Configuration Manual

MSc Research Project
Data Analytics

Raksha Muddegowdana Koppalu Prakasha
Student ID: x19193181

School of Computing
National College of Ireland

Supervisor: Dr. Rashmi Gupta

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Raksha Muddegowdana Koppalu Prakasha
Student ID:	x19193181
Programme:	Data Analytics
Year:	2020
Module:	MSc Research Project
Supervisor:	Dr. Rashmi Gupta
Submission Due Date:	17/12/2020
Project Title:	Configuration Manual
Word Count:	4562
Page Count:	25

Signature:	
Date:	31st January 2021

Configuration Manual

Raksha Muddegowdana Koppalu Prakasha
x19193181

1 Introduction

The presented configuration manual, will describe in detail the system setup, software and hardware requirements in the implementation of research project - "Optimizing Waste Management Using IoT and Blockchain Based Machine Learning Forecasting Techniques".The manual also explains the coding used to model the various forecasting techniques.

2 System Configuration

2.1 Hardware Used

- **Processor:** Inter(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80GHz
- **RAM:** 8GB
- **Storage Capacity:** 1 TB (Terabyte) HDD (Hard Disk Drive)
- **GPU:** NVIDIA GEFORCE
- **Operating System:** Windows 10(64 bit operating system)
- **Model:** HP Pavilion

2.2 Software Used

- Gmail account for access to Google Drive.
- Google Colaboratory (Cloud based Jupyter Notebook environment)

3 Research Project Development

This research project follows the CRISP-DM methodology as mentioned by Wirth and Hipp (2000), for its implementation. According to the methodology, Data Collection, Data Processing/Preparation, Data Modeling and Model Evaluations are carried out, and the details of the same is discussed below.

3.1 Data Collection

Data for the implementing the proposal is collected from the reliable source by, Nikolaos (2019), which is a real time monitoring data of garbage bins set up as part of Composition EU Project. The data contains the fill levels of the garbage bin, and has 16,800 rows and 7 columns. The dataset is composed of hourly observations of garbage fill levels between the dates 20/06/2019 to 08/08/2019 and has various attributes like Fillpercentage, Battery Level, eventDate, Distance, etc. The raw data collected is as seen in 1

Code 1: Raw Data

```
{"data": [{
    "id": "5c2bf397530484f84463",
    "type": "Fill level",
    "request": {
        "measurements": {
            "FillPercentage": 70,
            "battery": 56
            },
        "eventDate": "2018-09-25T12:06:41.265Z",
        "updateState": true
        },
    "Distance" : 798
},
{
    "id": "5c2bf397530484f84463",
    "type": "Fill level",
    "request": {
        "measurements": {
            "FillPercentage": 70,
            "battery": 56
            },
        "eventDate": "2018-09-25T12:11:42.497Z",
        "updateState": true
        },
        "Distance" : 800
},
......]}
```

3.2 Data Preprocessing

The JSON data extracted is converted into a CSV file using Python programming in a Google Colab environment. The raw data collected is in JSON format and is converted to CSV format as part of data preparation(2).

Code 2: JSON to CSV Conversion

```
import csv
import json
from datetime import datetime
from dateutil import parser
```

```
fw_access_layers_data = open('/content/out_bins.json', 'r')
fw_access_layers_parsed = json.loads(fw_access_layers_data.read())
with open("bin.csv", "w") as outfile:
    f = csv.writer(outfile)
    f.writerow(["id", "type", "FillPercentage", "Battery", "eventDate",
        "updateState","Distance"])
    for i in range(16801):
      dt =  fw_access_layers_parsed['data'][i]['request']['eventDate']
      date_object = parser.parse(dt)
      fw_access_layers_parsed['data'][i]['request']['eventDate'] =
          date_object.date()
      f.writerow([fw_access_layers_parsed['data'][i]['id'],
          fw_access_layers_parsed['data'][i]['type'],
                  fw_access_layers_parsed['data'][i]['request']['measurements']['FillPercenta
                  fw_access_layers_parsed['data'][i]['request']['measurements']['battery'],
                  fw_access_layers_parsed['data'][i]['request']['eventDate'],
                  fw_access_layers_parsed['data'][i]['request']['updateState'],
                  fw_access_layers_parsed['data'][i]['Distance']])
```

Various other packages such as numpy, matplotlib, seaborn,etc needed to carry out data exploring are loaded in the colab environment.The CSV generated is converted into a data frame using Python Pandas library(3) and is pre-processed and checked for any missing/null values same(4).

Code 3: CSV to Python Dataframe

```
data =pd.read_csv('/content/bin.csv') #csv file generated is read using
    pandas library
data.head()
```

Code 4: Check for Null Values

```
data.isnull().sum() #data is checked for null or missing values
```

The data attribute is converted to a time series using pandas library to use in time series forecasting (5) and Augmented Dickey-Fuller(ADF) test is carried out to check for stationarity of the timeseries data(6).

Code 5: Data Attribute to datetime

```
data['eventDate'] = pd.to_datetime(data['eventDate']) #converting the
    eventDate attribute of the dataset datetime to use in time series
    forecasting.
```

Code 6: ADF Test

```
from statsmodels.tsa.stattools import adfuller

#Perform Augmented DickeyFuller test:
print('Results of Dickey Fuller Test:')
dftest = adfuller(df['FillPercentage'], autolag='AIC')
```

```
dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags
    Used','Number of Observations Used'])
for key,value in dftest[4].items():
    dfoutput['Critical Value (%s)'%key] = value

print(dfoutput)
```

In the next step, the data is explored to get deeper insights into the trends and patterns of the variables in the dataset. Exploratory Data Analysis is carried out to get more insights into the trends and patterns in the dataset. A heatmap(7) is plotted to get insights into the degree of correlation between the attributes of the dataset. Additionally trend of waste fill levels(8), and average waste fill levels(9) were plotted.

Code 7: Heatmap for Data Attributes

```
plt.figure(figsize=(15, 10)) #heatmap to check correlation of various data
    attributes
sns.heatmap(data.corr(), annot=True, cmap='gray')
```

Code 8: Plot for trend of FillPercentage

```
plt.style.use('fivethirtyeight')
cg = data.groupby('eventDate', as_index=False)['FillPercentage'].mean()
plt.figure(figsize=(24, 5))
sns.barplot(data=cg, x='eventDate', y='FillPercentage', palette='gray')
```

Code 9: Plot for average Fill Percentage

```
plt.style.use('fivethirtyeight')
data.groupby('eventDate')['FillPercentage'].mean().plot(figsize=(18, 5),
    color='grey')
```

Data Modeling and evaluation is explained in next section.

4 Data Modeling and Evaluation

4.1 ARIMA

ARIMA model is implemented to forecast fill percentage levels of smart bin. ARIMA model implemented was inspired by Fattah et al. (2018). It follows the following step:

- **Step 1:** All the required libraries are loaded(10).

Code 10: ARIMA: Libraries Loaded

```
from datetime import datetime
import numpy as np          #for numerical computations like
    log,exp,sqrt etc
import pandas as pd         #for reading & storing data, pre-processing
import matplotlib.pylab as plt #for visualization
#for making sure matplotlib plots are generated in Jupyter notebook
    itself
```

```
%matplotlib inline
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.stattools import acf, pacf
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.arima_model import ARIMA
from matplotlib.pylab import rcParams
rcParams['figure.figsize'] = 18, 9
```

- **Step 2:** The CSV file containing the data is loaded and data is read and stored as a dataframe using python pandas The variable "eventDate" is converted to datetime and set as index(11).

Code 11: ARIMA: Load CSV file

```
 #Load the CSV file
path = "/content/bin.csv" # Path in Colab environment
path = "bin.csv" #Path for local execution
dataset = pd.read_csv(path)
data = dataset[['eventDate', 'FillPercentage']]
print(len(data))
data['eventDate'] = pd.to_datetime(data['eventDate'])
df = data.set_index('eventDate')
```

- **Step 3:** Data pre-processing is carried out wherein, Rolling statistics is calculated and plotted for the FillPercentage(12 & 13) and Trend is estimated(14).

Code 12: ARIMA: Determine Rolling Statistics

```
#Determine rolling statistics
rolmean = df['FillPercentage'].rolling(window=12).mean() #window size 12
    denotes 12 months, giving rolling mean at yearly level
rolstd = df['FillPercentage'].rolling(window=12).std()
print(rolmean,rolstd)
```

Code 13: ARIMA: Plot Rolling Statistics

```
#Plot rolling statistics
plt.figure( figsize=(18,9))
orig = plt.plot(df['FillPercentage'], color='blue', label='Original')
mean = plt.plot(rolmean, color='red', label='Rolling Mean')
std = plt.plot(rolstd, color='black', label='Rolling Std')
plt.legend(loc='best')
plt.title('Rolling Mean & Standard Deviation')
plt.show(block=False)
```

Code 14: ARIMA: Estimate Trend

```
#Estimating trend
df_logScale = np.log(df)
print(df_logScale.isnull().sum())
df_logScale.dropna(inplace=True)
```

```
plt.plot(df_logScale)
```

- **Step 4:** Calculate and Plot ACF and PACF values(15).

Code 15: ARIMA: ACF and PACF Plots

```
#ACF and PACF plots

lag_acf = acf(df['FillPercentage'], nlags=20)
lag_pacf = pacf(df['FillPercentage'], nlags=20, method='ols')

#Plot ACF:
plt.figure(figsize =(18,9))
plt.subplot(121)
plt.plot(lag_acf)
plt.axhline(y=0, linestyle='--', color='gray')
plt.axhline(y=-1.96/np.sqrt(len(df)), linestyle='--', color='gray')
plt.axhline(y=1.96/np.sqrt(len(df)), linestyle='--', color='gray')
plt.title('Autocorrelation Function')

#Plot PACF
plt.figure(figsize =(18,9))
plt.subplot(122)
plt.plot(lag_pacf)
plt.axhline(y=0, linestyle='--', color='gray')
plt.axhline(y=-1.96/np.sqrt(len(df)), linestyle='--', color='gray')
plt.axhline(y=1.96/np.sqrt(len(df)), linestyle='--', color='gray')
plt.title('Partial Autocorrelation Function')

plt.tight_layout()
```

- **Step 5:** Modeling ARIMA model using auto_arima function to find optimal value for "p", "q" and "d" and summarize its results(16).

Code 16: ARIMA: Model to find optimal p

```
#auto_arima to find the optimal p,d,q values and build ARIMA model
from statsmodels.tsa.arima_model import ARIMA
import pmdarima as pm

model = pm.auto_arima(df, start_p=1, start_q=1,
                    test='adf',      # use adftest to find optimal 'd'
                    max_p=8, max_q=8, # maximum p and q
                    m=1,              # frequency of series
                    d=None,           # let model determine 'd'
                    seasonal=False, # No Seasonality
                    start_P=0,
                    D=0,
                    trace=True,
                    error_action='ignore',
                    suppress_warnings=True,
                    stepwise=True)
```

```
print(model.summary())
```

- **Step 6:** Build ARIMA model using the values from the above step(17).

Code 17: ARIMA: Build ARIMA Model

```
#  ARIMA Model
import numpy as dragon
import statsmodels.tsa.api as smt
import math
size = int(len(df)-10)
train_arima, test_arima = df[0:size], df[size:len(df)]
history = [x for x in train_arima]
# print(history)
predictions = list()
originals = list()
error_list = list()
mse_list = list()

print('Printing Predicted vs Expected Values...')
print('\n')
print(test_arima['FillPercentage'][0])
for t in range(len(test_arima)):
    model = smt.ARMA(df[size:len(df)], order=(2,0, 3))
    model_fit = model.fit(disp=0)
    output = model_fit.forecast()
    # print("out",output[0])
    pred_value = output[0]
    pred_value1 = pd.Series(pred_value, copy=True)
    print(pred_value1.head())


    original_value = test_arima['FillPercentage'][t]
    history.append(original_value)

    print("exp",original_value)

    error = ((abs(pred_value - original_value)) / original_value) *100
    mse = ((abs(pred_value - original_value)) / original_value)
    mse_list.append(mse)
    error_list.append(error)
    print('predicted = %f, expected = %f, error = %f ' % (pred_value,
        original_value, error), '%')

    predictions.append(float(pred_value))
    originals.append(float(original_value))

#Calculating the Forecast Accuracy
print('\n Means Error in Predicting Test Case Articles : %f ' %
    (sum(error_list)/float(len(error_list))), '%')
print('MSE:', (sum(mse_list)/float(len(mse_list))))
print('RMSE:',(math.sqrt(sum(mse_list)/float(len(mse_list)))))
```

- **Step 7:** Plot the forecast results of model against the actual values(18).

Code 18: ARIMA: Actual vs Forecasted values plot

```
# Plot of Actual vs Predicted Values
plt.figure(figsize = (15,10))
plt.plot(df)
plt.plot(predictions_ARIMA_diff)
plt.title('Actual vs Prediction plot')
plt.ylabel('waste fill levels')
plt.xlabel('eventDate')
plt.legend(['actual', 'prediction'], loc='upper left')
plt.show()
```

- **Step 8:** Model Evaluation(19).

Code 19: ARIMA: Model Evaluation

```
# Evaluation metrics
def forecast_accuracy(forecast, actual):
    mape = np.mean(np.abs(forecast - actual)/np.abs(actual)) # MAPE
    me = np.mean(forecast - actual)         # ME
    mae = np.mean(np.abs(forecast - actual)) # MAE
    mpe = np.mean((forecast - actual)/actual) # MPE
    rmse = np.mean((forecast - actual)**2)**.5 # RMSE
    corr = np.corrcoef(forecast, actual)[0,1] # corr
    mins = np.amin(np.hstack([forecast[:,None],
                              actual[:,None]]), axis=1)
    maxs = np.amax(np.hstack([forecast[:,None],
                              actual[:,None]]), axis=1)
    minmax = 1 - np.mean(mins/maxs)         # minmax
    # acf1 = acf(fc-test)[1]                  # ACF1
    return({'mape':mape, 'me':me, 'mae': mae,
           'mpe': mpe, 'rmse':rmse})

forecast_accuracy(df['FillPercentage'], predictions_ARIMA_diff)
```

- **Step 9:** Forecast Fill Percentage values for the future dates and data points(20).

Code 20: ARIMA: Future Forecast

```
#And we want to forecast for additional 7000 data points or 1 month.
fc = model_fit.predict(start=1, end=17000, dynamic=False)
results_ARIMA.plot_predict(1,25000)
```

4.2 LSTM

LSTM model is implemented to forecast fill percentage levels of smart bin. LSTM model is widely used model for time series forecasting and an example of the same can be seen in the research carried out by Gers et al. (2001). LSTM model implementation follows the steps,

- **Step 1:** All the required libraries are loaded(21).

Code 21: LSTM: Libraries Loaded

```
#import librarries required for LSTM modeling
import pandas as pd
import numpy as np
import keras
import tensorflow as tf
import matplotlib.pyplot as plt
from keras.preprocessing.sequence import TimeseriesGenerator
```

- **Step 2:** The CSV file containing the data is loaded and data is read and stored as a dataframe using python pandas(22). The variable "eventDate" is converted to datetime and set as index(23).

Code 22: LSTM: Load CSV file

```
filename = "/content/bin.csv" #load CSV file
df = pd.read_csv(filename)
print(df.info())
```

Code 23: LSTM: Variable to form "datetime"

```
df['eventDate'] = pd.to_datetime(df['eventDate']) #convert to
    datetime(time series'd)
df.set_axis(df['eventDate'], inplace=True) #making date as index
df.drop(columns=['id', 'type', 'Battery', 'updateState', 'Distance'],
    inplace=True) #drop columns not required for our modeling
```

- **Step 3:** Data pre-processing is carried out wherein, The data is split into 80% train set and 20% test set(24). Additionally, the data which is still inform of a sequence are converted to supervised data to be trained in a neural network(25).

Code 24: LSTM: Train and Test Data Split

```
fill_data = df['FillPercentage'].values
fill_data = fill_data.reshape((-1,1))

split_percent = 0.80 #split data as 80% training set and 20% test set
split = int(split_percent*len(fill_data))

fill_train = fill_data[:split]
fill_test = fill_data[split:]

date_train = df['eventDate'][:split]
date_test = df['eventDate'][split:]

print(len(fill_train))
print(len(fill_test))
```

Code 25: LSTM: Conversion to supervised data

```
#converting the data from sequence to supervised data to train LSTM model

look_back = 15

train_generator = TimeseriesGenerator(fill_train, fill_train,
   length=look_back, batch_size=20)
test_generator = TimeseriesGenerator(fill_test, fill_test,
   length=look_back, batch_size=1)
```

- **Step 4:** The model is trained with a single layer architecture of LSTM, with reLu activation function, Adam Optimizer and Mean Squared Error loss function (26).

Code 26: LSTM: Model Training

```
#import keras models and layers for LSTM
from keras.models import Sequential
from keras.layers import LSTM, Dense

model = Sequential()
model.add(
    LSTM(10,
        activation='relu',
        input_shape=(look_back,1))
)
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse') #model compilation

num_epochs = 500
model.fit_generator(train_generator, epochs=num_epochs, verbose=1)
    #model training
```

- **Step 5:** Predictions from the model are carried out using the test data and goodness of fit of the model trained is validated and plot for forecasted vs actual values is plotted(27).

Code 27: LSTM: Model Predictions/Forecast

```
import plotly.graph_objects as go #library for visualization

prediction = model.predict_generator(test_generator) #forecast/predict
    for test data

fill_train = fill_train.reshape((-1))
fill_test = fill_test.reshape((-1))
prediction = prediction.reshape((-1))

#Visualize Actual vs Predicted /Forecasted values
trace1 = go.Scatter(
    x = date_train,
    y = fill_train,
    mode = 'lines',
    name = 'Fill Percentage'
)
trace2 = go.Scatter(
    x = date_test,
    y = prediction,
    mode = 'lines',
    name = 'Forecasts'
)
trace3 = go.Scatter(
    x = date_test,
    y = fill_test,
    mode='lines',
    name = 'Ground Truth'
)
layout = go.Layout(
    title = "Waste Fill Levels",
    xaxis = {'title' : "Date"},
    yaxis = {'title' : "Fill Percentage"}
)
fig = go.Figure(data=[trace1, trace2, trace3], layout=layout)
fig.show()
plt.close()
```

- **Step 6:** Fill Percentage values for the future dates are forecasted and plot for the same is visualized(28).

Code 28: LSTM: Function for future forecasting

```
fill_data = fill_data.reshape((-1))

# Function to predict FillPercentage values for future dates
def predict(num_prediction, model):
    prediction_list = fill_data[-look_back:]

    for _ in range(num_prediction):
        x = prediction_list[-look_back:]
        x = x.reshape((1, look_back, 1))
        out = model.predict(x)[0][0]
        prediction_list = np.append(prediction_list, out)
    prediction_list = prediction_list[look_back-1:]

    return prediction_list

# Function to generate future dates
def predict_dates(num_prediction):
    last_date = df['eventDate'].values[-1]
    prediction_dates = pd.date_range(last_date,
        periods=num_prediction+1).tolist()
    return prediction_dates
```

```
num_prediction = 30 #Forecast Future values for next 30 days i.e next
    one month.
forecast = predict(num_prediction, model)
forecast_dates = predict_dates(num_prediction)

# Plot values of future forecasted values
trace1 = go.Scatter(
    x = date_test,
    y = prediction,
    mode = 'lines',
    name = 'Test Forecast'
)
trace2 = go.Scatter(
    x = forecast_dates,
    y = forecast,
    mode='lines',
    name = 'Future Forecast'
)
layout = go.Layout(
    title = "Waste Fill Levels",
    xaxis = {'title' : "eventDate"},
    yaxis = {'title' : "Fill Percentage"}
)
fig = go.Figure(data=[trace1, trace2], layout=layout)
fig.show()
plt.close()
```

- **Step 8:** Model Evaluation(29).

Code 29: LSTM: Model Evaluation

```
# Evaluation metrics
def forecast_accuracy(forecast, actual):
    mape = np.mean(np.abs(forecast - actual)/np.abs(actual)) # MAPE
    me = np.mean(forecast - actual)         # ME
    mae = np.mean(np.abs(forecast - actual)) # MAE
    mpe = np.mean((forecast - actual)/actual) # MPE
    rmse = np.mean((forecast - actual)**2)**.5 # RMSE
    corr = np.corrcoef(forecast, actual)[0,1] # corr
    mins = np.amin(np.hstack([forecast[:,None],
                              actual[:,None]]), axis=1)
    maxs = np.amax(np.hstack([forecast[:,None],
                              actual[:,None]]), axis=1)
    return({'mape':mape, 'me':me, 'mae': mae,
           'mpe': mpe, 'rmse':rmse,
           'corr':corr})

forecast_accuracy(testD[1], prediction)
```

4.3 CNN

CNN model is implemented to forecast fill percentage levels of smart bin. CNN model is commonly used for time series forecasting and an example of the same can be seen in the research carried out by Mehtab et al. (2020). CNN model implementation follows the steps,

- **Step 1:** All the required libraries are loaded(30).

Code 30: CNN: Libraries Loaded

```
#import libraries required for CNN modeling
import pandas as pd
import numpy as np
import keras
import tensorflow as tf
import matplotlib.pyplot as plt
from keras.preprocessing.sequence import TimeseriesGenerator
```

- **Step 2:** The CSV file containing the data is loaded and data is read and stored as a dataframe using python pandas(31). The variable "eventDate" is converted to datetime and set as index(32).

Code 31: CNN: Load CSV file

```
filename = "/content/bin.csv" #load CSV file
df = pd.read_csv(filename)
print(df.info())
```

Code 32: CNN: Variable to form "datetime"

```
df['eventDate'] = pd.to_datetime(df['eventDate']) #convert to
    datetime(time series'd)
df.set_axis(df['eventDate'], inplace=True) #making date as index
df.drop(columns=['id', 'type', 'Battery', 'updateState', 'Distance'],
    inplace=True) #drop columns not required for our modeling
```

- **Step 3:** Data pre-processing is carried out wherein, The data is split into 80% train set and 20% test set(33). Additionally, the data which is still inform of a sequence are converted to supervised data to be trained in a neural network(34).

Code 33: CNN: Train and Test Data Split

```
fill_data = df['FillPercentage'].values
fill_data = fill_data.reshape((-1,1))

split_percent = 0.80 #split data as 80% training set and 20% test set
split = int(split_percent*len(fill_data))

fill_train = fill_data[:split]
fill_test = fill_data[split:]
```

```
date_train = df['eventDate'][:split]
date_test = df['eventDate'][split:]

print(len(fill_train))
print(len(fill_test))
```

Code 34: CNN: Conversion to supervised data

```
#converting the data from sequence to supervised data to train LSTM model

look_back = 15

train_generator = TimeseriesGenerator(fill_train, fill_train,
    length=look_back, batch_size=20)
test_generator = TimeseriesGenerator(fill_test, fill_test,
    length=look_back, batch_size=1)
```

- **Step 4:** The model is trained using keras.layers package, Conv1D and uses reLu activation function, Adam Optimizer and Mean Squared Error loss function(35).

Code 35: CNN: Model Training

```
#import keras models and layers for CNN
from keras.models import Sequential
from keras.models import Sequential
from keras.layers import Dense,RepeatVector
from keras.layers import Flatten
from keras.layers import TimeDistributed
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

model = Sequential()
model.add(Conv1D(filters=128, kernel_size=2, activation='relu',
    input_shape=(look_back,1)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam',metrics=["accuracy"]) #model
    compilation
model.summary() #model summary

num_epochs = 500
model.fit_generator(train_generator, epochs=num_epochs, verbose=1)
    #model training
```

- **Step 5:** Predictions from the model are carried out using the test data and goodness of fit of the model trained is validated and plot for forecasted vs actual values is plotted(36).

Code 36: CNN: Model Predictions/Forecast

```
import plotly.graph_objects as go #library for visualization

prediction = model.predict_generator(test_generator) #forecast/predict
    for test data

fill_train = fill_train.reshape((-1))
fill_test = fill_test.reshape((-1))
prediction = prediction.reshape((-1))

#Visualize Actual vs Predicted /Forecasted values
trace1 = go.Scatter(
    x = date_train,
    y = fill_train,
    mode = 'lines',
    name = 'Fill Percentage'
)
trace2 = go.Scatter(
    x = date_test,
    y = prediction,
    mode = 'lines',
    name = 'Forecasts'
)
trace3 = go.Scatter(
    x = date_test,
    y = fill_test,
    mode='lines',
    name = 'Ground Truth'
)
layout = go.Layout(
    title = "Waste Fill Levels",
    xaxis = {'title' : "Date"},
    yaxis = {'title' : "Fill Percentage"}
)
fig = go.Figure(data=[trace1, trace2, trace3], layout=layout)
fig.show()
plt.close()
```

- **Step 6:** Fill Percentage values for the future dates are forecasted and plot for the same is visualized(37).

Code 37: CNN: Function for future forecasting

```
fill_data = fill_data.reshape((-1))

# Function to predict FillPercentage values for future dates
def predict(num_prediction, model):
    prediction_list = fill_data[-look_back:]

    for _ in range(num_prediction):
        x = prediction_list[-look_back:]
        x = x.reshape((1, look_back, 1))
        out = model.predict(x)[0][0]
        prediction_list = np.append(prediction_list, out)
    prediction_list = prediction_list[look_back-1:]

    return prediction_list

# Function to generate future dates
def predict_dates(num_prediction):
    last_date = df['eventDate'].values[-1]
    prediction_dates = pd.date_range(last_date,
        periods=num_prediction+1).tolist()
    return prediction_dates

num_prediction = 30 #Forecast Future values for next 30 days i.e next
    one month.
forecast = predict(num_prediction, model)
forecast_dates = predict_dates(num_prediction)

# Plot values of future forecasted values
trace1 = go.Scatter(
    x = date_test,
    y = prediction,
    mode = 'lines',
    name = 'Test Forecast'
)
trace2 = go.Scatter(
    x = forecast_dates,
    y = forecast,
    mode='lines',
    name = 'Future Forecast'
)
layout = go.Layout(
    title = "Waste Fill Levels",
    xaxis = {'title' : "eventDate"},
    yaxis = {'title' : "Fill Percentage"}
)
fig = go.Figure(data=[trace1, trace2], layout=layout)
fig.show()
plt.close()
```

- **Step 8:** Model Evaluation(38).

Code 38: CNN: Model Evaluation

```
# Evaluation metrics
def forecast_accuracy(forecast, actual):
    mape = np.mean(np.abs(forecast - actual)/np.abs(actual)) # MAPE
    me = np.mean(forecast - actual)          # ME
    mae = np.mean(np.abs(forecast - actual)) # MAE
    mpe = np.mean((forecast - actual)/actual) # MPE
    rmse = np.mean((forecast - actual)**2)**.5 # RMSE
    corr = np.corrcoef(forecast, actual)[0,1] # corr
    mins = np.amin(np.hstack([forecast[:,None],
                              actual[:,None]]), axis=1)
    maxs = np.amax(np.hstack([forecast[:,None],
                              actual[:,None]]), axis=1)
    return({'mape':mape, 'me':me, 'mae': mae,
            'mpe': mpe, 'rmse':rmse,
            'corr':corr})

forecast_accuracy(testD[1], prediction)
```

4.4 MLP

MLP model is implemented to forecast fill percentage levels of smart bin. MLP model is a deep learning model for time series forecasting and an example of the same can be seen in the research carried out by Shiblee et al. (2008). MLP model implementation follows the steps,

- **Step 1:** All the required libraries are loaded(39).

Code 39: MLP: Libraries Loaded

```
#import libraries required for MLP modeling
import pandas as pd
import numpy as np
import keras
import tensorflow as tf
import matplotlib.pyplot as plt
from keras.preprocessing.sequence import TimeseriesGenerator
```

- **Step 2:** The CSV file containing the data is loaded and data is read and stored as a dataframe using python pandas(40). The variable "eventDate" is converted to datetime and set as index(41).

Code 40: MLP: Load CSV file

```
filename = "/content/bin.csv" #load CSV file
df = pd.read_csv(filename)
print(df.info())
```

Code 41: MLP: Variable to form "datetime"

```
df['eventDate'] = pd.to_datetime(df['eventDate']) #convert to
    datetime(time series'd)
df.set_axis(df['eventDate'], inplace=True) #making date as index
df.drop(columns=['id', 'type', 'Battery', 'updateState', 'Distance'],
    inplace=True) #drop columns not required for our modeling
```

- **Step 3:** Data pre-processing is carried out wherein, The data is split into 80% train set and 20% test set(42). Additionally, the data which is still inform of a sequence are converted to supervised data to be trained in a neural network(43).

Code 42: MLP: Train and Test Data Split

```
fill_data = df['FillPercentage'].values
fill_data = fill_data.reshape((-1,1))

split_percent = 0.80 #split data as 80% training set and 20% test set
split = int(split_percent*len(fill_data))

fill_train = fill_data[:split]
fill_test = fill_data[split:]

date_train = df['eventDate'][:split]
date_test = df['eventDate'][split:]

print(len(fill_train))
print(len(fill_test))
```

Code 43: MLP: Conversion to supervised data

```
#converting the data from sequence to supervised data to train MLP model

look_back = 15

train_generator = TimeseriesGenerator(fill_train, fill_train,
    length=look_back, batch_size=20)
test_generator = TimeseriesGenerator(fill_test, fill_test,
    length=look_back, batch_size=1)
```

- **Step 4:** A simple MLP architecture with single layer and reLu activation function is used to train the data. The model is trained using Adam Optimizer and Mean Squared Error Loss Function for 500 epochs(44).

Code 44: MLP: Model Training"

```
#import keras models and layers for MLP
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(1, activation='relu', input_shape=(look_back,1)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam') #model
    compilation
num_epochs = 500
model.fit_generator(train_generator, epochs=num_epochs, verbose=1)
    #model training
```

- **Step 5:** Predictions from the model are carried out using the test data and goodness of fit of the model trained is validated and plot for forecasted vs actual values is plotted(45).

Code 45: MLP: Model Predictions/Forecast

```
import plotly.graph_objects as go #library for visualization

prediction = model.predict_generator(test_generator) #forecast/predict
    for test data

fill_train = fill_train.reshape((-1))
fill_test = fill_test.reshape((-1))
prediction = prediction.reshape((-1))

#Visualize Actual vs Predicted /Forecasted values
trace1 = go.Scatter(
    x = date_train,
    y = fill_train,
    mode = 'lines',
    name = 'Fill Percentage'
)
trace2 = go.Scatter(
    x = date_test,
    y = prediction,
    mode = 'lines',
    name = 'Forecasts'
)
trace3 = go.Scatter(
    x = date_test,
    y = fill_test,
    mode='lines',
    name = 'Ground Truth'
)
layout = go.Layout(
    title = "Waste Fill Levels",
    xaxis = {'title' : "Date"},
    yaxis = {'title' : "Fill Percentage"}
)
fig = go.Figure(data=[trace1, trace2, trace3], layout=layout)
fig.show()
plt.close()
```

- **Step 6:** Fill Percentage values for the future dates are forecasted and plot for the same is visualized(46).

Code 46: MLP: Function for future forecasting

```
fill_data = fill_data.reshape((-1))

# Function to predict FillPercentage values for future dates
def predict(num_prediction, model):
    prediction_list = fill_data[-look_back:]

    for _ in range(num_prediction):
        x = prediction_list[-look_back:]
        x = x.reshape((1, look_back, 1))
        out = model.predict(x)[0][0]
        prediction_list = np.append(prediction_list, out)
    prediction_list = prediction_list[look_back-1:]

    return prediction_list

# Function to generate future dates
def predict_dates(num_prediction):
    last_date = df['eventDate'].values[-1]
    prediction_dates = pd.date_range(last_date,
       periods=num_prediction+1).tolist()
    return prediction_dates

num_prediction = 30 #Forecast Future values for next 30 days i.e next
    one month.
forecast = predict(num_prediction, model)
forecast_dates = predict_dates(num_prediction)

# Plot values of future forecasted values
trace1 = go.Scatter(
    x = date_test,
    y = prediction,
    mode = 'lines',
    name = 'Test Forecast'
)
trace2 = go.Scatter(
    x = forecast_dates,
    y = forecast,
    mode='lines',
    name = 'Future Forecast'
)
layout = go.Layout(
    title = "Waste Fill Levels",
    xaxis = {'title' : "eventDate"},
    yaxis = {'title' : "Fill Percentage"}
)
fig = go.Figure(data=[trace1, trace2], layout=layout)
fig.show()
plt.close()
```

- **Step 8:** Model Evaluation(47).

Code 47: MLP: Model Evaluation

```
# Evaluation metrics
def forecast_accuracy(forecast, actual):
    mape = np.mean(np.abs(forecast - actual)/np.abs(actual)) # MAPE
    me = np.mean(forecast - actual)          # ME
    mae = np.mean(np.abs(forecast - actual)) # MAE
    mpe = np.mean((forecast - actual)/actual) # MPE
    rmse = np.mean((forecast - actual)**2)**.5 # RMSE
    corr = np.corrcoef(forecast, actual)[0,1] # corr
    mins = np.amin(np.hstack([forecast[:,None],
                              actual[:,None]]), axis=1)
    maxs = np.amax(np.hstack([forecast[:,None],
                              actual[:,None]]), axis=1)
    return({'mape':mape, 'me':me, 'mae': mae,
            'mpe': mpe, 'rmse':rmse,
            'corr':corr})

forecast_accuracy(testD[1], prediction)
```

4.5 CNN-LSTM

CNN-LSTM hybrid model is implemented to forecast fill percentage levels of smart bin. CNN-LSTM is a hybrid deep learning model for time series forecasting and an example of the same can be seen in the research carried out by Livieris et al. (2020). CNN-LSTM model implementation follows the steps,

- **Step 1:** All the required libraries are loaded(48).

Code 48: CNN-LSTM: Libraries Loaded

```
#import libraries required for CNN-LSTM modeling
import pandas as pd
import numpy as np
import keras
import tensorflow as tf
import matplotlib.pyplot as plt
from keras.preprocessing.sequence import TimeseriesGenerator
```

- **Step 2:** The CSV file containing the data is loaded and data is read and stored as a dataframe using python pandas(49). The variable "eventDate" is converted to datetime and set as index(50).

Code 49: CNN-LSTM: Load CSV file

```
filename = "/content/bin.csv" #load CSV file
df = pd.read_csv(filename)
print(df.info())
```

Code 50: CNN-LSTM: Variable to form "datetime"

```
df['eventDate'] = pd.to_datetime(df['eventDate']) #convert to
    datetime(time series'd)
df.set_axis(df['eventDate'], inplace=True) #making date as index
df.drop(columns=['id', 'type', 'Battery', 'updateState', 'Distance'],
    inplace=True) #drop columns not required for our modeling
```

- **Step 3:** Data pre-processing is carried out wherein, The data is split into 80% train set and 20% test set(51). Additionally, the data which is still inform of a sequence are converted to supervised data to be trained in a neural network(52).

Code 51: CNN-LSTM: Train and Test Data Split

```
fill_data = df['FillPercentage'].values
fill_data = fill_data.reshape((-1,1))

split_percent = 0.80 #split data as 80% training set and 20% test set
split = int(split_percent*len(fill_data))

fill_train = fill_data[:split]
fill_test = fill_data[split:]

date_train = df['eventDate'][:split]
date_test = df['eventDate'][split:]

print(len(fill_train))
print(len(fill_test))
```

Code 52: CNN-LSTM: Conversion to supervised data

```
#converting the data from sequence to supervised data to train CNN-LSTM
   model

look_back = 15

train_generator = TimeseriesGenerator(fill_train, fill_train,
   length=look_back, batch_size=20)
test_generator = TimeseriesGenerator(fill_test, fill_test,
   length=look_back, batch_size=1)
```

- **Step 4:** An hybrid model combining the layers of CNN and LSTM is implemented. The model has a CNN layer, followed by a LSTM layer, both activated using the reLu activation function. And similar to the other neural networks mentioned above, the model is trained using Adam Optimizer and Mean Squared Error Loss Function for 500 epochs,(53).

Code 53: CNN-LSTM: Model Training

```
#import keras models and layers for CNN-LSTM
from keras.layers import Dense,RepeatVector
from keras.layers import Flatten
from keras.layers import TimeDistributed
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.models import Sequential

model = Sequential()
model.add(Conv1D(filters=128, kernel_size=2, activation='relu',
   input_shape=(look_back,1)))
model.add((MaxPooling1D(pool_size=2)))
model.add(TimeDistributed(Flatten()))
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam') #model compilation
model.summary() #model summary

num_epochs = 500
model.fit_generator(train_generator, epochs=num_epochs, verbose=1)
    #model training
```

- **Step 5:** Predictions from the model are carried out using the test data and goodness of fit of the model trained is validated and plot for forecasted vs actual values is plotted(54).

Code 54: CNN-LSTM: Model Predictions/Forecast

```
import plotly.graph_objects as go #library for visualization

prediction = model.predict_generator(test_generator) #forecast/predict
    for test data

fill_train = fill_train.reshape((-1))
fill_test = fill_test.reshape((-1))
prediction = prediction.reshape((-1))

#Visualize Actual vs Predicted /Forecasted values
trace1 = go.Scatter(
    x = date_train,
    y = fill_train,
    mode = 'lines',
    name = 'Fill Percentage'
)
trace2 = go.Scatter(
    x = date_test,
    y = prediction,
    mode = 'lines',
    name = 'Forecasts'
)
trace3 = go.Scatter(
    x = date_test,
    y = fill_test,
    mode='lines',
    name = 'Ground Truth'
)
layout = go.Layout(
    title = "Waste Fill Levels",
    xaxis = {'title' : "Date"},
    yaxis = {'title' : "Fill Percentage"}
)
fig = go.Figure(data=[trace1, trace2, trace3], layout=layout)
fig.show()
```

```
plt.close()
```

- **Step 6:** Fill Percentage values for the future dates are forecasted and plot for the same is visualized(55).

Code 55: CNN-LSTM: Function for future forecasting

```
fill_data = fill_data.reshape((-1))

# Function to predict FillPercentage values for future dates
def predict(num_prediction, model):
    prediction_list = fill_data[-look_back:]

    for _ in range(num_prediction):
        x = prediction_list[-look_back:]
        x = x.reshape((1, look_back, 1))
        out = model.predict(x)[0][0]
        prediction_list = np.append(prediction_list, out)
    prediction_list = prediction_list[look_back-1:]

    return prediction_list

# Function to generate future dates
def predict_dates(num_prediction):
    last_date = df['eventDate'].values[-1]
    prediction_dates = pd.date_range(last_date,
        periods=num_prediction+1).tolist()
    return prediction_dates

num_prediction = 30 #Forecast Future values for next 30 days i.e next
    one month.
forecast = predict(num_prediction, model)
forecast_dates = predict_dates(num_prediction)

# Plot values of future forecasted values
trace1 = go.Scatter(
    x = date_test,
    y = prediction,
    mode = 'lines',
    name = 'Test Forecast'
)
trace2 = go.Scatter(
    x = forecast_dates,
    y = forecast,
    mode='lines',
    name = 'Future Forecast'
)
layout = go.Layout(
    title = "Waste Fill Levels",
    xaxis = {'title' : "eventDate"},
    yaxis = {'title' : "Fill Percentage"}
)
fig = go.Figure(data=[trace1, trace2], layout=layout)
fig.show()
plt.close()
```

- **Step 8:** Model Evaluation(56).

Code 56: CNN-LSTM: Model Evaluation

```
# Evaluation metrics
def forecast_accuracy(forecast, actual):
    mape = np.mean(np.abs(forecast - actual)/np.abs(actual)) # MAPE
    me = np.mean(forecast - actual)         # ME
    mae = np.mean(np.abs(forecast - actual)) # MAE
    mpe = np.mean((forecast - actual)/actual) # MPE
    rmse = np.mean((forecast - actual)**2)**.5 # RMSE
    corr = np.corrcoef(forecast, actual)[0,1] # corr
    mins = np.amin(np.hstack([forecast[:,None],
                              actual[:,None]]), axis=1)
    maxs = np.amax(np.hstack([forecast[:,None],
                              actual[:,None]]), axis=1)
    return({'mape':mape, 'me':me, 'mae': mae,
            'mpe': mpe, 'rmse':rmse,
            'corr':corr})

forecast_accuracy(testD[1], prediction)
```

References

Fattah, J., Ezzine, L., Aman, Z., Moussami, H. and Lachhab, A. (2018). Forecasting of demand using arima model, *International Journal of Engineering Business Management* **10**: 184797901880867.

Gers, F., Eck, D. and Schmidhuber, J. (2001). Applying lstm to time series predictable through time-window approaches, pp. 669–676.

Livieris, I., Pintelas, E. and Pintelas, P. (2020). A cnn-lstm model for gold price time series forecasting, *Neural Computing and Applications* **32**.

Mehtab, S., Sen, J. and Dasgupta, S. (2020). Analysis and forecasting of financial time series using cnn and lstm-based deep learning models.

Nikolaos, A. (2019). Composition fill level sensors datasets, *Zenodo Dataset* .
URL: *https://doi.org/10.5281/zenodo.3375560*

Shiblee, M., Kalra, P. and Chandra, B. (2008). Time series prediction with multilayer perceptron (mlp): A new generalized error based approach, pp. 37–44.

Wirth, R. and Hipp, J. (2000). Crisp-dm: Towards a standard process model for data mining, *Proceedings of the 4th International Conference on the Practical Applications of Knowledge Discovery and Data Mining* .