Uber_dataset_ashwin

December 1, 2020

0.0.1 This notebook is compiled by Ashwin K Raghu as a part of the job application to Citility, Bengaluru

1 PROBLEM DESCRIPTION

1.0.1 Q) Analyze the dataset "Uber Pickups in the New York City - July 2014"

Dataset acquired from Kaggle: https://www.kaggle.com/fivethirtyeight/uber-pickups-in-new-york-city?select=uber-raw-data-jul14.csv Dataset Name: **uber-raw-data-jul14.csv**

If the dataset is not present in the root directory of this notebook, please download it from the kaggle link above

Reference: https://www.coursera.org/lecture/machine-learning-asset-management-alternative-data/lab-session-introduction-to-the-uber-dataset-R2ZOK

1.0.2 Details of the dataset:

The dataset contains information about the Datetime, Latitude, Longitude and Base of each uber ride that happened in the month of July 2014 at New York City, USA Date/Time : The date and time of the Uber pickup Lat : The latitude of the Uber pickup Lon : The longitude of the Uber pickup Base : The TLC base company code affiliated with the Uber pickup

The Base codes are for the following Uber bases: B02512 : Unter B02598 : Hinter B02617 : Weiter B02682 : Schmecken B02764 : Danach-NY

Run the cell below to meet the library requirements (Shell Command)

[1]:
```
!pip3 -q install numpy pandas matplotlib seaborn geopy folium datetime scipy␣
↪sklearn tensorflow
```

[2]:
```
#The following libraries are required to run this notebook

%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib
import seaborn as sns
import geopy.distance
```

```
from math import radians,cos,sin,asin,sqrt
import folium
import datetime
from folium.plugins import HeatMap
from scipy.stats import ttest_ind
from sklearn.model_selection import train_test_split
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import r2_score,mean_squared_error

matplotlib.rcParams.update({'font.size': 12})
```

Reading the uber dataset

[3]:
```
uber_data = pd.read_csv('uber-raw-data-jul14.csv')
```

[4]:
```
# Print the first 10 elements
uber_data.head(10)
```

[4]:

	Date/Time	Lat	Lon	Base
0	7/1/2014 0:03:00	40.7586	-73.9706	B02512
1	7/1/2014 0:05:00	40.7605	-73.9994	B02512
2	7/1/2014 0:06:00	40.7320	-73.9999	B02512
3	7/1/2014 0:09:00	40.7635	-73.9793	B02512
4	7/1/2014 0:20:00	40.7204	-74.0047	B02512
5	7/1/2014 0:35:00	40.7487	-73.9869	B02512
6	7/1/2014 0:57:00	40.7444	-73.9961	B02512
7	7/1/2014 0:58:00	40.7132	-73.9492	B02512
8	7/1/2014 1:04:00	40.7590	-73.9730	B02512
9	7/1/2014 1:08:00	40.7601	-73.9823	B02512

[5]:
```
#print the type of data in Date/Time
type(uber_data.loc[0,'Date/Time'])
```

[5]: str

The type is str!. Let's convert it to datetime format for easy indexing

[6]:
```
uber_data['Date/Time'] = pd.to_datetime(uber_data['Date/Time'])
```

Let us divide each hour in existing Date/Time column into four smaller bins of 15 mins each:

[0mins -15mins], [15mins - 30mins], [30mins - 45mins] and [45mins - 60mins]

This will allow us to visualize the time series more precisely.

[7]:
```python
#create a new column to store this new binned column
uber_data['BinnedHour']=uber_data['Date/Time'].dt.floor('15min')
```

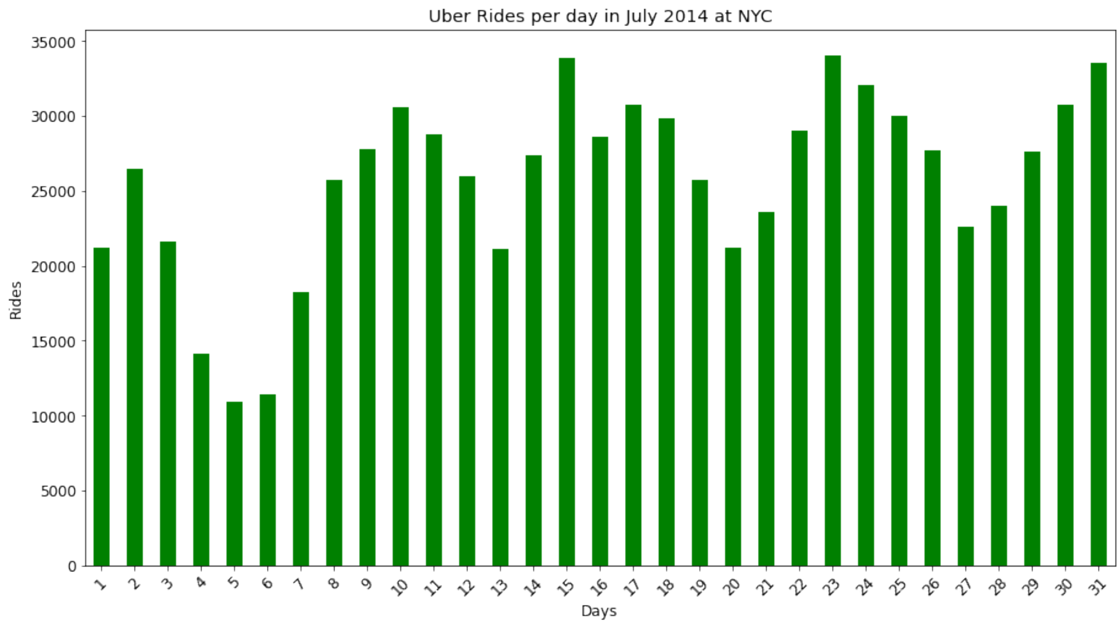
[8]:
```python
#printing the new column - BinnedHour
uber_data['BinnedHour']
```

[8]:
```
0         2014-07-01 00:00:00
1         2014-07-01 00:00:00
2         2014-07-01 00:00:00
3         2014-07-01 00:00:00
4         2014-07-01 00:15:00
                  …
796116    2014-07-31 23:15:00
796117    2014-07-31 23:15:00
796118    2014-07-31 23:15:00
796119    2014-07-31 23:30:00
796120    2014-07-31 23:45:00
Name: BinnedHour, Length: 796121, dtype: datetime64[ns]
```

1.0.3 Visualizing the Dataset

Let us visualize the total uber rides per day in the month of July 2014

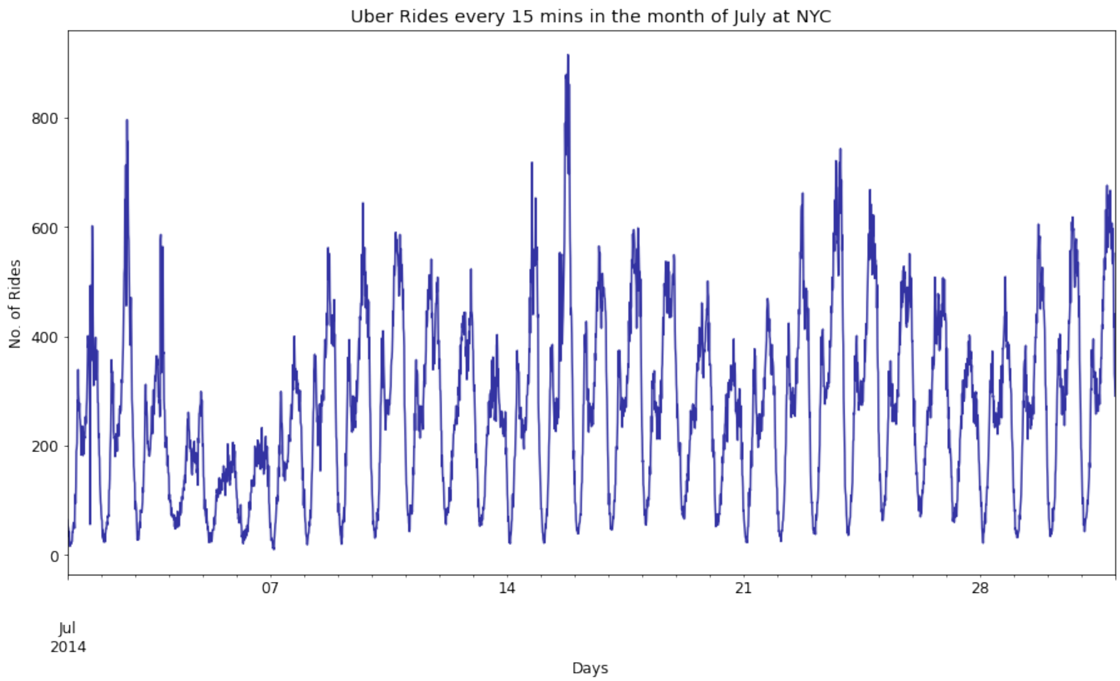
[9]:
```python
plt.figure(figsize=(15,8))
uber_data['BinnedHour'].dt.day.value_counts().sort_index().
 ↪plot(kind='bar',color='green')
for item in plt.gca().get_xticklabels():
    item.set_rotation(45)
plt.title('Uber Rides per day in July 2014 at NYC')
plt.xlabel('Days')
_=plt.ylabel('Rides')
```



Observe the nearly recurring pattern in the data!. It is very noticable after day 11.

Let us have a more closer look at it, say every 15 minutes from July 1 to July 31.

[10]:
```
plt.figure(figsize=(15,8))
uber_data['BinnedHour'].value_counts().sort_index().plot(c='darkblue',alpha=0.8)
plt.title('Uber Rides every 15 mins in the month of July at NYC')
plt.xlabel('Days')
_=plt.ylabel('No. of Rides')
```



The underlying trend is clearly visible now. It conveys that in a day there are times when the pickups are very low and very high, and they seem to follow a pattern.

Q) Which times correspond to the highest and lowest peaks in the plot above?

[11]:
```
uber_data['BinnedHour'].value_counts()
```

[11]:
```
2014-07-15 19:15:00    915
2014-07-15 18:15:00    879
2014-07-15 17:45:00    877
2014-07-15 18:00:00    872
2014-07-15 20:00:00    861
                      …
2014-07-01 02:00:00     17
2014-07-07 01:45:00     15
2014-07-07 02:15:00     14
2014-07-07 02:00:00     12
2014-07-07 02:30:00     10
Name: BinnedHour, Length: 2976, dtype: int64
```

The highest peak corresponds to the time 19:15(7:15 PM), 15th July 2014 and has a ride count of 915 and the lowest peak corresponds to the time 02:30, 7th July 2014 and has a ride count of 10

Now, Lets visualize the week wise trends in the data. For it, we have to map each date into its day name using a dictionary

[12]:
```
#defining a dictionary to map the weekday to day name
DayMap={0:'Monday', 1:'Tuesday', 2:'Wednesday', 3:'Thursday', 4:'Friday', 5:
 ↪'Saturday', 6:'Sunday'}
uber_data['Day']=uber_data['BinnedHour'].dt.weekday.map(DayMap)
```

[13]:
```
#Separating the date to another column
uber_data['Date']=uber_data['BinnedHour'].dt.date
```

[14]:
```
#Defining ordered category of week days for easy sorting and visualization
uber_data['Day']=pd.
 ↪Categorical(uber_data['Day'],categories=['Monday','Tuesday','Wednesday','Thursday','Friday'
```

[15]:
```
#Separating time from the "BinnedHour" Column
uber_data['Time']=uber_data['BinnedHour'].dt.time
```

Rearranging the dataset for weekly analysis

[16]:
```
weekly_data = uber_data.groupby(['Date','Day','Time']).count().dropna().
 ↪rename(columns={'BinnedHour':'Rides'})['Rides'].reset_index()
weekly_data.head(10)
```

[16]:

	Date	Day	Time	Rides
0	2014-07-01	Tuesday	00:00:00	64.0
1	2014-07-01	Tuesday	00:15:00	54.0
2	2014-07-01	Tuesday	00:30:00	51.0
3	2014-07-01	Tuesday	00:45:00	47.0
4	2014-07-01	Tuesday	01:00:00	34.0
5	2014-07-01	Tuesday	01:15:00	42.0
6	2014-07-01	Tuesday	01:30:00	17.0
7	2014-07-01	Tuesday	01:45:00	18.0
8	2014-07-01	Tuesday	02:00:00	17.0
9	2014-07-01	Tuesday	02:15:00	22.0

Grouping weekly_data by days to plot total rides per week in july 2014.

[17]:
```
#Grouping the weekly_data daywise
daywise = weekly_data.groupby('Day').sum()
daywise
```

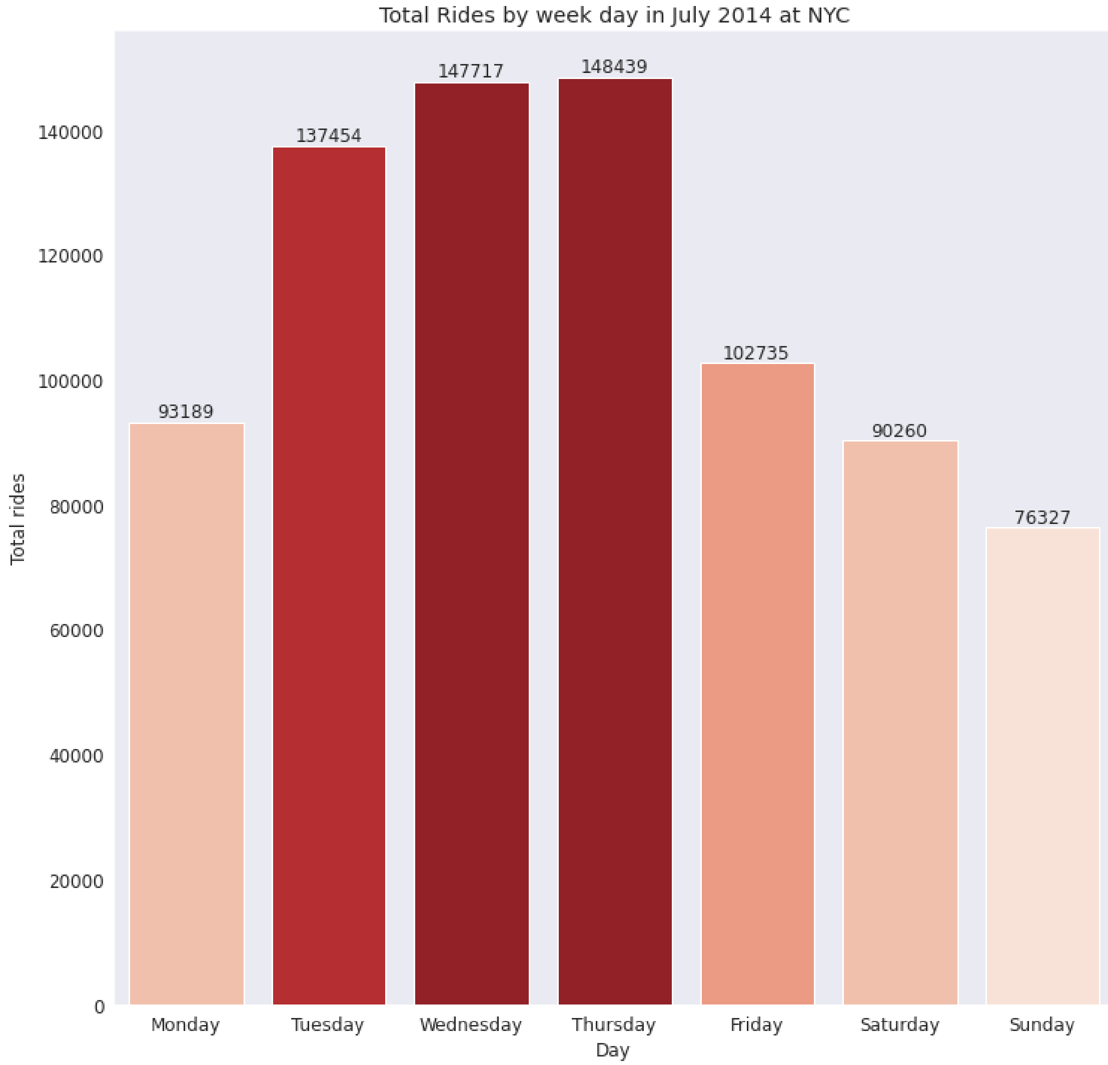
[17]:

Day	Rides
Monday	93189.0
Tuesday	137454.0
Wednesday	147717.0
Thursday	148439.0
Friday	102735.0
Saturday	90260.0

```
Sunday      76327.0
```

[18]:

```
#Plotting the graphs for a better visualization
sns.set_style("dark")
plt.figure(figsize=(12,12))

#Creating a customized color palette for custom hue according to height of bars
vals = daywise.to_numpy().ravel()
normalized = (vals - np.min(vals)) / (np.max(vals) - np.min(vals))
indices = np.round(normalized * (len(vals) - 1)).astype(np.int32)
palette = sns.color_palette('Reds', len(vals))
colorPal = np.array(palette).take(indices, axis=0)

#Creating a bar plot
ax=sns.barplot(x = daywise.index,y= vals,palette=colorPal)
plt.ylabel('Total rides')
plt.title('Total Rides by week day in July 2014 at NYC')
for rect in ax.patches:
    ax.text(rect.get_x() + rect.get_width()/2.0,rect.get_height(),int(rect.
 ↪get_height()), ha='center', va='bottom')
```



According to the bar plot above, rides are maximum on Thursdays and minimum on Sundays. Sundays having the lowest number of rides makes sense logically, as it's a holiday and people often take rest on that day.

[19]:
```
weekly_data = weekly_data.groupby(['Day','Time']).mean()['Rides']
weekly_data.head(10)
```

[19]:
```
Day     Time
Monday  00:00:00    102.50
        00:15:00     85.00
        00:30:00     67.75
        00:45:00     59.75
        01:00:00     53.75
        01:15:00     41.50
        01:30:00     29.75
```

```
          01:45:00     28.25
          02:00:00     20.25
          02:15:00     24.50
Name: Rides, dtype: float64
```

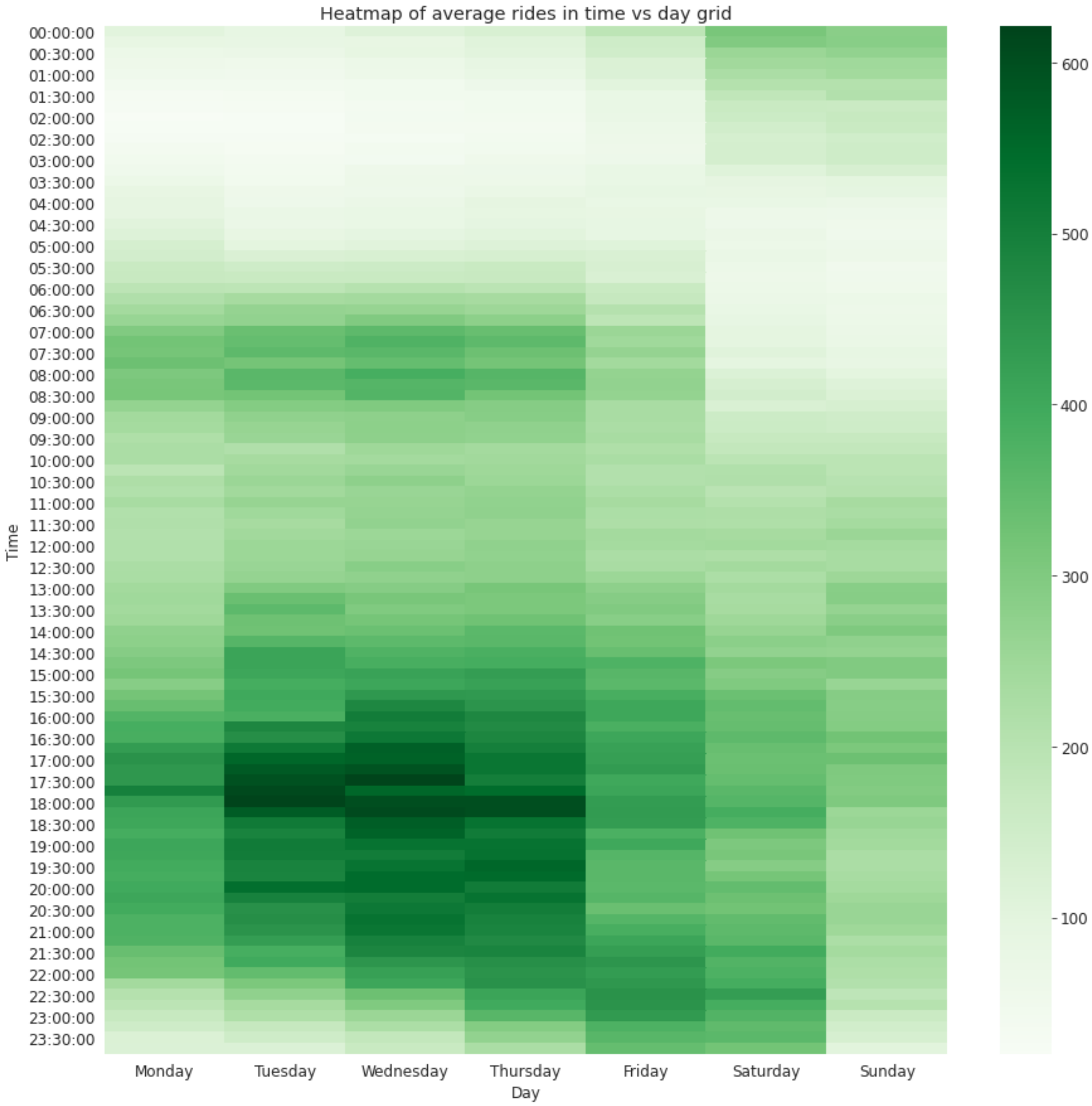
[20]:
```
#Unstacking the data to create heatmap
weekly_data= weekly_data.unstack(level=0)
weekly_data
```

[20]:

Day Time	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
00:00:00	102.50	87.6	112.2	130.4	191.25	312.00	284.50
00:15:00	85.00	82.4	85.2	109.4	154.00	297.75	287.75
00:30:00	67.75	74.2	89.2	103.0	148.00	256.50	270.50
00:45:00	59.75	57.6	68.0	87.2	121.75	244.00	247.25
01:00:00	53.75	48.8	62.8	76.6	120.00	225.75	242.75
…	…	…	…	…	…	…	…
22:45:00	193.00	234.0	299.8	397.2	450.25	389.50	204.00
23:00:00	172.25	218.8	255.8	360.8	435.00	372.00	164.75
23:15:00	152.25	174.2	223.8	294.6	379.25	349.00	146.25
23:30:00	121.25	152.0	179.0	270.0	361.25	358.00	134.50
23:45:00	120.75	119.6	166.6	225.8	343.75	321.00	104.00

[96 rows x 7 columns]

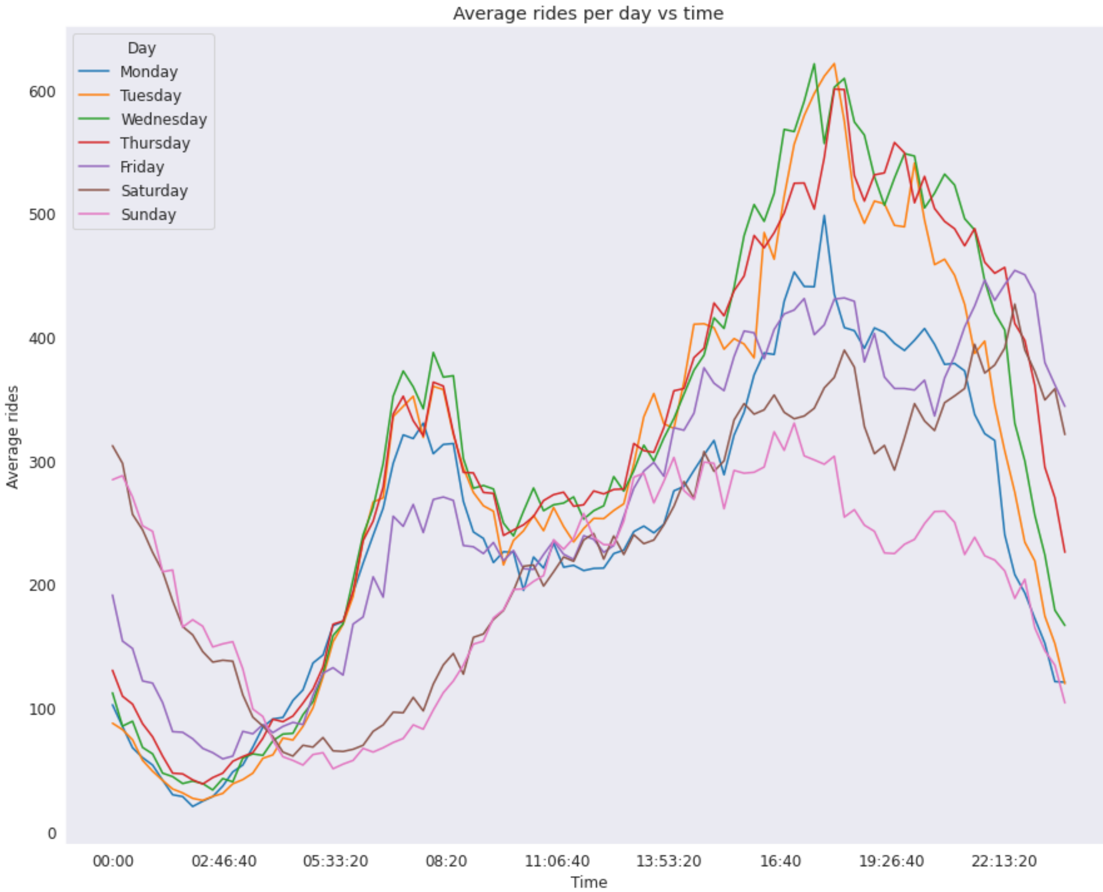
[21]:
```
plt.figure(figsize=(15,15))
sns.heatmap(weekly_data,cmap='Greens')
_=plt.title('Heatmap of average rides in time vs day grid')
```



The heatmap indicates that the maximum average uber rides occur around 5:30PM to 6:15PM on Wednesdays and Thursdays and their values fall between 550 to 620.

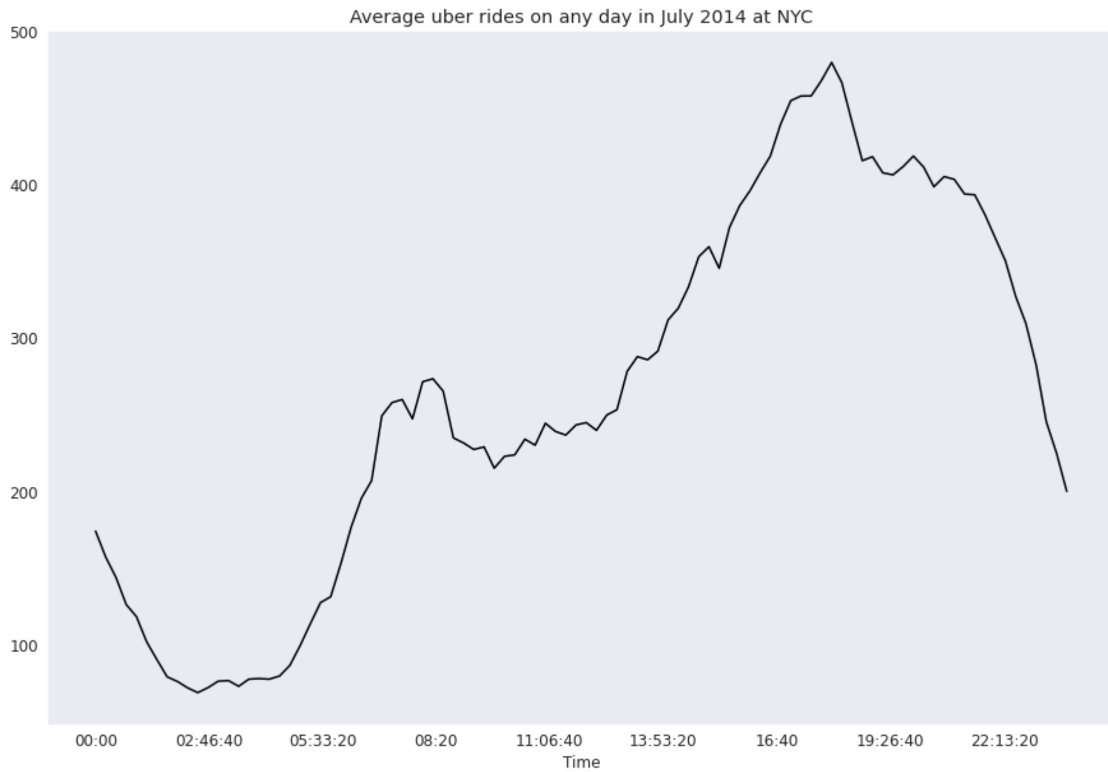
Here is another way of looking at it:

[22]:
```
plt.figure(figsize=(15,12))
weekly_data.plot(ax=plt.gca())
_=plt.title('Average rides per day vs time')
_=plt.ylabel('Average rides')
plt.locator_params(axis='x', nbins=10)
```



Finding average rides on any day

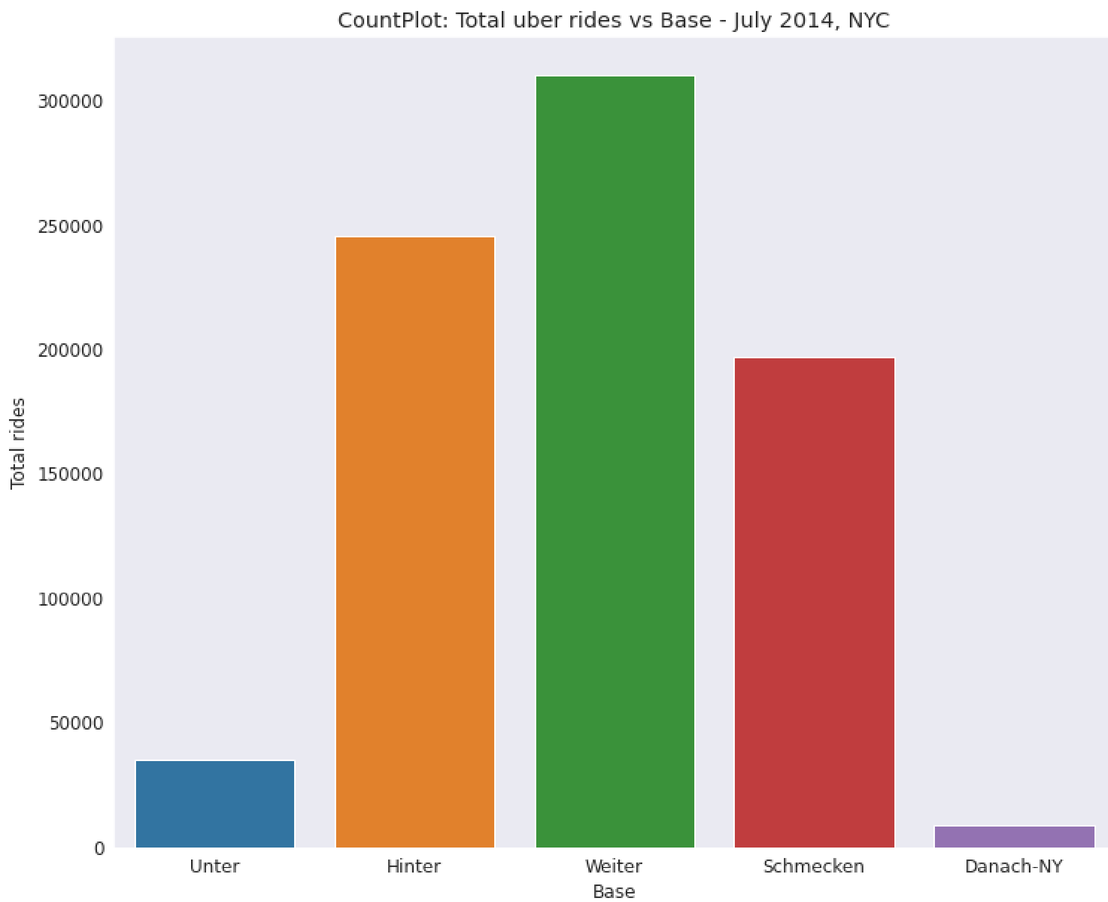
[23]:
```
plt.figure(figsize=(15,10))
weekly_data.T.mean().plot(c = 'black')
_=plt.title('Average uber rides on any day in July 2014 at NYC')
plt.locator_params(axis='x', nbins=10)
```



This plot further confirms that the average rides on any given day is lowest around 2 AM and highest in the around 5:30 PM.

Now, let's try visualizing the relationship between Base and total number of rides in July 2014:

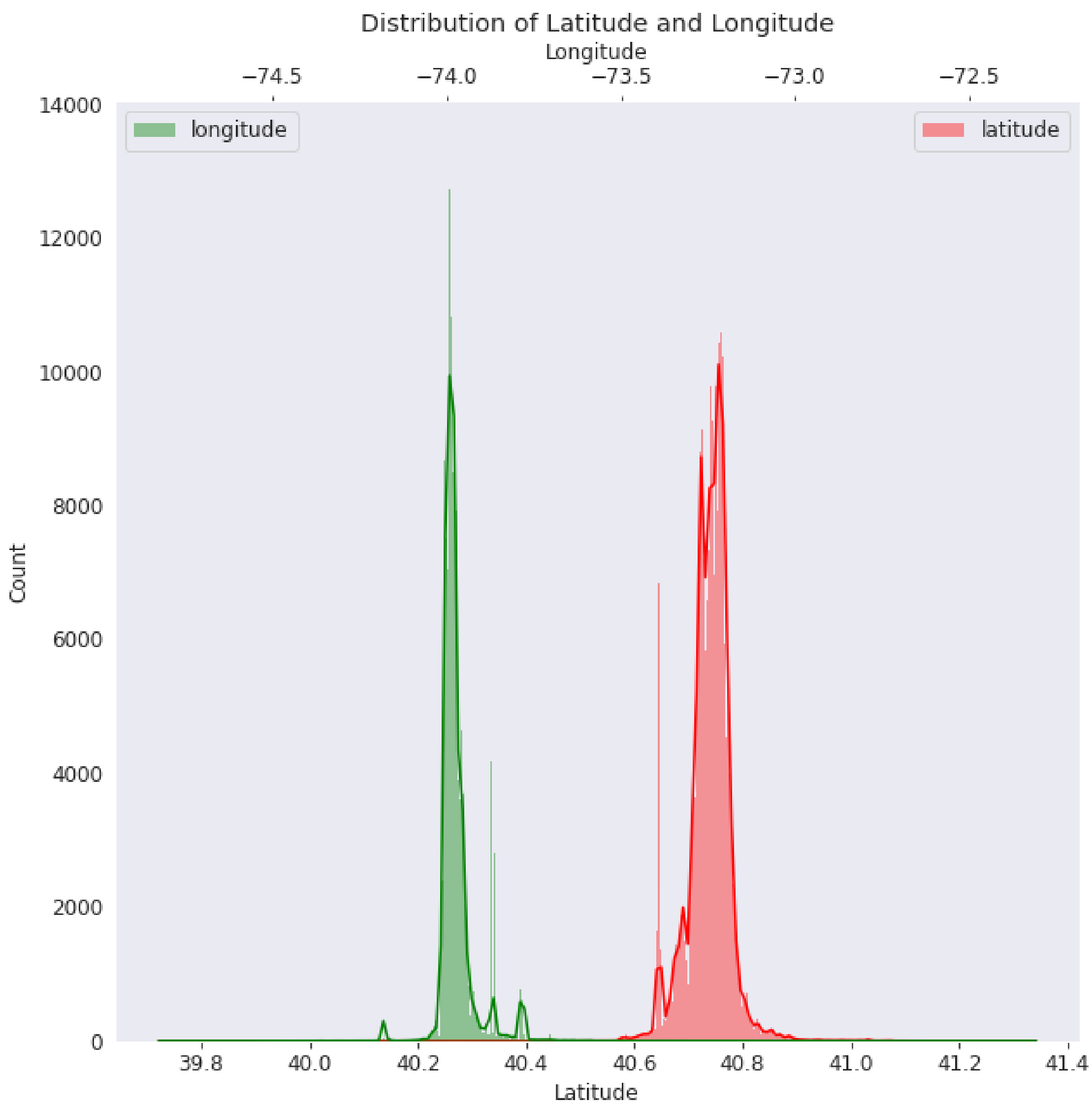
[24]:
```
#A mapper to map base number with its name
BaseMapper={'B02512' : 'Unter', 'B02598' : 'Hinter', 'B02617' : 'Weiter',␣
 ↪'B02682' : 'Schmecken','B02764' : 'Danach-NY'}

#Count plot of Base
plt.figure(figsize=(12,10))
sns.set_style("dark")
_=sns.countplot(x=uber_data['Base'].map(BaseMapper))
plt.ylabel('Total rides')
_=plt.title('CountPlot: Total uber rides vs Base - July 2014, NYC')
```



The above plot tells us that most uber rides originated from Weiter Base and least from Danach-NY

To know more about the distribution of latitudes and longitudes, let's plot their histograms along with KDEs

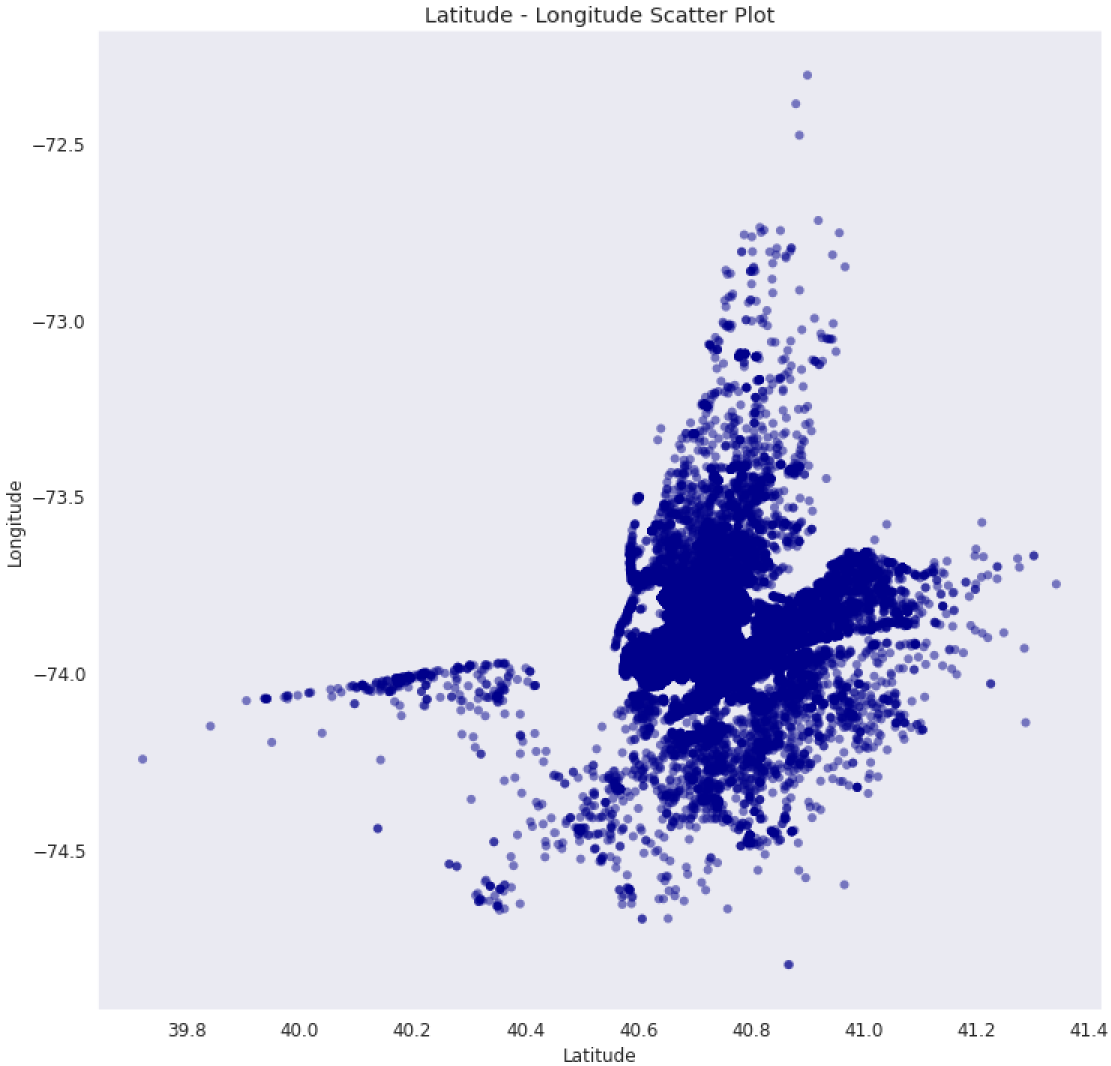
[25]:
```
plt.figure(figsize=(10,10))
sns.histplot(uber_data['Lat'], bins='auto',kde=True,color='r',alpha=0.4,label =␣
↪'latitude')
plt.legend(loc='upper right')
plt.xlabel('Latitude')
plt.twiny()
sns.histplot(uber_data['Lon'], bins='auto',kde=True,color='g',alpha=0.4,label =␣
↪'longitude')
_=plt.legend(loc='upper left')
_=plt.xlabel('Longitude')
_=plt.title('Distribution of Latitude and Longitude')
```



Most latitudes are around 40.25, and longitudes around 40.75. This is true as the dataset comprises information only around New York City. This also indicates that most rides happen around (lat,lon) = (40.25,40.75)

Let's display the latitude - longitude information in 2D:

[26]:
```
plt.figure(figsize=(12,12))
sns.scatterplot(x='Lat',y='Lon',data=uber_data,edgecolor='None',alpha=0.
 ↪5,color='darkblue')
plt.xlabel('Latitude')
plt.ylabel('Longitude')
_=plt.title('Latitude - Longitude Scatter Plot')
```



The dark blue area in the center shows the regions in New York City that had most number of uber rides in July 2014. The plot is better understood when a geographical map is placed underneath

Let's use geopy to calculate the distance between Metropolitan Museum and Emperical State Building

[27]:
```
#This is an example of using geopy
metro_art_coordinates = (40.7794,-73.9632)
empire_state_building_coordinates = (40.7484,-73.9857)

distance = geopy.distance.
 ↪distance(metro_art_coordinates,empire_state_building_coordinates)

print("Distance = ",distance)
```

```
Distance =  3.931943183851315 km
```

Using geopy on a larger dataset may be time consuming on slower PC's. Hence let's use the haversine method

[28]:
```
def haversine(coordinates1,coordinates2):

    lat1=coordinates1[0]
    lon1=coordinates1[1]
    lat2=coordinates2[0]
    lon2=coordinates2[1]

    #convert to radians and apply haverson formula
    lon1,lat1,lon2,lat2 = map(radians,[lon1,lat1,lon2,lat2])
    dlon = lon2 - lon1
    dlat = lat2 - lat1

    a = sin(dlat/2)**2 + cos(lat1)*cos(lat2)*sin(dlon/2)**2
    c = 2*asin(sqrt(a))
    r = 3956
    return c*r
print("Distance (mi) =␣
↪",haversine(metro_art_coordinates,empire_state_building_coordinates))
```

```
Distance (mi) =  2.442501323483997
```

Now, Let's try to predict which place they are more closer to, say MM or ESB. This can be done by individually calculating the distance between each uber ride coordinates with MM or ESB coordinates. If they are found to be in a particular threshold radius with MM, then we can predict that the ride is going to MM. Similarly for ESB.

[29]:
```
#calculating distance to MM and ESB for each point in the dataset
uber_data['Distance MM'] = uber_data[['Lat','Lon']].apply(lambda x:␣
↪haversine(metro_art_coordinates,tuple(x)),axis=1)
uber_data['Distance ESB'] = uber_data[['Lat','Lon']].apply(lambda x:␣
↪haversine(empire_state_building_coordinates,tuple(x)),axis=1)
```

[30]:
```
#printing the first 10 elements of the updated dataset
uber_data.head(10)
```

[30]:
```
             Date/Time      Lat      Lon    Base          BinnedHour      Day  \
0 2014-07-01 00:03:00  40.7586 -73.9706  B02512 2014-07-01 00:00:00  Tuesday
1 2014-07-01 00:05:00  40.7605 -73.9994  B02512 2014-07-01 00:00:00  Tuesday
2 2014-07-01 00:06:00  40.7320 -73.9999  B02512 2014-07-01 00:00:00  Tuesday
3 2014-07-01 00:09:00  40.7635 -73.9793  B02512 2014-07-01 00:00:00  Tuesday
4 2014-07-01 00:20:00  40.7204 -74.0047  B02512 2014-07-01 00:15:00  Tuesday
5 2014-07-01 00:35:00  40.7487 -73.9869  B02512 2014-07-01 00:30:00  Tuesday
6 2014-07-01 00:57:00  40.7444 -73.9961  B02512 2014-07-01 00:45:00  Tuesday
7 2014-07-01 00:58:00  40.7132 -73.9492  B02512 2014-07-01 00:45:00  Tuesday
```

```
8 2014-07-01 01:04:00  40.7590 -73.9730  B02512 2014-07-01 01:00:00  Tuesday
9 2014-07-01 01:08:00  40.7601 -73.9823  B02512 2014-07-01 01:00:00  Tuesday

         Date      Time  Distance MM  Distance ESB
0  2014-07-01  00:00:00     1.487358      1.058178
1  2014-07-01  00:00:00     2.299140      1.100642
2  2014-07-01  00:00:00     3.794105      1.354266
3  2014-07-01  00:00:00     1.383450      1.094999
4  2014-07-01  00:15:00     4.615925      2.173858
5  2014-07-01  00:30:00     2.455439      0.066098
6  2014-07-01  00:45:00     2.966517      0.610105
7  2014-07-01  00:45:00     4.629089      3.090933
8  2014-07-01  01:00:00     1.498848      0.988372
9  2014-07-01  01:00:00     1.665310      0.827171
```

[31]:

```
#Now, let's keep a threshold of 0.25 miles and calculate the number of points
 ↪that are closer to MM and ESB
#according to these thresholds

print((uber_data[['Distance MM','Distance ESB']]<0.25).sum())
```

```
Distance MM      2764
Distance ESB    15133
dtype: int64
```

The result above shows the number of rides predicted to MM and ESB

[32]:

```
distance_range = np.arange(0.1,5.1,0.1)
```

[33]:

```
distance_data = [(uber_data[['Distance MM','Distance ESB']] < dist).sum() for
 ↪dist in distance_range]
```

[34]:

```
distance_data
```

[34]:

```
[Distance MM      575
 Distance ESB    2387
 dtype: int64,
 Distance MM     1776
 Distance ESB    9661
 dtype: int64,
 Distance MM      4566
 Distance ESB    22166
 dtype: int64,
 Distance MM      8783
 Distance ESB    42427
 dtype: int64,
 Distance MM     13606
 Distance ESB    68011
```

```
dtype: int64,
Distance MM     20770
Distance ESB    92650
dtype: int64,
Distance MM      29408
Distance ESB    119621
dtype: int64,
Distance MM      38912
Distance ESB    147815
dtype: int64,
Distance MM      50497
Distance ESB    177759
dtype: int64,
Distance MM      63072
Distance ESB    206056
dtype: int64,
Distance MM      75474
Distance ESB    240003
dtype: int64,
Distance MM      89442
Distance ESB    277785
dtype: int64,
Distance MM     105692
Distance ESB    311312
dtype: int64,
Distance MM     123431
Distance ESB    335385
dtype: int64,
Distance MM     141656
Distance ESB    355731
dtype: int64,
Distance MM     157194
Distance ESB    375017
dtype: int64,
Distance MM     174148
Distance ESB    393510
dtype: int64,
Distance MM     190108
Distance ESB    412560
dtype: int64,
Distance MM     204501
Distance ESB    434040
dtype: int64,
Distance MM     219190
Distance ESB    453986
dtype: int64,
Distance MM     234681
```

```
Distance ESB    472681
dtype: int64,
Distance MM     250469
Distance ESB    489396
dtype: int64,
Distance MM     265164
Distance ESB    502460
dtype: int64,
Distance MM     276425
Distance ESB    518076
dtype: int64,
Distance MM     291165
Distance ESB    532569
dtype: int64,
Distance MM     306739
Distance ESB    544541
dtype: int64,
Distance MM     318762
Distance ESB    557718
dtype: int64,
Distance MM     329219
Distance ESB    571684
dtype: int64,
Distance MM     341297
Distance ESB    583354
dtype: int64,
Distance MM     354514
Distance ESB    592929
dtype: int64,
Distance MM     369129
Distance ESB    603990
dtype: int64,
Distance MM     381511
Distance ESB    615394
dtype: int64,
Distance MM     394652
Distance ESB    623231
dtype: int64,
Distance MM     409278
Distance ESB    629393
dtype: int64,
Distance MM     425672
Distance ESB    634332
dtype: int64,
Distance MM     441094
Distance ESB    638128
dtype: int64,
```

```
Distance MM     455258
Distance ESB    641331
dtype: int64,
Distance MM     468838
Distance ESB    644712
dtype: int64,
Distance MM     482153
Distance ESB    648893
dtype: int64,
Distance MM     496619
Distance ESB    652852
dtype: int64,
Distance MM     512662
Distance ESB    656735
dtype: int64,
Distance MM     529702
Distance ESB    661066
dtype: int64,
Distance MM     546998
Distance ESB    665748
dtype: int64,
Distance MM     563198
Distance ESB    670373
dtype: int64,
Distance MM     575552
Distance ESB    674744
dtype: int64,
Distance MM     588588
Distance ESB    678522
dtype: int64,
Distance MM     597941
Distance ESB    682262
dtype: int64,
Distance MM     614256
Distance ESB    685487
dtype: int64,
Distance MM     621624
Distance ESB    688588
dtype: int64,
Distance MM     626604
Distance ESB    691884
dtype: int64]
```

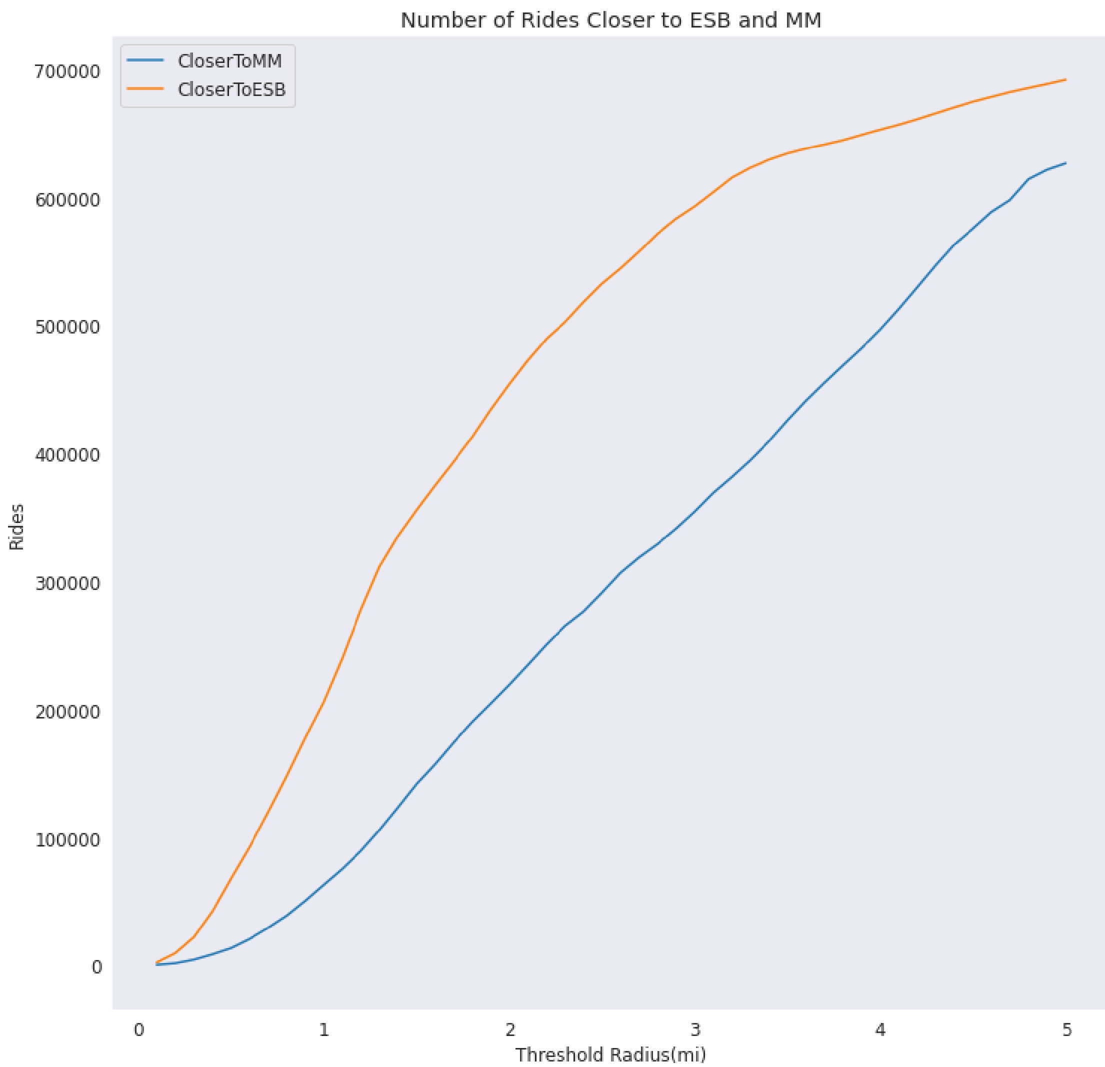
[35]:
```
#concatentate and transpose
distance_data = pd.concat(distance_data,axis=1)
distance_data = distance_data.T
```

[36]:
```
#Shifting index
distance_data.index = distance_range
```

[37]:
```
distance_data=distance_data.rename(columns={'Distance MM':
↪'CloserToMM','Distance ESB':'CloserToESB'})
```

[38]:
```
plt.figure(figsize=(12,12))
distance_data.plot(ax=plt.gca())
plt.title('Number of Rides Closer to ESB and MM')
plt.xlabel('Threshold Radius(mi)')
plt.ylabel('Rides')
```

[38]: Text(0, 0.5, 'Rides')



The number of riders to MM and ESB initially diverges, but comes closer as threshold

increases. Hence as radius increases, the rate of people going towards MM gets higher than that to ESB. In another way of thinking, as we expand the radius, most of the newly discovered rides are going to MM.

Now let us observe the heatmap plotted on geographical map (using folium)

[39]:
```
#inititilize the map around NYC and set the zoom level to 10
uber_map = folium.Map(location=metro_art_coordinates,zoom_start=10)

#lets mark MM and ESB on the map
folium.Marker(metro_art_coordinates,popup = "MM").add_to(uber_map)
folium.Marker(empire_state_building_coordinates,popup = "ESB").add_to(uber_map)

#convert to numpy array and plot it
Lat_Lon = uber_data[['Lat','Lon']].to_numpy()
folium.plugins.HeatMap(Lat_Lon,radius=10).add_to(uber_map)

#Displaying the map
uber_map
```

[39]: <folium.folium.Map at 0x7fc9056073a0>

Lets reduce the "Influence" of each point on the heatmap by using a weight of 0.5 (by default it is 1)

[40]:
```
uber_data['Weight']=0.5

#Take on 10000 points to plot (Just to speed up things)
Lat_Lon = uber_data[['Lat','Lon','Weight']].to_numpy()

#Plotting
uber_map = folium.Map(metro_art_coordinates,zoom_start=10)
folium.plugins.HeatMap(Lat_Lon,radius=15).add_to(uber_map)
uber_map
```

[40]: <folium.folium.Map at 0x7fc905526280>

The plot looks easy to visualize now. Boundaries and intensity distribution is clear

Let's now create a HeatMap that changes with time. This will help us to visualize the number of uber rides geographically at a given time.

We are plotting only the points that are in a radius of 0.25 miles from MM or ESB

[41]:
```
i = uber_data[['Distance MM','Distance ESB']] < 0.25

i.head(10)
```

[41]:

```
   Distance MM  Distance ESB
0        False         False
1        False         False
2        False         False
3        False         False
4        False         False
5        False          True
6        False         False
7        False         False
8        False         False
9        False         False
```

[42]:

```python
#Create a boolean mask to choose the rides that satisfy the 0.25 radius␣
↪threshold
i=i.any(axis=1)

i[i==True]
```

[42]:

```
5          True
13         True
17         True
31         True
104        True
           …
795863     True
795910     True
795925     True
795940     True
796101     True
Length: 17897, dtype: bool
```

[43]:

```python
#Create a copy of the data
map_data = uber_data[i].copy()

#use a smaller weight
map_data['Weight'] = 0.1

#Restricting data to that before 8th july for faster calculations
map_data = uber_data[uber_data["BinnedHour"] < datetime.datetime(2014,7,8)].
↪copy()

#Generate samples for each timestamp in "BinnedHour" (these are the points that␣
↪are plotted for each timestamp)
map_data = map_data.groupby("BinnedHour").apply(lambda x:␣
↪x[['Lat','Lon','Weight']].sample(int(len(x)/3)).to_numpy().tolist())
```

[44]:

```python
map_data
```

[44]:
```
BinnedHour
2014-07-01 00:00:00    [[40.6948, -74.178, 0.5], [40.7114, -74.0075, …
2014-07-01 00:15:00    [[40.76, -73.9805, 0.5], [40.6605, -73.9607, 0…
2014-07-01 00:30:00    [[40.7202, -73.9957, 0.5], [40.7645, -73.9783,…
2014-07-01 00:45:00    [[40.7428, -73.9966, 0.5], [40.7159, -73.9953,…
2014-07-01 01:00:00    [[40.755, -73.9843, 0.5], [40.7739, -73.9605, …
                                              …
2014-07-07 22:45:00    [[40.724000000000004, -73.9929, 0.5], [40.7559…
2014-07-07 23:00:00    [[40.7264, -73.9563, 0.5], [40.6899, -73.9551,…
2014-07-07 23:15:00    [[40.7293, -73.9576, 0.5], [40.7592, -73.9945,…
2014-07-07 23:30:00    [[40.6447, -73.7819, 0.5], [40.771, -73.9833, …
2014-07-07 23:45:00    [[40.6574, -73.9587, 0.5], [40.7633, -73.9847,…
Length: 672, dtype: object
```

[45]:
```
#The index to be passed on to heatmapwithtime needs to be a time series of the␣
↪following format
data_hour_index = [x.strftime("%m%d%Y, %H:%M:%S") for x in map_data.index]

#convert to list to feed it to heatmapwithtime
date_hour_data = map_data.tolist()

#initialize map
uber_map = folium.Map(location=metro_art_coordinates,zoom_start=10)
```

[46]:
```
#plotting
hm = folium.plugins.HeatMapWithTime(date_hour_data,index=date_hour_data)

#add heatmap to folium map(uber_map)
hm.add_to(uber_map)
uber_map
```

[46]:
```
<folium.folium.Map at 0x7fc8fead6520>
```

Click the play button to visualize the timeseries

[47]:
```
uber_data
```

[47]:
```
                  Date/Time      Lat      Lon    Base          BinnedHour  \
0       2014-07-01 00:03:00  40.7586 -73.9706  B02512 2014-07-01 00:00:00
1       2014-07-01 00:05:00  40.7605 -73.9994  B02512 2014-07-01 00:00:00
2       2014-07-01 00:06:00  40.7320 -73.9999  B02512 2014-07-01 00:00:00
3       2014-07-01 00:09:00  40.7635 -73.9793  B02512 2014-07-01 00:00:00
4       2014-07-01 00:20:00  40.7204 -74.0047  B02512 2014-07-01 00:15:00
…                       …        …        …       …                   …
796116  2014-07-31 23:22:00  40.7285 -73.9846  B02764 2014-07-31 23:15:00
796117  2014-07-31 23:23:00  40.7615 -73.9868  B02764 2014-07-31 23:15:00
796118  2014-07-31 23:29:00  40.6770 -73.9515  B02764 2014-07-31 23:15:00
```

```
796119 2014-07-31 23:30:00  40.7225 -74.0038  B02764 2014-07-31 23:30:00
796120 2014-07-31 23:58:00  40.7199 -73.9884  B02764 2014-07-31 23:45:00

             Day        Date      Time  Distance MM  Distance ESB  Weight
0        Tuesday  2014-07-01  00:00:00     1.487358      1.058178     0.5
1        Tuesday  2014-07-01  00:00:00     2.299140      1.100642     0.5
2        Tuesday  2014-07-01  00:00:00     3.794105      1.354266     0.5
3        Tuesday  2014-07-01  00:00:00     1.383450      1.094999     0.5
4        Tuesday  2014-07-01  00:15:00     4.615925      2.173858     0.5
…            …           …        …           …            …       …
796116  Thursday  2014-07-31  23:15:00     3.688336      1.375205     0.5
796117  Thursday  2014-07-31  23:15:00     1.746524      0.906320     0.5
796118  Thursday  2014-07-31  23:15:00     7.096685      5.244699     0.5
796119  Thursday  2014-07-31  23:30:00     4.465889      2.023519     0.5
796120  Thursday  2014-07-31  23:45:00     4.314474      1.972853     0.5

[796121 rows x 11 columns]
```

[48]:
```
weekends = weekly_data[['Saturday','Sunday']]
```

[49]:
```
weekdays = weekly_data.drop(['Saturday','Sunday'],axis=1)
```

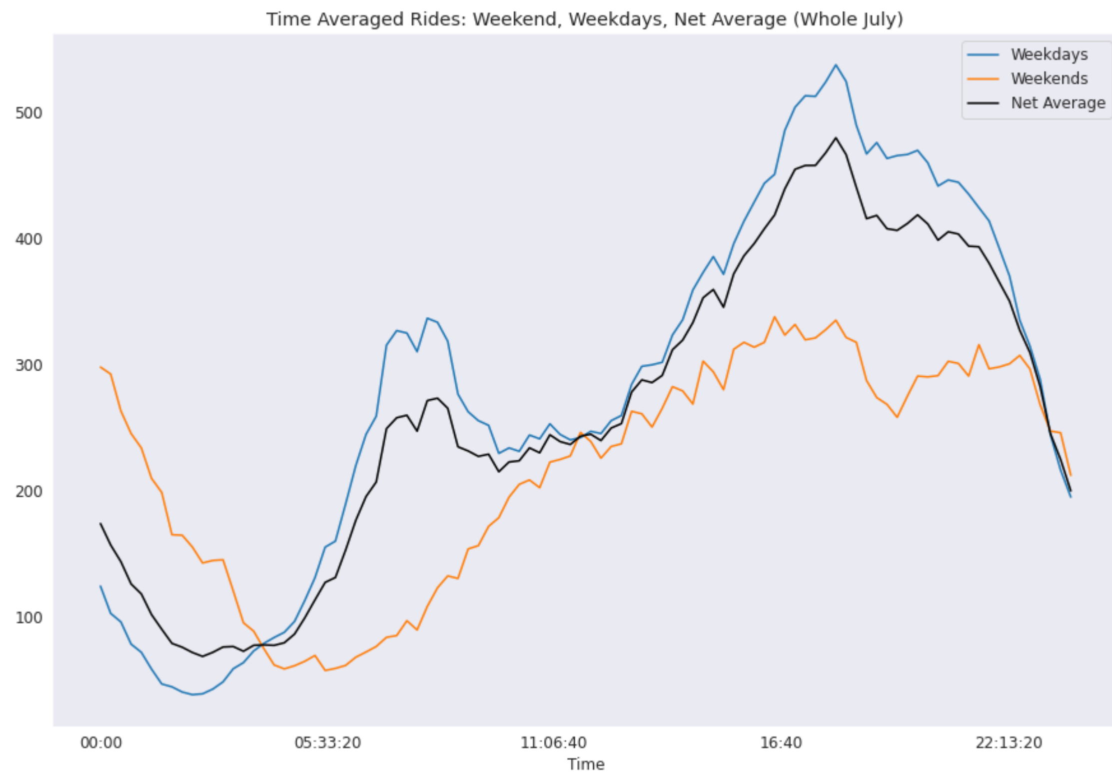
[50]:
```
weekends = weekends.mean(axis=1)
weekdays = weekdays.mean(axis=1)
```

[51]:
```
weekdays_weekends = pd.concat([weekdays,weekends],axis=1)
weekdays_weekends.columns = ['Weekdays','Weekends']
```

[52]:
```
weekdays_weekends
```

[52]:
```
          Weekdays  Weekends
Time
00:00:00    124.79   298.250
00:15:00    103.20   292.750
00:30:00     96.43   263.500
00:45:00     78.86   245.625
01:00:00     72.39   234.250
…              …        …
22:45:00    314.85   296.750
23:00:00    288.53   268.375
23:15:00    244.82   247.625
23:30:00    216.70   246.250
23:45:00    195.30   212.500

[96 rows x 2 columns]
```

[53]:
```
plt.figure(figsize=(15,10))
weekdays_weekends.plot(ax=plt.gca())
weekly_data.T.mean().plot(ax=plt.gca(),c = 'black',label='Net Average')
_=plt.title('Time Averaged Rides: Weekend, Weekdays, Net Average (Whole July)')
_=plt.legend()
```



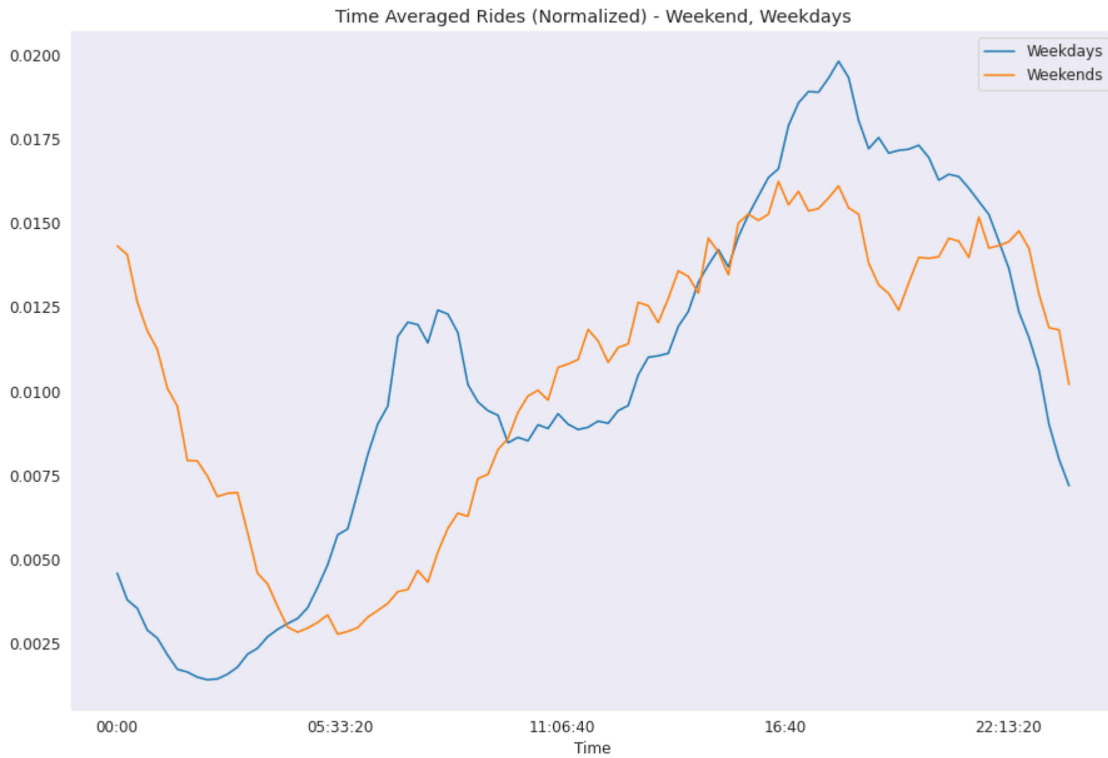
The Net average plot is more similar to the weekdays average because there are more weekdays than weekends.

In early morning, weekends have more rides. This makes sense as people often go out at night during the weekends.

The number of rides around 8 AM is less on weekends, but more on weekdays as it is usually the time when people goto work. Also, in the weekends, there is a surge in the number of evening rides as people return from work.

Let us normalize the weekday and weekends data with their own respective sums. This will give us an insight into the proportional data and help us answer questions like - "What percentage of rides happened around 12AM on weekends or weekdays"?

[54]:
```
plt.figure(figsize=(15,10))
(weekdays_weekends/weekdays_weekends.sum()).plot(ax=plt.gca())
_=plt.title('Time Averaged Rides (Normalized) - Weekend, Weekdays')
```



Nearly 1.5% of the total rides on weekends happen at midnight but only 0.5% of the total rides happen on weekdays! Also, nearly 2% of the total rides on weekdays happen around 5:30PM!

So far, we have made our observations by eye. Let us do a statistical T test to compare the time-averaged rides on weekdays and weekends

[55]:
```
#Grouping by date and time and creating a dataset that gives the total rides␣
↪every 15 mins
for_ttest = uber_data.groupby(['Date','Time']).count()['Day'].
↪reset_index(level=1)
```

[56]:
```
#Total rides on each day in july
uber_data.groupby(['Date']).count()['Day']
```

[56]:
```
Date
2014-07-01    21228
2014-07-02    26480
2014-07-03    21597
2014-07-04    14148
2014-07-05    10890
2014-07-06    11443
2014-07-07    18280
```

```
2014-07-08    25763
2014-07-09    27817
2014-07-10    30541
2014-07-11    28752
2014-07-12    25936
2014-07-13    21082
2014-07-14    27350
2014-07-15    33845
2014-07-16    28607
2014-07-17    30710
2014-07-18    29860
2014-07-19    25726
2014-07-20    21212
2014-07-21    23578
2014-07-22    29029
2014-07-23    34073
2014-07-24    32050
2014-07-25    29975
2014-07-26    27708
2014-07-27    22590
2014-07-28    23981
2014-07-29    27589
2014-07-30    30740
2014-07-31    33541
Name: Day, dtype: int64
```

[57]:

```
#Normalizing the dataset by dividing rides in each time slot on a day by total␣
↪number of rides on that day
for_ttest = pd.concat([for_ttest['Day']/uber_data.groupby(['Date']).
↪count()['Day'],for_ttest['Time']],axis=1)

#renaming
for_ttest=for_ttest.rename(columns={'Day':'NormalizedRides'})

for_ttest=pd.concat([for_ttest,uber_data.groupby(['Date','Time','Day']).count().
↪dropna().reset_index()[['Date','Day']].set_index('Date')],axis=1)

for_ttest
```

[57]:

Date	NormalizedRides	Time	Day
2014-07-01	0.003015	00:00:00	Tuesday
2014-07-01	0.002544	00:15:00	Tuesday
2014-07-01	0.002402	00:30:00	Tuesday
2014-07-01	0.002214	00:45:00	Tuesday
2014-07-01	0.001602	01:00:00	Tuesday
…	…	…	…

```
2014-07-31          0.012433  22:45:00  Thursday
2014-07-31          0.013148  23:00:00  Thursday
2014-07-31          0.009869  23:15:00  Thursday
2014-07-31          0.009511  23:30:00  Thursday
2014-07-31          0.008676  23:45:00  Thursday

[2976 rows x 3 columns]
```

[58]:
```
for_ttest
```

[58]:
```
            NormalizedRides      Time       Day
Date
2014-07-01          0.003015  00:00:00   Tuesday
2014-07-01          0.002544  00:15:00   Tuesday
2014-07-01          0.002402  00:30:00   Tuesday
2014-07-01          0.002214  00:45:00   Tuesday
2014-07-01          0.001602  01:00:00   Tuesday
…                        …         …         …
2014-07-31          0.012433  22:45:00  Thursday
2014-07-31          0.013148  23:00:00  Thursday
2014-07-31          0.009869  23:15:00  Thursday
2014-07-31          0.009511  23:30:00  Thursday
2014-07-31          0.008676  23:45:00  Thursday

[2976 rows x 3 columns]
```

The rides are first normalized by dividing the number of rides in each time slot by the total number of rides on that day

Then they are grouped by time and split to weekend and weekdays data and a T test is applied on them.

A Null hypothesis is assumed: The average ride counts are similar for each time slot on weekends and weekdays

[59]:
```
ttestvals = for_ttest.groupby('Time').apply(lambda x:␣
↪ttest_ind(x[x['Day']<'Saturday']['NormalizedRides'],x[x['Day']>='Saturday']['NormalizedRide
```

[60]:
```
ttestvals=pd.DataFrame(ttestvals.to_list(),index = ttestvals.index)
```

[61]:
```
ttestvals
```

[61]:
```
          statistic        pvalue
Time
00:00:00 -11.584950  2.120318e-12
00:15:00 -13.122326  1.001784e-13
00:30:00 -12.804495  1.843203e-13
00:45:00 -12.473992  3.515026e-13
```

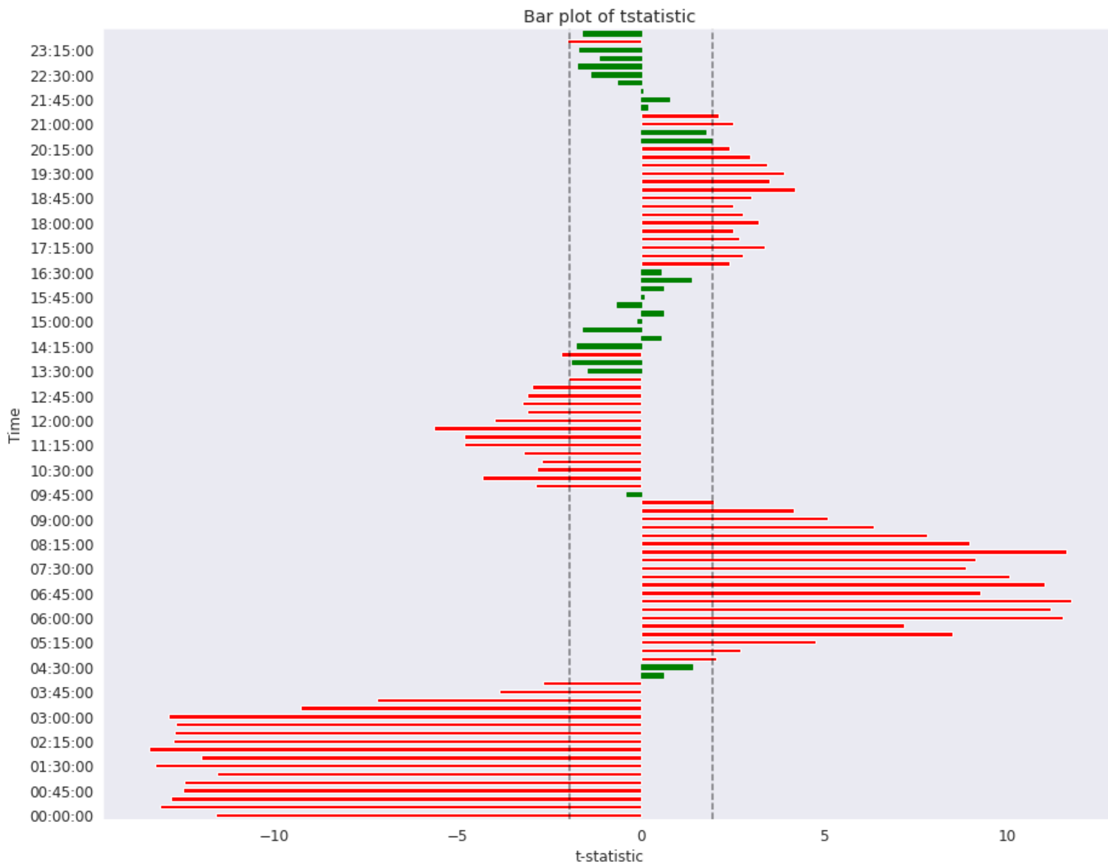
```
01:00:00 -12.447197  3.705828e-13
…              …           …
22:45:00  -1.711875  9.759578e-02
23:00:00  -1.103853  2.787324e-01
23:15:00  -1.678154  1.040634e-01
23:30:00  -2.013655  5.340771e-02
23:45:00  -1.574392  1.262446e-01

[96 rows x 2 columns]
```

The t-statistic value is -11.5 around midnight! This means that the assumption(hypothesis) does not hold at that time. The pvalue is very low, hence the null hypthesis is rejected around midnight

Let's plot and see the values for all timeslots

if we hold a p-value threshold of 5% (confidence level = 95%), corresponding t-statistic value is 1.96

[62]:
```python
#Let's plot the "statistic" column
plt.figure(figsize=(15,12))
ax=ttestvals['statistic'].plot(kind='barh',color='red',ax=plt.gca())
plt.locator_params(axis='y', nbins=40)
plt.locator_params(axis='x', nbins=10)
plt.xlabel('t-statistic')
plt.axvline(x=1.96,alpha=0.5,color='black',linestyle='--')
plt.axvline(x=-1.96,alpha=0.5,color='black',linestyle='--')

for rect in ax.patches:
    if(abs(rect.get_width())<1.96):
        rect.set_color('green')
_=plt.title('Bar plot of tstatistic')
```

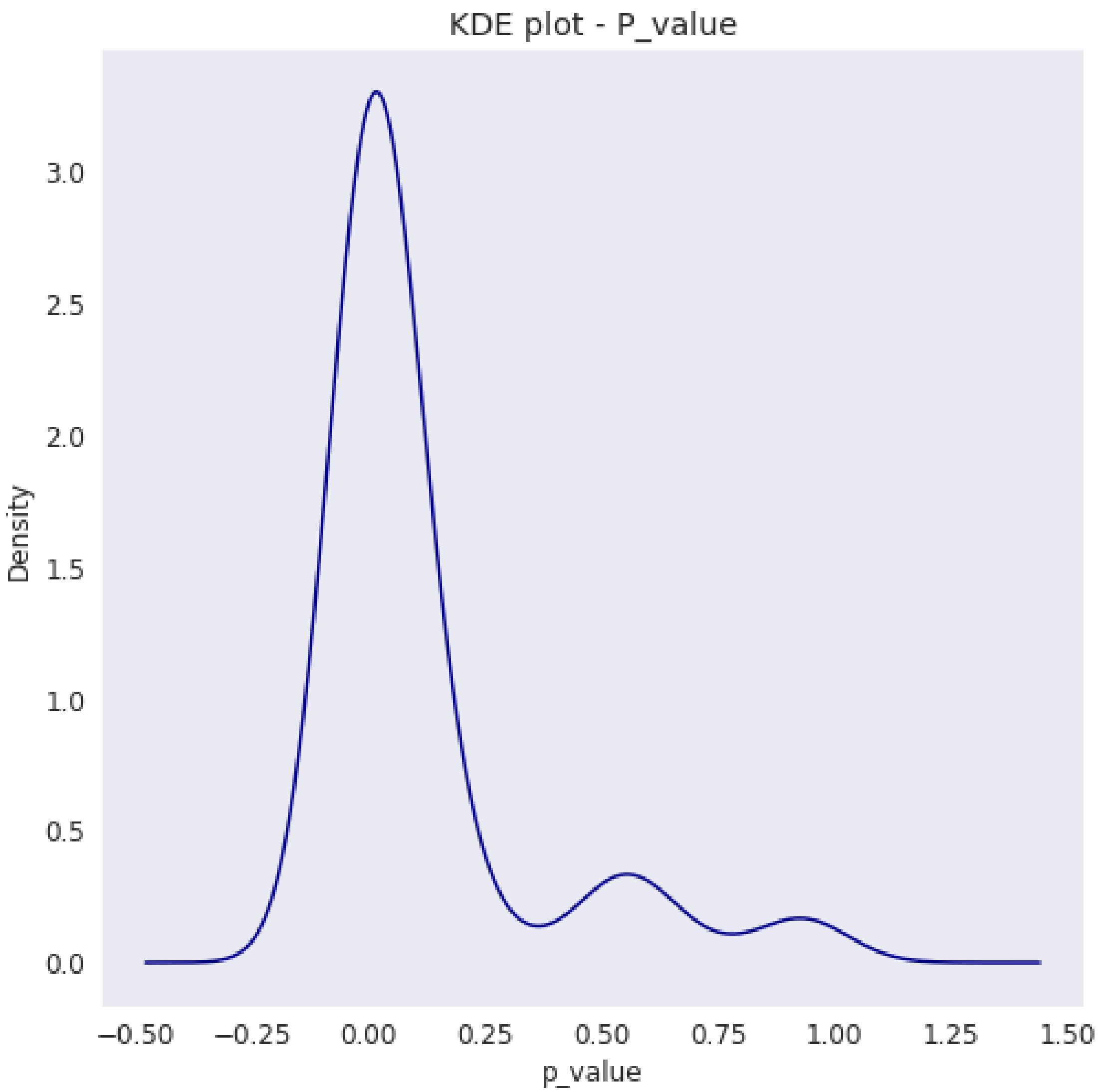


The time-average ride counts are assumed similar on weekdays and weekends if the width of the bar plot is less than 1.96. Such values are colored in green.

Note that their count is very low

Let's visualize a KDE plot of the pvalue to confirm this:

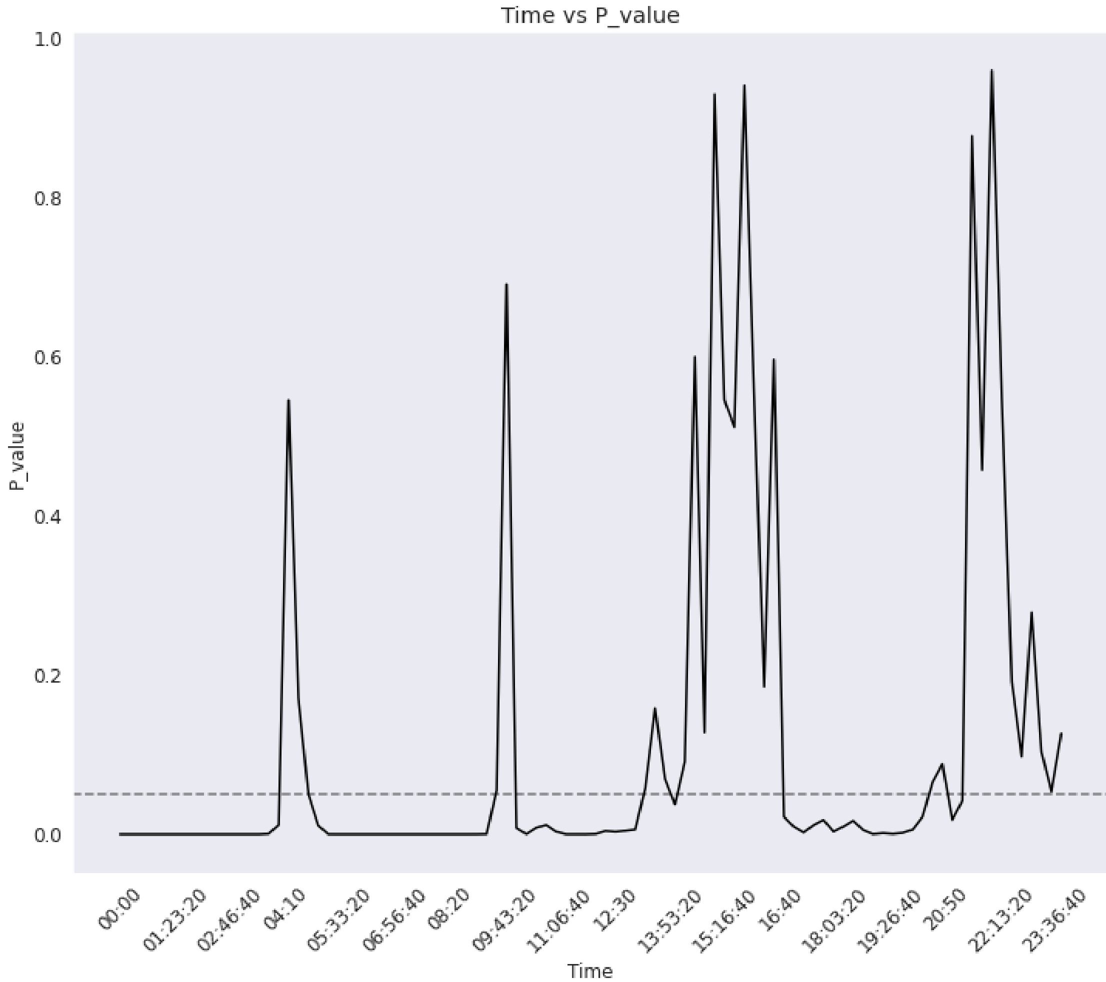
[63]:
```
#KDE plot
plt.figure(figsize=(8,8))
ttestvals['pvalue'].plot(kind='kde',color='darkblue',ax=plt.gca())
plt.title('KDE plot - P_value')
_=plt.xlabel('p_value')
```



Density peaks around p_value=0. Hence it confirms that the time-averaged rides vary greatly at most time slots on weekends and weekdays

P-value distribution:

[64]:
```
plt.figure(figsize=(12,10))
ax=ttestvals['pvalue'].plot(kind='line',color='black',ax=plt.gca())
plt.axhline(y=0.05,alpha=0.5,color='black',linestyle='--')
plt.locator_params(axis='x',nbins=20)
for item in plt.gca().get_xticklabels():
    item.set_rotation(45)

_=plt.title('Time vs P_value')
_=plt.ylabel('P_value')
```



The threshold is p = 0.05. The null hypothesis is accepted at p__values below 0.05

1.1 Data Processing for Training

[65]:
```
uber_data
```

[65]:
```
                   Date/Time      Lat      Lon    Base          BinnedHour  \
0       2014-07-01 00:03:00  40.7586 -73.9706  B02512 2014-07-01 00:00:00
1       2014-07-01 00:05:00  40.7605 -73.9994  B02512 2014-07-01 00:00:00
2       2014-07-01 00:06:00  40.7320 -73.9999  B02512 2014-07-01 00:00:00
3       2014-07-01 00:09:00  40.7635 -73.9793  B02512 2014-07-01 00:00:00
4       2014-07-01 00:20:00  40.7204 -74.0047  B02512 2014-07-01 00:15:00
…                       …        …        …       …                   …
796116 2014-07-31 23:22:00  40.7285 -73.9846  B02764 2014-07-31 23:15:00
796117 2014-07-31 23:23:00  40.7615 -73.9868  B02764 2014-07-31 23:15:00
796118 2014-07-31 23:29:00  40.6770 -73.9515  B02764 2014-07-31 23:15:00
```

```
796119 2014-07-31 23:30:00  40.7225 -74.0038  B02764 2014-07-31 23:30:00
796120 2014-07-31 23:58:00  40.7199 -73.9884  B02764 2014-07-31 23:45:00

             Day        Date      Time  Distance MM  Distance ESB  Weight
0        Tuesday  2014-07-01  00:00:00     1.487358      1.058178     0.5
1        Tuesday  2014-07-01  00:00:00     2.299140      1.100642     0.5
2        Tuesday  2014-07-01  00:00:00     3.794105      1.354266     0.5
3        Tuesday  2014-07-01  00:00:00     1.383450      1.094999     0.5
4        Tuesday  2014-07-01  00:15:00     4.615925      2.173858     0.5
…            …          …        …          …            …        …
796116  Thursday  2014-07-31  23:15:00     3.688336      1.375205     0.5
796117  Thursday  2014-07-31  23:15:00     1.746524      0.906320     0.5
796118  Thursday  2014-07-31  23:15:00     7.096685      5.244699     0.5
796119  Thursday  2014-07-31  23:30:00     4.465889      2.023519     0.5
796120  Thursday  2014-07-31  23:45:00     4.314474      1.972853     0.5

[796121 rows x 11 columns]
```

[66]:
```python
#create a copy
df = uber_data.copy()
```

[67]:
```python
#get numbers of each weekday
df['WeekDay']=df['Date/Time'].dt.weekday
```

[68]:
```python
#Convert datetime to float. egs: 1:15AM will be 1.25, 12:45 will be 12.75 etc
def func(x):
    hr = float(x.hour)
    minute = int(x.minute/15)
    return hr + minute/4
df['Time']=df['Date/Time'].apply(func)
```

[69]:
```python
#Get the day number, removing month and year
df['Day']=df['Date/Time'].dt.day
```

[70]:
```python
df
```

[70]:
```
                  Date/Time      Lat      Lon    Base          BinnedHour  Day  \
0       2014-07-01 00:03:00  40.7586 -73.9706  B02512 2014-07-01 00:00:00    1
1       2014-07-01 00:05:00  40.7605 -73.9994  B02512 2014-07-01 00:00:00    1
2       2014-07-01 00:06:00  40.7320 -73.9999  B02512 2014-07-01 00:00:00    1
3       2014-07-01 00:09:00  40.7635 -73.9793  B02512 2014-07-01 00:00:00    1
4       2014-07-01 00:20:00  40.7204 -74.0047  B02512 2014-07-01 00:15:00    1
…                       …        …       …      …                   …  …
796116 2014-07-31 23:22:00  40.7285 -73.9846  B02764 2014-07-31 23:15:00   31
796117 2014-07-31 23:23:00  40.7615 -73.9868  B02764 2014-07-31 23:15:00   31
796118 2014-07-31 23:29:00  40.6770 -73.9515  B02764 2014-07-31 23:15:00   31
796119 2014-07-31 23:30:00  40.7225 -74.0038  B02764 2014-07-31 23:30:00   31
```

```
796120 2014-07-31 23:58:00  40.7199 -73.9884  B02764 2014-07-31 23:45:00   31

              Date   Time  Distance MM  Distance ESB  Weight  WeekDay
0       2014-07-01   0.00     1.487358      1.058178     0.5        1
1       2014-07-01   0.00     2.299140      1.100642     0.5        1
2       2014-07-01   0.00     3.794105      1.354266     0.5        1
3       2014-07-01   0.00     1.383450      1.094999     0.5        1
4       2014-07-01   0.25     4.615925      2.173858     0.5        1
…             …     …           …            …      …       …
796116  2014-07-31  23.25     3.688336      1.375205     0.5        3
796117  2014-07-31  23.25     1.746524      0.906320     0.5        3
796118  2014-07-31  23.25     7.096685      5.244699     0.5        3
796119  2014-07-31  23.50     4.465889      2.023519     0.5        3
796120  2014-07-31  23.75     4.314474      1.972853     0.5        3

[796121 rows x 12 columns]
```

We are trying to predict the number of rides active in NYC on a given Day, Time and Base

[71]:
```
#Remove unwanted columns that were created for visualization
df = df.drop(['Date/Time','BinnedHour','Date','Distance MM','Distance␣
↪ESB','Lat','Lon'],axis=1)
```

[72]:
```
#create a redundant columns for easy counting of tolal rides
df['DropMe']=1
```

[73]:
```
#count the number of rides for a given day, weekday number, time and base
df = df.groupby(['Day','WeekDay','Time','Base']).count()['DropMe'].
↪reset_index().rename(columns={'DropMe':'Rides'})
```

[74]:
```
df
```

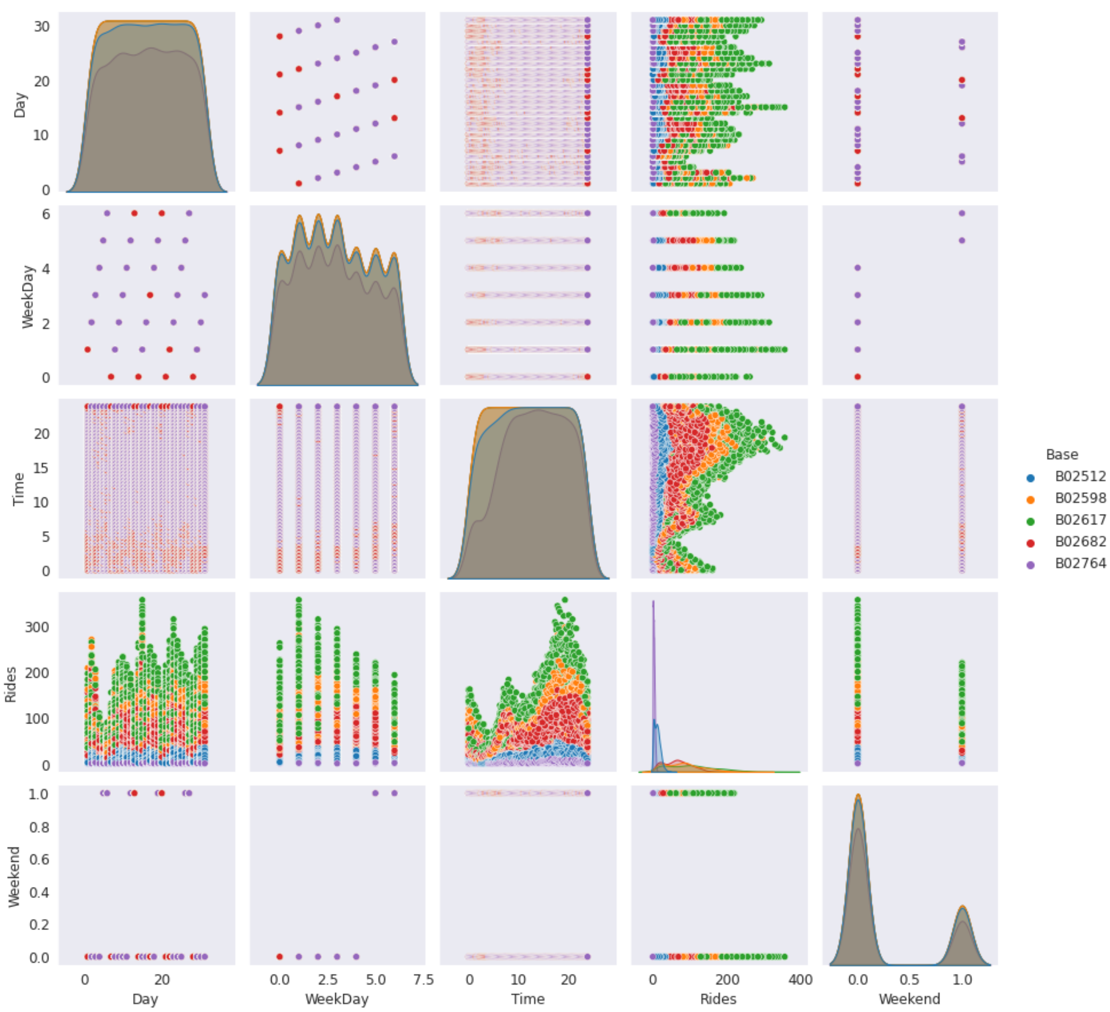
[74]:
```
       Day  WeekDay   Time    Base  Rides
0        1        1   0.00  B02512      4
1        1        1   0.00  B02598     15
2        1        1   0.00  B02617     23
3        1        1   0.00  B02682     22
4        1        1   0.25  B02512      1
…      …        …      …      …
14193   31        3  23.75  B02512      7
14194   31        3  23.75  B02598     83
14195   31        3  23.75  B02617    130
14196   31        3  23.75  B02682     70
14197   31        3  23.75  B02764      1

[14198 rows x 5 columns]
```

[75]:
```
#Weekends are given special emphasis, as their trends were very different from that on weekdays.
#so we devote a special columns indicating whether the day is weekday or not
df['Weekend']=df.apply(lambda x: 1 if(x['WeekDay']>4) else 0,axis=1)
```

Let's visualize a pairplot

[76]:
```
sns.pairplot(df,hue='Base')
```

[76]: <seaborn.axisgrid.PairGrid at 0x7fc8f3913ac0>

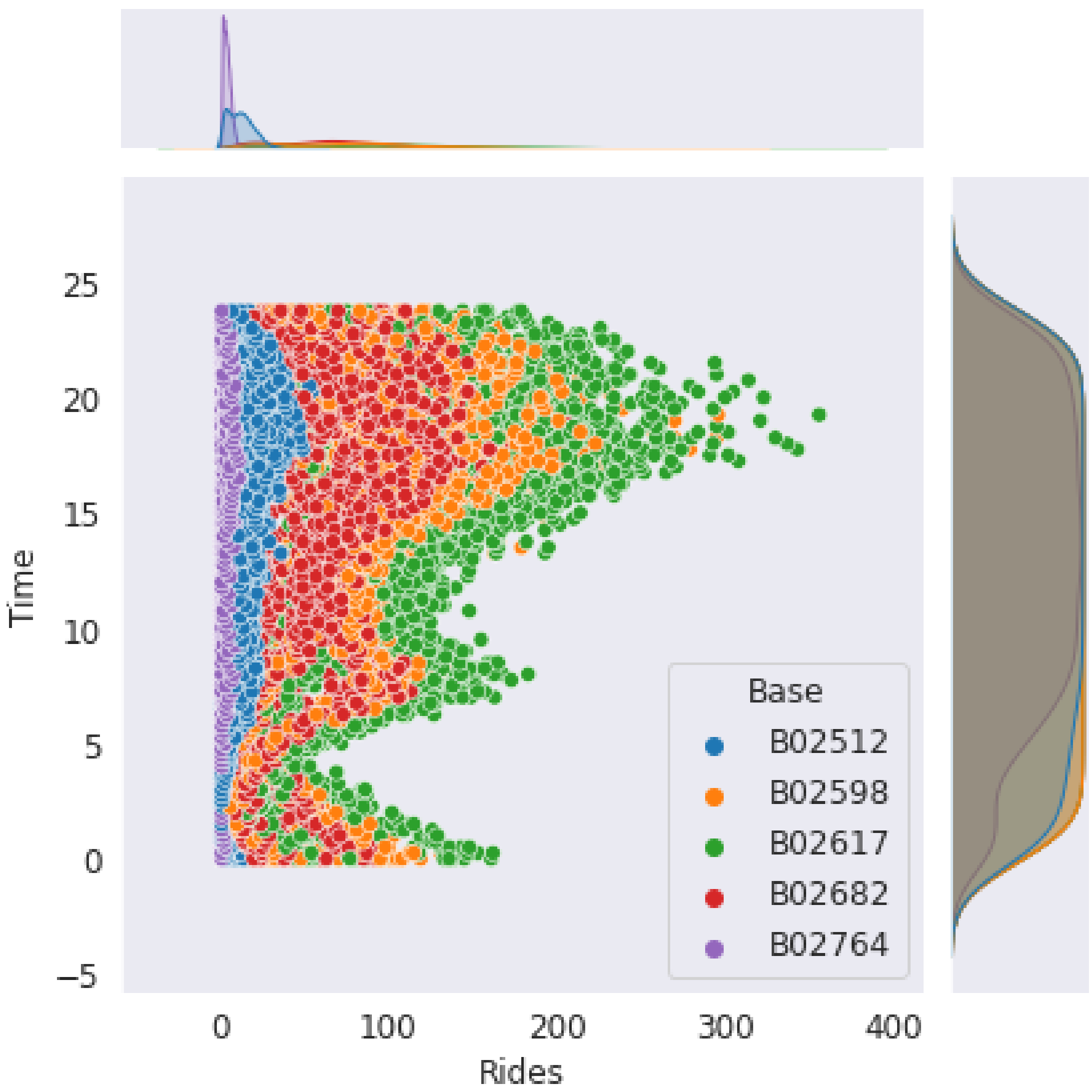


Notice the clusters in data! Especially time-rides, day-rides.

Let's create a jointplot of Rides vs Time

[77]:
```
plt.figure()
_=sns.jointplot(x='Rides',y='Time',data = df,hue='Base')
```

<Figure size 432x288 with 0 Axes>



[78]:
```python
#Split the categorical variable base into dummies
df=pd.get_dummies(data=df,columns=['Base'])
```

[79]:
```python
#Final Dataframe
df
```

[79]:

	Day	WeekDay	Time	Rides	Weekend	Base_B02512	Base_B02598	\
0	1	1	0.00	4	0	1	0	
1	1	1	0.00	15	0	0	1	
2	1	1	0.00	23	0	0	0	
3	1	1	0.00	22	0	0	0	

```
4         1         1   0.25       1         0            1            0
…     …         …     …      …      …          …          …
14193    31         3  23.75       7         0            1            0
14194    31         3  23.75      83         0            0            1
14195    31         3  23.75     130         0            0            0
14196    31         3  23.75      70         0            0            0
14197    31         3  23.75       1         0            0            0

       Base_B02617  Base_B02682  Base_B02764
0                0            0            0
1                0            0            0
2                1            0            0
3                0            1            0
4                0            0            0
…             …           …          …
14193            0            0            0
14194            0            0            0
14195            1            0            0
14196            0            1            0
14197            0            0            1

[14198 rows x 10 columns]
```

1.2 Training the model (Regression)

[98]:
```
#Split training and test data
X = df.drop('Rides',axis=1)
y = df['Rides']
X_train,X_test,y_train,y_test=train_test_split(X,y)
```

[99]:
```
#let's use a sequential model
model = Sequential()

#earlystopping criterion
early = EarlyStopping(monitor='val_loss',patience=5)
```

[100]:
```
#lets add the layers! Used 3 layers as the data too complex with high␣
 ↪oscillations
model.add(Dense(10,activation='relu'))
model.add(Dense(8,activation='relu'))
model.add(Dense(6,activation='relu'))
model.add(Dense(1,activation='linear'))

#Compile the model
model.compile(optimizer='adam',loss='mse')
```

[101]:
```
#Train the model
model.fit(x=X_train.to_numpy(),y=y_train.to_numpy(),epochs =
 ↪500,validation_data=(X_test.to_numpy(),y_test.to_numpy()),callbacks=[early])
```

```
Epoch 1/500
333/333 [==============================] - 1s 2ms/step - loss: 3624.4573 -
val_loss: 2561.8162
Epoch 2/500
333/333 [==============================] - 1s 2ms/step - loss: 2153.2153 -
val_loss: 1687.2501
Epoch 3/500
333/333 [==============================] - 0s 1ms/step - loss: 1245.4591 -
val_loss: 974.4487
Epoch 4/500
333/333 [==============================] - 0s 1ms/step - loss: 897.1102 -
val_loss: 850.6807
Epoch 5/500
333/333 [==============================] - 1s 2ms/step - loss: 824.8204 -
val_loss: 827.2253
Epoch 6/500
333/333 [==============================] - 1s 2ms/step - loss: 794.8353 -
val_loss: 770.6412
Epoch 7/500
333/333 [==============================] - 1s 2ms/step - loss: 771.1216 -
val_loss: 745.9926
Epoch 8/500
333/333 [==============================] - 1s 2ms/step - loss: 752.2471 -
val_loss: 727.3864
Epoch 9/500
333/333 [==============================] - 1s 2ms/step - loss: 736.2394 -
val_loss: 713.7685
Epoch 10/500
333/333 [==============================] - 1s 2ms/step - loss: 722.6155 -
val_loss: 706.8997
Epoch 11/500
333/333 [==============================] - 1s 2ms/step - loss: 712.4614 -
val_loss: 690.7827
Epoch 12/500
333/333 [==============================] - 1s 2ms/step - loss: 697.3582 -
val_loss: 680.5424
Epoch 13/500
333/333 [==============================] - 1s 2ms/step - loss: 688.2250 -
val_loss: 661.0859
Epoch 14/500
333/333 [==============================] - 1s 2ms/step - loss: 679.2216 -
val_loss: 680.9598
Epoch 15/500
```

```
333/333 [==============================] - 1s 2ms/step - loss: 671.0305 -
val_loss: 649.7668
Epoch 16/500
333/333 [==============================] - 1s 2ms/step - loss: 663.9362 -
val_loss: 669.1340
Epoch 17/500
333/333 [==============================] - 1s 2ms/step - loss: 651.6335 -
val_loss: 632.9362
Epoch 18/500
333/333 [==============================] - 1s 2ms/step - loss: 647.5186 -
val_loss: 621.3767
Epoch 19/500
333/333 [==============================] - 1s 2ms/step - loss: 642.6412 -
val_loss: 613.6165
Epoch 20/500
333/333 [==============================] - 1s 2ms/step - loss: 632.4163 -
val_loss: 607.3402
Epoch 21/500
333/333 [==============================] - 1s 2ms/step - loss: 626.4186 -
val_loss: 610.0090
Epoch 22/500
333/333 [==============================] - 1s 2ms/step - loss: 625.6596 -
val_loss: 598.9922
Epoch 23/500
333/333 [==============================] - 1s 2ms/step - loss: 618.3979 -
val_loss: 603.9066
Epoch 24/500
333/333 [==============================] - 1s 2ms/step - loss: 616.6130 -
val_loss: 587.7757
Epoch 25/500
333/333 [==============================] - 1s 2ms/step - loss: 611.3876 -
val_loss: 622.8671
Epoch 26/500
333/333 [==============================] - 1s 2ms/step - loss: 607.5728 -
val_loss: 579.5491
Epoch 27/500
333/333 [==============================] - 1s 2ms/step - loss: 606.7796 -
val_loss: 588.5816
Epoch 28/500
333/333 [==============================] - 1s 2ms/step - loss: 601.6901 -
val_loss: 577.5186
Epoch 29/500
333/333 [==============================] - 1s 2ms/step - loss: 597.7528 -
val_loss: 583.5020
Epoch 30/500
333/333 [==============================] - 1s 2ms/step - loss: 595.1072 -
val_loss: 568.6975
Epoch 31/500
```

```
333/333 [==============================] - 1s 2ms/step - loss: 599.2161 -
val_loss: 577.9228
Epoch 32/500
333/333 [==============================] - 1s 2ms/step - loss: 596.6141 -
val_loss: 605.6187
Epoch 33/500
333/333 [==============================] - 1s 2ms/step - loss: 590.7023 -
val_loss: 560.9277
Epoch 34/500
333/333 [==============================] - 1s 2ms/step - loss: 592.7302 -
val_loss: 563.8113
Epoch 35/500
333/333 [==============================] - 1s 2ms/step - loss: 591.5257 -
val_loss: 559.9681
Epoch 36/500
333/333 [==============================] - 1s 2ms/step - loss: 587.3145 -
val_loss: 556.4615
Epoch 37/500
333/333 [==============================] - 1s 2ms/step - loss: 584.9218 -
val_loss: 558.2294
Epoch 38/500
333/333 [==============================] - 1s 2ms/step - loss: 584.1163 -
val_loss: 609.4501
Epoch 39/500
333/333 [==============================] - 1s 2ms/step - loss: 584.3669 -
val_loss: 586.3705
Epoch 40/500
333/333 [==============================] - 1s 2ms/step - loss: 582.6738 -
val_loss: 552.4874
Epoch 41/500
333/333 [==============================] - 1s 2ms/step - loss: 577.4987 -
val_loss: 546.2685
Epoch 42/500
333/333 [==============================] - 1s 2ms/step - loss: 575.3602 -
val_loss: 545.1703
Epoch 43/500
333/333 [==============================] - 1s 2ms/step - loss: 576.2678 -
val_loss: 544.1259
Epoch 44/500
333/333 [==============================] - 1s 2ms/step - loss: 571.6109 -
val_loss: 544.5748
Epoch 45/500
333/333 [==============================] - 1s 2ms/step - loss: 571.6226 -
val_loss: 544.3657
Epoch 46/500
333/333 [==============================] - 1s 2ms/step - loss: 567.3814 -
val_loss: 552.7330
Epoch 47/500
```

```
333/333 [==============================] - 1s 2ms/step - loss: 562.2141 -
val_loss: 536.8662
Epoch 48/500
333/333 [==============================] - 1s 2ms/step - loss: 561.3638 -
val_loss: 538.9101
Epoch 49/500
333/333 [==============================] - 1s 2ms/step - loss: 560.2212 -
val_loss: 547.0568
Epoch 50/500
333/333 [==============================] - 1s 2ms/step - loss: 553.5842 -
val_loss: 531.6404
Epoch 51/500
333/333 [==============================] - 1s 2ms/step - loss: 548.3469 -
val_loss: 529.5342
Epoch 52/500
333/333 [==============================] - 1s 2ms/step - loss: 547.6683 -
val_loss: 523.5659
Epoch 53/500
333/333 [==============================] - 1s 2ms/step - loss: 547.3170 -
val_loss: 534.1754
Epoch 54/500
333/333 [==============================] - 0s 1ms/step - loss: 542.4702 -
val_loss: 511.9936
Epoch 55/500
333/333 [==============================] - 0s 1ms/step - loss: 541.6069 -
val_loss: 510.7058
Epoch 56/500
333/333 [==============================] - 0s 1ms/step - loss: 535.9667 -
val_loss: 511.0563
Epoch 57/500
333/333 [==============================] - 0s 1ms/step - loss: 534.1154 -
val_loss: 520.9407
Epoch 58/500
333/333 [==============================] - 0s 1ms/step - loss: 535.5923 -
val_loss: 499.0119
Epoch 59/500
333/333 [==============================] - 0s 1ms/step - loss: 528.9327 -
val_loss: 586.5483
Epoch 60/500
333/333 [==============================] - 1s 2ms/step - loss: 528.8610 -
val_loss: 499.5807
Epoch 61/500
333/333 [==============================] - 1s 2ms/step - loss: 520.0955 -
val_loss: 493.4511
Epoch 62/500
333/333 [==============================] - 1s 2ms/step - loss: 518.4415 -
val_loss: 489.3799
Epoch 63/500
```

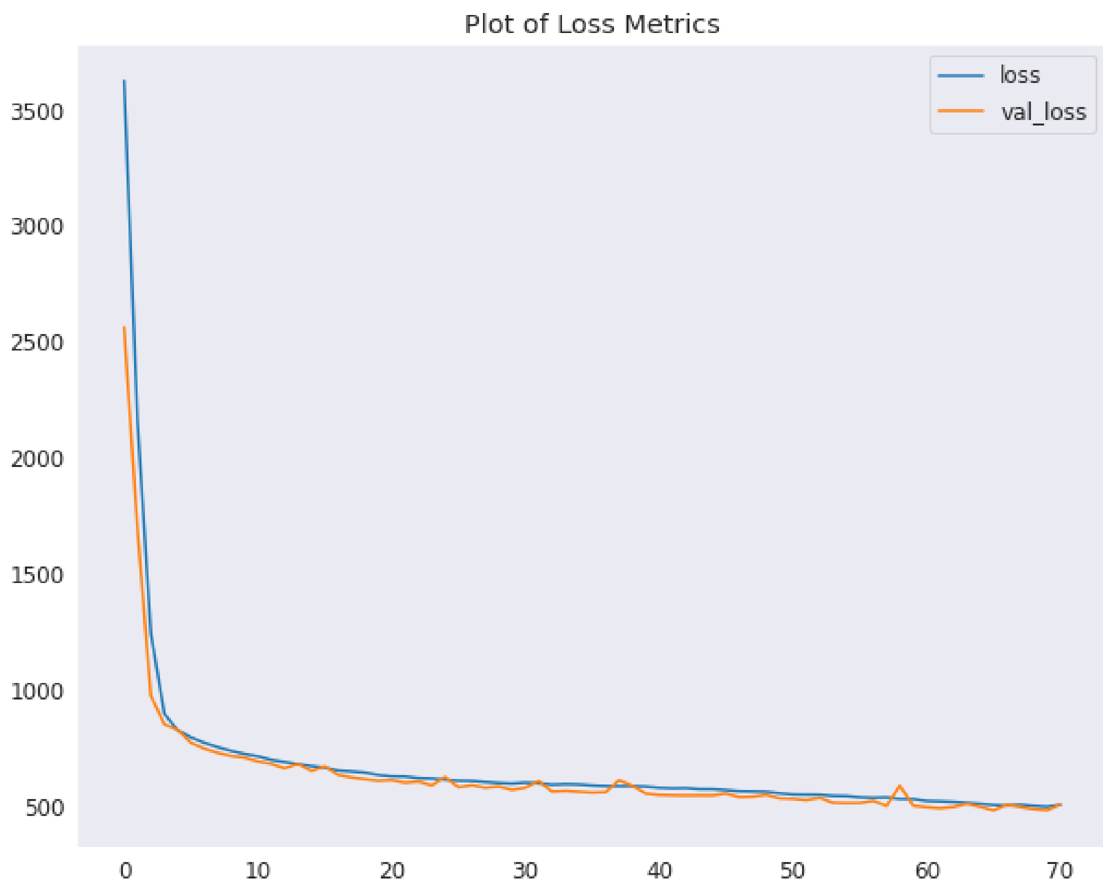
```
333/333 [==============================] - 1s 2ms/step - loss: 516.0013 -
val_loss: 494.1177
Epoch 64/500
333/333 [==============================] - 1s 2ms/step - loss: 511.6513 -
val_loss: 507.8582
Epoch 65/500
333/333 [==============================] - 1s 2ms/step - loss: 508.0159 -
val_loss: 495.7601
Epoch 66/500
333/333 [==============================] - 1s 2ms/step - loss: 503.0770 -
val_loss: 480.0312
Epoch 67/500
333/333 [==============================] - 1s 2ms/step - loss: 500.9879 -
val_loss: 504.1208
Epoch 68/500
333/333 [==============================] - 1s 2ms/step - loss: 504.4816 -
val_loss: 494.4320
Epoch 69/500
333/333 [==============================] - 1s 2ms/step - loss: 498.9584 -
val_loss: 485.5440
Epoch 70/500
333/333 [==============================] - 1s 2ms/step - loss: 495.3907 -
val_loss: 480.3152
Epoch 71/500
333/333 [==============================] - 1s 2ms/step - loss: 503.2463 -
val_loss: 504.9218
```

[101]:
```
<tensorflow.python.keras.callbacks.History at 0x7fc8e41004c0>
```

let's plot the losses curve

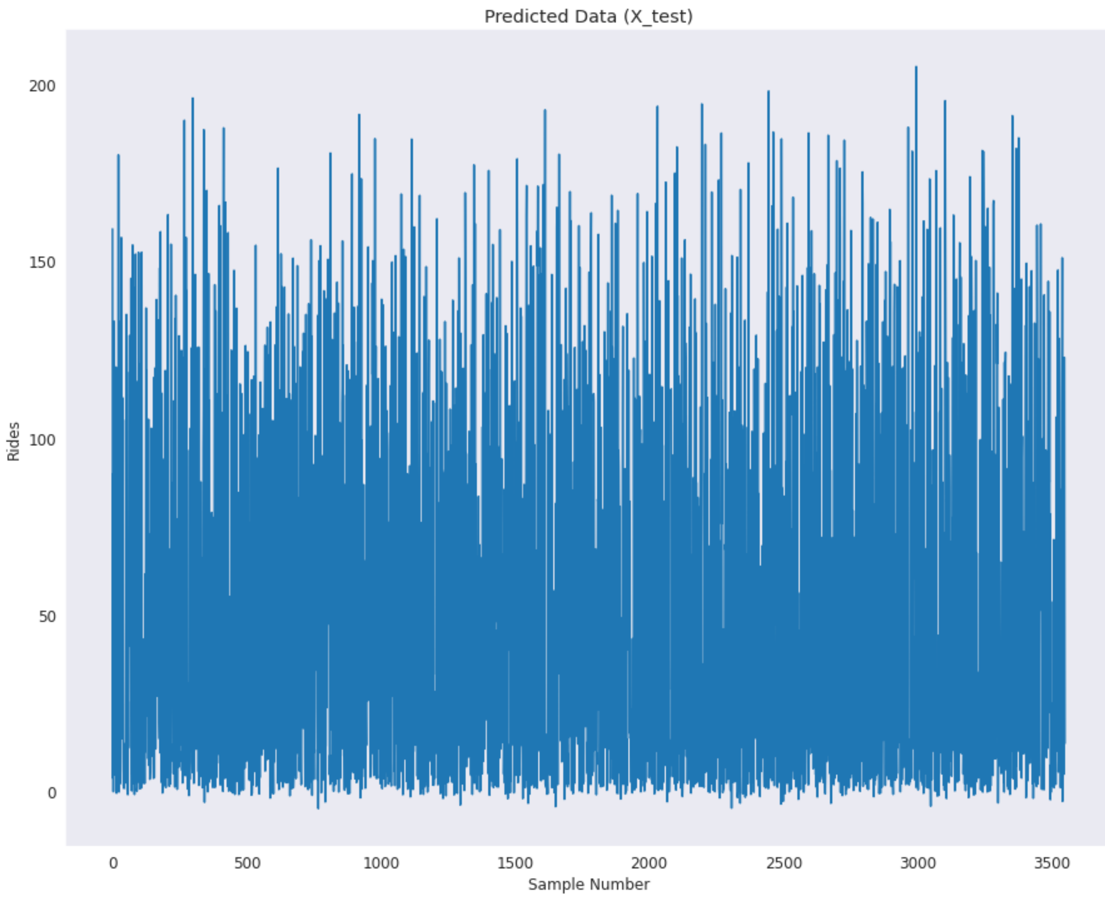
[102]:
```
plt.figure(figsize=(10,8))
pd.DataFrame(model.history.history).plot(ax=plt.gca())
plt.title('Plot of Loss Metrics')
```

[102]:
```
Text(0.5, 1.0, 'Plot of Loss Metrics')
```

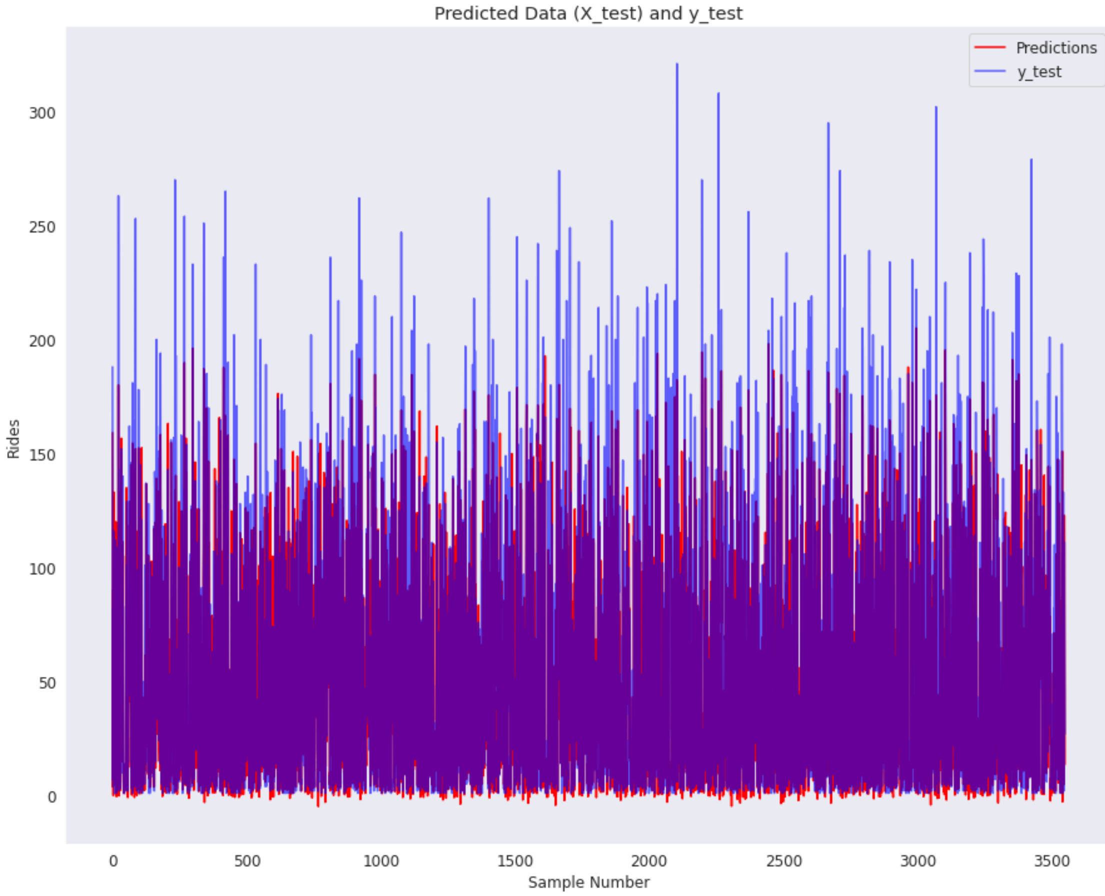


[103]:
```
#getting the predictions
predictions=model.predict(X_test).ravel()

#converting y_test series to numpy
y_test = y_test.to_numpy()
```

[104]:
```
#plotting predictions alone
plt.figure(figsize=(15,12))
plt.plot(predictions)
plt.title('Predicted Data (X_test)')
plt.ylabel('Rides')
_=plt.xlabel('Sample Number')
```

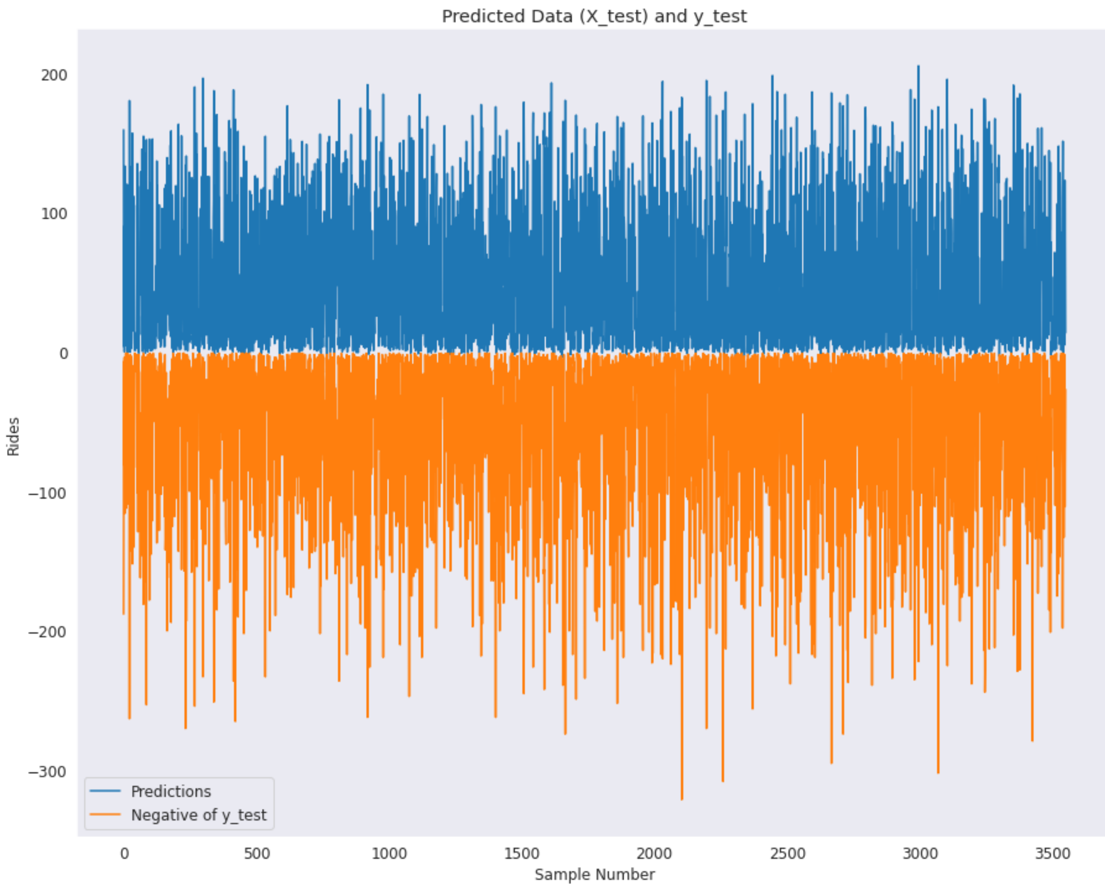


[105]:
```
y_test = abs(y_test)
plt.figure(figsize=(15,12))
plt.plot(predictions,label='Predictions',color='red')
plt.plot(y_test,label='y_test',color='blue',alpha=0.6)
plt.title('Predicted Data (X_test) and y_test')
plt.ylabel('Rides')
plt.xlabel('Sample Number')
_=plt.legend()
```



[106]:
```
plt.figure(figsize=(15,12))
ax = plt.gca()
y_test = -1*abs(y_test)
ax.plot(predictions,label='Predictions')
ax.plot(y_test,label='Negative of y_test')
plt.title('Predicted Data (X_test) and y_test')
plt.ylabel('Rides')
plt.xlabel('Sample Number')
plt.legend()
```

[106]: <matplotlib.legend.Legend at 0x7fc8e43b4160>



[107]:
```
print(f'The mean squared error is {mean_squared_error(abs(y_test),predictions):.
↪3f}')
print(f'The R2_Score is {r2_score(abs(y_test),predictions):.3f}')
```

```
The mean squared error is 504.922
The R2_Score is 0.843
```

[108]:
```
#Uncomment the line below to save the model
#model.save('uberLinReg.h5')
```

1.2.1 Thank you for your time!

[ ]:
```

```