft Excel C...	28 KB
Natural Gas Electric Power Sector .csv	11/14/2019 9:16 PM	Microsoft Excel C...	40 KB
Non-Biomass Waste Electric Power Sector.csv	12/1/2019 8:41 PM	Microsoft Excel C...	28 KB
Petroleum Coke Electric Power Sector.csv	12/1/2019 8:34 PM	Microsoft Excel C...	41 KB
Petroleum Electric Power Sector.csv	12/1/2019 8:37 PM	Microsoft Excel C...	38 KB
Residual Fuel Oil Electric Power Sector.csv	12/1/2019 8:35 PM	Microsoft Excel C...	43 KB
Total Energy Electric Power Sector .csv	12/1/2019 8:45 PM	Microsoft Excel C...	41 KB

Figure 4: CSV files by sector

he exploratory analysis done on the cleaned data using the Power BI tool revealed three most CO_2 emitting sectors, the Coal Electric Power Sector, the Natural Gas Electric Power Sector, and the Total Energy Electric Power Sector. These three sectors are used in the machine learning models for time series forecasting.

4 Machine Learning Algorithms

This section includes the python code for constructing the forecasting models ARIMA (Autoregressive Integrated Moving Average), SVM (Support Vector Machine), SVM-PSO (Support Vector Machine optimized using Particle Swarm Optimization) and Prophet along with the code explanation. The coding was done in Python language using Jupyter Notebook by Anaconda and the common libraries used in the models are pandas for reading CSV or converting data into dataframe, numpy for dealing with arrays, matplotlib for plotting graphs and controlling the size of image and scikit-learn or sklearn for getting mean squared error and mean absolute error function.

4.1 ARIMA Model

```
# all imports are defined
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from pandas import datetime
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from matplotlib.pylab import rcParams
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.tsa.stattools import acf, pacf
from statsmodels.tsa.stattools import adfuller
rcParams['figure.figsize']= 20,10 #figure size is defined
#year column in data is parsed as date and made as index
def parser(x):
    return datetime.strptime(x,'%Y/%m')
df2 = pd.read_csv("C:/Users/Admin/Desktop/NCI courses/final research/USGHG/sector excel files/
Natural Gas Electric Power Sector .csv", index_col= 1,
 parse_dates=[1],date_parser=parser)
df2.head()
# columns deleted
del df2['SECTOR']
del df2['SECTOR DESCRIPTION']
df2.head()
```

Figure 5: ARIMA imports and datetime parsing

Figure 5 shows all the imports required for constructing and successfully forecasting from the ARIMA model. The datetime is imported from the pandas library to parse the year column of the CSV into a date format. The columns SECTOR and SECTOR DESCRIPTION are removed from the dataset as they are not required.

A test_stationarity function is created that checks if a time series is stationary or not by plotting the rolling mean and performing the Dickey-Fuller test (Figure 6). If the rolling mean plot looks stationary and the Dickey-Fuller test has a p-value of less than 0.05 then the time series is believed to be stationary. The adfuller is imported from statsmodels to perform the Dickey-Fuller test. The data is converted to log (df2_log) and subtracted by a shifted value of itself using .shift() function to produce a stationary time series. The number of times this differencing is done to make a time series stationary becomes the d input for the ARIMA. The d is 1 for all the sectors in the research.

```
# test stationarity fxn is created to test stationarity of different time series(usually transformed series)
def test_stationarity(timeseries):
 #Determine rolling stats
    movingavg=timeseries.rolling(window=12).mean()
    movingstd=timeseries.rolling(window=12).std()

    #Plotting rolling mean and standard deviation
    original=plt.plot(timeseries,color='blue',label='Original')
    mean=plt.plot(movingavg,color='red',label='Rolling Mean')
    std=plt.plot(movingstd,color='black',label='Rolling Std deviation')
    plt.legend(loc='best')
    plt.title('Rolling mean and Std deviation')
    plt.show(block=False)

    #Perform Dickey-Fuller test
    print('Dickey Fuller has been performed and the results are:')
    out=adfuller(timeseries['Value'], autolag='AIC')
    op=pd.Series(out[0:4],index=['Test-Statistic','p-value','Lags','Observations Used'])
    for key, value in out[4].items():
      op['Critical Value(%s)'%key] = value
    print(op)

# checking stationarity for df2
test_stationarity(df2)

# Making time series stationary (lowering rate at which rolling mean increases by converting data to log)
df2_log=np.log(df2)
plt.plot(df2_log)

# differencing of 1 is done and then the stationarityis checked
datashift=df2_log - df2_log.shift()
datashift.dropna(inplace=True)
test_stationarity(datashift)
#the graph shows a constant mean and the p value of DF test is also less than .05 meaning series is stationary
```

Figure 6: ARIMA time series stationarity checking function

To find the p and q values of ARIMA, ACF (Auto Correlation Function) plot and PACF (Partial Auto Correlation Function) plots are plotted (Figure 7). The p value (order of the AR term) is determined from the PACF plot while the q (order of the MA term) value is determined using the ACF plot. The lags at which the correlation value looks significant are selected as p and q values for the ARIMA model.

```
# plot acf plot
lag_acf = acf(datashift, nlags=30)
plt.figure(figsize=(16,7))
plt.plot(lag_acf, marker='.')
plt.axhline(y=0,linestyle='--',color='black')
plt.axhline(y=-1.96/np.sqrt(len(datashift)),linestyle='-',color='black')
plt.axhline(y=1.96/np.sqrt(len(datashift)),linestyle='-',color='black')
plt.title('ACF Plot')
#the lags having significant correlation ( crossing thresholds) values  are the q values

# plot pacf plot
lag_pacf = pacf(datashift, nlags=30,method='ols')
plt.figure(figsize=(16,7))
plt.plot(lag_pacf, marker='.')
plt.axhline(y=0,linestyle='--',color='black')
plt.axhline(y=-1.96/np.sqrt(len(datashift)),linestyle='-',color='black')
plt.axhline(y=1.96/np.sqrt(len(datashift)),linestyle='-',color='black')
plt.title('PACF Plot')
#the lags having significant correlation ( crossing thresholds) values  are the p values

# splitting the training and testing data into 90:10 ratio
train= np.log(df2.Value[:int(len(df2)*.90)])
test= np.log(df2.Value[len(train):])
#train.size
```

Figure 7: ARIMA ACF and PACF Plot

```
#ARIMA model training of order 8,1,12 for Natural Gas Electric Power Sector
# ARIMA model order of 12,1,3 for Coal Electric Power Sector
#ARIMA model order of 11,1,9 for Total Energy Electric Power Sector
model = ARIMA(train, order=(8,1,12))
results = model.fit(disp=-1)
datanew=datashift[0:len(train)]
plt.plot(datanew, label='Original')
plt.plot(results.fittedvalues, color='red', label= 'Fitting')
plt.legend(loc='upper left', fontsize=10)
plt.tight_layout()
#the graph below shows the training values fitted by the model vs actual train value

# Forecast the test values
forecst, se, conf = results.forecast(len(test))

# Making the forecast as panda series
forecst_series = pd.Series(forecst, index=test.index)

# Plotting the forecast
plt.figure(figsize=(12,5), dpi=100)
plt.plot(np.exp(test), label='Original', color='blue')
plt.plot(np.exp(forecst_series), label='Forecast',color='red')
plt.title('Forecast vs Original')
plt.legend(loc='upper left', fontsize=10)
plt.show()
print('Test Root Mean Squared Error for log:',np.sqrt(mean_squared_error(forecst_series, test)))
print('Test Mean Absolute Error for log:', mean_absolute_error(forecst_series, test))
print('Test Root Mean Squared Error:',np.sqrt(mean_squared_error(np.exp(forecst_series), np.exp(test))))
print('Test Mean Absolute Error:', mean_absolute_error(np.exp(forecst_series), np.exp(test)))
print('Test MAPE:',np.mean(np.abs((np.exp(test) - np.exp(forecst_series)) / np.exp(test)))* 100)
```

Figure 8: ARIMA training and forecasting

The p,d,q value found for ARIMA is used for training ARIMA(p,d,q) model (Figure 8). Each sector has a different ARIMA model for giving the best results, Coal Electric Power Sector uses ARIMA(12,1,3), Natural Gas Electric Power Sector uses ARIMA(8,1,12) and Total Energy Electric Power Sector uses ARIMA(11,1,9). The model is trained and fitted using .fit() function and the fitting is checked by plotting the .fittedvalues function. The .forecast() function gives the forecast for the number of time steps given as an input.

Since the results and the test values are in the log they are converted to their original form by using np.exp() function. These values are then plotted and used for calculating the evaluation metrics. The reference for forecasting from ARIMA model was taken from the works of Etienne (2019) and Brownlee (2017).

4.2 SVM Model

```
import numpy as np
from sklearn.svm import SVR
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt
from pandas import datetime
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from matplotlib.pylab import rcParams
rcParams['figure.figsize']= 20,10
def parser(x):
    return datetime.strptime(x,'%Y/%m')
df2 = pd.read_csv("C:/Users/Admin/Desktop/NCI courses/final research/USGHG/sector excel files/
Natural Gas Electric Power Sector .csv", index_col= 1, parse_dates=[1],date_parser=parser)
df2.head()

# deleting sector and sector description column
del df2['SECTOR']
del df2['SECTOR DESCRIPTION']


N= len(df2)
#making dataframe with icreasing index numbers
data = pd.DataFrame({
        'Steps': range(1, N + 1), #making incremental index named Steps for emission values
        'Value' : df2['Value']})
def get_list_data(data):
  data['Steps'] = pd.to_numeric(data['Steps'])
  return [ data['Steps'].tolist(), data['Value'].tolist() ] # Convert Series to list
steps, value = get_list_data(data)
```

Figure 9: SVM imports and separating time steps and data

SVM imports are similar to ARIMA except for MinMaxScaler import and SVR import from sklearn. The year is parsed as dates using parser function and unnecessary columns are removed. A dataframe having time steps in increasing order along with the emission values is made and these values are then passed to get_list_data function that converts series to list. So Steps become a separate series and it's corresponding emission Value becomes another series (Figure 9).
The data is then divided into four parts (steps1 and value1) for training and (steps2 and value2) for testing. MinMaxScalar is used for feature scaling the value1 and value2 in the range of 0 to 1 to make SVM forecasts more efficiently (Figure 10). The fit_transform function transforms the reshaped data and this data is then used for modeling. After trying linear, sigmoid and rbf kernel for SVM, rbf was chosen as the preferred kernel for forecasting. The model is fitted with train data using .fit() function and the predicted values from SVM are obtained by .predict() function. The transformed values are converted to their original form using .inverse_transform() function. The test data and predicted data are plotted against each other (Figure 11) and the evaluation metrics that

determine the accuracy of SVM model are also calculated. The reference for forecasting from SVM model is taken from the work of Nguyen (2019).

```
# dividing data into a ratio of 90:10 for train and test
steps1= steps[0:int(len(steps)*.90)]
steps2= steps[len(steps1):]
value1= value[0:int(len(value)*.90)]
value2= value[len(value1):]
#data is normalized for better performance
dataset1 = np.array(value1, dtype=np.float32)
dataset1 = np.reshape(dataset1, (-1, 1))
scaler1 = MinMaxScaler(feature_range=(0, 1))
dataset1 = scaler1.fit_transform(dataset1)
#global dataset2
dataset2 = np.array(value2, dtype=np.float32)
dataset2 = np.reshape(dataset2, (-1, 1))
scaler2 = MinMaxScaler(feature_range=(0, 1))
dataset2 = scaler2.fit_transform(dataset2)

# SVM used with kernel RBF regression with test and train
def predict_value(steps1, dataset1, steps2, dataset2):#SVM RBF
    steps1 = np.reshape(steps1,(len(steps1), 1))
    steps2 = np.reshape(steps2,(len(steps2), 1))
    #x = np.reshape(x,(len(x), 1))
    svr_rbf = SVR(kernel="rbf", gamma=0.1)
    # Fit regression model
    svr_rbf .fit(steps1, dataset1)
    predicted_value=svr_rbf.predict(steps2)
    predicted_value=scaler2.inverse_transform(np.reshape(predicted_value,(len(predicted_value), 1)))
    dataset2=scaler2.inverse_transform(np.reshape(dataset2,(len(dataset2), 1)))
    predicted_value1=svr_rbf.predict(steps1)
    predicted_value1=scaler2.inverse_transform(np.reshape(predicted_value1,(len(predicted_value1), 1)))
    dataset1=scaler2.inverse_transform(np.reshape(dataset1,(len(dataset1), 1)))
```

Figure 10: SVM training and testing data creation, feature scaling and model fitting

```
    dataset1=scaler2.inverse_transform(np.reshape(dataset1,(len(dataset1), 1)))
    # plotting train data along with SVM forecasted data
    plt.subplot(411)
    plt.plot(steps1, dataset1, label='Train Data')
    plt.plot(steps1, predicted_value1, label='RBF model',color='red')
    plt.title('Support Vector Regression Train data prediction')
    plt.ylabel('CO2 emission by the sector')
    plt.title('Support Vector Regression Forecasting')
    plt.legend()
    plt.show()
    plt.subplot(412)
    # plotting test data along with SVM forecasted data
    svm_predict= df2[len(steps1):]
    # making the array as a dataframe
    predicted_value= pd.DataFrame(predicted_value, columns=['Prediction'])
    # getting the values of prediction in svm_predict that contains original value and year
    svm_predict["Prediction"] = predicted_value['Prediction'].values
    plt.plot(svm_predict.index, dataset2, label='Test Data')
    plt.plot(svm_predict.index, svm_predict["Prediction"].values,label='RBF model',color='red')
    plt.xlabel('Year')
    plt.ylabel('CO2 emission by the sector')
    plt.title('Support Vector Regression Forecasting')
    plt.legend()
    plt.show()
    # printing evaluation metrics for SVM forecasting
    print('Test Mean Absolute Error:', mean_absolute_error(dataset2, predicted_value))
    print('Test Root Mean Squared Error:',np.sqrt(mean_squared_error(dataset2, predicted_value)))
    print('Test MAPE:',np.mean(np.abs((dataset2 - predicted_value) / dataset2))* 100)

# call the function predict_value function
predict_value(steps1, dataset1.ravel(), steps2, dataset2.ravel())
```

Figure 11: SVM forecast plotting and evaluation metric calculation

4.3 SVM-PSO

```
import pandas
import numpy as np
from pyswarm import pso
from sklearn.svm import SVR
import pandas as pd
#from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from pandas import datetime
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
def parser(x):
    return datetime.strptime(x,'%Y/%m')
df2 = pd.read_csv("C:/Users/Admin/Desktop/NCI courses/final research/USGHG/sector excel files/
Natural Gas Electric Power Sector .csv", index_col= 1, parse_dates=[1],date_parser=parser)
df2.head()
# columns deleted
del df2['SECTOR']
del df2['SECTOR DESCRIPTION']
N= len(df2)
#making dataframe with icreasing index numbers
data = pd.DataFrame({
        'Steps': range(1, N + 1), #making incremental index named Steps
        'Value' : df2['Value']})

# function for returning a list of the given data
def get_list_data(data):
  data['Steps'] = pd.to_numeric(data['Steps'])
  return [ data['Steps'].tolist(), data['Value'].tolist() ] # Converting to list
steps, value = get_list_data(data)

#defining 10 kfold splits
kf=KFold(n_splits=10)
#making a dataframe of steps and value
X_DF=pd.DataFrame(list(zip(data['Steps'],data['Value'])), columns=['X','Y'])
```

Figure 12: SVM-PSO libraries and KFold split

The imports in SVM-PSO are the same as SVM except for the KFold from the sklearn library and pso from the pyswarm library. A KFold split of 10 is applied to the data (Figure 12). The steps followed are similar to the SVM model except there are few new functions. The svrPso method splits the data into 10 fold split and after that MinMaxScalar is used for feature scaling. This function fit and predicts from the SVM model and calculates the MAPE (Mean Absolute Percentage Error) for all C and epsilon values (Figure 13). The calsMAPE function that calculates the MAPE value (Figure14) is called through svrPso. The data is again scaled using MinMaxScalar to be used by the predict_value function (Figure 15). The optimized value for C and epsilon values obtained from the PSO algorithm in the main_run method is passed to the predict_value method (Figure 16). The main_run functions have the upper bound and lower bound values for C and epsilon and it calls the pso function to iteratively run svrPso method until optimal parameters are found. The reference for forecasting from SVM-PSO model is taken from the work of Singh (2016).

```
def svrPso(params):
    for train, test in kf.split(X=X_DF['X'],y=X_DF['Y']):
        mape_total=0
        X_train, X_test, y_train, y_test = X_DF['X'][train], X_DF['X'][test], X_DF['Y'][train], X_DF['Y'][test]
        mape_total = 0
        y_train = y_train.values
        #print('X_train', X_train)
        y_train = np.array(y_train, dtype=np.float32)
        y_train = np.reshape(y_train, (-1, 1))
        scaler1 = MinMaxScaler(feature_range=(0, 1))
        y_train = scaler1.fit_transform(y_train)
        y_test = y_test.values
        y_test = np.array(y_test, dtype=np.float32)
        y_test = np.reshape(y_test, (-1, 1))
        scaler2 = MinMaxScaler(feature_range=(0, 1))
        y_test = scaler1.fit_transform(y_test)
        xtest=[]
        xtrain=[]
        xtrain=np.array(X_train)
        xtrain=np.reshape(xtrain,(len(xtrain),1))
        #print(xtrain)
        xtest=np.array(X_test)
        xtest=np.reshape(xtest,(len(xtest),1))
        #print('shape',xtrain.shape)
        nn = SVR(kernel='rbf', verbose = 2, cache_size=7000,C=params[0],epsilon=params[1], gamma=0.1)
         # Fit regression model
        nn.fit(xtrain, y_train)
        predict_value=nn.predict(xtest)
        y_test=scaler1.inverse_transform(np.reshape(y_test,(len(y_test), 1)))
        predict_value=scaler1.inverse_transform(np.reshape(predict_value,(len(predict_value), 1)))
        thisMape = calsMAPE(y_test, predict_value)
        mape_total = mape_total + thisMape
        print('Optimizing the Parameters ..... C = {c}, epsilon={e}, MAPE={m}'.format(c=params[0], e=params[1], m=mape_total))
        #predict_value(optiparams)
    return mape_total
```

Figure 13: SVM-PSO svrPso method

```
def calsMAPE(y_test, predict_value):
    y_test=np.array(y_test)
    MAPE = np.mean(np.abs((y_test - predict_value) / y_test))* 100
    return MAPE

N= len(df2)
data = pd.DataFrame({
        'Steps': range(1, N + 1),
        'Value' : df2['Value']})
def get_data(samplel_data):
  data['Steps'] = pd.to_numeric(samplel_data['Steps'])
  return [ data['Steps'].tolist(), data['Value'].tolist() ] # Convert Series to list
steps, value = get_data(data)
#print(data)
steps1= steps[0:int(len(steps)*.90)]
steps2= steps[len(steps1):]
value1= value[0:int(len(value)*.90)]
value2= value[len(value1):]
dataset1 = np.array(value1, dtype=np.float32)
dataset1 = np.reshape(dataset1, (-1, 1))
scaler1 = MinMaxScaler(feature_range=(0, 1))
dataset1 = scaler1.fit_transform(dataset1)
global dataset2
dataset2 = np.array(value2, dtype=np.float32)
dataset2 = np.reshape(dataset2, (-1, 1))
scaler2 = MinMaxScaler(feature_range=(0, 1))
dataset2 = scaler2.fit_transform(dataset2)
steps1 = np.reshape(steps1,(len(steps1), 1))
steps2 = np.reshape(steps2,(len(steps2), 1))
dataset2=scaler2.inverse_transform(np.reshape(dataset2,(len(dataset2), 1)))
```

Figure 14: SVM-PSO MAPE function and feature scaling

```
#SVM-PSO with with rbf kernel
def predict_value(c,e,m):
    svr_rbf = SVR(kernel='rbf', C=c,epsilon=e, gamma=0.1)
    # Fit regression model
    svr_rbf .fit(steps1, dataset1)
    predicted_value=svr_rbf.predict(steps2)
    predicted_value=scaler2.inverse_transform(np.reshape(predicted_value,(len(predicted_value), 1)))
    plt.subplot(411)
    plt.plot(steps2, dataset2, label='Test Data')
    plt.plot(steps2, predicted_value, c='g', label='RBF model')
    predicted_value1=svr_rbf.predict(steps1)
    plt.legend()
    plt.show()
    plt.subplot(412)
    # plotting test data along with SVM forecasted data
    svm_predict= df2[len(steps1):]
    # making the array as a dataframe
    predicted_value= pd.DataFrame(predicted_value, columns=['Prediction'])
    # getting the values of prediction in svm_predict that contains original value and year
    svm_predict["Prediction"] = predicted_value['Prediction'].values
    plt.plot(svm_predict.index, dataset2, label='Test Data')
    plt.plot(svm_predict.index, svm_predict["Prediction"].values,label='RBF model',color='red')
    plt.xlabel('Year')
    plt.ylabel('CO2 emission by the sector')
    plt.title('Support Vector Regression Forecasting')
    plt.legend()
    plt.show()
    print('Test Mean Absolute Error:', mean_absolute_error(dataset2, predicted_value))
    print('Test Root Mean Squared Error:',np.sqrt(mean_squared_error(dataset2, predicted_value)))
    print('Test MAPE:',np.mean(np.abs((dataset2 - predicted_value) / dataset2))* 100)
```

Figure 15: SVM-PSO predict_value function that gives forecatsing with optimized parameters

```
def main_run():
    #Upper bound and lower bound values are defined foe C and epsilon
      lb = [500.0, 0.05]
      ub = [800.0, 0.09]
      xopt, fopt = pso(svrPso, lb, ub, maxiter=1, debug=True,phip=10, swarmsize=200, minfunc=0.001 )
      print("######################")
      print(" ")
      print('ALL Parameters optimized: C = {c}, epsilon={e}, Overall MAPE={m}'.format(c=xopt[0], e=xopt[1], m=fopt))
      predict_value(xopt[0], xopt[1], fopt)
      print(" ")


print("****Searching optimization parameters****")
main_run()
print("***** Optimization parameters found *********")
```

Figure 16: SVM-PSO main function that calls the pso function to get optimal parameters and predict_value method to get forecasting with those parameters

4.4 Prophet

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from fbprophet import Prophet
from pandas import datetime
from sklearn.metrics import mean_absolute_error
from statsmodels.tools.eval_measures import rmse
from matplotlib.pylab import rcParams
rcParams['figure.figsize']= 20,10
def parser(x):
    return datetime.strptime(x,'%Y/%m')
df2 = pd.read_csv("C:/Users/Admin/Desktop/NCI courses/final research/USGHG/sector excel files/
Natural Gas Electric Power Sector .csv", index_col= 1, parse_dates=[1],date_parser=parser)
df2.head()

df1=df2['SECTOR DESCRIPTION']# getting sector name
# columns deleted
del df2['SECTOR']
del df2['SECTOR DESCRIPTION']
df2.head()
df2_cp = df2.copy()
df2_cp = df2.reset_index()
```

Figure 17: Prophet import

```
#Rename column names
df2_cp.columns = ['ds','y']
#train and testing data in 90:10 ratio
train= df2_cp[:int(len(df2_cp)*.90)]
test= df2_cp[len(train):]
#train the model
m = Prophet()
m.fit(train) #ditting the training data
#make future predictions
# M means Monthly frequency of data
future = m.make_future_dataframe(periods=len(test),freq='M')
prophet_pred = m.predict(future)
prophet_pred.tail

#making a dataframe containing year and prediction with year set as index
prophet_pred= pd.DataFrame({"Year" : prophet_pred[-len(test):]['ds'],
 "Prediction" : prophet_pred[-len(test):]["yhat"]})
prophet_pred = prophet_pred.set_index("Year")
prophet_pred.index.freq = "M" # M means Monthly frequency of data
#prophet_pred

# making new column for prediction values in test
test["Prophet_Predictions"] = prophet_pred['Prediction'].values
test = test.set_index("ds")
plt.figure(figsize=(20,7))
sns.lineplot(x= test.index, y=test["y"],color='blue', marker='.',label='Original')
sns.lineplot(x=test.index, y = test["Prophet_Predictions"],color='red', marker='.',label='Prediction');

# print the evaluation metrics
print('Test Mean Absolute Error:', mean_absolute_error(test['y'], test["Prophet_Predictions"]))
print('Test Root Mean Squared Error:',np.sqrt(rmse(test['y'], test["Prophet_Predictions"])))
print('Test MAPE:',np.mean(np.abs((test['y'] - test["Prophet_Predictions"]) / test['y']))* 100)
```

Figure 18: Prophet model traininig, testing and calculation of evaluation metrics

```
# predicting 36 months into future
future = m.make_future_dataframe(periods=len(test)+36,freq='M')
prophet_pred = m.predict(future)
prophet_pred.tail()

#making a dataframe containing year and prediction
prophet_pred= pd.DataFrame({"Year" : prophet_pred[-len(test):]['ds'], "Prediction" : prophet_pred[-len(test):]["yhat"]})
prophet_pred = prophet_pred.set_index("Year")
prophet_pred.index.freq = "M"
#prophet_pred

# defining fontsize for the x axis and y axis variables
plt.rcParams.update({'font.size': 18})

#plotting future values
plt.plot(prophet_pred,color='green', marker='.')
plt.title("Predicting Future Emissions from Prophet",fontsize=20)
plt.xlabel('Year',fontsize=20)
plt.ylabel(df1.values[0],fontsize=20)
```

Figure 19: Forecasting future results with Prophet

Prophet uses both matplotlib and seaborn library for making graphs and the Prophet model is imported from fbprophet (Figure 17). The year and value column names are converted to ds and y as used by the model. Model is trained and fitted with training data using .fit() function, The .make_future_dataframe() function makes a dataframe with future dates till the given input times steps and by calling .predict() function CO_2 emissions for those future dates are obtained. The frequency of the prediction is set to M which means monthly forecasts. A new column for predictions is made in the test data this is then used for plotting and calculating evaluation metric (Figure 18). The future forecast for 36 months is done by adding 36 to the periods in the .make_future_dataframe() function as the forecast frequency is set to monthly (Figure 19). These future forecasts are then plotted with the help of matplotlib library. The reference for forecasting from Prophet model was taken from the works of Vincent (2017) and Dabakoglu (2019).

References

Brownlee, J. (2017), 'How to Create an ARIMA Model for Time Series Forecasting in Python'.
URL: *https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/*

Dabakoglu, C. (2019), 'Time Series Forecasting — ARIMA, LSTM, Prophet with Python'.
URL: *https://medium.com/@cdabakoglu/time-series-forecasting-arima-lstm-prophet-with-python-e73a750a9887*

Etienne, B. (2019), 'Time Series in Python — Exponential Smoothing and ARIMA processes'.
URL: *https://towardsdatascience.com/time-series-in-python-exponential-smoothing-and-arima-processes-2c67f2a52788*

Nguyen, D. (2019), 'Learning Data Science — Predict Stock Price with Support Vector Regression (SVR)'.
URL: *https://itnext.io/learning-data-science-predict-stock-price-with-support-vector-regression-svr-2c4fdc36662*

Singh, R. (2016), 'PSO-Based-SVR to forecast potential delay time of bus arrival. Applied on City of Edmonton real data.'.
URL: *https://github.com/RamanSinghca/PSO-Based-SVR*

Vincent, T. (2017), 'A Guide to Time Series Forecasting with Prophet in Python 3'.
URL: *https://www.digitalocean.com/community/tutorials/a-guide-to-time-series-forecasting-with-prophet-in-python-3*