National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Tulio Begena Araujo
Student ID: 22133721

School of Computing
National College of Ireland

Supervisor: Musfira Jilani

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Tulio Begena Araujo
Student ID:	22133721
Programme:	Data Analytics
Year:	2023
Module:	MSc Research Project
Supervisor:	Musfira Jilani
Submission Due Date:	14/12/2023
Project Title:	Configuration Manual
Word Count:	957
Page Count:	14

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:	Tulio Begena Araujo
Date:	30th January 2024

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

Tulio Begena Araujo
22133721

1 Introduction

This manual has the software requirements to fully implement the project it is attached to. The code was written in Python language using the open-source IDE Jupyter Notebook. The following sections present the main steps on how to run the code.

2 Requirements

In this project, the experiments were ran using a hardware with the following specifications: AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx, 2.00 GHz, and 8 GB RAM. The use of more computer power than this is recommended, since the data processed consists in more than 1.5 million observations and the runtime for some models was up to 48 hours.

A Python 3 kernel in Jupyter Notebook App was used to load the data, starting with the reading of dataset's CSV file through Pandas library, and its conversion to dataframe format.

2.1 Libraries

The Python version, as the versions of the libraries used in this project are listed below:

Python version: 3.9.13 (main, Aug 25 2022, 23:51:50) [MSC v.1916 64 bit (AMD64)]
Pandas version: 1.4.4
Numpy version: 1.24.3
Seaborn version: 0.11.2
Scikit-learn version: 1.0.2
TensorFlow version: 2.13.0
Keras version: 2.13.1

The installation of these libraries can be done using the code showed in Figure 1.

To load the libraries, the commands are shown in Figure 2

3 Data Collection and Preparation

The dataset used was made available by Davari et al. (2021) and it is available for download in http://www.archive.ics.uci.edu/dataset/791/metropt+3+dataset.

It needs to be extracted and the archive with the data is in CSV format. The data can be loaded as showed in Figure 3. In this figure the format of the data can be seen.

```
#pandas and numpy
!pip install pandas numpy

#matplotlib and seaborn
!pip install matplotlib seaborn

#scikit-learn
!pip install scikit-learn

#TensorFlow
!pip install tensorflow

#Keras
!pip install keras
```

Figure 1: Commands to install required libraries.

```
import pandas as pd
import numpy as np

from matplotlib import pyplot as plt
import seaborn as sns

from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScal
import tensorflow as tf
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import RepeatVector
from keras.layers import TimeDistributed
```

Figure 2: Commands to load required libraries.

In [2]:
```
#import dataset
df = pd.read_csv("MetroPT3(AirCompressor).csv")
df
```

Out[2]:

	Unnamed: 0	timestamp	TP2	TP3	H1	DV_pressure	Reservoirs	Oil_temperature	Motor_current	COMP	DV_eletric	Towers	MPG	LPS	Pressure
0	0	2020-02-01 00:00:00	-0.012	9.358	9.340	-0.024	9.358	53.600	0.0400	1.0	0.0	1.0	1.0	0.0	
1	10	2020-02-01 00:00:10	-0.014	9.348	9.332	-0.022	9.348	53.675	0.0400	1.0	0.0	1.0	1.0	0.0	
2	20	2020-02-01 00:00:19	-0.012	9.338	9.322	-0.022	9.338	53.600	0.0425	1.0	0.0	1.0	1.0	0.0	
3	30	2020-02-01 00:00:29	-0.012	9.328	9.312	-0.022	9.328	53.425	0.0400	1.0	0.0	1.0	1.0	0.0	
4	40	2020-02-01 00:00:39	-0.012	9.318	9.302	-0.022	9.318	53.475	0.0400	1.0	0.0	1.0	1.0	0.0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
1516943	15169430	2020-09-01 03:59:10	-0.014	8.918	8.906	-0.022	8.918	59.675	0.0425	1.0	0.0	1.0	1.0	0.0	
1516944	15169440	2020-09-01 03:59:20	-0.014	8.904	8.888	-0.020	8.904	59.600	0.0450	1.0	0.0	1.0	1.0	0.0	
1516945	15169450	2020-09-01 03:59:30	-0.014	8.890	8.876	-0.022	8.892	59.600	0.0425	1.0	0.0	1.0	1.0	0.0	
1516946	15169460	2020-09-01 03:59:40	-0.012	8.876	8.864	-0.022	8.878	59.550	0.0450	1.0	0.0	1.0	1.0	0.0	
1516947	15169470	2020-09-01 03:59:50	-0.014	8.860	8.848	-0.022	8.864	59.475	0.0425	1.0	0.0	1.0	1.0	0.0	

1516948 rows × 17 columns

Figure 3: Command to load the data and its format.

3.1 Resampling

The data needs to be resampled aiming to regularize the time frequency and decrease the amount of processed data loaded into models. In Figure 4 are the commands used to perform the resampling.

```
#resampling to regularize the time frequency and decrease data size

#resample to each 20 minutes by the mean
df_20min = df.resample('1200S').mean()

#resample to each 20 minutes by the mean
df_60min = df.resample('3600S').mean()
```

Figure 4: Commands to resample data.

The resampling process creates rows with missing values. To deal with them, some rows were filled with the next non-null values and others were dropped, according to the criteria specified in the project. Figures 5 displays the commands used to perform these processes.

4 Defining Models Architecture

To build the models tested in the project, first it is needed to define the hyperparameters, as shown in Figure 6.

```
#fill null rows with the next non-null, if there are no consecutive null
#otherwise it is considered another trip and just drop the rows

def fill_df(df):
    for column in df.columns:
        consecutive_nulls = 0
        for i in range(len(df)):
            if pd.isnull(df[column].iloc[i]):
                consecutive_nulls += 1 #interval of null rows
            else:
                if consecutive_nulls > 0 and consecutive_nulls <= 2:
                    #fill with next non-null value
                    df[column].iloc[i - consecutive_nulls:i] = df[column].iloc[i]
                consecutive_nulls = 0
    return df
```

Figure 5: Commands to deal with null values.

```
#time windows for each frequency
#n_train ==> no of observations using for training
#n_test ==> no of observations using for testing and step size to create another window

#2 min resample
#n_train_2min = 5040 #1 week
#n_test_2min = 720 #1 day

#20 min resample
n_train_20min = 504 #1 week
n_test_20min = 72 #1 day

#60 min resample
n_train_60min = 168 #1 week
n_test_60min = 24 #1 day
```

```
#define dataframes for different types of sensors and frequencies
df_20min_analog = df_20min.iloc[:, 1:8]
df_20min_digital = df_20min.iloc[:, 8:]

df_60min_analog = df_60min.iloc[:, 1:8]
df_60min_digital = df_60min.iloc[:, 8:]

n_features_analog = 7
n_features_digital = 8
```

Figure 6: Commands to define hyperparameters.

Different models are trained for different types of sensors and different frequencies of the observations. To define dataframes for each case, the dataset is splitted using the iloc command from pandas library, as displayed in Figure 7.

```
#define dataframes for different types of sensors and frequencies
df_20min_analog = df_20min.iloc[:, 1:8]
df_20min_digital = df_20min.iloc[:, 8:]

df_60min_analog = df_60min.iloc[:, 1:8]
df_60min_digital = df_60min.iloc[:, 8:]

n_features_analog = 7
n_features_digital = 8
```

Figure 7: Commands to define dataframes for analog and digital sensors.

A total of 24 models are tested with these dataframes. 8 models with one lstm layer in encoder and decoder, 8 with two layers, and 8 with three layers. The decoder have the same number of layers as the encoder in every model. For each category one function is defined having the parameters of training size, number of features, and size of the layers as parameters. The Figures 8, 9, and 10 show the functions and parameters to define the models' architecture.

```
def create_model_11(n_train, n_features, layer1):

    model = Sequential()
    model.add(LSTM(layer1, activation='relu', input_shape=(n_train,n_features), return_sequences=False))
    model.add(RepeatVector(n_train))
    model.add(LSTM(layer1, activation='relu', return_sequences=True))
    model.add(TimeDistributed(Dense(n_features)))
    model.compile(optimizer='adam', loss='mae')
    model.summary()
    models_11_list.append(model)
```

```
#60min
##analog
create_model_11(n_train_60min, n_features_analog, 128)
create_model_11(n_train_60min, n_features_analog, 4)
##digital
create_model_11(n_train_60min, n_features_digital, 128)
create_model_11(n_train_60min, n_features_digital, 4)

#20min
##analog
create_model_11(n_train_20min, n_features_analog, 128)
create_model_11(n_train_20min, n_features_analog, 4)
##digital
create_model_11(n_train_20min, n_features_digital, 128)
create_model_11(n_train_20min, n_features_digital, 4)
```

Figure 8: Commands to define function and architectures for models with one layer in encoder.

5 Creating the Sliding Windows

This part is the instructions to create different windows from the same dataframe. In order to do that, a function was create that returns a list with the data corresponding to each window, as can be seen in Figure 11.

```
def create_model_22(n_train, n_features, layer1, layer2):

    model = Sequential()
    model.add(LSTM(layer1, activation='relu', input_shape=(n_train,n_features), return_sequences=True))
    model.add(LSTM(layer2, activation='relu', return_sequences=False))
    model.add(RepeatVector(n_train))
    model.add(LSTM(layer2, activation='relu', return_sequences=True))
    model.add(LSTM(layer1, activation='relu', return_sequences=True))
    model.add(TimeDistributed(Dense(n_features)))
    model.compile(optimizer='adam', loss='mae')
    model.summary()
    models_22_list.append(model)
```

```
#60min
##analog
create_model_22(n_train_60min, n_features_analog, 128, 64)
create_model_22(n_train_60min, n_features_analog, 4, 2)
##digital
create_model_22(n_train_60min, n_features_digital, 128, 64)
create_model_22(n_train_60min, n_features_digital, 4, 2)

#20min
##analog
create_model_22(n_train_20min, n_features_analog, 128, 64)
create_model_22(n_train_20min, n_features_analog, 4, 2)
##digital
create_model_22(n_train_20min, n_features_digital, 128, 64)
create_model_22(n_train_20min, n_features_digital, 4, 2)
```

Figure 9: Commands to define function and architectures for models with two layers in encoder.

```
def create_model_33(n_train, n_features, layer1, layer2, layer3):
    model = Sequential()
    model.add(LSTM(layer1, activation='relu', input_shape=(n_train,n_features), return_sequences=True))
    model.add(LSTM(layer2, activation='relu', return_sequences=True))
    model.add(LSTM(layer3, activation='relu', return_sequences=False))
    model.add(RepeatVector(n_train))
    model.add(LSTM(layer3, activation='relu', return_sequences=True))
    model.add(LSTM(layer2, activation='relu', return_sequences=True))
    model.add(LSTM(layer1, activation='relu', return_sequences=True))
    model.add(TimeDistributed(Dense(n_features)))
    model.compile(optimizer='adam', loss='mae')
    model.summary()
    models_33_list.append(model)
```

```
#60min
##analog
create_model_33(n_train_60min, n_features_analog, 128, 64, 32)
create_model_33(n_train_60min, n_features_analog, 8, 4, 2)
##digital
create_model_33(n_train_60min, n_features_digital, 128, 64, 32)
create_model_33(n_train_60min, n_features_digital, 8, 4, 2)

#20min
##analog
create_model_33(n_train_20min, n_features_analog, 128, 64, 32)
create_model_33(n_train_20min, n_features_analog, 8, 4, 2)
##digital
create_model_33(n_train_20min, n_features_digital, 128, 64, 32)
create_model_33(n_train_20min, n_features_digital, 8, 4, 2)
```

Figure 10: Commands to define function and architectures for models with three layers in encoder.

```python
#creating windows (splitting df)

def split_df(df, n_train, n_test):
    X, y, f = list(), list(), list()

    #creating train and test
    for train_start in range(0, len(df) - n_train - n_test + 1, n_test): #move window with step = n_test
        train_end = train_start + n_train
        test_end = train_end + n_test
        if test_end + n_test > len(df):
            break

        train, test = df.iloc[train_start:train_end, :], df.iloc[train_end:test_end, :]
        X.append(train)
        y.append(test)


    #creating failure index
        failure = False
        for start_failure, end_failure in zip(failure_times['start_times'], failure_times['end_times']):
            if (start_failure <= df.index[test_end + n_test]) and (end_failure >= df.index[test_end]):
                failure = True
        if failure:
            f.append(1)
        else:
            f.append(0)

    return X, y, f
```

Figure 11: Commands to define a function to create sliding windows.

```python
#analog sensors
##60min
train, test, failures_index60 = split_df(df_60min_analog,n_train_60min, n_test_60min)
analog60 = (train,test)
##20min
train, test, failures_index20 = split_df(df_20min_analog,n_train_20min, n_test_20min)
analog20 = (train,test)

#digital sensors
##60min
train, test, failures_index60 = split_df(df_60min_digital,n_train_60min, n_test_60min)
digital60 = (train,test)
##20 min
train, test, failures_index20 = split_df(df_20min_digital,n_train_20min, n_test_20min)
digital20 = (train,test)

#creating windows list corresponding to the models_list
windows_list = [analog60, analog60, digital60, digital60, analog20, analog20, digital20, digital20]
windows_list = windows_list*3
```

Figure 12: Commands to create the sliding windows.

Figure 12 show the creation of sliding windows and a list with one set of windows to each model.

The last step before training the models is to define the training and tests sets for each window and to reshape the data into a three dimensional format, as required to train LSTM models. The way to perform this step is displayed in Figure 13

```
#checking reshaping dimensions 60min

X_train = windows_list[0][0][0].copy()
X_test = windows_list[0][1][0].copy()
#reshape arrays
X_train, X_test = X_train.values, X_test.values
X_train = X_train.reshape((1, X_train.shape[0], X_train.shape[1]))
X_test = X_test.reshape((1, X_test.shape[0], X_test.shape[1]))
X_train.shape, X_test.shape
```

```
((1, 168, 7), (1, 24, 7))
```

```
#checking reshaping dimensions 20min

X_train = windows_list[4][0][0].copy()
X_test = windows_list[4][1][0].copy()
#reshape arrays
X_train, X_test = X_train.values, X_test.values
X_train = X_train.reshape((1, X_train.shape[0], X_train.shape[1]))
X_test = X_test.reshape((1, X_test.shape[0], X_test.shape[1]))
X_train.shape, X_test.shape
```

```
((1, 504, 7), (1, 72, 7))
```

Figure 13: Commands to create the sliding windows.

6 Training and Results

The training of the models are made using loops. The code to run the loop is showed below, divided into three figures (Figures 14, 15, and 16.

There are several steps in the loops. For each model, a scaler (MinMaxScaler from -1 to 1) is fitted and applied to the training set, then applied to the test set. The data is reshaped and fed to the model. Then, the model is used to make predictions and the mean absolute error is computed for the training and the test. This results are used to compute the evaluation metrics.

To conclude, the top-performing models are assessed based on their predictive capabilities for forecasting failures across various time windows. This evaluation involves testing the models with prediction windows of one (as done for every model), two, and three days. This multi-window evaluation provides a comprehensive understanding of how well the models can anticipate failures over different temporal scopes. The additional codes for two and three days windows are showed in Figures 17 and 18.

```
#learning rate
reduce_lr = tf.keras.callbacks.LearningRateScheduler(lambda x: 1e-3 * 0.90 ** x)
```

```
#looping through models and windows

#models
for i in range(0, len(models_list)):
    train = train_list[i]
    test = test_list[i]
    model = models_list[i]

    print(f'======== START MODEL {i} ========')
    ############### starting evaluation records ###############

    #start lists and variables to record results
    anomalies_list, anomalies_index = list(), list()
    mae_train_list, mae_test_list = list(), list()
    TP, TN, FP, FN = 0, 0, 0, 0

    ############### training and prediction ###############

    #windows
    for w in range(0, len(train)):

    #scaling
        X_train = train[w].copy()
        columns = X_train.columns.tolist()
        scalers={}
        for i in columns:
            scaler = MinMaxScaler(feature_range=(-1,1))
            s_s = scaler.fit_transform(X_train[i].values.reshape(-1,1))
            s_s=np.reshape(s_s,len(s_s))
            scalers['scaler_'+ i] = scaler
            X_train[i]=s_s

        X_test = test[w].copy()
        for i in columns:
            scaler = scalers['scaler_'+i]
            s_s = scaler.transform(X_test[i].values.reshape(-1,1))
            s_s=np.reshape(s_s,len(s_s))
            scalers['scaler_'+i] = scaler
            X_test[i]=s_s

    #reshape arrays
        X_train, X_test = X_train.values, X_test.values
        X_train = X_train.reshape((1, X_train.shape[0], X_train.shape[1]))
        X_test = X_test.reshape((1, X_test.shape[0], X_test.shape[1]))
        X_test_padded = np.pad(X_test, ((0, 0), (0, X_train.shape[1] - X_test.shape[1]), (0, 0)), mode='constant', constant_value
```

Figure 14: Training loop part 1.

```
#training model
    history_1=model.fit(X_train,X_train,epochs=25,validation_data=(X_train,X_train),batch_size=32,verbose=0,callbacks=[reduce

#predictions
    pred_train_1=model.predict(X_train)
    pred_test_1=model.predict(X_test_padded)
    pred_test_1 = pred_test_1[:, :X_test.shape[1], :]

#inverse scaling
    for index,i in enumerate(columns):
        scaler = scalers['scaler_'+i]
        pred_train_1[:,:,index]=scaler.inverse_transform(pred_train_1[:,:,index])
        pred_test_1[:,:,index]=scaler.inverse_transform(pred_test_1[:,:,index])

#computing error
    mae_train = np.mean(np.abs(X_train - pred_train_1))
    mae_test = np.mean(np.abs(X_test - pred_test_1))
    mae_train_list.append(mae_train)
    mae_test_list.append(mae_test)

#train loss
    train_loss = model.evaluate(X_train, X_train)

#test loss
    test_loss = model.evaluate(X_test_padded, X_test_padded)

    print()
    print(f'{w}/{len(train)}:')
    print(f'Train loss: {train_loss}')
    print(f'Test loss: {test_loss}')
    print(f'Train MAE: {mae_train}')
    print(f'Test MAE: {mae_test}')

#preliminar anomaly detection
    if mae_test > mae_train:
        anomalies_list.append(w)
        anomalies_index.append(1)
        print(f'Anomaly detected in test {w}!')
    else:
        anomalies_index.append(0)


#verify if result is TP, FP, TN, or TP
    if len(train) == len(failures_index60):
        failures_index = failures_index60
    else:
        failures_index = failures_index20
```

Figure 15: Training loop part 2.

```
    if failures_index[w] and anomalies_index[w]:
        TP +=1
    elif not failures_index[w] and anomalies_index[w]:
        FP += 1
    elif failures_index[w] and not anomalies_index[w]:
        FN += 1
    else:
        TN +=1


    print('=====')

############### printing metrics and results ###############


print(f'======== RESULTS ========')
print()
print(f'TP: {TP}, TN: {TN}, FP: {FP}, FN:{FN}')
print()

try:
    recall = (TP/(TP+FN))*100
    precision = (TP/(TP+FP))*100
    f1 = 2*precision*recall/(precision+recall)
    print(f'Recall: {recall}')
    print(f'Precision: {precision}')
    print(f'F1 score: {f1}')
except ZeroDivisionError:
    print('No TP to calculate metrics.')

#plotting difference between mae_train and mae_test
#reported failures in red

differences = [(val1 - val2) for val1, val2 in zip(mae_train_list, mae_test_list)]
plt.figure(figsize=(10, 5))

blue = differences.copy()
red = differences.copy()
for i in range(len(differences)):
    if failures_index[i]:
        blue[i] = np.nan
    else:
        red[i] = np.nan

plt.bar(range(len(differences)), blue, color='blue')
plt.bar(range(len(differences)), red, color='red')
plt.xlabel('Index')
plt.ylabel('Absolute Difference')
plt.title('Differences between mae_train and mae_test')
plt.show()
```

Figure 16: Training loop part 3.

```python
#2 days
TP, TN, FP, FN = 0, 0, 0, 0
for w in range(0, len(train)-1):
    if failures_index[w] and anomalies_index[w]:
        TP +=1
    elif failures_index[w+1] and anomalies_index[w]:
        TP +=1
    elif anomalies_index[w]:
        FP += 1
    elif failures_index60[w] and not anomalies_index[w]:
        FN += 1
    else:
        TN +=1


############### printing metrics and results ###############

print(f'======== RESULTS ========')
print()
print(f'TP: {TP}, TN: {TN}, FP: {FP}, FN:{FN}')
print()

try:
    recall = (TP/(TP+FN))*100
    precision = (TP/(TP+FP))*100
    f1 = 2*precision*recall/(precision+recall)
    print(f'Recall: {recall}')
    print(f'Precision: {precision}')
    print(f'F1 score: {f1}')
except ZeroDivisionError:
    print('No TP to calculate metrics.')
```

Figure 17: Code to evaluate best models' predictions for two days time window.

```python
#3 days
TP, TN, FP, FN = 0, 0, 0, 0
for w in range(0, len(train)-2):
    if failures_index[w] and anomalies_index[w]:
        TP +=1
    elif failures_index[w+1] and anomalies_index[w]:
        TP +=1
    elif failures_index[w+2] and anomalies_index[w]:
        TP +=1
    elif anomalies_index[w]:
        FP += 1
    elif failures_index60[w] and not anomalies_index[w]:
        FN += 1
    else:
        TN +=1


############### printing metrics and results ###############

print(f'======== RESULTS ========')
print()
print(f'TP: {TP}, TN: {TN}, FP: {FP}, FN:{FN}')
print()

try:
    recall = (TP/(TP+FN))*100
    precision = (TP/(TP+FP))*100
    f1 = 2*precision*recall/(precision+recall)
    print(f'Recall: {recall}')
    print(f'Precision: {precision}')
    print(f'F1 score: {f1}')
except ZeroDivisionError:
    print('No TP to calculate metrics.')
```

Figure 18: Code to evaluate best models' predictions for three days time window.

References

Davari, N., Veloso, B., Ribeiro, R. P., Pereira, P. M. and Gama, J. (2021). Predictive maintenance based on anomaly detection using deep learning for air production unit in the railway industry, *2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA)*, IEEE, pp. 1–10.