National College of Ireland

Configuration Manual

Demand prediction in a bike-sharing system using machine learning techniques
Masters in Data Analytics (B)

Ashish Rawat
X18185801

School of Computing
National College of Ireland

Supervisor: Paul Stynes

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Ashish Rawat
Student ID:	X18185801
Programme:	Masters in Data Analytics (B)
Year:	2020
Module:	Demand prediction in a bike-sharing system using machine learning techniques
Supervisor:	Paul Stynes
Submission Due Date:	17/08/2020
Project Title:	Configuration Manual
Word Count:	901
Page Count:	11

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:	
Date:	17th August 2020

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

Ashish Rawat
X18185801

1 Introduction

The following configuration manual discusses the hardware and software requirements for this research. It also describes the programming stages of the implementation in detail.

2 System Setup

2.1 Hardware specifications

- Processor: Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99GHz
- RAM: 8 GB
- System Type: 64-bit Operating System
- Storage: 512 SSD with 1 TB HDD

2.2 Software required

- Microsoft Excel: It is a spreadsheet program, which is used to store data in a grid of rows and columns. And in context to this study, has been used to access CSV (comma separated files) format files.
- Google Colaboratory: It is a free cloud service that has allowed continuous accessibility of 12 GB NVIDIA Tesla K80 GPU for 12 hours (Bar-El; 2018). This study uses the environment of colab to examine metro bike data and create models for machine learning. GPU setting is activated for some models to execute the code in less time.
- Tableau: Some visualizations in the final report were made through Tableau. It is a free visualization software which can be downloaded from Tableau website[1].

3 Project Development

The research work was conducted using version 3.6.7 of Python. And it was implemented in four phases : Data pre-processing and feature engineering, data preparation, modelling machine learning techniques and finally, evaluation.

[1]https://www.tableau.com/

3.1 Data Collection

The data for the study is obtained from the Metro Bike website[2]. It consists of 4 csv files containing quarterly data of bike trips made in 2019 and 1 csv file with station details. The data files are downloaded and stored in Google Drive for it to be accessed by the Google Colab.

3.2 Data Processing and Feature Engineering

First of all, Google Drive is connected to the Colab and the necessary libraries are imported as shown in figure 1.

Link the Google Drive with the Colab, to access data files.

```
[ ]  from google.colab import drive
     drive.mount('/content/drive')
```

Installing the libraries required.

```
[ ]  import os
     from os import listdir, makedirs, remove
     import os.path as osp
     import pandas as pd
     from datetime import datetime, timedelta
     import numpy as np
     from numpy.random import Generator, PCG64
```

Figure 1: Data Setup

To facilitate the processing and achieve the desired data set following functions are developed:

1. "time_bins" function as shown in figure 2 creates a dataframe of start date time and end date time with a fixed interval value and returns it.

```
def time_bins(start_date, end_date, interval_size):

    start = datetime.strptime(start_date, '%Y/%m/%d')
    end = datetime.strptime(end_date, '%Y/%m/%d')
    gap = timedelta(minutes=interval_size)

    interval_id = 0
    outer_limit = start
    vals = []
    while outer_limit < end:
        inner_limit = outer_limit
        outer_limit += gap
        i = [interval_id, inner_limit.strftime('%d/%m/%Y %H:%M'),
                          outer_limit.strftime('%d/%m/%Y %H:%M')]
        vals.append(pd.Series(i))
        interval_id += 1

    df_out = pd.DataFrame(vals)
    df_out = df_out.rename(columns={0: 'interval_id', 1: 'inner_limit', 2: 'outer_limit'})
    df_out['inner_limit'] = pd.to_datetime(df_out['inner_limit'], dayfirst=True)
    df_out['outer_limit'] = pd.to_datetime(df_out['outer_limit'], dayfirst=True)

    return df_out
```

Figure 2: time_bin function

[2]https://bikeshare.metro.net/about/data/

2. "format_data" function loads the bike trip data, one file at a time and performs pre-preprocessing on it. And at end combines it with the time interval dataset obtained from "time_bins" function to develop transformed dataset and weights for graph (figure 3).

```
def format_data(file_path, tinterval, interval_size, duration_upper, station_exclude, station_const_ids):

    def round_minutes(dt, direction, resolution):
        new_minute = (dt.minute // resolution + (1 if direction == 'up' else 0)) * resolution
        return dt + timedelta(minutes=new_minute - dt.minute)

    # Read bike trip data and load it into a pandas dataframe.
    df_raw = pd.read_csv(file_path, low_memory= False)
    df_raw['End Date'] = pd.to_datetime(df_raw['end_time'], dayfirst=True)
    df_raw['Start Date'] = pd.to_datetime(df_raw['start_time'], dayfirst=True)
    df_raw = df_raw.rename(columns={'end_station' : 'EndStation Id', 'start_station': 'StartStation Id', 'duration': 'Duration'})
    df_raw = df_raw[~df_raw['EndStation Id'].isna()]
    df_raw['EndStation Id'] = df_raw['EndStation Id'].astype(int)
    df_raw['StartStation Id'] = df_raw['StartStation Id'].astype(int)
    df_raw = df_raw.loc[df_raw['Duration'] < duration_upper]

    # Check to confirm no unknown station value is present in the data which is not recorded in bike station details dataset.
    s_diff = set(df_raw['StartStation Id'].to_list()) - station_const_ids
    e_diff = set(df_raw['EndStation Id'].to_list()) - station_const_ids
    if len(s_diff) > 0:
        raise RuntimeError('In file {} unknown start station IDs: {}'.format(file_path, s_diff))
    if len(e_diff) > 0:
        raise RuntimeError('In file {} unknown end station IDs: {}'.format(file_path, e_diff))

    # Remove unwanted stations like virtual stations
    df_raw = df_raw.loc[~df_raw['StartStation Id'].isin(station_exclude)]
    df_raw = df_raw.loc[~df_raw['EndStation Id'].isin(station_exclude)]

    # Count the number of bike trips made for station-to-station.
    df_graph_weight = df_raw.groupby(['StartStation Id', 'EndStation Id']).size()

    # Maps time of bike trip to a starting time of the interval, as defined by the dataset created by time_bins().
    df_raw['End Date Lower Bound'] = df_raw['End Date'].apply(round_minutes,
                                                              **{'direction': 'down', 'resolution': interval_size})
    df_raw['Start Date Lower Bound'] = df_raw['Start Date'].apply(round_minutes,
                                                                  **{'direction': 'down', 'resolution': interval_size})


    #Merge df_raw dataset with time interval dataset obtained from the time_bins().
    df1 = pd.merge(df_raw, tinterval, left_on='End Date Lower Bound', right_on='inner_limit').drop(
        columns=['inner_limit', 'outer_limit'])
    df1 = df1.rename({'interval_id': 'End Date ID'}, axis=1)
    df1 = pd.merge(df1, tinterval, left_on='Start Date Lower Bound', right_on='inner_limit').drop(
        columns=['inner_limit', 'outer_limit'])
    df1 = df1.rename({'interval_id': 'Start Date ID'}, axis=1)
    df1 = df1.drop(columns=['bike_id', 'Duration', 'passholder_type','plan_duration','start_lon','start_lat',
                            'End Date','Start Date', 'trip_route_category','bike_type','end_lat', 'end_lon',
                            'End Date Lower Bound', 'Start Date Lower Bound'])
    # Raises an error
    if len(df1) == 0:
        raise RuntimeError('Incorrect time intervals for file {}'.format(file_path))

    g_arrival = df1.groupby(by=['EndStation Id', 'End Date ID'])
    df_arrival = g_arrival.size()
    g_departure = df1.groupby(by=['StartStation Id', 'Start Date ID'])
    df_departure = g_departure.size()

    # Zero is substituted in place of records with missing value of time ids,
    s1 = df_arrival.index.get_level_values('EndStation Id').unique()
    s2 = df_departure.index.get_level_values('StartStation Id').unique()
    s_index = s1.union(s2)
    t1 = df_arrival.index.get_level_values('End Date ID').min()
    t2 = df_departure.index.get_level_values('Start Date ID').max()
    t_index = pd.Int64Index(list(range(t1, t2 + 1)), dtype='int64')
    new_index = pd.MultiIndex.from_product([s_index, t_index],
                                           names=['station_id', 'time_id'])
    df_arrival = pd.DataFrame(df_arrival.reindex(new_index, fill_value=0), columns=['arrivals'])
    df_departure = pd.DataFrame(df_departure.reindex(new_index, fill_value=0), columns=['departures'])

    df_all = df_arrival.join(df_departure, how='outer')

    return df_all, df_graph_weight
```

Figure 3: format_data function

3. "merge_data" function combines individual formatted files into a single file and returns the complete station Id list (figure 4).

4. "make_graph_weights" function uses the weights generated from "format_data" function to create a new features "average percent" which is used to assign weights to edges (figure 5).

```python
def merge_data(file_name_root, max_file_index):

    drop_index = None
    df_all = []

    #iterating every files to remove duplicate element
    for ind in range(max_file_index):
        df = pd.read_csv('{}_{}.csv'.format(file_name_root, ind), index_col=(0, 1))
        for ind_ in range(ind + 1, max_file_index):
            print('Processing file pair {}-{} of {}'.format(ind, ind_, max_file_index))
            df1 = pd.read_csv('{}_{}.csv'.format(file_name_root, ind_), index_col=(0, 1))
            df2 = df.join(df1, how='inner', lsuffix='_0', rsuffix='_1')

            # If time-place coordinate overlap found put coordinates in new common file
            if len(df2) > 0:
                df2['arrivals'] = df2['arrivals_0'] + df2['arrivals_1']
                df2['departures'] = df2['departures_0'] + df2['departures_1']
                df2 = df2.drop(['arrivals_0', 'arrivals_1', 'departures_0', 'departures_1'], axis=1)
                df_all.append(df2)

                if drop_index is None:
                    drop_index = df2.index
                else:
                    drop_index = drop_index.union(df2.index)

    # Drops duplicatte records and the collective data from all the files is saved.
    for ind in range(max_file_index):
        df = pd.read_csv('{}_{}.csv'.format(file_name_root, ind), index_col=(0, 1))

        if not drop_indeces is None:
            df = df[~df.index.isin(drop_indeces)]
        df_all.append(df)

    df_all = pd.concat(df_all)
    df_all.reset_index(level=0, inplace=True)
    df_all.reset_index(level=0, inplace=True)

    df_all.to_csv('/content/drive/My Drive/data_int_{}/'.format(t_interval)+ 'metro_formated_data.csv', index=False)

    return list(df_all['station_id'].unique())
```

Figure 4: merge_data function

```python
def make_graph_weights(graphs, stations_include, norm='median'):

    # Create index that contains all station pairs that ever appears in the raw data files
    # and apply this index to all graphs.
    mega_index = graphs[0].index
    for graph in graphs[1:]:
        mega_index = mega_index.union(graph.index)
    g_expand = [g.reindex(mega_index, fill_value=0) for g in graphs]
    cat_graph = pd.concat(g_expand).reset_index()

    cat_graph = cat_graph.loc[cat_graph['StartStation Id'].isin(stations_include)]
    cat_graph = cat_graph.loc[cat_graph['EndStation Id'].isin(stations_include)]

    cat_graph_all = cat_graph.groupby(['StartStation Id', 'EndStation Id']).sum()
    cat_graph_all = cat_graph_all.rename(columns={0: 'total'})

    all_df = []
    #Create another feature in the form of percentage of usage.
    for s_ind, df_slice in cat_graph_all.groupby('StartStation Id'):
        mean_slice = df_slice['total'].sum()
        df_slice.loc[:, 'average_percent'] = 100.0 * df_slice['total'] / mean_slice
        all_df.append(df_slice)
    cat_graph_all = pd.concat(all_df)

    return cat_graph_all
```

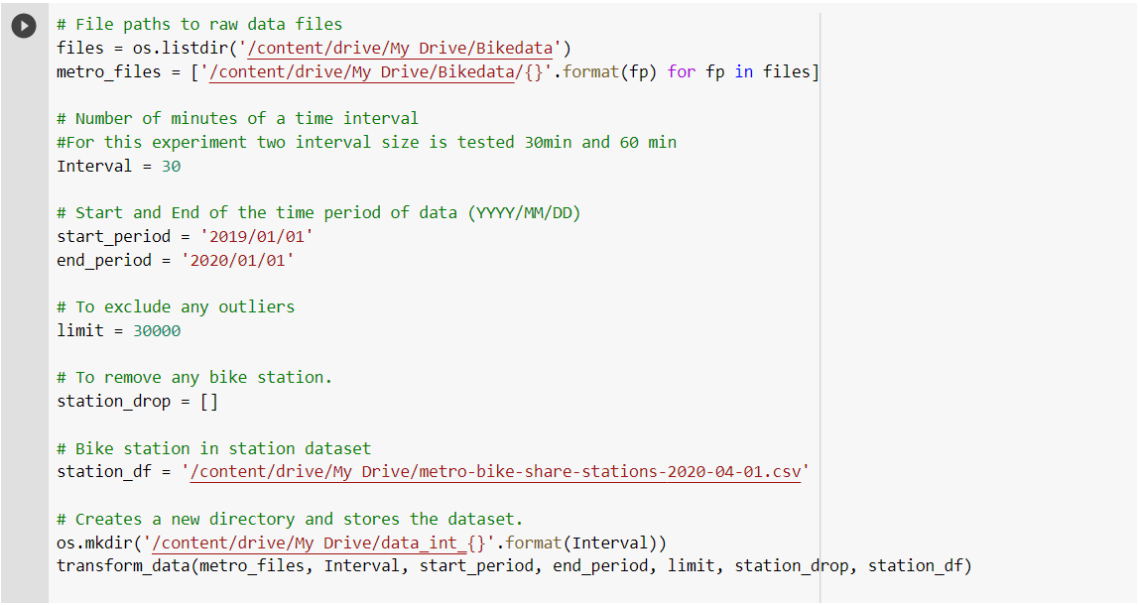
Figure 5: make_graph_weights function

5. "transform_data" function as shown in figure 6 is the parent function which calls all the above functions and saves the datasets formed into a directory in google drive.

```
def transform_data(raw_data_file_paths, time_interval_size, lower_t_bound, upper_t_bound,
        duration_upper, stations_exclude, station_filename):
    # Create a time interval id mapping to real-world times
    tinterval = time_bins(lower_t_bound, upper_t_bound, time_interval_size)

    tinterval.to_csv('/content/drive/My Drive/data_int_{}/tinterval.csv'.format(time_interval_size))

    # Load station dataset
    stations_const = pd.read_csv(station_filename)
    stations_const_ids = set(stations_const['Station_ID'].to_list())

    # Format multiple bike trip data files
    graphs = []
    for k, file_path in enumerate(raw_data_file_paths):
        print('Processing... {}'.format(file_path))
        spatiotemp, graph_weights = format_data(file_path, tinterval, time_interval_size,
                                                duration_upper, stations_exclude, stations_const_ids)
        spatiotemp.to_csv('/content/drive/My Drive/data_int_{}/data_reformat_{}.csv'.format(time_interval_size,k))
        graphs.append(graph_weights)

    stations_include = merge_data('/content/drive/My Drive/data_int_{}/data_reformat'.format(time_interval_size), k + 1, time_interval_size)
    print(stations_include)

    # Put together the graph weight data
    df_graph_all = make_graph_weights(graphs, stations_include)
    print(df_graph_all)
    df_graph_all.to_csv('/content/drive/My Drive/data_int_{}/graph_weight.csv'.format(time_interval_size))
```

Figure 6: transform_data function

Finally, the values of variables are passed to the "transform_data" function to start the execution process as shown in the figure 7.



```
# File paths to raw data files
files = os.listdir('/content/drive/My Drive/Bikedata')
metro_files = ['/content/drive/My Drive/Bikedata/{}'.format(fp) for fp in files]

# Number of minutes of a time interval
#For this experiment two interval size is tested 30min and 60 min
Interval = 30

# Start and End of the time period of data (YYYY/MM/DD)
start_period = '2019/01/01'
end_period = '2020/01/01'

# To exclude any outliers
limit = 30000

# To remove any bike station.
station_drop = []

# Bike station in station dataset
station_df = '/content/drive/My Drive/metro-bike-share-stations-2020-04-01.csv'

# Creates a new directory and stores the dataset.
os.mkdir('/content/drive/My Drive/data_int_{}'.format(Interval))
transform_data(metro_files, Interval, start_period, end_period, limit, station_drop, station_df)
```

Figure 7: Variables used to generate transformed datasets

3.3 Data Preparation

1. For ARIMA and LSTM Model the following data preparation followed in the research. The datasets obtained from processing are utilised to generate individual bike station dataset for both bike demands and docks demands as shown in figure 8.

```
interval = 60
```

```
# Loading datasets obtained from pre-processing stage.
tinterval_df = pd.read_csv("/content/drive/My Drive/data_int_{}/tinterval.csv".format(interval))
data_df= pd.read_csv("/content/drive/My Drive/data_int_{}/data.csv".format(interval))
```

```
#Joining both dataframes
time_df = data_df.merge(tinterval_df, how= 'left', left_on='time_id', right_on= 'interval_id')
time_df = time_df.drop(columns= ['time_id', 'Unnamed: 0','interval_id','outer_limit'])
time_df = time_df.rename(columns={'inner_limit' : 'date_time'})
time_df = time_df[['date_time', 'station_id', 'docks_demand', 'bikes_demand']]
time_df['date_time'] = pd.to_datetime(time_df['date_time'])
```

```
#Removing Virtual Station and Fake Stations from Data
print(time_df.shape)
time_df = time_df[time_df.station_id != 3000]
time_df = time_df[time_df.station_id != 4285]
time_df = time_df[time_df.station_id != 4286]
```

```
# Sorting and reindexing columns
time_df.sort_values(by ='date_time')
time_df.reset_index()
```

```
# Obtain values for station 3005's bike and docks demand separately.
df_3005 = time_df[time_df.station_id == 3005]
df_3005_bike = df_3005[['date_time', 'bikes_demand']]
df_3005_docks = df_3005[['date_time', 'docks_demand']]
df_3005_bike.set_index('date_time', inplace=True)
df_3005_docks.set_index('date_time', inplace=True)
print(df_3005_bike.head(2))
print(df_3005_docks.head(2))
```

Figure 8: Data preparation for ARIMA and LSTM models

2. However, for STGCN and TAGCN model graph data is generated using Pytorch Geometric library. And libraries shown in figure 9 are mandatory for the process. And To ease the impplementation following functions are developed.

```
import torch
import torch.nn.functional as F
from torch_geometric.data import Data
import os
from os import listdir, makedirs, remove
import os.path as osp
import pandas as pd
from datetime import datetime, timedelta
import numpy as np
from numpy.random import Generator, PCG64
from torch_geometric.data import Dataset, DataLoader
from torch_geometric.data import Data
import torch.nn.functional as funct
```

Figure 9: Libraries necessary for graph data preparation

- "make_edges" function converts the graph weight dataset into edge data compatible with Pytorch geometric (figure 10).

```python
def make_edges(station_exclusion):

  df_gw = pd.read_csv('/content/drive/My Drive/data_int_60/graph_weight.csv')
  df_gw['StartStation Id'] = df_gw['StartStation Id'].astype(int)
  df_gw['EndStation Id'] = df_gw['EndStation Id'].astype(int)
  df_gw['weight'] = df_gw['average_percent'].astype(float)
  df_gw = df_gw[['StartStation Id', 'EndStation Id', 'weight']]

  if not station_exclusion is None:
    df_gw = df_gw.loc[~df_gw['StartStation Id'].isin(station_exclusion)]
    df_gw = df_gw.loc[~df_gw['EndStation Id'].isin(station_exclusion)]

# Create map between station ids and node id, needed because Pytorch geometric reindex nodes
  all_ids = set(df_gw['StartStation Id'].unique()).union(set(df_gw['EndStation Id'].unique()))
  node_id_2_station_id_map = dict((n, sid) for n, sid in enumerate(all_ids))
  station_id_2_node_id_map = {v: k for k, v in node_id_2_station_id_map.items()}

  # Put data in format expected by Pytorch
  index = torch.tensor([[station_id_2_node_id_map[k] for k in df3['StartStation Id']],
                        [station_id_2_node_id_map[k] for k in df3['EndStation Id']]], dtype=torch.long)
  attr = torch.tensor(df3['weight'].tolist(), dtype=torch.float)

  return index, attr, station_id_2_node_id_map
```

Figure 10: make_edges function

- "time_windows" functions slice the dependent features from the data and for it to be converted into tensors as shown in figure 11.

```python
def time_windows(df, sample_size,t_shuffle, time_input_number,
                 time_forward_pred, station_id_2_node_id_map,t_start, t_end):

  stride = None
  if sample_size is None and t_shuffle is False:
      t_val_iter = range(t_start, t_end)

  elif (not sample_size is None) and t_shuffle:
      seed_rg = rg = Generator(PCG64())
      t_val_iter = seed_rg.choice(range(t_start, t_end), size=sample_size, replace=False)

  null_list = [0] * time_input_number
  for t_val in t_val_iter:
    df_x = df.loc[(df['time_id'] < t_val + time_input_number) & (df['time_id'] >= t_val)]
    df_y = df.loc[df['time_id'] == t_val + time_input_number + time_forward_pred - 1]

    if len(df_x) == 0 or len(df_y) == 0:
      raise RuntimeError('Encountered missing time period')

    # Move from pandas to torch tensor compatible data
    data_xt = [[null_list, null_list]] * len(station_id_2_node_id_map)
    for station_id, chunk in df_x.groupby('station_id'):
      ind = station_id_2_node_id_map[station_id]
      data_xt[ind] = [chunk['arrivals'].tolist(),chunk['departures'].tolist()]

    data_yt = [[0, 0]] * len(station_id_2_node_id_map)
    for station_id, chunk in df_y.groupby('station_id'):
      ind = station_id_2_node_id_map[station_id]
      data_yt[ind] = [chunk['arrivals'].item(), chunk['departures'].item()]

    yield data_xt, data_yt
```

Figure 11: time_windows function

- Finally, "create_torch_data" function creates a graph structure of nodes and edges from the transformed bikes data (figure 12).

3.4 Modelling

- To train the model 80% of data has been utilised in all the models and rest 20% is used for training. In the case of LSTM model 20% of data from train set id used for validation.

- The model has been implemented as following:

```python
def create_torch_data(source_data_files, station_exclusion, time_id_bounds,
                      time_input_number, time_forward_pred, sample_size, t_shuffle,
                      store_dir):

  edge_index, edge_attr, station_id_2_node_id_map = make_edges(station_exclusion)
  processed_file_names = []
  for raw_file_name in source_data_files:
    df = pd.read_csv(raw_file_name)
    # Apply filters on data to situate in the graph
    if not station_exclusion is None:
      df = df.loc[~df['station_id'].isin(station_exclusion)]
    if not time_id_bounds is None:
      df = df.loc[(df['time_id'] >= time_id_bounds[0]) & \
                  (df['time_id'] <= time_id_bounds[1])]
    # In the event that filters removed all data, jump to next iteration
    if len(df) == 0:
      continue

    slice_generator = time_windows(df, sample_size,t_shuffle, time_input_number, time_forward_pred,
                                   station_id_2_node_id_map, t_start=df['time_id'].min(),
                                   t_end=(df['time_id'].max() - time_input_number - time_forward_pred))

    for count, (data_g_xt, data_g_yt) in enumerate(slice_generator):
      data_graph = Data(x=torch.tensor(data_g_xt, dtype=torch.float),
                        y=torch.tensor(data_g_yt, dtype=torch.float),
                        edge_index=edge_index, edge_attr=edge_attr)
      torch.save(data_graph,  store_dir + '/time_slice' + '_{}.pt'.format(count))
      processed_file_names.append('{}_{}.pt'.format('time_slice', count))

  return station_id_2_node_id_map
```

Figure 12: create_torch_data function

1. ARIMA Model
   The ARIMA model is implemented using Statsmodel library. To utilise the data for ARIMA stationary test has also been performed and model is implemented as shown in figure 13.

```python
#Identifying ideal parameter value of p,d and q with lowest AIC value

df_aic = pd.DataFrame(columns = ['Parameters', 'AIC'])
param_list =[]
aic_list=[]
p=q=range(0,5)
d=range(1,2)
pdq = list(itertools.product(p,d,q))
for param in pdq:
  try:
    model_arima = ARIMA(train,order=param)
    model_arima_fit = model_arima.fit()
    param_list.append(param)
    aic_list.append(model_arima_fit.aic)
    print(param,model_arima_fit.aic)
  except:
    continue
df_aic['Parameters'] = param_list
df_aic['AIC'] = aic_list
```

```python
[ ] #Training the Model
    model_arima = ARIMA(train, order= df_aic.Parameters[df_aic.AIC.idxmin()])
    model_arima_fit = model_arima.fit()
    print(model_arima_fit.aic)
```

```python
[ ] # Prediction for test set
    predictions= model_arima_fit.forecast(steps=len(test))[0]
    print(len(predictions))
```

Figure 13: ARIMA model train and test implementation

2. LSTM Model
   A five LSTM model has been implemented using Keras library and the data is scaled using MinMaxScaler before training. The code can be referred from figure 14.

```
model = Sequential()
#Adding the first LSTM layer and some Dropout regularisation
model.add(LSTM(units = 50, return_sequences = True, input_shape = (X_train.shape[1], 1)))
model.add(Dropout(0.2))
# Adding a second LSTM layer and some Dropout regularisation
model.add(LSTM(units = 50, return_sequences = True))
model.add(Dropout(0.2))
# Adding a third LSTM layer and some Dropout regularisation
model.add(LSTM(units = 50, return_sequences = True))
model.add(Dropout(0.2))
# Adding a fourth LSTM layer and some Dropout regularisation
model.add(LSTM(units = 50))
model.add(Dropout(0.2))
# Adding the output layer
model.add(Dense(units = 1))

model.summary()
```

```
[ ] model.compile(optimizer = 'adam', loss = 'mean_squared_error')
    # Fitting the model to the Training set
    history = model.fit(X_train, y_train, epochs = 30, batch_size = 32, validation_split= 0.2,
                        callbacks = EarlyStopping(monitor='val_loss', patience = 5))
```

Figure 14: LSTM model implementation

3. STGCN Model
   The implementation of STGCN model is a intricate process and was referred from (Ohrn; 2020). The model consists of 4 classes which are referenced in the main "STGCN" class. The figure 15 shows the 3 individual classes and figure 16, the main STGCN class.

```
class SpatioTemporal(torch.nn.Module):
  def __init__(self, n_spatial, n_temporal, channel_inputs, channel_outputs):
    super(SpatioTemporal, self).__init__()
    self.n_spatial = n_spatial
    self.n_temporal = n_temporal
    self.channel_inputs = channel_inputs
    self.channel_outputs = channel_outputs
```

```
[ ] class Time1dConvGLU(SpatioTemporal):
      def __init__(self, n_spatial, n_temporal, channel_inputs, channel_outputs, time_convolution_length):
        super(Time1dConvGLU, self).__init__(n_spatial, n_temporal, channel_inputs, channel_outputs)
        self.one_d_conv_1 = torch.nn.Conv1d(in_channels=self.channel_inputs,
                                            out_channels=2 * self.channel_outputs,
                                            kernel_size=time_convolution_length,
                                            groups=1)
        n_temporal_out = self.n_temporal - time_convolution_length + 1
        assert n_temporal_out > 0

      def forward(self, x):
        total = []
        for k_node, tensor_node in enumerate(x.split(1, dim=IDX_SPATIAL)):
          pq_out = self.one_d_conv_1(tensor_node)
          y_conv_and_gated_out = F.glu(pq_out, dim=1)
          total.append(y_conv_and_gated_out)

        nodes_spatial = torch.cat(total, dim=IDX_SPATIAL)
        return nodes_spatial
```

```
[ ] class SpatialGraphConv(SpatioTemporal):
      def __init__(self, n_spatial, n_temporal, channel_inputs, channel_outputs, graph_conv_kwargs):
        super(SpatialGraphConv, self).__init__(n_spatial, n_temporal, channel_inputs, channel_outputs)

        self.gcn = GCNConv(self.channel_inputs, self.channel_outputs)

      def forward(self, data):
        assert self.channel_inputs == data.x.shape[IDX_CHANNEL]
        x_s = data.x
        total = []
        for k_t in range(data.x.shape[IDX_TEMPORAL]):
          x_t = x_s[:,:,k_t]
          y_t = self.gcn(x_t, data.edge_index, data.edge_attr)
          y_t = F.relu(y_t)
          total.append(y_t)

        nodes_times = torch.stack(total, dim=IDX_TEMPORAL)
        return nodes_times
```

Figure 15: Three convolution layer classes of the STGCN model

```python
class STGCN(torch.nn.Module):
  def __init__(self, n_temporal_dim, n_spatial_dim, n_input_channels,
               co_temporal=64, co_spatial=16, time_conv_length=3,
               graph_conv_kwargs={}):
    super(STGCN, self).__init__()

    self.n_temporal = n_temporal_dim
    self.n_spatial = n_spatial_dim
    self.n_input_channels = n_input_channels
    self.co_temporal = co_temporal
    self.co_spatial = co_spatial

    self.permute_norm = (IDX_CHANNEL, IDX_SPATIAL, IDX_TEMPORAL)
    assert n_temporal_dim - 4 * time_conv_length + 4 == 1

    self.model_t_1a = Time1dConvGLU(n_spatial_dim, n_temporal_dim,
                                    channel_inputs=n_input_channels,
                                    channel_outputs=co_temporal,
                                    time_convolution_length=time_conv_length)
    self.model_s_1 = SpatialGraphConv(n_spatial_dim, n_temporal_dim - time_conv_length + 1,
                                      channel_inputs=co_temporal,
                                      channel_outputs=co_spatial,
                                      graph_conv_kwargs=graph_conv_kwargs)
    self.model_t_1b = Time1dConvGLU(n_spatial_dim, n_temporal_dim - time_conv_length + 1,
                                    channel_inputs=co_spatial,
                                    channel_outputs=co_temporal,
                                    time_convolution_length=time_conv_length)
    self.layer_norm_1 = torch.nn.LayerNorm([n_spatial_dim, n_temporal_dim - 2 * time_conv_length + 2])

    self.model_t_2a = Time1dConvGLU(n_spatial_dim, n_temporal_dim - 2 * time_conv_length + 2,
                                    channel_inputs=co_temporal,
                                    channel_outputs=co_temporal,
                                    time_convolution_length=time_conv_length)
    self.model_s_2 = SpatialGraphConv(n_spatial_dim, n_temporal_dim - 3 * time_conv_length + 3,
                                      channel_inputs=co_temporal,
                                      channel_outputs=co_spatial,
                                      graph_conv_kwargs=graph_conv_kwargs)
    self.model_t_2b = Time1dConvGLU(n_spatial_dim, n_temporal_dim - 3 * time_conv_length + 3,
                                    channel_inputs=co_spatial,
                                    channel_outputs=co_temporal,
                                    time_convolution_length=time_conv_length)
    self.layer_norm_2 = torch.nn.LayerNorm([n_spatial_dim, n_temporal_dim - 4 * time_conv_length + 4])

    self.model_output = torch.nn.Sequential(torch.nn.Linear(in_features=n_spatial_dim * co_temporal,
                                                             out_features=n_spatial_dim * n_input_channels),
                                            torch.nn.ReLU(),
                                            torch.nn.Linear(in_features=n_spatial_dim * n_input_channels,
                                                            out_features=n_spatial_dim * n_input_channels))
```

```python
  def forward(self, data_graph):
    data_step_0 = data_graph
    edge_index = data_graph.edge_index
    edge_attr = data_graph.edge_attr
    self.n_graphs_in_batch = self._infer_batch_size(data_step_0.x)

    data_step_0_x = data_step_0.x
    data_step_1_x = self.model_t_1a(data_step_0_x)
    data_step_1 = Data(data_step_1_x, edge_index, edge_attr)
    data_step_2_x = self.model_s_1(data_step_1)
    data_step_3_x = self.model_t_1b(data_step_2_x)
    data_step_3_x = self._compute_layer_norm(data_step_3_x, self.layer_norm_1)

    data_step_4_x = self.model_t_2a(data_step_3_x)
    data_step_4 = Data(data_step_4_x, edge_index, edge_attr)
    data_step_5_x = self.model_s_2(data_step_4)
    data_step_6_x = self.model_t_2b(data_step_5_x)
    data_step_6_x = self._compute_layer_norm(data_step_6_x, self.layer_norm_2)

    data_out_reshape = data_step_6_x.reshape(self.n_graphs_in_batch, self.n_spatial * self.co_temporal)
    data_output_x = self.model_output(data_out_reshape)
    data_output_x = data_output_x.reshape(self.n_graphs_in_batch * self.n_spatial, self.n_input_channels)

    return Data(y=data_output_x, edge_index=edge_index, edge_attr=edge_attr)

  def _infer_batch_size(self, data_x):
    if data_x.shape[IDX_SPATIAL] % self.n_spatial == 0:
      n_batches = data_x.shape[IDX_SPATIAL] // self.n_spatial

    else:
      raise RuntimeError('Incorrect batch size')

    return n_batches

  def _compute_layer_norm(self, data_x, norm_func):
    ret = []
    for k_batch in range(self.n_graphs_in_batch):
      data_x_pergraph = data_x.narrow(IDX_SPATIAL, k_batch * self.n_spatial, self.n_spatial)
      data_x_pergraph_norm = norm_func(data_x_pergraph.permute(self.permute_norm)).permute(self.permute_norm)
      ret.append(data_x_pergraph_norm)

    return torch.cat(ret)
```

Figure 16: Main Class of STGCN model

4. TAGCN Model
   Implementation of TAGCN model is similar to STCGN model, with the only difference in the "SpatialGraphConv" class in the main TAGCN class. Figure 17 represents the SpatialGraphConv class in TAGCN.

```
class SpatialGraphConv(SpatioTemporal):
  def __init__(self, n_spatial, n_temporal, channel_inputs, channel_outputs, graph_conv_kwargs):
    super(SpatialGraphConv, self).__init__(n_spatial, n_temporal, channel_inputs, channel_outputs)

    self.gcn = TAGConv(self.channel_inputs, self.channel_outputs, **graph_conv_kwargs)

  def forward(self, data):
    assert self.channel_inputs == data.x.shape[IDX_CHANNEL]
    x_s = data.x
    total = []
    for k_t in range(data.x.shape[IDX_TEMPORAL]):
      x_t = x_s[:,:,k_t]
      y_t = self.gcn(x_t, data.edge_index, data.edge_attr)
      y_t = F.relu(y_t)
      total.append(y_t)

    nodes_times = torch.stack(total, dim=IDX_TEMPORAL)
    return nodes_times
```

Figure 17: SpatialGraphConv class in TAGCN model

3.5 Evaluation

To evaluate the models, three metrics; RMSE, MSE and MAE are utilised which are implemented using sklearn library. The code can be referenced through figure 18

```
#Evaluation
print('Evaluation')
print( "Mean Squareed Error: ", mean_squared_error(dataset_test_docks.values,predicted_docks_demand))
print("Mean Absolute Error: ", mean_absolute_error(dataset_test_docks.values,predicted_docks_demand))
print( "Root Mean Square Error: ", np.sqrt(mean_squared_error(dataset_test_docks.values,predicted_docks_demand)))
```

```
Evaluation
Mean Squareed Error:  1.530671222670672
Mean Absolute Error:  0.7023111926098259
Root Mean Square Error:  1.2372029836169456
```

Figure 18: Evaluation metrics used in the research.

References

Bar-El, O. (2018). Getting the most out of your google colab.
URL: *https://medium.com/@oribarel/getting-the-most-out-of-your-google-colab-2b0585f82403*

Ohrn, A. (2020). London bike ride forecasting with graph convolutional networks.
URL: *https://towardsdatascience.com/london-bike-ride-forecasting-with-graph-convolutional-networks-aee044e48131*