Predicting the Impact of Weather and Holidays on Recreation Centre Traffic

Emiliano Penaloza

Western University

Statistics 3860

April 30, 2020

Predicting the Impact of Weather and Holidays on Recreation Centre Traffic

Considering the importance of fitness to student health and productivity, a better understanding of the factors which impact students' use of recreation facilities is gained through a case study of the Western Student Recreation Centre (WSRC). Due to a lack of existing data on the WSRC's hourly traffic, a dataset is created by compiling hourly traffic figures obtained from the facility's *Twitter* page "Western Weight Room" (Figure 1). This data is subsequently used to build a predictive model targeted at predicting total hourly traffic at the WSRC. In this paper, two factors, weather and holidays, are considered to impact the response variable, traffic. Intuitively, poor weather conditions may result in less traffic since people tend to prefer engaging in physical activity when weather conditions are more favourable. Furthermore, traffic may decrease on holidays since a large proportion of students leave proximity to the WSRC. Considering that the data set is a time series, in order to obtain the most accurate estimates of these predictors a Recurrent Neural Network (RNN) is used. Using an RNN for time series data allows use of memory cells which can recall previous observations and extract key trends from past data entries. For reference, Hastie, Friedman, and Tisbshirani (2017) is used as a guide.

Figure 1

Example Tweet with Hourly Traffic Figures ("WE" is "weight room," "CM" is "cardio mezzanine")

Web Scraping

Web scraping was performed to obtain the WSRC traffic figures from the tweets, using the package *tweepy* (Figure 2). Extraction of only the 3220 of the most recent tweets was possible, given tweet archiving limitations on *Twitter*, which restricted data compilation to the most recent seven months. Additionally, the COVID-19 global pandemic has restricted data compilation to the most recent six months. The total amount of data entries decreased to 3217 after deleting non-traffic tweets.

Figure 2

Example Entry of Scraped Tweet

	date	tweets	CM	WR
0	2020-03-15 20:37:58	WR 47 CM 12	12	47

The individual figures for WR and CM in each tweet were added to obtain the total traffic in an hour. Hourly weather data was scraped using the package *Selenium*; hourly entries for time, temperature, dew, humidity, wind direction, windspeed, wind gust, precipitation, and condition were collected for the most recent twelve months.[1] The extraction data was subsequently converted into a *Python* dictionary and exported as a callable pickle object for later use.

Data Preparation

For modelling purposes, the weather data was changed into a *Pandas* data frame in order to merge with the *Twitter* data (Figure 3).

Figure 3

Formatted Weather Data Frame

	Temperature	Dew	Humidity	Wind	WindSpeed	WindGust	Pressure	Precip	Condition
Time									
2015-01-01 01:00:00	18 F	3 F	53 %	WSW	20 mph	32 mph	29.03 in	0.0 in	Cloudy
2015-01-01 02:00:00	18 F	5 F	58 %	WSW	20 mph	30 mph	29.00 in	0.0 in	Mostly Cloudy
2015-01-01 03:00:00	18 F	7 F	63 %	WSW	18 mph	25 mph	29.00 in	0.0 in	Cloudy
2015-01-01 04:00:00	18 F	9 F	68 %	SW	16 mph	23 mph	29.00 in	0.0 in	Cloudy

[1] For in-depth detail of data extraction, see Appendix A.

Varying time ranges between entries in the *Twitter* data presented a problem since weather data entries occurred in hourly timestamps. This problem was addressed by rounding time down to the nearest half-hour. For example, a timestamp of 12:14:00 would be rounded down to 12:00:00. Additionally, timestamps for both data frames were converted into datetime objects in the format “%Y-%m-%d %H:%M:%S,” using *Python*’s datetime package. After completing the pre-processing, a left join of the tweets data frame onto the weather data frame was performed. As a result, all of the *Twitter* data remained intact and also held a weather entry (Figure 4).

Figure 4

Joint Data Frame of Weather Data and Twitter Data

	date	tweets	CM	WR	Temperature	Dew	Humidity	Wind	WindSpeed	WindGust	Pressure	Precip	Condition	Total
0	2020-03-15 16:30:00	WR 47 CM 12	12	47	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	59
1	2020-03-15 13:30:00	WR 52 CM 22 🎾	22	52	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	74
2	2020-03-15 12:30:00	WR 48 CM 24 😛	24	48	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	72
3	2020-03-15 11:30:00	WR 23 CM 13	13	23	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	36
4	2020-03-15 11:00:00	WR 19\nCM 6 🏃🏿‍♀️😊👍🏼👟😷	6	19	32 F	16 F	51 %	NE	8 mph	0 mph	29.60 in	0.0 in	Fair	25

Since the timestamps for the *Twitter* data were irregular while those for the weather data were in hourly increments, missing values in some weather entries existed due to the merger. Backfilling was performed on the data to completely eliminate missing values (Figure 5). Since the data entries are hourly, there is little concern for replacing a weather entry with one which is between one and two hours in the future.

Figure 5

Filled Data Frame of Weather Data and Twitter Data

	date	tweets	CM	WR	Temperature	Dew	Humidity	Wind	WindSpeed	WindGust	Pressure	Precip	Condition	Total
0	2020-03-15 16:30:00	WR 47 CM 12	12	47	32 F	16 F	51 %	NE	8 mph	0 mph	29.60 in	0.0 in	Fair	59
1	2020-03-15 13:30:00	WR 52 CM 22 🎾	22	52	32 F	16 F	51 %	NE	8 mph	0 mph	29.60 in	0.0 in	Fair	74
2	2020-03-15 12:30:00	WR 48 CM 24 😛	24	48	32 F	16 F	51 %	NE	8 mph	0 mph	29.60 in	0.0 in	Fair	72
3	2020-03-15 11:30:00	WR 23 CM 13	13	23	32 F	16 F	51 %	NE	8 mph	0 mph	29.60 in	0.0 in	Fair	36
4	2020-03-15 11:00:00	WR 19\nCM 6 🏃🏿‍♀️😊👍🏼👟😷	6	19	32 F	16 F	51 %	NE	8 mph	0 mph	29.60 in	0.0 in	Fair	25

Next, weather values were converted into numeric values since they were extracted as strings (Figure 6). Multiple convertor functions were created to simplify tasks[2](Figure 7).

Figure 6

Numeric Data Frame

	date	CM	WR	Temperature	Dew	Humidity	WindSpeed	Pressure	Precip	Total
0	2019-09-29 12:00:00	17.000000	59.000000	13.888889	10.000000	0.77	13.0	29.39	0.0	76.0
1	2019-09-29 13:00:00	16.666667	61.333333	15.000000	8.888889	0.67	15.0	29.38	0.0	78.0
2	2019-09-29 14:00:00	12.000000	54.000000	15.000000	8.888889	0.67	18.0	29.35	0.0	66.0
3	2019-09-29 15:00:00	15.000000	52.000000	16.111111	7.777778	0.59	16.0	29.35	0.0	67.0
4	2019-09-29 16:00:00	21.000000	63.000000	16.111111	7.777778	0.59	16.0	29.35	0.0	84.0

Figure 7

Stripping Function for Temperature

```
def F_strip(val):
    return Celcius(float(val.strip('F')) )
```

Pandas' *groupby* function was used to turn the data into hourly entries since before multiple entries existed for some timestamp. The mean value of each hourly entry was taken and then used as a single row entry. With the data in a consistent format, the holidays parameter was added to the data. Since most holidays occur on certain Mondays of months containing such holidays, the holiday parameter was created by flagging certain days around the corresponding holiday as a positive entry. The three days prior to the official start date of the holiday and the two days after the official end date of the holiday were flagged. The dates for official holidays were obtained as per those listed as an "important date" in Western University's 2019-2020 Academic Calendar.

[2] For in-depth detail of convertor functions see Appendix E.3.

Next, the data was converted into a format appropriate for modelling. Time series models require a continuous time stream, meaning an entry at every interval of time for the given start and end range. Hourly data entries resulted in entries existing for each hour from September 2019 to March 2020 (see Appendix for additional detail).

The creation of missing values resulted from the addition of a continuous time stream to the data. A closed column was implemented to address the issue of missing values; the entry was equal to one if the WSRC was closed and zero otherwise. In addition, an interaction term between the holiday and the closed columns was created, allowing the model to differentiate between a holiday for which the facility remains open and those for which it is closed. Finally, the data was split into a training set, a test set, and a validation set. The training set consisted of 70% of the total observations while the test set and the validation set consisted of 15% each. The final data-frame is presented in Figure 10.

Figure 10

Final Data-Frame

date	CM	WR	Temperature	Dew	Humidity	WindSpeed	Pressure	Precip	Total	holidays	closed	closedxholidays
2020-02-19 10:00:00	12.0	28.0	-5.0	-11.0	1.0	15.0	29.0	0.0	40.0	1	0	0
2020-02-19 11:00:00	12.0	33.0	-5.0	-11.0	1.0	15.0	29.0	0.0	45.0	1	0	0
2020-02-19 12:00:00	18.0	34.0	-4.0	-10.0	1.0	10.0	29.0	0.0	52.0	1	0	0
2020-02-19 13:00:00	11.0	42.0	-4.0	-10.0	1.0	13.0	29.0	0.0	54.0	1	0	0
2020-02-19 14:00:00	19.0	51.0	-5.0	-11.0	1.0	12.0	29.0	0.0	70.0	1	0	0

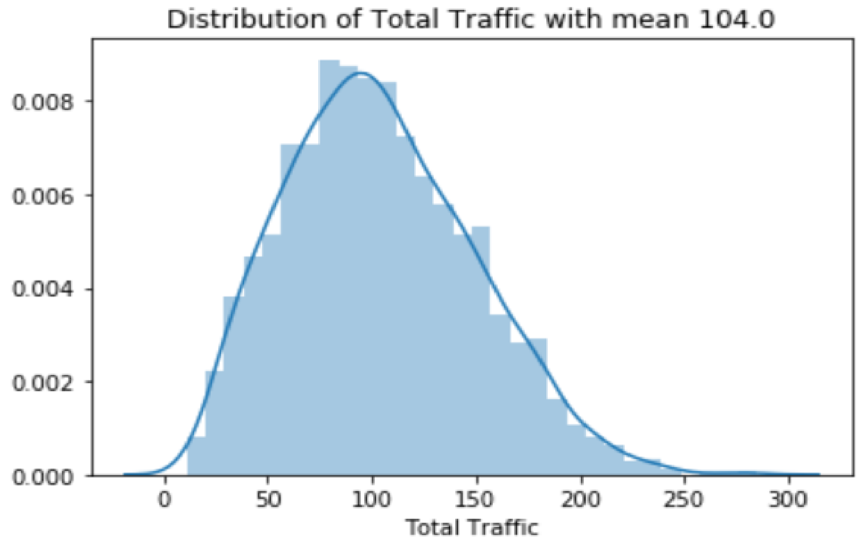
Results

Data Visualization

The distribution of total traffic was plotted (Figure 11). The distribution of traffic appears fairly normally distributed with a very slight right-skew; the small deviation from the mean makes it ideal for modelling.

Figure 11

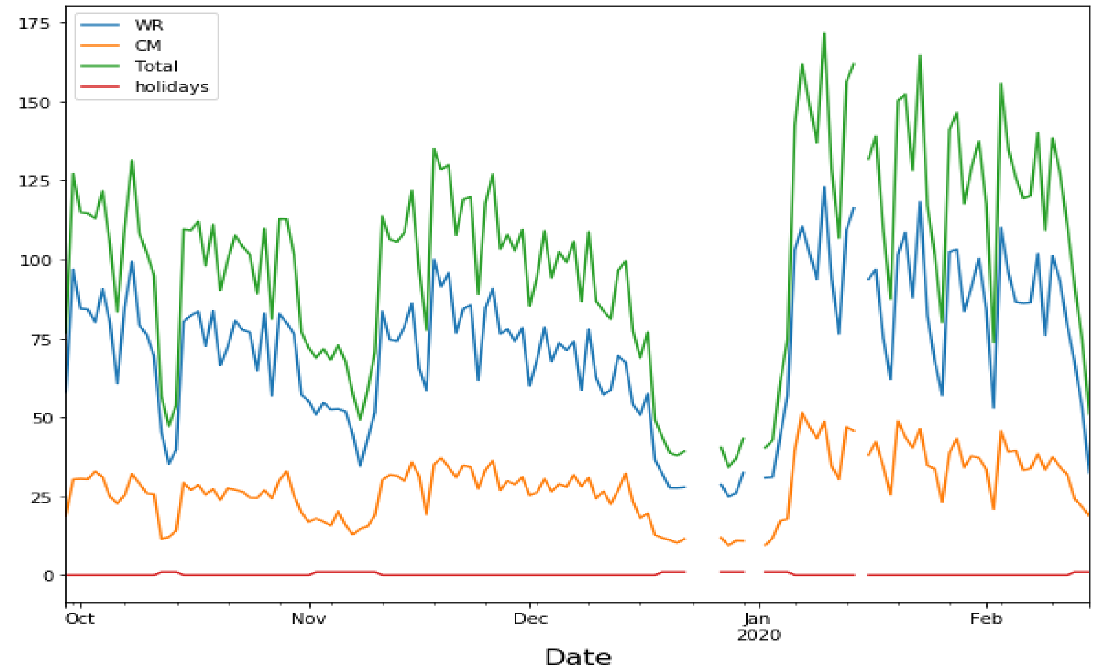
Plot of the Distribution of Total Traffic



Moreover, the traffic over time when accounting for holidays was plotted to detect the effectiveness of the holiday parameter (Figure 12). The dips in traffic appear to be extracted correctly by the parameter. Furthermore, traffic seems to be fairly constant over the fall term, with a slight average increase over the spring term. The spring term trend could be due to "New Year's resolutions," where people commit to improve their fitness in the New Year, resulting in individuals visiting recreation facilities more frequently. Thus, this could be a seasonal factor that could be implemented in the future with greater data compilation.

Figure 12

Traffic Data Over Training Set Range

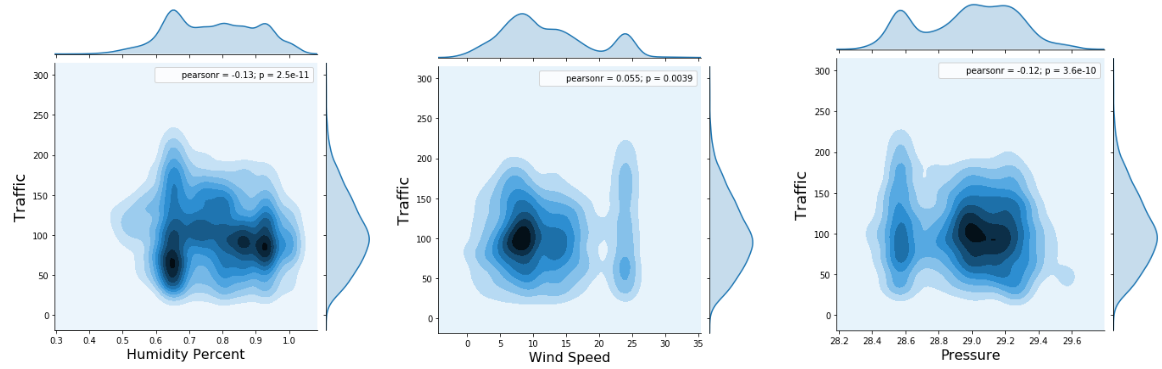


Next, various weather variables were explored, the first being the impact of temperature on total traffic (Figure 13). There appears to be no correlation between traffic and temperature, however, a more condensed spread of traffic appears when the temperature approaches extreme values. Specifically, traffic tends to slow down as temperature falls to negative values, but traffic consistency appears to peak at moderate temperatures. Overall, temperature does not appear to have a great impact on total traffic for most of the data set, which is supported by an overall Pearson correlation of approximately 0.1 with a significant p-value.

Following, the impacts of humidity percent, wind speed, and pressure were observed (Figure 14). Overall, wind speed appears to be a very weak determinant for total traffic, as the joint distribution fails to exhibit any meaningful trend. Although higher correlation values are observed in humidity percent and pressure, these values are too low to make any claims. In summary, humidity percent, win speed, and pressure appear to have insignificant impact on total traffic.

Figure 14

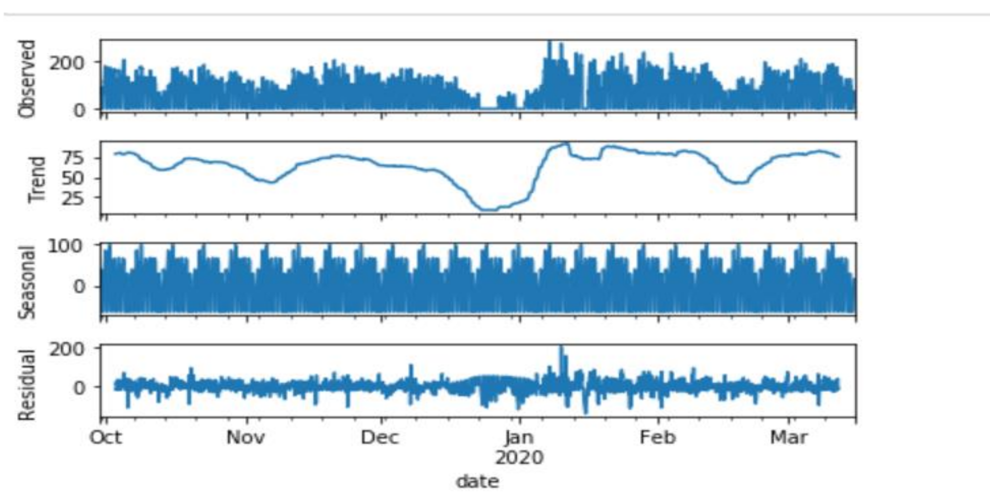
Joint Distributions of Traffic and Weather Predictors



Finally, traffic data was decomposed in order to visualize weekly trends (Figure 15). The data appears to follow a constant trend with the only anomaly occurring during December where traffic dips as expected. A very frequent seasonal factor is observed, indicating that traffic is predictable based on the day of the week. Furthermore, a small deviation in the residuals is observed, as they hold a tight fit around zero. These results indicate the series is stable and predictable, which is ideal for modelling.

Figure 15

Time Series Decomposition of Overall Traffic with a Relative Frequency of 168 Steps (Hourly Weekly Steps)



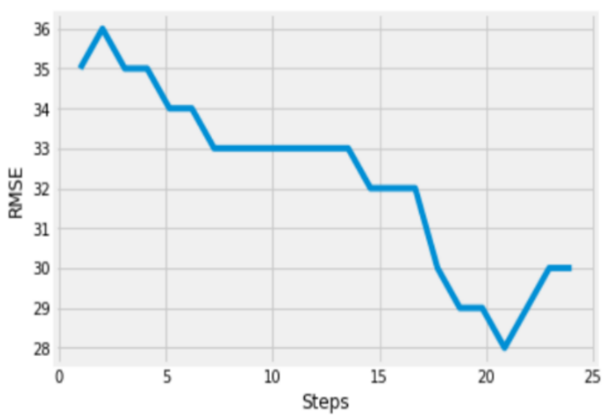
Modelling

A RNN was used for modelling as these networks can identify key trends in data by using memory cells, which are able to decompose patterns in data. This task requires special lag variables at varying time steps in order to remember data. The function *series_to_supervise* was used to create these lag variables (Brownlee, 2017). This function allowed for the specification of how many steps the lag variable should take, meaning if only one step was specified then only one lag variable for each predictor would be created. Meanwhile, if three steps were specified, each variable would have an entry for the last three entries in the data. Then, the function *grid_search* was created to address the flexibility and uncertainty as to what specific lag to use. This function iterated over different amounts of timesteps and returned a list of the root mean squared error for each timestep for a given model.

The first model tested was a simple RNN with a single Long Short-Term Memory (LSTM) layer with a batch size of 32 and 50 input cells (Figure 16). The *grid_search* function was used to search for the best timestep over the validation set. The lowest validation RMSE is observed to be 28 at 21 timesteps, which is not optimal for a response with a mean of 104.

Figure 16

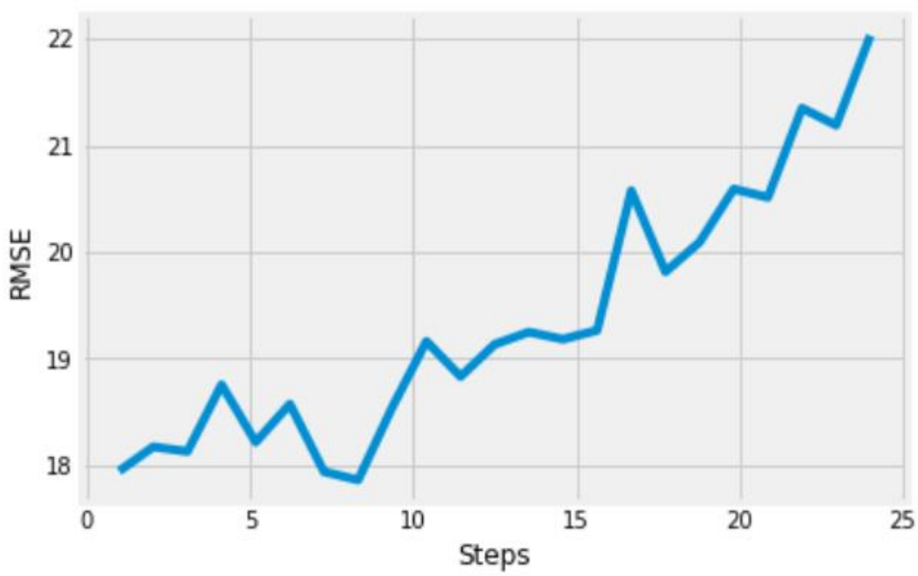
RMSE of Varying Timesteps for Single Layer NN with Batch Size of 32



Next, the batch size was increased to observe possible enhanced model performance (Figure 17). The results are observed to improve by decreasing RMSE by a factor of 11 with eight timesteps. A dropout layer was added to address the issue of overfitting the model when using the larger batch size.

Figure 17

RMSE of Varying Timesteps for Single Layer NN with Batch Size of 74



Regularization does not appear to have a large impact on RMSE but there appears to be a more constant trend with less dips, which could be due to this change (Figure 18). As a result, the dropout layer was kept. Thus far, the most important factor in reducing RMSE is increasing batch size. To explore if a constant increase in batch size would decrease RMSE, the batch size was increased from 74 to 148 in the regularized model (Figure 19).

Figure 18

RMSE of Varying Timesteps for a Regularized Singly Layer NN with Batch Size of 74

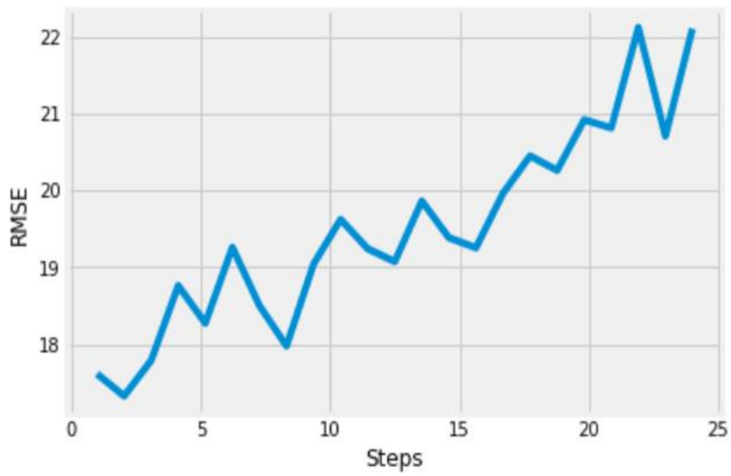
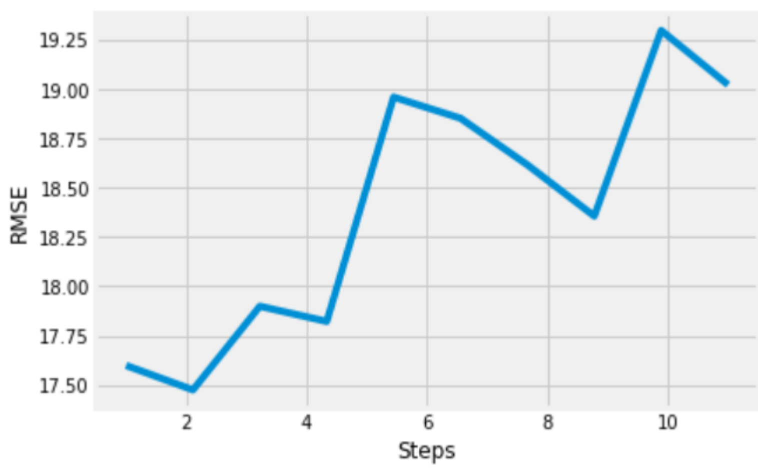


Figure 19

Regularized Model with Batch Size of 148

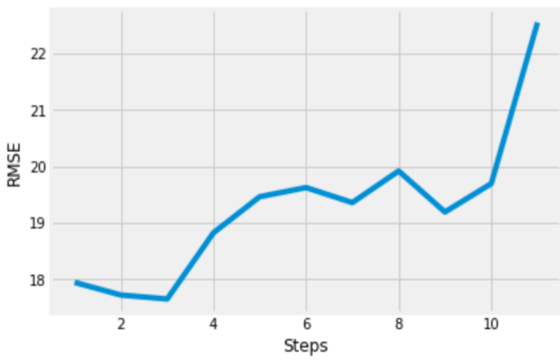


Due to restricted computational power, the timestep range was decreased to 1-11. The increasing batch size did not appear to have a large impact on RMSE, thus it may be concluded that a batch size of 74 is sufficient.

An RNN with one hidden layer was the final model to be tested (Figure 20). The input layer had 50 input units, like the other models, while the hidden layer had 100 input units. The batch size was 74 and all other parameters were left unchanged. The addition of a new layer did not appear to improve performance and as a result, the final model that will be used for prediction is the regularized single layer RNN with a batch size of 74.

Figure 20

Multilayer Model with batch size of 74



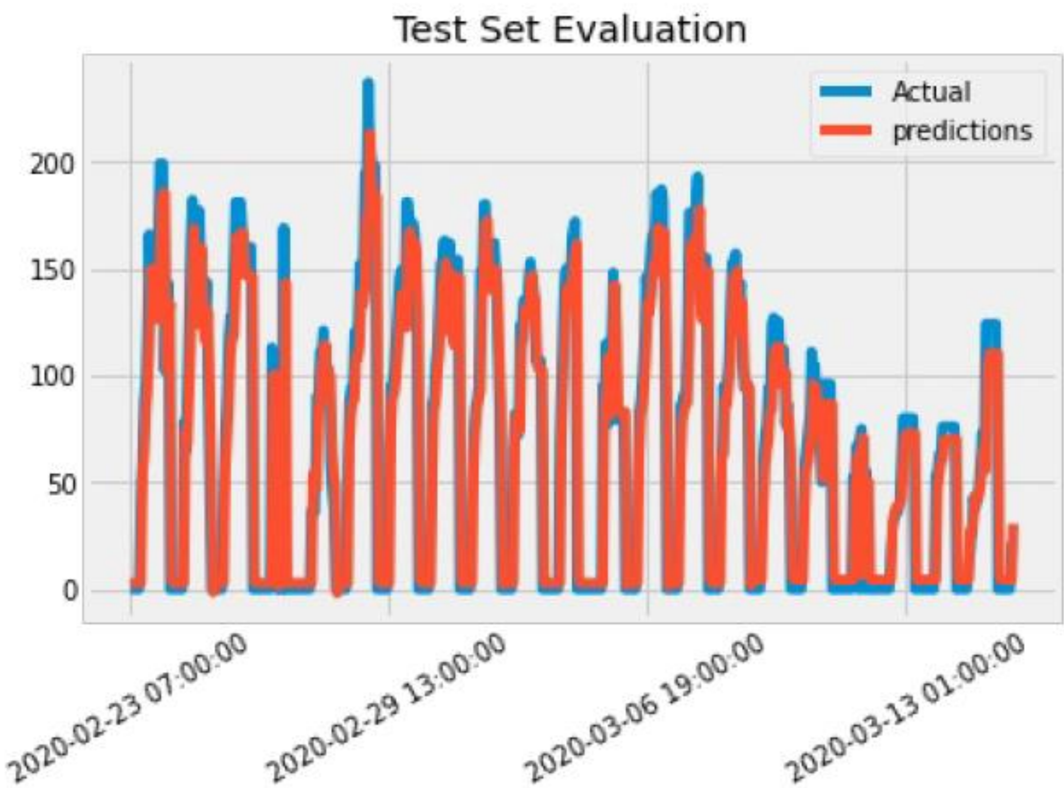
Since the regularized single layer RNN with a batch size of 74 achieved the lowest RMSE on the validation set, it is used going forward and evaluated using the test set. The actual values against the predicted values for the test set are plotted, as well as bootstrapped values of test RMSE(Figure 21) ; Unfortunately, test set statistics do seem to deviate from those in the validation set (Figure 22). Although, the majority of the deviation in error does seem to be during peak hours and closing hours. Overall, the model achieved an average RMSE of 32.7(Figure 21), which is not on par with the validation values.

Figure 21

Box Plot of 50 Bootstrapped Test Set RMSE Values

Figure 21

Test Set Predictions Plotted Against Actual Values



Conclusion

The predictions of the WSRC traffic data over different time frames, although they deviate in unseen data, were fairly accurate. Improvements to the method used in the paper could involve optimization of a deep neural network, which would likely be able to identify trends more effectively than a single layer and produce more accurate results. To achieve the previously stated more data would be needed. More data would allow for further decomposition of weekly, monthly and yearly trends.

References

Brownlee, J. (2017). *Deep Learning for Time Series Forecasting*. Machine Learning Mastery.

Hastie, T., Freidman, J., and Tisbshirani R. (2017). The Elements of Statistical Learning. Springer Publishing.

Western University. (2019). *Key Dates 2019-20*. Statistical and Actuarial Sciences. Accessed April 23, 2020. https://www.uwo.ca/stats/undergraduate/key-dates.html.

Appendix A

A.1 Importing Libraries

In [1]:

```
import tweepy
import pandas as pd
from tweepy import Cursor
import pytz
import numpy as np
```

A.2 Tweepy set up

In [2]:

```
#my keys should put them in another file and hide them before putting this on git
API_key = "7M1xKUhbvU4NITzZAoeeuSBZb"
API_secret_key = "ksSxd57SkrL6kzg1PM5uZ0TDLvUz6u7078Im2KqL8bBKpxc168"
access_token = "705069617295790080-2fmdiurkwA4MrclmfrcXbCZOPmwspEc"
access_token_secret = "mAYVj092P1KLTyZUjuOm2cBWGgy1OAYQYrFZhOFmW2K5N"
```

In [3]:

```
# Setting up the tweet extraction
auth  = tweepy.OAuthHandler(API_key,API_secret_key)
auth.set_access_token(access_token,access_token_secret)
api = tweepy.API(auth)
weightroom_id  = "WesternWeightRm"
```

In [4]:

```
#specifying weight room twitter id as user to be extracted
rec_user  = api.get_user(weightroom_id)
timeline = rec_user.timeline()
```

A.2 Tweet Retrival

In [5]:

```
#Using tweepy interfact to fetch tweets
#using build in interfact to format dates and tweets into python lists
tweets = Cursor(api.user_timeline, id=weightroom_id).items()
dates = []
text = []
for tweet in tweets:
        dates.append(tweet.created_at)
        text.append(tweet.text)
```

In [6]:

```
#Function deEmojify
#input
#InputString
def deEmojify(inputString):
    return inputString.encode('ascii', 'ignore').decode('ascii')
```

In [7]:

```
#Iterating through tweet and date list to make a pandas dataframe
#Uses demojify  to remove emojis from tweets
#Uses regular expresions to find only number values from tweets then storing them into numbers_wr
```

```
and numbers_cm lists
#creates final dataframe and makes a list of not used tweets
import string
import re
data = pd.DataFrame({'date' : dates, 'tweets' : text})
no_emoji= []
numbers_wr = []
numbers_cm = []
u = []
not_used = []
for i in range(len(data.iloc[:,1]) ):
    emoji_free  = deEmojify(data.iloc[i,1])
    emoji_free = emoji_free.replace("&amp;","")
    emoji_free = emoji_free.replace("\n" , "")
    values  = re.findall(r'(\w*\d+)',emoji_free)

    if(len(values) == 2):
        numbers_wr.append(values[0])
        numbers_cm.append(values[1])

    else:
        not_used.append(data.iloc[i,1])
    #if(nums == None ):
     #   u.append(emoji_free)
```

A.3 Data Exportation

In [8]:

```
#Dropping tweets to not use and making CM and WR value columns using values obtained before
to_drop = data[data.tweets.isin( not_used)].index
clean_data = data.drop(to_drop)
clean_data['CM'] =  numbers_cm
clean_data['WR'] =  numbers_wr
```

In [10]:

```
clean_data.head()
```

Out[10]:

	date	tweets	CM	WR
0	2020-03-15 20:37:58	WR 47 CM 12	12	47
1	2020-03-15 17:55:33	WR 52 CM 22 □	22	52
2	2020-03-15 16:34:29	WR 48 CM 24 😀	24	48
3	2020-03-15 15:55:35	WR 23 CM 13	13	23
4	2020-03-15 15:23:41	WR 19\nCM 6 □□♀😊□□□😷	6	19

In [9]:

```
clean_data.to_csv("clean_tweeets.csv" )
```

Appendix B

B.1 Importing Libraries

In [1]:

```
from selenium import webdriver
import pandas as pd
import numpy as np
from datetime import *
import pickle
```

B.2 Specifying Chrome as Web Driver

In [2]:

```
#Experimenting with the packages
driver  = webdriver.Chrome()
```

B.3 Data Retrival Functions

In [24]:

```
#Main functions used for web scrapping



#Function list_maker
#input column HTML column to convert into a python list
#outputs python list of strings for the specified HTML column
def list_maker(column):
    make = []
    for i in range(len(column)):
        make.append(column[i].text)
    return make

#WebsSraper
#inputs
#Date specific date to get data from
#driver specified selenium web driver to fetch data fram
#will use the driver element to refresh the current webpage and find the table within the page.
#It will go through each column and fetch data for each row
#Then it will use lsit maker to make an appropriate list for every element stored in the table
#Outputs a dictionary with all the column headers and their values for that specific date
def WeatherScraper(Date, driver):
    driver.get('https://www.wunderground.com/history/daily/ca/london/CYXU/date/{0}'.format(Date))
    table = driver.find_element_by_xpath('//*[@id="inner-
content"]/div[2]/div[1]/div[5]/div[1]/div/lib-city-history-observation/div/div[2]/table')
    headers = table.find_elements_by_xpath('//tr[2]/td')
    rData = []
    col = np.arange(1,20)
    for i in col:
        col = table.find_elements_by_xpath("//tr/td["+str(i)+"]")
        rData.append(col)
    Time = list_maker(rData[0])
    Temperature = list_maker(rData[1])
    DewPoint =list_maker(rData[2])
    Humidity = list_maker(rData[3])
    Wind=list_maker(rData[4])
    WindSpeed =list_maker(rData[5])
    WindGust=list_maker(rData[6])
    Pressure = list_maker(rData[7])
    Precip = list_maker(rData[8])
    Condition = list_maker(rData[9])

    temp_clean = Temperature[22:]
```

```
    time_clean = Time[22:]
    dew_clean =  DewPoint[22:]
    humidity_clean = Humidity[6:]
    Date = {'Temperature':temp_clean,'Time' :time_clean,'Dew':dew_clean, 'Humidity' : humidity_clea
n, 'Wind' :Wind
             ,'WindSpeed': WindSpeed,'WindGust' : WindGust,'Pressure' : Pressure,'Precip' : Precip,
'Condition' :Condition}
    return Date


#Function DateRangeCollector
#inputs
#start starting date
#end ending date
#file file name to save to
#This is how we itterate through the webpage in full.
#It will use function WebScrapper to collect the data for the given end and start range
#it will then dump the returning dictionary into a pickel file to be available for retrieving late
r.
#Trouble shooting is done within the function so if for some reason the data is unavailable when w
ebscrapper is trying to retrieve it
#this will dump the current dictionary into the file and print out a statement indicating what day
the error occured in to
#allow the proccess to be resumed in that day.
#Outputs day_dict a dictionary of a ll the weather data

def DateRangeCollector(start,end,file):
    try:
        f = open(f"{file}","wb")
        delta =  end - start
        day_dict  = {}
        driver  = webdriver.Chrome()
        driver.implicitly_wait(3)
        day = start
        for i in range (delta.days + 1):
            day = start + timedelta(i)
            string_format = day.date().isoformat()
            test = WeatherScraper(string_format,driver)
            day_dict[string_format] =  test
        pickle.dump(day_dict,f)
        f.close()
    except:
        print('Error occured during this this date {0}'.format(string_format) )
        pickle.dump(day_dict,f)
        f.close()


    return day_dict
```

B.4 Data Exportation

In [27]:

```
#Function calls to collect the days
start =datetime(2020,1,21)
end =datetime(2020,4,1)
dictionary_of_days = DateRangeCollector(start,end,'Weather2020_jan21')
```

Appendix C

C.1 Importing Libraries

In [2]:

```
import pickle
import pandas as pd
import numpy as np
from datetime  import *
import matplotlib.pyplot as plt
from collections import ChainMap
```

C.2 Data Merging

In [43]:

```
#Merge
#input two dictionaries
#outputs merged dictionaries
def Merge(dict1, dict2):
    res = {**dict1, **dict2}
    return res
```

In [54]:

```
#loading weather data  to clean it up
Data_0 = pickle.load(open('Weather_data_2020.pkl',"rb") )
Data_1 = pickle.load(open('Weather2020',"rb") )
Data_2 = pickle.load(open('Weather2020_jan21',"rb") )
Data =  Merge(Data_1,Data_2)
Data = Merge(Data_0,Data_2)
```

C.3 Data Formating

In [ ]:

```
##function timestamp_maker
#Input Data dictionary
#Outputs dataframe with datetime object as timesampts
#Iterates through dictionary first taking the time keys to making them a column as timestamps
#After making time column it iterates through other columns and appends them onto a pandas datafra
me

def timestamp_maker(Data):
    copy_data = Data.copy()
    keys = list(Data.keys())
    for j in range(len(Data)):
        day_data = Data.get(keys[j])
        time = day_data['Time']
        for i in range(len(time)):
            day = time[i] + " " + keys[j]
            copy_data[keys[j]]['Time'][i] = datetime.strptime(day,'%I:%M %p %Y-%m-%d')
            rows = []
    for i in range(len(copy_data) ):
        d = copy_data[keys[i]]
        rows.append( pd.DataFrame(dict([ (k,pd.Series(v)) for k,v in d.items() ])) )

    Weather = pd.concat(rows).set_index('Time')

        return Weather
Weather = timestamp_maker(Data)
```

C.4 Data Exportation

In [60]:

```
#saving weather dataframe as a csv file
Weather.to_csv('weather_data.csv')
```

Appendix D

D.1 Importing Libraries

In [91]:

```
import pandas as pd
import numpy as np
from datetime import *
import pickle as pkl
```

D.2 Importing Data

In [92]:

```
#reading in data from other files
tweets =pd.read_csv('clean_tweeets.csv').drop(['Unnamed: 0','Unnamed: 5','Unnamed: 6','Unnamed: 7'
                                                ,'Unnamed: 8','Unnamed: 9'],axis =1)
weather = pd.read_csv('weather_data')
```

In [93]:

```
tweets.head()
```

Out[93]:

	date	tweets	CM	WR
0	3/15/2020 20:37	WR 47 CM 12	12	47
1	3/15/2020 17:55	WR 52 CM 22 □	22	52
2	3/15/2020 16:34	WR 48 CM 24 😁	24	48
3	3/15/2020 15:55	WR 23 CM 13	13	23
4	3/15/2020 15:23	WR 19\nCM 6 □□♀😊□□□😷	6	19

D.3 Data Cleansing

In [94]:

```
#dropping rows with na values
weather_nona = weather[weather['Time'].notna()]
```

In [95]:

```
#Function rount_time
#inputs
#dt datetime object
#direction what way to rount_to as string up or down
#rount_to amount to rount_to to
#outputs rounded datetime object
def rount_to_time(dt, direction, rount_to):
    new_minute = (dt.minute // rount_to + (1 if direction == 'up' else 0)) * rount_to
    return dt + timedelta(minutes=new_minute - dt.minute)
```

In [96]:

```
#Formating the data into datetime format
time_format = []
for day in weather.Time.dropna():
    time_format.append( datetime.strptime(day, '%Y-%m-%d %H:%M:%S') )
```

In [97]:

```
#formating time data into datetime format
#I also adjusted the data for a timezone error and rounded up each timestamp to enarest 30 min
time = tweets.date
for i in range(len(time)):
    time[i] = round_time(datetime.strptime(time[i], '%m/%d/%Y %H:%M') -  timedelta(hours = 4),
                        'down',
                        30)
tweets.date  = time
```

```
C:\Users\Emiliano\Anaconda3\lib\site-packages\ipykernel_launcher.py:7: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  import sys
```

D.4 Data Exportation

In [98]:

```
#Merging the dataframes into one for easy access to all data
#Filling the missing values in the data as the weather repor came in hourly increments while the t
weets come in half hour basis
# I will be using hourly weather reports and just fill the weather report with the neaerest data
tweets.date = pd.to_datetime(tweets.date)
weather_nona.Time = pd.to_datetime(weather_nona.Time)
Data_missing = tweets.merge(weather_nona,how = 'left' , left_on = 'date',right_on = 'Time')
Data_missing.to_csv("Raw_Formated_Data.csv")
```

Appendix E

E.1 Importing Libraries

In [1]:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.ticker as plticker
from scipy.stats  import pearsonr
from datetime import *
import warnings
warnings.filterwarnings("ignore")
```

E.2 Loading and Formatting Dataset

In [2]:

```
raw= pd.read_csv("raw_Formated_Data.csv").drop(["Unnamed: 0",'Time'],axis =1)
raw['Total'] = raw['CM'] + raw['WR']
raw['holidays'] = pd.Series ( np.where( raw.WR == -1, 1, 0 ) )
raw.date = pd.to_datetime(raw.date)
df = raw.fillna(method = 'backfill')
```

E.3 Declaring Functions

In [3]:

```
#Function date_range
#inputs
#end end date for range
#start start date for range
#delta hourly change to take
#Ouputs range beggingingat start range with delta increments in timestamps

def date_range(end, start, delta):
    change = timedelta(hours =delta)
    time = []
    curr = start
    while curr < end:
        time.append(curr)
        curr+= change
    return time
```

In [4]:

```
#Function Holiday
#inputs
#day the specified day
#data specified data to get holiday from
#delta end amount of days after holiday to stop tagging as holiday
#delta start how many days before holiday date to start flagging as holiday
#outputs index in dataframe with holiday values
def holiday(day,data,delta_end = 1,delta_start = 2 ):
    end_range = pd.Timestamp(day + timedelta(days = delta_end) )

    start_range =pd.Timestamp(day - timedelta(days = delta_start) )

    holidays =   pd.Series(np.where( ( (end_range > data.date) & ( start_range < data.date) ) ,1,0)
)
    index   =holidays.index[holidays ==1 ].tolist()
    return index
```

In [5]:

```
#function celcius
#input F farenheit temperature value/s
#converts farenhait values to celcius

def Celcius(F):
    return (F - 32) * 5.0/9.0
```

In [6]:

```
#function strip_preassure
#input values
#outputs numeric values
def strip_preassure(val):
    return val.strip('in')
```

In [7]:

```
#function strip_precip
#input values
#outputs numeric values
def strip_precip(val):
    return float(val.strip('in'))
```

In [8]:

```
#function F_strip
#input values
#outputs numeric values
def F_strip(val):
    return Celcius(float(val.strip('F')) )
```

In [9]:

```
#function convert_to_percent
#input values
#outputs numeric values
def convert_to_percent(val):
    return float(val.strip('%'))/100
```

In [10]:

```
#function strip_mph
#input values
#outputs numeric values
def strip_mph(val):
    return float(val.strip('mph'))
```

In [11]:

```
#Function looper
#column column to convert
#fun specific function to use to convert
#outputs column numeric values
#loops through a series of values and converts them into numeric values
def looper(column, fun):
    j = 0
    for i in column:

        column[j] = fun(i)
        j+=1
    return column
```

In [12]:

```
#Holiday_appender
#frame dataframe input
#fun specific function to use to convert
```

```
#outputs column numeric values
#lAppends holiday column to specified dataframe with the holidays below

def holiday_appender(frame):
    data = frame.copy()
    holi_list = []
    christmas = holiday(datetime(year = 2019,month =12,day = 19) ,data,18,0)
    reading_week  = holiday(datetime(year = 2019,month =11,day = 4 ),data,7)
    thanks_giving = holiday(datetime(year = 2019, month = 10, day = 14),data)
    reading_week_spring = holiday(datetime(year = 2020, month = 2, day = 15),data,7)
    holi_list.append( (christmas, thanks_giving  ,reading_week_spring,reading_week) )
    for i in holi_list[0]:
        for index in i:
            data.holidays[index] = 1
    return data
```

In [13]:

```
#Function Filler
#input df dataframe to be filled
#iterates throught dataframe filling missing values
#if the rec is closed it will fill all values with zero
#if the rec is open and there is a missing values it will fill the dataframe with the most recent
valid entry
#operating hours are indicated by boolean statements weekend weekend_hour regular and regular_hour


def filler(df):
    grouped = df.copy()
    for f,i in enumerate(grouped.Total.index):

        weekend =  (grouped.date[i].dayofweek ==  5 | grouped.date[i].dayofweek ==  4)
        weekend_hour = (grouped.date[i].hour > 6 & grouped.date[i].hour <= 20 )
        regular =  (grouped.date[i].dayofweek !=  5 | grouped.date[i].dayofweek !=  4)
        regular_hour = (grouped.date[i].hour > 6 & grouped.date[i].hour <= 23 )

        if(pd.isna(grouped.Total[i]) & (weekend) & (weekend_hour)):


            grouped.Temperature[i] = grouped.Temperature[i-1]
            grouped.Dew[i] = grouped.Dew[i-1]
            grouped.Humidity[i] = grouped.Humidity[i-1]
            grouped.WindSpeed[i] = grouped.WindSpeed[i-1]
            grouped.Pressure[i] = grouped.Pressure[i-1]
            grouped.Precip[i] = grouped.Precip[i-1]
            grouped.CM[i] = grouped.CM[i-1]
            grouped.WR[i] = grouped.WR[i-1]
            grouped.Total[i] = grouped.CM + grouped.WR

        elif(pd.isna(grouped.Total[i]) & (regular) & (regular_hour)):

            grouped.Total[i] = grouped.Total[i - 1]
            grouped.Temperature[i] = grouped.Temperature[i-1]
            grouped.Dew[i] = grouped.Dew[i-1]
            grouped.Humidity[i] = grouped.Humidity[i-1]
            grouped.WindSpeed[i] = grouped.WindSpeed[i-1]
            grouped.Pressure[i] = grouped.Pressure[i-1]
            grouped.Precip[i] = grouped.Precip[i-1]
            grouped.CM[i] = grouped.CM[i-1]
            grouped.WR[i] = grouped.WR[i-1]
            grouped.Total[i] = grouped.CM[i] + grouped.WR[i]


        elif(pd.isna(grouped.Total[i])):

            grouped.Total[i] = 0
            grouped.Total[i] = 0
            grouped.Temperature[i] = 0
            grouped.Dew[i] = 0
            grouped.Humidity[i] = 0
            grouped.WindSpeed[i] = 0
            grouped.Pressure[i] = 0
            grouped.Precip[i] = 0
            grouped.CM[i] = 0
            grouped.WR[i] = 0
            grouped.Total[i] = 0
```

```
    return grouped
```

E.4 Object Conversion

In [14]:

```
#Converting traffic values to numeric

datetime_df = holiday_appender(df)
datetime_df.date = pd.to_datetime(datetime_df.date)
datetime_df.CM  = pd.to_numeric(datetime_df['CM'] )
datetime_df.WR = pd.to_numeric(datetime_df['WR'] )
datetime_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3217 entries, 0 to 3216
Data columns (total 15 columns):
date            3217 non-null datetime64[ns]
tweets          3217 non-null object
CM              3217 non-null int64
WR              3217 non-null int64
Temperature     3217 non-null object
Dew             3217 non-null object
Humidity        3217 non-null object
Wind            3217 non-null object
WindSpeed       3217 non-null object
WindGust        3217 non-null object
Pressure        3217 non-null object
Precip          3217 non-null object
Condition       3217 non-null object
Total           3217 non-null int64
holidays        3217 non-null int32
dtypes: datetime64[ns](1), int32(1), int64(3), object(10)
memory usage: 364.6+ KB
```

The following lines of code are using the previous functions to convert string values in the dataframe to numeric values

In [15]:

```
datetime_df.Pressure = looper(datetime_df.Pressure, strip_preassure)
datetime_df.Pressure = pd.to_numeric(datetime_df['Pressure'] )
```

In [16]:

```
datetime_df.Dew = looper(datetime_df.Dew, F_strip)
datetime_df.Dew = pd.to_numeric(datetime_df['Dew'] )
```

In [17]:

```
datetime_df.Humidity = looper(datetime_df.Humidity, convert_to_percent)
datetime_df.Humidity = pd.to_numeric(datetime_df['Humidity'] )
```

In [18]:

```
datetime_df.Temperature = looper(datetime_df.Temperature,F_strip)
datetime_df.Temperature = pd.to_numeric(datetime_df['Temperature'] )
```

In [19]:

```
datetime_df.WindSpeed = looper(datetime_df.WindSpeed,strip_mph)
datetime_df.WindSpeed = pd.to_numeric(datetime_df['WindSpeed'] )
```

In [20]:

```
datetime_df.Precip = looper(datetime_df.Precip,strip_precip)
datetime_df.Precip = pd.to_numeric(datetime_df['Precip'] )
```

E.5 Time Series Formatting

In [21]:

```
grouped= datetime_df.groupby(pd.Grouper(key = 'date' ,freq="H")).mean()
grouped = grouped.reset_index()
```

In [22]:

```
grouped
```

Out[22]:

	date	CM	WR	Temperature	Dew	Humidity	WindSpeed	Pressure	Precip	Total	holidays
0	2019-09-29 12:00:00	17.000000	59.000000	13.888889	10.000000	0.77	13.0	29.39	0.0	76.0	0.0
1	2019-09-29 13:00:00	16.666667	61.333333	15.000000	8.888889	0.67	15.0	29.38	0.0	78.0	0.0
2	2019-09-29 14:00:00	12.000000	54.000000	15.000000	8.888889	0.67	18.0	29.35	0.0	66.0	0.0
3	2019-09-29 15:00:00	15.000000	52.000000	16.111111	7.777778	0.59	16.0	29.35	0.0	67.0	0.0
4	2019-09-29 16:00:00	21.000000	63.000000	16.111111	7.777778	0.59	16.0	29.35	0.0	84.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...
4032	2020-03-15 12:00:00	24.000000	48.000000	0.000000	-8.888889	0.51	8.0	29.60	0.0	72.0	0.0
4033	2020-03-15 13:00:00	22.000000	52.000000	0.000000	-8.888889	0.51	8.0	29.60	0.0	74.0	0.0
4034	2020-03-15 14:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4035	2020-03-15 15:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4036	2020-03-15 16:00:00	12.000000	47.000000	0.000000	-8.888889	0.51	8.0	29.60	0.0	59.0	0.0

4037 rows × 11 columns

In [23]:

```
#grouping data into hourly frequencies for constant time entries

grouped= datetime_df.groupby(pd.Grouper(key = 'date' ,freq="H")).mean().round()
grouped = grouped.reset_index()

#converting data into a time series using date_range function
time_series = date_range(grouped.date[len(grouped.date)-1],grouped.date[0],1)
time_df = pd.DataFrame({'date':time_series})
time_df = time_df.merge(grouped, left_on='date', right_on='date')
```

In [24]:

```
#using filler function to fill in missing values then using holiday function to fill in holiday co
lumn

time_df = filler(time_df)
time_df['holidays'] = pd.Series ( np.where( time_df.WR == -1, 1, 0 ) )
time_df = holiday_appender(time_df)
```

In [25]:

```
#Creating closed column by specifying to when the total traffic equals zero
time_df['closed'] = pd.Series( np.where( time_df.Total == 0, 1, 0 ) )
time_df['closedxholidays'] = pd.Series( np.where( (time_df.closed == 1) & (time_df.holidays == 1),
1, 0) )
```

E.6 Dataset Splitting

In [26]:

```
#Creating date ordered training, testing and validation sets
#Training corresponds to first 70% of observations
#validation set corresponds 15% of observations after training set
#test set consists of last 15% of observations

train_ts = time_df[:int(time_df.shape[0]*0.85)]
test_ts = time_df[int(time_df.shape[0]*0.85):]
validation_ts = train_ts[int(train_ts.shape[0]*0.85):]
train_ts = train_ts[:int(train_ts.shape[0]*0.85) ]


train = datetime_df[int(datetime_df.shape[0]*0.15):]
test = datetime_df[:int(datetime_df.shape[0]*0.15)]
```

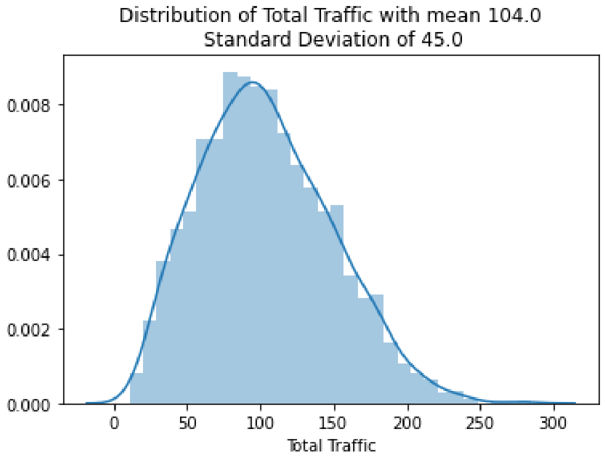
E.7 Data Vizualization

In [30]:

```
#Plotting distribution of rec centre traffic
ax = sns.distplot(train['Total'])
ax.set_title(f'Distribution of Total Traffic with mean {np.round(np.mean(train.Total))}\n Standard
Deviation of {np.round(np.std(train.Total))}')
ax.set_xlabel('Total Traffic', fontsize=10)
```

Out[30]:

```
Text(0.5, 0, 'Total Traffic')
```

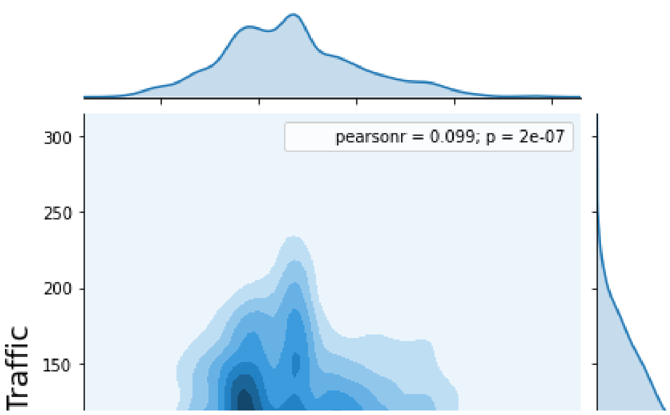


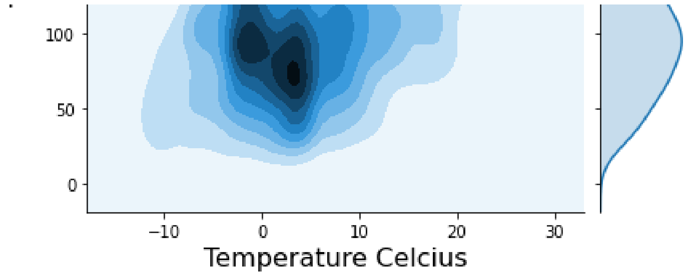
In [31]:

```
#plotting joint distribution of Temperature and Total
ax = sns.jointplot(x = 'Temperature',y = 'Total',data = train, kind = 'kde', stat_func = pearsonr)
ax.ax_joint.set_ylabel('Traffic', fontsize=18)
ax.ax_joint.set_xlabel('Temperature Celcius', fontsize=16)
```

Out[31]:

```
Text(0.5, 32.99999999999995, 'Temperature Celcius')
```



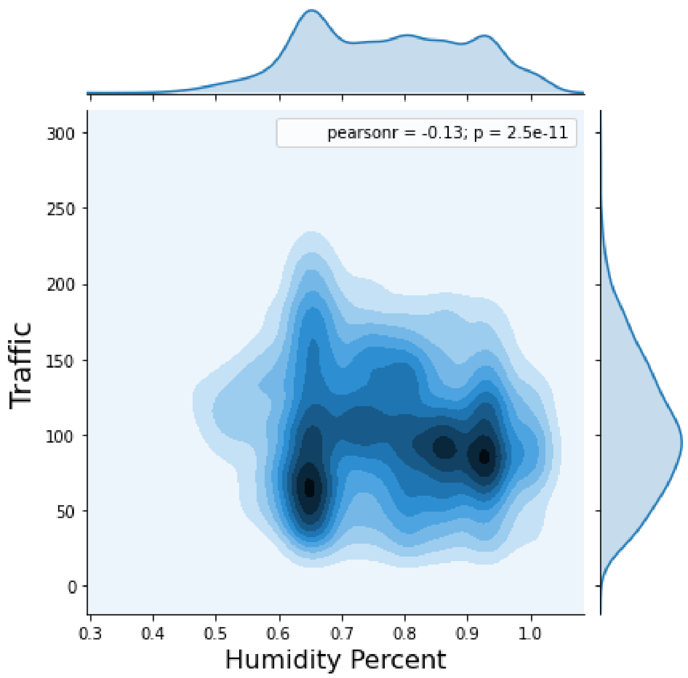


In [32]:

```
#plotting joint distribution of humidity and Total
ax = sns.jointplot(x = 'Humidity',y = 'Total',data = train, kind = 'kde',stat_func = pearsonr)
ax.ax_joint.set_ylabel('Traffic', fontsize=18)
ax.ax_joint.set_xlabel('Humidity Percent', fontsize=16)
```

Out[32]:

```
Text(0.5, 32.99999999999995, 'Humidity Percent')
```

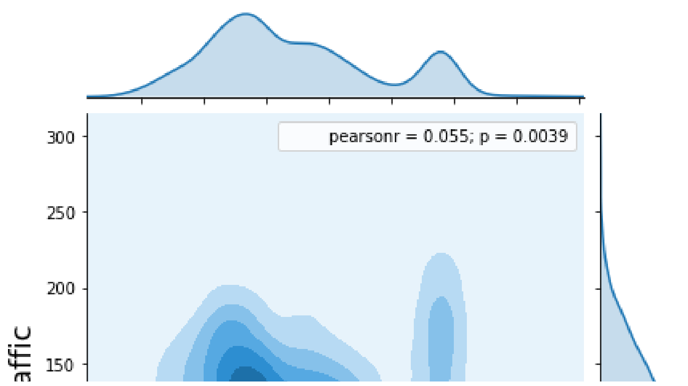


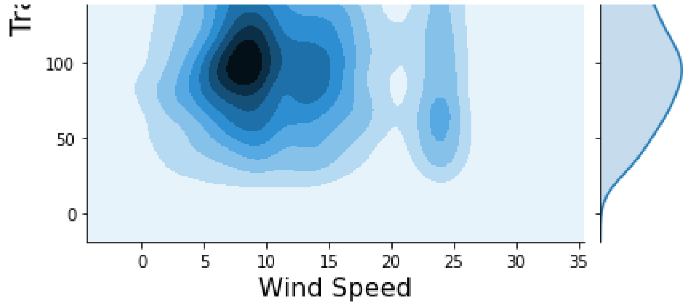
In [33]:

```
#plotting joint distribution of wind speed and Total
ax = sns.jointplot(x = 'WindSpeed',y = 'Total',data = train, kind = 'kde', stat_func = pearsonr)
ax.ax_joint.set_ylabel('Traffic', fontsize=18)
ax.ax_joint.set_xlabel('Wind Speed', fontsize=16)
```

Out[33]:

```
Text(0.5, 32.99999999999995, 'Wind Speed')
```



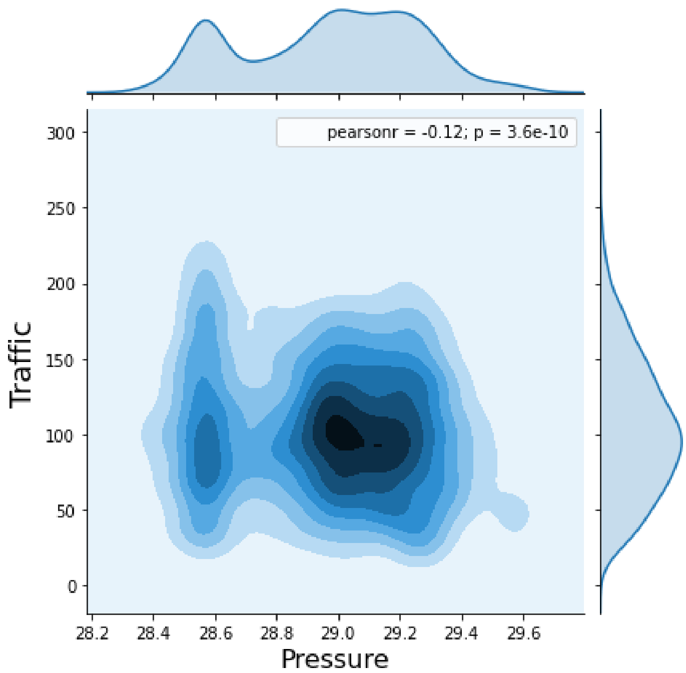


In [34]:

```
#plotting joint distribution of Pressure and Total
ax = sns.jointplot(x = 'Pressure',y = 'Total',data = train, kind = 'kde',stat_func = pearsonr)
ax.ax_joint.set_ylabel('Traffic', fontsize=18)
ax.ax_joint.set_xlabel('Pressure', fontsize=16)
```

Out[34]:

```
Text(0.5, 32.99999999999995, 'Pressure')
```

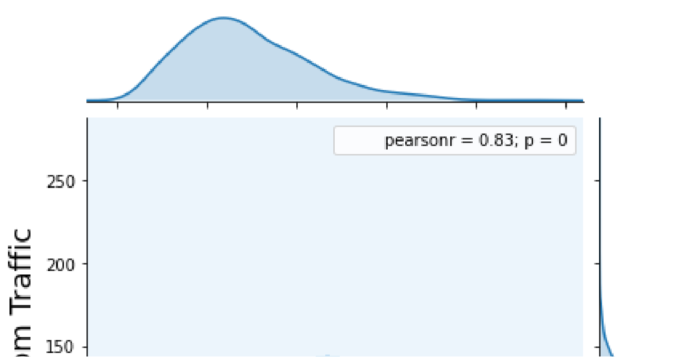


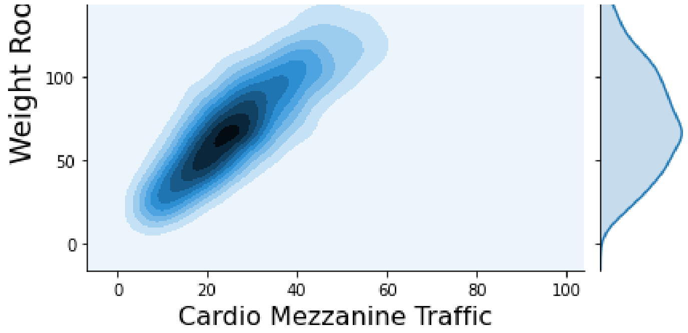
In [35]:

```
#plotting joint distribution of Temperature and Total
ax = sns.jointplot(x = 'CM',y = 'WR',data = train, kind = 'kde',stat_func = pearsonr)
ax.ax_joint.set_ylabel('Weight Room Traffic', fontsize=18)
ax.ax_joint.set_xlabel('Cardio Mezzanine Traffic', fontsize=16)
```

Out[35]:

```
Text(0.5, 32.99999999999995, 'Cardio Mezzanine Traffic')
```



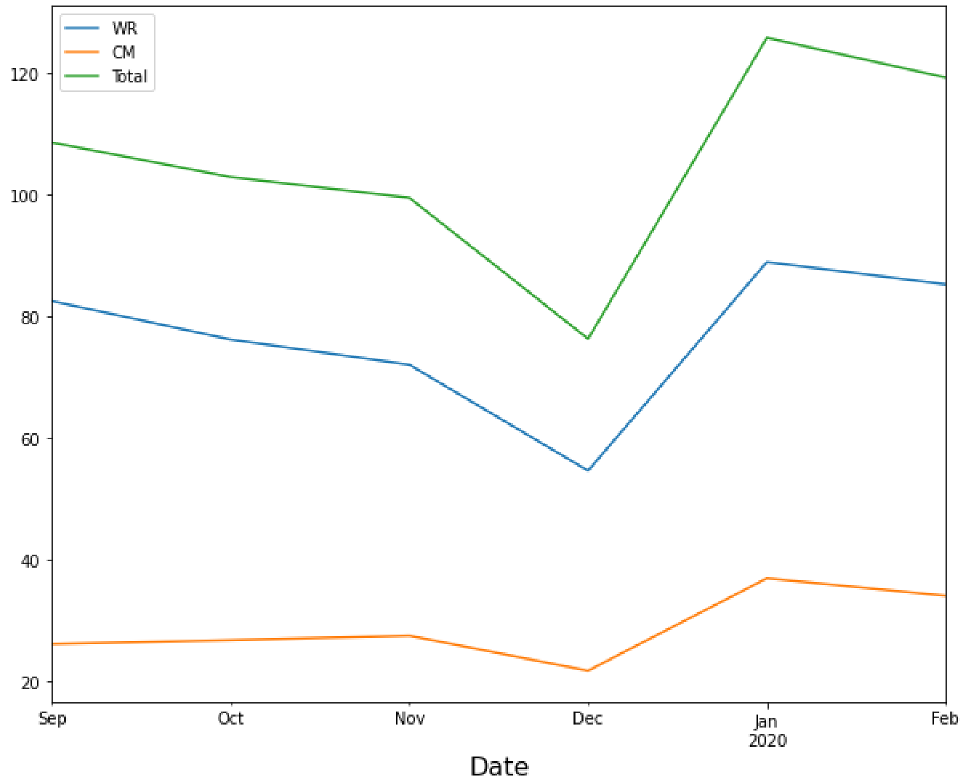


In [36]:

```
#plotting weight room cardio mezzanie and total traffic with monthly frequency averages
numbers = train[['WR','CM','Total','date']]

ax = numbers.groupby(pd.Grouper(key= 'date', freq="M")).mean().plot(figsize=(10,8))

ax.set_xlabel('Date', fontsize=16)
```

Out[36]:

```
Text(0.5, 0, 'Date')
```

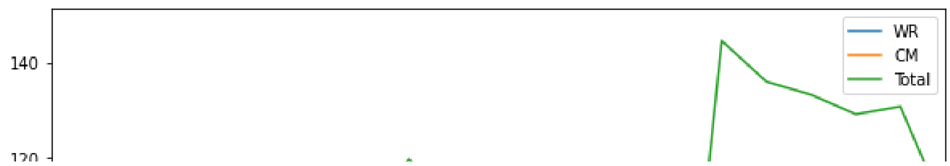


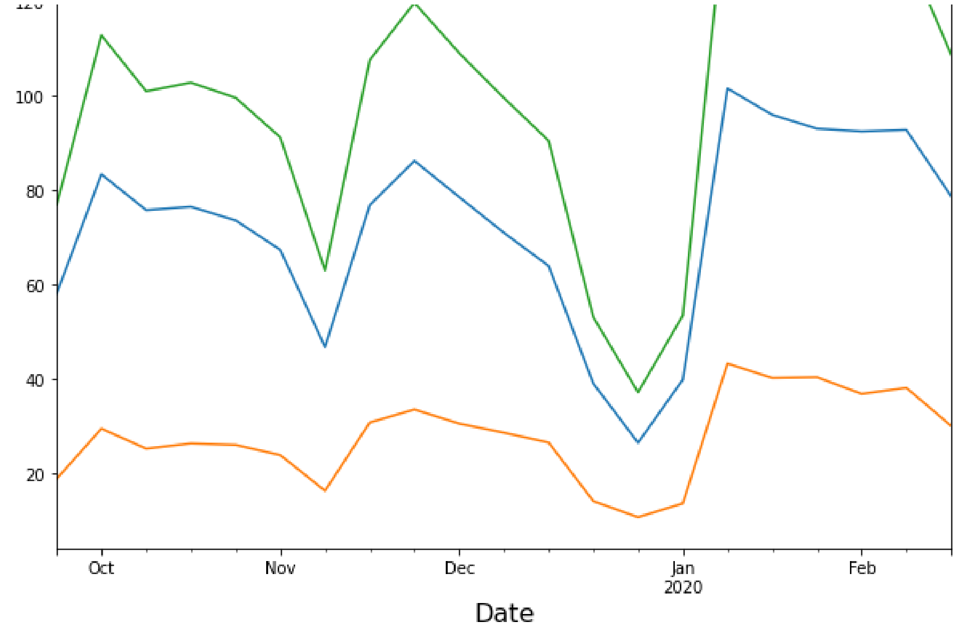
In [37]:

```
#plotting weight room cardio mezzanie and total traffic with weekly frequency averages
ax = numbers.groupby(pd.Grouper(key= 'date', freq="W")).mean().plot(figsize=(10,8))

ax.set_xlabel('Date', fontsize=16)
```

Out[37]:

```
Text(0.5, 0, 'Date')
```



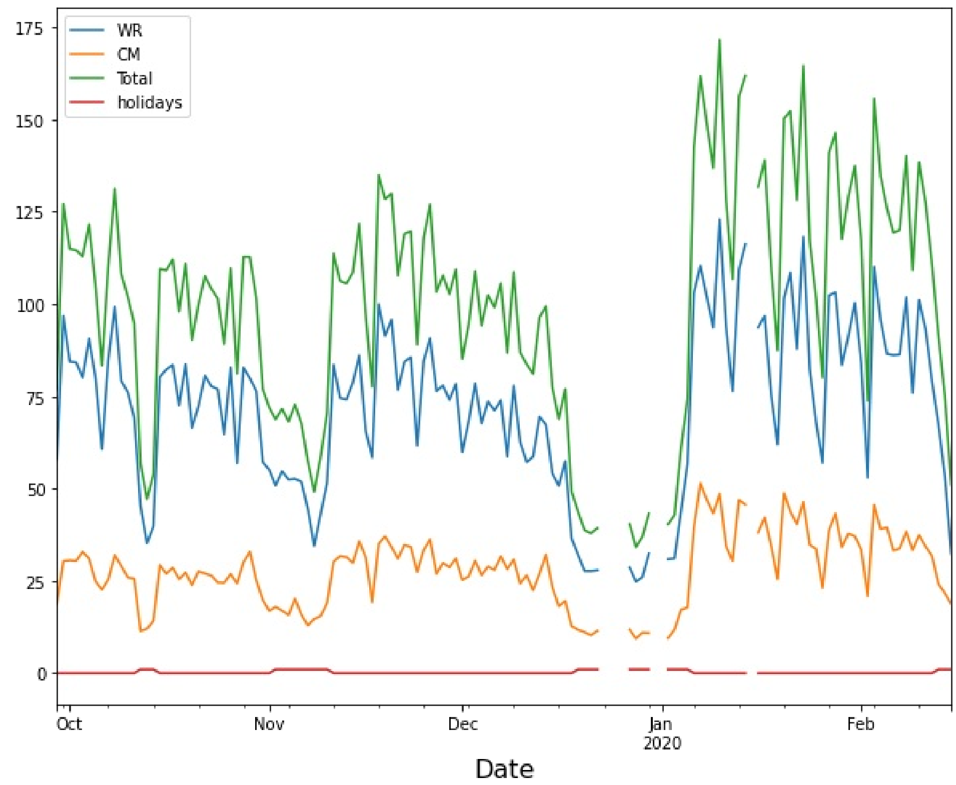


In [38]:

```
#plotting weight room cardio mezzanie and total traffic with hourly frequency averages
holi = train[['WR','CM','Total','date','holidays']]
ax = holi.groupby(pd.Grouper(key= 'date', freq="D")).mean().plot(figsize=(10,8))

ax.set_xlabel('Date', fontsize=16)
```

Out[38]:

```
Text(0.5, 0, 'Date')
```

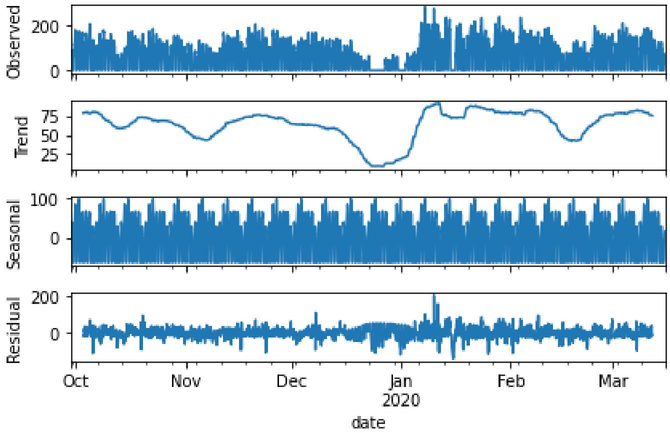


In [39]:

```
from statsmodels.tsa.seasonal import seasonal_decompose
#decomposing time series of total traffic with weekly hour (24*7 =168) frequency
indexed = time_df.copy()
indexed = indexed.set_index('date')
result = seasonal_decompose(indexed.Total, model='additive',freq = 168)
```

In [40]:

```
#plotting time series decomposition of total traffic
ax = result.plot()
```



E.8 Data Exportation

In [41]:

```
data= train.drop('tweets',axis  = 1)
data.to_csv('train.csv')
train_ts.to_csv('train_ts.csv')
```

In [42]:

```
data_test= train.drop('tweets',axis  = 1)
data_test.to_csv('test.csv')
test_ts.to_csv('test_ts.csv')
```

In [43]:

```
validation_ts .to_csv('validation_ts.csv')
```

Appendix F

Apendix F.1 Importing Libraries

In [1]:

```
# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from keras import Sequential
from keras.layers import Dense, LSTM, Dropout, GRU, Bidirectional
from keras.optimizers import SGD
import math
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import mean_squared_error
from tqdm import tqdm
import time
```

```
Using TensorFlow backend.
```

F.2 Loading Dataset

In [2]:

```
#reading in datasets and selecting subset of columns that will be used for modelling

train = pd.read_csv('train_ts.csv').drop(['Unnamed: 0'],axis = 1)
test = pd.read_csv('test_ts.csv').drop(['Unnamed: 0'],axis = 1)
validation = pd.read_csv('validation_ts.csv').drop(['Unnamed: 0'],axis = 1)
train_subset =
train[['date','Total','WindSpeed','Precip','Temperature','holidays','closed','closedxholidays']]
test_subset =
test[['date','Total','WindSpeed','Precip','Temperature','holidays','closed','closedxholidays']]
validation_subset =
validation[['date','Total','WindSpeed','Precip','Temperature','holidays','closed','closedxholidays'
]]
```

In [3]:

```
train_subset.head()
```

Out[3]:

	date	Total	WindSpeed	Precip	Temperature	holidays	closed	closedxholidays
0	2019-09-29 12:00:00	76.0	13.0	0.0	14.0	0	0	0
1	2019-09-29 13:00:00	78.0	15.0	0.0	15.0	0	0	0
2	2019-09-29 14:00:00	66.0	18.0	0.0	15.0	0	0	0
3	2019-09-29 15:00:00	67.0	16.0	0.0	16.0	0	0	0
4	2019-09-29 16:00:00	84.0	16.0	0.0	16.0	0	0	0

In [4]:

```
#setting index as date for each dataset and then obtaining their values as a numpy array
test_values = test_subset.copy().set_index('date')
test_values = test_values.values
train_values = train_subset.copy().set_index('date')
train_values = train_values.values
```

```
validation_values = validation_subset.copy().set_index('date')
validation_values = validation_values.values
```

F.2 Declaring Data Prepration Functions

In [5]:

```
#Extracted from https://machinelearningmastery.com/convert-time-series-supervised-learning-problem
-python/
#Few tweaks to make it work for my data
def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):

    n_vars = 1 if type(data) is list else data.shape[1]
    df = pd.DataFrame(data)
    cols, names = list(), list()
    # input sequence (t-n, ... t-1)
    for i in range(n_in, 0, -1):
        cols.append(df.shift(i))
        names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
    # forecast sequence (t, t+1, ... t+n)
    for i in range(0, n_out):
        cols.append(df.shift(-i))
        if i == 0:
            names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
        else:
            names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars)]
    # put it all together
    agg = pd.concat(cols, axis=1)
    agg.columns = names
    # drop rows with NaN values
    if dropnan:
        agg.dropna(inplace=True)
    return agg
```

In [33]:

```
#Input steps specifying the amount of lag variables to be outputed
#train_values corresponding set of values
#validation_values corresponding set of values
#test_values  corresponding set of values
#outputs dataframe with corresponding lag variables
def timestep_maker(steps,train_values,validation_values,test_values):
    #Using min max scaler to scale data between 0 and 1 as RNNs use a hyperbolic tangent
activation function which outputs values between 0 and 1
    scaler  = MinMaxScaler()
    scaled  = scaler.fit_transform(train_values)
    scaled_validation = scaler.fit_transform(validation_values)
    scaled_test  = scaler.fit_transform(test_values)

    #Using series to supervised function to make the datasets
    vals = series_to_supervised(scaled, steps, 1)
    vals_test  = series_to_supervised(scaled_test, steps, 1)
    vals_val  = series_to_supervised(scaled_validation, steps, 1)
    #Returns training test and validation sets
    return vals, vals_test, vals_val,scaler
```

In [7]:

```
#Set maker function
#Correctly formats each data set into correct shape to input into nueral network
#inputs
#step dataframe with timestep_maker format training set values
#step dataframe with timestep_maker format test test set values
#step_val dataframe with timestep_maker format validation set values
#columns intiger number of columns
#steps intiger number of steps to take

def set_maker(step,step_test,step_val,columns,steps):
    #Gets var1 which is the target value
    test_y  = step_test[['var1(t)']].values
    train_y  = step[['var1(t)']].values
    val_y  = step_val[['var1(t)']].values
```

```
    #drops val1 and reshapes the dataframes for input into a numpy tensor
    test_X  = step_test.iloc[:,:columns * steps + 1]
    test_X  = test_X.copy().drop(['var1(t)'],axis =1 ).values
    test_X  = test_X.reshape((test_X.shape[0], 1, test_X.shape[1]))

    val_X  = step_val.iloc[:,:columns * steps + 1]
    val_X  = val_X.copy().drop(['var1(t)'],axis =1 ).values
    val_X  = val_X.reshape((val_X.shape[0], 1, val_X.shape[1]))

    train_X  = step.iloc[:,:columns * steps + 1]
    train_X  = train_X.copy().drop(['var1(t)'],axis =1 ).values
    train_X = train_X.reshape((train_X.shape[0], 1, train_X.shape[1]))
    return train_X,test_X,train_y,test_y,val_y,val_X
```

F.2 Declaring Modelling Functions

In [8]:

```
#functions single
#single layer neural network uses an LSTM or long short term emmory cell as the recurrent part ini
tially takes 50 nuerons
#Dense layer for a single output
#Ueses adam optimizer and mean absolute error as loss function
#inputs
#train_x predictor values for training set
#test_X predictor values for test set
#train_y response values for training set
#test_y response values for test set
#val_y response values for validation set
#val_x predictor values for validation set
#batch number of training batches
def grid_search(iterations,train_values,test_values,validation_values,model_maker,cols = 7, batch =
32 ):

    error_list = []
    for i in tqdm(range(1,iterations) ):
        step, step_test, step_val,scaler =
timestep_maker(i,train_values,test_values,validation_values)

        train_X,test_X,train_y,test_y,val_y,val_X = set_maker(step,step_test,step_val,cols,i)

        fit = model_maker(train_X,test_X,train_y,test_y,val_y,val_X , batch)
        rmse = evaluation(fit,val_X,val_y,scaler )
        error_list.append(rmse)

    return error_list
```

In [32]:

```
#Helper function
#Extracted from https://machinelearningmastery.com/convert-time-series-supervised-learning-problem
-python/
def evaluation(net,test_X,test_y,scaler):
    # make a prediction
    yhat = net.predict(test_X)
    test_X = test_X.reshape((test_X.shape[0], test_X.shape[2]))
    # invert scaling for forecast

    # invert scaling for actual
    inv_yhat = np.concatenate((yhat, test_X[:, -6:]), axis=1)
    inv_yhat = scaler.inverse_transform(inv_yhat)
    inv_yhat = inv_yhat[:,0]

    test_y = test_y.reshape((len(test_y), 1))
    inv_y = np.concatenate((test_y, test_X[:, -6:]), axis=1)
    inv_y = scaler.inverse_transform(inv_y)
    inv_y = inv_y[:,0]
    # calculate RMSE
    rmse = np.sqrt(mean_squared_error(inv_y, inv_yhat))
    return rmse
```

In [34]:

```
#model_spitter
#grid searches for best amount of steps with input model outputs list rmse of each model in the gr
id
#inputs
#train_values corresponding dataset
#test_values corresponding dataset
#validation values corresponding dataset
#model_maker specifies the type of model to use
#cols number of columns by default 7
#outputs trained model with specific iterations
#outputs model predictions and real values

def model_spitter(train_values,test_values,validation_values,model_maker,steps = 2, cols = 7 ):


    step, step_test, step_val,scaler = timestep_maker(steps,train_values,test_values,validation_val
ues)

    train_X,test_X,train_y,test_y,val_y,val_X = set_maker(step,step_test,step_val,cols,steps)

    fit = model_maker(train_X,test_X,train_y,test_y,val_y,val_X , 74)

    yhat = fit.predict(test_X)
    test_X = test_X.reshape((test_X.shape[0], test_X.shape[2]))
    # invert scaling for forecast

    # invert scaling for actual
    inv_yhat = np.concatenate((yhat, test_X[:, -6:]), axis=1)
    inv_yhat = scaler.inverse_transform(inv_yhat)
    inv_yhat = inv_yhat[:,0]

    test_y = test_y.reshape((len(test_y), 1))
    inv_y = np.concatenate((test_y, test_X[:, -6:]), axis=1)
    inv_y = scaler.inverse_transform(inv_y)
    inv_y = inv_y[:,0]


    return fit,inv_yhat,inv_y
```

In [10]:

```
#functions single
#single layer neural network uses an LSTM or long short term emmory cell as the recurrent part ini
tially takes 50 nuerons
#Dense layer for a single output
#Ueses adam optimizer and mean absolute error as loss function
#inputs
#train_x predictor values for training set
#test_X predictor values for test set
#train_y response values for training set
#test_y response values for test set
#val_y response values for validation set
#val_x predictor values for validation set
#batch number of training batches
def single(train_X,test_X,train_y,test_y,val_y,val_X , batch = 32):
    single_layer = Sequential()
    single_layer.add(LSTM(50,input_shape = (train_X.shape[1],train_X.shape[2])))
    single_layer.add(Dense(1))
    single_layer.compile(loss = 'mae',optimizer  = 'adam')
    hist = single_layer.fit(train_X,train_y,epochs = 50, batch_size =batch, verbose =0,shuffle = Fa
lse,validation_data=(val_X, val_y))



    return single_layer
```

In [12]:

```
#functions dropout_layer
#single layer neural network uses an LSTM or long short term emmory cell as the recurrent part ini
tially takes 50 nuerons
#Dense layer for a single output
#introduces a dropout layer for regularization
#Ueses adam optimizer and mean absolute error as loss function
```

```
#inputs
#train_x predictor values for training set
#test_X predictor values for test set
#train_y response values for training set
#test_y response values for test set
#val_y response values for validation set
#val_x predictor values for validation set
#batch number of training batches
def dropout_layer(train_X,test_X,train_y,test_y,val_y,val_X,batch):
    model = Sequential()
    model.add(LSTM(50,input_shape = (train_X.shape[1],train_X.shape[2])))
    model.add(Dropout(0.2))
    model.add(Dense(1))
    model.compile(loss = 'mae',optimizer  = 'adam')
    hist = model.fit(train_X,train_y,epochs = 50, batch_size = 74, verbose =0,shuffle = False,valid
ation_data=(val_X, val_y))



    return model
```

In [13]:

```
#functions multi_layer
#single layer neural network uses an LSTM or long short term emmory cell as the recurrent part ini
tially takes 50 nuerons
#Adds a second LSTM layer with 100 input cells
#both LSTM cells have dropout layers for regularization
#Ueses adam optimizer and mean absolute error as loss function
#inputs
#train_x predictor values for training set
#test_X predictor values for test set
#train_y response values for training set
#test_y response values for test set
#val_y response values for validation set
#val_x predictor values for validation set
#batch number of training batches
def multi_layer(train_X,test_X,train_y,test_y,val_y,val_X,batch):
    model = Sequential()
    model.add(LSTM(50,input_shape = (train_X.shape[1],train_X.shape[2]),return_sequences=True))
    model.add(Dropout(0.2))

    model.add(LSTM(100,input_shape = (train_X.shape[1],train_X.shape[2])))
    model.add(Dropout(0.2))

    model.add(Dense(1))
    model.compile(loss = 'mae',optimizer  = 'adam')
    hist = model.fit(train_X,train_y,epochs = 50, batch_size =32, verbose =0,shuffle = False,valida
tion_data=(val_X, val_y))



    return model
```

F.4 Training Models

In [ ]:

```
#Models will be saved into csv files due to how time consuming grid search is
grid = grid_search(24,train_values,test_values,validation_values,single)
np.savetxt('grid.csv', grid, delimiter=',', fmt='%d')
```

In [ ]:

```
grid_batch = grid_search(24,train_values,test_values,validation_values,single, 74 )
np.savetxt('grid_batch.csv', grid_batch, delimiter=',', fmt='%d')
```

In [ ]:

```
grid_batch_bigger = grid_search(11,train_values,test_values,validation_values,dropout_layer, 148 )
np.savetxt('grid_batch_bigger.csv', grid_batch_bigger, delimiter=',', fmt='%d')
```

In [ ]:

```
grid_dropout = grid_search(24,train_values,test_values,validation_values,dropout_layer)
np.savetxt('grid_dropout.csv', grid_dropout, delimiter=',', fmt='%d')
```

In [ ]:

```
grid_dropout_batch = grid_search(24,train_values,test_values,validation_values,dropout_layer,74)
np.savetxt('grid_dropout_batch.csv', grid_dropout_batch, delimiter=',', fmt='%d')
```

In [ ]:

```
grid_multilayer = grid_search(12,train_values,test_values,validation_values,multi_layer,74)
np.savetxt('grid_multilayer.csv', grid_dropout_batch, delimiter=',', fmt='%d')
```

F.5 Model Evaluation

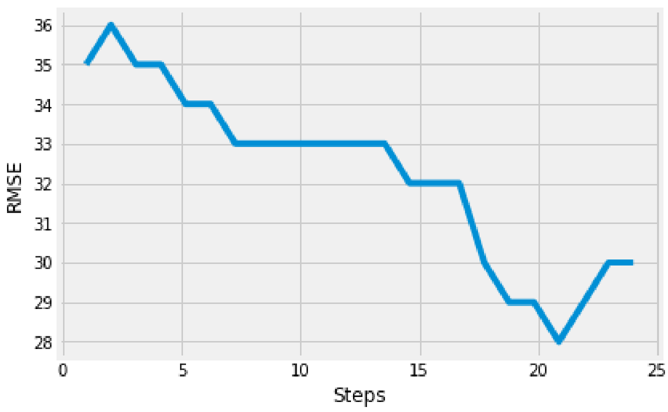
In [14]:

```
grid = np.loadtxt('grid.csv')
```

In [15]:

```
x_range = np.linspace(1,24,23)
plt.plot(x_range,grid)
plt.xlabel('Steps')
plt.ylabel('RMSE')
print(np.min(grid))
```

```
28.0
```



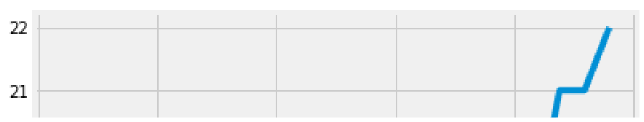
In [16]:

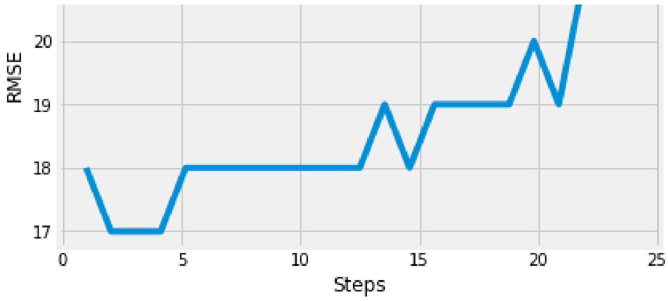
```
grid_batch = np.loadtxt('grid_batch.csv')
```

In [17]:

```
x_range = np.linspace(1,24,23)
plt.plot(x_range,grid_batch)
plt.xlabel('Steps')
plt.ylabel('RMSE')
print(np.min(grid_batch))
```

```
17.0
```





In [18]:

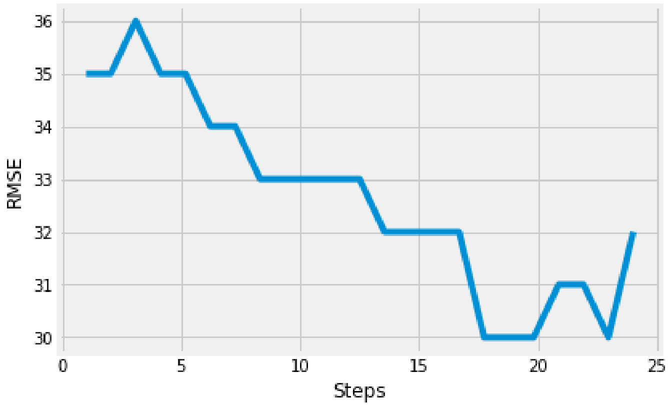
```
grid_dropout = np.loadtxt('grid_dropout.csv')
```

In [19]:

```
x_range = np.linspace(1,24,23)
plt.plot(x_range,grid_dropout)
plt.xlabel('Steps')
plt.ylabel('RMSE')
```

Out[19]:

```
Text(0, 0.5, 'RMSE')
```



In [20]:

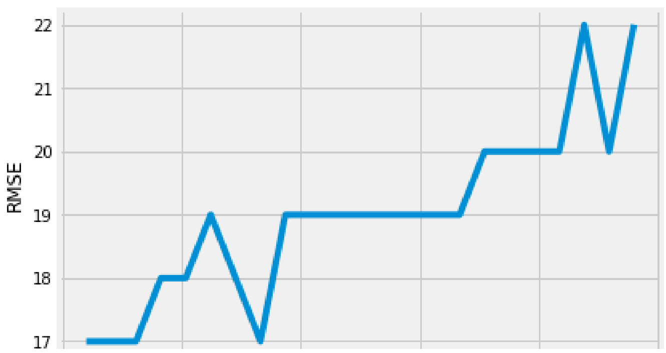
```
grid_dropout_batch = np.loadtxt('grid_dropout_batch.csv')
```

In [21]:

```
x_range = np.linspace(1,24,23)
plt.plot(x_range,grid_dropout_batch)
plt.xlabel('Steps')
plt.ylabel('RMSE')
```

Out[21]:

```
Text(0, 0.5, 'RMSE')
```



In [22]:

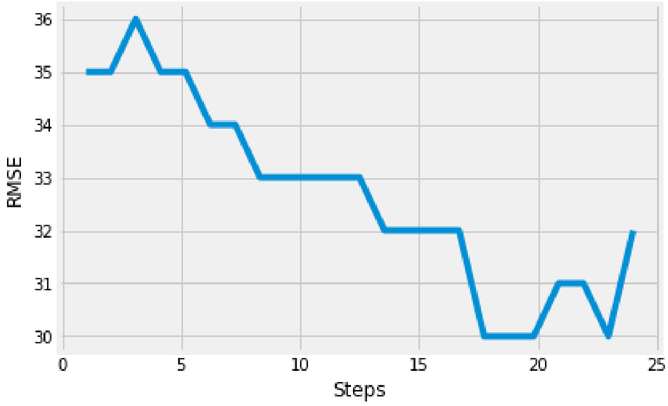
```
grid_multilayer = np.loadtxt('grid_multilayer.csv')
```

In [23]:

```
x_range = np.linspace(1,24,23)
plt.plot(x_range,grid_dropout)
plt.xlabel('Steps')
plt.ylabel('RMSE')
```

Out[23]:

```
Text(0, 0.5, 'RMSE')
```



In [24]:

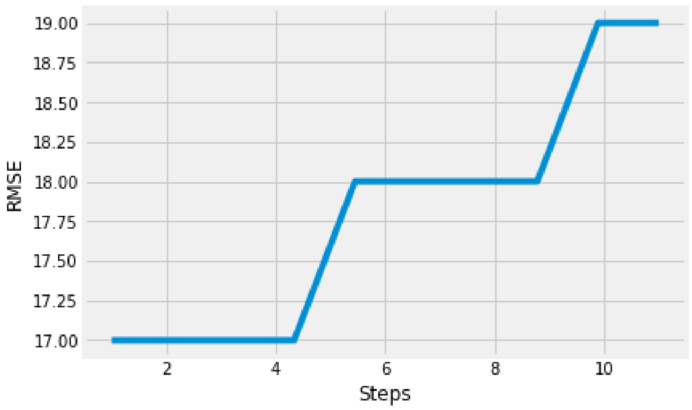
```
grid_batch_bigger = np.loadtxt('grid_batch_bigger.csv')
```

In [25]:

```
x_range = np.linspace(1,11,10)
plt.plot(x_range,grid_batch_bigger)
plt.xlabel('Steps')
plt.ylabel('RMSE')
```

Out[25]:

```
Text(0, 0.5, 'RMSE')
```



F.6 Test Set Evaluation

In [38]:

```
rmse_arr = np.zeros(50)
for i in range(50):
    final_fit,inv_yhat,inv_y  = model_spitter(train_values,test_values,validation_values,dropout_la
yer,cols = 7 )
    rmse = np.sqrt(mean_squared_error(inv_y, inv_yhat))
    rmse_arr[i] = rmse
```

In [39]:

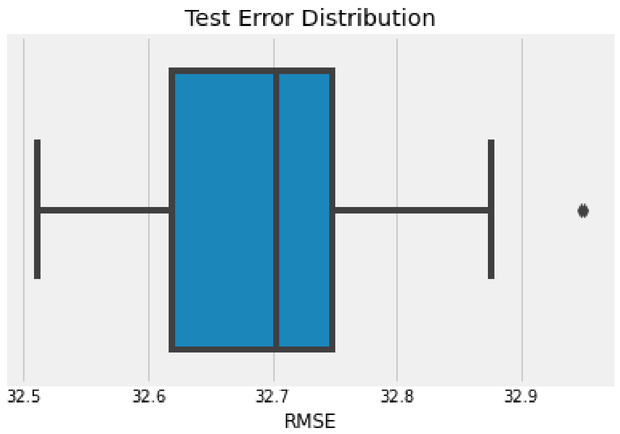
```
import seaborn as sns
```

In [44]:

```
ax = sns.boxplot(rmse_arr)
ax.set_title('Test Error Distribution')
ax.set_xlabel('RMSE')
```

Out[44]:

```
Text(0.5, 0, 'RMSE')
```

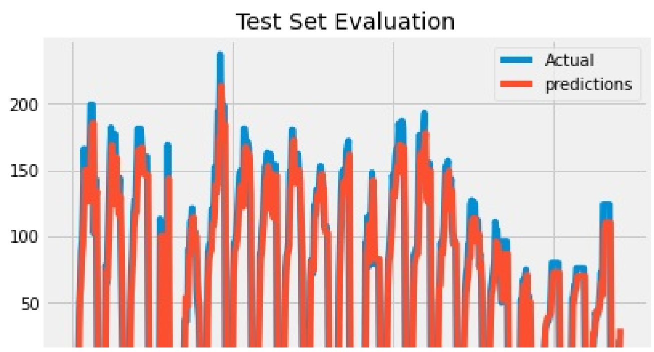


In [46]:

```
import matplotlib.ticker as plticker

fig, ax = plt.subplots()



loc = plticker.MultipleLocator(base=150) # this locator puts ticks at regular intervals
ax.xaxis.set_major_locator(loc)
ax.plot(test.date[-len(inv_y):],inv_y,label = 'Actual')
ax.plot(test.date[-len(inv_y):],inv_yhat,label = 'predictions')
ax.set_title("Test Set Evaluation")
plt.xticks(rotation=30)
plt.legend()
```

Out[46]:

```
<matplotlib.legend.Legend at 0x168f13717c8>
```




0
2020-02-23 07:00:00
2020-02-29 13:00:00
2020-03-06 19:00:00
2020-03-13 01:00:00